Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

Лабораторная работа №5
Оптимизация вычислений в задаче матричного умножения. Оптимизация работы с памятью

При поддержке компании Intel

Козинов Е.А., Мееров И.Б., Сиднев А.В.
Кафедра математического обеспечения ЭВМ

Содержание

- Задача умножения плотных матриц.
- Применение intel mkl для умножения плотных матриц.
- Последовательная реализация алгоритма умножения матриц:
  - простейшая реализация алгоритма умножения матриц;
  - влияние порядка циклов на скорость вычислений.
- Векторизация вычислений.
- Реализация блочного алгоритма умножения матриц:
  - использование блока квадратной формы;
  - использование блоков прямоугольной формы.
- Параллельная реализация.
- Дополнительные задания и литература.

Цели

- Изучение методов применения библиотеки Intel MKL для решения задачи умножения матриц на хосте и на сопроцессоре.
- Реализация последовательной и параллельной версий блочного алгоритма для умножения матриц.
- Изучение вопроса о производительности базовой версии, рассмотрение способов сокращения времени вычислений с использованием векторизации, выбор подходящего порядка обращения к данным.

Задача матричного умножения…

- Пусть даны две матрицы A и B размерности *n* x *n*.
- По определению результатом умножения двух матриц является матрица C размерности *n* x *n*, где каждый элемент матрицы вычисляется по формуле:

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} * b_{k,j}, \text{где}$$

$$0 \leq i < n,$$

$$0 \leq j < n$$

- Сложность алгоритма $O(n^3)$

Задача матричного умножения

- Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью:
  - алгоритм Штрассена $O(n^{2,81})$;
  - алгоритм Копперсмита-Винограда $O(n^{2,376})$;
  - алгоритм Вирджинией Вильямс $O(n^{2,373})$.
- На практике алгориты применяют редко.
  - Как правило, в оценке сложности присутствует большая константа.
  - Алгоритмы обладают лучшей производительностью только при слишком больших размерах матриц, не помещающихся в оперативную память современных компьютеров.

Тестовая инфраструктура

Процессор	2 процессора на узел Xeon E5-2690 (2.9 GHz, 8 ядер)
Память	64 Gb
Сопроцессор	Intel Xeon Phi 7110X
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик, отладчик	Intel C/C++ Compiler 14

Применение Intel MKL для умножения плотных матриц

BLAS из Intel MKL…

- В C/C++ для умножения матриц с использованием Intel MKL необходимо использовать функцию BLAS третьего уровня – cblas_?gemm.
  - ? – для типа данных float заменяется на “s”.
- Функция позволяет вычислять выражения:

$$C = \alpha A*B + \beta C$$

где A, B и C – матрицы, а α и β – вещественные коэффициенты.

BLAS из Intel MKL…

- Пример вызова функции:

```
cblas_sgemm(CblasRowMajor, CblasNoTrans,
            CblasNoTrans, m, n, k, alpha,
            A, k, B, n, beta, C, n);
```

- CblasRowMajor – матрица хранится по строкам.
- CblasNoTrans – матрицы A и B не транспонированы.
- Переменные m, n и k задают размеры исходных матриц:
  - A: m строк и k столбцов.
  - B: k строк и n столбцов.
  - C: m строк и n столбцов.
- Переменные alpha и beta – коэффициенты, используемые в формуле.
- A, B – массивы, содержащие исходные матрицы.
- C – результат вычислений.

BLAS из Intel MKL…

- Разделим реализацию алгоритма умножения матриц на несколько файлов:
  - Файл содержащий функцию main().
  - Файл содержащий операции выделения, инициализации и удаления матриц.
  - Файлы содержащие проторит вычислительной функции умножения матиц и его реализацию.

BLAS из Intel MKL…

- Объявим определение вещественного типа данных и ряд прототипов функций (реализуйте самостоятельно):

```
// routine.h
#ifndef _ROUTINE_H
#define _ROUTINE_H

#ifndef ELEMENT_TYPE
#define ELEMENT_TYPE float
#endif

// Управление памятью
void allocMatrix(ELEMENT_TYPE ** mat, int n);
void freeMatrix (ELEMENT_TYPE ** mat);

// генерация элементов матрицы
void genMatrix(ELEMENT_TYPE * mat, int n);
// определение ошибки вычислений
ELEMENT_TYPE calculationError(ELEMENT_TYPE * A,
                ELEMENT_TYPE * B, ELEMENT_TYPE * C, int n);

#endif
```

BLAS из Intel MKL…

- Создадим заголовочный файл с прототипом функции умножения матриц:

```
// mult.h
#ifndef _MULT_
#define _MULT_

#include "routine.h"

//умножение матриц
void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n);

#endif
```

BLAS из Intel MKL…

- Реализуем функцию умножения с использованием MKL:

```
// MKL.cpp
#include "mult.h"
#include "mkl.h"

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n)
{
  ELEMENT_TYPE alpha, beta;
  alpha = 1.0;
  beta  = 0.0;
  cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                n, n, n, alpha, A, n, B, n, beta, C, n);
}
```

BLAS из Intel MKL

- Функцию main предлагается реализовать самостоятельно (файл с именем **`main.cpp`**)
- Содержание функции:
  - Подключение заголовочных файлов.
  - Объявление используемых переменных для хранение матриц их размеры и временные переменные.
  - Чтение размера матриц.
  - Выделение памяти для матриц A, B и C.
  - Генерация значений матриц A и B.
  - Запуск вычислительного эксперимента с замером времени (рекомендуется для замера времени использовать функцию **`omp_get_wtime()`**).
  - Вывод времени вычислений.
  - Освобождение памяти.

Компиляция и запуск…

- Для компиляции кода с использованием многопоточной версии Intel MKL, можно использовать следующую строку:

```
icpc -mkl=parallel -openmp ./MKL.cpp
./main.cpp ./routine.cpp –oMKL
```

- Если необходимо скомпилировать код, гарантируя использование последовательных версий алгоритмов из Intel MKL, строка компиляции незначительно меняется:

```
icpc -mkl=sequential -openmp ./MKL.cpp
./main.cpp ./routine.cpp –oMKL
```

- Запустить полученный код можно строкой следующего вида:

```
./MKL 3072
```

Компиляция и запуск

- Для компиляции программ, исполняемых на сопроцессоре Intel Xeon Phi, необходимо в строку компиляции добавить еще один ключ компилятора (-mmic):

```
icpc -mmic -mkl=parallel -openmp ./MKL.cpp
./main.cpp ./routine.cpp -oMKL
```

- Для запуска кода можно зайти на сопроцессор, используя ssh:

```
ssh mic0
./MKL 3072
```

Сравнение времени умножения матриц (Intel MKL) в разных конфигурациях (время в секундах)…

- Запустим код:

```
/home1/unnkoz/kozinov/MatrixMult/src $ ./MKL 3072
Intel Xeon Phi thread: 244

Size of matrix   : 3072
Calculation time : 0.240
Calculation error : 0.00000
/home1/unnkoz/kozinov/MatrixMult/src $
```

- Сравним время работы:

MKL sequential			MKL parallel	
N	Intel Xeon	Intel Xeon Phi	Intel Xeon	Intel Xeon Phi
1024	0,128	0,207	0,031	0,110
2048	0,337	1,363	0,055	0,140
3048	1,059	4,411	0,193	0,244

Сравнение времени умножения матриц (Intel MKL) в разных конфигурациях…

- Время работы на одном ядре CPU ожидаемо лучше, чем время работы на одном ядре Xeon Phi.
  - Как неоднократно говорилось в лекциях по архитектуре, ядра сопроцессора существенно слабее ядер CPU.
- В параллельных запусках ожидалось, что MKL на сопроцессоре покажет лучший результат, чем MKL на 16 ядрах CPU. **Этого не происходит.**
  - Так называемый «прогрев» (warm up) оказывает существенное влияние на время работы программы, использующей MKL, особенно на сопроцессоре.
  - Причин тому несколько:
    - накладные расходы на создание потоков,
    - настройка TLB-кеша,
    - особенности внутреннего устройства MKL (достоверно подтвердить или опровергнуть этот факт мы не можем) и др.

Сравнение времени умножения матриц (Intel MKL) в разных конфигурациях (время в секундах)

MKL sequential			MKL parallel		MKL parallel + warm up	
N	Intel Xeon	Intel Xeon Phi	Intel Xeon	Intel Xeon Phi	Intel Xeon	Intel Xeon Phi
1024	0,128	0,207	0,031	0,110	0,012	0,004
2048	0,337	1,363	0,055	0,140	0,047	0,019
3048	1,059	4,411	0,193	0,244	0,129	0,062
10000	35,198	130,983	4,05	2,06	2,765	1,835

- Добавлены результаты с прогревом и для N = 10000.
- Прогрев сокращает время работы как на CPU, так и на Xeon Phi, при этом сопроцессор **начинает обгонять 16 ядер CPU.**
- *В дальнейшем все эксперименты будут проводиться на сопроцессоре Intel Xeon Phi.*

Последовательная реализация алгоритма умножения матриц

Базовая реализация алгоритма умножения матриц

Базовая реализация алгоритма умножения матриц

- Реализуем алгоритм умножения согласно определению.

```
//single.cpp
#include "mult.h"

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n)
{
  ELEMENT_TYPE s;
  int i, j, k;
  for(j = 0; j < n; j++ )
    for(i = 0; i < n; i++ )
      C[j * n + i] = 0;

  for(i = 0 i < n; i++ )
    for(j = 0; j < n; j++ )
      for(k = 0; k < n; k++ )
        C[j * n + i] += A[j * n + k] * B[k * n + i];
}
```

Компиляция и запуск

- Укажем в строке компиляции файл с реализацией алгоритма:

```
icpc -mmic -mkl=parallel -openmp ./single.cpp
./main.cpp ./routine.cpp -osingle
```

- Скомпилируем и запустим приложение.

```
/home1/unnkoz/kozinov/MatrixMult/src $ ../bin/single 1024
Intel Xeon Phi thread: 244

Size of matrix   : 1024
Calculation time : 57.406
Calculation error : 0.00000
/home1/unnkoz/kozinov/MatrixMult/src $
```

Сравнение времени умножения матриц (Intel MKL) с наивной реализацией (время в секундах)…

N	Наивная реализация	**MKL sequential**	**MKL parallel**
1024	57,34	0,207	0,004

- Видим, что при N = 1024 время работы составляет около 57с. Время работы аналогичной (более сложной) функции в библиотеке MKL составляет около 0,1с.
- Для определенности сделаем запуск для N = 1025.

N	Наивная реализация	**MKL sequential**	**MKL parallel**
1024	**57,34**	0,207	0,004
1025	**33,09**	0,200	0,004

- *Мистика или объективная реальность?*

Сравнение времени умножения матриц (Intel MKL) с наивной реализацией

- Время работы наивной реализации сильно отличается от MKL
  - Разработчики высокопроизводительных библиотек хорошо знают свое дело. ☺
- **Существенное сокращение времени при увеличении размера на единицу – более чем странный факт.**
- При N = 1024 мы обращаемся в память с шагом таким, что нужные данные должны быть загружены в одну и ту же кеш-линию.
- В результат – крайне неэффективное использование кэш-памяти:
  - при следующей операции с данными мы вынуждены вытеснить кеш-линейку, которую недавно загружали и планировали использовать далее;
  - значительная часть других кеш-линеек не используется.
- При размере матрицы N = 1025 этого не происходит.
- *Составьте фрагмент карты обращений в память с учетом того, что кеш является 8-ассоциативным, размер кеш-линии составляет 64 байта, размер кеш-памяти составляет 32KB.*

Влияние порядка циклов на скорость вычислений

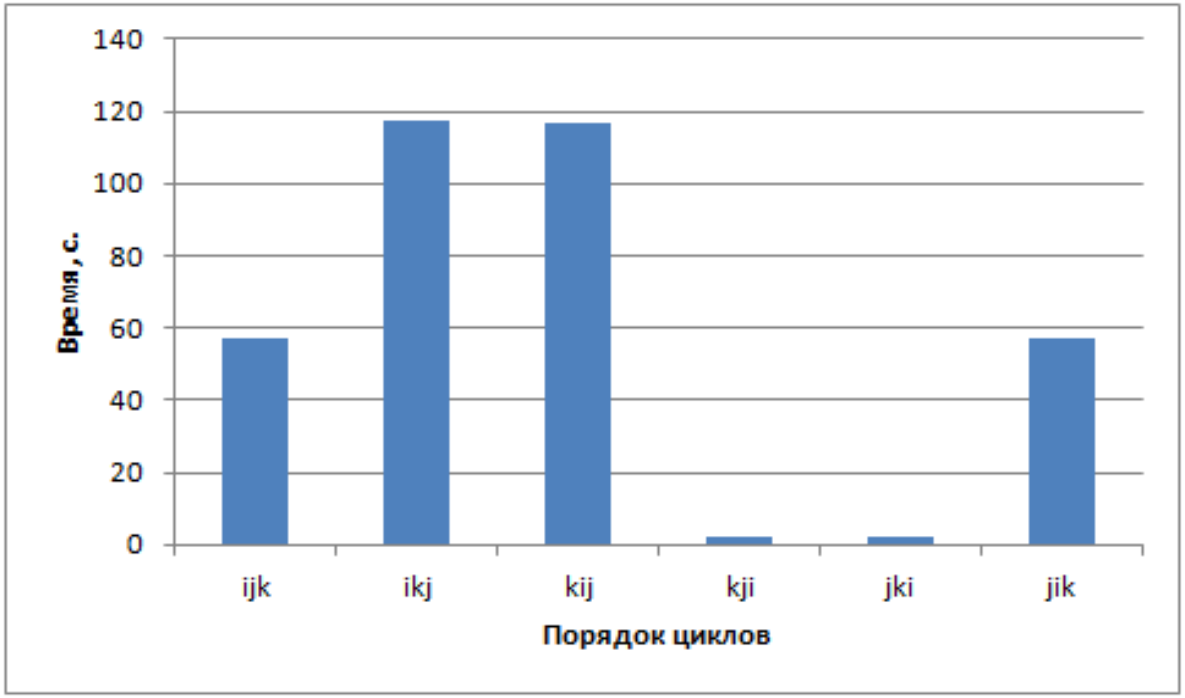
Влияние порядка циклов на скорость вычислений

- Посмотрев внимательно на карту обращений в память, составленную ранее, можно заметить, что мы обращаемся в память не в том порядке, в котором храним данные
- Попробуем изменить порядок циклов и исследуем вопрос о том, как это влияет на время работы программы.
- Всего существует шесть возможных перестановок.
  - *Разработку реализаций алгоритма при разных порядках циклов предлагается провести самостоятельно.*

Компиляция и запуск

- Сравнение времени вычислений при разном порядке циклов на Intel Xeon Phi (N = 1024, время в секундах)

Ijk	Ikj	Kij	kji	jki	jik	MKL seq.
57,34	117,5	116,81	2,149	12,084	56,904	0,207

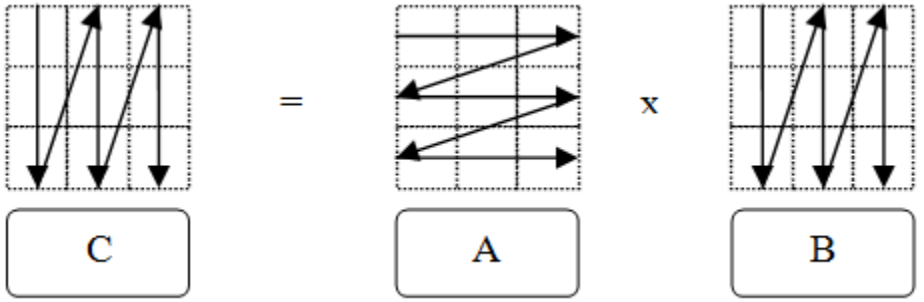


Время выполнения алгоритма в зависимости от порядка циклов …

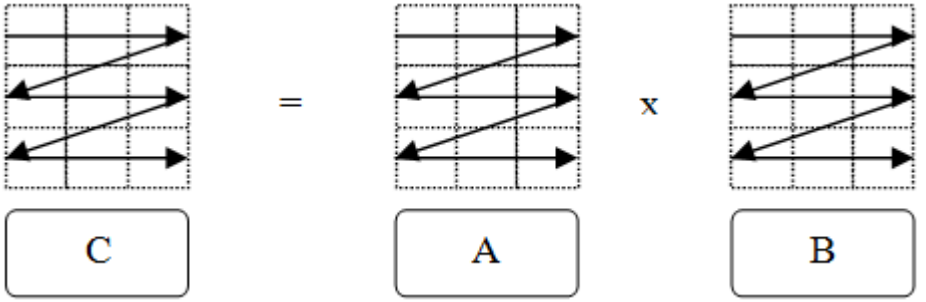
- *Почему времена так сильно отличаются?*
- Одна из главных причин низкой производительности при «неправильном порядке» циклов – плохо организованная работа с памятью.
- Процессоры чувствительны к тому, в каком порядке происходит чтение и запись в память.
- Если чтение и запись происходит последовательно, то процессор может это предсказать и заранее загрузить данные в кэш-память.
  - Доступ к кэш-памяти гораздо быстрее доступа к оперативной памяти.
- Данные в кэш-память загружаются не поэлементно, а сразу группами (размер кэш-линейки равный 64 КБ.).
  - Если из кеш-линии использовать только один элемент, то много данных будет загружено в процессор и не использовано в вычислениях.

Время выполнения алгоритма в зависимости от порядка циклов

- Порядок ijk:



- Порядок jki:



О работе с памятью

- *Следует отметить, что низкая производительность программ из-за неудачно организованной работы с памятью является общей тенденцией и проявляется не только при реализации алгоритма умножении матриц, но во многих других практически значимых задачах.*

Векторизация вычислений

Векторизация вычислений…

- Существенная влияние на производительность программ на Intel Xeon Phi оказывает применении векторных инструкций сопроцессора.
- Если производительность алгоритма недостаточно высокая, полезно проверить, смог ли компилятор «векторизовать» код.
- Соберем отчет о векторизации, добавив ключ компиляции:

```
icpc -mmic -mkl -openmp -vec-report3
./single.cpp ./mainBlock.cpp
./routine.cpp -osingle
```

Векторизация вычислений…

- Результат отчета:

```
./single.cpp(8): (col. 5) remark: loop was not
  vectorized: loop was transformed to memset or memcpy
./single.cpp(7): (col. 3) remark: loop was not
  vectorized: not inner loop
./single.cpp(14): (col. 7) remark: LOOP WAS VECTORIZED
./single.cpp(14): (col. 7) remark: PEEL LOOP WAS
  VECTORIZED
./single.cpp(14): (col. 7) remark: REMAINDER LOOP WAS
  VECTORIZED
./single.cpp(14): (col. 7) remark: loop skipped:
  multiversioned
./single.cpp(13): (col. 5) remark: loop was not
  vectorized: not inner loop
./single.cpp(12): (col. 3) remark: loop was not
  vectorized: not inner loop
```

Векторизация вычислений…

- Из отчета видно, что внутренний цикл векторизован.
- При этом компилятор создал несколько версий кода.
  - Как минимум создана версия, использующая векторные инструкции и аналогичная скалярная версия.
- Компилятор заранее не знает размеров матрицы. Также учитывая, что матрицы передаются в функцию как указатель на массив, массивы теоретически могут пересекаться.
- *Как следствие, компилятор обязан предположить, что реализацию алгоритма векторизовать нельзя.*
- Д помощи компилятору можно использовать директивы simd, ivdep или ключевое слово restrict.
- *В рамках данной лабораторной работы ограничимся использованием директивы simd.*

Векторизация вычислений…

- Применим директиву simd к внутреннему циклу.

```
void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n)
{
  ELEMENT_TYPE s;
  int i, j, k;
  for(j = 0; j < n; j++ )
    for(i = 0; i < n; i++ )
      C[j * n + i] = 0;


  for(j = 0; j < n; j++ )
    for(k = 0; k < n; k++ )
#pragma simd
      for(i = 0; i < n; i++ )
        C[j * n + i] += A[j * n + k] * B[k * n + i];
}
```

Векторизация вычислений…

- Еще раз соберем отчет о векторизации.

```
./single.cpp(7): (col. 3) remark: loop was not
  vectorized: not inner loop
./single.cpp(15): (col. 7) remark:
  SIMD LOOP WAS VECTORIZED
./single.cpp(15): (col. 7) remark:
  PEEL LOOP WAS VECTORIZED
./single.cpp(15): (col. 7) remark:
  REMAINDER LOOP WAS VECTORIZED
./single.cpp(13): (col. 5) remark: loop was not
  vectorized: not inner loop
./single.cpp(12): (col. 3) remark: loop was not
  vectorized: not inner loop
```

- *Из отчета видно, что осталась только векторная ветка кода.*

Векторизация вычислений…

- Для векторизации важно, что бы данные были выровнены по границе 64 байт.
- Если данные не выровнены, то цикл разбивается на три части.
  - Первая часть – часть до первого выровненного элемента.
  - Вторая часть – основная часть цикла, векторизованная.
  - Третья часть – «хвост» цикла.
- Наличие подобного разделения цикла может снизить производительность.

Векторизация вычислений

- Для оптимизации выполнения программ можно воспользоваться специальной функцией выделения памяти с выравниванием `__mm_malloc`.

```
#include <immintrin.h>
…
void allocMatrix(ELEMENT_TYPE ** mat, int n)
{
  // выделяем память, выровненную по 64 байт
  (*mat) = (ELEMENT_TYPE *)
           _mm_malloc(sizeof(ELEMENT_TYPE)* (n * n), 64);
}
void freeMatrix(ELEMENT_TYPE ** mat)
{
  _mm_free((*mat));
}
```

Компиляция и запуск

- Время работы алгоритма умножения матриц при добавлении директивы simd и выравнивании памяти на Intel Xeon Phi (время в секундах).

N	jki	jki + simd + _mm_malloc
1024	2,085	**1,912**
2048	15,929	**15,126**
3072	52,934	**50,816**

Реализация блочного алгоритма умножения матриц

Использование блока квадратной формы

Использование блока квадратной формы…

- Реализуем блочный алгоритм.
- В первой версии будем использовать два предположения:
  - Блоки квадратные;
  - порядок матрицы делится на размер блока без остатка.
- Для реализации изменим прототип функции умножения (добавим в параметры размер блока):

```
// multBloc.h
#ifndef _MULT_
#define _MULT_

#include "routine.h"

//умножение матриц
void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n, int bSize);

#endif
```

Использование блока квадратной формы…

- Модифицируйте функцию `main()`
  - После чтения размера матриц добавьте чтение размера блока.
  - В вызове функции умножения матриц добавьте параметр – размер блока.

Использование блока квадратной формы…

- Реализуем функцию блочного умножения:

```
//singleBlock.cpp
#include "mult.h"
#include "assert.h"

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n, int bSize)
{
  ELEMENT_TYPE s, err;
  int i, j, k, ik, jk, kk;

  assert(n % bSize == 0);

  for(j = 0; j < n; j++ )
  {
    for(i = 0; i < n; i++ )
    {
      C[j * n + i] = 0;
    }
  }
```

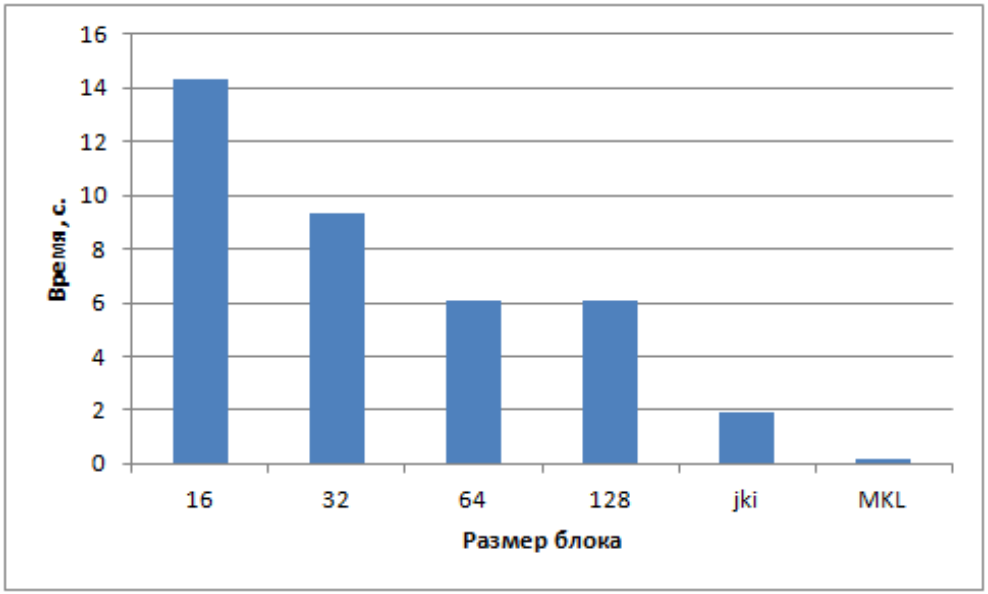
Использование блока квадратной формы…

```
 for(jk = 0; jk < n; jk+= bSize)
    for(kk = 0; kk < n; kk+= bSize)
      for(ik = 0; ik < n; ik+= bSize)
        for(j = 0; j < bSize; j++ )
          for(k = 0; k < bSize; k++ )
#pragma simd
            for(i = 0; i < bSize; i++ )
              C[(jk + j) * n + (ik + i)] +=
                A[(jk + j) * n + (kk + k)] *
                B[(kk + k) * n + (ik + i)];
}
```

Компиляция и запуск

- Время работы алгоритма умножения матриц при разных размерах блока на Intel Xeon Phi (время в секундах)

Размер блока:	16	32	64	128	jki	MKL seq.
N=1024	14,3	9,318	6,058	6,065	1,95	0,207



Использование блока квадратной формы…

- Из результатов экспериментов видно, что блочный алгоритм вместо ускорения дал замедление.
- *В чем может быть причина?*

- Соберем отчет об оптимизации программы:

```
icpc -mmic -mkl -openmp -opt-report=3
./singleBlock.cpp ./mainBlock.cpp
./routine.cpp -osingleBlock
```

Использование блока квадратной формы…

```
…
./singleBlock.cpp(12:5-12:5):VEC:_Z4multPfS_S_ii:  loop was not vectorized: loop was transformed to memset or
      memcpy
./singleBlock.cpp(10:3-10:3):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
./singleBlock.cpp(29:13-29:13):VEC:_Z4multPfS_S_ii:  LOOP WAS VECTORIZED
loop skipped: multiversioned
./singleBlock.cpp(26:11-26:11):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
./singleBlock.cpp(24:9-24:9):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
./singleBlock.cpp(22:3-22:3):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
./singleBlock.cpp(21:3-21:3):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
./singleBlock.cpp(20:3-20:3):VEC:_Z4multPfS_S_ii:  loop was not vectorized: not inner loop
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 26=6
  # of initial-value prefetches in _Z4multPfS_S_ii for loop at line 26=1
  # of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 26=6, dist=6
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 29=8
  # of initial-value prefetches in _Z4multPfS_S_ii for loop at line 29=6
  # of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 29=8, dist=8
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 29=8
  # of initial-value prefetches in _Z4multPfS_S_ii for loop at line 29=6
  # of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 29=8, dist=8
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 29=4
  # of initial-value prefetches in _Z4multPfS_S_ii for loop at line 29=2
  # of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 29=4, dist=64
…
```

Использование блока квадратной формы…

- Из отчета видно, что компилятор добавил в код много вызовов программной предвыборки данных.
  - При генерации кода компилятор заранее не знает, чему будут равны параметры, передаваемые в функцию.
  - Компилятор должен генерировать код, используя введенные в нем эвристики.
  - Компилятор мог предположить, что внутренний цикл имеет большую длину
  - В нашем случае цикл имеет малый размер
  - Как следствие, в кэш-память загружается много не используемых данных, что негативно влияет на производительность.
- **Используем задание размера блока в виде константы.**
  - В результате подобных действий компилятор иногда может избавиться от короткого цикла, полностью развернув его и реализовав с использованием векторной арифметики.

Использование блока квадратной формы

- ***Еще раз соберем отчет компилятора об оптимизации:***

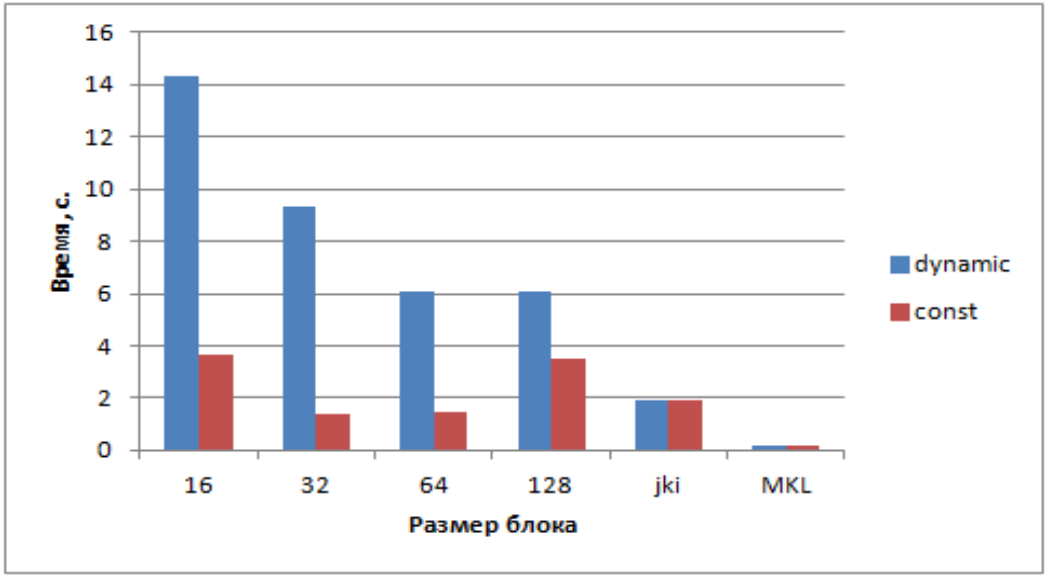
./singleBlock.cpp(15:5-15:5):VEC:_Z4multPfS_S_ii: loop was not vectorized: loop was transformed to memset or memcpy
./singleBlock.cpp(13:3-13:3):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
./singleBlock.cpp(30:17-30:17):VEC:_Z4multPfS_S_ii: LOOP WAS VECTORIZED
./singleBlock.cpp(27:11-27:11):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
./singleBlock.cpp(25:9-25:9):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
./singleBlock.cpp(23:3-23:3):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
./singleBlock.cpp(22:3-22:3):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
./singleBlock.cpp(21:3-21:3):VEC:_Z4multPfS_S_ii: loop was not vectorized: not inner loop
Estimate of max_trip_count of loop at line 27=128
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 27=2
# of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 27=2, dist=8
Estimate of max_trip_count of loop at line 30=4
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 30=4
# of initial-value prefetches in _Z4multPfS_S_ii for loop at line 30=6
# of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 30=4, dist=2
Estimate of max_trip_count of loop at line 30=4
Total #of lines prefetched in _Z4multPfS_S_ii for loop at line 30=4
# of initial-value prefetches in _Z4multPfS_S_ii for loop at line 30=6
# of dynamic_mapped_array prefetches in _Z4multPfS_S_ii for loop at line 30=4, dist=2
Using second-level distance 2 for prefetching dyn-map memory reference in stmt at line 30

- **Предсказаний загрузки данных в кэш-память стало меньше, также изменились их параметры.**

Компиляция и запуск

- Время работы алгоритма умножения матриц при разных размерах блока, заданных константой, на Intel Xeon Phi (время в секундах).

Размер блока:	16	32	64	128	jki	MKL seq.
N=1024 Размер блока параметр	14,3	9,318	6,058	6,065	1,95	0,207
N=1024 Размер блока константа	3,68	1,4	1,45	3,48	1,95	0,207



Задание

- Вместо задания размера блока равного константе также можно попытаться использовать директивы компилятора, такие как #pragma loop_count
- *Попробовать данные директивы предлагается самостоятельно.*
- В рассматриваемом примере приведены результаты только при четырех размерах блока.
- *В качестве дополнительного задания предлагается найти оптимальный размер блока для рассматриваемых размеров матриц (1024, 2048 и 3072).*

Использование блоков прямоугольной формы

Использование блоков прямоугольной формы…

- Разработаем еще одну реализацию блочного умножения матриц, в которой в качестве блока матрицы B берется не квадратная область, а *горизонтальная полоса.*
- Блок матрицы A остается квадратным.
- Также сохраним предположение о том, что порядок матрицы делится на размер блока без остатка.

Использование блоков прямоугольной формы…

- Реализация вычислительного цикла:

```
for(int j = 0; j < n; j++ )
    for(int i = 0; i < n; i++ )
      C[j * n + i] = 0;

  for(int jk = 0; jk < n / bSize; jk++ )
    for(int ik = 0; ik < n / bSize; ik++ )
      for(int j = jk * bSize;
          j < jk * bSize + bSize;
          j++)
        for(int k = ik * bSize;
            k < ik * bSize + bSize;
            k++)
#pragma simd
          for(int i = 0; i < n; i++ )
            C[j * n + i] += A[j * n + k] * B[k * n + i];
```

Использование блоков прямоугольной формы

- Данный вариант обладает рядом преимуществ.
  - Во-первых, элементы блока матрицы A повторно используются несколько раз, как и при реализации с квадратными блоками. Как следствие, часть данных читается не из памяти, а из кэш-памяти.
  - Во-вторых, если матрица сравнительно небольшого размера, то полоса матрицы B и C может поместиться в кэш-память последнего уровня, тем самым повышая эффективность использования памяти.
  - В-третьих, внутренний цикл содержит много итераций, что, по всей видимости, лучше соответствует эвристикам компилятора.

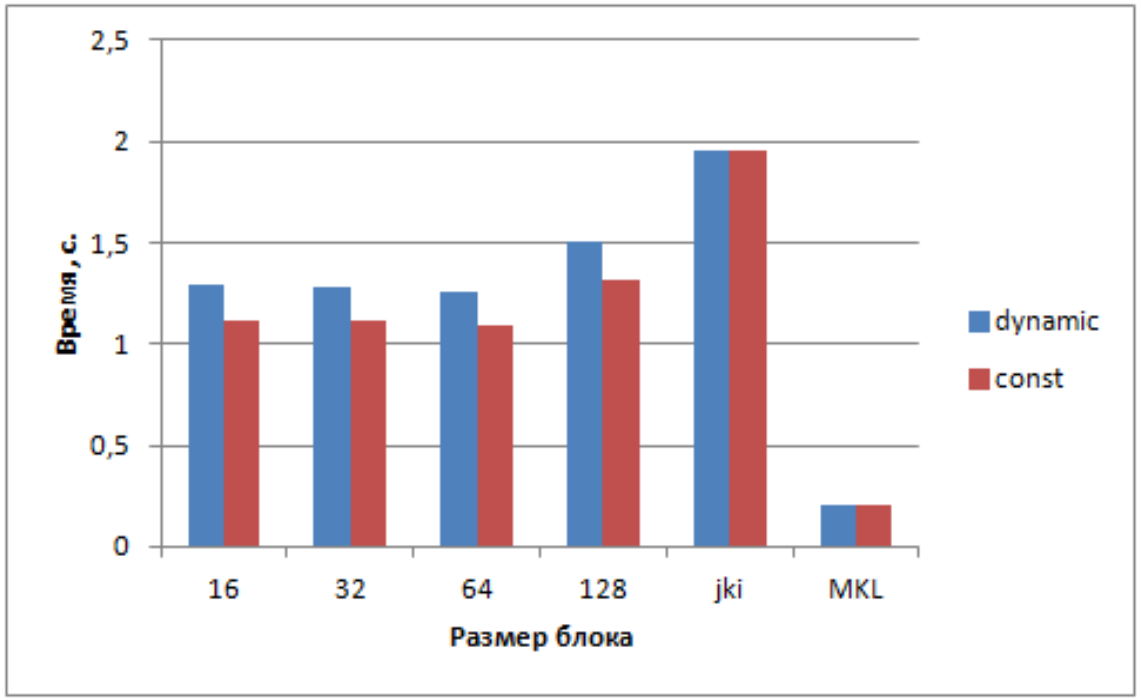
Компиляция и запуск…

- Сравнение времени работы блочной реализации алгоритма и Intel MKL на Intel Xeon Phi (время в секундах).

Размер блока:	16	32	64	128	jki	MKL seq.
N=1024 Размер блока параметр	1,29	1,28	**1,26**	1,51	1,95	0,207
N=1024 Размер блока константа	1,12	1,11	**1,09**	1,32	1,95	0,207
N=2048 Размер блока константа	8,53	**8,34**	15,05	15,05	15,4	1,363
N=3072 Размер блока константа	28,97	**28,07**	49,4	50,6	50,8	4,411

Компиляция и запуск

- Время работы алгоритма в зависимости от размера блока (N = 1024) на Intel Xeon Phi.



Задания

- При вычислениях с матрицами большого размера происходить резкое замедление при использовании размера блока, равного 64.
  - По всей видимости, при переходе с размера блока равного 32 на 64 полоса матрицы B начинает не помещаться в кеш-память.
- Для решения данной проблемы на матрицах еще большего размера необходимо в качестве блока матрицы B взять только часть полосы, т.е. *использовать прямоугольные блоки.*
- *В качестве дополнительного задания предлагается реализовать блочное умножение матриц с прямоугольными блоками и подобрать оптимальный размер блока.*

Параллельная реализация

Параллельная реализация…

- Попробуем распараллелить реализованный алгоритм, используя директивы OpenMP. Достаточно добавить одну директиву.

```
for(int j = 0; j < n; j++ )
    for(int i = 0; i < n; i++ )
      C[j * n + i] = 0;

#pragma omp parallel for
  for(int jk = 0; jk < n / bSize; jk++ )
    for(int ik = 0; ik < n / bSize; ik++ )
      for(int j = jk * bSize; j < jk * bSize + bSize; j++)
        for(int k = ik * bSize; k < ik * bSize + bSize;k++)
#pragma simd
          for(int i = 0; i < n; i++ )
            C[j * n + i] += A[j * n + k] * B[k * n + i];
}
```

Компиляция и запуск

- Сравнение времени работы параллельной блочной реализации алгоритма и Intel MKL на Intel Xeon Phi. (время в секундах)

Размер блока:	32	64	MKL parallel
N=1024	0,040	0,070	0,004
N=2048	0,250	0,370	0,019
N=3072	0,760	1,000	0,062

Параллельная реализация…

- Из таблицы видно, что время вычислений отличается от Intel MKL на порядок.

- *Можно ли еще ускорить работу реализованного алгоритма?*

- Умножение матриц очень интенсивно использует память.
- При большом количестве потоков существенно возрастает нагрузка на шину данных.
- Для уменьшения нагрузки можно попробовать сократить количество потоков, используя функцию OpenMP – **`omp_set_num_threads()`**.

Параллельная реализация…

- Сократим количество потоков до 120:

```
for(int j = 0; j < n; j++ )
    for(int i = 0; i < n; i++ )
      C[j * n + i] = 0;

omp_set_num_threads(120);

#pragma omp parallel for
  for(int jk = 0; jk < n / bSize; jk++ )
    for(int ik = 0; ik < n / bSize; ik++ )
      for(int j = jk * bSize; j < jk * bSize + bSize; j++)
        for(int k = ik * bSize; k < ik * bSize + bSize;k++)
#pragma simd
          for(int i = 0; i < n; i++ )
            C[j * n + i] += A[j * n + k] * B[k * n + i];
}
```

Компиляция и запуск

- Сравнение времени работы параллельной блочной реализации алгоритма и Intel MKL на Intel Xeon Phi. (время в секундах)

Размер блока:	244 потока		120 потоков		MKL parallel
	32	64	32	64	
1024	**0,04**	0,07	**0,04**	0,08	0,004
2048	0,25	0,37	**0,16**	0,35	0,019
3048	0,76	1	**0,44**	0,94	0,062

Заключение

- В целом работа над кодом может быть продолжена.
- Желающие достигнуть результаты, показываемые Intel MKL, могут обратиться к соответствующей литературе, например:

- Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy , Jeff R. Hammond, and Field G. Van Zee. Opportunities for Parallelism in Matrix Multiplication // FLAME Working Note #71 . The University of Texas at Austin, Department of Computer Science. Technical Report TR-13-20. 2013.

- Для достижения производительности Intel MKL, по всей вероятности, обойтись исключительно использованием языка C не удастся, потребуется программировать на ассемблере (или в интринсиках).

Дополнительные задания и литература

Задания для самостоятельной работы

- Подберите оптимальные размеры блоков для разных версий блочной реализации алгоритма умножения матриц (с квадратным блоком, с прямоугольным блоком).
- Подберите оптимальное количество создаваемых OpenMP потоков для реализованного блочного алгоритма умножения матриц в зависимости от размера матриц и блоков.
- Исследуйте влияние прогрева на время работы созданных программ.

Литература…

1. Страница, посвященная векторному расширению SIMD на сайте корпорации Intel®. – URL: http://software.intel.com/en-us/node/462300 (дата обращения: 10.12.2013)
2. Страница технологии Intel® Cilk™ Plus на сайте корпорации Intel®. – URL: http://software.intel.com/en-us/intel-cilk-plus (дата обращения: 10.09.2013)
3. Международный Вычислительный Центр Российской Академии Наук. – URL: http://www.jscc.ru/ (дата обращения: 10.09.2013)
4. Гергель В.П. Теория и практика параллельных вычислений // БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2007г.
5. Описание параметров ?gemm. – URL: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-97718E5C-6E0A-44F0-B2B1-A551F0F164B2.htm (дата обращения: 10.09.2013)

Литература

6. Лекция №2. Архитектура Intel Xeon Phi.
7. Лекция №4. Векторные расширения Intel Xeon Phi.
8. Лабораторная работа №4. Оптимизация расчетов на примере задачи вычисления справедливой цены опциона Европейского типа.
9. Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, George Chrysos, Pradeep Dubey Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel® Xeon Phi™ co-processor. //In Proceedings of the 2013 IEEE International Parallel and Distributed Processing Symposium, May 2013.
10. Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy , Jeff R. Hammond, and Field G. Van Zee. Opportunities for Parallelism in Matrix Multiplication // FLAME Working Note #71 . The University of Texas at Austin, Department of Computer Science. Technical Report TR-13-20. 2013.
11. Kazushige Goto, Robert van de Geijn. Anatomy of high-performance matrix multiplication //ACM Trans. Math. Soft., 34(3), May 2008.

Авторский коллектив

- Козинов Евгений Александрович,
  ассистент кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ
  kozinov@vmk.unn.ru
- Мееров Иосиф Борисович,
  к.т.н., доцент, зам. зав. кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ
  meerov@vmk.unn.ru
- Сиднев Алексей Александрович,
  ассистент кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ
  sidnev@vmk.unn.ru