Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Введение в принципы функционирования и применения современных мультиядерных архитектур (на примере Intel Xeon Phi)»

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

Козинов Е.А., Мееров И.Б., Сиднев А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

С одной стороны, программирование для Intel Xeon Phi по своей сути мало чем отличается от программирования для традиционных центральных процессоров. Как и раньше, мы должны эффективно использовать векторные наборы команд (парадигма SIMD, параллелизм по данным), обеспечивать по возможности равномерную загрузку потоков (параллелизм на уровне ядер), эффективно работать с памятью (учет многоуровневой схемы организации памяти).

Методы достижения цели также являются в определенной степени стандартными. Так, для векторизации вычислений мы можем использовать высокопроизводительные библиотеки, возможности оптимизирующих компиляторов, помогая им специальными опциями и директивами, применять технологию Intel Cilk Plus, при необходимости реализовывать некоторые части программы с использованием интринсиков или ассемблерных вставок. При этом более широкие регистры, наличие FMA, расширенные операции для работы с памятью и некоторые другие особенности векторных расширений в системе команд Xeon Phi способны значительно ускорить расчеты. Для достижения параллелизма на уровне ядер мы можем использовать хорошо знакомые технологии (MPI, OpenMP, Intel Cilk Plus и др.). При этом выбор подходящего сочетания числа процессов и потоков может оказывать существенное влияние на итоговую производительность программы. Базовые методы оптимизации работы с памятью также являются достаточно стандартными. Так, мы должны заботиться об эффективном использовании кеш-памяти, стараться избегать перегрузки шины данных, а также минимизировать обмены с оперативной памятью хоста. В целом, оптимизация на традиционных процессорах Intel часто приводит к сокращению времени работы на Xeon Phi и наоборот. Тем не менее, это не одно и то же. В лекциях и лабораторных работах данного курса на примерах демонстрируется, как меняется время работы на хосте и на сопроцессоре в результате применения различных техник оптимизации.

Основная идея данной работы заключается в изучении базовых принципов применения перечисленных выше способов повышения производительности на примере известной, широко распространенной в приложениях и хорошо исследованной в литературе задачи – умножении плотных матриц. В отличие от предыдущих лабораторных работ, подробно изучающих разные техники оптимизации расчетов, основное внимание уделяется вопросам оптимизации работы с памятью.

1. Методические указания

1.1. Цели и задачи работы

Цель данной лабораторной работы – рассмотрение вопросов оптимизации работы с памятью при разработке программ для Intel Xeon Phi. Основное внимание уделяется базовым методам повышения эффективности использования кеш-памяти. В качестве примера используется задача умножения квадратных плотных матриц. Считается, что все данные, участвующие в расчетах, целиком помещаются в оперативную память.

Данная цель предполагает решение следующих основных *задач*:

1. Использование библиотеки Intel MKL для решения задачи умножения матриц на хосте и на сопроцессоре.
2. Последовательная реализация алгоритма умножения матриц по определению.
3. Изучение вопроса о производительности базовой версии, рассмотрение способов сокращения времени вычислений с использованием векторизации, выбор подходящего порядка обращения к данным.
4. Реализация последовательной и параллельной версий блочного алгоритма для умножения матриц.

1.2. Структура работы

Рассматривается задача умножения плотных матриц. Для сравнения результатов вычислений и времени работы реализуется умножение плотных матриц с использованием высокопроизводительной библиотеки Intel MKL. Приводится наивная реализация алгоритма умножения матриц. Рассматриваются вопросы влияния порядка циклов на время вычислений на сопроцессор Intel Xeon Phi. Изучается влияние выравнивания и подсказок компилятору на векторизацию вычислений. Реализуется несколько редакций блочного алгоритма умножения матриц. Формулируются задания для самостоятельной проработки.

1.3. Тестовая инфраструктура

Для проведения экспериментов использовались вычислительные ресурсы МСЦ РАН (МВС-10П) [3]. Тестовая инфраструктура представлена в таблице 1.

Таблица 1. 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

2. Задача умножения плотных матриц

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:

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

$$0 \le i < n,$$

$$0 \le j < n$$

Этот алгоритм предполагает выполнение n^3 операций умножения и $n^2(n-1)$ операций сложения элементов исходных матриц ($O(n^3)$ операций).

Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью. В частности, к таким алгоритмам относится алгоритм Штрассена. В данном алгоритме каждое восьмое умножение заменяется сложениями, что приводит к оценке сложности $\Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2,81})$. Еще меньшей оценкой сложности обладает алгоритм Копперсмита-Винограда $O(n^{2,376})$. Данный алгоритм был улучшен в 2010 году Вирджинией Вильямс. Сложность алгоритма составила $O(n^{2,373})$. Ряд исследователей считают, что данную оценку можно существенно улучшить. Желающие ознакомиться с теоретическими аспектами данной проблемы могут, например, изучить обзор алгоритмов в диссертации Вирджинии Вильямс (http://theory.stanford.edu/~virgi/thesis.pdf) и ряд публикаций, свободно доступных с ее персональной страницы (http://theory.stanford.edu/~virgi).

Несмотря на существование более совершенных алгоритмов с точки зрения асимптотики, на практике их применяют редко. Как правило, в оценке сложности присутствует большая константа. Как следствие, алгоритмы

обладают лучшей производительностью только при слишком больших размерах матриц, не помещающихся в оперативную память современных компьютеров.

В рамках лабораторной работы для демонстрации подходов к реализации и последующей оптимизации алгоритма умножения матриц на сопроцессоре Intel Xeon Phi будем использовать алгоритм, получаемый из определения. Будем считать, что матрицы хранят данные с плавающей запятой в одинарной точности.

3. Применение Intel MKL для умножения плотных матриц

В C/C++ для умножения матриц с использованием Intel MKL необходимо использовать функцию BLAS третьего уровня – `cblas_?gemm`. На месте вопросительного знака в имени функции применяется символ, идентифицирующий используемый тип данных. Так, для типа float используется символ «**s**». Функция позволяет вычислять выражения:

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

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

Далее представлен пример вызова функции:

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

Рассмотрим значение параметров вызова.

– Значение CblasRowMajor показывает, что матрица хранится по строкам.

– Значение CblasNoTrans сообщает, что исходные 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.

Более подробную информацию о параметрах можно получить в официальной документации по Intel MKL [5].

Разделим реализацию алгоритма умножения матриц на несколько файлов. В начале, объявим определение вещественного типа данных и ряд прототипов функций:

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

#ifndef ELEMENT_TYPE
#define ELEMENT_TYPE float
#endif

// выделение памяти для хранения матриц
void allocMatrix(ELEMENT_TYPE ** mat, int n);
// освобождение памяти
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
```

Реализацию функций выделения/освобождения памяти и генерации плотных матриц читателю предлагается реализовать самостоятельно. Для определения корректности разработанных нами программ в дальнейшем предлагается сравнивать результат с результатом работы функции умножения матриц, предоставляемой в Intel MKL. Реализовать подобную функцию также предлагается самостоятельно.

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
```

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);
}
```

Подготовим функцию main():

```
// main.cpp
#include <stdio.h>
#include "omp.h"

#include "mult.h"

int testThreadCount();

int main(int argc, char **argv)
{
  double time_s, time_f;
  int th;
  int n = 0;

  // чтение размера матиц
  if(argc < 2)
  {
    printf("<exec> <n>");
    return -1;
  }

  n = atoi(argv[1]);

  // вывод количества потоков
  th = testThreadCount();
  printf("Intel Xeon Phi threads number: %d \n\n", th);

  ELEMENT_TYPE *A, *B, *C;

  // выделение памяти
  allocMatrix(&A, n);
  allocMatrix(&B, n);
  allocMatrix(&C, n);

  // генерация матриц A и B
  genMatrix(A, n);
  genMatrix(B, n);

  // умножение матриц
  time_s = omp_get_wtime( );
```

```
  mult(A, B, C, n);
  time_f = omp_get_wtime( );

  printf( "Size of matrix   : %d \n", n);
  printf( "Calculation time : %3.3f \n",
          (time_f - time_s));

  // проверка корректности реализованного алгоритма
#ifndef NO_DEBUG
  double err;
  err = calculationError(A, B, C, n);
  printf( "Calculation error : %3.5f \n", err);
#endif

  // освобождение памяти
  freeMatrix(&A);
  freeMatrix(&B);
  freeMatrix(&C);

  return 0;
}

// функция, определяющая количество потоков
int testThreadCount() {
  int thread_count;
  #pragma omp parallel
  {
    #pragma omp single
      thread_count = omp_get_num_threads();
  }
  return thread_count;
}
```

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
```

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
```

Запустим полученный код. На Рис. 1 представлен результат запуска разработанной программы на сопроцессоре Intel Xeon Phi.

```
/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 $
```

Рис. 1. Пример запуска программы на сопроцессоре Intel Xeon Phi с использованием Intel MKL

В таблице 2 представлены результаты запуска умножения матриц на хосте и на сопроцессоре в последовательном и параллельном режиме при разных размерах матрицы. Для замера времени здесь и далее проводилась серия экспериментов, в качестве времени выполнения бралось минимальное время работы программы.

Таблица 2. Сравнение времени умножения матриц (Intel MKL) в разных конфигурациях (время в секундах)

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

Сравним время работы на хосте и на сопроцессоре. 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. Да, конечно, их намного больше, но ведь в последовательных запусках мы это никак не используем!

Обратим внимание на параллельные запуски. Здесь мы ожидаем, что MKL на сопроцессоре покажет лучший результат, чем MKL на 16 ядрах CPU. Тем не менее, этого не происходит. Попробуем разобраться с этим фактом. Вспомним результаты, полученные нами в предыдущей лабораторной работе. Там нами было выяснено, что так называемый «прогрев» (warm up) оказывает существенное влияние на время работы программы, использу-

ющей MKL (а иногда и не только), 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[1]. В той же работе обсуждался вопрос о корректности использования прогрева и доверия соответствующим результатам. Забегая вперед, скажем, что для всех последующих наших реализаций прогрев при существенных размерах матриц не оказывает значимого воздействия на время работы, поэтому мы будем приводить результаты, не уточняя, использовался прогрев или нет. В то же время, для MKL наличие прогрева существенно влияет на время работы. Учитывая, что современные программы обычно делают что-то более значимое, чем однократное умножение двух матриц, будем анализировать результаты MKL, полученные с использованием прогрева – в оптимальной для библиотеки конфигурации.

Таблица 3. Сравнение времени умножения матриц
(Intel MKL, включая прогрев)
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

Перестроим таблицу (см. таблица 3), добавив результаты с прогревом и для N = 10000. Здесь мы видим, что 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.

[1] Те, кто не очень верят в «шаманство» с прогревом или считают это каким-то жульничеством, могут провести следующий эксперимент. Создаем программу, в которой запускаем sgemm, выполняющий C=D*E. Далее той же программе в цикле, скажем, 10 раз запускаем sgemm, выполняющий C=A*B. Авторы обнаружили, что первый sgemm(D, E) работает значительно дольше, чем первый из sgemm(A, B), который, в свою очередь, несколько быстрее, чем следующий sgemm(A, B). Далее время стабилизируется. Эти результаты позволяют сделать предположения о том, какие из эффектов прогрева оказывают влияние на время работы и в какой момент это происходит.

4. Последовательная реализация алгоритма умножения матриц

Попробуем разработать собственную реализацию алгоритма умножения матриц. В начале, реализуем алгоритм согласно определению. Затем попробуем повысить эффективность реализации с точки зрения времени вычислений.

4.1. 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];
}
```

Укажем в строке компиляции файл с реализацией алгоритма:

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

Скомпилируем и запустим приложение.

На Рис. 2 представлен результат работы программы.

```
/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 $
```

Рис. 2. Результат умножения матриц последовательной версии.

Итак, мы видим (таблица 4), что при N = 1024 время работы составляет около 57с. При этом время работы аналогичной (более сложной) функции в

библиотеке MKL составляет около 0,1с. Для определенности сделаем запуск для N = 1025. Заметим, что время работы нашей программы существенно сократилось, в то время как время работы функции из MKL осталось прежним. *Мистика или объективная реальность*?

Таблица 4. 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
1025	33,09	0,200	0,004

Попробуем ответить на поставленный вопрос. Тот факт, что время работы первой наивной реализации сильно отличается от «эталона», не является сюрпризом. Разработчики высокопроизводительных библиотек хорошо знают свое дело. А вот существенное сокращение времени при увеличении размера на единицу – более чем странный факт. Причина может быть найдена при помощи профилировщика, который обнаружит, что при размере матрицы N = 1024 мы обращаемся в память с таким шагом, что нужные нам данные должны быть загружены в одну и ту же кеш-линию. В результате кеш-память используется крайне неэффективно: при следующей операции с данными мы вынуждены вытеснить кеш-линейку, которую недавно загружали и планировали использовать далее, тогда как значительная часть других кеш-линеек не используется. При размере матрицы N = 1025 этого не происходит. Слушателям предлагается составить фрагмент карты обращений в память с учетом того, что кеш является 8-ассоциативным, размер кеш-линии составляет 64 байта, размер кеш-памяти составляет 32KB. Данная карта поможет понять рассмотренный выше эффект, а также пригодится для дальнейшего изучения вопроса эффективного использования кеш-памяти.

4.2. 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 (рисунок 3). Попробуем изменить порядок циклов и исследуем вопрос о том, как это влияет на время работы программы.

Всего существует шесть возможных перестановок. Попробуем каждую из них для экспериментального определения оптимального порядка циклов. Разработку реализаций алгоритма при разных порядках циклов предлагается провести самостоятельно.

В таблице 5 представлены времена вычислений:

Таблица 5. 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

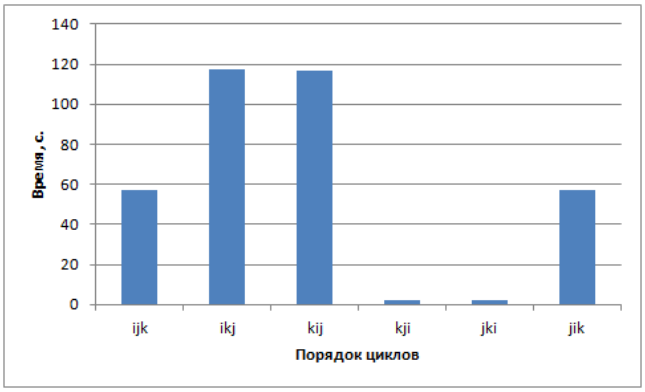

Рис. 3. Время выполнения алгоритма в зависимости от порядка циклов (время в секундах)

Почему времена так сильно отличаются?

Одна из главных причин низкой производительности при «неправильном порядке» циклов – плохо организованная работа с памятью. Современные процессоры чувствительны к тому, в каком порядке происходит чтение и запись в память. Чувствительность связана со сложной архитектурой организации памяти, как процессоров, так и сопроцессоров [6]. 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. Кроме того, 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. Обратим внимание на то, что обращение к элементам матриц A и B таково, что многие элементы читаются и используются в кеш-памяти (при больших размерах матриц) один раз.

В реализации алгоритма умножения матриц из раздела 4.1 (порядок циклов ijk соответствующий определению), обход матрицы происходит, как показано на Рис. 4.

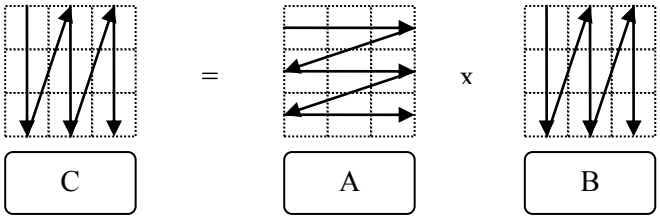

Рис. 4. Обход матриц в реализации умножения по определению

Учитывая, что матрицы хранятся непрерывным вектором, обход матриц B и C не оптимален (хоть и имеет регулярный характер). Значительную роль в снижение производительности играет обход матрицы B. Элементы матрицы B используются n^3 раз, а матрицы C – n^2 раз.

При оптимальном порядке циклов jki обход матриц показан на Рис. 5. Доступ к элементам стал последовательным. Как следствие, повысилась производительность.

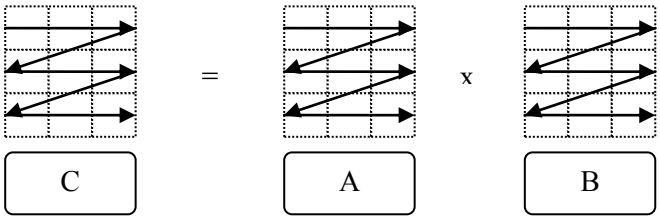

Рис. 5. Обход матриц с порядком циклов jki.

На Рис. 6 представлен результат вычислений при лучшем расположении циклов.

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

Size of matrix   : 1024
Calculation time : 2.078
Calculation error : 0.00000
/home1/unnkoz/kozinov/MatrixMult/src $
```

Рис. 6. Результат вычислений при порядке циклов jki.

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.

5. 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.

Проверим, векторизован ли внутренний цикл в алгоритме умножения матриц. Для этого соберем отчет о векторизации, добавив ключ компиляции:

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

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 VEC-
TORIZED
./single.cpp(14): (col. 7) remark: loop skipped: multiver-
sioned
./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 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. Для того, что бы помочь компилятору векторизовать код и не порождать лишних версий, можно использовать директивы simd, ivdep или ключевое слово restrict [7, 8]. В рамках данной лабораторной работы ограничимся использованием директивы simd.

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];
}
```

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 VEC-
TORIZED
./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.

Стоит также отметить, что 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. Для оптимизации выполнения программ можно воспользоваться специальной функцией выделения памяти с выравниванием `__mm_malloc`. Ниже приведен код выделения и освобождения памяти для матриц:

```
#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));
}
```

Из таблицы 6 видно, что за счет выравнивания памяти и помощи компилятору путем применения директив удалось получить чуть лучшую реализа-

цию алгоритма. Соберите отчет о векторизации и проверьте, что компилятор больше не создает «пролог» для цикла.

Таблица 6. 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

6. Реализация блочного алгоритма умножения матриц

Один из способов улучшения доступа к памяти при реализации алгоритма умножения матриц – применение блочных алгоритмов. При использовании блочных алгоритмов повышается локальность доступа к памяти. Как правило, при повышении локальности доступа к памяти повышается эффективность ее использования за счет уменьшения количества кеш-промахов.

6.1. Использование блока квадратной формы

Реализуем блочный алгоритм. В первой версии будем использовать два предположения – блоки квадратные и 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
```

Частично изменим функцию main:

```
#include <stdio.h>
#include "omp.h"

#include "multBlock.h"
```

```
int testThreadCount();

int main(int argc, char **argv)
{
  double time_s, time_f;
  int mic_th;

  int n = 0, blockSize = 1;

  if(argc < 3)
  {
    printf("<exec> <n> <block Size>");
    return -1;
  }

  n = atoi(argv[1]);
  blockSize = atoi(argv[2]);

…

  time_s = omp_get_wtime( );
  mult(A, B, C, n, blockSize);
  time_f = omp_get_wtime( );
…
  return 0;
}
```

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;
    }
  }


  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)];
}
```

Скомпилируем и запустим код. На Рис. 7 представлен результат выполнения блочного алгоритма на сопроцессоре Intel Xeon Phi.

```
/home1/unnkoz/kozinov/MatrixMult/bin $ ./singleBlock 1024 16
Intel Xeon Phi thread: 244

Size of matrix   : 1024
Calculation time : 14.248
Calculation error : 0.000000
/home1/unnkoz/kozinov/MatrixMult/bin $
```

Рис. 7. Результат выполнения блочного алгоритма на сопроцессоре Intel Xeon Phi.

В таблице 7 приведены времена работы программной реализации блочного алгоритма при разных размерах блока:

Таблица 7. 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

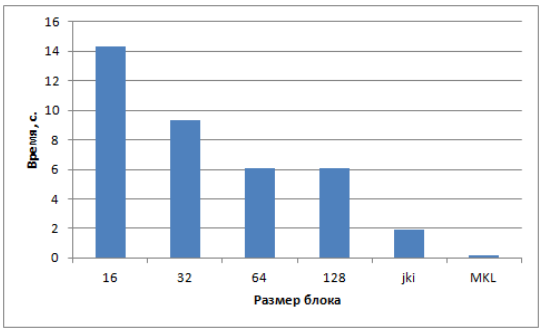

Рис. 8. Время работы алгоритма в зависимости от размера блока на Intel Xeon Phi.

Из результатов экспериментов и Рис. 8 видно, что блочный алгоритм вместо ускорения дал замедление.

В чем может быть причина?

Соберем отчет об оптимизации программы. Для этого можно использовать строку компиляции, представленную ниже:

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

Далее приведена часть отчета об оптимизации:

```
…
./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
…
```

Из отчета видно, что компилятор добавил в код много вызовов программной предвыборки данных.

Для оптимизации работы с памятью компилятор использует программную предвыборку данных. При генерации кода компилятор заранее не знает, чему будут равны параметры, передаваемые в функцию. Из-за этого компилятор должен генерировать код, используя введенные в нем эвристики. Эвристики срабатывают не всегда. Например, компилятор мог предположить, что внутренний цикл имеет большую длину и соответствующим образам вставил предсказание загрузки данных в кэш-память. В нашем случае цикл имеет малый размер (сомнительно, что большие блоки больших размеров приведут к хорошей локальности при доступе к данным). Как следствие, в кэш-память загружается много не используемых данных, что негативно влияет на производительность.

Попробуем помочь компилятору, используя задание размера блока в виде константы. Заметим, что в результате подобных действий компилятор иногда может избавиться от короткого цикла, полностью развернув его и реализовав с использованием векторной арифметики.

```
const int bSize = 64;
```

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
```

Из отчета видно, что предсказания компилятора существенным образом поменялись. В частности предсказаний загрузки данных в кэш-память стало меньше, также изменились их параметры.

Попробуем замерить время вычислений при разных размерах блока, заданных при помощи константы и через параметр функции. В таблице 8 приведены результаты сравнения.

Таблица 8. 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

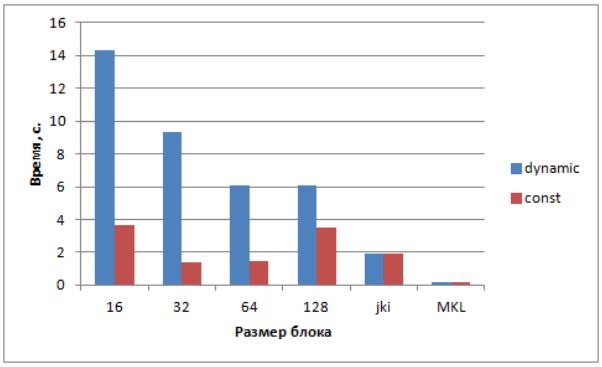

Рис. 9. Время работы алгоритма в зависимости от размера блока и типа его задания на Intel Xeon Phi.

Из Рис. 9 и таблицы 8 видно, что блочный алгоритм позволил уменьшить время вычислений. Время сократилось из-за уменьшения количества кэш-промахов.

Следует отметить, что вместо задания размера блока равного константе также можно попытаться использовать директивы компилятора, такие как `#pragma loop_count`, позволяющую компилятору подсказать информацию о характерных длинах цикла. 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).

6.2. 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.

```
//singleBlock_t.cpp
#include "mult.h"
#include "assert.h"
#include <omp.h>

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n, int bSize)
{
  ELEMENT_TYPE s, err;

  assert(n % bSize == 0);

  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];
}
```

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. Подробнее о преимуществах данного подхода можно прочитать в работах [9, 10, 11].

Попробуем скомпилировать и запустить полученный код. На Рис. 10 представлен результат запуска кода с измененным размером блока матрицы B.

```
/home1/unnkoz/kozinov/MatrixMult/bin $ ./singleBlock_t 1024 64
Intel Xeon Phi thread: 244

Size of matrix   : 1024
Calculation time : 1.290
Calculation error : 0.000000
/home1/unnkoz/kozinov/MatrixMult/bin $
```

Рис. 10. Результат запуска алгоритма с блоком не квадратной формы

В таблице 9 представлено сравнение времени выполнения последовательного блочного алгоритма с разным размером блока и Intel MKL. Как и в случае с блочным алгоритмом для размера 1024 в начале размер блока задавался через параметр, а затем была использована константа. Для матриц размера 2048 и 3073 размер блока задавался константой.

Таблица 9. 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

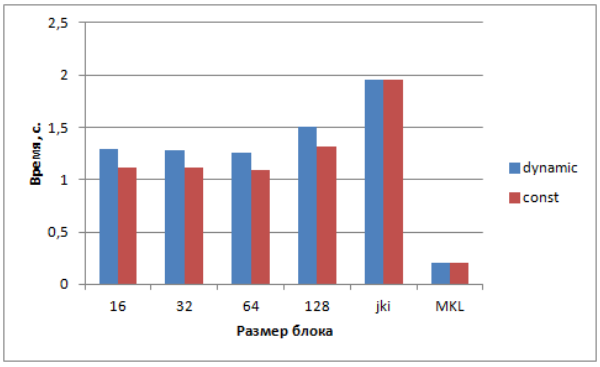

Рис. 11. Время работы алгоритма в зависимости от размера блока (N = 1024) на Intel Xeon Phi.

Новая блочная реализация алгоритма позволила получить лучшие результаты, чем версия с квадратными блоками.

Анализируя таблицу можно заметить следующий факт – 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.

7. Параллельная реализация

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

```
#include "mult.h"
#include "assert.h"
#include <omp.h>

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n, int bSize)
{
  ELEMENT_TYPE s, err;

  assert(n % bSize == 0);

  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];
}
```

Скомпилируем и выполним код. На Рис. 12 представлен результат вычислений.

```
/home1/unnkoz/kozinov/MatrixMult/bin $ ./parBlock 1024 32
Intel Xeon Phi thread: 244

Size of matrix   : 1024
Calculation time : 0.040
Calculation error : 0.000000
/home1/unnkoz/kozinov/MatrixMult/bin $
```

Рис. 12. Результат выполнения параллельного алгоритма

В таблице 10 приведено сравнение времени выполнения параллельного алгоритма с Intel MKL.

Таблица 10. 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

Из таблицы 10 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`. Сократим количество потоков до 120:

```
#include "mult.h"
#include "assert.h"
#include <omp.h>

void mult(ELEMENT_TYPE * A, ELEMENT_TYPE * B,
          ELEMENT_TYPE * C, int n, int bSize)
```

```
{
  ELEMENT_TYPE s, err;

  assert(n % bSize == 0);

  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];
}
```

В таблице 11 приведено сравнение времени выполнения параллельного алгоритма в зависимости от размера блока с Intel MKL при 120 потоках OpenMP.

Таблица 11. 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

Как видно из таблицы 11, отставание от Intel MKL на больших матрицах несколько сократилось. В качестве дополнительного задания предлагается подобрать оптимальное количество потоков.

В целом работа над кодом может быть продолжена. Желающие достигнуть результаты, показываемые Intel MKL, могут обратиться к соответствующей литературе (см., например, 10). При этом, по всей вероятности, обойтись исключительно использованием языка C не удастся, потребуется программировать на ассемблере (или в интринсиках).

8. Дополнительные задания

1. Подберите оптимальные размеры блоков для разных версий блочной реализации алгоритма умножения матриц (с квадратным блоком, с прямоугольным блоком).
2. Подберите оптимальное количество создаваемых OpenMP потоков для реализованного блочного алгоритма умножения матриц в зависимости от размера матриц и блоков.
3. 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)
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 Mul-

tiplication // 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.