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)»

Лекция №7
Принципы переноса прикладных программных пакетов на Intel Xeon Phi

Бастраков С.И., Горшков А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

1. Введение

В 2012 году корпорация Intel представила на рынке новый сопроцессор – Intel Xeon Phi, построенный в рамках парадигмы manycore и содержащий 61 вычислительное ядро близкой к x86 архитектуры. В отличие от других активно применяющихся представителей manycore-архитектур, в частности, GPU, Intel сделала акцент не только на пиковой производительности устройства, но и на существенном упрощении процесса создания новых и портирования существующих программных пакетов путем использования стандартных языков и технологий для параллельного программирования.

Так, в ряде случаев можно обойтись без какой-либо переработки существующего кода, добавив ключ компилятора и получив программу, способную выполняться на Xeon Phi. Производительность результата подобного портирования на Xeon Phi зависит от многих факторов и может варьироваться от многократного замедления до многократного ускорения по сравнению с исходной версией на CPU.

Существенное замедление может наблюдаться для приложений, значительная часть (по вкладу в общее время) которых выполняется последовательно, либо задач, не имеющих достаточный ресурс параллелизма для использования большого количества ядер Xeon Phi. Напротив, приложения с большой долей и степенью параллелизма, написанные с учетом векторизации, могут достигать высокой производительности на Xeon Phi без дополнительных усилий. На практике же обычно имеет место некий средний случай – в результате начального портирования приложение на Xeon Phi демонстрирует производительность, сравнимую с версией на CPU, но далекую от пиковой производительности Xeon Phi; таким образом, огромные вычислительные ресурсы Xeon Phi используются далеко не полностью.

В этом случае возникает вопрос, какие усилия требуются для того, чтобы превратить программу, работающую на CPU, в программу, эффективно работающую на Xeon Phi? Этот и другие подобные вопросы, определенно, представляют интерес для научного сообщества. В литературе появляются первые работы, рассказывающие об опыте портирования на Xeon Phi приложений из разных областей.

Анализ позволяет сделать следующий вывод: портирование программ на Xeon Phi может быть выполнено в весьма сжатые сроки (несколько дней) даже при весьма значительных объемах кода. При этом код будет работать достаточно эффективно только в тех случаях, когда приложение уже было оптимизировано для CPU и содержало большой запас внутреннего параллелизма как с точки зрения многопоточности, так и с точки зрения использования инструкций SIMD. Данное условие является необходимым, но не достаточным. Так, многие алгоритмы успешно распараллеливаются на

8-16, но не на 120-240 потоков, допускают эффективное использование SIMD для небольшой длины регистра, упираются в ограниченный на Xeon Phi объем встроенной памяти, требуют вдумчивой реализации с целью активного использования команд Fused Multiply-Add (FMA) и т.д. Все это означает, что получение максимальной производительности требует от программиста определенных усилий.

В данной лекции рассматриваются подходы к портированию и оптимизации на Xeon Phi двух прикладных программных пакетов, осуществляющих решение задач вычислительной физики: моделирование динамики электромагнитного поля методом FDTD (раздел 2) и моделирование переноса излучения методом Монте-Карло (раздел 3). Используются типичные приемы оптимизации, способные привести к выигрышу производительности в рамках multicore- и manycore-архитектур.

2. Подходы к оптимизации программного пакета для моделирования динамики электромагнитного поля методом FDTD

Одним из широко используемых методов вычислительной электродинамики является метод FDTD (Finite-Difference Time-Domain) [1]. Это явный конечно-разностный метод численного решения уравнений Максвелла. Особенностью метода является использование специальной сетки для компонент электромагнитного поля. Точки пространства, соответствующие разным компонентам поля, сдвинуты относительно друг друга на половинные шаги по пространству и времени, благодаря чему все конечно-разностные аппроксимации первых производных являются центральными, и достигается второй порядок точности по времени и пространственным компонентам.

В данном разделе рассматривается портирование на Xeon Phi и оптимизация реализации метода FDTD, созданной на основе программного пакета для моделирования плазмы PICADOR [2]. Для простоты рассматривается базовая версия FDTD. На практике вместе с FDTD часто используется приграничный поглощающий слой PML, оптимизация с его учетом рассматривается в [3].

2.1. Общее описание метода FDTD

Рассматривается трехмерная область в виде прямоугольного параллелепипеда $\{(x.y.z): a_x \le x \le b_x, a_y \le y \le b_y, a_z \le z \le b_z\}$, которую в дальнейшем будем называть расчетной областью. В каждый момент времени t в каждой точке расчетной области (x, y, z) определены 3-компонентное

электрическое поле, которое будем обозначать $\boldsymbol{E}(x,y,z,t) = (E_x(x,y,z,t), E_y(x,y,z,t), E_z(x,y,z,t))$, и 3-компонентное магнитное поле $\boldsymbol{B}(x,y,z,t) = (B_x(x,y,z,t), B_y(x,y,z,t), B_z(x,y,z,t))$. Пара векторных полей $(\boldsymbol{E}, \boldsymbol{B})$ называется электромагнитным полем.

Рассматривается задача моделирования динамики электромагнитного поля от начального момента времени $t_0 = 0$ до заданного конечного момента времени $t = t_{max}$. Динамика электромагнитного поля подчиняется системе уравнений Максвелла (приведена запись двух уравнений для случая вакуума в системе единиц СГС, c – скорость света в вакууме):

$$\begin{cases} \dfrac{\partial \boldsymbol{E}}{\partial t} = \mathrm{c} \cdot \mathrm{rot}\, \boldsymbol{B} \\ \dfrac{\partial \boldsymbol{B}}{\partial t} = -\mathrm{c} \cdot \mathrm{rot}\, \boldsymbol{E} \end{cases}$$

Для численного решения задачи расчетная область покрывается равномерной пространственной сеткой, содержащей $n_x + 1$, $n_y + 1$. $n_z + 1$ узлов (n_x, n_y. n_z ячеек) по соответствующим размерностям. Шаги сетки равны $\Delta x = \frac{b_x - a_x}{n_x}$, $\Delta y = \frac{b_y - a_y}{n_y}$, $\Delta z = \frac{b_z - a_z}{n_z}$. Моделирование по времени происходит с заданным шагом Δt, т.е. в дискретные моменты 0, Δt, $2\Delta t$, $3\Delta t$, … , последовательность завершается при превышении t_{max}. Для индексирования узлов сетки используются естественные трехмерные индексы (i,j,k): $0 \le i \le n_x, 0 \le j \le n_y, 0 \le k \le n_z$.

FDTD использует специальную сетку (сетку Yee [1]) следующего вида. Сеточное значение $E_x(i,j,k)$ соответствует точке физического пространства $\left(a_x + i\Delta x, a_y + (j + 0.5)\Delta y, a_z + (k + 0.5)\Delta z\right)$, $E_y(i,j,k)$ соответствует точке $\left(a_x + (i + 0.5)\Delta x, a_y + j\Delta y, a_z + (k + 0.5)\Delta z\right)$ и $E_z(i,j,k)$ – точке $\left(a_x + (i + 0.5)\Delta x, a_y + (j + 0.5)\Delta y, a_z + k\Delta z\right)$. Сеточные значения компонент магнитного поля $B_x(i,j,k)$, $B_y(i,j,k)$, $B_z(i,j,k)$ сооветствуют точкам $\left(a_x + (i + 0.5)\Delta x, a_y + j\Delta y, a_z + k\Delta z\right)$, $\left(a_x + i\Delta x, a_y + (j + 0.5)\Delta y, a_z + k\Delta z\right)$, $\left(a_x + i\Delta x, a_y + j\Delta y, a_z + (k + 0.5)\Delta z\right)$. Таким образом, разные компоненты поля сдвинуты относительно центра соответствующей ячейки на полшага по одной или двум осям. Кроме того, компоненты магнитного поля сдвинуты относительно компонент электрического поля на полшага вперед по времени.

Моделирование производится итерационно по времени, на каждом шаге хранится текущий набор сеточных значений поля. Начальные значения поля определяются из заданных начальных условий. Итерация метода состоит из двух этапов: использование текущих значений $\boldsymbol{E}$ и $\boldsymbol{B}$ для вычисления новых значений $\boldsymbol{E}$ (обновление $\boldsymbol{E}$), использование текущих значений $\boldsymbol{B}$ и новых значений $\boldsymbol{E}$ для вычисления новых значений $\boldsymbol{B}$ (обновление $\boldsymbol{B}$). В приводимых далее формулах подразумевается, что вычисления произво-

дятся для всех узлов сетки (i, j, k), для которых все члены правой части определены. Обновление $\boldsymbol{E}$ выполняется по следующей схеме:

$$E_x(i,j,k) \coloneqq E_x(i,j,k) + c \cdot \Delta t \cdot \left(\frac{B_z(i,j+1,k) - B_z(i,j,k)}{\Delta y} - \frac{B_y(i,j,k+1) - B_y(i,j,k)}{\Delta z} \right)$$

$$E_y(i,j,k) \coloneqq E_y(i,j,k) - c \cdot \Delta t \cdot \left(\frac{B_z(i+1,j,k) - B_z(i,j,k)}{\Delta x} - \frac{B_x(i,j,k+1) - B_x(i,j,k)}{\Delta z} \right)$$

$$E_z(i,j,k) \coloneqq E_z(i,j,k) + c \cdot \Delta t \cdot \left(\frac{B_y(i+1,j,k) - B_y(i,j,k)}{\Delta x} - \frac{B_x(i,j+1,k) - B_x(i,j,k)}{\Delta y} \right)$$

Обновление $\boldsymbol{B}$ выполняется по следующей схеме:

$$B_x(i,j,k) \coloneqq B_x(i,j,k) - c \cdot \Delta t \cdot \left(\frac{E_z(i,j,k) - E_z(i,j-1,k)}{\Delta y} - \frac{E_y(i,j,k) - E_y(i,j,k-1)}{\Delta z} \right)$$

$$B_y(i,j,k) \coloneqq B_y(i,j,k) + c \cdot \Delta t \cdot \left(\frac{E_z(i,j,k) - E_z(i-1,j,k)}{\Delta x} - \frac{E_x(i,j,k) - E_x(i,j,k-1)}{\Delta z} \right)$$

$$B_z(i,j,k) \coloneqq B_z(i,j,k) - c \cdot \Delta t \cdot \left(\frac{E_y(i,j,k) - E_y(i-1,j,k)}{\Delta x} - \frac{E_x(i,j,k) - E_x(i,j-1,k)}{\Delta y} \right)$$

Очевидно, данные формулы не могут быть использованы для обновления значений $\boldsymbol{E}$ на «правой» границе сетки ($i = n_x$, $j = n_y$ или $k = n_z$), и для обновления значений $\boldsymbol{B}$ на «левой» границе ($i = 0, j = 0$ или $k = 0$). Способ вычисления данных сеточных значений зависит от используемых граничных условий. В данном разделе используются периодические граничные условия по всем осям: $E_x(n_x, j, k) \coloneqq E_x(0, j, k)$, $B_x(0, j, k) \coloneqq B_x(n_x, j, k)$, аналогично для других границ и компонент поля.

2.2. Программная реализация

Рассмотрим базовую программную реализацию метода FDTD. Сеточные значения электрического и магнитного поля будем хранить как динамические 3-мерные массивы из 3-компонентных векторов. Ядро реализации составляют функции обновления сеточных значений $\boldsymbol{E}$ и $\boldsymbol{B}$, являющиеся прямой записью разностной схемы метода FDTD. Приведем код данных функций:

```
struct Double3 {
    double x, y, z;
};

struct Parameters {
    int nx, ny, nz;
    double dx, dy, dz, dt;
};

const double C = 29979245800.0;

void updateE(const Parameters & parameters, Double3 *** b,
Double3 *** e)
{
    const double cx = C * parameters.dt / parameters.dx;
    const double cy = C * parameters.dt / parameters.dy;
    const double cz = C * parameters.dt / parameters.dz;
    #pragma omp parallel for
    for (int i = 0; i < parameters.nx; i++)
    for (int j = 0; j < parameters.ny; j++)
    for (int k = 0; k < parameters.nz; k++)
    {
        e[i][j][k].x += cy * (b[i][j + 1][k].z -
b[i][j][k].z) - cz * (b[i][j][k + 1].y - b[i][j][k].y);
        e[i][j][k].y += cz * (b[i][j][k + 1].x -
b[i][j][k].x) - cx * (b[i + 1][j][k].z - b[i][j][k].z);
        e[i][j][k].z += cx * (b[i + 1][j][k].y -
b[i][j][k].y) - cy * (b[i][j + 1][k].x - b[i][j][k].x);
    }
}

void updateB(const Parameters & parameters, Double3 *** e,
Double3 *** b)
{
    const double cx = C * parameters.dt / parameters.dx;
    const double cy = C * parameters.dt / parameters.dy;
    const double cz = C * parameters.dt / parameters.dz;
    #pragma omp parallel for
    for (int i = 1; i <= parameters.nx; i++)
    for (int j = 1; j <= parameters.ny; j++)
    for (int k = 1; k <= parameters.nz; k++)
    {
        b[i][j][k].x += cz * (e[i][j][k].y - e[i][j][k -
1].y) - cy * (e[i][j][k].z - e[i][j - 1][k].z);
        b[i][j][k].y += cx * (e[i][j][k].z - e[i -
1][j][k].z) - cz * (e[i][j][k].x - e[i][j][k - 1].x);
        b[i][j][k].z += cy * (e[i][j][k].x - e[i][j -
1][k].x) - cx * (e[i][j][k].y - e[i - 1][j][k].y);
    }
}
```

В функциях updateE, updateB из циклов вынесены инварианты $c\Delta t / \Delta x$, $c\Delta t / \Delta y$, $c\Delta t / \Delta z$. Произведено прямолинейное распараллеливание циклов с независимыми итерациями с помощью OpenMP. Данные функции выполняют обновление сеточных значений области во «внутренней области» – без учета граничных условий. Учет граничных условий делается в отдельных функциях для исключения условных операций в основных циклах и упрощения использования других типов граничных условий. Таким образом, основной вычислительный цикл имеет вид:

```
for (int t = 0; t < numSteps; t++)
{
    updateE(parameters, b, e);
    updateBoundaryE(parameters, e);
    updateB(parameters, b, e);
    updateBoundaryB(parameters, b);
}
```

В силу сходства функций updateE, updateB в дальнейшем будем демонстрировать техники оптимизации лишь для updateE, подразумевая выполнение аналогичных преобразований и для updateB. Для анализа производительности будем использовать следующий бенчмарк: моделирование распространения плоской волны на сетке 256x256x256 со 100 итерациями по времени.

Эксперименты проводились на следующей инфраструктуре:

Процессор	2x Intel Xeon Xeon E5-2690 (2.9 GHz, 8 ядер)
Сопроцессор	Intel Xeon Phi 7110X
Память	64 GB
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик, отладчик	Intel C/C++ Compiler 13

Результаты производительности базовой версии на CPU и Xeon Phi в режиме только сопроцессора приведены на рис. 1. CPU-версия неидеально масштабируется с 1 до 4 ядер и демонстрирует замедление при переходе от 4 до 8 ядер. Версия на Xeon Phi немного обгоняет CPU-версию. Во всех запусках на Xeon Phi используется 1 поток на ядро. Данный выбор нетипичен - для большинства приложений рекомендуется использование от 2 до 4 потоков на ядро. Однако для рассматриваемого приложения опытным путем было определено, что оптимальной конфигурацией запуска является 1 поток на ядро. Возможное обоснование состоит в том,

что производительность ограничена в первую очередь пропусккой способностью памяти, а не скоростью выполнения вычислительных операций.

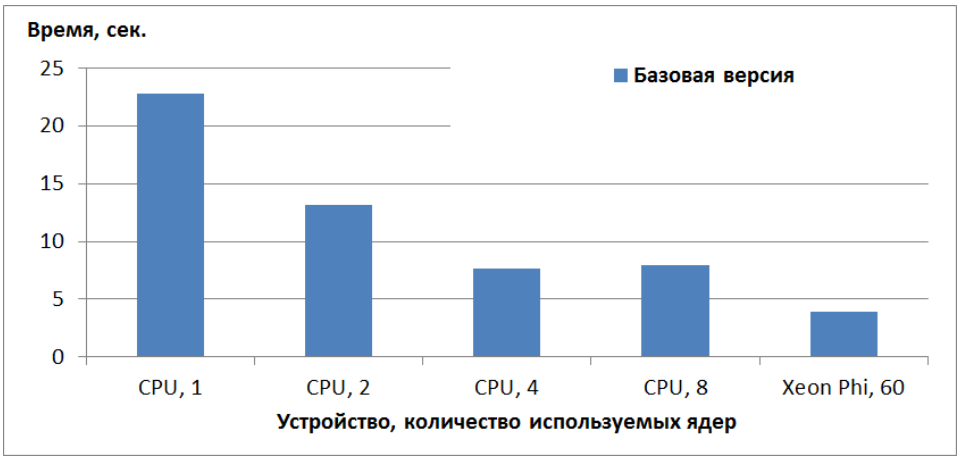

Рис. 1. Время работы базовой версии на CPU и Xeon Phi

2.3. Оптимизация 1: векторизация и изменение структуры данных

Для исследования возможности векторизации кода воспользуемся отчетом о векторизации с помощью опции компилятора –vec-report3. Как видно из отчета, основные циклы (внутренние циклы в updateE, updateB) не векторизуются из-за неподдерживаемой структуры цикла:

```
naive.cpp(24): (col. 5) remark: loop was not vectorized:
unsupported loop structure
```

Причина в том, что в условии цикла используется поле структуры Parameters. Для преодоления данной проблемы скопируем parameters.nz в локальную переменную и будем использовать ее в условии цикла.

После этого структура цикла становится пригодной для векторизации, однако векторизация не происходит из-за потенциальных зависимостей между итерациями (список потенциальных зависимостей зависимостей очень длинный, приведено его начало):

```
naive.cpp(25): (col. 5) remark: loop was not vectorized:
existence of vector dependence
naive.cpp(27): (col. 9) remark: vector dependence: assumed
ANTI dependence between b line 27 and e line 29
naive.cpp(29): (col. 9) remark: vector dependence: assumed
FLOW dependence between e line 29 and b line 27
naive.cpp(27): (col. 9) remark: vector dependence: assumed
ANTI dependence between b line 27 and e line 31
```

```
naive.cpp(31): (col. 9) remark: vector dependence: assumed
FLOW dependence between e line 31 and b line 27
naive.cpp(27): (col. 9) remark: vector dependence: assumed
ANTI dependence between b line 27 and e line 27
```

Действительно, в случае перекрытия массивов в памяти векторизация цикла может быть некорректной, поэтому компилятор не вправе ее осуществить.

Разработчик кода обладает информацией о том, что массивы не пересекаются в памяти и может повлиять на векторизацию цикла с использованием нескольких средств: #pragma ivdep, #pragma simd, ключевое слово restrict, ключ -ansi-alias. Воспользуемся #pragma simd для векторизации внутреннего цикла и соберем программу с ключом -ansi-alias. Кроме того, для повышения производительности сделаем выделение памяти выровненным по 64, используя при выделении массивов сеточных значений функцию _mm_malloc. Функция updateE принимает вид:

```
void updateE(const Parameters & parameters, Double3 *** b,
Double3 *** e)
{
    const double cx = C * parameters.dt / parameters.dx;
    const double cy = C * parameters.dt / parameters.dy;
    const double cz = C * parameters.dt / parameters.dz;
    const int nz = parameters.nz;
    #pragma omp parallel for
    for (int i = 0; i < parameters.nx; i++)
    for (int j = 0; j < parameters.ny; j++)
    #pragma simd
    for (int k = 0; k < nz; k++)
    {
        e[i][j][k].x += cy * (b[i][j + 1][k].z -
b[i][j][k].z) - cz * (b[i][j][k + 1].y - b[i][j][k].y);
        e[i][j][k].y += cz * (b[i][j][k + 1].x -
b[i][j][k].x) - cx * (b[i + 1][j][k].z - b[i][j][k].z);
        e[i][j][k].z += cx * (b[i + 1][j][k].y -
b[i][j][k].y) - cy * (b[i][j + 1][k].x - b[i][j][k].x);
    }
}
```

Результаты производительности данной версии приведены на рис. 2.

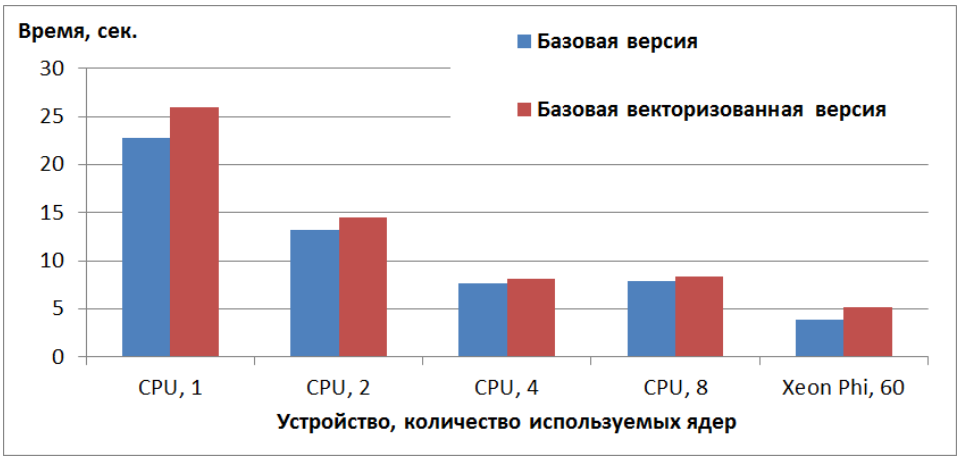

Рис. 2. Время работы базовой векторизованной версии на CPU и Xeon Phi

Как видно из рис. 2, векторизация не только привела к повышению производительности, но даже немного замедлила код. Наиболее вероятная причина данного эффекта состоит в том, что загрузка данных из памяти производится неэффективно и векторизация лишь усиливает расходы на загрузку данных.

Для повышения эффективности загрузки данных из памяти воспользуемся стандартной техникой преобразования массива структур в структуру массивов. Для каждого из полей вместо хранения массива Double3 будем использовать отдельный массив из double для каждой компоненты. Кроме того, перепишем внутренний цикл в терминах работы с одномерными массивами. Код приобретает следующий вид:

```
void updateE(const Parameters & parameters, double *** bx,
double *** by, double *** bz,
    double *** ex, double *** ey, double *** ez)
{
    const double cx = C * parameters.dt / parameters.dx;
    const double cy = C * parameters.dt / parameters.dy;
    const double cz = C * parameters.dt / parameters.dz;
    const int nz = parameters.nz;
    #pragma omp parallel for
    for (int i = 0; i < parameters.nx; i++)
    for (int j = 0; j < parameters.ny; j++)
    {
        double * ex_ij = ex[i][j];
        double * ey_ij = ey[i][j];
        double * ez_ij = ez[i][j];
        const double * bx_ij = bx[i][j];
        const double * bx_ij1 = bx[i][j + 1];
        const double * by_ij = by[i][j];
        const double * by_i1j = by[i + 1][j];
        const double * bz_ij = bz[i][j];
```

```
            const double * bz_i1j = bz[i + 1][j];
            const double * bz_ij1 = bz[i][j + 1];
            #pragma simd
            for (int k = 0; k < nz; k++)
            {
                ex_ij[k] += cy * (bz_ij1[k] - bz_ij[k]) -
                            cz * (by_ij[k + 1] - by_ij[k]);
                ey_ij[k] += cz * (bx_ij[k + 1] - bx_ij[k]) -
                            cx * (bz_i1j[k] - bz_ij[k]);
                ez_ij[k] += cx * (by_i1j[k] - by_ij[k]) -
                            cy * (bx_ij1[k] - bx_ij[k]);
            }
        }
}
```

Результаты версии с учетом данных изменений приведены на рис. 3. Векторизация дает почти двукратное ускорение на Xeon Phi и небольшое ускорение на CPU. Данные ускорения весьма далеки от теоретически максимально возможных: на Xeon Phi векторные операции могут осуществляться с 8 значениями типа double, а на используемом CPU – с 4. Это объясняется тем, что задача в первую очередь ограничена доступом к памяти. С другой стороны, именно большая пропускная способность памяти главным образом обеспечивает превосходство Xeon Phi над CPU.

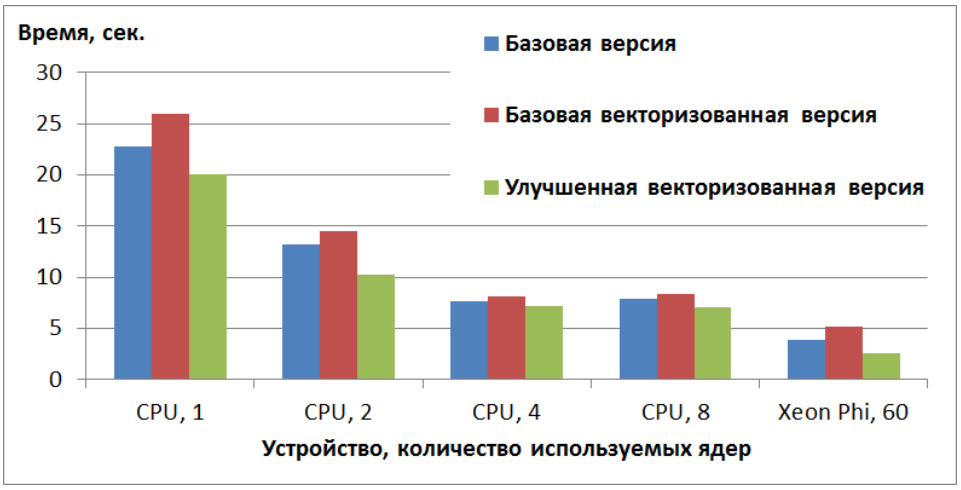

Рис. 3. Время работы улучшенной векторизованной версии на CPU и Xeon Phi

2.4. Оптимизация 2: улучшение масштабируемости

Помимо векторизации, производительность на Xeon Phi существенно зависит от эффективности масштабируемости приложения. В предыдущих версиях с помощью технологии OpenMP распараллеливался внешний цикл. Количество его итераций слишком мало для эффективного использования

Xeon Phi: на рассматриваемом бенчмарке имеется 257 итераций, в зависимости от конфигурации запуска количество потоков составляет от 60 до 240. Таким образом, при некоторых конфигурациях запуска большинство потоков делает лишь одну итерацию внешнего цикла, и малое количество потоков делает две итерации, в то время как остальные простаивают. Параллелизм в данной задаче чрезмерно крупнозернистый (coarse grained) для Xeon Phi. Данная проблема также рассмотрена в [3].

Используем стандартную технику уменьшения зернистости параллелизма - объединим два внешних цикла в один и будем распараллеливать его. Тогда на рассматриваемом бенчмарке число итераций будет уже достаточно велико. В приводимом ниже коде объединение циклов произведено вручную, его также можно делать автоматически с помощью #pragma omp collapse (2).

```
void updateE(const Parameters & parameters, double *** bx,
double *** by, double *** bz,
    double *** ex, double *** ey, double *** ez)
{
    const double cx = C * parameters.dt / parameters.dx;
    const double cy = C * parameters.dt / parameters.dy;
    const double cz = C * parameters.dt / parameters.dz;
    const int nz = parameters.nz;
    const int numIterations = parameters.nx *
parameters.ny;
    #pragma omp parallel for
    for (int iteration = 0; iteration < numIterations;
iteration++)
    {
        int i = iteration / parameters.ny;
        int j = iteration % parameters.ny;
        double * ex_ij = ex[i][j];
        double * ey_ij = ey[i][j];
        double * ez_ij = ez[i][j];
        const double * bx_ij = bx[i][j];
        const double * bx_ij1 = bx[i][j + 1];
        const double * by_ij = by[i][j];
        const double * by_i1j = by[i + 1][j];
        const double * bz_ij = bz[i][j];
        const double * bz_i1j = bz[i + 1][j];
        const double * bz_ij1 = bz[i][j + 1];
        #pragma simd
        for (int k = 0; k < nz; k++)
        {
            ex_ij[k] += cy * (bz_ij1[k] - bz_ij[k]) -
                        cz * (by_ij[k + 1] - by_ij[k]);
            ey_ij[k] += cz * (bx_ij[k + 1] - bx_ij[k]) -
                        cx * (bz_i1j[k] - bz_ij[k]);
            ez_ij[k] += cx * (by_i1j[k] - by_ij[k]) -
                        cy * (bx_ij1[k] - bx_ij[k]);
```

```
            }
        }
}
```

Этот подход позволяет немного увеличить производительность на 8 ядрах CPU и Xeon Phi, данные приведены на рис. 4.

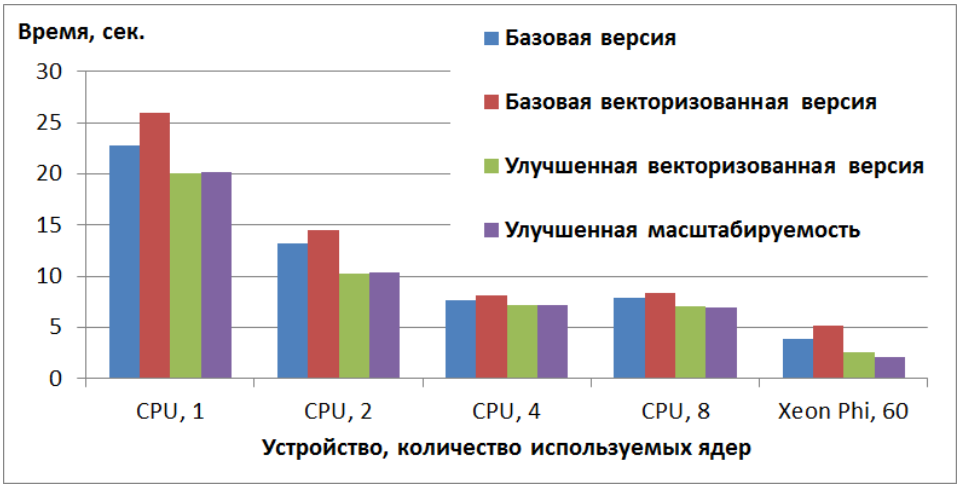

Рис. 4. Время работы версии с улучшенной масштабируемостью на CPU и Xeon Phi

2.5. Общие результаты оптимизации

В результате проделанных оптимизаций удалось повысить производительность программы для Intel Xeon Phi вдвое, при этом производительность на CPU также увеличилась на 10-30% (в зависимости от количества используемых ядер). При этом лучшая версия на Xeon Phi обгоняет лучшую версию на CPU примерно в 3.3 раза.

Как уже отмечалось, производительность реализации метода FDTD ограничена в первую очередь пропускной способностью памяти. Оценим пропускную способность памяти, достигаемую рассмотренными реализациями. В рассматриваемом бенчмарке используется сетка 256x256x256 и 100 итераций по времени. На каждой итерации выполняется две эквивалентных с точки зрения доступа к памяти операции. Для каждой из операций происходит 15 обращений к памяти (выполнение += приводит к чтению и записи и, поэтому, считается за 2 операции). Таким образом, всего обрабатывается $256 * 256 * 256 * 100 * 2 * 15 \approx 5.03 * 10^{10}$ элементов типа double, что составляет 375 ГБ данных. Разделив объем обрабатываемой информации на время выполнения бенчмарка, получаем оценку достигаемой пропускной способности памяти, данные представлены на рис. 5.

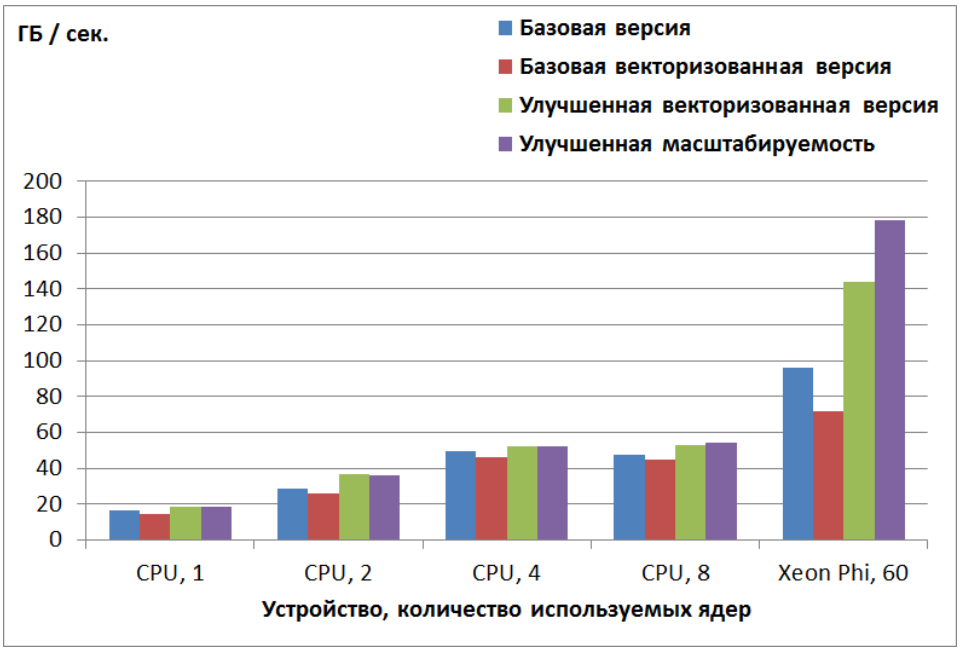

Рис. 5. Достигнутая пропускная способность памяти на CPU и Xeon Phi

При этом пиковая пропускная способность памяти используемой модели Xeon Phi составляет 352 ГБ / сек., а на каждом из используемых CPU – 53 ГБ / сек. Таким образом, на обоих устройствах достигается примерно половина от пиковой пропускной способности памяти (в запусках на CPU половина потоков работает на одном CPU, а половина – на втором). Большое преимущество Xeon Phi в пропускной способности памяти является основной причиной значительного превосходства Xeon Phi над CPU в данной задаче. Достигнутый результат в 178 ГБ / сек. на Xeon Phi очень близок к результату на бенчмарке для измерения пропускной способности памяти STREAM, представленному в статье http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad, данная статья также использует конфигурацию запуска 1 поток на ядро.

Авторы выражают благодарность студенту ВМК ННГУ А. Ларину за помощь в проверке некоторых идей по оптимизации.

3. Подходы к оптимизации программного пакета для Монте-Карло моделирования переноса излучения

В настоящее время в медицинских исследованиях, в том числе предклинических, существует потребность в развитии новых неинвазивных и доступных методов диагностики, поскольку используемые традиционные методы (магнитно-резонансная томография, компьютерная томография, позитрон-

но-эмиссионная томография) имеют ряд ограничений связанных с их небезопасностью, высокими требованиями к инфраструктуре и стоимостью оборудования. Классом наиболее перспективных методов диагностики, которые могут применяться как в сочетании с существующими методами, так и в некоторых случаях вместо них, являются оптические методы. Их основными преимуществами являются безопасность для пациента, сравнительно невысокая стоимость приборов и широкие функциональные возможности, обусловленные возможной вариативностью параметров зондирующего излучения (длина волны, модуляция, длина импульса и т.д.).

Для диагностики биотканей на больших глубинах необходимо применять методы, для которых информативным является многократно рассеянное излучение. Одним из таких методов является оптическая диффузионная спектроскопия (ОДС), предоставляющая широкие возможности для неинвазивной диагностики. Метод основан на регистрации многократно рассеянного объектом зондирующего излучения на нескольких длинах волн, определяемых спектрами поглощения исследуемых компонент организма.

Применение метода оптической диффузионной спектроскопии позволяет решать такие задачи, как диагностика и лечение раковых опухолей, в частности, рака груди; мониторинг активности зон коры головного мозга; планирование фотодинамической терапии; мониторинг состояния пациента при хирургическом вмешательстве; определение состояния кожных покровов; и др.

Для успешного применения метода ОДС на практике необходимо выполнять подбор параметров этого метода (таких как взаимное расположение источника и детекторов, длина волны зондирования и др.) путем проведения предварительного моделирования распространения зондирующего излучения в исследуемых биологических тканях. Алгоритм решения этой задачи обсуждается в рамках данного раздела.

3.1. Общее описание алгоритма Монте-Карло моделирования переноса излучения

Рассматривается объект в трехмерном пространстве, состоящий из набора слоев. Каждый слой описывает определенный тип биологической ткани, обладающей набором оптических характеристик. Оптические характеристики постоянны в рамках слоя. Например, если моделировать перенос излучения в голове человека, то в ней можно выделить такие слои, как кожа головы, жировая ткань, череп, цереброспинальная жидкость, серое и белое вещество головного мозга (рис. 6).

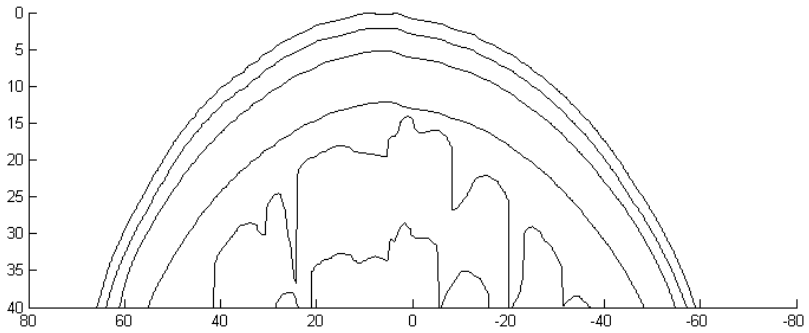

Рис. 6. Двумерное сечение слоев головы человека

Каждый слой помимо оптических параметров характеризуется набором границ. Границы слоя описываются в виде одной или нескольких поверхностей в трехмерном пространстве. Каждая из поверхностей состоит из набора треугольников.

Источник излучения представляет собой бесконечно тонкий луч фотонов и описывается направлением и положением в трехмерном пространстве.

Для решения задач ОДС было введено в рассмотрение понятие детектора как некоторой замкнутой области на поверхности исследуемого объекта, которая способна улавливать проходящие через нее фотоны [4].

Идея метода Монте-Карло в данной задаче состоит в случайной трассировке набора фотонов в биоткани. Фотоны объединяются в пакеты, каждый пакет обладает весом. Далее понятия «фотон» и «пакет фотонов» будут отождествляться. Начинает движение пакет фотонов от источника излучения. Далее на каждом шаге трассировки случайным образом определяется его направление и величина смещения, определяется поглощенный вес. Моделирование пакета завершается либо при его поглощении средой (когда вес пакета становится меньше минимального), либо если он вылетает за границы исследуемого объекта (рис. 7).

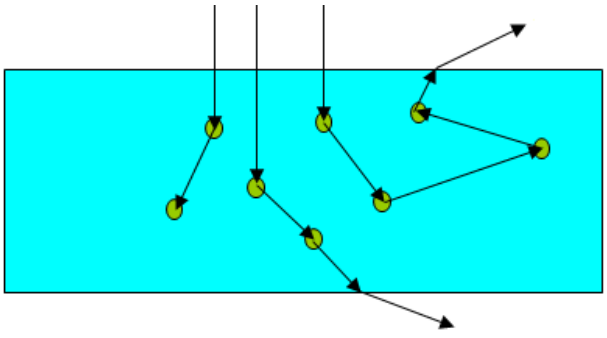

Рис. 7. Трассировка фотонов в биоткани

Так как рассматриваются обычно многослойные биоткани, на каждом шаге трассировки фотона необходимо дополнительно проверять, не пересекает ли траектория его движения границу текущего слоя. Для этого может использоваться, например, алгоритм, основанный на переборе всех треугольников, из которых строится поверхность границы, и поиске пересечения с каждым из этих треугольников. Однако более эффективным подходом здесь будет поиск пересечений с помощью BVH деревьев [5].

Особенностью данного алгоритма является полная независимость процесса трассировки различных фотонов друг от друга. Соответственно, каждый параллельный поток может выполнять трассировку своего набора фотонов, и теоритически алгоритм является идеально распараллеливаемым.

Результатами моделирования являются:

- интенсивность рассеянного назад излучения на детекторах (сигнал на детекторах);
- фотонные карты траекторий для каждого детектора (для фотонов, попавших из источника на детектор);
- общая карта траекторий (для всех фотонов).

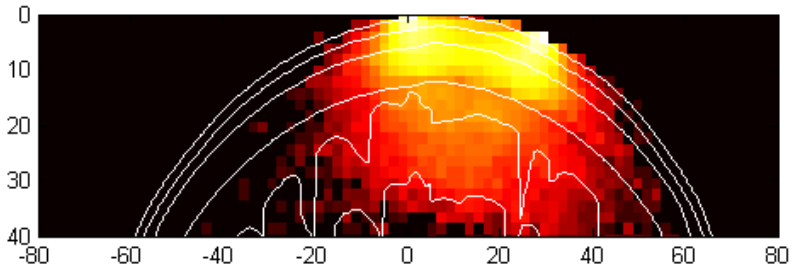

Рис. 8. Двумерное сечение фотонной карты траекторий для детектора, расположенного на расстоянии 30 мм от источника излучения (источник находится в начале координат, детектор – справа от него)

На рис. 8 показано двумерное сечение фотонной карты траекторий. Рассматриваются траектории фотонов, попавших из источника излучения на детектор. Источник находится в начале координат, детектор – справа от него на расстоянии 30 мм. Цветовая шкала используется для отображения частоты попадания фотонов в определенную подобласть. Чем светлее цвет – тем больше фотонов попало в данную подобласть. Цветовая шкала (белый – желтый – красный – черный) является логарифмической.

Для хранения фотонных карт используется равномерная трехмерная сетка, в ячейках которой содержится число посещений фотоном данной подобласти пространства (рис. 9).

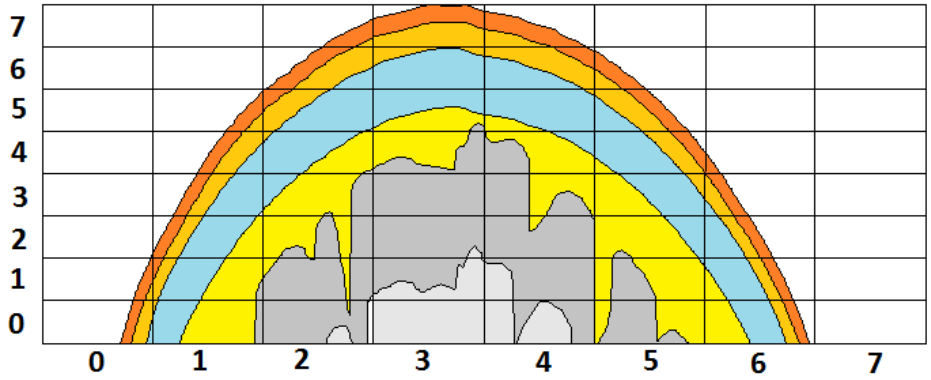

Рис. 9. Сетка для хранения информации о фотонных картах траекторий

3.2. Прямой перенос программного пакета на сопроцессор

Кратко опишем особенности исходной версии программного пакета.

Распараллеливание ведется по фотонам, каждый поток обсчитывает свой набор траекторий. Фотоны делятся поровну между потоками:

```
void LaunchOMP(InputInfo* input, OutputInfo* output,
    MCG59* randomGenerator, int numThreads)
{
    omp_set_num_threads(numThreads);

    …

    #pragma omp parallel
    {
        int threadId = omp_get_thread_num();

        for (uint64 i = threadId;
            i < input->numberOfPhotons; i += numThreads)
        {
            ComputePhoton(specularReflectance, input,
                &(threadOutputs[threadId]),
                &(randomGenerator[threadId]), trajectory);
        }
    }
    …
}
```

Работа ведется с числами двойной точности.

Результатами трассировки являются:

- Величина сигнала на детекторе – массив, количество элементов которого равно числу детекторов, размер для 10 детекторов – 80 Б.

- Общая фотонная карта траекторий – массив, количество элементов которого равно числу ячеек сетки. Для хранения результатов используется трехмерная сетка, сохраняются данные о количестве посещений фотонами каждой из ее ячеек (рис. 10). Если рассматривать сетку размером 100*100*50 элементов, размер ее будет равен 4 МБ.
- Фотонные карты траекторий для каждого детектора. Фотонная карта для каждого детектора хранится в массиве, аналогичном массиву с общей фотонной картой. Размер результатов для 10 детекторов – 40 МБ.

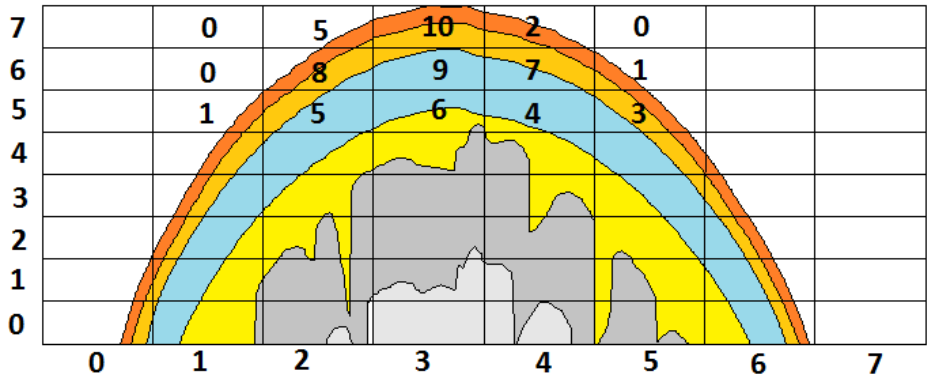

Рис. 10. Хранение числа посещений фотоном ячеек сетки в самой сетке

Каждому фотону требуется доступ к массивам результатов для их обновления, а значит необходимо либо использовать синхронный доступ к данным на запись, либо воспользоваться техникой дублирования данных.

В исходной версии программы применяется второй подход: создаются копии результирующих массивов для каждого потока исполнения, а после окончания расчетов данные этих копий суммируются. Применение данной техники позволяет, с одной стороны, избежать затрат на синхронизацию, но, с другой, приводит к дополнительным затратам памяти.

И если для исполнения на центральном процессоре объем дополнительной памяти будет равен 350 МБ при использовании 8 потоков, то для работы на сопроцессоре Intel Xeon Phi с 240 потоками понадобится хранить уже более 10 ГБ данных.

Еще одна особенность алгоритма состоит в необходимости хранения траектории каждого отдельного фотона в процессе его трассировки. Это связано с тем, что до окончания трассировки фотона узнать, попал ли он в детектор, нельзя. Соответственно, если фотон попал в детектор, траектория

его движения суммируется с общей картой траекторий для данного детектора, а если нет – игнорируется.

В исходной версии выделяется массив для хранения траектории для каждого потока. С точки зрения структур данных используется все та же трехмерная сетка, соответственно размер массива равен 4 МБ. Перед началом трассировки каждого отдельного фотона этот массив обнуляется.

Отметим, что для переноса используется режим работы только на сопроцессоре. Выбор этого режима обусловлен отсутствием необходимости модифицировать код для его запуска на Intel MIC. А значит мы можем выполнять оптимизацию и отладку одного и того же кода параллельно на CPU и на сопроцессоре.

Тестовые запуски проводились на следующем оборудовании: CPU Intel Xeon E5-2690 2.9 ГГц (8 ядер), Intel Xeon Phi 7110X, 64 ГБ RAM, использовался компилятор Intel C/C++ Compiler 14.0, операционная система – CentOS 6.2.

Во всех экспериментах расчеты выполнялись с двойной точностью.

Результаты прямого переноса приведены на рис. 11.

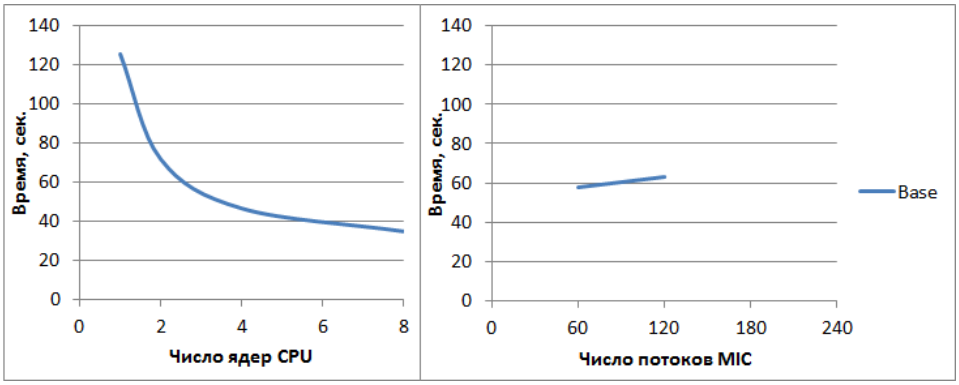

Рис. 11. Время работы исходной версии программы на CPU и Intel Xeon Phi

Прямой перенос базовой версии программы позволяет получить производительность на сопроцессоре, вдвое меньшую, чем производительность одного 8-ми ядерного CPU. Запуск программы в 240 потоков невозможен в силу недостатка памяти на сопроцессоре.

3.3. Оптимизация 1: обновление структуры данных для хранения траектории фотона в процессе трассировки

Перед началом оптимизации имеет смысл выявить наиболее медленные участки программы. Для этого воспользуемся профилировщиком Intel VTune Amplifier XE. Результаты профилировки приведены на рис. 12.

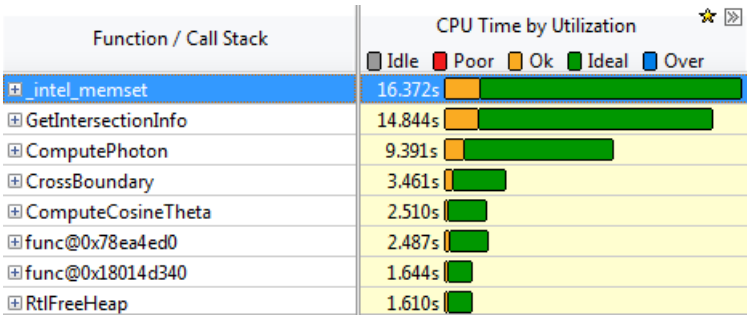

Рис. 12. Результаты профилировки базовой версии программы

Как видно из графиков, наибольшее время занимает функция memset(), которая используется в момент начала процесса трассировки фотона для обнуления памяти, предназначенной для хранения траектории его движения. Напомним, что текущая траектория движения фотона хранится в виде трехмерной сетки (рис. 10).

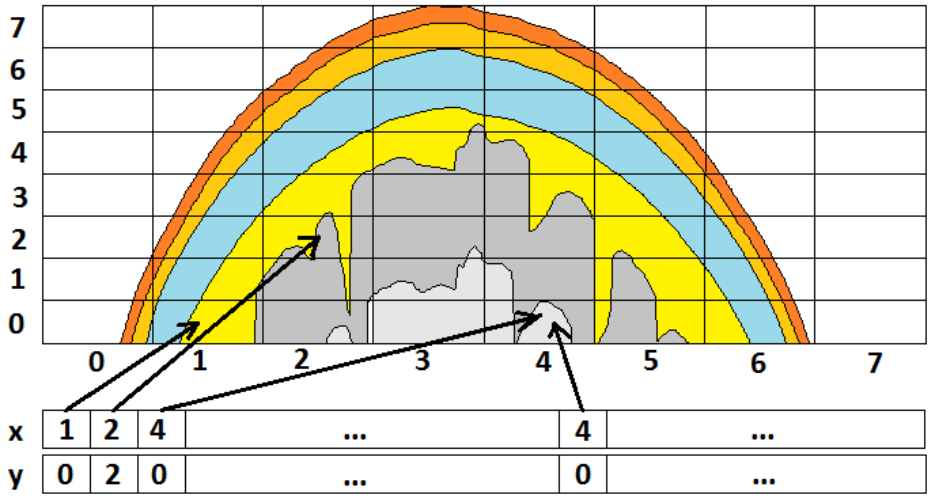

Рис. 13. Хранение числа посещений фотоном ячеек сетки в виде списка координат этих ячеек

Кроме необходимости обнуления памяти на каждой итерации, данный подход обладает еще несколькими недостатками. А именно, размер массива слишком велик по сравнению с размером L2 кэш памяти, и доступ к элементам массива осуществляется в случайном порядке (в силу случайности траектории фотона).

Для устранения описанных выше недостатков предлагается использовать другую структуру данных для хранения траектории фотона: следует хра-

нить не число посещений данной ячейки для всей сетки, а список координат посещенных ячеек (рис. 13).

Экспериментальные данные показывают, что максимальная длина траектории фотона в данном примере не превосходит 2048 шагов (эта величина существенно зависит от параметров биотканей и размера сетки). Размер структуры данных в этом случае составляет всего 6 КБ.

Результаты применения этой оптимизации приведены на рис. 14.

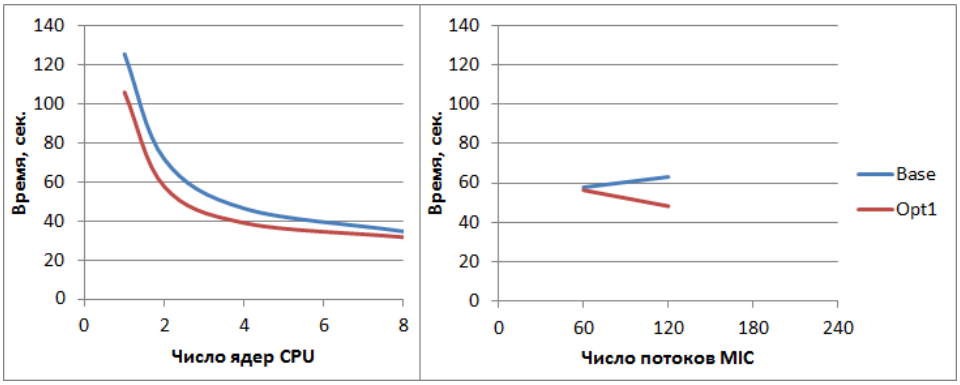

Рис. 14. Время работы программы после оптимизации структуры данных для хранения траектории движения фотона на CPU и Intel Xeon Phi

Изменение используемой структуры данных позволило сократить время вычислений на CPU в 1 поток на 18%, а в 8 – на 9%. Время работы программы на MIC в 120 потоков сократилось на 31%.

3.4. Оптимизация 2: отказ от использования дублирующих массивов

Основной недостаток текущей версии программы состоит в том, что мы не можем использовать все возможности сопроцессора – число потоков, на которых мы можем запустить программу, ограничено объемом памяти сопроцессора. И если при работе на CPU использование дублирующих массивов – вполне нормальная и часто применяемая практика, то при переходе на Intel Xeon Phi эта техника не всегда себя оправдывает.

В нашем случае дублирование выполнено для двух типов результатов: общей фотонной карты траекторий и фотонных карт для каждого из детекторов. Объем операций доступа к этим массивам существенно различен. Общая фотонная карта траекторий обновляется каждым фотоном, в то время как фотонные карты детекторов обновляются реже, так как далеко не каждый фотон попадает в тот или иной детектор. С другой стороны, суммарный размер фотонных карт для всех детекторов на порядок превосходит размер общей фотонной карты (в примере это 40 и 4 МБ соответственно).

Отказ от дублирования общей фотонной карты с одной стороны приводит к существенным затратам на синхронизацию, а с другой – выигрыш по памяти в этом случае не так велик.

Таким образом, наиболее эффективным выглядит отказ от дублирующих массивов только для фотонных карт детекторов. В этом случае существенно уменьшается размер дополнительно используемой памяти, а затраты на синхронизацию будут не так велики. Вероятность, что в одно и то же время фотоны из разных потоков попадут в детекторы, невелика.

Результаты применения этой оптимизации приведены на рис. 15.

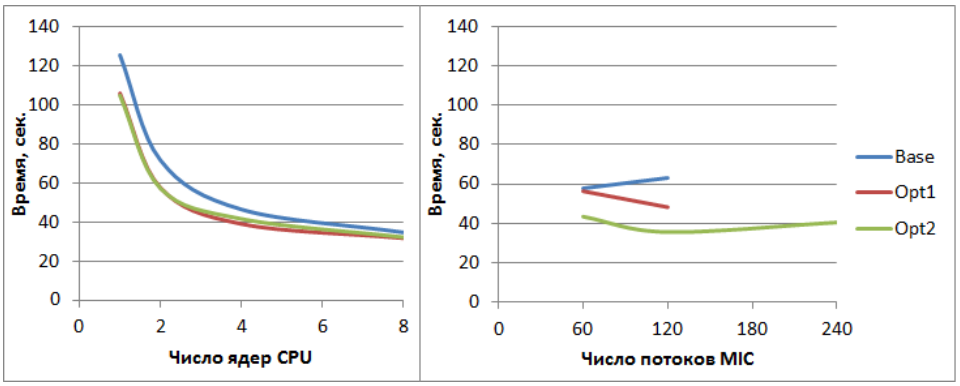

Рис. 15. Время работы программы после отказа от использования дублирующих массивов на CPU и Intel Xeon Phi

Время работы программы на центральном процессоре после применения оптимизации практически не изменилось, а на сопроцессоре в 120 потоков сократилось еще на 35%. Отметим, что использование 240 потоков не дает ожидаемого повышения производительности.

3.5. Оптимизация 3: балансировка нагрузки

Профилировка программы с использованием Intel VTune Amplifier XE выявила еще одну особенность используемого алгоритма – разное время выполнения отдельных потоков (рис. 16).

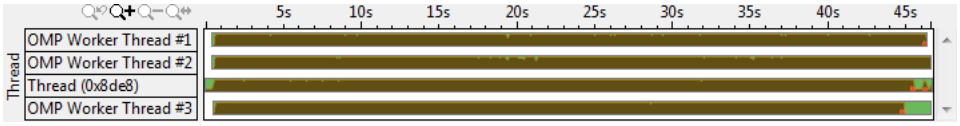

Рис. 16. Диаграмма исполнения потоков на CPU (до балансировки нагрузки)

В текущей версии программы используется статическая схема балансировки нагрузки, при которой каждый поток проводит трассировку примерно одинакового количества фотонов.

Более эффективным будет использование динамической схемы балансировки нагрузки. Динамическая балансировка осуществляется средствами библиотеки OpenMP:

```
void LaunchOMP(InputInfo* input, OutputInfo* output, MCG59*
randomGenerator, int numThreads)
{
    …
    #pragma omp parallel for schedule(dynamic)
    for (uint64 i = 0; i < input->numberOfPhotons; ++i)
    {
        int threadId = omp_get_thread_num();
        ComputePhoton(specularReflectance, input,
            &(threadOutputs[threadId]),
            &(randomGenerator[threadId]),
            &(trajectory[threadId]));
    }
    …
}
```

Диаграмма выполнения потоков после оптимизации приведена на рис. 17.

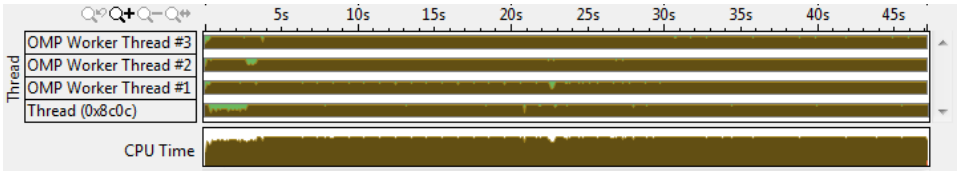

Рис. 17. Диаграмма исполнения потоков на CPU (после балансировки нагрузки)

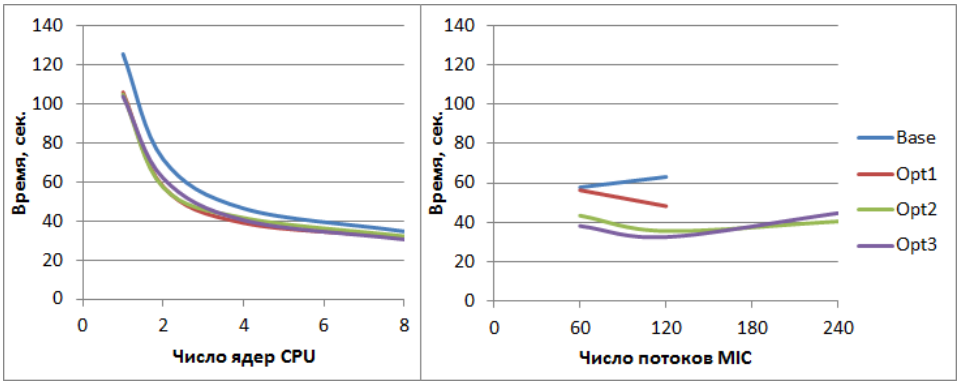

Рис. 18. Время работы программы после выполнения динамической балансировки нагрузки на CPU и Intel Xeon Phi

Время работы программы на CPU после проведенной оптимизации незначительно уменьшилось, а на сопроцессоре уменьшилось еще на 9% до 32 секунд.

3.6. Общие результаты оптимизации

В результате проделанных оптимизаций удалось повысить производительность программы для Intel Xeon Phi вдвое. Причем минимальное время работы достигается за счет использования 120 потоков сопроцессора. Такой результат, а так же невысокая эффективность распараллеливания программы, объясняется достаточно большим количеством операций чтения/записи данных в алгоритме, а также наличием синхронизации в оптимизированных версиях.

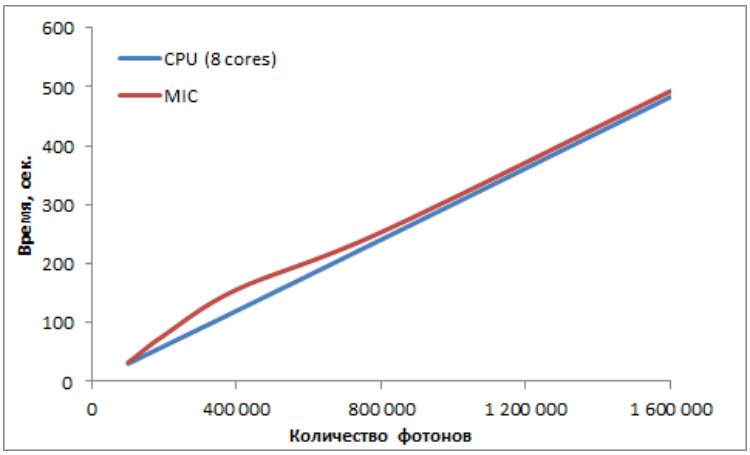

Рис. 19. Сравнение времени работы программы для моделирования переноса излучения на CPU и Intel MIC

Сравнение времени работы лучших версий программы для процессора и сопроцессора при различном числе трассируемых фотонов продемонстрировано на рис. 19.

Приведенные результаты показывают линейную сложность алгоритма в зависимости от количества трассируемых фотонов, время работы программы на сопроцессоре Intel Xeon Phi примерно соответствует времени ее работы на CPU.

Отметим, что продемонстрированные в работе техники оптимизации не являются исчерпывающими для данной задачи, в частности, не описана векторизация кода, применение которой является одним из важнейших способов повысить эффективность программы на Intel Xeon Phi.

Еще одна особенность программы – применение алгоритма поиска пересечения фотона с границами слоя на каждом шаге моделирования. Как показывают результаты профилировки (рис.), это один из самых трудоемких этапов алгоритма моделирования, а значит, необходим анализ существующих алгоритмов поиска пересечений с целью выбора наиболее подходящего для работы на сопроцессоре, и его дальнейшая оптимизация.

Отдельного обсуждения заслуживает и выбор модели программирования на сопроцессоре, которую следует использовать для переноса. Режим «исполнения только на сопроцессоре» оптимален на начальном этапе, когда оптимизация проводится одновременно на CPU и на Intel Xeon Phi. Однако для более эффективного использования сопроцессора имеет смысл использовать один из гетерогенных (CPU + MIC) режимов: offload или симметричный.

И, наконец, не стоит пренебрегать такими стандартными подходами к оптимизации кода на Intel Xeon Phi, как работа с выровненными данными, использование команд программной предвыборки данных и др.

4. Заключение

В данной лекции были рассмотрены принципы переноса прикладных программных пакетов на Intel Xeon Phi на примере двух приложений из области вычислительной физики: моделирования динамики электромагнитного поля методом FDTD и моделирования переноса излучения методом Монте-Карло. Для обоих приложений были рассмотрены некоторые типичные проблемы, связанные с недостаточно эффективным использованием Xeon Phi, и стандартные подходы к их решению.

В задаче моделирования динамики электромагнитного поля методом FDTD была произведена векторизация кода и перепаковка данных для обеспечения более эффективного паттерна обращения к памяти. Также была продемонстрирована стандартная техника снижения гранулярности параллелизма путем объединения циклов. В результате проделанных оптимизаций удалось повысить производительность программы для Intel Xeon Phi вдвое по сравнению с исходной версией, при этом производительность на CPU также увеличилась в среднем на 20%. При этом лучшая версия на Xeon Phi обгоняет лучшую версию на CPU примерно в 3.3 раза. Реализации на CPU и Xeon Phi достигаются примерно половину пиковой пропускной способности памяти – 54 ГБ / сек. и 178 ГБ / сек. соответственно.

В задаче моделирования переноса излучения методом Монте-Карло было произведено изменение структуры данных для хранения траектории фотона, отказ от использования дублирующих массивов и балансировка нагрузки. В результате проделанных оптимизаций удалось повысить производительность программы для Intel Xeon Phi вдвое по сравнению с начальной версией.

Литература

Использованные источники информации

1. Taflove A. Computational Electrodynamics: The Finite-Difference Time-Domain Method. -Artech House, London, 1995.
2. Bastrakov S., Donhenko R., Gonoskov A., Emenko E., Malyshev A., Meyerov I., Surmin I. Partile-in-cell plasma simulation on heterogeneous cluster systems // Journal of Computational Science. -2012. –V. 3. –P. 474-479.
3. Zhou S., Tan G. FDTD Algorithm Optimization on Intel Xeon Phi coprocessor. http://software.intel.com/sites/default/files/article/373519/fdtd-optimization-on-mic.pdf
4. Gorshkov A.V., Kirillin M.Yu. Monte Carlo simulation of brain sensing by optical diffuse spectroscopy // Journal of Computational Science. -2012. -Vol. 3, No. 6. -P. 498-503.
5. Горшков А.В., Коршунова А.Л. Оптимальный алгоритм поиска пересечений в задаче Монте-Карло моделирования распространения зондирующего излучения в головном мозге человека // Вестник нижегородского университета им. Н.И. Лобачевского. -2012. –Т. 5, Ч. 2. –С. 73-80.

Дополнительная литература

6. Intel Xeon Phi Coprocessor System Software Developers Guide, revision 2.03, 2012
7. Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture, Volume 1a, January 29, 2013
8. Jeffers J., Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013. -432 p.

Информационные ресурсы сети Интернет

9. Intel Developer Zone [http://software.intel.com/en-us/mic-developer]