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 №6
Оптимизация вычислительно трудоемкого программного модуля для архитектуры Intel Xeon Phi. Метод Монте-Карло

Горшков А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

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, позитронно-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 данной лабораторной работы.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – обозначить основные направления и описать техники оптимизации алгоритма моделирования

распространения излучения в сложных биологических тканях методом Монте-Карло для эффективного использования сопроцессоров Intel Xeon Phi.

Данная цель предполагает решение следующих основных задач:

1. Изучение базовых принципов и особенностей алгоритма моделирования переноса излучения.
2. Прямой перенос алгоритма на сопроцессор Intel Xeon Phi.
3. Выявление «узких мест» в алгоритме с использованием соответствующих инструментов профилировки.
4. Выполнение оптимизации алгоритма с последующей проверкой результатов его производительности.

1.2. Структура работы

Работа построена следующим образом: дан краткий обзор алгоритма моделирования распространения излучения в сложных биологических тканях, описана базовая программная реализация этого алгоритма. Описаны особенности распараллеливания и структуры данных, используемые в базовой версии программы. Проведен анализ эффективности базовой версии с помощью профилировщика Intel VTune Amplifier, выявлены направления ее оптимизации. Описаны техники оптимизации, даны результаты их применения.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	Intel 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 14

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий:

1. Дать студентам краткое описание алгоритма моделирования распространения излучения в сложных биологических тканях.
2. Рассмотреть прилагаемый к лабораторной работе программный код базовой реализации алгоритма, описать метод проверки корректности результатов. Описать способ распараллеливания и структуры данных базовой версии алгоритма.
3. Провести анализ эффективности предложенной реализации, выделить направления для оптимизации.
4. Последовательно описать каждую из трех техник оптимизации в данной задаче с демонстрацией полученных результатов.
5. Сформулировать выводы, дать задания для самостоятельной работы.

2. Алгоритм Монте-Карло моделирования переноса излучения

2.1. Общее описание задачи

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. 1).

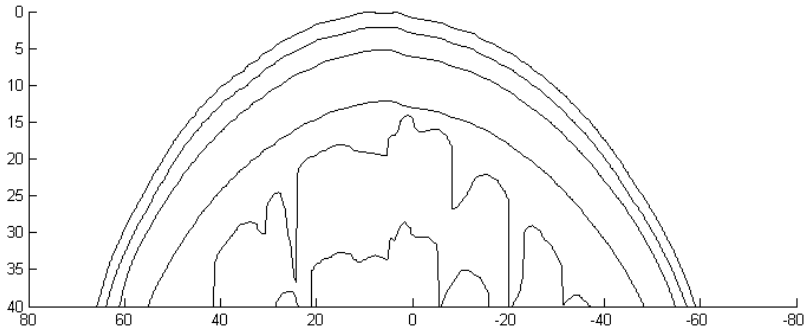

Рис. 1. Двумерное сечение слоев головы человека

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. В данной лабораторной работе используются прямоугольные детекторы.

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. 2).

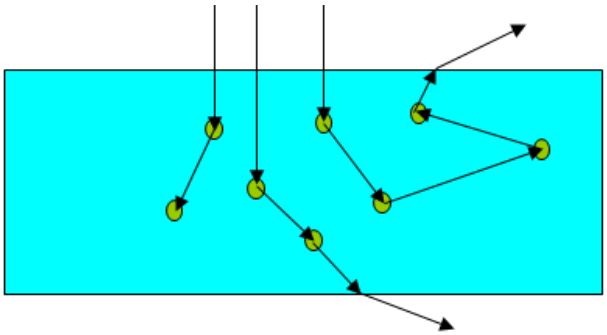

Рис. 2. Трассировка фотонов в биоткани

Так как рассматриваются обычно многослойные биоткани, на каждом шаге трассировки фотона необходимо дополнительно проверять, не пересекает ли траектория его движения границу текущего слоя. В данной лабораторной работе для этого используется алгоритм, основанный на переборе всех треугольников, из которых строится поверхность границы, и поиске пересечения с каждым из этих треугольников. Более эффективным подходом здесь будет поиск пересечений с помощью BVH деревьев.

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).

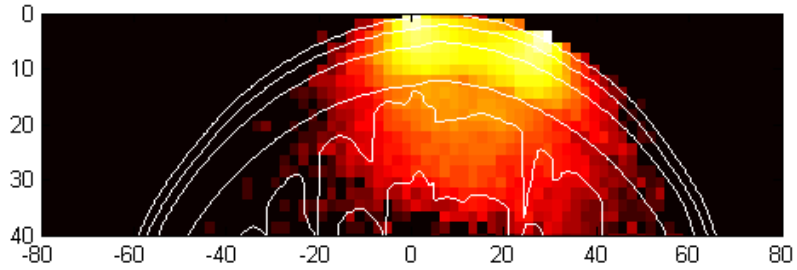

Рис. 3. Двумерное сечение фотонной карты траекторий для детектора, расположенного на расстоянии 30 мм от источника излучения (источник находится в начале координат, детектор – справа от него)

На рис. 3 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 которых содержится число посещений фотоном данной подобласти пространства (рис. 4).

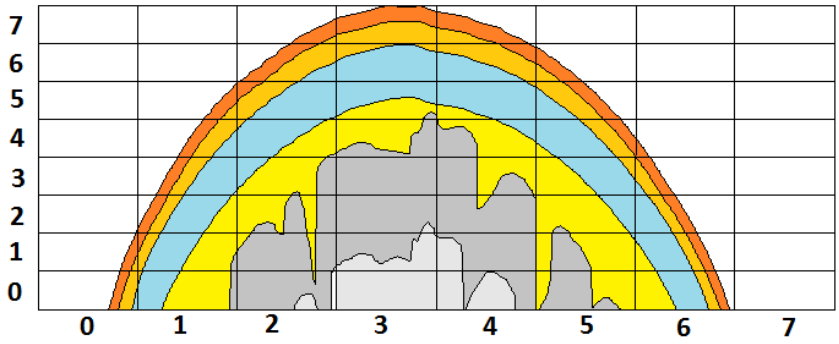

Рис. 4. Сетка для хранения информации о фотонных картах траекторий

2.2. Общее описание базовой версии программы

Базовая версия алгоритма, используемая в данной лабораторной работе, состоит из трех компонентов:

- Библиотеки для чтения XML файлов TinyXML [http://www.grinninglizard.com/tinyxml/];
- Библиотеки, содержащей набор функций для трассировки фотона в среде (xmcml);
- Исполняемого модуля, в задачи которого входит чтение параметров задачи, параллельный запуск процесса моделирования и запись результатов работы программы в файл (xmcmlLauncher).

Для компиляции программы в операционных системах семейства Windows следует воспользоваться проектом Visual Studio 2010. При работе на ОС Linux компиляция осуществляется скриптами *cpu_make.sh* (компиляция для CPU) и *mic_make.sh* (компиляция для Intel Xeon Phi в режиме работы только на сопроцессоре).

Входными файлами программы являются:

- XML файл с описанием входных параметров задачи. В качестве примера к программе прилагается файл *brain_915.xml*, описывающий биоткани головы человека при длине волны источника излучения в 915 нм.
- SURFACE файл с описанием геометрии границ биотканей. В качестве примера к программе прилагается файл *planes.surface*. В данном случае реальная геометрия тканей головы человека аппроксимируется набором параллельных плоскостей. Файл имеет бинарный формат.

Для исследования производительности и проведения оптимизации программы имеет смысл варьировать только следующие параметры задачи (содержатся в XML файле):

- NumberOfPhotons – количество трассируемых фотонов. От этого параметра напрямую зависит время работы программы. Зависимость линейна.
- Area.PartitionNumber – содержит количество разбиений сетки по осям X, Y и Z. Сетка используется для хранения фотонных карт траекторий. Чем больше ее размер, тем больше памяти необходимо программе для работы.

При запуске программы используются следующие параметры командной строки (запуск программы без параметров приводит к выводу соответствующей справки):

```
xmcmlLauncher.exe
      -i <ИмяXMLФайла>.xml
      -s <ИмяSURFACEФайла>.surface
      -o <ИмяФайлаСРезультатамиМоделирования>.mcml.out
      -nthreads <КоличествоПараллельныхПотоков>
```

Пример запуска программы приведен на рис. 5.

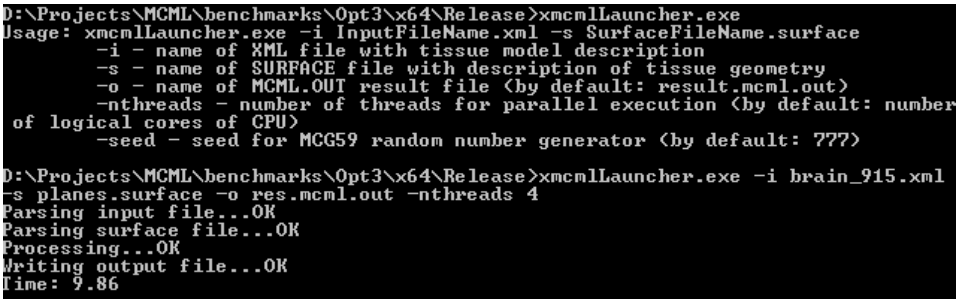

Рис. 5. Запуск программы xmcmlLauncher для ОС Windows

Результатом работы программы является бинарный файл *.mcml.out, который содержит как фотонные карты траекторий, так и данные о сигналах на детекторе.

Для просмотра результатов моделирования следует использовать программу mcmlVisualizer. Программа использует .NET Framework 4.0 и предназначена для работы на ОС Windows.

Помимо визуализации результатов, данная программа позволяет проводить сравнение двух *.mcml.out файлов на предмет их совпадения. Найденные ошибки с указанием места выводятся на экран. Отметим, что для проверки корректности полученных после оптимизации результатов следует выполнять сравнение *.mcml.out файлов. Обычное побитовое сравнение не всегда является показательным, поэтому для этих целей лучше использовать предложенную программу.

Отметим также, что полное совпадение результатов в рамках одной версии программы гарантируется только при запуске с неизменным числом потоков. В силу специфики алгоритма, результаты запусков с разным числом потоков будут различны.

3. Перенос алгоритма на Intel Xeon Phi

3.1. Прямой перенос базовой версии

Перед тем, как начать выполнение переноса программы на сопроцессор, необходимо обсудить некоторые ее особенности.

Распараллеливание ведется по фотонам, каждый поток обсчитывает свой набор траекторий (*./Base/xmcmlLauncher/launcher_omp.cpp*):

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

    …

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

        InitOutput(input, &(threadOutputs[threadId]));
        threadOutputs[threadId].specularReflectance =
            specularReflectance;

        uint64* trajectory =
            new uint64[threadOutputs[threadId].gridSize];

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

        delete[] trajectory;
    }
    …
}
```

Результаты трассировки сохраняются в структуре OutputInfo (*./Base/xmcml/mcml_kernel_types.h*):

```
typedef struct __OutputInfo
{
    uint64 numberOfPhotons;
    double specularReflectance;
    uint64* commonTrajectory;
```

```
    int gridSize;
    double* weightInDetector;
    DetectorTrajectory* detectorTrajectory;
    int numberOfDetectors;
} OutputInfo;
```

Сохраняются три типа результатов:

- Величина сигнала на детекторе – массив **weightInDetector**, количество элементов равно числу детекторов (в примере – 10), размер – 80 Б.
- Общая фотонная карта траекторий – массив **commonTrajectory**, количество элементов равно числу ячеек сетки (рис. 6) (в примере – 100*100*50=500 000 элементов), размер – 4 МБ.
- Фотонные карты траекторий для каждого детектора – массив **detectorTrajectory**. Для каждого из детекторов хранится массив, аналогичный **commonTrajectory**, того же размера. Общий размер данных для 10 детекторов – 40 МБ.

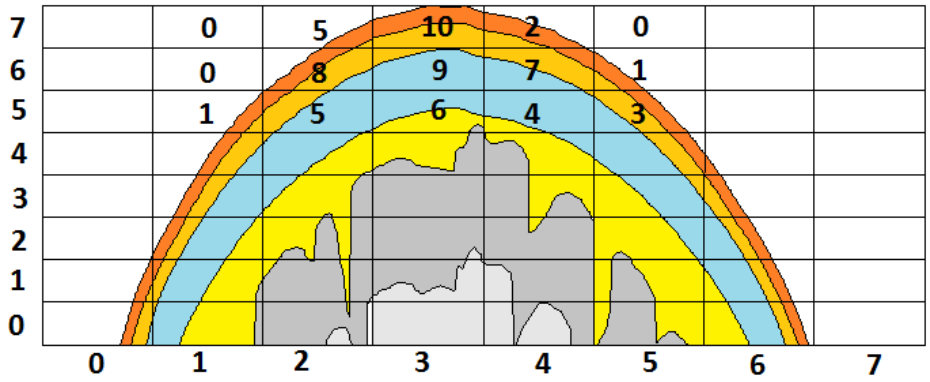

Рис. 6. Хранение числа посещений фотоном ячеек сетки в самой сетке

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 базовой 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 GB 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 базовой 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.

Процесс оптимизации программы начнем с выполнения ее прямого переноса на сопроцессор. Для этого скомпилируем программу под Intel Xeon Phi с помощью скрипта *mic_make.sh*.

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.

Выполнение программы на 8 потоках CPU для моделирования 100 000 фотонов занимает 35 секунды. При запуске в 1 поток это время равно 125 секундам.

Используемый в тестах сопроцессор обладает 8 GB встроенной памяти, поэтому запустить программу в текущей версии в 240 потоков не представляется возможным. Использование же 120 потоков дает время выполнения в 63 секунды. А на 60 потоках Intel Xeon Phi программа работает еще быстрее – 57 секунд.

Таким образом, прямой перенос базовой версии программы позволяет получить производительность на сопроцессоре, вдвое меньшую, чем производительность одного 8-ми ядерного CPU.

3.2. 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. Запустим программу в 4 потока с количеством трассируемых фотонов 20 000. Результаты времени работы функций программы приведены на рис. 7.

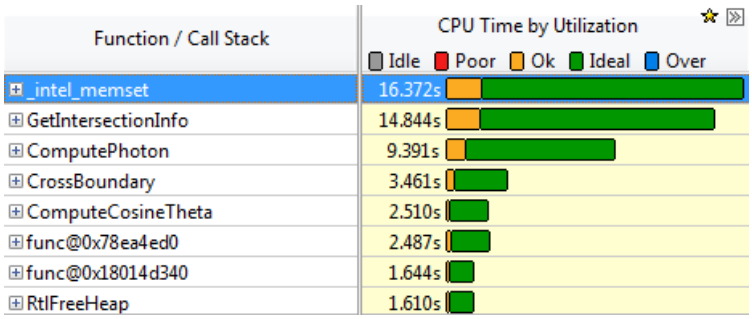

Рис. 7. Результаты профилировки базовой версии программы

Как видно из графиков, наибольшее время занимает функция memset(). Единственным местом в программе, где она используется, является функция трассировки фотона ComputePhoton() (*./Base/xmcml/mcml_kernel.cpp*):

```
void ComputePhoton(double specularReflectance,
    InputInfo* input, OutputInfo* output,
    MCG59* randomGenerator, uint64* trajectory)
{
    PhotonState photon;

    int trajectorySize = input->area->partitionNumber.x*
        input->area->partitionNumber.y*
        input->area->partitionNumber.z;
    memset(trajectory, 0, trajectorySize * sizeof(uint64));
    …
}
```

Функция memset() предназначена для того, чтобы обнулять память для хранения текущей траектории фотона перед его трассировкой. Эта память выделяется один раз для каждого потока в начале работы программы. Напомним, что элемент массива trajectory соответствует ячейке трехмерной сетки и хранит количество посещений фотоном данной ячейки. Размер массива для нашего примера равен 100*100*50 элементов или 4 МБ.

Следует отметить, что 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).

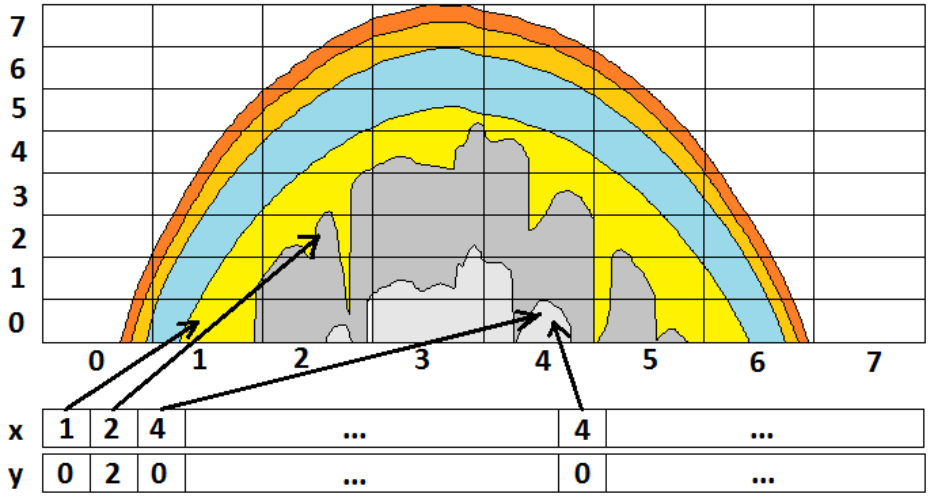

Рис. 8. Хранение числа посещений фотоном ячеек сетки в виде списка координат этих ячеек

Для устранения описанных выше недостатков предлагается использовать другую структуру данных для хранения траектории фотона: следует хранить не число посещений данной ячейки для всей сетки, а список координат посещенных ячеек (рис. 8) (*./Opt1/xmcml/mcml_kernel_types.h*):

```
#define MAX_TRAJECTORY_SIZE 2048

typedef struct __PhotonTrajectory
{
    byte x[MAX_TRAJECTORY_SIZE];
    byte y[MAX_TRAJECTORY_SIZE];
    byte z[MAX_TRAJECTORY_SIZE];
    uint size;
} PhotonTrjectory;
```

Массивы x, y и z хранят координаты ячеек сетки в трехмерном пространстве, size – текущая длина траектории.

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.

Для того чтобы использовать эту новую структуру данных, в программу необходимо внести некоторые изменения.

Во-первых, в функции LaunchOMP() меняем часть, связанную с выделением памяти под траекторию (*./Opt1/xmcmlLauncher/launcher_omp.cpp*):

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

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

        InitOutput(input, &(threadOutputs[threadId]));
        threadOutputs[threadId].specularReflectance =
            specularReflectance;

        PhotonTrjectory trajectory;

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

Во-вторых, в функции ComputePhoton() избавляемся от обнуления памяти (*./Opt1/xmcml/mcml_kernel.cpp*):

```
void ComputePhoton(double specularReflectance, InputInfo*
input, OutputInfo* output,
    MCG59* randomGenerator, PhotonTrjectory* trajectory)
{
    PhotonState photon;

    trajectory->size = 0;
    …
}
```

И в-третьих, необходимо исправить функции работы с траекторией. Исправляем код функции UpdatePhotonTrajectory() (*./Opt1/xmcml/mcml_kernel.cpp*):

```
void UpdatePhotonTrajectory(PhotonState* photon, InputInfo*
input, OutputInfo* output,
    PhotonTrjectory* trajectory, double3
previousPhotonPosition)
{
    …
    while ((plane <= finishPosition.z) && (plane <
        input->area->corner.z + input->area->length.z))
    {
```

```
        double3 intersectionPoint =
            GetPlaneSegmentIntersectionPoint(startPosition,
                finishPosition, plane);

        byte3 areaIndexVector =
            GetAreaIndexVector(intersectionPoint,
            input->area);
        if (areaIndexVector.x == 255 &&
            areaIndexVector.y == 255 &&
            areaIndexVector.z == 255)
            break;

        assert(trajectory->size < MAX_TRAJECTORY_SIZE);

        trajectory->x[trajectory->size] =
            areaIndexVector.x;
        trajectory->y[trajectory->size] =
            areaIndexVector.y;
        trajectory->z[trajectory->size] =
            areaIndexVector.z;
        ++(trajectory->size);

        int index = areaIndexVector.x*
            input->area->partitionNumber.y*
            input->area->partitionNumber.z +
            areaIndexVector.y*
            input->area->partitionNumber.z +
            areaIndexVector.z;
        ++(output->commonTrajectory[index]);

        plane += step;
    }
}
```

Функция assert() используется для генерации ошибки в случае, если реальная длина траектории фотона будет больше, чем размер массива, выделенного под ее хранение. Для использования этой функции следует подключить заголовочный файл «assert.h».

Здесь byte3 – новый тип данных (*./Opt1/xmcml/mcml_types.h)*:

```
typedef union __byte3
{
      struct { int x, y, z; };
      struct { int cell[3]; };
} byte3;
```

Функцию GetAreaIndexVector() также следует добавить (*./Opt1/xmcml/mcml_kernel.cpp*):

```
byte3 GetAreaIndexVector(double3 photonPosition,
    Area* area)
{
    byte3 result;
    result.x = 255;
    result.y = 255;
    result.z = 255;

    double indexX = area->partitionNumber.x*
       (photonPosition.x - area->corner.x)/area->length.x;
    double indexY = area->partitionNumber.y*
       (photonPosition.y - area->corner.y)/area->length.y;
    double indexZ = area->partitionNumber.z*
       (photonPosition.z - area->corner.z)/area->length.z;

    bool isPhotonInArea = !((indexX < 0.0 ||
        indexX >= area->partitionNumber.x) ||
        (indexY < 0.0 || indexY >= area->partitionNumber.y)
        || (indexZ < 0.0 ||
        indexZ >= area->partitionNumber.z));

    if (isPhotonInArea)
    {
        result.x = (byte)indexX;
        result.y = (byte)indexY;
        result.z = (byte)indexZ;
    }

    return result;
}
```

И наконец, необходимо полностью обновить функцию UpdateDetectorTrajectory() (*./Opt1/xmcml/mcml_kernel.cpp*):

```
void UpdateDetectorTrajectory(OutputInfo* output,
    Area* area, PhotonTrjectory* trajectory,
    int detectorId)
{
    int index;
    int ny = area->partitionNumber.y;
    int nz = area->partitionNumber.z;

    uint64* detectorTrajectory =
        output->detectorTrajectory[detectorId].trajectory;

    for (int i = 0; i < trajectory->size; ++i)
    {
        index = trajectory->x[i]*ny*nz +
            trajectory->y[i]*nz + trajectory->z[i];
        ++(detectorTrajectory[index]);
    }
```

```
    ++(output->detectorTrajectory[detectorId].
            numberOfPhotons);
}
```

Изменение используемой структуры данных позволило сократить время вычислений на CPU в 1 поток до 106 секунд (на 18%). При этом моделирование с использованием 8 потоков CPU стало занимать 32 секунды (ускорение на 9%). Время работы программы на MIC в 60 потоков сократилось до 56, а на 120 – до 48 секунд. Таким образом, описанная выше оптимизация позволила уменьшить время вычислений с использованием сопроцессора на 31%.

3.3. 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.

Заметим, что в данной версии мы отказываемся от дублирования еще одного массива – массива сигналов на детекторах (output->weightInDetector). Целесообразность такого решения предлагается оценить слушателю в рамках дополнительных заданий.

Для реализации предложенной идеи прежде всего следует избавиться от дублирования данных. Основные изменения коснутся функции параллельного запуска процесса моделирования LaunchOMP() (*./Opt2/xmcmlLauncher/launcher_omp.cpp*):

```
void LaunchOMP(InputInfo* input, OutputInfo* output,
    MCG59* randomGenerator, int numThreads)
{
    …
    #pragma omp parallel
    {
        int threadId = omp_get_thread_num();
        InitThreadOutput(&(threadOutputs[threadId]),
            output);

        PhotonTrjectory trajectory;

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

    output->specularReflectance = specularReflectance;
    for (int i = 0; i < numThreads; ++i)
    {
        for (int j = 0; j < output->gridSize; ++j)
        {
            output->commonTrajectory[j] +=
                threadOutputs[i].commonTrajectory[j];
        }
    }

    for (int i = 0; i < numThreads; ++i)
    {
        FreeThreadOutput(threadOutputs + i);
    }
    delete[] threadOutputs;
}
```

Финальное суммирование осталось только для общей фотонной карты, вместо функций InitOutput() и FreeOutput() используются (*./Opt2/xmcmlLauncher/launcher_omp.cpp*):

```
void InitThreadOutput(OutputInfo* dst, OutputInfo* src)
{
    dst->gridSize = src->gridSize;
    dst->detectorTrajectory = src->detectorTrajectory;
```

```
    dst->numberOfDetectors = src->numberOfDetectors;
    dst->numberOfPhotons = src->numberOfPhotons;
    dst->specularReflectance = src->specularReflectance;
    dst->weightInDetector = src->weightInDetector;

    dst->commonTrajectory = new uint64[dst->gridSize];
    memset(dst->commonTrajectory, 0, dst-
>gridSize*sizeof(uint64));
}

void FreeThreadOutput(OutputInfo* output)
{
    if (output->commonTrajectory != NULL)
    {
        delete[] output->commonTrajectory;
    }
}
```

Избавившись от дублирования, необходимо позаботиться о синхронизации. Для этого прежде нужно изменить функцию UpdateDetectorTrajectory() (*./Opt2/xmcml/mcml_kernel.cpp*):

```
void UpdateDetectorTrajectory(OutputInfo* output, Area*
area, PhotonTrjectory* trajectory, int detectorId)
{
    int index;
    int ny = area->partitionNumber.y;
    int nz = area->partitionNumber.z;

    uint64* detectorTrajectory =
        output->detectorTrajectory[detectorId].trajectory;

    for (int i = 0; i < trajectory->size; ++i)
    {
        index = trajectory->x[i]*ny*nz +
            trajectory->y[i]*nz + trajectory->z[i];

        #pragma omp atomic
        ++(detectorTrajectory[index]);
    }

    #pragma omp atomic
    ++(output->detectorTrajectory[detectorId].
        numberOfPhotons);
}
```

Также изменению подвергнется функция UpdateWeightInDetector() (*./Opt2/xmcml/mcml_kernel.cpp*):

```
void UpdateWeightInDetector(OutputInfo* output,
```

```
    double photonWeight, int detectorId)
{
    #pragma omp atomic
    output->weightInDetector[detectorId] += photonWeight;
}
```

Время работы программы на центральном процессоре после применения оптимизации практически не изменилось. Время работы программы на сопроцессоре в 120 потоков уменьшилось до 36 секунд. Более того, появилась возможность запуска программы в 240 потоков. Однако время моделирования в этом случае составило 40 секунд. Итого, данная оптимизация позволила сократить время вычислений на сопроцессоре еще на 35%.

3.4. Оптимизация 3: балансировка нагрузки

Профилировка программы с использованием Intel VTune Amplifier XE выявила еще одну особенность используемого алгоритма – разное время выполнения отдельных потоков (рис. 9).

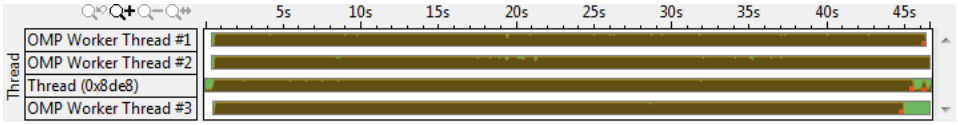

Рис. 9. Диаграмма исполнения потоков на CPU (до балансировки нагрузки)

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 (*./Opt3/xmcmlLauncher/launcher_omp.cpp*):

```
void LaunchOMP(InputInfo* input, OutputInfo* output, MCG59*
randomGenerator, int numThreads)
{
    …
    OutputInfo* threadOutputs = new OutputInfo[numThreads];
    for (int i = 0; i < numThreads; ++i)
    {
        InitThreadOutput(&(threadOutputs[i]), output);
    }

    PhotonTrjectory* trajectory =
        new PhotonTrjectory[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]));
    }

    …

    delete[] trajectory;
}
```

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

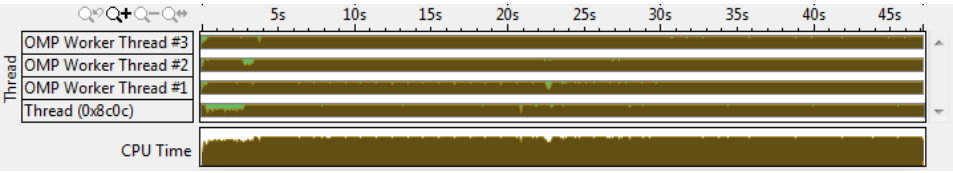

Рис. 10. Диаграмма исполнения потоков на CPU (после балансировки нагрузки)

3.5. Сводные результаты

Результаты тестовых запусков алгоритма для моделирования переноса излучения на центральном процессоре и сопроцессоре приведены ниже (таблица 1 и таким образом, 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.

таблица 2 соответственно).

Таблица 1. Время работы (в секундах) программы для моделирования переноса излучения на CPU, 100 000 фотонов

Число ядер CPU				
	1	2	4	8
Базовая версия	125,41	71,57	46,41	34,65
Оптимизация 1	105,85	57,49	38,98	31,7
Оптимизация 2	104,74	57,19	41,56	32,17
Оптимизация 3	103,79	61,85	40,41	**30,54**

Таким образом, 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.

Таблица 2. Время работы (в секундах) программы для моделирования переноса излучения на сопроцессоре Intel Xeon Phi, 100 000 фотонов

Число потоков MIC			
	60	120	240
Базовая версия	57,74	63,05	-
Оптимизация 1	56,28	48,15	-
Оптимизация 2	43,36	35,5	40,41
Оптимизация 3	38,1	**32,45**	44,64

Сравнение времени работы лучших версий программы для процессора и сопроцессора при различном числе трассируемых фотонов продемонстрировано на рис. 11.

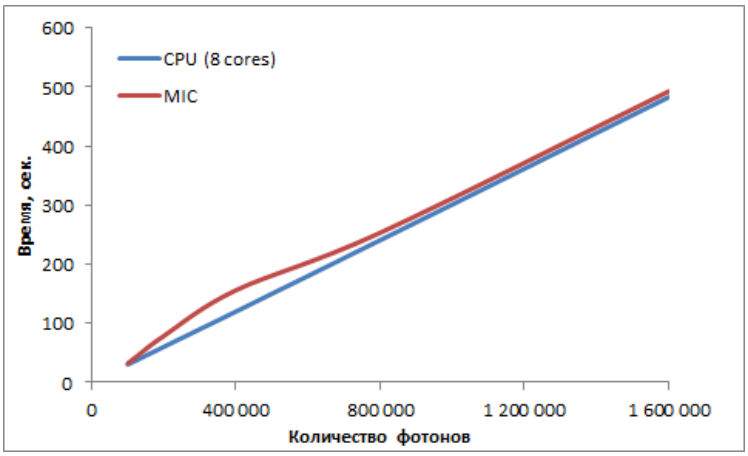

Рис. 11. Сравнение времени работы программы для моделирования переноса излучения на CPU и Intel MIC

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. 7), 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. Дополнительные задания

1. Реализовать версию алгоритма с дублированием массива сигналов на детекторах – output->weightInDetector (избавиться от синхронизации записи результатов в этот массив). Оценить целесообразность такой оптимизации.

2. Реализовать версию алгоритма, работающего в режиме offload. Выносить на сопроцессор имеет смысл только распараллеленную часть кода, все остальное должно делаться на процессоре.

3. Применить технику двойной буферизации в режиме offload для обеспечения эффективной передачи результатов моделирования с сопроцессора на CPU. На каждой итерации, начиная со второй, в момент обсчета новой порции фотонов можно организовать передачу результатов трассировки предыдущей порции на CPU.

5. Литература

5.1. Дополнительная литература

1. Intel Xeon Phi Coprocessor System Software Developers Guide, revision 2.03, 2012

2. Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture, Volume 1a, January 29, 2013

3. J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013. -432 p.

5.2. Информационные ресурсы сети Интернет

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