Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

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

При поддержке компании Intel

Бастраков С.И., Горшков А.В.
Кафедра математического обеспечения ЭВМ

Содержание

- Подходы к оптимизации программного пакета для моделирования динамики электромагнитного поля методом FDTD
  - Общее описание метода FDTD
  - Программная реализация
  - Оптимизация 1: векторизация и изменение структуры данных
  - Оптимизация 2: улучшение масштабируемости
  - Общие результаты оптимизации

Содержание

- Подходы к оптимизации программного пакета для Монте-Карло моделирования переноса излучения
  - Общее описание алгоритма
  - Прямой перенос программного пакета на сопроцессор
  - Оптимизация 1: обновление структуры данных для хранения траектории фотона
  - Оптимизация 2: отказ от использования дублирующих массивов
  - Оптимизация 3: балансировка нагрузки
  - Общие результаты оптимизации
- Литература

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

Введение

- **FDTD** (**Finite-Difference Time-Domain**) - один из широко используемых методов вычислительной электродинамики.
- Явный конечно-разностный метод численного решения уравнений Максвелла.
- Особенностью метода является использование специальной сетки для компонент электромагнитного поля.
- Рассматривается портирование на Xeon Phi и оптимизация реализации метода FDTD, созданной на основе реализации в составе программного пакета для моделирования плазмы PICADOR [2].

Постановка задачи

- **Расчетная область** – трехмерный прямоугольный параллелепипед $\{(x.y.z)\colon a_x \le x \le b_x, a_y \le y \le b_y, a_z \le z \le b_z\}$.
- В каждый момент времени t в каждой точке расчетной области (x, y, z) определены **электрическое поле** $\boldsymbol{E}(x, y, z, t) = (E_x(x, y, z, t), E_y(x, y, z, t), E_z(x, y, z, t))$ и **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 $t = t_{max}$.

Уравнения Максвелла

- Динамика электромагнитного поля подчиняется системе уравнений Максвелла (приведена запись двух уравнений для случая вакуума в системе единиц СГС, c – скорость света в вакууме):

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

Сетка

- 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).
- **Шаги сетки** $\Delta x = \frac{b_x - a_x}{n_x}, \Delta y = \frac{b_y - a_y}{n_y}, \Delta z = \frac{b_z - a_z}{n_z}$.
- Моделирование по времени происходит с заданным шагом Δt в дискретные моменты $0, \Delta t, 2\Delta t, 3\Delta t, \ldots$, последовательность завершается при превышении t_{max}.
- Для индексирования узлов сетки используются естественные трехмерные индексы $(i, j, k): 0 \le i \le n_x, 0 \le j \le n_y, 0 \le k \le n_z$.

Сетка Yee

- FDTD использует специальную сетку - сетку Yee.
- $E_x(i,j,k) \leftrightarrow \left(a_x + i\Delta x, a_y + (j+0.5)\Delta y, a_z + (k+0.5)\Delta z\right)$
- $E_y(i,j,k) \leftrightarrow \left(a_x + (i+0.5)\Delta x, a_y + j\Delta y, a_z + (k+0.5)\Delta z\right)$
- $E_z(i,j,k) \leftrightarrow \left(a_x + (i+0.5)\Delta x, a_y + (j+0.5)\Delta y, a_z + k\Delta z\right)$
- $B_x(i,j,k) \leftrightarrow \left(a_x + (i+0.5)\Delta x, a_y + j\Delta y, a_z + k\Delta z\right)$
- $B_y(i,j,k) \leftrightarrow \left(a_x + i\Delta x, a_y + (j+0.5)\Delta y, a_z + k\Delta z\right)$
- $B_z(i,j,k) \leftrightarrow \left(a_x + i\Delta x, a_y + j\Delta y, a_z + (k+0.5)\Delta z\right)$
- Разные компоненты поля сдвинуты относительно центра соответствующей ячейки на полшага по одной или двум осям.
- Компоненты ***B*** сдвинуты относительно ***E*** на $\Delta t/2$.

Метод FDTD

- 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 ***E*** и ***B*** для вычисления новых значений ***E*** (обновление ***E***),
  - использование текущих значений ***B*** и новых значений ***E*** для вычисления новых значений ***B*** (обновление ***B***).

Обновление E и B

- $E_x(i,j,k) := E_x(i,j,k) + c \cdot \Delta t \cdot \left(\frac{B_z(i,j+1,k) - B_z(i,j,k)}{\Delta y} - \frac{B_y(i,j,k+1) - B_y(i,j,k)}{\Delta z} \right)$
- $E_y(i,j,k) := E_y(i,j,k) - c \cdot \Delta t \cdot \left(\frac{B_z(i+1,j,k) - B_z(i,j,k)}{\Delta x} - \frac{B_x(i,j,k+1) - B_x(i,j,k)}{\Delta z} \right)$
- $E_z(i,j,k) := E_z(i,j,k) + c \cdot \Delta t \cdot \left(\frac{B_y(i+1,j,k) - B_y(i,j,k)}{\Delta x} - \frac{B_x(i,j+1,k) - B_x(i,j,k)}{\Delta y} \right)$

- $B_x(i,j,k) := B_x(i,j,k) - c \cdot \Delta t \cdot \left(\frac{E_z(i,j,k) - E_z(i,j-1,k)}{\Delta y} - \frac{E_y(i,j,k) - E_y(i,j,k-1)}{\Delta z} \right)$
- $B_y(i,j,k) := B_y(i,j,k) + c \cdot \Delta t \cdot \left(\frac{E_z(i,j,k) - E_z(i-1,j,k)}{\Delta x} - \frac{E_x(i,j,k) - E_x(i,j,k-1)}{\Delta z} \right)$
- $B_z(i,j,k) := B_z(i,j,k) - c \cdot \Delta t \cdot \left(\frac{E_y(i,j,k) - E_y(i-1,j,k)}{\Delta x} - \frac{E_x(i,j,k) - E_x(i,j-1,k)}{\Delta y} \right)$

Граничные условия

- Приведенные 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.
- Будут использоваться периодические граничные условия по всем осям: $E_x(n_x, j, k) := E_x(0, j, k$, $B_x(0, j, k) := B_x(n_x, j, k)$, аналогично для других границ и компонент поля.

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 ***E*** и ***B***, являющиеся прямой записью разностной схемы метода FDTD.
- Данные функции аналогичны, будем иллюстрировать все оптимизации на примере обновления ***E***.

Программная реализация…

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

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

const double C = 29979245800.0;
```

Программная реализация…

```
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);
    }
}
```

Программная реализация

- Основной вычислительный цикл:

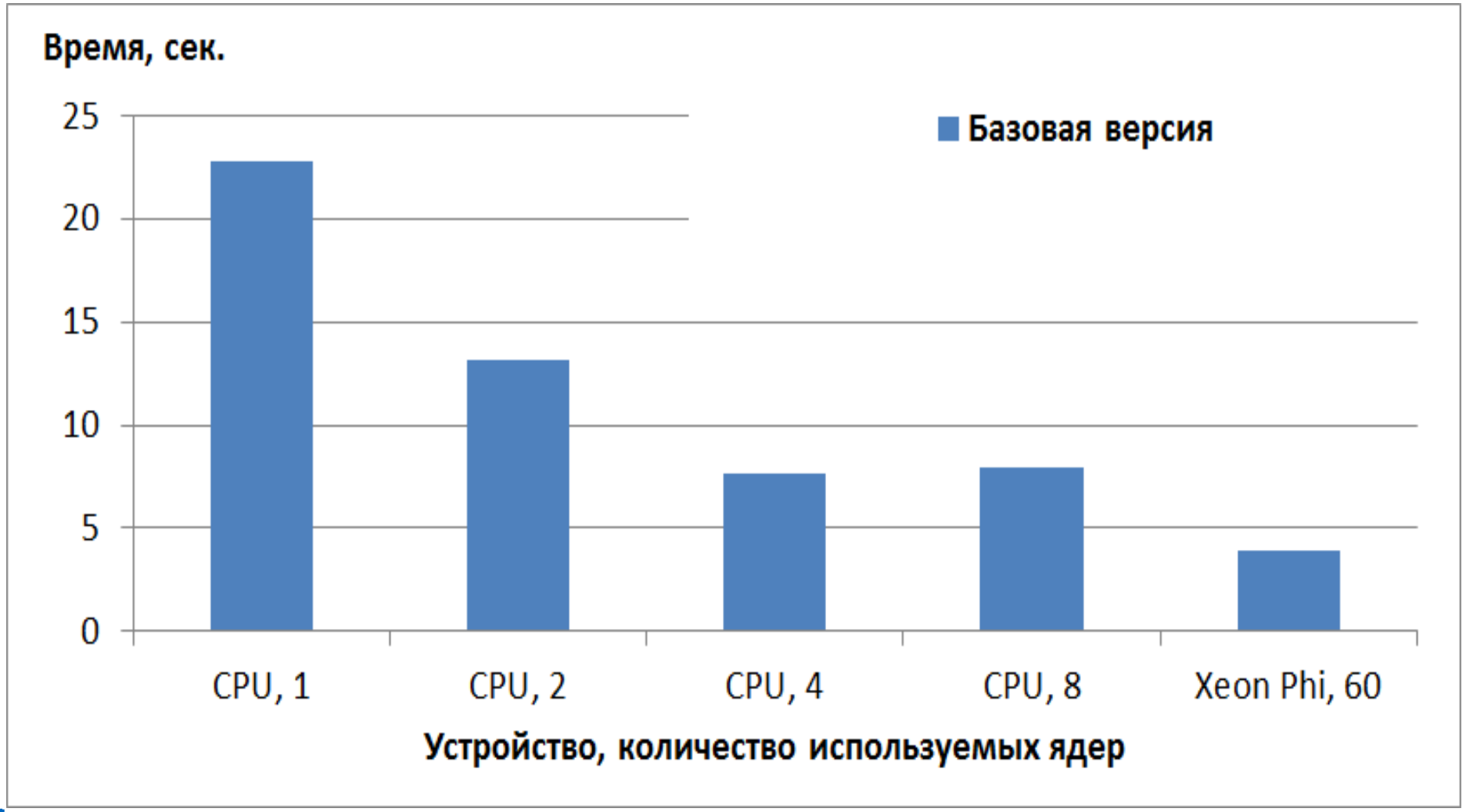
```
for (int t = 0; t < numSteps; t++)
{
    updateE(parameters, b, e);
    updateBoundaryE(parameters, e);
    updateB(parameters, b, e);
    updateBoundaryB(parameters, b);
}
```

Бенчмарк и инфраструктура

- Моделирование распространения плоской волны на сетке 256x256x256 со 100 итерациями по времени.

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

Производительность базовой версии



Исследование возможности векторизации

- Как видно из отчета о векторизации –vec-report3, основные циклы не векторизуются:

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

- Причина в том, что в условии цикла используется поле структуры Parameters. Скопируем parameters.nz в локальную переменную и будем использовать ее.
- После этого структура цикла становится пригодной, однако векторизация не происходит из-за потенциальных зависимостей между итерациями:

```
naive.cpp(25): (col. 5) remark: loop was not
vectorized: existence of vector dependence
```

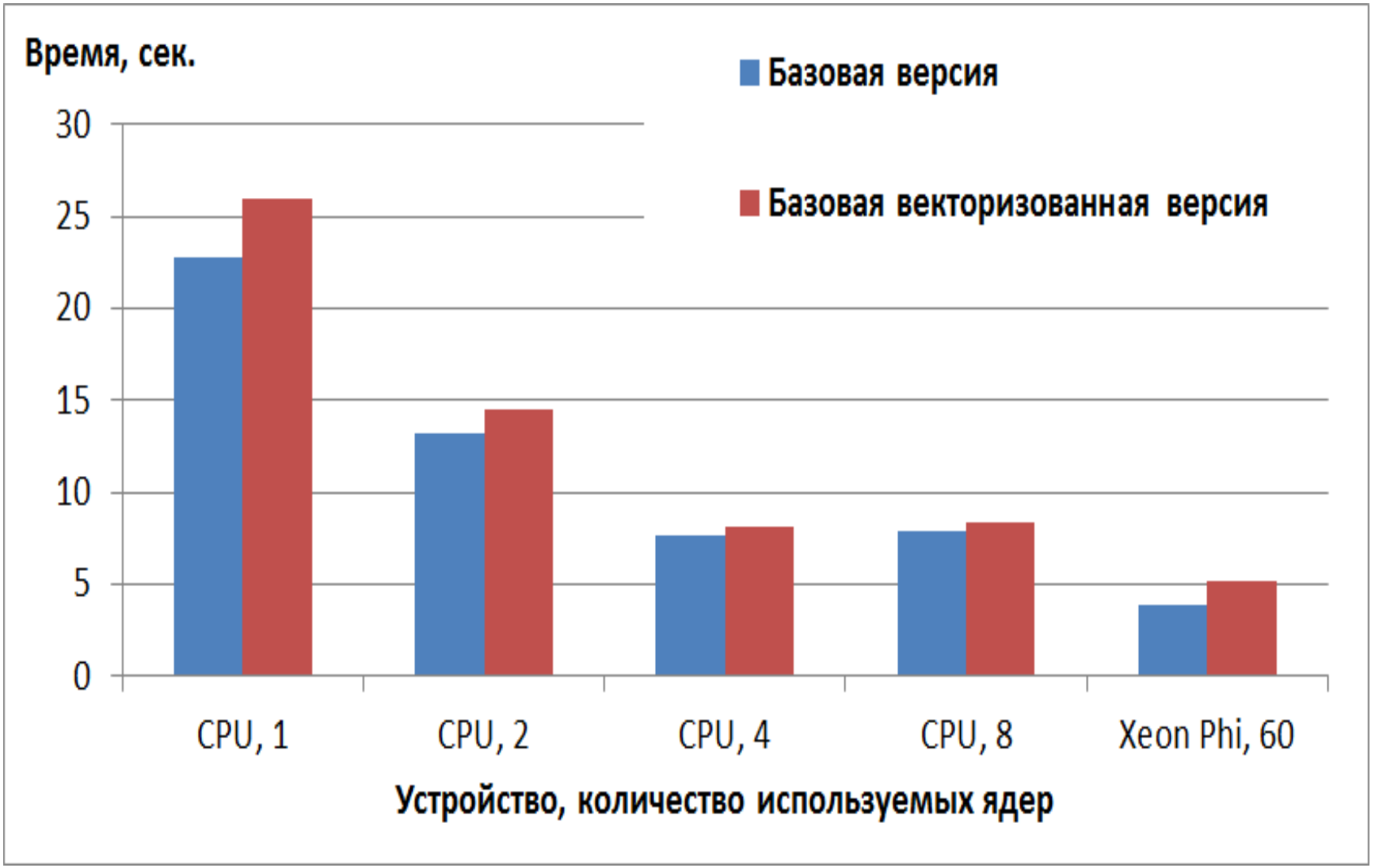
Векторизация кода

- 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.

Базовая векторизованная версия

```
void updateE(const Parameters & parameters,
Double3 *** b, Double3 *** e) {
    …
    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++) {
     …
    }
}
```

Производительность базовой векторизованной версии



Анализ векторизации

- Векторизация не только привела к повышению производительности, но даже немного замедлила код.
- Наиболее вероятная причина 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.

Улучшенная векторизованная версия…

```
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++)
    {
```

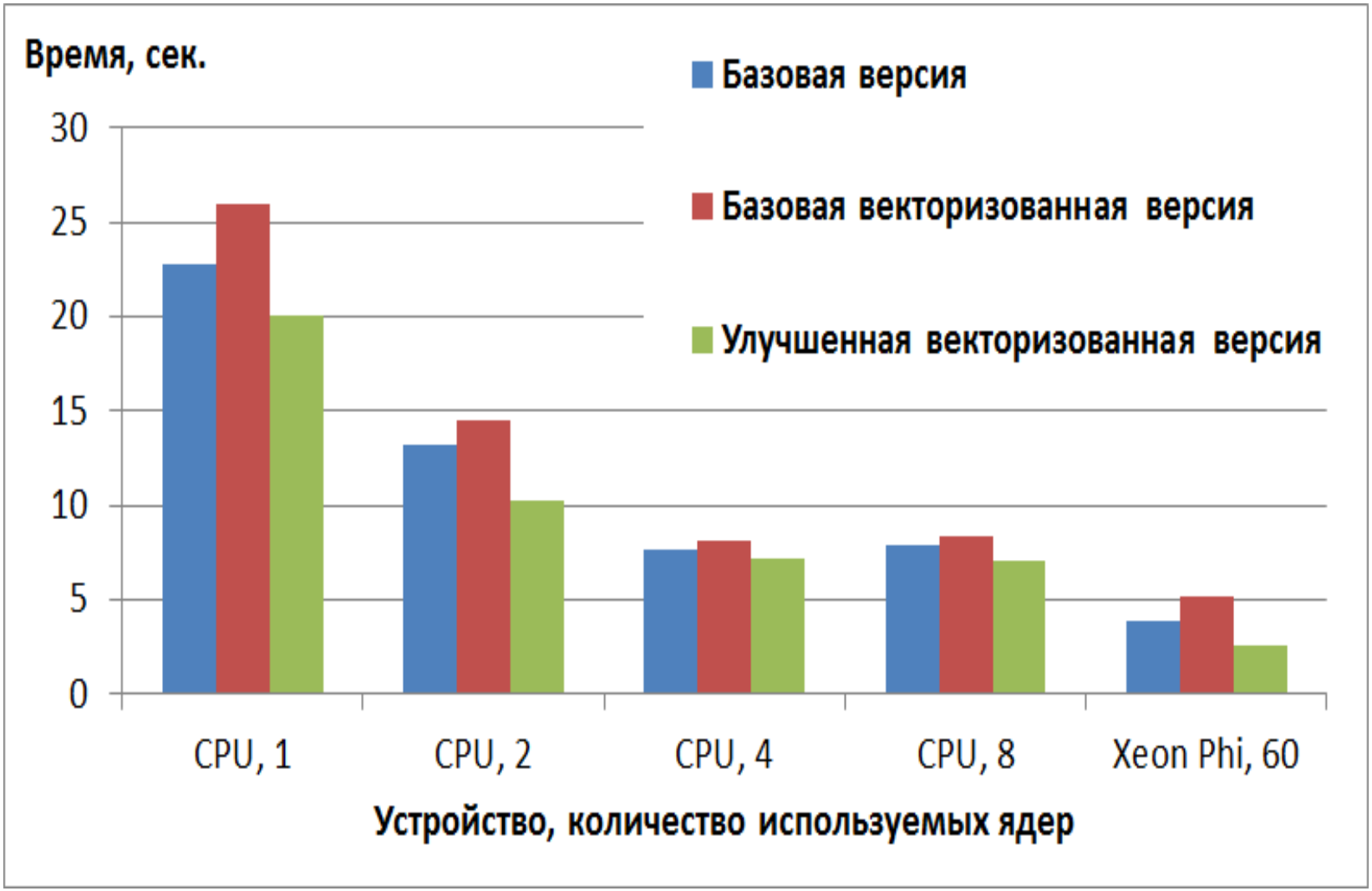
Улучшенная векторизованная версия…

```
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];
```

Улучшенная векторизованная версия

```
#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]);
        }
    }
}
```

Производительность улучшенной векторизованной версии



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.
- 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.

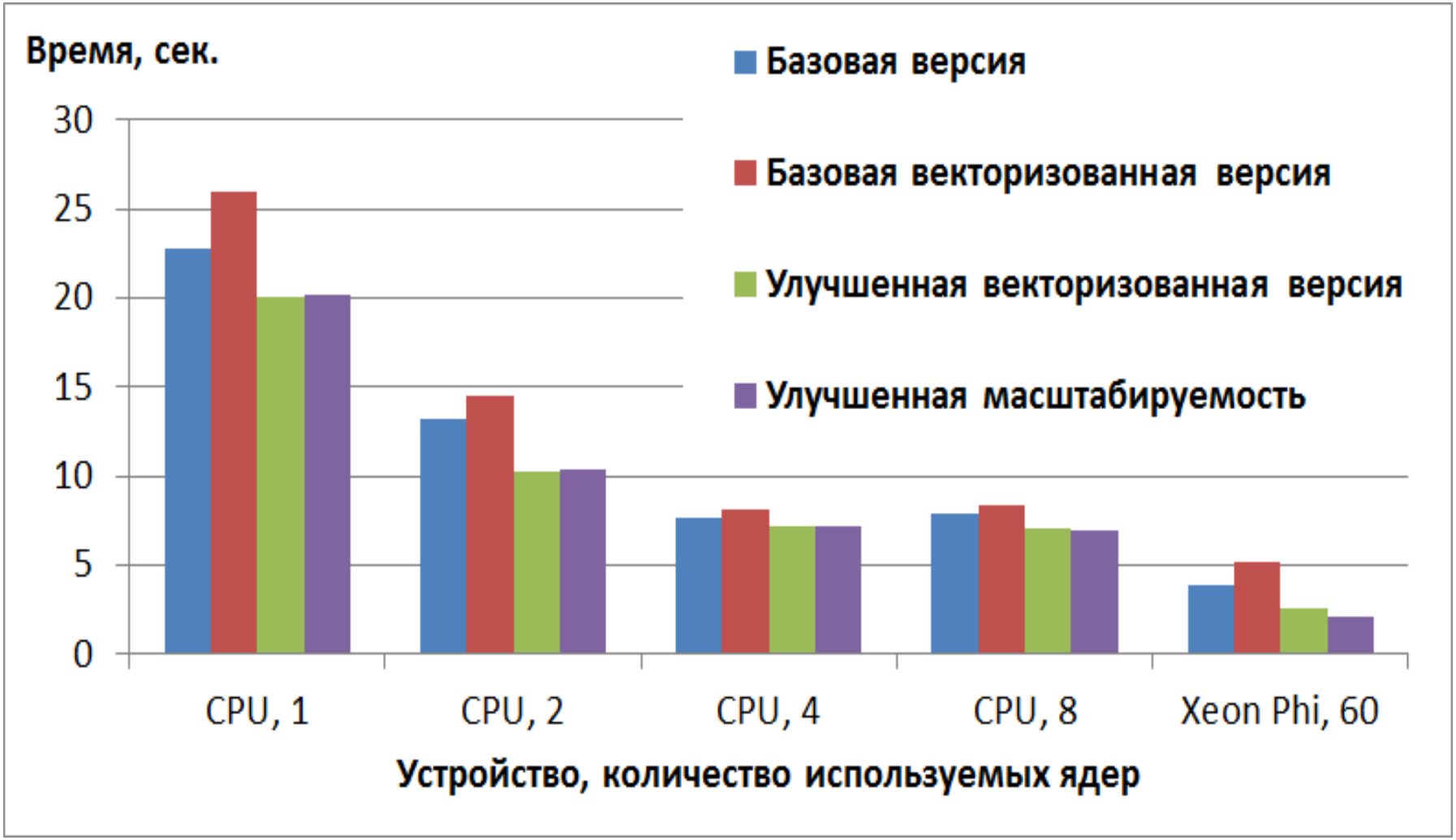
Улучшение масштабируемости

- Параллелизм в данной задаче чрезмерно крупнозернистый (coarse grained) для Xeon Phi.
- 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).

Версия с улучшенной масштабируемостью

```
void updateE(const Parameters & parameters, double *** bx, double *** by, double *** bz, double *** ex, double *** ey, double *** ez) {
    …
    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;

        …
    }
}
```

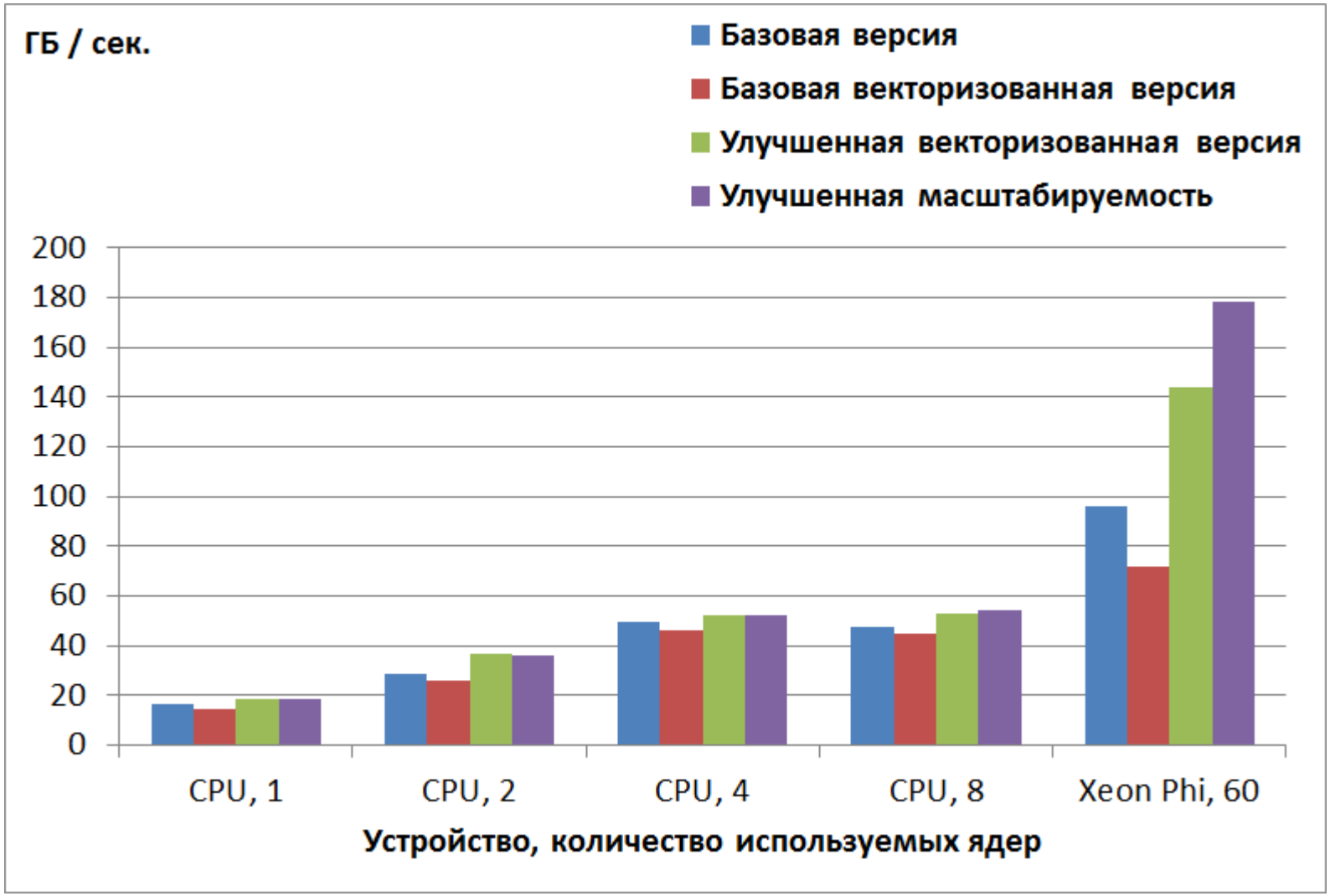
Производительность всех версий



Пропускная способность 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.

Пропускная способность памяти



Результаты

- 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).
- Лучшая версия на Xeon Phi обгоняет лучшую версию на CPU примерно в 3.3 раза.
- Производительность реализации метода FDTD ограничена в первую очередь пропускной способностью памяти.
- На CPU и Xeon Phi достигается примерно 50% пиковой пропускной способности памяти.

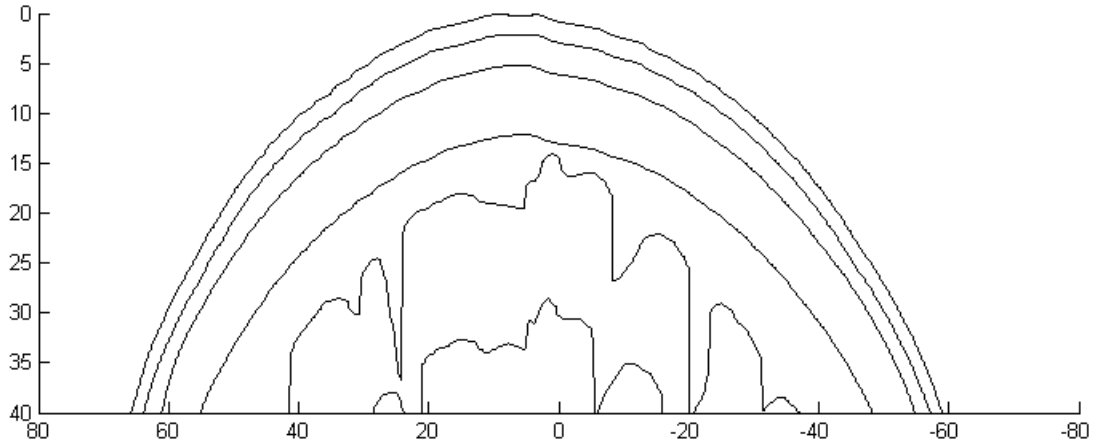
Подходы к оптимизации программного пакета для Монте-Карло моделирования переноса излучения

Актуальность

- Традиционные методы диагностики:
  - Магнитно-резонансная томография (МРТ)
  - Компьютерная томография (КТ)
  - Позитронно-эмиссионная томография (ПЭТ)
- Оптическая диагностика
  - Оптическая когерентная томография (ОКТ)
  - **Оптическая диффузионная спектроскопия (ОДС)**

- Возможные приложения ОДС:
  - Диагностика и лечение раковых опухолей, в частности, рака груди
  - Мониторинг активности зон коры головного мозга
  - Планирование фотодинамической терапии
  - Мониторинг состояния пациента при хирургическом вмешательстве

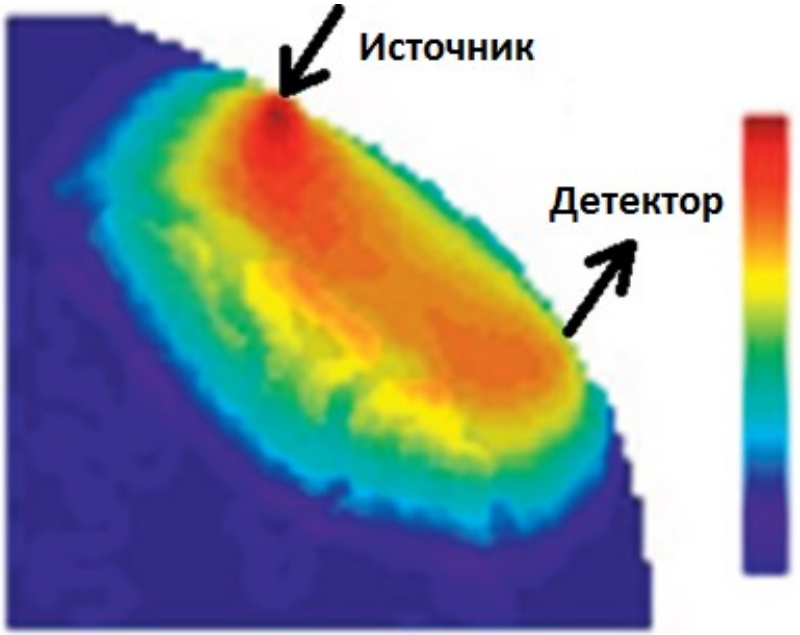
Общее описание алгоритма…



- 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.

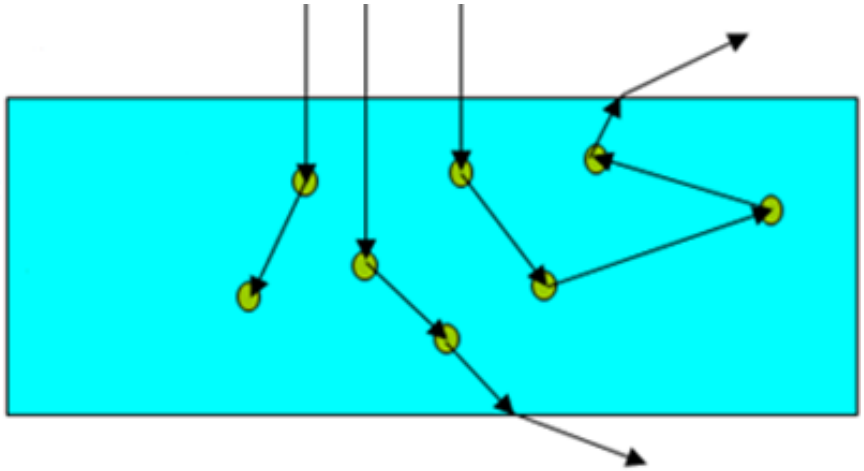
Общее описание алгоритма…

- Каждый слой характеризуется набором границ, описываемых триангулированными поверхностями.
- Источник излучения представляет собой бесконечно тонкий луч фотонов и описывается направлением и положением в трехмерном пространстве.
- Детектор - некоторая замкнутая область на поверхности исследуемого объекта, которая способна улавливать проходящие через нее фотоны.



Общее описание алгоритма…

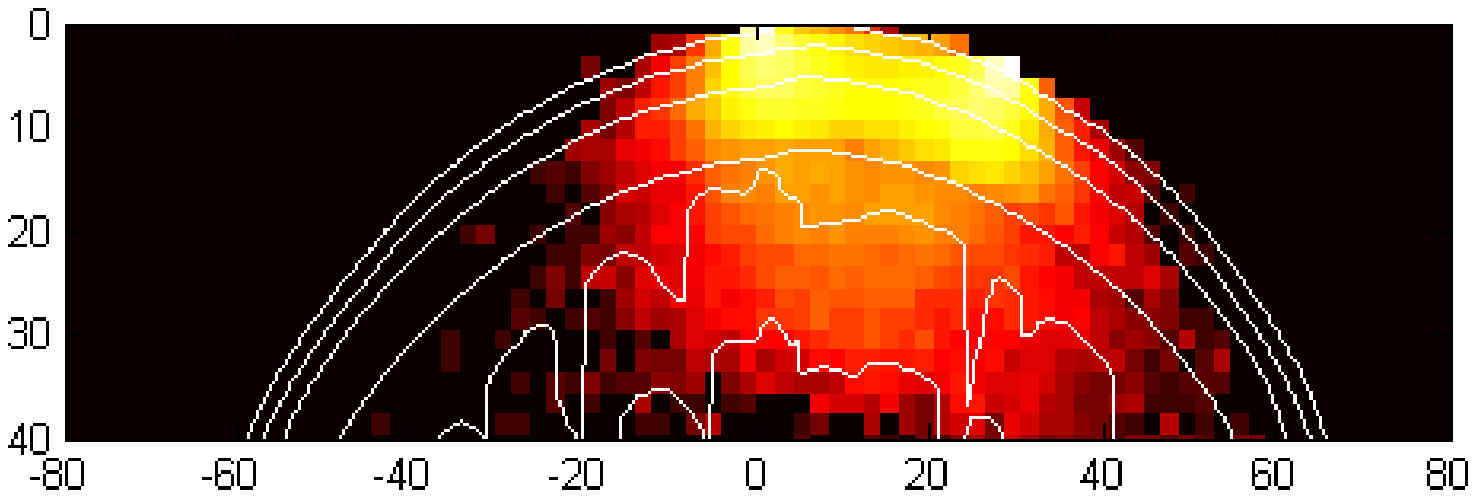
- Моделируется поведение набора фотонов в биоткани.
- Фотоны с одинаковой траекторией движения объединяются в пакет, вес пакета принимается равным 1.
- Пакет начинает движение от источника излучения.



- На каждом шаге трассировки:
  - Случайно выбирается новое направление движения пакета;
  - Случайно выбирается длина свободного пробега пакета фотонов;
  - Часть фотонов пакета поглощается средой (вес пакета уменьшается).
- Трассировка завершается, если пакет вылетел за границы среды, либо если вес пакета становится меньше минимального.

Общее описание алгоритма

- 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).



Прямой перенос: особенности программы…

- Распараллеливание ведется по фотонам, каждый поток обсчитывает свой набор траекторий:

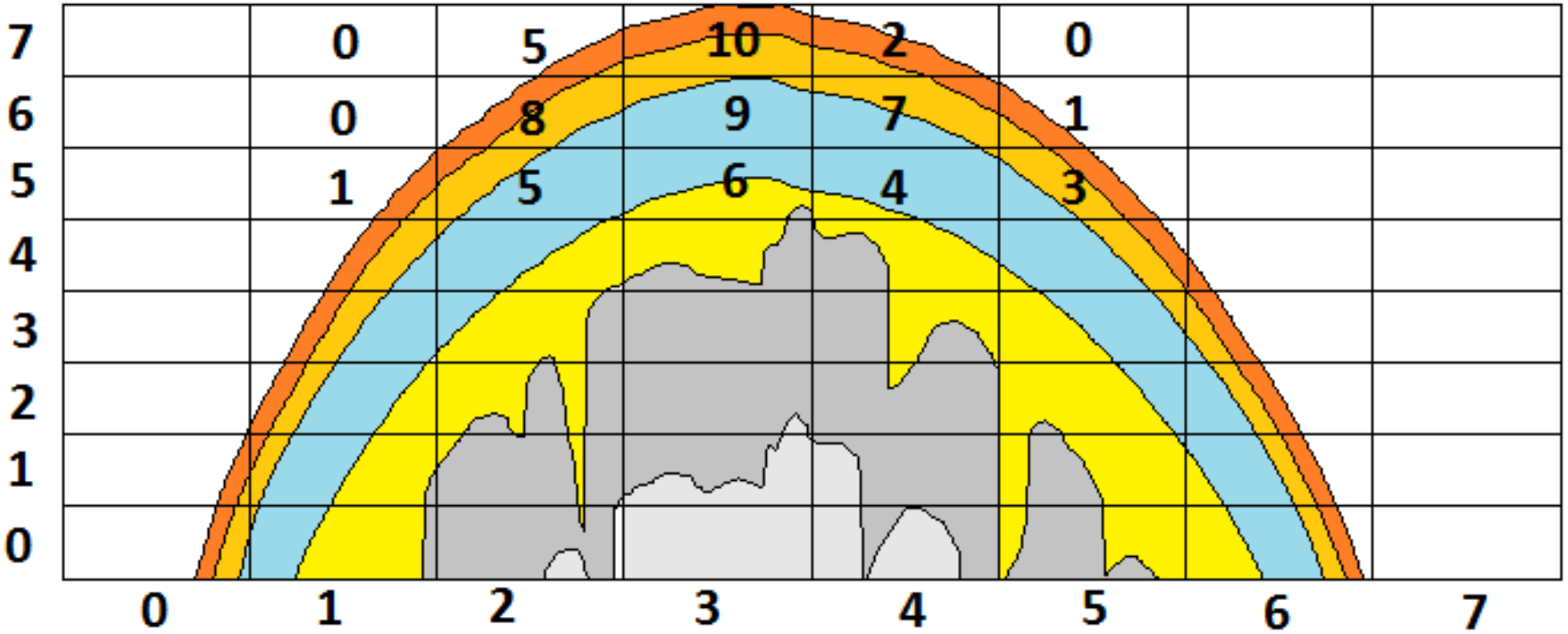
```
void LaunchOMP(InputInfo* input, OutputInfo* output,
    MCG59* randomGenerator, int numThreads)
{
    omp_set_num_threads(numThreads);

    …

    #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);
        }
    }
    …
}
```

Прямой перенос: особенности программы…

- Результатами моделирования 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. 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.

Прямой перенос: особенности программы…

- Фотонные карты траекторий хранятся в виде трехмерной сетки следующего вида:



Прямой перенос: особенности программы…

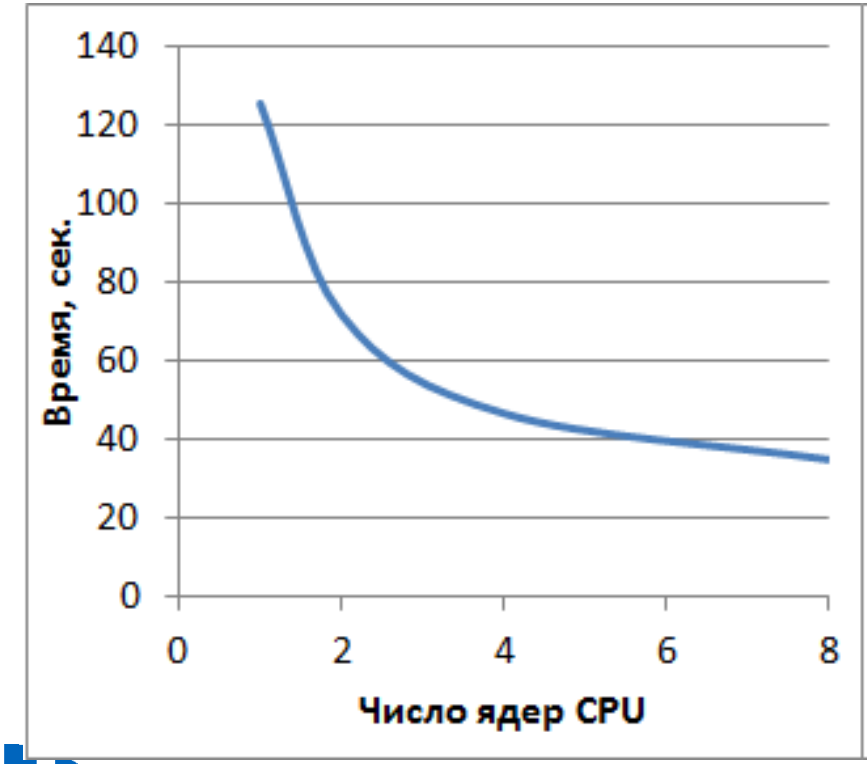
- Каждому фотону требуется доступ к массивам результатов для их обновления, а значит необходимо либо использовать синхронный доступ к данным на запись, либо воспользоваться техникой дублирования данных.
- В исходной версии программы применяется второй подход: создаются копии результирующих массивов для каждого потока исполнения, а после окончания расчетов данные этих копий суммируются.
  - Объем дополнительной памяти на CPU (8 потоков): 350 МБ
  - Объем дополнительной памяти на MIC (240 потоков): более 10 ГБ **(!)**

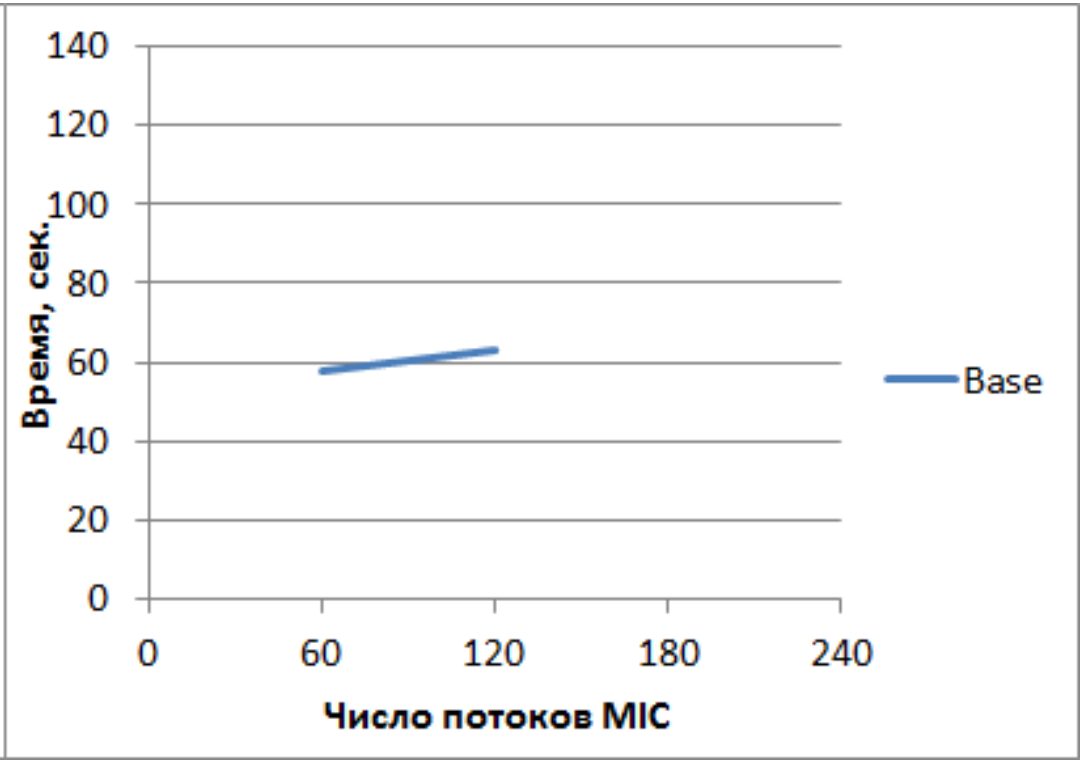
Прямой перенос: особенности программы

- Еще одна особенность алгоритма состоит в необходимости хранения траектории каждого отдельного фотона в процессе его трассировки.
- 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 на поток.
- Перед началом трассировки каждого отдельного фотона этот массив обнуляется.

Прямой перенос: результаты

- Программа на MIC работает в 2 раза медленнее, чем на CPU (8 ядер).
- Запуск программы в 240 потоков на MIC невозможен в силу недостатка памяти на сопроцессоре.





Оптимизация 1: обновление структуры данных для хранения траектории фотона…

- Результаты профилировки базовой версии с помощью Intel VTune Amplifier XE:

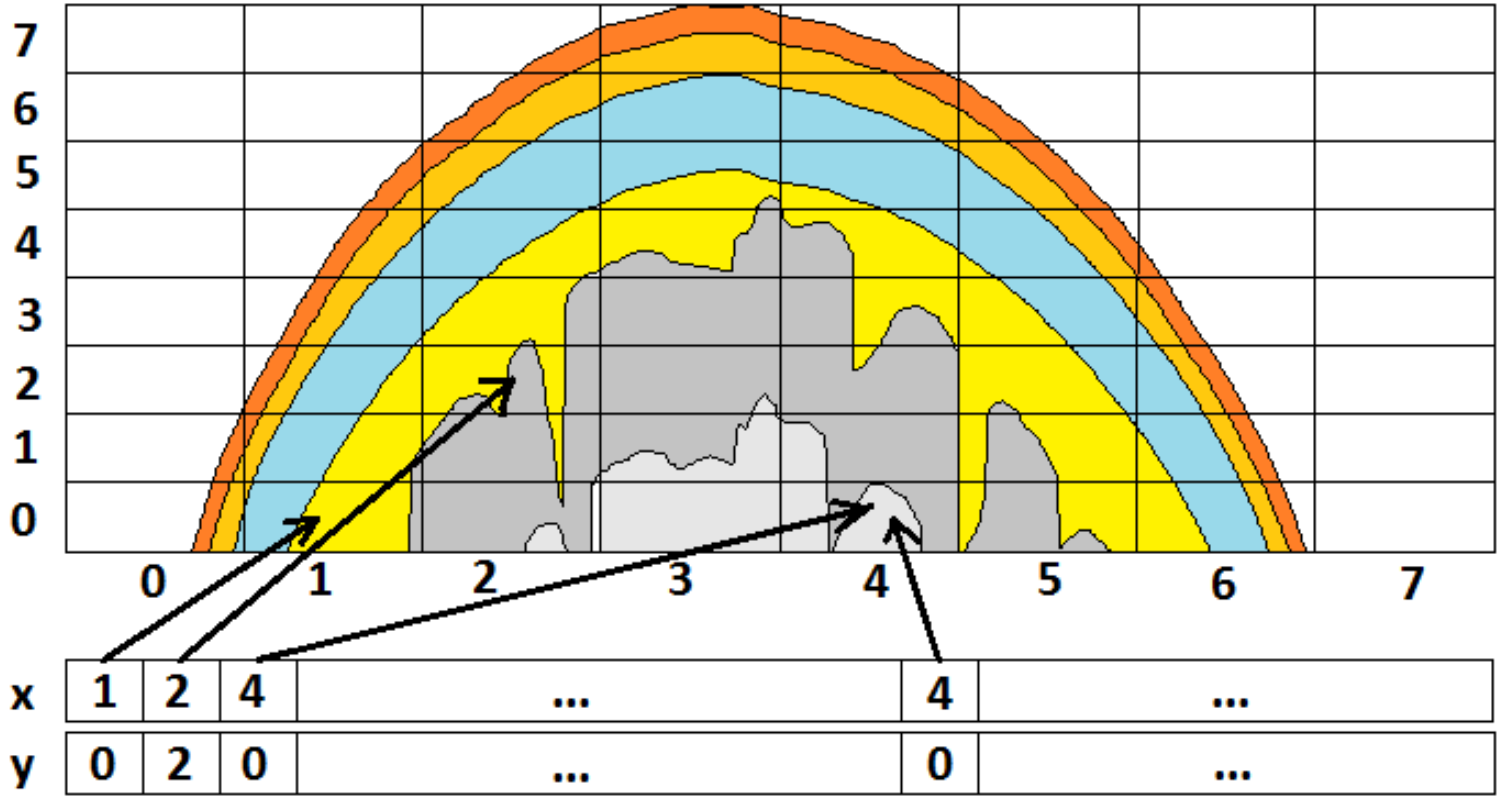
Function / Call Stack	CPU Time by Utilization Idle Poor Ok Ideal Over
_intel_memset	16.372s
GetIntersectionInfo	14.844s
ComputePhoton	9.391s
CrossBoundary	3.461s
ComputeCosineTheta	2.510s
func@0x78ea4ed0	2.487s
func@0x18014d340	1.644s
RtlFreeHeap	1.610s

Оптимизация 1: обновление структуры данных для хранения траектории фотона…

- Наибольшее время занимает функция memset(), которая используется в момент начала процесса трассировки фотона для обнуления памяти, предназначенной для хранения траектории его движения.
- Кроме необходимости обнуления памяти на каждой итерации, текущий 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).

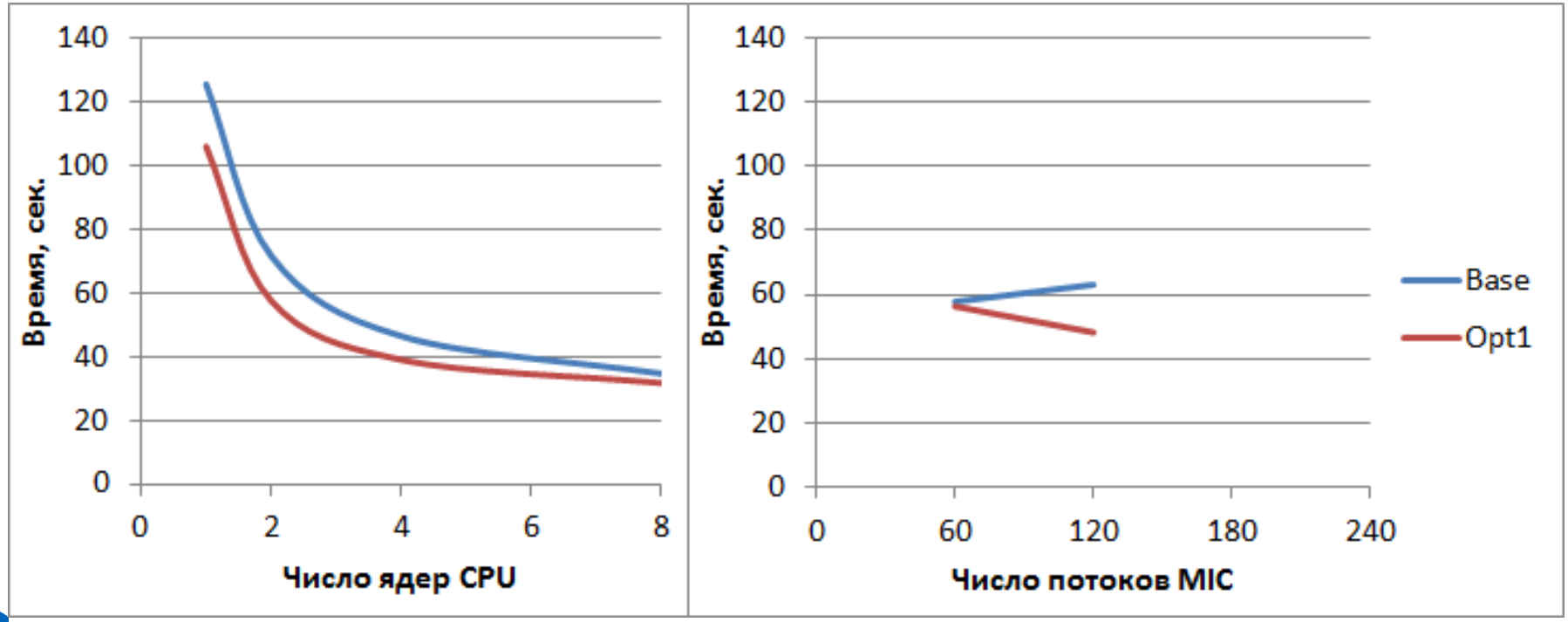
Оптимизация 1: обновление структуры данных для хранения траектории фотона…

- Для устранения описанных выше недостатков предлагается хранить не число посещений данной ячейки для всей сетки, а список координат посещенных ячеек:



Оптимизация 1: обновление структуры данных для хранения траектории фотона

- Ускорение на CPU
  - В 1 поток – 18%;
  - В 8 потоков – 9%.
- Ускорение на MIC в 120 потоков – 31%.

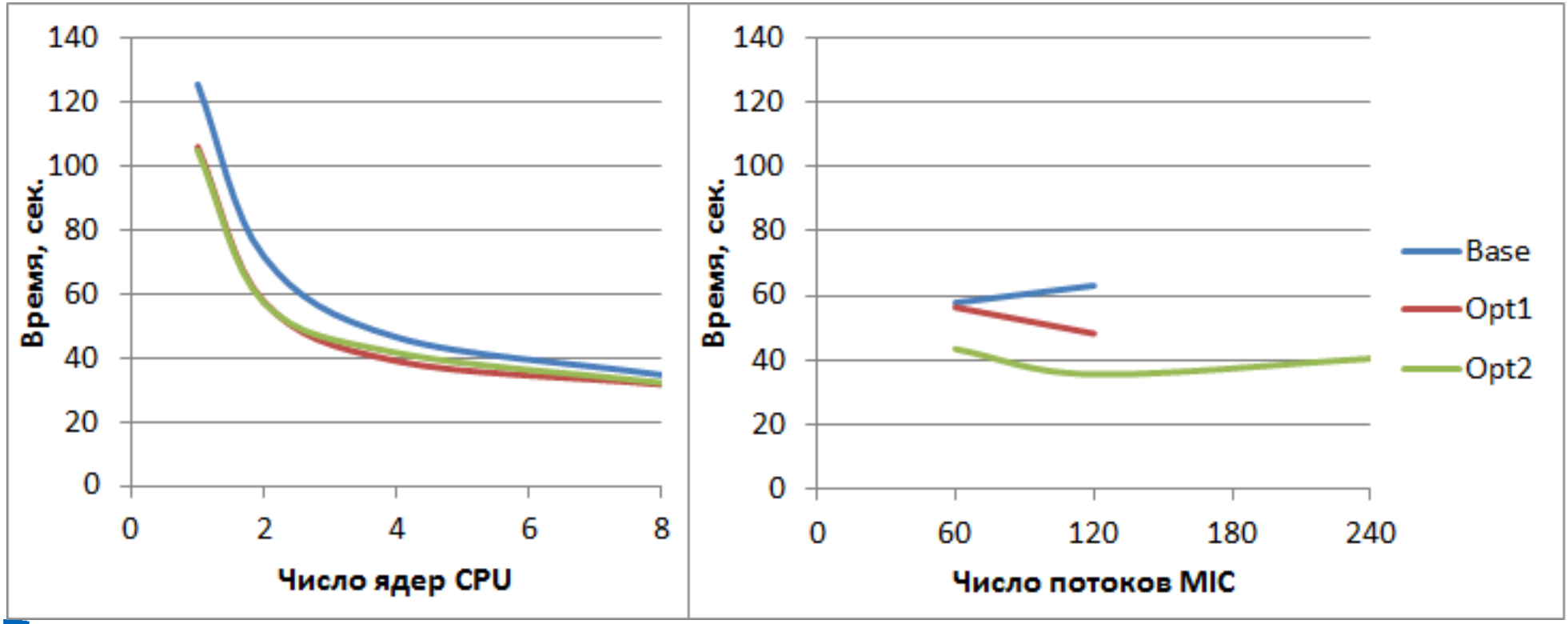


Оптимизация 2: отказ от использования дублирующих массивов…

- 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**.
- В нашем случае дублирование выполнено для двух типов результатов: общей фотонной карты траекторий и фотонных карт для каждого из детекторов:
  - Общая фотонная карта траекторий: размер – 4 МБ на поток; число одновременных операций доступа велико => **отказ от дублирования нецелесообразен**.
  - Фотонные карты для детекторов: размер – 40 МБ на поток; вероятность, что фотоны из разных потоков одновременно попадут на детектор мала => **отказ от дублирования целесообразен.**

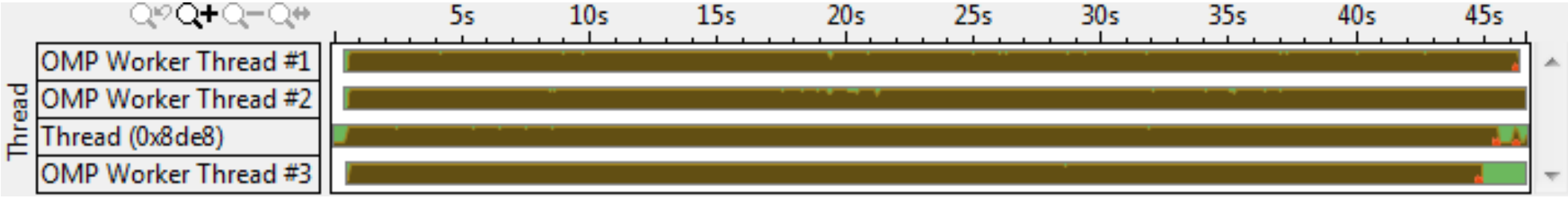
Оптимизация 2: отказ от использования дублирующих массивов

- Время работы на CPU практически не изменилось.
- Ускорение на MIC в 120 потоков – 35%.
- Появилась возможность запуска программы в 240 потоков.



Оптимизация 3: балансировка нагрузки…

- Профилировка программы с использованием Intel VTune Amplifier XE выявила еще одну особенность используемого алгоритма – разное время выполнения отдельных потоков:



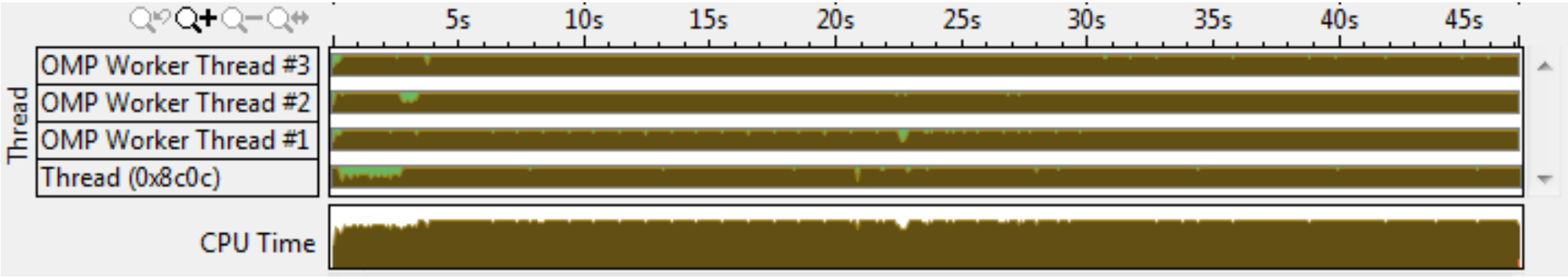
Оптимизация 3: балансировка нагрузки…

- Меняем статическую схему балансировки нагрузки на динамическую:

```
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]));
    }
    …
}
```

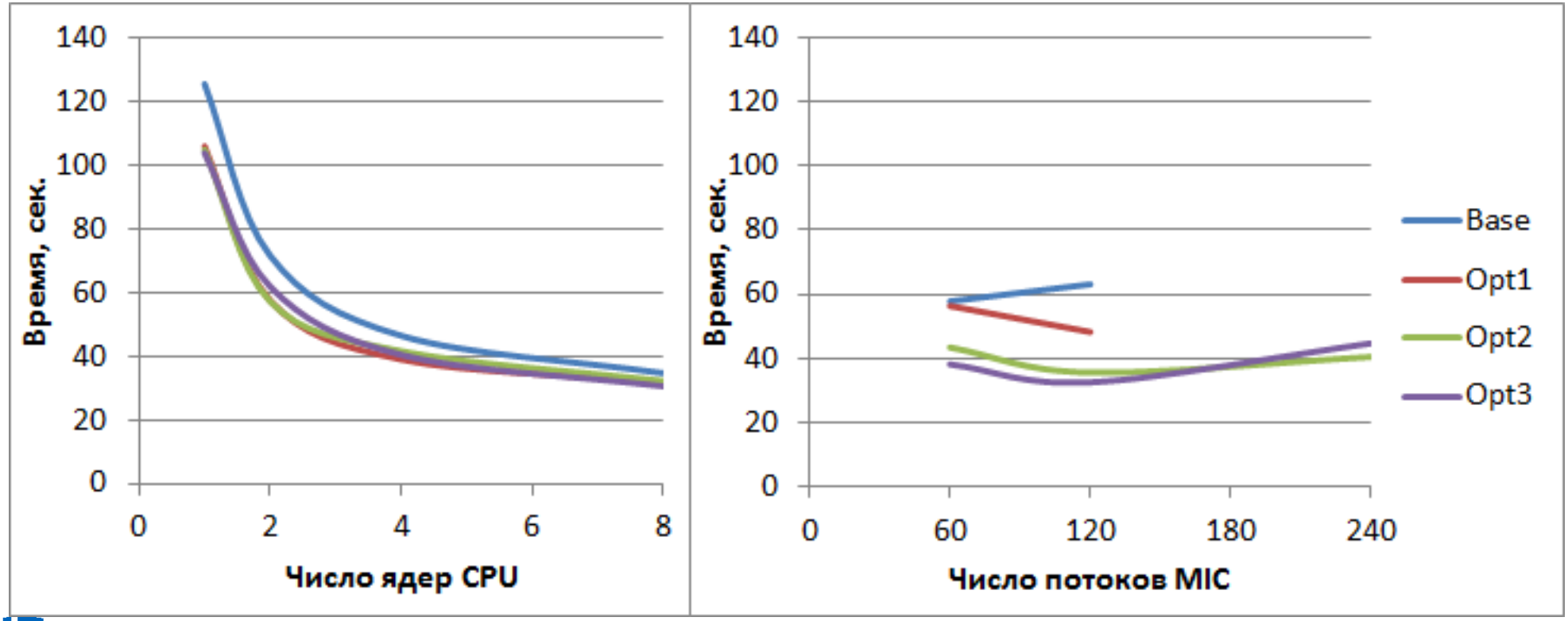
Оптимизация 3: балансировка нагрузки…

- Результаты профилировки после применения оптимизации:



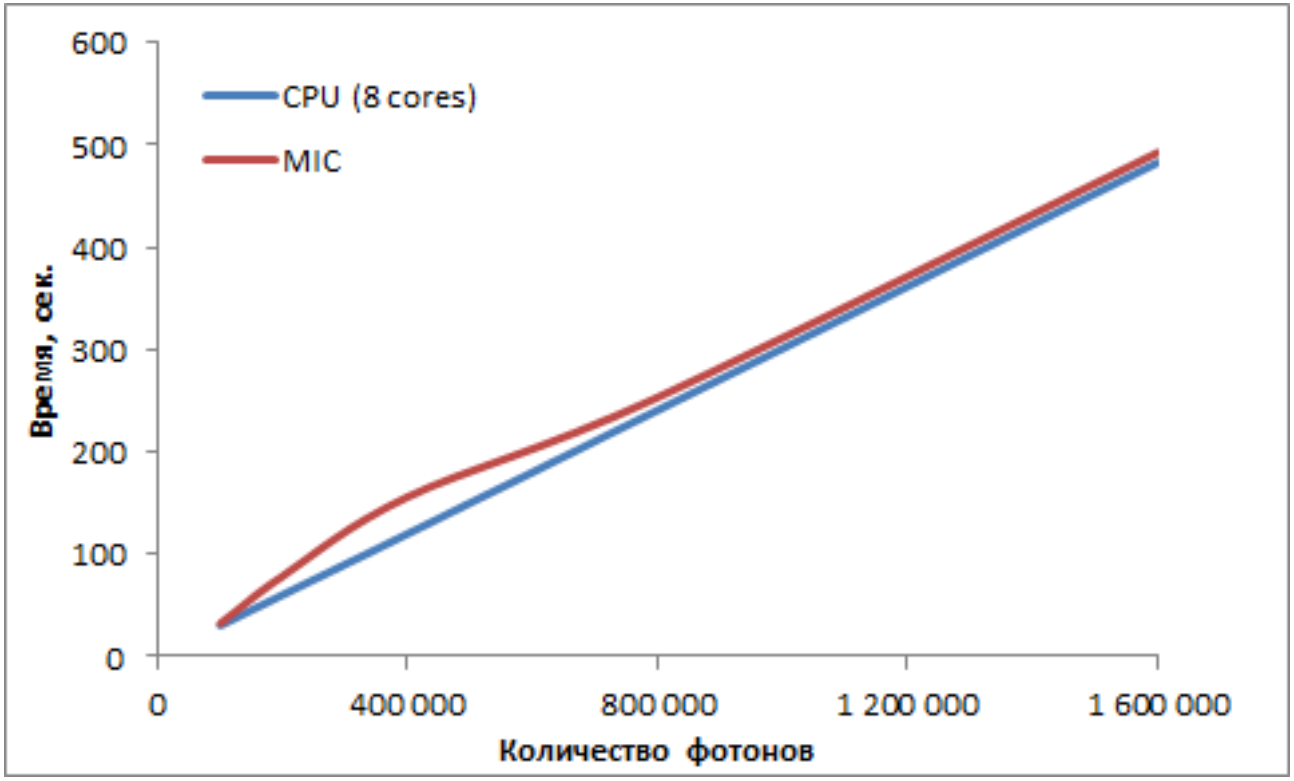
Оптимизация 3: балансировка нагрузки

- Время работы на CPU незначительно уменьшилось.
- Время работы на MIC уменьшилось на 9%.
- Минимальное время на MIC достигается на 120 потоках.



Общие результаты оптимизации

- Удалось повысить производительность программы для Intel Xeon Phi вдвое.
- Время работы программы на сопроцессоре Intel Xeon Phi примерно соответствует времени ее работы на CPU.



Дальнейшая оптимизация

- Для эффективной работы программы на Intel Xeon Phi требуется ее векторизация.
- Необходим анализ существующих алгоритмов поиска пересечений с целью выбора наиболее подходящего для работы на сопроцессоре, и его дальнейшая оптимизация.
- Отдельного обсуждения заслуживает выбор модели программирования на сопроцессоре, которую следует использовать для переноса.
- 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.

Дальнейшая оптимизация

- Для эффективной работы программы на Intel Xeon Phi требуется ее векторизация.
- Необходим анализ существующих алгоритмов поиска пересечений с целью выбора наиболее подходящего для работы на сопроцессоре, и его дальнейшая оптимизация.
- Отдельного обсуждения заслуживает выбор модели программирования на сопроцессоре, которую следует использовать для переноса.
- 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.

Литература

- Taflove A. Computational Electrodynamics: The Finite-Difference Time-Domain Method. -Artech House, London, 1995.
- 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.
- 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

Литература

- 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.
- 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.

Литература

- Intel Xeon Phi Coprocessor System Software Developers Guide, revision 2.03, 2012
- Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture, Volume 1a, January 29, 2013
- J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013. -432 p.
- Intel Developer Zone [http://software.intel.com/en-us/mic-developer]

Авторский коллектив

- Бастраков Сергей Иванович,
  ассистент кафедры Математического обеспечения ЭВМ
  факультета ВМК ННГУ.
  bastrakov@vmk.unn.ru
- Горшков Антон Валерьевич,
  ассистент кафедры Математического обеспечения ЭВМ
  факультета ВМК ННГУ.
  anton.v.gorshkov@gmail.com