ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 16 за 2013 г.

Березин А.В., Воронцов А.С., Марков М.Б., Садовничий Д.Н., Сысенко А.В.

Дифракция плоской электромагнитной волны: гибридное распараллеливание вычислений

Рекомендуемая форма библиографической ссылки: Дифракция плоской электромагнитной волны: гибридное распараллеливание вычислений / А.В.Березин [и др.] // Препринты ИПМ им. М.В.Келдыша. 2013. № 16. 20 с. URL: http://library.keldysh.ru/preprint.asp?id=2013-16

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

А.В. Березин, А.С. Воронцов, М.Б. Марков,
Д.Н. Садовничий, А.В. Сысенко

Дифракция плоской электромагнитной волны: гибридное распараллеливание

Москва — 2013

А.В. Березин, А.С. Воронцов, М.Б. Марков, Д.Н. Садовничий, А.В. Сысенко
Дифракция плоской электромагнитной волны: гибридное распараллеливание вычислений

Алгоритм решения сеточных уравнений явной разностной схемы для уравнений Максвелла сформулирован в векторизованном виде. Вычислительный модуль составлен на основе технологий Cuda и OpenMP с оптимизацией доступа к памяти графической платы. Векторизуемые операции выполняются на графических процессорах. Операции, которые требуют случайного доступа к памяти, выполняются центральным процессором в многопоточном режиме. Использование графических плат в узлах кластера обеспечило ускорение вычислений по сравнению с традиционной параллельной архитектурой. Вычисление ускорено в 11 раз для задач с малым объемом оперативной памяти и в 2-3 раза для остальных задач.

Ключевые слова: сеточное уравнение, векторизованное вычисление, центральный процессор, графический процессор, оперативная память, связанный запрос

A.V. Berezin, A.S. Vorontsov, M.B. Markov, D.N. Sadovnichii, A.V. Sysenko
Plane electromagnetic wave diffraction: hybrid paralleling

The algorithm for solving of implicit mesh Maxwell equations is vectorized. Calculation module is developed in Cuda and OpenMP technologies with optimization of access to graphic processor memory. Vector operations are carrying out at graphic processors. Operations with random access to memory are carrying out at central processor in multithread regime. Graphic processors in supercomputer nodes accelerate calculations in comparison with traditional architecture. Calculation is accelerated in 11 times for problems with small RAM volume and in 2-3 times for rest problems.

Key words: mesh equation, vectorized calculation, central processor, graphic processor, random access memory, coalescent request

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 11-01-00342-а

Оглавление

Введение 3
1 Постановка задачи 3
2 Организация вычислений при гибридном распараллеливании 5
3 Хранение данных в памяти центрального процессора 7
4 Вычисления на графическом процессоре 10
5 Хранение данных в памяти видеокарты 11
6 Особенности реализации вычислений на видеокарте 13
Заключение 20
Список использованных источников 20

Введение

Математическое моделирование является эффективным инструментом исследования взаимодействия электромагнитного излучения с техническими объектами.

Данная работа содержит описание программной реализации математической модели взаимодействия, основанной на численном решении уравнений Максвелла в трехмерной постановке. Работа продолжает исследование и основывается на следующих результатах относительно рассматриваемой модели. Задание эффективной плотности тока на одной из граней прямоугольной расчетной области и постановка специального вида граничных условий обеспечивает единственное решение начально-краевой задачи для уравнений Максвелла в виде плоской электромагнитной волны в отсутствии рассеивающего объекта. Начально-краевая задача с граничными «условиями излучения энергии», обеспечивающими совпадение направлений вектора Пойнтинга и внешней нормали к границе области, для разности полной и плоской волн описывает дифракцию плоской волны на ограниченном объекте и имеет единственное решение. Разностная аппроксимация такого граничного условия сохраняет второй порядок точности разностной схемы. Более подробно постановка задачи и разностная схема рассматриваются в работе [1].

Существуют практически важные задачи, в которых длина воздействующего импульса существенно превышает характерный размер объекта. Примером является задача исследования воздействия электромагнитного излучения молнии [2] на летательный аппарат в дальней зоне. Возникает проблема решения сеточных уравнений разностной схемы на большом числе временных слоев. Необходимо существенное ускорение вычислений.

В работе рассмотрены принципы распараллеливания вычислений при решении уравнений явной разностной схемы для уравнений Максвелла, ориентированные на суперкомпьютеры с гибридной архитектурой, которые используют графические процессоры для ускорения вычислений.

1 Постановка задачи

Рассмотрим начально-краевые задачи для уравнений Максвелла в ограниченной области $\Omega = \{x, y, z : x \in [x_{\min}, x_{\max}], y \in [y_{\min}, y_{\max}], z \in [z_{\min}, z_{\max}]\}$ с границей $\partial\Omega$:

$$\begin{gathered}
\operatorname{rot}\mathbf{H} = \varepsilon \frac{1}{c}\frac{\partial \mathbf{E}}{\partial t} + \frac{4\pi}{c}\left(\sigma\mathbf{E} + \mathbf{j}\right), \qquad \operatorname{rot}\mathbf{E} = -\mu\frac{1}{c}\frac{\partial \mathbf{H}}{\partial t}, \\
\frac{\partial \rho}{\partial t} + \operatorname{div}\left(\sigma\mathbf{E} + \mathbf{j}\right) = 0, \qquad \mathbf{E}\big|_{t=0} = \mathbf{H}\big|_{t=0} = \rho\big|_{t=0} = 0, \qquad (1) \\
E^y = H^z, E^z = -H^y + H^y_{pl} \text{ при } x = x_{\max},
\end{gathered}$$

$$E^y = -H^z,\, E^z = H^y - H^y_{pl} \text{ при } x = x_{\min},$$
$$E^z = H^x,\, E^x - E^x_{pl} = -H^z \quad \text{при} \quad y = y_{\max},$$
$$E^z = -H^x,\, E^x - E^x_{pl} = H^z \quad \text{при} \quad y = y_{\min},$$
$$E^x = H^y,\, E^y = -H^x \text{ при } z = z_{\max},\; E^x = -H^y,\, E^y = H^x \text{ при } z = z_{\min},$$

где: $\mathbf{E} = \mathbf{E}(t,\mathbf{r})$ – напряженность электрического поля, $\mathbf{H} = \mathbf{H}(t,\mathbf{r})$ – напряженность магнитного поля, t – лабораторное время, $\mathbf{r} = (x,y,z) \in \Omega$, $\rho = \rho(t,\mathbf{r})$ – плотность электрического заряда, $\mathbf{j} = \mathbf{j}(t,\mathbf{r}) = \varphi(t)(\delta(z),0,0)$ – плотность электрического тока, $\delta(z)$ – δ-функция Дирака, ставящая в соответствие функции основного пространства значение ее интеграла по плоской поверхности $z = 0$. Электрофизические материалы объекта заданы функциями координат $\varepsilon = \varepsilon(\mathbf{r})$ – диэлектрическая проницаемость, $\mu = \mu(\mathbf{r})$ – магнитная проницаемость, $\sigma = \sigma(\mathbf{r})$ – проводимость. Функции ε или μ могут быть отличны от единицы или σ отлична от нуля только в подобласти D области Ω, причем $D \subset \Omega$, $\operatorname{dist}(\partial\Omega, D) = d > 0$.

При вычислении граничных условий в задаче (1) используются функции $\mathbf{E}_{pl} = \mathbf{E}_{pl}(t,\mathbf{r})$ и $\mathbf{H}_{pl} = \mathbf{H}_{pl}(t,\mathbf{r})$, являющиеся соответственно электрической и магнитной компонентой падающей плоской волны. Они вычисляются решением следующей краевой задачи:

$$\begin{gathered}
\operatorname{rot}\mathbf{H}_{pl} = \frac{1}{c}\frac{\partial \mathbf{E}_{pl}}{\partial t} + \frac{4\pi}{c}\mathbf{j}, \qquad \operatorname{rot}\mathbf{E}_{pl} = -\frac{1}{c}\frac{\partial \mathbf{H}_{pl}}{\partial t}, \qquad \frac{\partial \rho_{pl}}{\partial t} = 0, \\
\mathbf{E}_{pl}\big|_{t=0} = \mathbf{H}_{pl}\big|_{t=0} = \rho_{pl}\big|_{t=0} = 0, \\
E^y_{pl} = E^z_{pl} = 0 \quad \text{при } x = x_{\min},\; x = x_{\max}, \\
H^x_{pl} = H^z_{pl} = 0 \text{ при } \; y = y_{\min},\; y = y_{\max}, \\
E^x_{pl} = -H^y_{pl} \quad \text{при } \; z = z_{\min}, \quad E^x_{pl} = H^y_{pl} \quad \text{при } \; z = z_{\max}.
\end{gathered} \tag{2}$$

Приведем также первое из разностных уравнений, на примере которого будет рассматриваться организация вычислений:

$$\begin{gathered}
\frac{H^z_{i+1/2,j+1/2,k,n+1/2} - H^z_{i+1/2,j-1/2,k+1/2}}{\delta y_j} - \frac{H^y_{i+1/2,j,k+1/2,n+1/2} - H^y_{i+1/2,j,k-1/2,n+1/2}}{\delta z_k} = \\
= \overset{\circ}{\varepsilon}_{i+1/2,j,k}\frac{E^x_{i+1/2,j,k,n+1} - E^x_{i+1/2,j,k,n}}{\Delta t_n} + \overset{\circ}{I}{}^x_{i+1/2,j,k,n+1/2}
\end{gathered}, \tag{3}$$

где:

$$\overset{\circ}{\mathbf{I}}_{i+1/2,j,k} = \frac{4\pi}{c}(\overset{\circ}{\sigma}_{i+1/2,j,k}\mathbf{E}_{i+1/2,j,k} + \overset{\circ}{\mathbf{J}}_{i+1/2,j,k}), \qquad \overset{\circ}{u}_{i+1/2,j,k} = \langle\langle u\rangle\rangle_{i+1/2,j,k},$$

$$\langle u\rangle_{i+1/2,j,k+1/2} = \frac{\Delta_{j-1}}{2\delta_j}u_{i+1/2,j-1/2,k-1/2} + \frac{\Delta_j}{2\delta_j}u_{i+1/2,j+1/2,k-1/2}.$$

Разностные уравнения рассматриваются на прямоугольной декартовой сетке:

$$x_{i+1} = x_i + \Delta_i; \quad i = 0,\dots,N_x - 1, \quad x_0 = x_{\min}, \quad x_{N_x} = x_{\max},$$
$$x_{i+1/2} = (x_i + x_{i+1})/2; \quad i = 0,\dots,N_x - 1, \quad x_{-1/2} = x_0, \quad x_{N_x+1/2} = x_{N_x},$$
$$\delta_i = x_{i+1/2} - x_{i-1/2}; \quad i = 0,\dots,N_x, \quad \delta_0 = \Delta_0/2, \quad \delta_{N_x} = \Delta_{N_x-1}/2.$$

Разностная сетка для переменных (y,z) вводится аналогично. Сетка по времени:

$$t_{n+1} = t_n + \Delta t_n; \quad n = 0,\dots,N_t - 1, \quad t_0 = t_{\min}, \quad t_{N_t} = t_{\max},$$
$$t_{n+1/2} = (t_n + t_{n+1})/2; \quad n = 0,\dots,N_t - 1,$$
$$\delta t_n = t_{n+1/2} - t_{n-1/2}; \quad n = 2,\dots,N_t - 1.$$

Разностное граничное условие на плоскости $y = y_{\min}$ имеет вид:

$$\frac{1}{2}\left(H^z_{i+1/2,-1/2,k,n+3/2} + H^z_{i+1/2,1/2,k,n+3/2}\right) = E^x_{i+1/2,0,k,n+1} - E^x_{pl\,i+1/2,0,k,n+1}. \qquad (4)$$

2 Организация вычислений при гибридном распараллеливании

Выразим значения компонент электрического поля с верхнего слоя в уравнении (3):

$$E^x_{i+1/2,j,k,n+1} = E^x_{i+1/2,j,k,n} - \frac{\Delta t_n}{\overset{\circ}{\varepsilon}_{i+1/2,j,k}} \times \left(\overset{\circ}{I}{}^x_{i+1/2,j,k,n+1/2} + \right.$$
$$\left. + \frac{H^z_{i+1/2,j+1/2,k,n+1/2} - H^z_{i+1/2,j-1/2,k+1/2}}{\delta y_j} - \frac{H^y_{i+1/2,j,k+1/2,n+1/2} - H^y_{i+1/2,j,k-1/2,n+1/2}}{\delta z_k} \right).$$

Вычисление на верхнем слое разбивается на определение электростатического поля:

$$E^{x\,st}_{i+1/2j,j,k,n+1} = E^{x}_{i+1/2,j,k,n} - \frac{\Delta t_n}{\overset{\circ}{\varepsilon}_{i+1/2,j,k}}\left(\overset{\circ}{I}{}^{x}_{i+1/2,j,k,n+1/2}\right)$$

и добавление к нему волновой компоненты:

$$E^{x}_{i+1/2,j,k,n+1} = E^{x\,st}_{i+1/2,j,k,n+1} - \frac{\Delta t_n}{\overset{\circ}{\varepsilon}_{i+1/2,j,k}} \times \left(\frac{H^{z}_{i+1/2,j+1/2,k,n+1/2} - H^{z}_{i+1/2,j-1/2,k+1/2}}{\delta y_j} - \right.$$
$$\left. - \frac{H^{y}_{i+1/2,j,k+1/2,n+1/2} - H^{y}_{i+1/2,j,k-1/2,n+1/2}}{\delta z_k} \right).$$

На каждом шаге по времени можно выделить следующие логические этапы вычислений:

1) Вычисление электростатического поля;
2) Вычисление электрического поля на верхнем шаге по времени;
3) Вычисление магнитного поля на верхнем шаге по времени;
4) Определение граничных условий;
5) Вывод результатов.

Распределение работы между центральным и графическим процессорами в целом подчиняется следующим соображениям. Задачи, которые требуют рассмотрения всей расчетной области, то есть вычисление электрического и магнитного поля, отправляются на графический процессор. Задачи, которые требуют меньшего объема вычислений, решаются с помощью центрального процессора, поскольку большинство из них требует случайного доступа к памяти. Исключение составляет случай, когда расчет целиком помещается в памяти графического процессора. В этом случае более эффективно выполнение всех операции на графической плате, несмотря на их неэффективность. Это позволяет уменьшить количество обменов данными между графическим и центральным процессорами.

Сетка по пространственным осям и все связанные с ней величины, такие как обратные величины шага, вычисляются один раз и хранятся в texture memory – области памяти видеокарты, оптимизирующей доступ за счет кэширования.

Помимо значений полей, для перехода на верхний слой по времени требуются значения диэлектрической и магнитной проницаемости, а также проводимости. Эти значения определяются тем, к какому материалу отнесена ячейка и усредняются для каждой компоненты магнитного поля. Такой подход неэффективен для графической карты. С одной стороны, он требует перенесения в видеопамять достаточно сложной структуры, в которой хранятся характеристики слоев физико-геометрической модели расчетной области. С другой стороны, требуется многократное обращение к памяти для вычисления усредненного

значения. При этом обращения происходят без явного устойчивого порядка, что понижает эффективность работы с памятью. Вместо этого в память графической карты записываются массивы, в которых хранятся усредненные (предварительно рассчитанные в памяти центрального процессора) значения характеристик для каждой компоненты электрического и магнитного поля. Несмотря на существенное увеличение объема данных, копируемых на графическую карту, такой подход повышает эффективность вычислений.

Определение граничных условий требует решения задачи (2) для плоской волны. Одномерная задача (2) оперирует с небольшим объемом данных, поэтому ее можно решать как на видеокарте, так и на центральном процессоре, в зависимости от того, на какой платформе реализуется вычисление граничных условий.

Вывод результатов реализуется только в памяти центрального процессора. Если расчет целиком помещается в память графической карты, то это приводит к необходимости обмена данными между графической картой и центральным процессором для моментов выдачи. Возможным решением было бы формирование массива выводимых данных с помощью графической карты и передача лишь его, однако это влечет две трудности. Во-первых, создание такого массива может потребовать случайного доступа к памяти, то есть не будет эффективным на графической карте. Во-вторых, поскольку программа поддерживает сложную систему описания выводимых данных, информация о них хранится в памяти в виде структуры, перенос которой в память графической карты достаточно сложен.

3 Хранение данных в памяти центрального процессора

В памяти центрального процессора хранится вся необходимая информация о расчете. Эта информация включает в себя разностную сетку (Grid), электрофизические характеристики модели (Model), массивы, в которых хранятся соответствующие значения полей (Fields), информацию о правой части (Currents) и выводимых результатах (Output). Для удобства эти объекты хранятся в виде экземпляров соответствующих классов.

```
class Grid {
public:
  int nx, ny, nz, nt;
  void make_arrays();
  float *dx, *dy, *dz, *dt;
  float *dx05,     *dy05,     *dz05;
  float *one_dx,   *one_dy,   *one_dz;
  float *one_dx05, *one_dy05, *one_dz05;
  float min_step;
```

```
  //Cuda part
  Dev_grid dev;
public:
  vector<double> x_, y_, z_;
  string description;

public:
  vector<double> t_;
  int index(int ix, int iy, int iz) {
  return ((ix)*(nY()+1) * (nZ()+1) + iy * (nZ()+1) + iz);
  };
  void InitCuda();
  unsigned int nX() {return nx;};
  unsigned int nY() {return ny;};
  unsigned int nZ() {return nz;};
  unsigned int nT() {return nt;};
  void read_data(string filename);
  void read_axe(fstream &grd_in, vector<double> *axe);
};

class Fields {
public:
  float *_ex, *_ey, *_ez, *_hx, *_hy, *_hz;
  float *d1sol_ex, *d1sol_hy;
  float *d1sol_ey, *d1sol_hx;
  Grid *_grid;
  Dev_fields dev;

public:
  void initialize(Grid *grid);
  void initialize_CUDA();
};

class Layer {
public:
  float eps;
  float mu;
  float sg;
};

class Model {
public:
```

```
  int *cel;
  float *eps_x, *eps_y, *eps_z;
  float *mu_x,  *mu_y,  *mu_z;
  float *sg_x,  *sg_y,  *sg_z;
  map<int, Layer> layers;
  Dev_model dev;

public:
  void read_cel(string filename, Grid *g);
  void read_layers(string flename);
  void prepare_epsmu(Grid *g);
};

class Currents {
public:
  int typex, typey;
  float emax_x,  emax_y;
  float omega_x, omega_y;
  void setCurrents(float t, float *jx, float* jy);
};

class Output {
public:
    int time;
    vector<int> output_moments;
     vector<int> savepoints;
    vector<float*> pointers;
    string filename;
    int output_ind;

public:
    void read_output(string filename, Fields *f, Grid *g);
    void make_output(float t);
};
```

Для чтения и обработки входных данных используется стандартная библиотека STL. Однако из-за того, что компилятор CUDA неправильно обрабатывает соответствующие объявления, вся информация, необходимая для работы графического процессора, хранится в отдельных структурах, не использующих стандартную библиотеку. Указатели на эти структуры хранятся в соответствующих полях классов с префиксом Dev. Эти структуры описаны в разделе, посвященном хранению данных на графическом процессоре.

В памяти центрального процессора поля хранятся в виде одномерных массивов размера (nX+1)×(nY+1)×(nZ +1). Это приводит к некоторому перерасходу памяти, поскольку некоторые элементы массивов не используются. Но единообразное хранение массивов в памяти позволяет единообразно обращаться к ним и, что более важно, использовать встроенные функции для копирования данных из памяти центрального процессора в память видеокарты с учетом особенностей хранения данных в памяти графического процессора. Таким образом, инициализация памяти для хранения полей выглядит одинаково для всех полей:

```
_ex = new float [(grid->nX()+1)*(grid->nY()+1)*(grid->nZ()+1)];
```

Также единообразное хранение всех полей позволяет обращаться к данному элементу с помощью одной и той же функции (которая реализована как один интерфейс index класса Grid, используемого для хранения сетки).

4 Вычисления на графическом процессоре

Графический процессор оптимизирован для векторных вычислений. Это означает применение одного и того же набора операций к разным наборам данных (архитектура SIMD (Single Instruction Multiple Data)) [3,4]. Эта архитектура хорошо подходит для расчета по явной разностной схеме. Для перехода на верхний слой в каждой точке сетки выполняется один и тот же набор операций. Конвейер графического процессора предоставляет практически неограниченный ресурс распараллеливания, так что узким местом становится работа с памятью. Важной особенностью работы графической платы является возможность делать «связанные» (coalescent) запросы к памяти. Если потоки вычисления обращаются к ячейкам, расположенным в памяти подряд, то запрос может быть выполнен за один такт работы независимо от количества потоков, делающих запросы. «Связывание» запроса происходит при выполнении ряда условий, наиболее важным из которых является правильное выравнивание в памяти графического процессора блока данных, к которому осуществляется запрос. Это означает, что адрес начала блока, к которому обращается нулевой поток выполнения, должен быть кратен 32 байтам.

В программе используются трехмерные массивы, которые хранятся в памяти в виде одной связной области. Адрес элемента массива с индексами i, j, k равен i*ny * nz + j * nz + k. Это означает, что элементы, имеющие близкие индексы по оси z, расположены в соседних ячейках памяти. Таким образом, если различные нити обрабатывают различные ячейки по оси z, они будут запрашивать элементы из соседних элементов массивов.

С другой стороны, такая форма хранения может приводить к следующему. Даже если первый элемент массива выровнен правильно, то есть его адрес кратен 32 байтам, то адрес начала следующих строк не будет выровнен, если соответствующие размеры массива не кратны 8, то есть количество байт памяти, занимаемое одной строкой массива, не кратно 32 байтам. Эту проблему

можно решить, если отводить память с помощью команды CudaMalloc3D, которая автоматически расширяет массив так, чтобы выравнивание каждой строки было правильным.

В некоторых случаях для вычисления производной по z производятся два запроса: один запрос к элементу с тем же индексом по z, что и в вычисляемом поле и один запрос к элементу с индексом, отличающимся на единицу (рис. 1). Такая ситуация является достаточно распространенной и современные видеокарты достаточно эффективно обрабатывают подобные запросы.

а) связываемый запрос к памяти при вычислении производной по y

б) несвязываемый запрос к памяти при вычислении производной по z

Рис. 1. Запрос к памяти

Графический процессор может использоваться в двух режимах. Первый режим обеспечивает минимальное количество обменов между ее памятью и оперативной памятью центрального процессора, но требует хранения в памяти графической карты всех необходимых массивов. Второй режим предполагает, что массивы полей не хранятся в памяти графической карты. Расчетная сетка разделяется на слои по оси X от x_i до x_{i+d}, а вычисления производятся последовательно для каждого из слоев. Размер слоя d выбирается так, чтобы памяти хватало на хранение всех необходимых массивов.

Детали реализации различаются для этих двух случаев, они будут описаны отдельно.

5 Хранение данных в памяти видеокарты

В памяти графического процессора хранятся данные о сетке, электрофизических характеристиках модели и вычисляемые поля. Все данные для удобства работы собраны в структуры Dev_grid, Dev_model, Dev_fields соответственно:

```
class Dev_grid {
public:
  float *one_dx_dev;
  float *one_dy_dev;
  float *one_dz_dev;
  float *one_dx05_dev;
  float *one_dy05_dev;
  float *one_dz05_dev;

  int ix_min, ix_max;
  int iy_min, iy_max;
  int iz_min, iz_max;
  size_t pitch, xsize, ysize;
};

class Dev_fields {
public:
  float *ex_dev, *ey_dev, *ez_dev,
        *hx_dev, *hy_dev, *hz_dev;
  float *d1sol_ex, *d1sol_hy;
  float *d1sol_ey, *d1sol_hx;
};

class Dev_model {
public:
  float *eps_x_dev, *eps_y_dev, *eps_z_dev;
  float *mu_x_dev,  *mu_y_dev,  *mu_z_dev;
  float *sg_x_dev,  *sg_y_dev,  *sg_z_dev;
};
```

Отличие от стандартной схемы состоит в том, что указатели хранятся не в виде CudaPitchedPtr, а в виде обычных указателей float*. Это связано с тем, что память для массивов выделяется единообразно. Поэтому их параметры, необходимые для выравнивания, зависят только от сетки и могут храниться вместе с ней. Доступ к элементам массива осуществляется с помощью макроса index, преобразующего тройку индексов (ix, iy, iz) в сдвиг адреса:

```
index(ix, iy, iz) =
   slicePitch* z + pitch * y + sizeof(float) * z;
```

В зависимости от того, в каком режиме используется графическая карта, в массивах хранится копия соответствующего массива в памяти центрального процессора или его часть.

6 Особенности реализации вычислений на видеокарте

Рассмотрим вначале ситуацию, когда все необходимые массивы могут быть размещены в памяти видеокарты. В этом случае алгоритм расчета может быть изображен в виде блок-схемы на рис. 2.

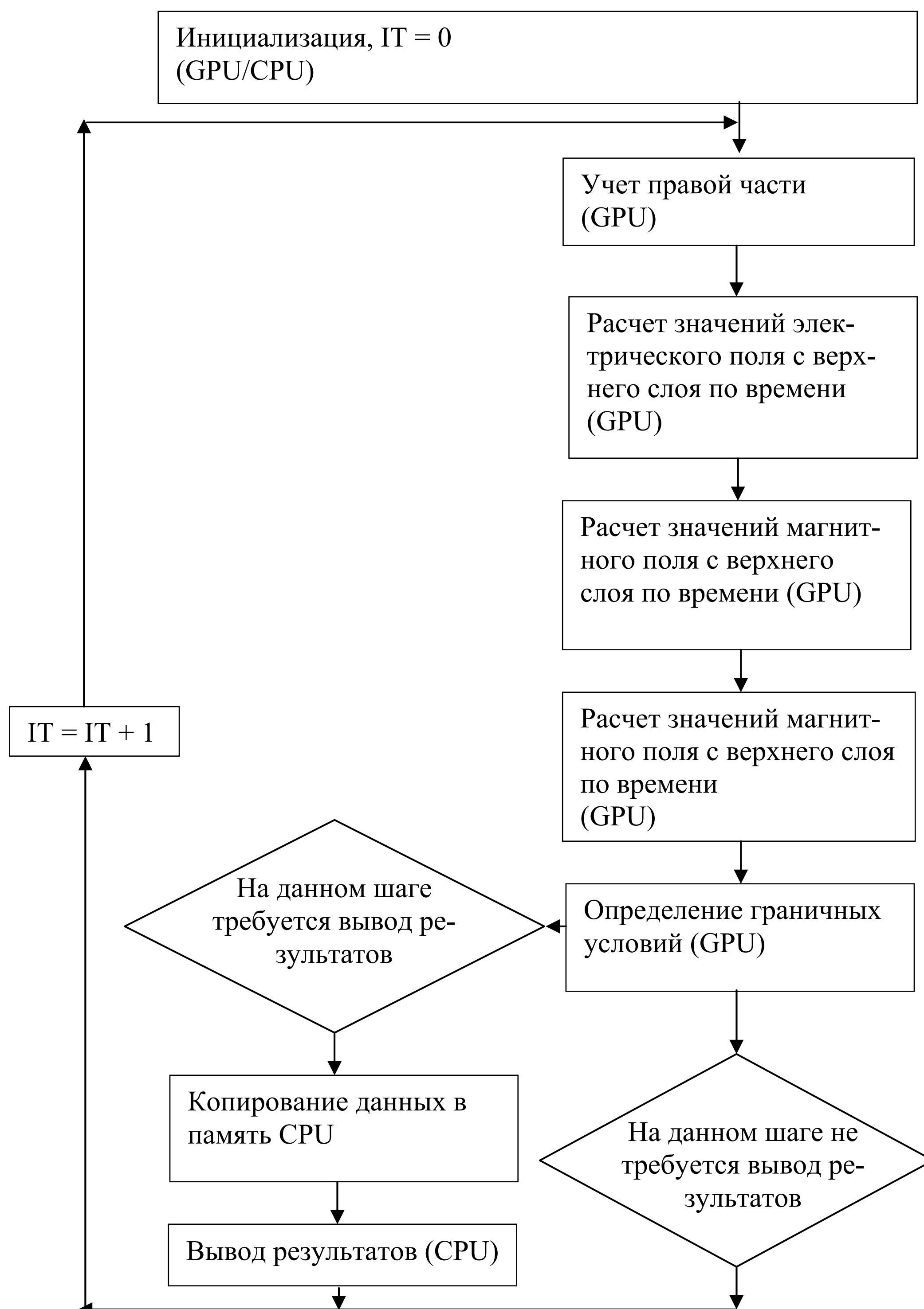

Рис. 2. Блок-схема алгоритма расчета для случая, когда все необходимые массивы помещаются в память видеокарты

Функция расчета полей сводится к вызову нескольких расчетных ядер на соответствующих сетках. Потоки, с помощью которых выполняется расчет на графическом процессоре, организованы в вычислительную сетку. Размеры этой сетки равны числу точек по осям X и Y. Для каждого элемента расчетной сетки запускается блок нитей, размер которого равен числу точек по оси Z. Таким образом, каждая нить выполняет расчет значения поля с верхнего слоя для одной точки. Нити из одного блока обращаются к ячейкам с близкими индексами по оси Z, то есть расположенным в соседних ячейках памяти. Процедуры вызова вычислительных ядер для расчета значений электрического и магнитного поля на верхнем слое выглядят так:

```
void cuda_step_e(Dev_fields f,
                 Dev_model m,
                 Dev_grid g,
                 float dct) {
  dim3 numBlocks_x(g.range-1, g.iy_max); // y and z
  cuda_step_ex<<<numBlocks_x, g.iz_max>>>(f, m, g, dct);
  dim3 numBlocks_y(g.range, g.iy_max-1); // y and z
  cuda_step_ey<<<numBlocks_y, g.iz_max>>>(f, m, g, dct);
  dim3 numBlocks_z(g.range, g.iy_max); // y and z
  cuda_step_ez<<<numBlocks_z, g.iz_max-1>>>(f, m, g, dct);
}

void cuda_step_h(Dev_fields f,
                 Dev_model m,
                 Dev_grid g,
                 float dct) {
  dim3 numBlocks_x(g.range, g.iy_max-1); // y and z
  cuda_step_hx<<<numBlocks_x, g.iz_max-1>>>(f, m, g, dct);
  dim3 numBlocks_y(g.range-1, g.iy_max); // y and z
  cuda_step_hy<<<numBlocks_y, g.iz_max-1>>>(f, m, g, dct);
  dim3 numBlocks_z(g.range-1, g.iy_max-1); // y and z
  cuda_step_hz<<<numBlocks_z, g.iz_max>>>(f, m, g, dct);
}
```

Для примера приведем одно расчетное ядро для поля Ex:

```
__global__ void cuda_step_ex(Dev_fields f,
                             Dev_model m,
                             Dev_grid g,
                             float dct) {
  int ix, iy, iz;
  float roth;
```

```
    int ix_min, iy_min, iz_min;
    ix_min = g.ix_min+1;
    iy_min = 0;
    iz_min = 0;

    ix = blockIdx.x  + ix_min;
    iy = blockIdx.y  + iy_min;
    iz = threadIdx.x + iz_min;
    roth = Dy(f.hz_dev) * g.one_dy_dev[iy] –
           Dz(f.hy_dev) * g.one_dz_dev[iz];
                        // (rot H)_x  = dHz/dy - dHy/dz
    f.ex_dev[ind(ix, iy, iz)] =  (roth  * dct +
      m.eps_x_dev[ind(ix, iy, iz)] *
      f.ex_dev[ind(ix, iy, iz)]) /
      (m.eps_x_dev[ind(ix, iy, iz)]
      + m.sg_x_dev[ind(ix,iy,iz)] * dct);
}
```

Здесь ind(ix, iy, iz) – макрос, преобразующий тройку индексов в соответствующий сдвиг адреса с учетом выравнивания массивов. Остальные расчетные ядра отличаются именами массивов и индексами.

Три ядра, вычисляющие три компоненты электрического поля, могут вычисляться на графической плате параллельно, поскольку каждое из них изменяет значения переменных лишь в соответствующем массиве. Затем, перед вызовом ядер, вычисляющих магнитное поле, выполняется синхронизация потока выполнения программы с видеокартой. Параллельно запускаются три ядра, вычисляющие магнитное поле.

Вычисление граничных условий осуществляется аналогично, но здесь не приходится заботиться о «связывании» запросов к памяти, поскольку данные в любом случае оказываются неудачно расположенными в памяти. Вызов вычислительных ядер:

```
void cuda_boundary(Dev_fields f, Dev_model m, Dev_grid g){
  cuda_boundary_x<<<g.iy_max, g.iz_max>>>(f, m, g);
  cuda_boundary_y<<<g.ix_max, g.iz_max>>>(f, m, g);
  cuda_boundary_z<<<g.ix_max, g.iy_max>>>(f, m, g);
}
```

Для примера приведем вычислительное ядро, вычисляющее граничные условия для границ, перпендикулярных оси *x*.

```
__global__ void cuda_boundary_x(Dev_fields f,
                                Dev_model m,
                                Dev_grid g) {
    int iy = blockIdx.x;
```

```
  int iz = threadIdx.x;
  int ix;
  float edm = 1.0f;
  ix = 0;
  f.hz_dev[ind(ix, iy, iz)] = - edm *
    (f.ey_dev[ind(ix, iy, iz)] - f.d1sol_ey[iz]);
  f.hy_dev[ind(ix, iy, iz)] = edm *
    f.ez_dev[ind(ix, iy, iz)]+f.d1sol_hy[iz];
  ix = g.ix_max;
  f.hz_dev[ind(ix, iy, iz)] = edm *
    (f.ey_dev[ind(ix-1, iy, iz)] - f.d1sol_ey[iz]);
  f.hy_dev[ind(ix, iy, iz)] = - edm *
    f.ez_dev[ind(ix-1, iy, iz)] + f.d1sol_hy[iz];
};
```

Учет внешнего тока также реализован в виде вызова ядра на графической плате. Значения тока для поляризации вдоль оси X и вдоль оси Y при этом передаются в качестве параметров:

```
void cuda_apply_j(Dev_fields f,
                  Dev_grid g,
                  float dctjx,
                  float dctjy) {
  cuda_apply_j_<<<g.ix_max, g.iy_max>>>(f,g,dctjx, dctjy);
}

__global__ void cuda_apply_j_(Dev_fields f,
                              Dev_grid g,
                              float dctjx,
                              float dctjy) {
  int ix = blockIdx.x;
  int iy = threadIdx.x;
  int iz = 0;
  f.ex_dev[ind(ix, iy, iz)] =
    ( dctjx + f.ex_dev[ind(ix, iy, iz)]);
  f.ey_dev[ind(ix, iy, iz)] =
    ( dctjy + f.ey_dev[ind(ix, iy, iz)]);
}
```

В ситуации, когда расчетная сетка целиком не помещается в памяти видеокарты, алгоритм незначительно меняется: изменяется распределение работы между центральным процессором и графической платой и несколько меняется процедура вызова расчетных ядер. Блок-схема алгоритма в этом случае приведена на рис. 3.

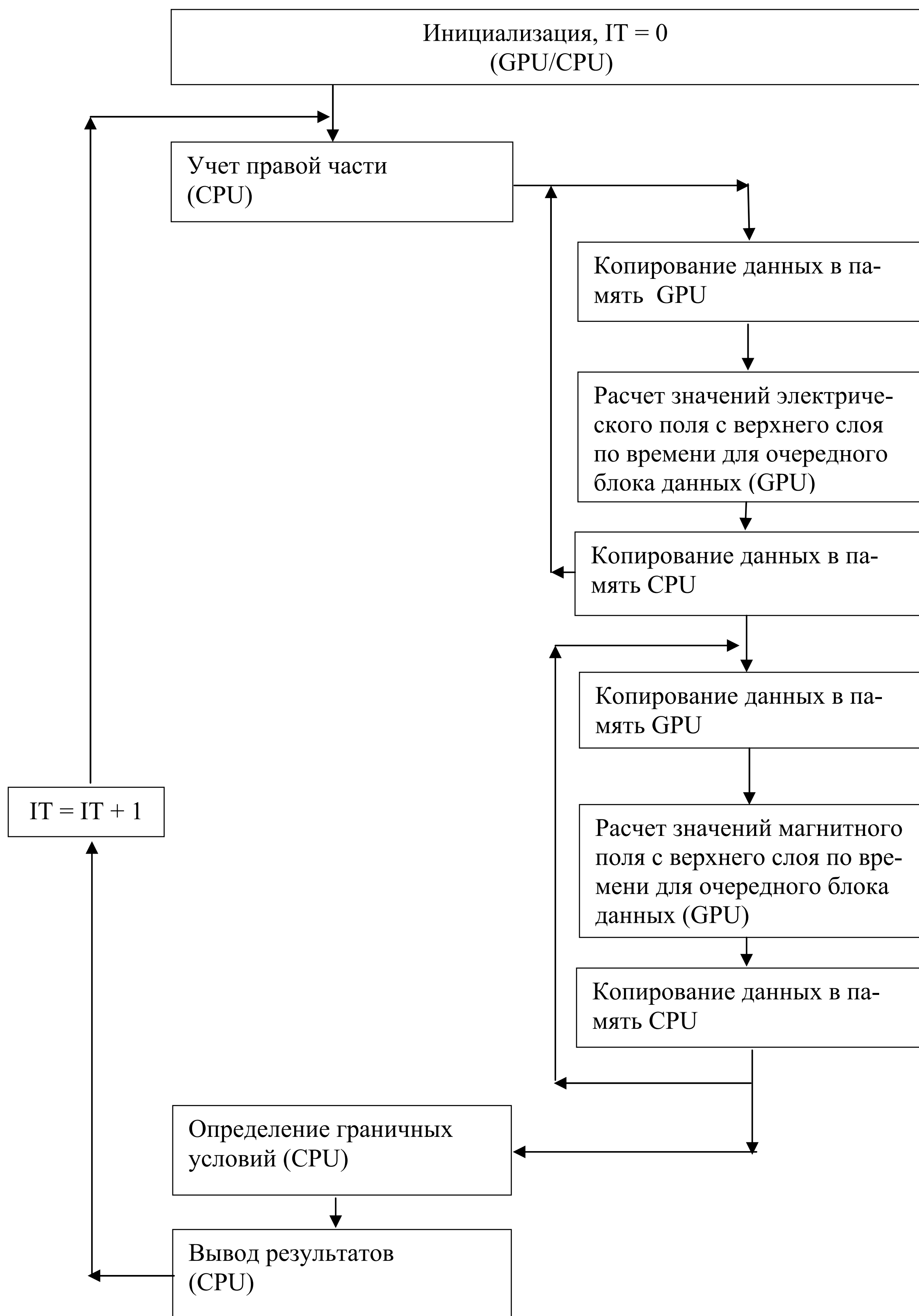

Рис. 3. Блок-схема алгоритма расчета для случая, когда все необходимые массивы не помещаются в память видеокарты

Вызов расчетных ядер теперь осуществляется соответствующей процедурой, осуществляющей копирование данных на видеокарту, расчет и возврат результата в память центрального процессора.

```
void cuda_step_e_part(Dev_fields f,
                      Dev_model m,
                      Dev_grid g,
                      float  dct,
                      int x_min,
                      int x_max) {
  int range = x_max – x_min + 1;
  cuda_upload_e_part(f, *f_host, g, ix_min, ix_max);
  cuda_upload_h_part(f, *f_host, g, ix_min, ix_max);
  dim3 numBlocks_x(range-1, g.iy_max); // y and z
  cuda_step_ex<<<numBlocks_x, g.iz_max>>>(f, m, g, dct);
  dim3 numBlocks_y(range, g.iy_max-1); // y and z
  cuda_step_ey<<<numBlocks_y, g.iz_max>>>(f, m, g, dct);
  dim3 numBlocks_z(range, g.iy_max); // y and z
  cuda_step_ez<<<numBlocks_z, g.iz_max-1>>>(f, m, g, dct);
  cuda_download_e_part(f, f_host, g, ix_min, ix_max);
}
```

Эта процедура вызывается в цикле для необходимых значений ix_min и ix_max. Функция cuda_upload_e_part помимо значений электрического поля из нужного диапазона ячеек копирует значения диэлектрической проницаемости и проводимости для этого диапазона ячеек.

Вычисление значений магнитного поля происходит аналогично, соответствующая функция cuda_upload_h_part помимо значений магнитного поля копирует в память видеокарты значения магнитной проницаемости для всех ячеек. Для хранения этих значений используется та же область памяти, которая при вычислении электрического поля использовалась для хранения диэлектрической проницаемости.

Вычисление электростатического поля и граничных условий выполняются на центральном процессоре, при этом работа делится между ядрами с помощью технологии OpenMP.

```
void boundaries_radiative_h_xmin(Fields *f,
                                 Grid *g,
                                 Model *m) {
      if (g->id != 0) return;
      int ix = 0;
#pragma omp parallel for
      for (int iy = 1; iy < g->nY(); iy++) {
```

```
        for (int iz = 0; iz < g->nZ(); iz++) {
          float edm =
           sqrt(m->eps_y[g->index(ix+1, iy, iz)] /
           m->mu_z[g->index(ix, iy, iz)]);
          f->_hz[g->index(ix, iy, iz)] =
           - edm * (f->_ey[g->index(ix, iy, iz)] –
             f->d1sol_ey[iz]);
        }
      }

//        Hy = Ez
#pragma omp parallel for
      for (int iy = 0; iy < g->nY(); iy++) {
        for (int iz = 1; iz < g->nZ(); iz++) {
          float edm = sqrt(m->eps_z[g->index(ix, iy, iz)]
           / m->mu_y[g->index(ix+1, iy, iz)]);
          f->_hy[g->index(ix, iy, iz)] =
           edm * f->_ez[g->index(ix, iy, iz)] +
           f->d1sol_hy[iz];
        }
      }
}

void apply_jx(Fields *f, Grid *g, float dctjx) {
  int ix_max, iy_max, iz_max;
  int ix_min, iy_min, iz_min;

  ix_max = g->nX();
  iy_max = g->nY();
  iz_max = g->nZ();

  ix_min = 1;
  iy_min = 0;
  iz_min = 0;
  int iz = 0;
  #pragma omp parallel for
  for (int ix = ix_min; ix < ix_max; ix++) {
    for (int iy = iy_min; iy < iy_max; iy++) {
      f->_ex[g->index(ix, iy, iz)] =
         ( dctjx + f->_ex[g- >index(ix, iy, iz)]);
    }
  }
}
```

Заключение

Эффективность использования графической карты проверялась на одном узле суперкомпьютера ГВК К-100. Использование одной графической карты сравнивалось с 11 центральными процессорами, объединенными с помощью технологии OpenMP. Графическая плата увеличила производительность в 10-15 раз в расчете, поместившемся целиком в ее оперативной памяти. Если на каждом шаге требуется многократно производить обмен данными между центральным процессором и графической картой, то производительность увеличивается в 2-3 раза.

Список использованных источников

1 Березин А.В. и др. Дифракция плоской электромагнитной волны: постановка задачи // Препринты ИПМ им.М.В.Келдыша. – 2013. – № 15. – 18 с. — URL: http://library.keldysh.ru/preprint.asp?id=2013-15

2 Базелян Э. М., Райзер Ю П. Физика молнии и молниезащиты. – М.: ФИЗМАТЛИТ, 2001.

3 NVidia CUDA C Programming guide, http://docs.nvidia.com/cuda

4 J. Sanders and E. Kandrot, CUDA by example. An introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.