Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Воронежский государственный университет»

Физический факультет
Кафедра электроники

Расчёт электромагнитных полей методом конечных разностей во временной области на графических процессорах

Выпускная квалификационная бакалаврская работа

Направление 03.03.03 «Радиофизика»
Профиль «Информационные системы и технологии»

Обучающийся _____ Н. А. Нагорный __.__.____
подпись
Руководитель _____ П. А. Кретов __.__.____
подпись

Воронеж, 2017

Содержание

Введение

Метод конечных разностей во временной области (англ. *Finite Difference Time Domain*, *FDTD*) является одним из самых популярных и часто используемых на практике методов вычислительной электродинамики: он позволяет исследовать поведение системы сразу на широком диапазоне частот и обладает высоким потенциалом для параллелизации. Последнее обстоятельство делает особенно предпочтительным для проведения FDTD-симуляции использование не традиционных микропроцессоров с последовательной архитектурой, а специализированных высокопараллельных вычислительных устройств. Одними из наиболее доступных устройств такого рода являются *графические процессорные устройства* (*ГПУ*).

Большинство доступных на рынке пакетов программ для электромагнитной симуляции включают в себя высокопроизводительные реализации FDTD как для *центральных процессорных устройств* (*ЦПУ*), так и для видеопроцессоров, однако имеющиеся проекты с открытым исходным кодом поддержки ГПУ не имеют. Проприетарные реализации часто недоступны из-за высокой стоимости соответствующих программных пакетов. Кроме того, применение закрытого программного обеспечения затрудняет изучение и непосредственную модификацию самого метода. Ввиду вышеперечисленных факторов достаточно актуальной задачей является разработка свободной реализации метода, эффективно использующей ресурсы ГПУ для выполнения симуляции.

В рамках данной работы была разработана программа, реализующая метод конечных разностей во временной области, способная эффективно использовать ресурсы ГПУ и позволяющая симулировать возбуждение электромагнитных колебаний при помощи точечных источников напряжения. При помощи этой программы была произведена тестовая симуляция

распространения гармонического сигнала, излучаемого тонким симметричным вибратором в замкнутом счётном объёме. Также была разработана референсная ЦПУ-реализация, и произведено сравнение производительности расчётов с использованием ЦПУ и ГПУ.

1. Метод конечных разностей во временной области

Метод конечных разностей во временной области относится к общему классу сеточных методов решения дифференциальных уравнений. В его рамках уравнения Максвелла подвергаются дискретизации с использованием центрально-разностной аппроксимации как по временной, так и по пространственным координатам. Полученные конечно-разностные уравнения решаются программными или аппаратными средствами в каждой точке временной сетки, причём компоненты вектора напряжённости магнитного поля смещены на половину шага дискретизации относительно компонент вектора напряженности электрического поля вдоль каждой оси. Расчёт полей в ячейках сетки повторяется до тех пор, пока не будет получено решение поставленной задачи в интересующем промежутке времени.

Существует также большое количество расширений метода, наиболее популярным из которых являются разнообразные поглощающие граничные условия, преобразование ближнего поля в дальнее, моделирование сосредоточенных активных и пассивных элементов. В данной работе были реализованы только базовый алгоритм FDTD, сосредоточенный резистивный источник напряжения и поглощающие граничные условия PML.

1.1. Принципы работы метода

Рассмотрим систему из четырёх векторных уравнений Максвелла, записанных в системе единиц СИ:

$$\begin{cases} \operatorname{rot} \vec{E} = \sigma_H \vec{H} - \frac{\partial \vec{B}}{\partial t}, \\ \operatorname{rot} \vec{H} = \sigma_E \vec{E} + \frac{\partial \vec{D}}{\partial t}, \\ \operatorname{div} \vec{D} = \rho, \\ \operatorname{div} \vec{B} = 0. \end{cases} \tag{1.1}$$

В системе (1.1) использованы следующие обозначения:

$\vec{E}$ — напряжённость электрического поля,

$\vec{H}$ — напряжённость магнитного поля,

$\vec{B}$ — магнитная индукция ($\vec{B} = \mu\mu_0\vec{H}$),

$\vec{D}$ — электрическая индукция ($\vec{D} = \varepsilon\varepsilon_0\vec{E}$),

ε — относительная диэлектрическая проницаемость,

μ — относительная магнитная проницаемость,

σ_E — удельная электрическая проводимость,

σ_H — удельная магнитная проводимость,

ρ — плотность стороннего электрического заряда,

ε_0 — электрическая постоянная ($\varepsilon_0 \approx 8,8542$ Ф/м),

μ_0 — магнитная постоянная ($\mu_0 = 4\pi \cdot 10^{-7}$ Гн/м).

Рассматривая уравнения Максвелла, можно заметить, что изменение значения вектора индукции электрического поля во времени (частная производная вектора $\vec{D}$ по времени) зависит от изменения магнитного поля в пространстве (ротор вектора $\vec{H}$). Поэтому значение вектора электрического поля в каждой точке пространства в определённый момент времени будет зависеть от значения этого же вектора в предыдущий момент времени и от изменения распределения вектора напряжённости магнитного поля в пространстве. Аналогичным образом значение вектора $\vec{H}$ в определённой точке и в определённый момент времени зависит от своего значения в предыдущий момент времени и от изменения распределения вектора $\vec{E}$ в пространстве.

Исходя из этих требований, на время выполнения каждой итерации алгоритма нам необходимо хранить в памяти компьютера значения векторов $\vec{E}$ и $\vec{H}$ в предыдущий момент времени. Под *итерацией* здесь и далее подразумевается расчёт значения вектора $\vec{E}$ или $\vec{H}$ в определённой точке в определённый момент времени.

Расчёт трёхмерных электромагнитных структур сильно усложняет вычисление ротора полей. В связи с этим американским математиком китайского происхождения Кейном И была разработана схема расчёта, в которой электрическая и магнитная сетки сдвинуты относительно друг друга так, что магнитное поле по каждой оси рассчитывается в точках, расположенных ровно между точками, в которых рассчитывается электрическое поле, и наоборот. Эта схема сейчас известна как *сетка И*. Её графическая модель представлена на рис. 1.

1.2. Входные параметры задачи

На практике для численного решения какой-либо задачи электродинамики необходимо задать счётный объём. *Счётный объём* (или *счётная область*) — это та область пространства, в пределах которой выполняется численное моделирование, и осуществляется непосредственный расчёт электромагнитных полей.

Счётная область разбивается на ячейки при помощи сетки И; в каждом узле сетки задаются значения электрических и магнитных проницаемостей и проводимостей. Чаще всего в качестве базового материала счётного объёма рассматривают вакуум (или воздух), в отдельных узлах сетки помещаются металлические или диэлектрические структуры. Тем не менее, алгоритм вполне позволяет задать произвольные значения вышеперечисленных величин для каждой точки объёма.

Кроме того, для моделирования реальных задач необходимы источники поля: некоторая структура, способная создавать электромагнитное возмущение внутри счётного объёма. Так, среди прочего, FDTD позволяет симулировать возбуждение электромагнитных колебаний при помощи падающей электромагнитной волны или точечного источника напряжения.

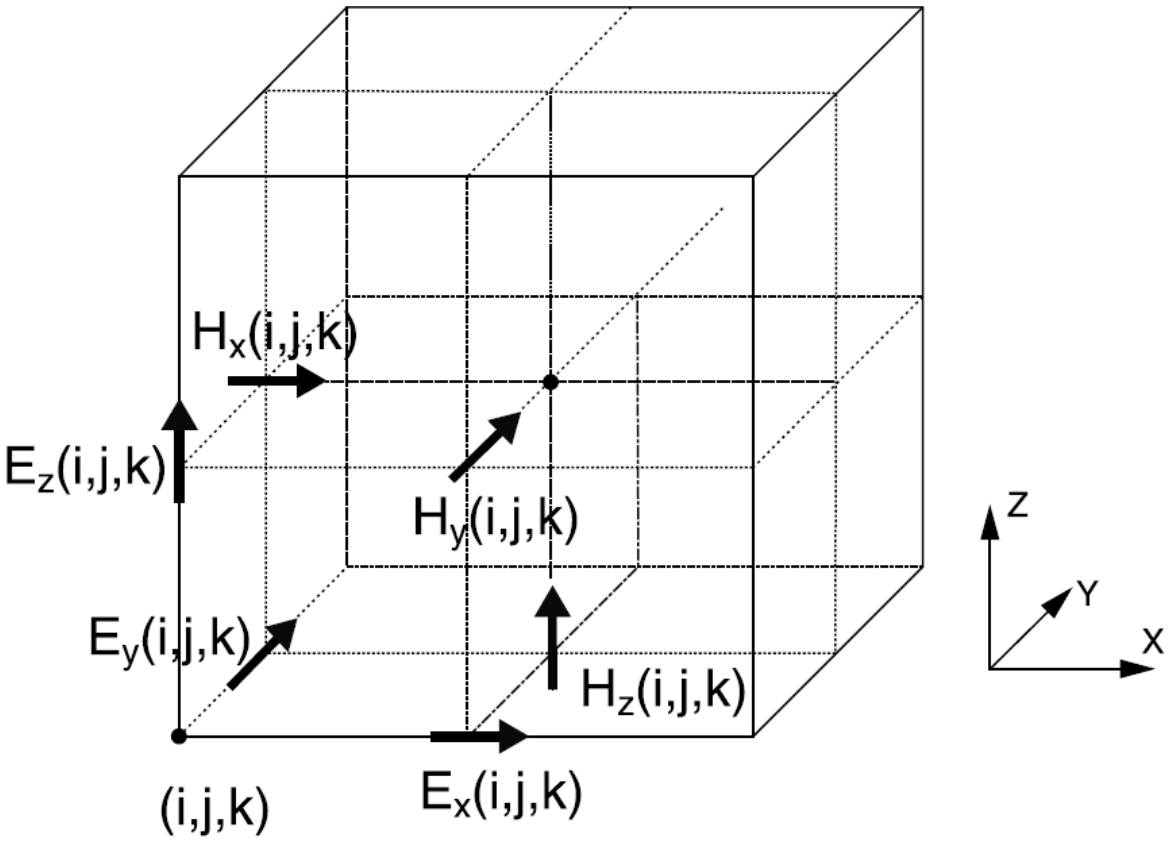

Рисунок 1 – Поля в ячейке сетки И.

1.3. Особенности метода

Отличительной особенностью метода конечных разностей во временной области является его относительная простота. К достоинствам метода также можно отнести возможность создавать анимированные изображения распространения волновых процессов в счётном объеме, что может быть очень полезно для понимания происходящих в модели процессов и позволяет удостовериться в её корректности.

Основной недостаток метода — обязательность разбиения счётного объёма на ячейки сетки И, причём величина шага дискретизации по пространственным координатам должна быть достаточно малой по сравнению с наименьшей длиной волны, встречающейся в конкретной задаче. Кроме того, эта величина ограничивает детализацию распределения материалов в пространстве, поэтому может оказаться, что счётный объём должен быть разделен на очень большое число ячеек, что влечёт за собой большие затраты памяти и увеличивает время моделирования.

Ещё одним недостатком FDTD является обязательность вычисления параметров поля в каждой точке счётного объёма. Так, при необходимости найти поле на некотором отдалении от источника придётся производить расчёт во всех точках, что находятся между источником и интересующей точкой.

К тому же, счётная область обязательно должна быть конечной. В большинстве случаев это достигается использованием искусственных граничных условий, но они, как правило, вызывают дополнительные искажения.

1.4. Базовые уравнения

Как уже было сказано, метод FDTD предполагает введение сетки, которая в простейшем случае представляет собой обыкновенный трёхмерный массив, в котором хранятся векторы полей и пространственная структура. Процедура

расчёта заключается в поочерёдном обращении всех элементов этого массива в порядке возрастания индексов и последующем перевычислении его элементов по приведенным ниже формулам.

Выражения для компонент магнитного поля:

$$H_x\Big|_{i,j,k}^{n+1/2} = H_x\Big|_{i,j,k}^{n-1/2} - \frac{\frac{\Delta t}{\mu}}{1+\frac{\sigma_H \Delta t}{2\mu}}\left[\frac{E_z|_{i,j+1,k}^{n} - E_z|_{i,j,k}^{n}}{\Delta y} - \frac{E_y|_{i,j,k+1}^{n} - E_y|_{i,j,k}^{n}}{\Delta z}\right],$$

$$H_y\Big|_{i,j,k}^{n+1/2} = H_y\Big|_{i,j,k}^{n-1/2} - \frac{\frac{\Delta t}{\mu}}{1+\frac{\sigma_H \Delta t}{2\mu}}\left[\frac{E_x|_{i,j,k+1}^{n} - E_x|_{i,j,k}^{n}}{\Delta z} - \frac{E_z|_{i+1,j,k}^{n} - E_z|_{i,j,k}^{n}}{\Delta x}\right],$$

$$H_z\Big|_{i,j,k}^{n+1/2} = H_z\Big|_{i,j,k}^{n-1/2} - \frac{\frac{\Delta t}{\mu}}{1+\frac{\sigma_H \Delta t}{2\mu}}\left[\frac{E_y|_{i+1,j,k}^{n} - E_y|_{i,j,k}^{n}}{\Delta x} - \frac{E_x|_{i,j+1,k}^{n} - E_x|_{i,j,k}^{n}}{\Delta y}\right],$$

Выражения для компонент электрического поля:

$$E_x\Big|_{i,j,k}^{n+1} = \frac{1-\frac{\sigma_E \Delta t}{2\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}} E_x\Big|_{i,j,k}^{n} + \frac{\frac{\Delta t}{\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}}\left[\frac{H_z|_{i,j,k}^{n+1/2} - H_z|_{i,j-1,k}^{n+1/2}}{\Delta y} - \frac{H_y|_{i,j,k}^{n+1/2} - H_y|_{i,j,k-1}^{n+1/2}}{\Delta z}\right],$$

$$E_y\Big|_{i,j,k}^{n+1} = \frac{1-\frac{\sigma_E \Delta t}{2\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}} E_y\Big|_{i,j,k}^{n} + \frac{\frac{\Delta t}{\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}}\left[\frac{H_x|_{i,j,k}^{n+1/2} - H_x|_{i,j,k-1}^{n+1/2}}{\Delta z} - \frac{H_z|_{i,j,k}^{n+1/2} - H_z|_{i-1,j,k}^{n+1/2}}{\Delta x}\right],$$

$$E_z\Big|_{i,j,k}^{n+1} = \frac{1-\frac{\sigma_E \Delta t}{2\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}} E_z\Big|_{i,j,k}^{n} + \frac{\frac{\Delta t}{\varepsilon}}{1+\frac{\sigma_E \Delta t}{2\varepsilon}}\left[\frac{H_y|_{i,j,k}^{n+1/2} - H_y|_{i-1,j,k}^{n+1/2}}{\Delta x} - \frac{H_x|_{i,j,k}^{n+1/2} - H_x|_{i,j-1,k}^{n+1/2}}{\Delta y}\right].$$

Выше приведены формулы, позволяющие вычислить каждую из компонент векторов напряжённости электрического и магнитного полей. В этих формулах используются следующие обозначения:

σ_E — удельная электрическая проводимость материала в ячейке сетки;

σ_H — удельная магнитная проводимость материала в ячейке сетки;

ε — абсолютная диэлектрическая проницаемость материала;

μ — абсолютная магнитная проницаемость материала;

Δt — шаг дискретизации по времени;

Δx, Δy, Δz — шаги дискретизации по пространственным координатам.

Необходимо заметить, что величина Δt определяет частотные характеристики метода: наивысшая частота в спектре сигналов, распространение которых моделируется, не должна превышать $f_{max} = \frac{1}{\Delta t}$.

Также величина Δt должна удовлетворять условию Куранта:

$$\Delta t < \frac{1}{c\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}}$$

Пересчёт значений компонент выполняется «на месте», то есть рассчитанное в каждый последующий момент времени значение помещается в ту же ячейку сетки И, в которой находилось значение для предыдущего момента. Это позволяет несколько снизить требования к оперативной памяти, предъявляемые методом.

1.5. Точечный резистивный источник

Возможность моделировать сосредоточенные элементы вводится дополнительно, при этом границы применения метода существенно расширяются. В частности, точечный генератор напряжения с заданным внутренним сопротивлением оказывается весьма удобной моделью источника питания.

Достоинством этого дополнения FDTD является то, что вид уравнений меняется весьма незначительно. Если источник ориентирован вдоль оси Z, то в системе уравнений Максвелла (1.1) изменится только одна составляющая векторного уравнения для ротора вектора магнитной напряжённости:

$$(\mathrm{rot}\,\vec{H})_z = \varepsilon\frac{\partial E_z}{\partial t} + \sigma E_z + \frac{I_L}{\Delta x \Delta y}, \tag{1.2}$$

где I_L — ток через источник.

Подвергнув уравнение (1.2) действию конечно-разностной схемы и положив $\sigma_E = 0$, получим следующую формулу:

$$E_z\Big|_{i,j,k}^{n+1} = E_z\Big|_{i,j,k}^{n} + \frac{\Delta t}{\varepsilon|_{i,j,k}} \left[\frac{H_y|_{i,j,k}^{n+1} - H_y|_{i-1,j,k}^{n+1}}{\Delta x} - \frac{H_x|_{i,j,k}^{n+1} - H_x|_{i,j-1,k}^{n+1}}{\Delta y} \right] - \\ - \frac{I_L|^{n+1} \Delta t}{\varepsilon|_{i,j,k} \Delta x \Delta y}. \quad (1.3)$$

Согласно закону Ома, для рассматриваемого сосредоточенного генератора напряжения с внутренним сопротивлением R будет иметь место равенство

$$I_L\Big|^{n+1} = \frac{\Delta z}{2R} \left(E_z\Big|_{i,j,k}^{n+1} + E_z\Big|_{i,j,k}^{n} \right) + \frac{U_s|^{n+1}}{R}, \quad (1.4)$$

где U_s — генерируемое источником напряжение.

После подстановки (1.4) в (1.3) получим простое уравнение для E_z:

$$E_z\Big|_{i,j,k}^{n+1} = C_E\Big|_{i,j,k} E_z\Big|_{i,j,k}^{n} + C_H\Big|_{i,j,k} \left[\frac{H_y|_{i,j,k}^{n+1} - H_y|_{i-1,j,k}^{n+1}}{\Delta x} - \right. \\ \left. - \frac{H_x|_{i,j,k}^{n+1} - H_x|_{i,j-1,k}^{n+1}}{\Delta y} + \frac{U_s|^{n+1}}{R \Delta x \Delta y} \right],$$

где C_E и C_H определяются по следующим формулам:

$$C_E\Big|_{i,j,k} = \frac{1 - \dfrac{\Delta t \Delta z}{2R\, \varepsilon|_{i,j,k} \Delta x \Delta y}}{1 + \dfrac{\Delta t \Delta z}{2R\, \varepsilon|_{i,j,k} \Delta x \Delta y}}, \qquad C_H\Big|_{i,j,k} = \frac{\dfrac{\Delta t}{\varepsilon|_{i,j,k}}}{1 + \dfrac{\Delta t \Delta z}{2R\, \varepsilon|_{i,j,k} \Delta x \Delta y}}.$$

Формулы для прочих компонент остаются неизменными.

2. Неспециализированные вычисления на графических процессорах

Прообразом первых графических процессоров, появившихся в 90-е годы XX века, были специализированные чипы аркадных автоматов [1]. Их использование было обусловлено малыми объёмами оперативной памяти, что не позволяло хранить в ней кадры перед отправкой на устройство видеовывода. В дальнейшем разделение вычислений на графические и неграфические лишь усугубилось, что оказало существенное влияние на архитектуру современных компьютеров.

В начале XXI века графические процессоры получили поддержку шейдеров и возможность работы с числами с плавающей запятой. Это событие положило начало ряду экспериментов с организацией неграфических параллельных расчётов на графических процессорах. При помощи графических API данные передавались в виде текстур, а расчётные программы — в виде шейдеров [2]. Таким образом учёные начали производить вычисления, связанные с матрицами и векторами, на ГПУ. Первой программой, выполнившейся заметно быстрее на ГПУ, чем на ЦПУ, стала реализация LU-разложения (2005) [3].

С увеличением популярности использования ГПУ для научных расчётов начали формироваться идеи фреймворков общего назначения, позволяющих отойти от графической парадигмы работы с данными и отказаться от использования OpenGL или DirectX. Такими фреймворками впоследствии стали технологии NVIDIA CUDA и OpenCL.

2.1. Краткий обзор технологии NVIDIA CUDA

Первоначальная версия NVIDIA CUDA SDK была представлена 15 февраля 2007 г. В основе NVIDIA CUDA API лежит диалект языка C++ [4].

Для компиляции программ NVIDIA CUDA SDK предлагает специализированный компилятор nvcc. При этом код программ разделяется на host-часть, выполняющуюся ЦПУ, и device-часть, выполняющуюся графическим процессором. В результате получаются как минимум два объектных файла, готовых к сборке в конечный исполняемый файл в любой среде программирования [5].

По сравнению с использовавшимся ранее подходом к организации вычислений общего назначения посредством возможностей графических API, архитектура CUDA имеет ряд преимуществ:

- использование диалекта языка C++, что позволяет упростить процесс изучения архитектуры;
- полная аппаратная поддержка целочисленных и побитовых операций;
- разделяемая между потоками память размером в 16 Кбайт может быть использована под организованный пользователем кэш с более широкой полосой пропускания, чем при выборке из обычных текстур.

2.2. Краткий обзор технологии OpenCL

OpenCL — фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических и центральных процессорах, а также ППВМ. В OpenCL входят язык программирования, который базируется на стандарте C99, и интерфейс программирования приложений [6].

Основной задачей проекта OpenCL является создание и поддержка открытого стандарта, позволяющего создавать универсальные программы для параллельных вычислений на различных процессорах и вычислительные машины, использующие несколько процессоров различных архитектур одновременно.

2.3. Иерархия потоков выполнения в NVIDIA CUDA

Как упоминалось ранее, одной из особенностей написания программ с использованием технологии NVIDIA CUDA является разделение всего программного кода на host- и device-части. Для этого используются спецификаторы функций:

- `__host__` — означает, что данный код предназначен для центрального процессора (используется по умолчанию);
- `__device__` — означает, что данный код работает на видеокарте;
- `__global__` — особый спецификатор для так называемых функций-ядер (kernel), которые запускаются с центрального процессора, а работают на видеокарте.

Остановимся подробнее на *функциях-ядрах*. Их отличие от обычных функций языка C++ заключается в том, что при вызове они выполняются N раз параллельно в N потоках выполнения. При этом количество потоков выполнения, которые можно создать на ГПУ, практически не ограничено.

Для организации работы со столь большим количеством потоков используется иерархическая структура: потоки объединяются в варпы (warp), варпы, в свою очередь, — в блоки (block), а блоки составляют сетку (grid).

Варп — это минимальная независимая от других единица выполнения кода. Размер варпа всегда равен 32 потокам. Эти потоки всегда выполняются физически синхронно. *Блок* — это автономная группа потоков. Взаимодействия потоков между блоками невозможны.

Вызов функции-ядра осуществляется следующим образом:

```
kernel_name<<<grid_size, block_size>>>(arguments);
```

В тройных угловых скобках в участке программного кода выше указываются размеры сетки и блока.

Примером может послужить программа для перемножения матриц: она иллюстрирует параллельное выполнения потоков с помощью многократного повторения операции сложения с разными числами.

```
__global__ void MatAdd(
  float A[N][N], float B[N][N], float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
  ...
  int numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  ...
}
```

Во фрагменте исходного кода выше `threadIdx` — трёхкомпонентный вектор, хранящий координаты потока в блоке (рис. 2). Таким образом, видим: имеется один блок размеров $N \times N$ и три массива той же размерности. Каждому потоку ставится в соответствие по одному элементу каждой из трёх матриц. Следовательно, все потоки покрывают все элементы матриц. Блок выполняет перемножение матриц как одну операцию благодаря параллельной работе потоков, работающих всего лишь с тремя элементами матриц по отдельности.

2.4. Иерархия памяти в NVIDIA CUDA

Выделяют следующие виды памяти видеокарты:

– глобальная память — аналог оперативной памяти ПК и основной вид памяти, используемый для обмена данными между ЦПУ и ГПУ;

– разделяемая память — управляемый программно L1-кэш (64 Кбайт на каждый потоковый мультипроцессор);
– регистровая локальная память — память, из которой выделяются регистры для каждого потока (64 000 регистров на каждый потоковый мультипроцессор);
– дополнительная локальная память — участок глобальной памяти, выделяемый потоку при нехватке регистров;
– константная память — память, модификация которой возможна только с ЦПУ;
– текстурная память — интерфейс чтения глобальной памяти с использованием специфических для текстур операций [5].

Для каждого потока выделяется локальная память (регистровая и, опционально, дополнительная). Для каждого блока выделяется разделяемая память, доступ к которой имеют все потоки блока. Разделяемая память выделена, пока существует блок. Все существующие потоки имеют доступ к глобальной памяти. Также каждый поток имеет доступ к чтению константной и текстурной памяти. Глобальная, константная и текстурная память не перевыделяются на протяжении работы программы, т. о. разные функции-ядра могут работать с этими видами памяти (рис. 3).

2.5. Иерархия потоков выполнения в OpenCL

Иерархическая модель потоков выполнения OpenCL концептуально не отличается от своего аналога от компании NVIDIA.

Аналогом потоков в OpenCL являются рабочие элементы (work-item). Они объединяются в рабочие группы (work-group), причём рабочие элементы, принадлежащие разным рабочим группам, не могут взаимодействовать.

На рис. 4 изображён пример *двухмерного пространства индексов* (2D-range index space), являющегося аналогом сетки в NVIDIA CUDA. В общем случае пространство индексов является N-мерным.

Данное пространство индексов разделено на 16 рабочих групп, имеющих собственные координаты и размер 8×8. Рабочие элементы имеют два типа координат: локальные (относительно рабочей группы) и глобальные (относительно сетки целиком).

По аналогии с NVIDIA CUDA используются функции-ядра, выполняющиеся на OpenCL-устройстве.

2.6. Иерархия памяти в OpenCL

Так как хост-машина и OpenCL-устройство не имеют общего адресного пространства, взаимодействие между памятью хоста и памятью OpenCL-устройства происходит посредством использования различных областей памяти (рис. 5).

Глобальная память — это участок памяти, к которому все рабочие элементы, группы, а также хост, имеют полный доступ (чтение и запись). Эта область памяти может быть выделена только хостом.

Константная память — это участок глобальной памяти, остающийся нетронутым на протяжении выполнения функции-ядра. Рабочие группы могут только читать данные из этой области, хост же имеет к ней полный доступ.

Локальная память — это место обмена данными между рабочими элементами рабочей группы. Все элементы имеют полный доступ к этой области.

Внутренняя память — это память, принадлежащая конкретному рабочему элементу.

В большинстве случаев память хоста и память OpenCL-устройства работают независимо друг от друга. Соответственно, имеется специфика обмена данными между ними: необходимо перемещать данные из памяти хоста в глобальную память, затем в локальную, и обратно.

2.7. Специфика работы с видеопамятью

Перед организацией параллельных вычислений следует ознакомиться с некоторыми приёмами оптимизации, которые помогут сохранить производительность, выигранную от использования ГПУ. Наиболее широко распространённым и лёгким в освоении приёмом является согласованный доступ к памяти, аналогичный выравниванию памяти при работе ЦПУ с оперативной памятью.

Глобальная память физически расположена на device-устройстве, доступ к которой происходит посредством 32-, 64- и 128-байтных транзакций. Необходимое условие для их существования — обращение к выровненным участкам памяти. Таковыми являются участки, адреса первых элементов которых кратны 32, 64 и 128 байтам соответственно.

Когда варп выполняет инструкцию, обращающуюся к глобальной памяти, обращения каждого потока внутри варпа сливаются в одну или несколько транзакций, в зависимости от размера слова, запрашиваемого каждым потоком, и разброса адресов, к которым обращаются потоки.

Команды работы с глобальной памятью поддерживают чтение и запись слов размером 1, 2, 4, 8 и 16 байтов. Любой запрос к глобальной памяти возможен только в случае, если размер запрашиваемых данных равен 1, 2, 4, 8 или 16 байтам, и они выровнены (т. е. их адреса кратны размеру).

Затем одиночные запросы к глобальной памяти объединяются в транзакции. Например, запросы 32 потоков одного варпа, обращающиеся к четырём последовательно располагающимся байтам каждый, объединятся в транзакцию размером 128 байтов в том случае, если адрес, к которому обращается первый поток варпа, кратен 128 байтам.

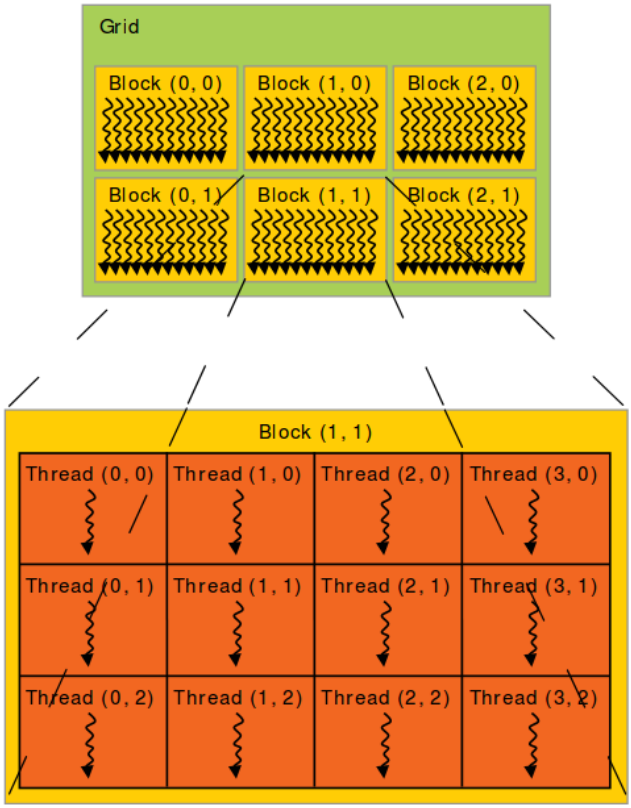

Рисунок 2 – Иерархическая структура потоков выполнения.

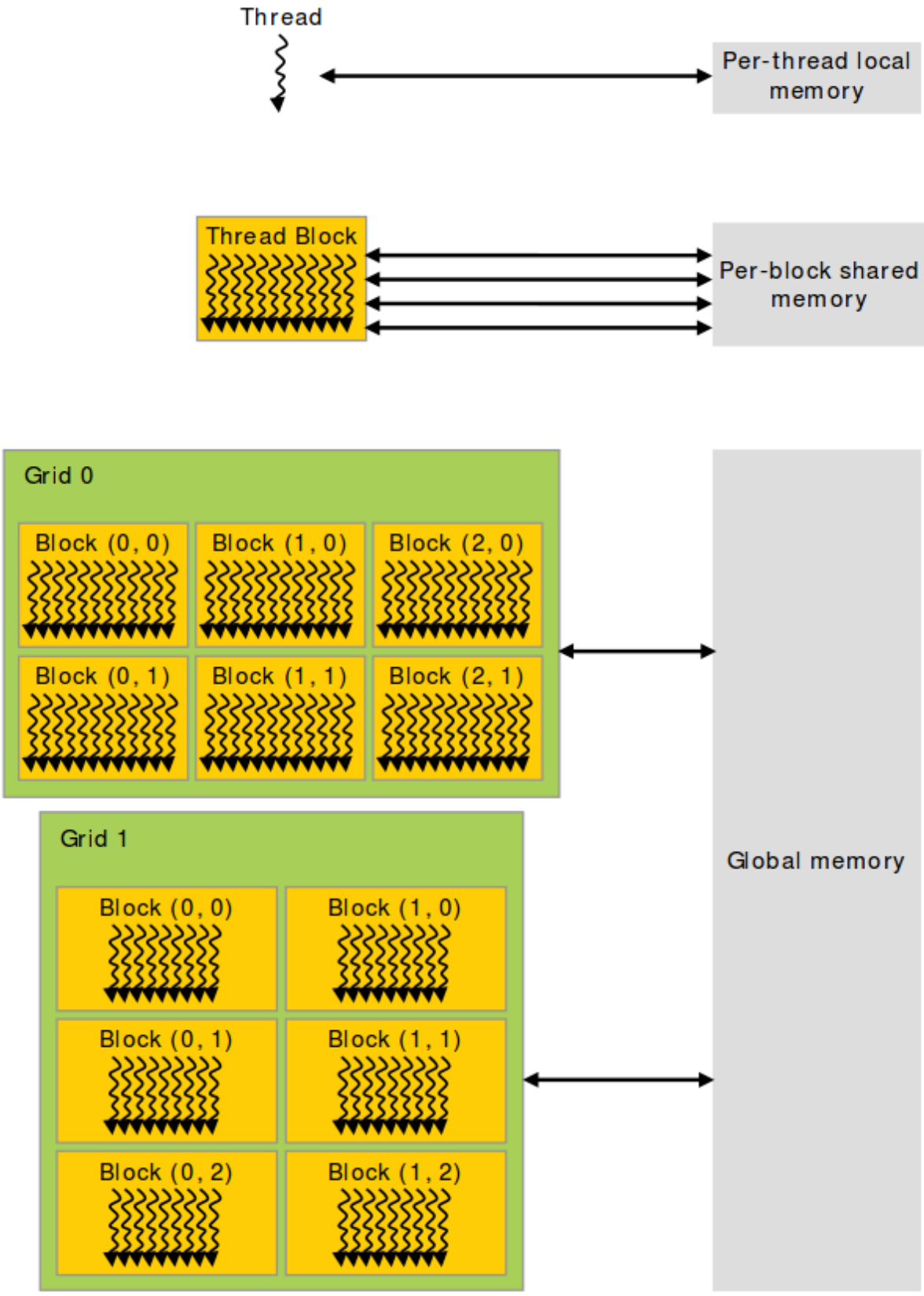

Рисунок 3 – Иерархическая структура памяти и права доступа к ней.

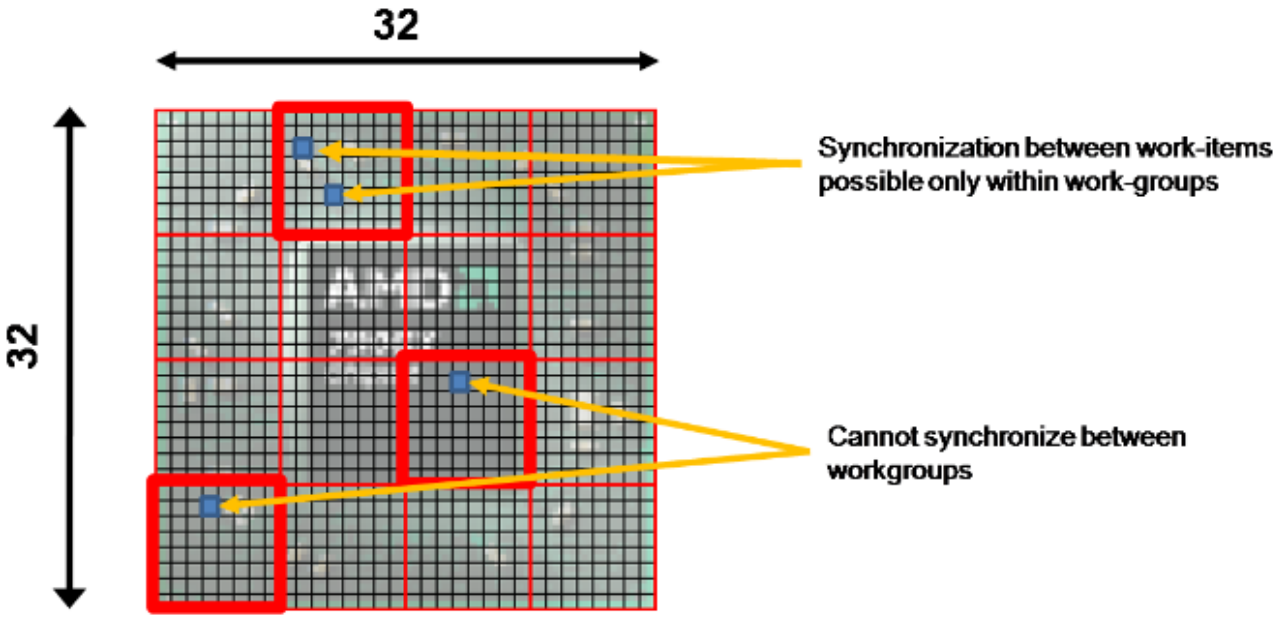

Рисунок 4 – Иерархическая структура рабочих групп.

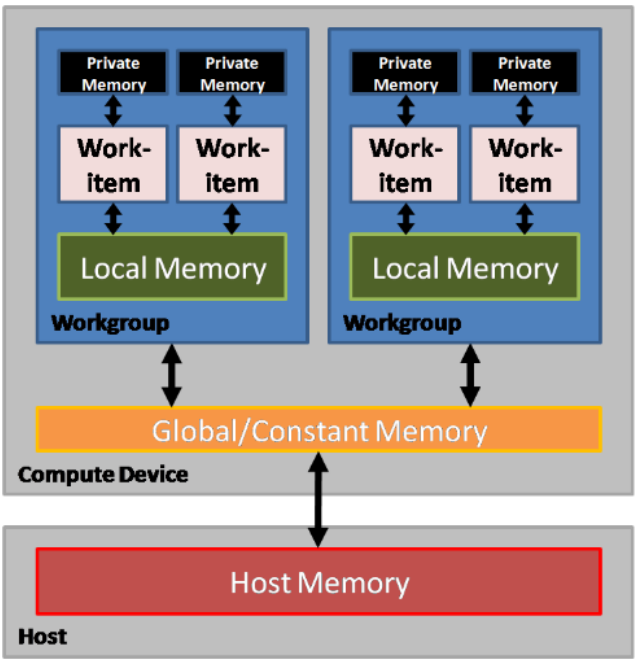

Рисунок 5 – Иерархия памяти в OpenCL.

3. Программная реализация

В рамках курсовой работы был разработан прототип программного обеспечения, рассчитывающего электродинамические структуры методом конечных разностей во временной области с использованием графического процессора. Применение ГПУ было призвано улучшить производительность вычислений.

3.1. Программная реализация сетки И

Разработка программы велась на языке C++, поэтому в качестве программного представления сетки И было решено использовать тип данных «класс». Входными аргументами конструктора класса выступают линейные размеры сетки в ячейках и величины Δx, Δy, Δz, Δt.

Из рис. 1 легко заметить, что все данные сетки (значения диэлектрической и магнитной проницаемости, удельной электрической и магнитной проводимостей, проекций векторов электрической и магнитной напряжённостей поля на координатные оси) представляют собой трёхмерные массивы с номерами ячеек i, j, k в качестве индексов.

3.2. Базовый алгоритм FDTD

Первым этапом создания программной реализации метода конечных разностей во временной области стало выделение из формул базового алгоритма коэффициентов и написание функций для их расчёта. Неизменность коэффициентов позволяет рассчитывать их единожды при старте программы и использовать готовые значения при пересчёте характеристик поля в каждой ячейке в каждый момент времени.

$$C = \frac{\frac{\Delta t}{P}}{1 + \frac{\sigma \Delta t}{2P}}, \quad D = \frac{1 - \frac{\sigma \Delta t}{2P}}{1 + \frac{\sigma \Delta t}{2P}}. \tag{3.1}$$

Данные коэффициенты приведены в формуле (3.1), где P — проницаемость материала (диэлектрическая и магнитная для проекций векторов $\vec{E}$ и $\vec{H}$ соответственно), а σ — удельная проводимость (электрическая и магнитная для проекций векторов $\vec{E}$ и $\vec{H}$ соответственно).

Следующим шагом стало написание функций, предназначенных для расчёта проекций векторов электрической и магнитной напряжённости во всех точках счётного объёма в какой-либо момент времени. Программный код, производящий расчёт компонент вектора магнитной напряжённости, приведён в листинге 1.

3.3. Тестовая задача

Симметричный вибратор — простейшая система для получения электромагнитных колебаний. Представляет собой электрический диполь, дипольный момент которого быстро изменяется во времени, и является развёрнутым колебательным контуром с минимальной ёмкостью и индуктивностью.

В исходном коде конечной программы симметричный вибратор был представлен двадцатью ячейками счётного объёма, расположенными вдоль оси Z, с отличными от остальных удельной электрической проводимостью материала σ и абсолютной диэлектрической проницаемостью материала ε. Длина волны была подобрана таким образом, чтобы отношение длины диполя к длине волны было равно двум.

Листинг 1 – Код расчёта компоненты вектора магнитной напряжённости на ЦПУ

```
int nx = grid.Hx.getCountX();
int ny = grid.Hx.getCountY();
int nz = grid.Hx.getCountZ();

float delta_x = grid.delta_x;
float delta_y = grid.delta_y;
float delta_z = grid.delta_z;

for(int ix = 0; ix < nx-1; ix++)
for(int iy = 0; iy < ny-1; iy++)
for(int iz = 0; iz < nz-1; iz++) {
    float& curHx    = grid.Hx       .at(ix,   iy,   iz);
    float  curD_Hx  = grid.D_Hx     .at(ix,   iy,   iz);
    float  curEz1   = grid.Ez       .at(ix,   iy+1, iz);
    float  curEz0   = grid.Ez       .at(ix,   iy,   iz);
    float  curEy1   = grid.Ey       .at(ix,   iy,   iz+1);
    float  curEy0   = grid.Ey       .at(ix,   iy,   iz);

    curHx -= curD_Hx * ((curEz1 - curEz0) / delta_y -
                        (curEy1 - curEy0) / delta_z);
}
```

3.4. Реализация резистивного источника

Следующим этапом моделирования резистивного источника стал пересчёт значений проекции вектора $\vec{E}$ на ось Z. Во избежание проверки каждой ячейки на наличие там проводящих структур было принято решение сохранять информацию обо всех ячейках до расчёта значений вектора электрической напряжённости, затем выполнять этот расчёт и, наконец, пересчёт значений проекции вектора $\vec{E}$ на ось Z только в точках присутствия элементов источника.

3.5. Перенос вычислительной нагрузки на графический процессор

В целях увеличения производительности функции расчёта значений проекций векторов электрической и магнитной напряжённостей были изменены для исполнения на графических процессорах. Из двух технологий, позволяющих осуществить расчёт на GPU: NVIDIA CUDA и OpenCL — была выбрана вторая, так как она поддерживает процессоры не только производства компании NVIDIA, но и AMD.

Для облегчения работы с фреймворком OpenCL была использована библиотека EasyCL. Код расчёта компонент вектора магнитной напряжённости приведён на листинге 2.

Листинг 2 – Код расчёта компонент вектора магнитной напряжённости на ГПУ

```
__kernel void calcH(int nx, int ny, int nz, float delta_x, float
   delta_y, float delta_z,
    __global float *Ex,
    __global float *Ey,
    __global float *Ez,
    __global float *Hx,
    __global float *Hy,
    __global float *Hz,
    __global float *D_Hx,
    __global float *D_Hy,
    __global float *D_Hz) {

    int ix = get_global_id(0);
    int iy = get_global_id(1);
    int iz = get_global_id(2);

    int idx = ix * ny * nz + iy * nz + iz;

    int idx010 = ix * ny * nz + (iy + 1) * nz + iz;
    int idx001 = ix * ny * nz + iy * nz + (iz + 1);
    int idx100 = (ix + 1) * ny * nz + iy * nz + iz;

        Hx[idx] -= D_Hx[idx] * ((Ez[idx010] - Ez[idx]) / delta_y -
                                (Ey[idx001] - Ey[idx]) / delta_z);

        Hy[idx] -= D_Hy[idx] * ((Ex[idx001] - Ex[idx]) / delta_z -
                                (Ez[idx100] - Ez[idx]) / delta_x);


        Hz[idx] -= D_Hz[idx] * ((Ey[idx100] - Ey[idx]) / delta_x -
                                (Ex[idx010] - Ex[idx]) / delta_y);
}
```

Заключение

В рамках работы был создан прототип программы для FDTD-моделирования, способный эффективно использовать ресурсы GPU. При помощи разработанной программы была произведена тестовая симуляция распространения гармонического сигнала, излучаемого тонким симметричным вибратором в замкнутом счётном объёме. Также была разработана референсная ЦПУ-реализация и произведено сравнение производительности расчётов с использованием ЦПУ и ГПУ.

В ходе работы было выяснено, что использование графических процессоров обеспечивает увеличение производительности до 17 раз, причём в программном коде не был реализован выровненный доступ к памяти, что является перспективным направлением разработки. Также, в отличие от CPU, производительность графических процессоров меньше зависит от размеров счётного объёма.

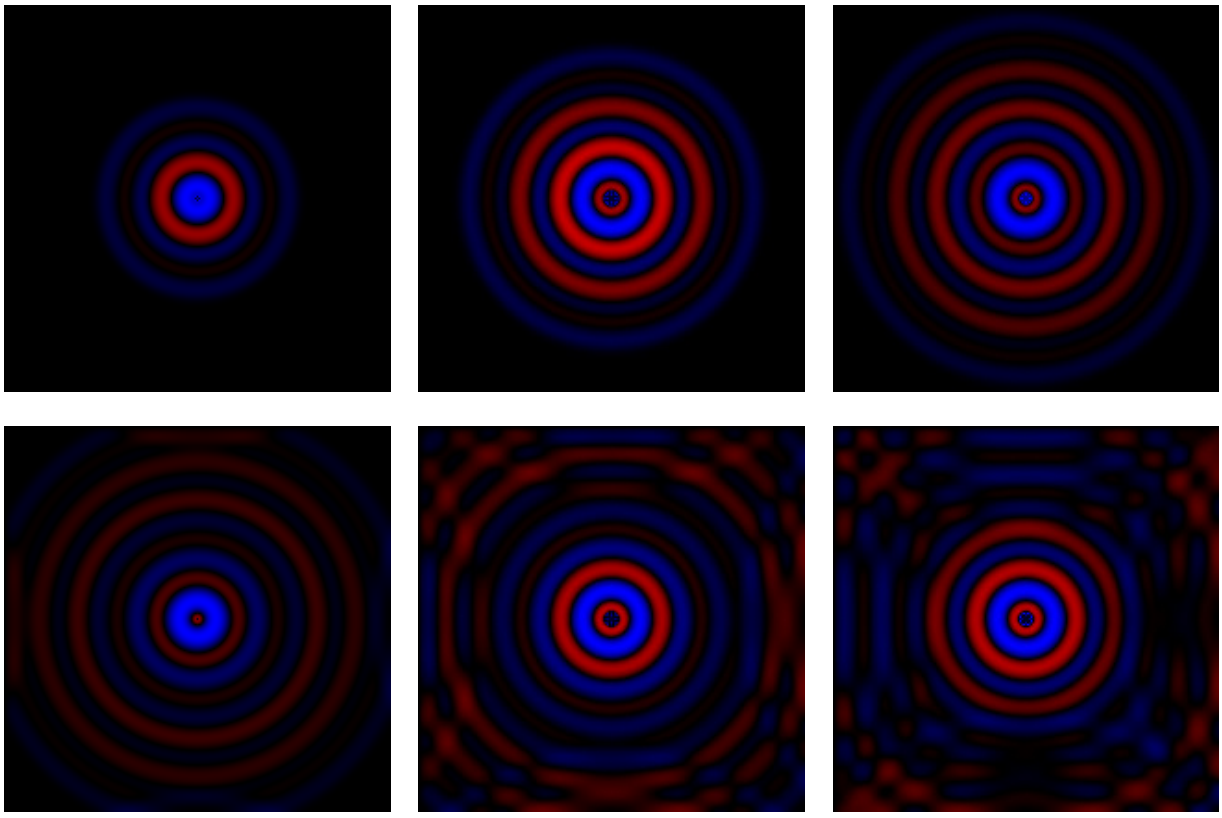

Рисунок 6 – Изменение электрического поля во времени (вид сверху)

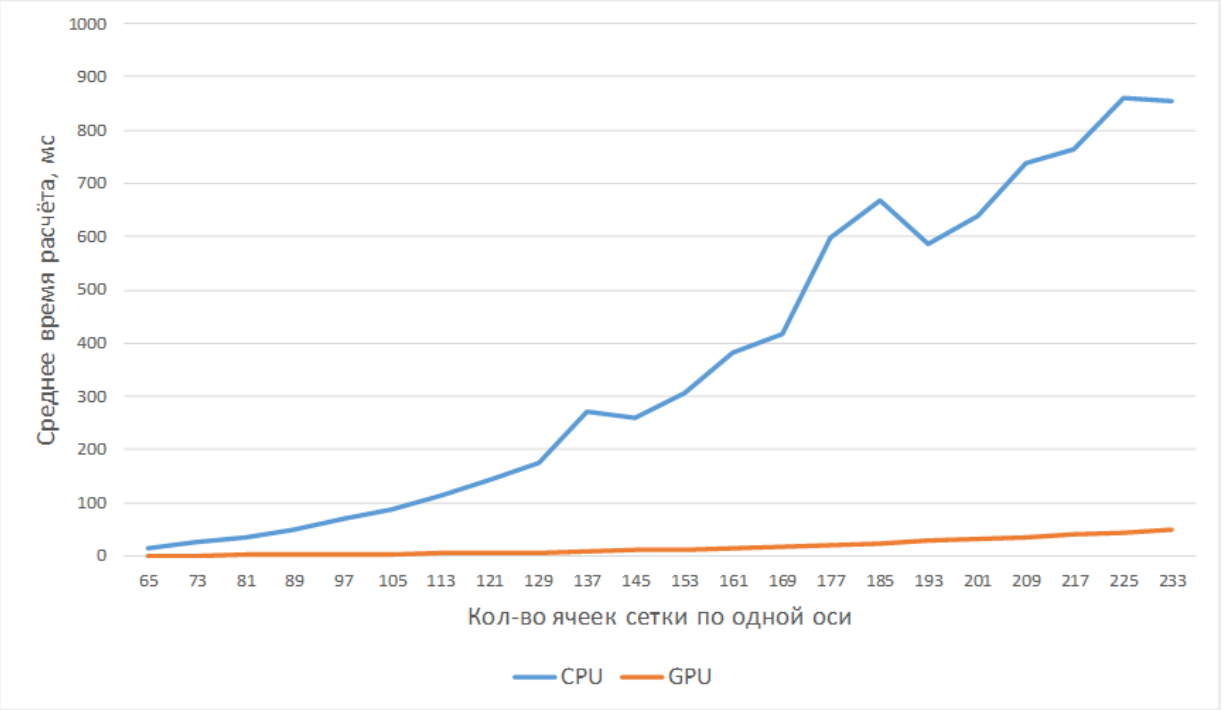

Рисунок 7 – Зависимость среднего времени расчёта компонент вектора магнитной напряжённости в конкретный момент времени во всех ячейках от размеров сетки.

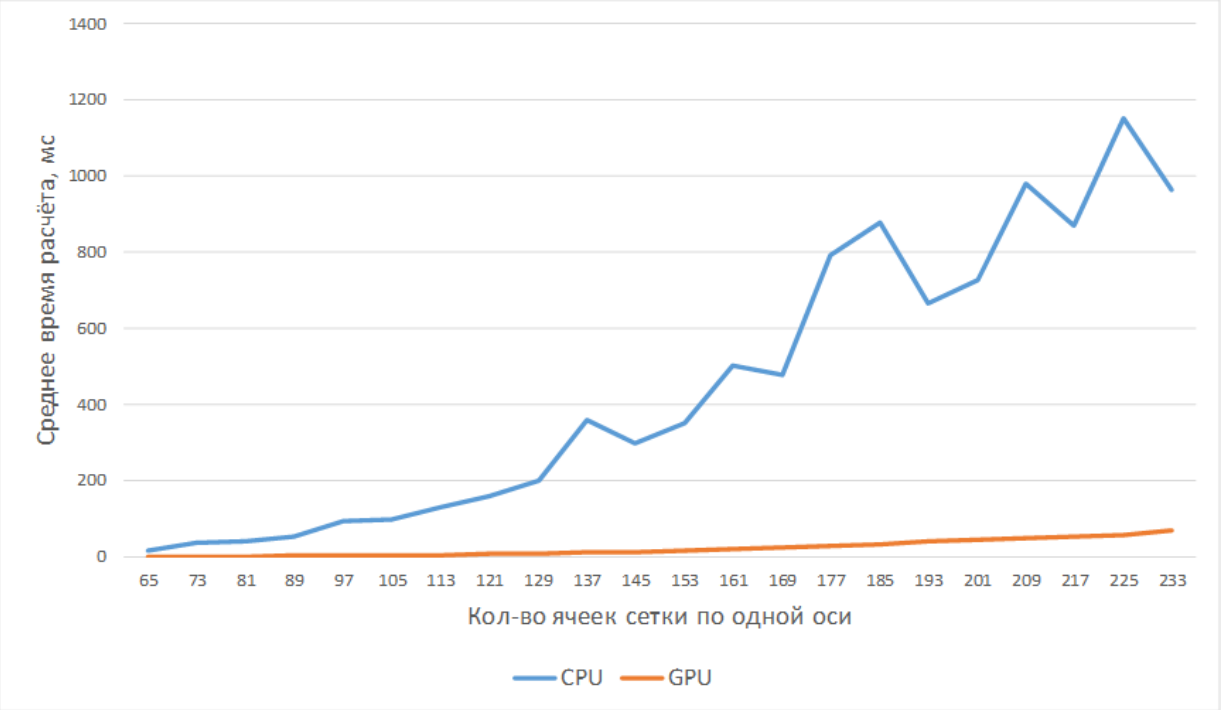

Рисунок 8 – Зависимость среднего времени расчёта компонент вектора электрической напряжённости в конкретный момент времени во всех ячейках от размеров сетки.

Список использованной литературы

[1] Wikipedia. Graphics processing unit. — URL: http://en.wikipedia.org/wiki/Graphics_processing_unit (дата обращения: 31.05.2017).

[2] А. Берилло. Nvidia CUDA — неграфические вычисления на графических процессорах. — Режим доступа: http://www.ixbt.com/video3/cuda-1.shtml.

[3] LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware / Nico Galoppo, Naga K. Govindaraju, Michael Henson, Dinesh Manocha // SC ’05 Proceedings of the 2005 ACM/IEEE conference on Supercomputing. — 2005.

[4] Wikipedia. CUDA. — URL: https://en.wikipedia.org/wiki/CUDA (дата обращения: 31.05.2017).

[5] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. — URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] Википедия. OpenCL. — URL: https://ru.wikipedia.org/wiki/OpenCL (дата обращения: 31.05.2017).