Алгоритм DiamondTorre и высокопроизводительная реализация FDTD метода для суперкомпьютеров с графическими ускорителями*

А.В. Закиров[1], В.Д. Левченко[1], А.Ю. Перепёлкина[1], Я. Земпо[2]

ИПМ им. М.В. Келдыша РАН[1], Hosei University[2]

Описана реализация конечно-разностного метода на сдвинутых сетках (FDTD) для решения задач электродинамики, в том числе нанооптики, требующих больших вычислительных ресурсов. Реализация основана на локально-рекурсивном нелокально-асинхронном (LRnLA) алгоритме DiamondTorre, учитывающем особенности иерархии подсистемы памяти и параллельности графических процессоров общего назначения (GPGPU). Это позволяет достигать высокого темпа вычислений (более 1 млрд обновлений ячеек Йи в секунду на одном ускорителе для схемы четвертого порядка аппроксимации) без ограничения размера задачи памятью устройства. В модели roofline суперкомпьютера TSUBAME 2.5 получены качественные и количественные оценки производительности и параллельного масштабирования, в зависимости от варьируемых параметров алгоритма. Параллельное масштабирование до 768 ускорителей является линейным для weak-метрики и насыщается на ∼100 ускорителях для strong-метрики.

Ключевые слова: суперкомпьютер с графическими ускорителями, локально–рекурсивные нелокально–асинхронные алгоритмы, конечно-разностный метод на сдвинутых сетках, модель вычислений «покатая крыша»

1. Введение

Конечно-разностный метод на сдвинутых сетках FDTD (Finite Difference Time Domain) [1] широко распространен для численного решения задач электродинамики, в частности, для разработки новых оптических устройств, сложных покрытий, наноантенн. Метод также легко адаптируется для моделирования волновых процессов различной физической природы, в том числе акустики и упругости [2].

В данной статье рассматривается один из путей переосмысления традиционного подхода численного решения эволюционной задачи Коши — локально-рекурсивные нелокально-асинхронные (LRnLA) алгоритмы [4]. Основываясь на этом подходе, мы разработали программное обеспечение для моделирования реальных задач оптики методом FDTD на графических процессорах общего назначения (GPGPU) и измерили его производительность как на одном ускорителе, так и на суперкомпьютере с множеством ускорителей.

Традиционные реализации FDTD базируются на алгоритмах, соответствующих фон-Неймановской модели вычислений с обобщением на параллельные модели Амдала (при использовании OpenMP) и Хоара (при использовании MPI). Среди относительно новых параллельных технологий следует отметить CUDA (OpenCL), дополнительно предлагающих наиболее адекватную поддержку векторизации.

Для эволюционных задач, послойная синхронизация параллельных вычислений является наиболее очевидным подходом: вычисления на следующем шаге по времени начинаются только после обновления данных всех полей во всей области на предшествующем шаге по времени. Это приводит к ограничениям масштабируемости из-за латентности коммуникационной среды и несбалансированности нагрузки вычислительных узлов.

Более серьёзная проблема связана с вызываемой послойным обходом нелокальностью данных. Традиционно, в качестве основного хранилища обновляемых на шаге полей исполь-

*Работа частично поддержана грантом Hosei International Fellowship, грантом РФФИ 14-01-31483

зуется оперативная память, а сами поля хранятся в виде линейных многомерных массивов. Однако подсистема памяти современных вычислительных систем является иерархической, и темп вычислений согласован с темпам доступа лишь к самым верхним уровням её иерархии. Увеличение же счетной области приводит к скачкообразному снижению производительности каждый раз при снижении уровня иерархии, в котором локализуются данные.

Эти проблемы в результате означают крайне низкую эффективность приложений, использующих послойную синхронизацию. Например, популярные программные пакеты, такие как Meep [5], Lumerical обладают эффективностью менее 1%. В то время как реализации, ориентированные на максимальную производительность с использованием быстрой памяти ускорителей [6], ограничены размером этой памяти.

Отличие между методами LRnLA и послойным обходом состоит в том, что оптимизация вычислений проводится не только на одном временном слое, а с отслеживанием зависимостей данных в области 4D пространства и времени (итераций) [7–9].

Данный подход не является широко распространенным, хотя подобные работы ведутся и другими авторами. Разрабатываемые с 1980-х годов так называемые методы разбиения цикла на блоки (loop tiling, loop blockig) и «скос» цикла (loop skewing) приводят к схожим алгоритмам для геометрически простых областей [10–13]. Исследования в области разбиения цикла не блоки привели к созданию кэш-совместных (cache-aware) и кэш-независимых (cache-oblivious) алгоритмов [14], которые позже были использованы при вычислениях по шаблону (stencil computations) [15–17]. Для 1D моделирования эти методологии приводят к разбиению пространства на блоки в форме трапеций и ромбов с обобщением на 2D и 3D.

Среди этих подходов отличие LRnLA лежит в области подходов к построению алгоритмов. В методе LRnLA наилучший алгоритм выбирается на основе анализа как вычислителя (путем создания соответствующей модели подсистемы памяти и уровней параллельности), так и свойств численной схемы (путем конструирования графа зависимостей-влияния и отслеживания зависимостей по операциям и данным).

2. Численный метод FDTD

В задачах оптики и других электродинамических процессов FDTD используется для численного решения уравнений Максвелла в пространственно-временной области:

$$\frac{\partial \vec{D}}{\partial t} = \nabla \times \vec{H}; \qquad \frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E},$$

где $\vec{E}$ ($\vec{H}$) и $\vec{D}$ ($\vec{B}$) означают напряженность электрического (магнитного) поля и его индукцию соответственно. $\vec{E}$ с $\vec{D}$, также как $\vec{H}$ с $\vec{B}$ связаны через материальные уравнения. В простейшем случае $\vec{D} = \vec{E}$, $\vec{B} = \vec{H}$. Скорость света равна 1 в выбранной системе обезразмеривания.

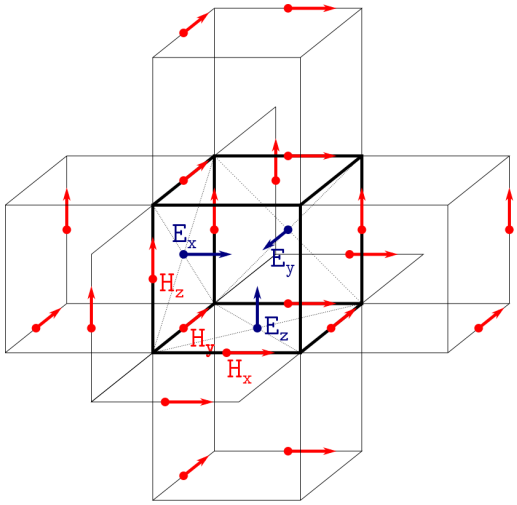

Рис. 1: Ячейка Йи. Наряду с основной ячейкой показаны также все поля, от которых зависят напряженности электрического поля (для схемы 4-го порядка аппроксимации).

Счетная область разбивается на сетку из ячеек Йи. Компоненты электрического и магнитного полей задаются в различных позициях внутри одной ячейки (рис. 1), и сдвинуты на полшага по времени. Мы используем схему 4-го порядка аппроксимации по пространству. Шаблон схемы имеет тип «крест». По сравнению со схемой 2-го порядка шаблон более широкий, то есть каждое обновление поля требует большего числа вычислений с большим числом данных. Основной выигрыш при этом состоит в том, что для достижения одинаковой точности решения достаточно более грубой сетки [18].

3. Описание алгоритма

Метод LRnLA предлагает отслеживание зависимостей численного метода для нахождения оптимального порядка вычислений. Оптимизация основывается на модели компьютера. В данной работе описано построение алгоритмов и преимущества этого метода на примере алгоритма DiamondTorre.

Для оценки эффективности алгоритмов будем использовать модель «покатой крыши» (Roofline) [19], отражающую ограничения пропускной способности памяти ускорителя. Такая модель в данном случае представляется более подходящей по сравнению с более сложными [20], поскольку процессы кэширования для GPGPU не так существенны как для процессоров общего назначения.

Модель Roofline определяет пиковую производительность в зависимости от параметра вычислительной интенсивности. Вычислительная интенсивность, являющаяся атрибутом используемого алгоритма, рассчитывается как отношение числа выполняемых операций для данного количества данных по отношению к размеру этих данных. В зависимости от этого параметра, алгоритмы разделяются на 2 класса: ограниченные по пропускной способности памяти (memory bound) и ограниченные по скорости обработки данных (compute bound). Наивная реализация метода FDTD с послойной синхронизацией обладает низкой вычислительной интенсивностью и относится к типу memory bound для любой актуальной вычислительной системы.

Для увеличения производительности необходимо увеличить вычислительную интенсивность. Это можно сделать, отказавшись от послойной синхронизации на каждом шаге по времени. В этом заключается основная идея альтернативных алгоритмов stencil-вычислений (trapezoids, time-slicing, time-skewing [16, 17, 20–24])

3.1. Алгоритм DiamondTorre и его CUDA реализация

Опишем алгоритм как порядок вычислений, который следует из декомпозиции 4D X-Y-Z-t счетной области. Каждый узел сетки в данном пространстве имеет зависимости от нескольких узлов на предыдущем слое. В 4D пространстве-времени область влияния (и зависимости) одного узла представляет из себя световой конус. Для численного метода FDTD в дискретной 4D области «пространственного обновления сетки» области влияния и зависимости представляют из себя 4D пирамиды с основаниями в виде трехмерных ромбов (октаэдров). Ромбы (или октаэдры в 3D) в основании областей влияния и зависимости позволяют считать, что разбиение счетной области на ромбовидные фигуры (DiamondTile) приводит к наибольшей вычислительной интенсивности. Фигуры DiamondTile можно объединить в «башни» (DiamondTorre, рис. 2).

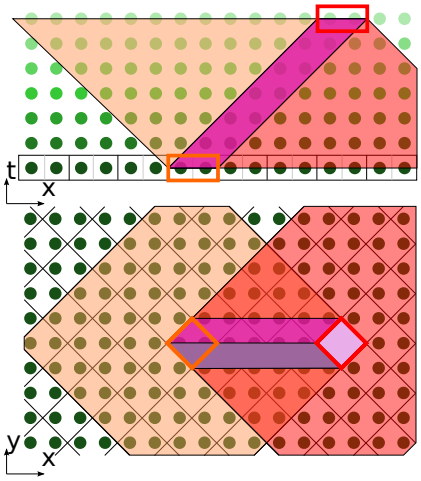

Рис. 2: Пересечение области влияния и области зависимости двух двумерных ромбовидных оснований в 3D пространстве-времени

Каждая такая фигура соответствует процедуре вычислений всех точек внутри по некоторому условному порядку. Мы можем оценить вычислительную интенсивность как количество этих точек (пропорционально количеству необходимых операций) делённое на количество зависимостей, пересекающих ее поверхность (пропорционально количеству необходимых данных).

Разбивая всю область на подобные фигуры и отслеживая зависимости, можно обнаружить, что некоторые фигуры имеют зависимости и требуют синхронизации вычислений, другие — асинхронны и могут быть выполнены параллельно. Набор таких фигур и состав-

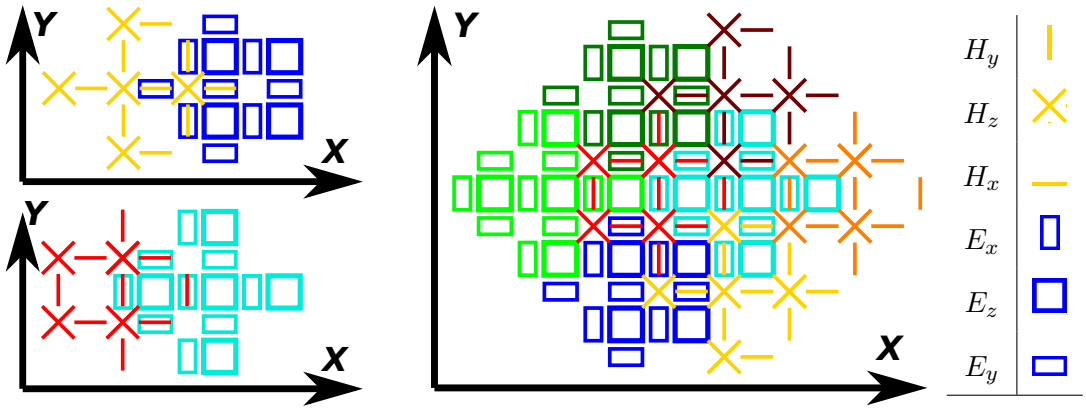

Рис. 3: Компоненты полей в одном DiamondTile, Два варианта. (слева) Зависимости одного DiamondTile (Тип II). Предположим, что один DiamondTile типа II загружается в регистры GPU. Для вычисления ромба с полями H необходимо дополнительно загрузить три соседних ромба с полями E. Далее, для вычисления ромба с полями E необходимо загрузить три соседних ромба (справа, сверху и снизу от него) с полями H. (справа)

ляет описание алгоритма.

На рис. 2 изображен элемент декомпозиции в 3D в области X-Y-t с 2D-ромбами в основании. В CUDA реализации один DiamondTorre выполняется одним CUDA-блоком. Узлы сетки вдоль оставшейся координатной оси (Z) обновляются посредством векторного параллелизма (CUDA-threads) в одном блоке. Размеры DiamondTorre определяются двумя параметрами: размер основания `DTS` и его высота `TH`. Более детальное описание алгоритма, его параметры и реализации для скалярного волнового уравнения на одном GPU можно найти в работе [7]. Здесь описывается специфика реализации алгоритма для метода FDTD и мульти-GPU систем.

Соответствие фигуры DiamondTile компонентам сдвинутой сетки изображено на рис. 3. Определим это как базовый элемент для алгоритма DiamondTile для схемы FDTD для уравнений Максвелла с 4-м порядком аппроксимации с размером `DTS=1`. Он состоит из двух ромбов (поля E и H), сдвинутых вдоль временной оси (на полшага по времени), а также вдоль одной координатной оси (на 1.5 пространственных шагов — полуширина шаблона выбранной схемы). DiamondTorre состоит из `TH` пар DiamondTile, сдвинутых аналогичным образом друг относительно друга. Будем называть алгоритмом DiamondTorre процесс выполнения вычислений для всех улов в описанной 4D фигуре. В GPGPU реализации, DiamondTorre — это ядро CUDA. Фигуры DiamondTorre с той же позицией по оси Y выполняются асинхронно CUDA-блоками.

Размер основания DiamondTorre `DTS`=1 в нашем случае оптимальный. Хотя увеличение `DTS` приводит к более высокой вычислительной интенсивности, уже для `DTS`=2 основными лимитирующим факторами становятся размер регистрового файла и кэша инструкций мультипроцессора (SM). Значительное падение производительности наблюдается в случае, если данные в одном DiamondTile превышают размер регистрового файла (на одном SM), или если количество инструкций в одном ядре DiamondTorre превысит размер кэша команд.

Количество памяти регистров равно 256 КБ. Если выбрать количество узлов вдоль оси Z равным $Nz = 384$ (двойная точность) или $Nz = 768$ (одинарная точность), то лимит регистров на один cuda-thread равен $256K/(768 \cdot 4) = 85$. Это примерно соответствует количеству данных для `DTS`=1. Также, в этом случае, одно ядро CUDA, соответствующее алгоритму DiamondTorre, содержит ~ 1500 инструкций. Размер одной команды ~ 8 байт, что приближает нас к пределу кэша инструкций (оцениваемый как 16 КБ).

Если мы предположим, что один DiamondTile загружается в регистры GPU (рис. 3), то вычисление одного ромба с H-полями требует трех ромбов с E-полями. Полученные в

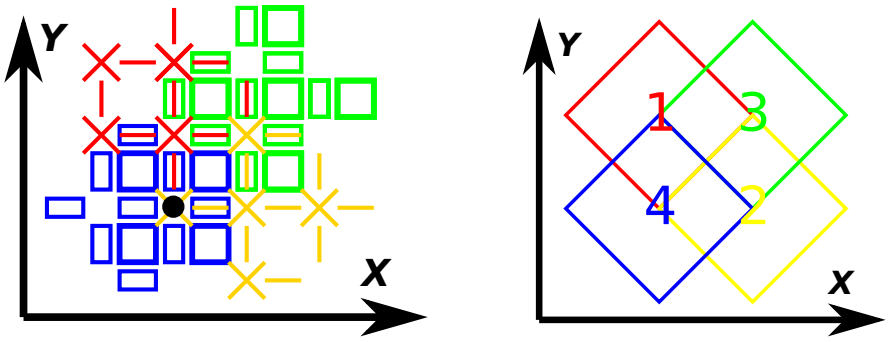

Рис. 4: Один элемент структуры данных (слева). Каждый элемент состоит из 4 ромбов (справа)

результате поля H нужно сохранить. Это составляет ровно половину процедуры обновления в DiamondTile, для другой половины три ромба с H-полями загружаются и один ромб с E-полями рассчитывается и сохраняется. В соответствии с этим можно оценить скорость обмена данных. При $\texttt{TH} \to \infty$ в среднем загружается 3 ячейки Йи и 1 ячейка Йи должна быть сохранена в расчете на обновление данных в одной ячейке Йи. Это составляет 4×6(полей в ячейке Йи) $\times\, 8$(байт на одно поле, двойная точность) $= 192$ байт. Количество операций с плавающей запятой для обновления одной ячейки Йи составляет около 110 Flop. Вычислительная интенсивность получается из их отношения, то есть 0.57 Flop/Byte.

Для любого современного GPU производительность в этой задаче ограничена пропускной способностью памяти. Теоретически она оценивается как обновление $P/192$ ячеек Йи в секунду, где P равна пропускной способности памяти GDDR5, а 192 — необходимый обмен данными, полученный ранее. Например, для NVidia Tesla K20 ($P = 224 \cdot 10^9$ байт в секунду) скорость обновлений составляет $1.167 \cdot 10^9$ ячеек Йи в секунду (Ys).

Данные хранятся в линейном 2D массиве, где каждый элемент представляет из себя набор векторов длины Nz (количество узлов вдоль оси z). Один элемент такого массива представлен на рис. 4. Такая структура данных выбрана для обеспечения коалесцированного доступа к данным для выбранного алгоритма вычислений, а также для минимизации количества элементов в операциях загрузки/сохранения данных.

3.2. Окно вычислений

Важнейшим преимуществом алгоритмов DiamondTorre является высокая производительность для больших счетных областей, в том числе и для тех, размер которых превышает память ускорителя. Этого легко добиться при обновлении данных в «окне вычислений», движущемся справа налево (в соответствии с фигурами выше). Асинхронно с вычислениями загружаются и выгружаются данные в/из глобальной RAM памяти. При этом загружаются только те отсутствующие данные, которые понадобятся для вычисления последующей группы DiamondTorre, а выгружаются обратно и удаляются из памяти ускорителя те данные, которые для вычисления следующей DiamondTorre не потребуются.

Темп счета не падает в том случае, если время выполнения DiamondTorre превышает время копирования данных в/из ускорителя. Поскольку время вычислений растет почти линейно с увеличением TH, а время копирования данных неизменно, то при достаточно большом TH пересылки host-device могут быть полностью скрыты. Если не учитывать влияние границ, то вычислительная интенсивность линейно растет с TH.

3.3. Производительность на малых масштабах

Описанные алгоритмы реализованы в программном коде, который включает не только основные вычисления шаблона FDTD, но и все необходимые методы для реальных физи-

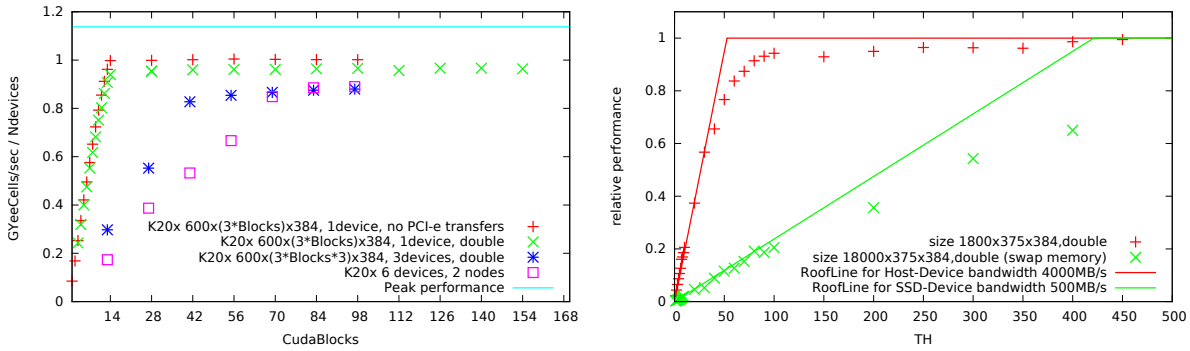

Рис. 5: Производительность для различного количества блоков (слева) и значений TH (справа)

ческих вычислений.

Для поиска оптимальных параметров алгоритма были проведены тесты производительности малого масштаба. Мы измеряли производительность в скорости обновления ялеек Йи в секунду (Ys). Это число как правило порядка 10^9, так что основная единица — GYs.

На рис. 5 слева показаны результаты тестирования для задачи размером $(600 \times (3 \times \mathit{blocks}) \times 384)$, с варьируемым количеством CUDA-блоков. Для DiamondTorre расположенных вдоль оси Y, в одном CUDA-блоке выполняются вычисления для одного DiamondTorre. Тесты проводились на ускорителе Tesla K20x. Он содержит 14 потоковых мультипроцессоров (streaming multiprocessors, SM). Если количество задействованных SM на ускорителе меньше 14, то производительность ограничена латентностью доступа к памяти GDDR5. Латентность составляет около 500 тактов или около 500нс, таким образом, исходя из закона Литтла [25], для того, чтобы эту латентность скрыть, необходимо не менее 10^4 запросов.

Максимальный размер вектора (Nz, также равный количеству используемых cuda-threads), при котором регистрового файла достаточно для хранения данных всех необходимых ромбов, на архитектуре Kepler составляет 384 для значений двойной точности. Поэтому при полном задействовании всех 14 мультипроцессоров пропускная способность шины данных не может быть полностью загружена ($\sim 14 \cdot 384 \approx 6000$ транзакций). Это является основной причиной, почему производительность составляет только 90% от пиковой и равна более 10^9 GYs на одном ускорителе при достаточно больших значениях `TH`.

На границах счетной области высота `TH` неизбежно становится меньше, поэтому на реальных задачах производительность будет немного падать (на несколько процентов).

Производительность в зависимости от различных значений параметра `TH` показана на рис. 5 справа. Так как величина `TH` пропорциональна локальности алгоритма, а ось ординат соответствует производительности (нормализованной на максимально достигнутую при текущем наборе параметров), то мы можем построить модель Roofline на одном графике, чтобы визуально понять ограничения. При малых значениях `TH` ограничениями производительности становится пропускная способность PCI-express. При увеличении `TH` происходит переход к ограничению пропускной способности GDDR5 памяти устройства. Плавность перехода объясняется отсечением части DiamondTorre по краям вычислительной области.

Оптимальным достаточным, как видно из графика, является размер `TH`=100. Данный размер и будет использоваться в тестах, если не уточнено специально. Аналогичное поведение зависимости скорости вычислений от высоты `TH` наблюдается в том случае, если данные хранятся на диске (в данном случае SSD). Но тогда оптимальное значение `TH` становится значительно больше (более 500).

3.4. Параллелизм по оси Y

Поскольку все DiamondTorre, расположенные рядом друг с другом вдоль по оси Y, асинхронны, то их можно выполнять также и на различных устройствах как в рамках одного вычислительного узла, так и на разных узлах (рис. 6). При этом данные на двух соседних устройствах перекрываются ровно на два DiamondTorre.

При расчете общий DiamondTorre выполняется только на одном устройстве в отдельном потоке (cuda stream), а затем измененные данные копируются на соседнее устройство в этом же потоке. В общем случае данные пересылаются через PCI-e и RAM (если отсутствует поддержка технологий NVidia Peer-to-Peer Memory Access или GPUDirect RDMA). Тогда необходимо переписать данные в буфер для посылки единым непрерывным блоком, который на другом устройстве аналогичным образом распаковать. Данная операция занимает столько же времени, сколько и вычисление одного DiamondTorre плюс время пересылки данных. В этом случае легко оценить требуемое количество асинхронных DiamondTorre на одном устройстве NA для того, чтобы пересылки между устройствами полностью скрывались: $NA > 2 \cdot Nsm + Tsend/Tcalc$. Здесь Nsm — количество мультипроцессоров (Streaming Multiprocessors, SM) на устройстве, Tsend — время пересылки данных одного DiamondTorre между двумя ускорителями, а Tcalc — время вычисления Nsm асинхронных DiamondTorre.

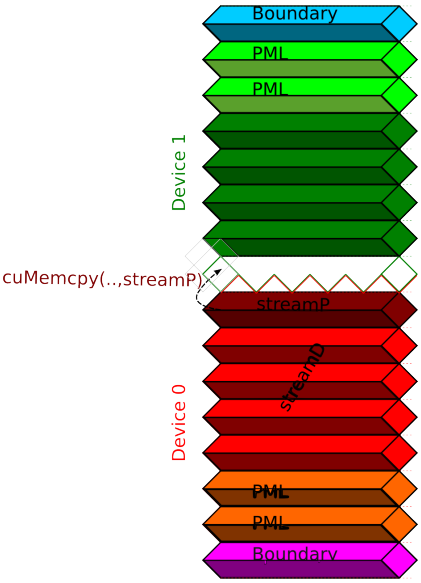

Рис. 6: Асинхронные DiamondTorre. Различные цвета соответствуют различным CUDA-ядрам, которые могут выполнятся на разных устройствах

Из предыдущих тестов (рис. 5) можно увидеть, что пересылки между 3 ускорителями в рамках одного узла могут быть полностью скрыты при количестве CUDA-блоков более 42 на каждом устройстве. Для ускорителей, установленных на разных узлах, время пересылки данных `Tsend` примерно в 2 раза больше, чем на одном узле. В этом случае пересылки могут быть полностью скрыты только при количестве задействованных CUDA-блоков более 70 на каждом устройстве.

3.5. Параллелизм по оси X

Дополнительно к предыдущему способу возможно параллельное выполнения задачи на многих узлах при декомпозиции по оси X. В этом случае данные разделяются на части по оси X. На соседних узлах данные перекрываются на размер окна вычислений: NW узлов по оси X (рис. 7). На узле i данные обновляются с шага $(n + i \cdot TH)$ до шага $(n + i \cdot TH + \mathrm{TH})$, где целое число n — шаг по времени на котором существуют данные крайнего левого узла.

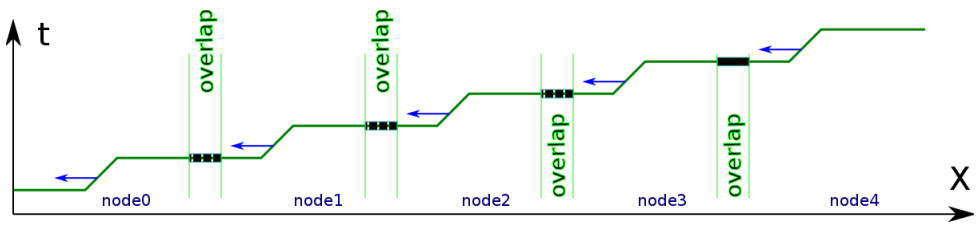

Рис. 7: Параллельность по оси X

Опишем алгоритм вычислений и обмена данными более подробно. Каждый узел, со-

держащий данные, которые не включают границу, выполняет поочередно следующие операции (рис. 8, 9): (waitR) Ожидание получения перекрывающихся справа данных с правого соседнего узла. (calcR) Выполнение расчета обновления перекрывающихся данных справа. (sendR) Неблокирующая посылка данных на узел справа. (calcM) Выполнение расчета обновления данных не зависящих от перекрывающихся областей. (waitL) Ожидание получения перекрывающихся слева данных с левого соседнего узла; эти данные были посланы на шаге sendR узла слева. (calcL) Выполнение расчета перекрывающихся данных слева. (sendL) Неблокирующая посылка данных на узел слева; эти данные будут получены во время шага waitR узлом слева.

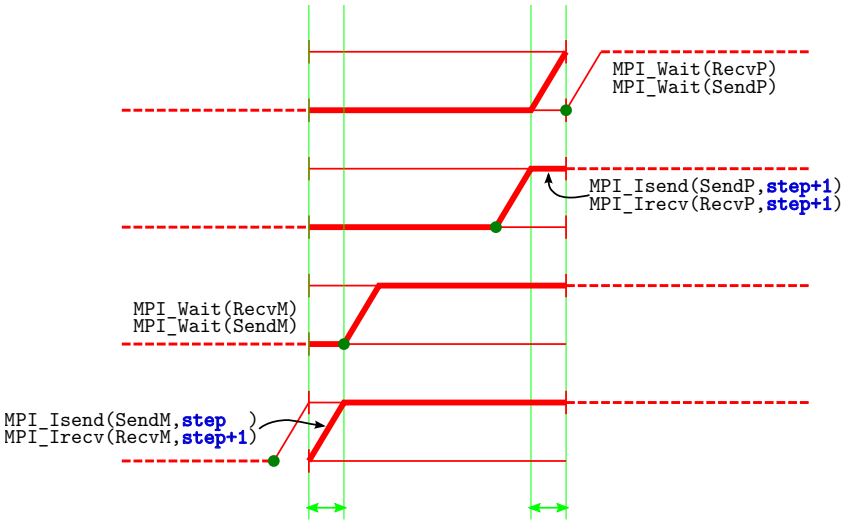

Рис. 8: Ключевые стадии одного узла при параллельной декомпозиции области по x-t

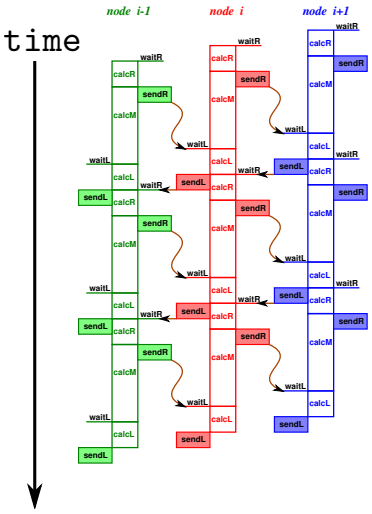

Рис. 9: Временная шкала процесса параллельного выполнения на узлах $i-1$,i,$i+1$

Зависимости между $(i-1)$-м, i-м и $(i+1)$-м узлами описанных стадий показаны на рис. 9. В случае, когда время выполнения calcM больше времени пересылок sendR+sendL, обмен данными между узлами будет полностью скрываться вычислениями.

4. Параллельное шкалирование на TSUBAME2.5

Тестирование параллельной производительности программного кода проводилось на суперкомпьютере TSUBAME2.5. Важные для для данного исследования характеристики суперкомпьютера следующие:

Количество доступных одновременно узлов для одной задачи до 300 в соответствии с условиями пользования (общее количество узлов 1408). На каждом узле установлено 3 GPGPU ускорителя NVIDIA Tesla K20x, содержащих суммарно $3 \times 5.625 = 16.875$ ГБ памяти GDDR5 с пиковой пропускной способностью 208 ГБ каждый. На каждом узле содержится от 54 ГБ до 96 ГБ (на нескольких узлах) оперативной памяти DDR3. Для вычислений доступно немного более 40 ГБ из них. Ускорители связаны по шине PCI-e 2.0 с пропускной способностью 4 ГБ/сек в каждом направлении. На каждом узле доступно 120 ГБ памяти SSD. Узлы соединены между собой интерконнектом Infiniband QDR с пропускной способностью до 4ГБ/сек.

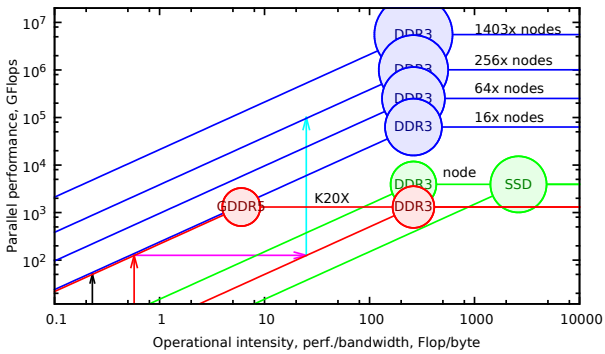

Рис. 10: График модели Roofline для TSUBAME2.5

5. Оценки из модели Roofline

Мы построили график модели Roofline для используемой системы (рис. 10). Подсистема памяти является иерархической, а алгоритмы LRnLA строятся с учетом этой иерархии. Именно поэтому мы рисуем графики Roofline для нескольких основных уровней памяти. Кроме того, необходимо учесть размер каждого уровня, поэтому график расширяется путем добавления соответствующих кругов. Размер круга показывает размер уровня памяти. Он расположен на углу соответствующей пары линий Roofline.

Красные линии показывают Roofline-линии для одного GPGPU устройства. Может быть использована как память ускорителя, так память узла, так что оба они оба показаны на одной Roofline-линии. Один узел (зеленые линии) содержит 3 ускорителя, поэтому его пиковая производительность показана в 3 раза выше. В тестах производительности было использовано до 256 узлов, а весь кластер содержит 1403 узла (синие линии).

Вычислительная интенсивность алгоритма, описаного в разделе 3.1, оценена как 0.57 (красная стрелка на рис. 10). Для данного алгоритма мы учитывали только внутреннюю память GPU-ускорителя. Эта простая оценка сделана при $Nt \to \infty$. Отличие от точного значения незначительно. Для сравнения, вычислительная интенсивность послойного наивного алгоритма (без пере-использования данных) оценивается в 0.23 (черная стрелка).

Когда используется RAM память узла, производительность ограничена пропускной способностью между канала хост-девайс (нижняя красная roofline-линия). При использовании окна вычислений вычислительная интенсивность увеличивается с параметром `TH` (фиолетовая горизонтальная стрелка). Стрелка упирается в roofline-линию хост-девайс, так как дальнейший сдвиг вправо не является необходимым из-за невозможности превзойти производительность алгоритма внутри ускорителя. С использованием многих узлов, теоретическая производительность масштабируется линейно (светло-синяя стрелка). Единственным ограничивающим фактором является пропускная способность межузловой связи.

5.1. Масштабирование в слабом смысле (Weak scaling)

Для масштабирования в слабом смысле (weak scaling) вычислительная область шкалируется пропорционально количеству используемых узлов. Был протестирован параллелизм как по X-, так и по Y-оси. Было выполнено три серии:

1. Шкалирование по оси Y. Пересылки данных между ускорителями можно полностью скрыть (112 CUDA-блоков на каждом ускорителе).

2. Шкалирование по оси Y. Пересылки данных между ускорителями на разных узлах ограничивают производительность (42 CUDA-блока на каждом ускорителе, см.рис. 5).

3. Шкалирование по оси X. Размер данных на каждом узле 1890 ячеек Йи (на каждом

узле задействованы все 3 ускорителя с параллельностью по Y).

Результаты представлены на рис. 11.

В первом случае, как и ожидается, параллельное ускорение составляет более 99%. Таким образом, общая достигнутая производительность равна $0.65 \cdot 10^{12}$ ячеек Йи в секунду на одной задаче с размером данных $300 \cdot 10^9$ узлов (занимающей 10 ТБ данных, 256 узлов).

Во втором случае происходит небольшое падение производительности при переходе с 1 на 2 узла, обусловленное несколько меньшей пропускной способностью Infiniband по сравнению со скоростью пересылки данных между ускорителями в рамках одного узла. Последующее увеличение узлов также не приводит к падению производительности.

В третьем случае также наблюдается практически идеальное параллельное ускорение. Оно все же несколько меньше, чем в первой серии, и падает в основном при переходе от 1 узла к 2. Это небольшое падение объясняется разбалансировкой загруженности узлов.

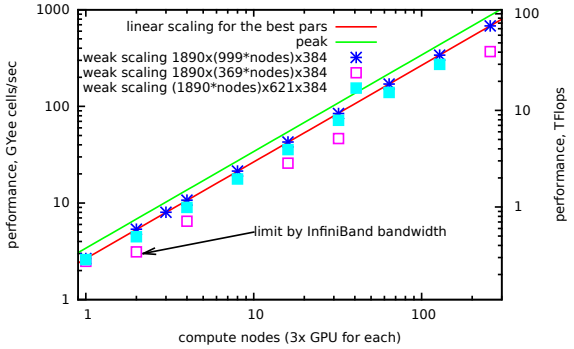

Рис. 11: Шкалирование в слабом смысле

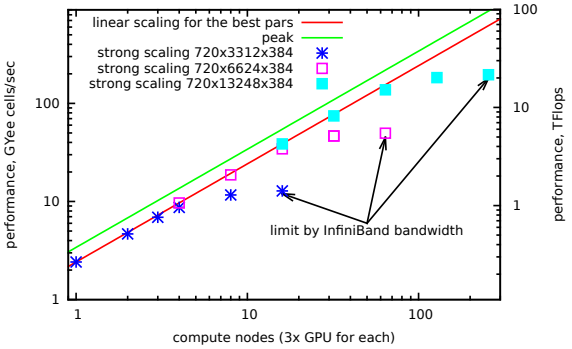

Рис. 12: Ускорение в метрике strong scaling.

5.2. Шкалирование в сильном смысле (Strong scaling)

Для сильного шкалирования по метрике strong scaling мы выбираем фиксированный размер счетной области, и с увеличением количества узлов, область подразделяется на все большее количество частей, которые обрабатываются параллельно. Было сделано три серии расчетов (рис. 12):

1. Область размером $720 \times 3312 \times 384$ улов шкалируется на 1–32 узлах;
2. В 4 раза большая область ($720 \times 13248 \times 384$ узлов) шкалируется на 4–64 узлах;
3. В 16 раз большая область ($720 \times 52992 \times 384$ узлов) шкалируется на 16–256 узлах;

При увеличении количества параллельных узлов производительность начинает падать из-за того, что уменьшается количество CUDA-блоков на каждом ускорителе (см. рис. 5). В то же время существует ограничение на максимальный размер Ny на одном узле, определяемый размером памяти ускорителя. Он, в принципе, может быть увеличен, но в то же время необходимо будет уменьшить размер Nx, а также вместе с этим и высоту DiamondTorre `TH`. Это приведет к снижению общей производительности. При использовании большего количества узлов для включения большего размера области высота `TH` может быть снова увеличена и производительность повысится.

Эта зависимость от `TH` показана на рис. 13. Размер задачи равен $450 \times 62208 \times 128$ узлов сетки. При использовании одного вычислительного узла `TH` равно 15, которое затем постепенно увеличивается до 150 при использовании 8 вычислительных узлов и больше.

На малом количестве узлов наблюдается сверх-линейное ускорение. Это возможно благодаря оптимизации по значению `TH`, которое возможно только при обработке достаточного количества данных на одном узле.

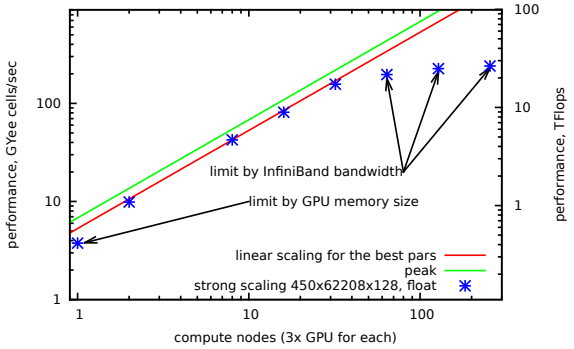

Рис. 13: Strong scaling по Y с переменным TH

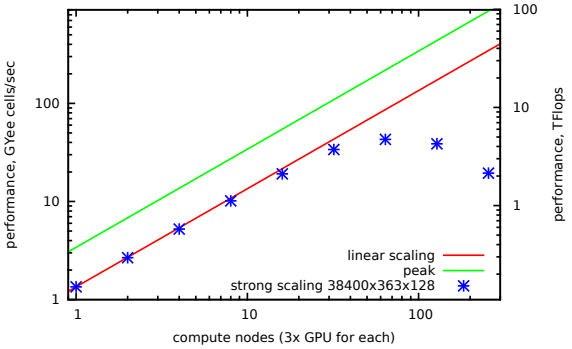

Рис. 14: Strong scaling по оси X

Последний результат касается серии strong-шкалирования по оси X на задаче размером $38400 \times 363 \times 128$ узлов сетки (рис. 14). Задача имеет примерно в 3 раза меньшую производительность на одном узле относительно пиковой из-за неоптимального размера области по оси Z. Он недостаточно большой для того, чтобы полностью скрыть латентность доступа к памяти GDDR5 (слишком мало одновременно выполняемых параллельных запросов). Но ускорение на такой задаче достигает почти 40 раз. Только при 128 вычислительных узлах и выше время пересылок данных начинают доминировать над вычислениями, что и приводит к наблюдаемому замедлению вычислений. Стоит отметить, что один вычислительный узел имеет крайне малый размер оперативной памяти (всего в 3 раза больше, чем вся память ускорителей), что и накладывает ограничения на ускорение. Кратное увеличение памяти должно при этом пропорционально увеличить возможности ускорения strong по оси X.

6. Заключение

Работу можно резюмировать следующим образом. Был разработан FDTD код, позволяющий проводить моделирование реальных оптических явлений. Отличительной особенностью кода является использование LRnLA алгоритма DiamondTorre, обеспечивающего максимальную производительность на одном устройстве, а также параллельную эффективность для мульти-GPU архитектур. Программное обеспечение было протестировано на суперкомпьютере TSUBAME2.5. Была остигнута высокая скорость вычислений (более 1 млрд обновлений ячеек Йи в секунду на одном ускорителе). Размер задачи при этом не ограничен памятью устройства. Параллельное масштабирование на ~ 1000 ускорителях является линейным для weak-метрики и насыщается на ~ 100 ускорителях для strong-метрики.

Параметры алгоритма (такие как `TH` и размер задачи) позволяют делать не только качественные, но и количественные оценки производительности и параллельного масштабирования. Максимальная достигнутая производительность составляет $0.65 \cdot 10^{12}$ ячеек Йи в секунду для трехмерной области в $0.3 \cdot 10^{12}$ ячеек Йи. Например, такой размер для задач волновой оптики соответствует моделированию 1 кубического миллиметра. Это означает существенный прорыв в вычислительной нанооптике, позволяя проводить моделирование в областях, которые ранее были слишком велики даже для суперкомпьютеров. Это может быть использовано для моделирования сложных оптических приборов и подложек.

Литература

1. Taflove A., Hagness S.C. Computational Electrodynamics: the Finite-Difference Time-Domain Method. 3rd edition. Norwood, MA : Artech House, 2005.

2. Virieux J.. P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method // Geophysics. 1986. Vol. 51, no. 4. P. 889–901.

3. Henning J.L. SPEC CPU2006 Benchmark Descriptions // Computer Architecture News. 2006. Vol. 34, no. 4. P. 1–17.

4. Левченко В.Д. Асинхронные параллельные алгоритмы как способ достижения эффективности вычислений // Инф. технологии и выч. системы. 2005. № 1. С. 68

5. Meep: A flexible free-software package for electromagnetic simulations by the FDTD method / A. F. Oskooi, D. Roundyb, M. Ibanescua et al. // Computer Physics Communications. 2010. Vol. 181. P. 687–702.

6. Acceleware Ltd. `http://www.acceleware.com/`. Accessed: 2016-04-18.

7. Perepelkina A.Yu, Levchenko V.D. DiamondTorre Algorithm for High-Performance Wave Modeling // Keldysh Institute Preprints. 2015. Vol. 18. P. 20.

8. Perepelkina A.Yu., Goryachev I.A., Levchenko V.D. CFHall Code Validation with 3D3V Weibel Instability Simulation // J. of Phys.: Conf. Ser. 2013. Vol. 441, no. 1. P. 012014.

9. Perepelkina A.Yu, Goryachev I.A., Levchenko V.D. Implementation of the Kinetic Plasma Code with Locally Recursive non-Locally Asynchronous Algorithms // J. of Phys.: Conf. Ser. 2014. Vol. 510, no. 1. P. 012042.

10. Wolf M.E., Lam M.S. A Data Locality Optimizing Algorithm // Proceedings of the ACM SIGPLAN 1991 Conf. on PLDI '91. New York, NY, USA : ACM, 1991. P. 30–44.

11. Wolfe M. More Iteration Space Tiling // Proceedings of the 1989 ACM/IEEE Conference on Supercomputing. Supercomputing '89. New York, NY, USA : ACM, 1989. P. 655–664.

12. Wolfe M.. Loops skewing: The wavefront method revisited // International Journal of Parallel Programming. 1986. Vol. 15, no. 4. P. 279–293.

13. Terrano A.E. Optimal tilings for iterative PDE solvers // Frontiers of Massively Parallel Comp., 1988. Proceedings., 2nd Symposium on the Frontiers of. 1988. Oct. P. 227–229.

14. Prokop H.. Cache-Oblivious Algorithms. 1999.

15. Frigo M., Strumpen V.. Cache Oblivious Stencil Computations // Proc. of the ICS '05. New York, NY, USA : ACM, 2005. P. 361–366.

16. Strzodka R., Shaheen M. et al. Cache Accurate Time Skewing in Iterative Stencil Computations // Proc. of ICPP /IEEE Computer Society, 2011. sep. P. 571–581.

17. Strzodka R., Shaheen M. et al. Cache Oblivious Parallelograms in Iterative Stencil Computations // Proc. of ICS '10. New York, NY, USA : ACM, 2010. P. 49–59.

18. Kantartzis N. V., Tsiboukis T.D. Higher Order FDTD Schemes for Waveguide and Antenna Structures. 1st edition. Morgan & Claypool Publishers, 2006.

19. Williams S., Waterman A., Patterson D.A. Roofline: an insightful visual performance model for multicore architectures. // Commun. ACM. 2009. Vol. 52, no. 4. P. 65–76.

20. Frigo M., Strumpen V.. The memory behavior of cache oblivious stencil computations // The Journal of Supercomputing. 2007. Vol. 39, no. 2. P. 93–112.

21. Tang Yu., Chowdhury R.A. et al. The Pochoir Stencil Compiler / Proc. of SPAA '11. New York, NY, USA : ACM, 2011. P. 117–128.

22. Grosser T., Cohen A., et al. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles // Proc. of the GPGPU-6. New York, NY, USA : ACM, 2013. P. 24–31.

23. Orozco D., Gao Guang. Mapping the FDTD Application to Many-Core Chip Architectures // Parallel Processing, 2009. ICPP '09. International Conference on. 2009. Sept. P. 309–316.

24. Wonnacott D. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality : // Rep. / Executor: John McCalpin, 1999.

25. Little J.D.C. A Proof for the Queuing Formula: L = lW // Operations Research. 1961. Vol. 9, no. 3. P. 383–387. http://dx.doi.org/10.1287/opre.9.3.383.

The DiamondTorre algorithm and high performance FDTD code implementation for GPGPU supercomputers

V.D. Levchenko[1], A.V. Zakirov[1], A.Yu. Perepelkina[1], Y. Zempo[2]

Keldysh Institute of Applied Mathematics[1], Hosei University[2]

An implementation of FDTD (Finite Difference Time Domain) method for solution of optical and other electrodynamic problems (including nanooptics) of high computational cost is described. The implementation is based on LRnLA (Locally Recursive non-Locally Asynchronous) algorithm DiamondTorre, which is developed with acount for memory subsystem and parallel levels hierarchy of GPGPU (General Purpose Graphical Processing Unit) hardware. High computation rate is achieved (more than 1 billion Yee cell updates per second on one device). The problem size is not limited by device memory. The specifics of the DiamondTorre algorithms for staggered grid (Yee cell) and many-GPU devices with the use of CUDA, OpenMP and MPI technologies are shown. The algorithm is implemented in software for real physics calculation with the use of CUDA, OpenMP, MPI technologies. In the roofline model of TSUBAME2.5 supercomputer the qualitative and quantitative estimates of performance and parallel scaling depending on the algorithm parameters are acquired. These estimates are compared with the achieved performance of the code on one device, as well as on parallel weak and strong scaling tests. Weak scaling is linear up to 768 devices. Strong scaling saturates at about 100 devices. Algorithm parameters allow not only qualitative, but also quantitative estimates of the performance and parallel scaling. Maximal achieved performance is 650 billions Yee Cells per second for 3D domain with 300 billions Yee cells total.

Keywords: GPGPU supercomputers, LRnLA algorithms, FDTD method, roofline model

References

1. Taflove A., Hagness S.C. Computational Electrodynamics: the Finite-Difference Time-Domain Method. 3rd edition. Norwood, MA : Artech House, 2005.

2. Virieux J.. P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method // Geophysics. 1986. Vol. 51, no. 4. P. 889–901.

3. Henning J.L. SPEC CPU2006 Benchmark Descriptions // Computer Architecture News. 2006. Vol. 34, no. 4. P. 1–17.

4. Levchenko V.D. Asynchronous parallel algorithms as a way to archive effectiveness of computations (in Russian) // J. of Inf. Tech. and Comp. Systems. 2005. № 1. P. 68

5. Meep: A flexible free-software package for electromagnetic simulations by the FDTD method / A. F. Oskooi, D. Roundyb, M. Ibanescua et al. // Computer Physics Communications. 2010. Vol. 181. P. 687–702.

6. Acceleware Ltd. `http://www.acceleware.com/`. Accessed: 2016-04-18.

7. Perepelkina A.Yu, Levchenko V.D. DiamondTorre Algorithm for High-Performance Wave Modeling // Keldysh Institute Preprints. 2015. Vol. 18. P. 20.

8. Perepelkina A.Yu., Goryachev I.A., Levchenko V.D. CFHall Code Validation with 3D3V Weibel Instability Simulation // J. of Phys.: Conf. Ser. 2013. Vol. 441, no. 1. P. 012014.

9. Perepelkina A.Yu, Goryachev I.A., Levchenko V.D. Implementation of the Kinetic Plasma Code with Locally Recursive non-Locally Asynchronous Algorithms // J. of Phys.: Conf. Ser. 2014. Vol. 510, no. 1. P. 012042.

10. Wolf M.E., Lam M.S. A Data Locality Optimizing Algorithm // Proceedings of the ACM SIGPLAN 1991 Conf. on PLDI '91. New York, NY, USA : ACM, 1991. P. 30–44.

11. Wolfe M. More Iteration Space Tiling // Proceedings of the 1989 ACM/IEEE Conference on Supercomputing. Supercomputing '89. New York, NY, USA : ACM, 1989. P. 655–664.

12. Wolfe M.. Loops skewing: The wavefront method revisited // International Journal of Parallel Programming. 1986. Vol. 15, no. 4. P. 279–293.

13. Terrano A.E. Optimal tilings for iterative PDE solvers // Frontiers of Massively Parallel Comp., 1988. Proceedings., 2nd Symposium on the Frontiers of. 1988. Oct. P. 227–229.

14. Prokop H.. Cache-Oblivious Algorithms. 1999.

15. Frigo M., Strumpen V.. Cache Oblivious Stencil Computations // Proc. of the ICS '05. New York, NY, USA : ACM, 2005. P. 361–366.

16. Strzodka R., Shaheen M. et al. Cache Accurate Time Skewing in Iterative Stencil Computations // Proc. of ICPP /IEEE Computer Society, 2011. sep. P. 571–581.

17. Strzodka R., Shaheen M. et al. Cache Oblivious Parallelograms in Iterative Stencil Computations // Proc. of ICS '10. New York, NY, USA : ACM, 2010. P. 49–59.

18. Kantartzis N. V., Tsiboukis T.D. Higher Order FDTD Schemes for Waveguide and Antenna Structures. 1st edition. Morgan & Claypool Publishers, 2006.

19. Williams S., Waterman A., Patterson D.A. Roofline: an insightful visual performance model for multicore architectures. // Commun. ACM. 2009. Vol. 52, no. 4. P. 65–76.

20. Frigo M., Strumpen V.. The memory behavior of cache oblivious stencil computations // The Journal of Supercomputing. 2007. Vol. 39, no. 2. P. 93–112.

21. Tang Yu., Chowdhury R.A. et al. The Pochoir Stencil Compiler / Proc. of SPAA '11. New York, NY, USA : ACM, 2011. P. 117–128.

22. Grosser T., Cohen A., et al. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles // Proc. of the GPGPU-6. New York, NY, USA : ACM, 2013. P. 24–31.

23. Orozco D., Gao Guang. Mapping the FDTD Application to Many-Core Chip Architectures // Parallel Processing, 2009. ICPP '09. International Conference on. 2009. Sept. P. 309–316.

24. Wonnacott D. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality : // Rep. / Executor: John McCalpin, 1999.

25. Little J.D.C. A Proof for the Queuing Formula: L = lW // Operations Research. 1961. Vol. 9, no. 3. P. 383–387. http://dx.doi.org/10.1287/opre.9.3.383.