УДК 519.6

ИЕРАРХИЧЕСКИЙ АЛГОРИТМ РАСПАРАЛЛЕЛИВАНИЯ ВЫЧИСЛЕНИЙ ПРИ РЕШЕНИИ ТРЕХМЕРНОГО УРАВНЕНИЯ ПЕРЕНОСА НЕЙТРОНОВ НА ГИБРИДНЫХ СУПЕР-ЭВМ

А. А. Нуждин
(ФГУП "РФЯЦ-ВНИИЭФ", г. Саров Нижегородской области)

Представлен иерархический алгоритм распараллеливания, который основан на выделении уровней архитектуры гибридной супер-ЭВМ с GPU и применении принципа геометрической декомпозиции на каждом из уровней. Внутри компактной группы нитей используется распараллеливание по элементам гиперплоскости фиксированного размера, состоящей из ячеек сетки. Между блоками одного GPU используется KBA-алгоритм на основе двумерной декомпозиции по столбцам и гиперплоскостям. Между различными GPU используется KBA-алгоритм на основе трехмерной декомпозиции. Программная реализация выполнена на примере тестовой программы ПАУК с помощью технологии CUDA. Эффективность адаптации теста ПАУК к одному GPU подтверждена результатами профилирования и сравнением с производительностью CPU-версии программы. Масштабируемость теста ПАУК в режиме multi-GPU исследована методом умножения.

Ключевые слова: GPU, CUDA, S_n-метод, алгоритм бегущего счета, KBA-алгоритм.

Введение

Графические ускорители (GPU) фирмы Nvidia уже более десяти лет активно используются в области суперкомпьютерных вычислений. В списке TOP-500 самых высокопроизводительных супер-ЭВМ за июнь 2020 г. суммарная пиковая производительность всех установленных GPU Nvidia различных поколений превысила рубеж в 10^{18} флопс, а в списке за ноябрь 2020 г. данная отметка была превышена уже одним поколением ускорителей — Nvidia Volta. За последнее десятилетие наибольший прирост производительности GPU был обеспечен увеличением числа потоковых мультипроцессоров (SM), что обусловлено совершенствованием архитектуры и технологического процесса. Число SM возросло с 14 для поколения Nvidia Kepler до 108 для Nvidia Ampere. При этом технология программирования CUDA [1] в части возможностей для взаимодействия процессов по данным и управлению развита сильнее для одного блока нитей (один SM), чем для сети блоков (множество SM).

Эволюционное увеличение числа ядер на кристалле универсального процессора (CPU) поддержано более совершенной экосистемой разработки параллельных приложений: модели памяти (разделенная, общая, PGAS), стандарты (MPI, OpenMP), инструментальные средства (компилятор, среда исполнения, библиотека MPI). Данная экосистема ориентирована на удобство выражения алгоритмов распараллеливания с учетом взаимодействия процессов как в части синхронизации, так и в части обмена данными.

В работе представлен иерархический алгоритм распараллеливания для решения уравнения переноса нейтронов разностным S_n-методом на трехмерной ортогональной пространственной сетке. Алгоритм характеризуется применением принципа геометрической декомпозиции на различных уровнях архитектуры гибридной супер-ЭВМ (1 SM, 1 GPU, множество GPU) и программной модели (компактная группа нитей, сетка блоков, множество MPI-процессов). На рис. 1 проиллюстрированы ключевые особенности алгоритма. Слева указаны выделяемые уровни архитектуры, справа —

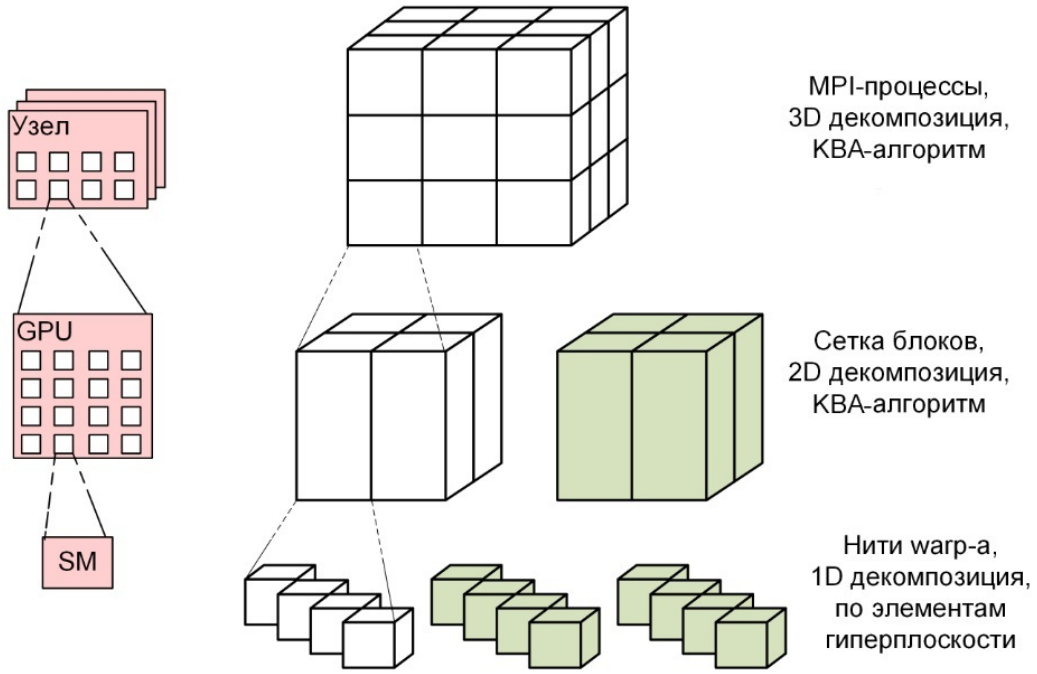

Рис. 1. Уровни архитектуры и типы пространственной декомпозиции на этих уровнях

тип пространственной декомпозиции на каждом из уровней. Здесь и далее warp обозначает компактную группу из 32 нитей одного блока.

На каждом из уровней реализуется распараллеливание для вложенного алгоритма бегущего счета, требующее взаимодействия между процессами. Каждый из рассматриваемых уровней обладает своими средствами для синхронизации и обмена данными, которые существенно отличаются масштабами накладных расходов. Это приводит к различной зернистости (granularity) алгоритмов распараллеливания.

Дополнительно на рис. 1 серо-зеленым цветом выделена вспомогательная декомпозиция по направлениям полета частиц (или группам), используемая для снижения давления на подсистему памяти ускорителя. В табл. 1 представлена иерархия алгоритмов распараллеливания (как основных, так и вспомогательных) на всех выделенных уровнях программной модели. Полужирным шрифтом отмечены алгоритмы на основе геометрической декомпозиции.

Апробация предлагаемого иерархического алгоритма распараллеливания выполнена на примере адаптации тестовой программы ПАУК [2] к архитектуре гибридных супер-ЭВМ с GPU. Исследования производительности и масштабируемости выполнены на различных образцах вычислительной техники, в том числе из состава программно-аппаратного полигона Национального центра физики и математики (НЦФМ) г. Сарова для исследования архитектур супер-ЭВМ.

Таблица 1

Уровни иерархического алгоритма распараллеливания

Уровень программной модели	Алгоритм
MPI-процессы	**KBA** — 8 конвейеров по направлениям/группам, 3D декомпозиция
Группы блоков	По направлениям/группам, 1D декомпозиция
Блоки	**KBA** — 1 конвейер по слоям, 2D декомпозиция
warp-ы блока	По направлениям, 1D декомпозиция
Нити warp-а	**По элементам гиперплоскости**, 1D декомпозиция

Аналогичные работы

Схожие работы по реализации на GPU алгоритма бегущего счета можно классифицировать по уровню программной модели, на которой реализуется распараллеливание по элементам одной гиперплоскости [3]. После обработки каждой гиперплоскости требуются точка синхронизации и обмен данными между параллельными процессами. Базовые возможности технологии программирования CUDA обеспечивают два способа синхронизации: между нитями одного блока с помощью функции `_syncthreads` и между всеми нитями при выходе из ядра функции, выполняемой на GPU.

Распараллеливание бегущего счета на уровне нитей одного блока реализуется при первом способе синхронизации: обмен данными выполняется через разделяемую память, каждая нить рассчитывает множество ячеек пространственной сетки. По такой схеме выполнена адаптация к GPU кодов DENOVO [4], Minisweep [5] и Ardra [6]. В качестве технологии параллельного программирования использовались CUDA в DENOVO и Minisweep, OpenACC [7] в Minisweep, инфраструктура RAJA [8] в Ardra. Особенностью алгоритма в Ardra является фиксированный размер одного блока — 20×20 нитей.

Распараллеливание бегущего счета на уровне сети блоков реализуется при втором способе синхронизации: обмен данными выполняется через глобальную память, каждая нить рассчитывает одну ячейку пространственной сетки. Такой способ синхронизации использовался в работах по адаптации к GPU тестового приложения SNAP [9] с помощью технологии CUDA и стандарта OpenCL [10]. Особенностью OpenCL-реализации является "плоская" формулировка алгоритма, в которой определяется только общее число нитей, а размер одной рабочей группы или блока нитей задается не связанным с логикой алгоритма значением компилятора по умолчанию.

Комбинация двух способов синхронизации использовалась при исследовании вариантов распараллеливания для гнезда вложенных циклов с зависимостями на примере счетного прототипа [11], одним из применений которого может быть алгоритм бегущего счета на ортогональной сетке. В этом случае сетка логически разбивается на фрагменты и организуется вложенный бегущий счет — по фрагментам и по ячейкам в каждом из этих фрагментов. Первый способ синхронизации используется при распараллеливании по элементам гиперплоскости внутри одного фрагмента, второй способ — при распараллеливании между фрагментами, при этом размер сети блоков определяется числом фрагментов в текущей гиперплоскости.

Во всех перечисленных приложениях для увеличения степени параллелизма на GPU активно используются остальные переменные фазового пространства задачи: направления полета частиц и группы. Распараллеливание по указанным переменным не является алгоритмически сложным, однако означает исключение ресурса этих переменных из MPI-конвейеров, что негативно влияет на масштабируемость в режиме multi-GPU [12].

Постановка задачи

В тестовой программе ПАУК решается стационарное трехмерное уравнение переноса в одногрупповом кинетическом приближении в декартовой системе координат на ортогональных пространственных сетках [2]:

$$\operatorname{div}\left(\vec{\Omega}N\right)+\alpha N=\frac{1}{4\pi}\left(\beta n^{(0)}+Q\right),\quad \operatorname{div}\left(\vec{\Omega}N\right)=\Omega_x\frac{\partial N}{\partial x}+\Omega_y\frac{\partial N}{\partial y}+\Omega_z\frac{\partial N}{\partial z}, \tag{1}$$

где α — коэффициент столкновения частиц; β — коэффициент размножения частиц; Q — независимый источник частиц; N — плотность потока частиц, летящих в направлении $\vec{\Omega}$ (для определенности скорость частиц $\upsilon=1$); $\vec{\Omega}\,(\Omega_x,\Omega_y,\Omega_z)$ — единичный вектор направления полета частиц: $\Omega_x=\sqrt{1-\mu^2}\cos\varphi$ — проекция вектора $\vec{\Omega}$ на ось OX; $\Omega_y=\sqrt{1-\mu^2}\sin\varphi$ — проекция $\vec{\Omega}$ на ось OY; $\Omega_z=\mu$ — проекция $\vec{\Omega}$ на ось OZ (косинус угла между вектором $\vec{\Omega}$ и осью OZ); φ — угол между проекцией $\vec{\Omega}$ на плоскость OXY и осью OX; $n^{(0)}=\int\limits_{-1}^{1}d\mu\int\limits_{0}^{2\pi}Nd\varphi$.

Уравнение (1) решается в области $d=\{(x,y,z)\in L,\,-1\le\mu\le 1,\,0\le\varphi\le 2\pi\}$.

На внешней поверхности задаются граничные условия в виде потока частиц, входящих в тело при $\vec{\Omega} \cdot \vec{n} < 0$, где $\vec{n}$ — внешняя нормаль к поверхности, ограничивающей область L.

Далее рассматривается конечно-разностная аппроксимация уравнения (1) в случае, когда пространственная сетка в области L состоит из прямоугольных параллелепипедов. Значения параметра μ выбираются из интервала $(-1, 1)$, значения параметра φ — из интервала $(0, 2\pi)$.

Уравнение баланса в счетной ячейке в конечно-разностной форме получается с помощью интегро-интерполяционного метода:

$$\mathrm{div}_h\left(\vec{\Omega}N\right) \equiv \Omega_x S_{yz}\left(N_2 - N_1\right) + \Omega_y S_{xz}\left(N_4 - N_3\right) + \Omega_z S_{xy}\left(N_6 - N_5\right) + V\alpha N_0 = V\overline{F}, \qquad (2)$$

где $N_i\,(i = 1, 2, \ldots, 6)$ — средние значения искомой функции N на гранях ячейки; N_0 — значение функции N в центре ячейки; S_{yz}, S_{xz}, S_{xy} — площади граней ячейки; V — объем ячейки; $\overline{F} = \dfrac{1}{4\pi}\left(\beta\overline{n}^{(0)} + Q\right)$. Скалярный поток $n^{(0)}$ в каждой счетной ячейке вычисляется следующим образом: $\overline{n}^{(0)} = \sum\limits_{w=1}^{n_w} N_0^w d\Omega_w$, $\sum\limits_{w=1}^{n_w} d\Omega_w = 4\pi$. Здесь w — номер направления полета частиц; n_w — число направлений полета частиц; $d\Omega_w$ — телесный угол.

Для замыкания системы сеточных уравнений по пространственным переменным используется DD-схема:

$$N_0 = \frac{N_1 + N_2}{2} = \frac{N_3 + N_4}{2} = \frac{N_5 + N_6}{2}. \qquad (3)$$

Система (2), (3) решается итерациями по источнику: $\mathrm{div}_h\left(\vec{\Omega}\overset{s+1}{N}\right) + V\alpha\overset{s+1}{N_0} = V\overset{s}{\overline{F}}$, где s — номер итерации.

Численное решение системы сеточных уравнений переноса нейтронов для одного направления полета частиц осуществляется с помощью алгоритма бегущего счета. Особенности данного алгоритма необходимо учитывать при организации параллельных вычислений на основе принципа пространственной декомпозиции.

Алгоритм бегущего счета

По сути алгоритм бегущего счета — это правило упорядочения ячеек пространственной сетки, которое обеспечивает преобразование разностного оператора к блочно-треугольному виду. Данное правило может быть построено не единственным способом.

На рис. 2, *а* показан пример двумерной сетки с 5 столбцами и 4 строками, на которой стрелками указана передача потоков частиц (далее просто потоков) между ячейками. Цифрами и цветом

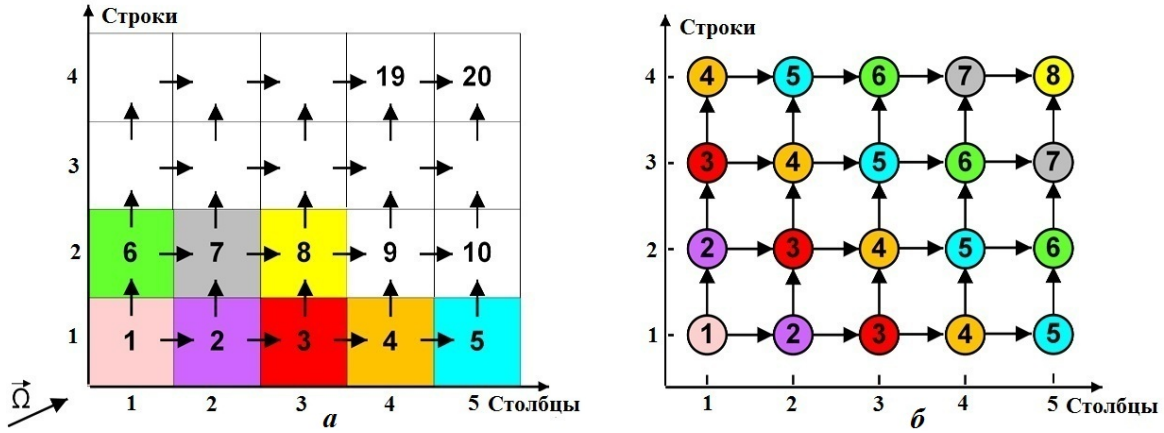

Рис. 2. Алгоритм бегущего счета на ортогональной сетке (*а*) и соответствующий ему орграф $G = (V, E)$ с выделенными гиперплоскостями (*б*)

отражено упорядочение ячеек по сеточным направлениям, которое чаще всего используется при организации бегущего счета на CPU. Циклы по столбцам и строкам имеют зависимость по данным при передаче потоков между ячейками, что мешает их параллельной обработке.

На рис. 2, *б* дано представление этого алгоритма в виде ациклического орграфа $G = (V, E)$, в котором каждой вершине $v \in \{V\}$ соответствует решение уравнения баланса в ячейке фазового пространства, а каждой дуге $e \in \{E\}$ соответствует передача потоков через грани смежных ячеек. Направление дуги определяется освещенностью грани. Цифрами и цветом в орграфе отмечены гиперплоскости [3]. Гиперплоскость — это подмножество вершин орграфа $\{H_{l_{\max}+1}\}$ с одинаковой максимальной длиной пути $l_{\max}$ от истока. Цикл по элементам одной гиперплоскости не имеет зависимостей по данным, что обеспечивает возможность его параллельной реализации.

Распараллеливание между нитями одной группы

Для уровня внутри компактной группы из 32 нитей (warp) одного блока предназначен алгоритм распараллеливания по ячейкам пространственной сетки, принадлежащим одной гиперплоскости $\{H_{l_{\max}+1}\}$. Эффективную реализацию данного алгоритма затрудняют разное число элементов в разных гиперплоскостях ($|H_{l_{\max}+1}| \in [1, \min(n_x, n_y)]$, где n_x, n_y — соответственно число столбцов и строк) и формат представления данных, не обеспечивающий компактного размещения в памяти элементов одной гиперплоскости. Решением перечисленных проблем является алгоритм, основанный на выделении гиперплоскостей фиксированного размера и записи данных в специальных форматах [13].

Фиксированный размер гиперплоскости определяется числом нитей в группе и совпадает с первым размером блока. Номер нити жестко привязан к номеру столбца пространственной сетки.

Сеточные данные записываются в двух форматах, описываемых массивами вида $\left(n_x^{loc}, n_y + n_x^{loc} - 1, n_z, \frac{n_x}{n_x^{loc}}\right)$, где $n_x^{loc} \equiv 32$; $n_y + n_x^{loc} - 1$ — число гиперплоскостей; n_z — число слоев. В зависимости от правила изменения сеточных индексов по столбцам и строкам бегущий счет в двумерном слое реализуется в виде одного из четырех различных орграфов. Для каждого из этих графов есть симметричная пара, полученная сменой направления у всех дуг. На рис. 3 симметричные пары графов обозначены разным цветом дуг. У каждой такой пары графов правило распределения вершин по гиперплоскостям одно и то же, т. е. достаточно одного формата данных. Поэтому для записи сеточных данных требуется всего два формата.

Алгоритм выделения гиперплоскостей фиксированного размера приводит к увеличению множества вершин в исходном графе. Для случая, показанного на рис. 2, алгоритм бегущего счета преобразуется к виду, представленному на рис. 3 слева. Здесь числа обозначают номера строк исходной сетки.

В новом орграфе первое сеточное направление совпадает с элементами одной гиперплоскости и не имеет зависимостей по данным. Вершины нового графа представляют собой объединение двух множеств: счетного $\{V\}$ ("цветные" вершины) и фиктивного $\{V_f\}$ (вершины белого цвета). Счетные вершины соответствуют ячейкам сетки, фиктивные — нет. Решение уравнения баланса выполняется в вершинах обоих типов $\{V\} \cup \{V_f\}$. Выходящий поток определяется с учетом типа вершины и по условию $N^{\text{вых}} = \begin{cases} N^{\text{вых}}, & v \in \{V\} \\ N^{\text{вх}}, & v \in \{V_f\} \end{cases}$, где $v \in \{V\}$, если $i_y \in [1, n_y]$, иначе $v \in \{V_f\}$. Взаимосвязь между индексами строк и гиперплоскостей: $i_y = i_{hyp} - i_x + 1$ — для орграфов на рис. 3 слева; $i_y = i_{hyp} - 32 + i_x$ — для орграфов на рис. 3 справа. Здесь i_x, i_y, i_{hyp} — индексы столбцов, строк и гиперплоскостей, $i_x \in [1, 32]$.

Каждая нить рассчитывает все гиперплоскости и все слои сетки для фиксированного номера столбца. После расчета каждой ячейки требуется взаимодействие нитей для передачи потоков только в первом топологическом направлении (между столбцами). Для этого используются CUDA-функции перестановки в пределах группы нитей `_shfl_up_sync` и `_shfl_down_sync`.

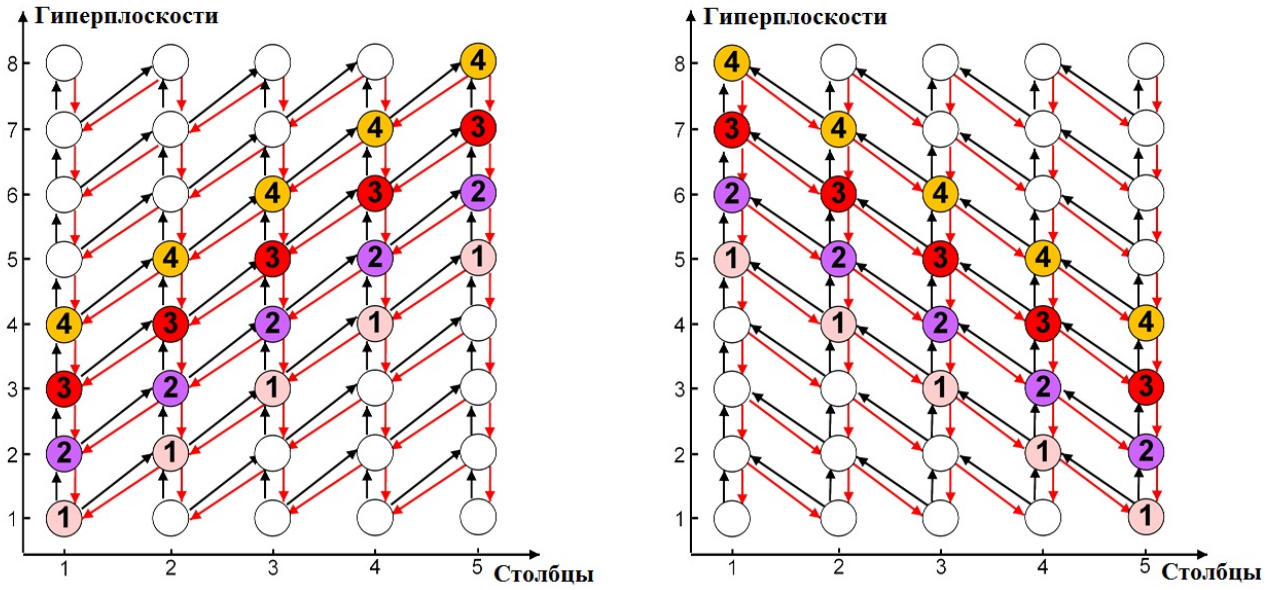

Рис. 3. Орграфы бегущего счета на новых форматах данных

Передача потоков в остальных направлениях не требует взаимодействия нитей, однако она реализована с учетом иерархии памяти на GPU. Для второго топологического направления (между гиперплоскостями одного столбца) используется регистровая память, для третьего — разделяемая память. При этом в алгоритм вводится порционность расчета циклов по гиперплоскостям (n_{hyp}^{loc}) и слоям (n_z^{loc}). Размеры порций определяются с учетом объема доступной разделяемой памяти.

Новые форматы данных требуют дополнительных затрат оперативной памяти. Все массивы увеличиваются в $\dfrac{n_y + n_x^{loc} - 1}{n_y}$ раз из-за перехода от строк к гиперплоскостям. Массивы данных, которые являются инвариантами относительно направлений полета частиц, дополнительно увеличиваются в два раза из-за записи данных в двух форматах (см. рис. 3 слева и справа). Двукратного увеличения общих затрат оперативной памяти не происходит при решении нестационарной задачи переноса нейтронов, т. е. когда в записи уравнения (1) присутствует производная по времени. В этом случае основной объем памяти приходится на массивы с зависимостью от направлений полета частиц, которые можно представить в нужном формате в зависимости от номера октанта.

Предлагаемый алгоритм распараллеливания по элементам гиперплоскости существенно отличается от аналогичных работ [4, 6]. Гиперплоскости выделены в одном трехмерном слое, фиксированный размер гиперплоскостей определен особенностью архитектуры GPU, данные записаны в специальных форматах, введена порционность при расчете циклов.

Зернистость алгоритма распараллеливания между нитями одной компактной группы составляет 1 ячейку фазового пространства задачи.

Распараллеливание между группами нитей одного блока

На уровне множества компактных групп нитей (warp) одного блока реализуется распараллеливание по направлениям полета частиц одного октанта. Этот уровень необходим для снижения нагрузки на подсистему памяти за счет использования разделяемой памяти. При решении уравнения переноса нейтронов часть данных не зависит от направлений полета частиц. Разделяемая память позволяет уменьшить число запросов в глобальную память для данных, инвариантных относительно направлений, в какой-то мере эмулируя работу кэш-памяти.

Распараллеливание между группами нитей одного блока организуется по порции направлений в октанте. Число направлений n_w^{loc} в порции определяет второй размер блока нитей. В тестовой

программе ПАУК используются три массива, которые не зависят от направлений. В разделяемой памяти заводится временный массив вида $(n_x^{loc}, 3)$. Данные читаются из глобальной памяти и пишутся в разделяемую, при этом каждая нить выполняет не более одной операции чтения. Такой подход в n_w^{loc} раз сокращает число запросов к глобальной памяти при работе с указанными массивами, если $n_w^{loc} \geq 3$.

Вычисление скалярного потока нейтронов $\overline{n}^{(0)}$ требует взаимодействия параллельных процессов и реализовано с помощью операции атомарного суммирования, которая эффективно кэшируется.

Зернистость алгоритма распараллеливания между компактными группами нитей одного блока составляет 1 ячейку фазового пространства задачи.

Исследование производительности выполнялось в режиме полной загрузки GPU Nvidia V100 16 GB SXM. Для этого в тесте ПАУК был реализован временный уровень распараллеливания между блоками — по порциям направлений одного октанта.

В табл. 2 представлены результаты исследования производительности и профилирования тестовой программы ПАУК в зависимости от формата представления данных и размера блока по направлениям. Метрика производительности вычислялась как число неизвестных фазового пространства, найденных за секунду, по формуле $R = 8\, n_x n_y n_z n_w^{oct} n_{it}/t$, где t — время в секундах, n_{it} — число итераций. Значения остальных метрик получены с помощью средства профилирования nvprof. Результаты получены при следующих параметрах теста: $n_x = 32$; $n_y = 169$; $n_{hyp} = 200$; $n_z = 4$; число направлений в октанте $n_w^{oct} = n_w^{loc} N_{SM} b_{SM} = 1\,600$, где $N_{SM} = 80$ — число SM, b_{SM} — число активных блоков на одном SM. Суммарное число нитей — 51 200.

Важной особенностью метрики R является расчет относительно числа строк, а не гиперплоскостей в задаче. Доля счетных вершин в графе бегущего счета $\dfrac{|V|}{|V| + |V_f|} = \dfrac{n_y}{n_y + n_x^{loc} - 1}$. Значения остальных метрик получены при вычислении всех вершин.

Результаты табл. 2 демонстрируют решающее влияние подсистемы памяти на производительность приложения. У данной модели GPU пропускная способность глобальной памяти составляет 900 Гбайт/с. В режиме одного направления в блоке не происходит сокращения числа запросов к глобальной памяти при работе с инвариантными массивами, что приводит к увеличенной нагрузке на память. В случае построчного формата соседние нити обращаются к данным, которые расположены в памяти со смещением в $n_x^{loc} \pm 1$ чисел в формате двойной точности, что также приводит к работе подсистемы памяти на пределе возможности — об этом свидетельствует значение метрики *Global Load Throughput*. Подсистема памяти не является ограничителем производительности только при записи данных в диагональных форматах и дополнительном распараллеливании по направлениям в блоке.

Таблица 2

Характеристики производительности вариантов программы ПАУК

Вариант программы	R, млрд. яч./с	FLOP Efficiency (Peak Double), %	Warp Execution Efficiency, %	Global Load Throughput, Гбайт/с
Диагональные форматы, 1 направление в блоке	14,9	16,1	89,4	860,4
Диагональные форматы, 4 направления в блоке	38,7	45,4	88,8	586,2
Построчный формат, 4 направления в блоке	17,3	18,1	83,6	937,6

Распараллеливание между блоками на основе геометрической декомпозиции

Для распараллеливания на следующем уровне архитектуры и программной модели используется KBA-алгоритм [14] на основе геометрической декомпозиции. Исходная пространственная сетка разбивается по столбцам и гиперплоскостям на геометрические фрагменты. Разбиение выполняется

регулярным образом, без перехлестов. Фиксированный размер гиперплоскостей учитывается при декомпозиции сетки. Предполагается жесткая привязка фрагмента к вычислительному ресурсу. Один блок выполняет расчет одного геометрического фрагмента.

Алгоритм бегущего счета в слое в случае геометрической декомпозиции имеет двухуровневую реализацию: по фрагментам и по ячейкам в каждом из этих фрагментов. На рис. 4, *а* представлен пример орграфа бегущего счета по ячейкам. Цифры в вершинах этого графа соответствуют номерам строк исходной сетки. Красными линиями выделены границы геометрических фрагментов.

На рис. 4, *б* представлен пример орграфа бегущего счета по фрагментам. Вершина здесь соответствует отдельному фрагменту и локальному бегущему счету по всем ячейкам этого фрагмента. Цифры в вершинах этого графа соответствуют номерам гиперплоскостей по фрагментам.

Орграфы на рис. 4 демонстрируют несколько важных особенностей в организации параллельного бегущего счета по фрагментам, которые обусловлены выбранными форматами представления данных. Эти особенности касаются объемов передаваемых данных и правила синхронизации.

Пусть n_x^{loc}, n_{hyp}^{loc} — размеры фрагментов по столбцам и гиперплоскостям. Передача потоков между фрагментами выполняется:

- при декомпозиции по столбцам для n_{hyp}^{loc} граней;
- при декомпозиции по гиперплоскостям для $2n_x^{loc} - 1$ граней.

Правило синхронизации между фрагментами учитывает максимальную длину пути в графе от источника до вершины, а также смещение гиперплоскостей через каждые n_x^{loc} столбцов. Длительность выполнения алгоритма параллельного бегущего счета по фрагментам можно оценить по формуле

$$T_{sweep} = n_{hyp}^{loc}(1 + (D_{hyp} - 1) + \left\lfloor \frac{n_x^{loc} + n_{hyp}^{loc}}{n_{hyp}^{loc}} \right\rfloor (D_x - 1)),$$

где "скобки" $\lfloor\ \rfloor$ используются для обозначения целой части числа; D_x, D_{hyp} — число фрагментов по столбцам и по гиперплоскостям:

$$D_x = \frac{n_x}{n_{nx}^{loc}}, \quad n_{nx}^{loc} \equiv 32; \qquad D_{hyp} = \frac{n_y + n_x^{loc} - 1}{n_{hyp}^{loc}}.$$

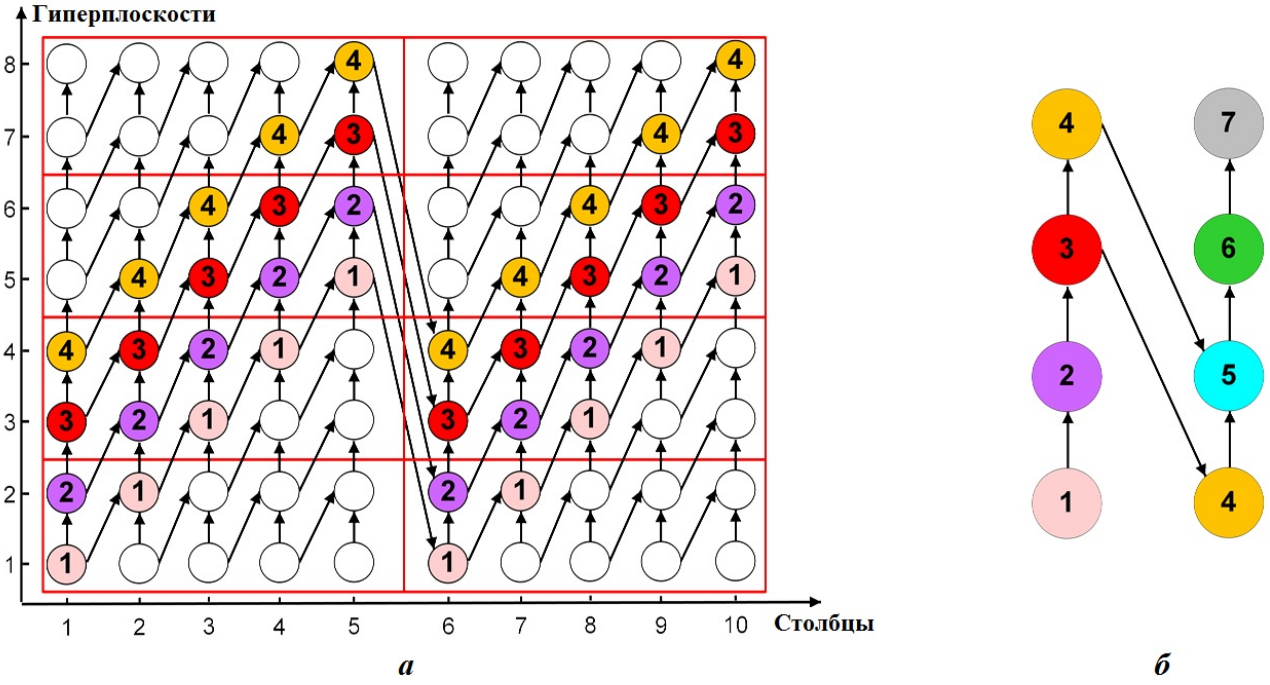

Рис. 4. Орграфы бегущего счета по ячейкам (*а*) и по фрагментам (*б*)

Параллельный бегущий счет можно организовать одновременно для порции в n_z^{loc} слоев ($n_z^{loc} \in \in [1, n_z]$), так как бегущий счет по ячейкам каждого слоя имеет одинаковую реализацию в виде орграфа. Размер порции по слоям определяет частоту межблочного взаимодействия и зернистость алгоритма распараллеливания.

Для эффективной загрузки всех блоков организуется конвейер по слоям пространственной сетки. В этом случае значение эффективности распараллеливания оценивается по формуле

$$E_{KBA} = \frac{\dfrac{n_z}{n_z^{loc}}}{\dfrac{n_z}{n_z^{loc}} + (D_{hyp} - 1) + \left\lfloor \dfrac{n_x^{loc} + n_{hyp}^{loc}}{n_{hyp}^{loc}} \right\rfloor (D_x - 1)} \cdot 100\,\%, \tag{4}$$

где n_z — число слоев сетки, рассчитываемых на одном GPU.

Передача потоков между блоками осуществляется через глобальную память GPU. Для указания компилятору особого режима работы с массивами для передачи потоков используется спецификатор `volatile`, запрещающий различные оптимизации при работе с такими массивами [1].

Параллельный конвейер требует организации вычислений в предопределенной последовательности, которая регулируется с помощью механизмов синхронизации. На программе ПАУК опробованы два механизма межблочной синхронизации: кооперативные группы и атомарные функции.

Для синхронизации с помощью кооперативных групп необходим режим раздельной компиляции. Такая компиляция приводит к увеличению нагрузки на регистровый файл и снижению значения параметра b_{SM} с 5 до 4 на GPU Nvidia Volta. Кроме того, синхронизация с помощью кооперативных групп реализуется по глобальной схеме — сразу по всей сети блоков. В конвейере по слоям необходимо столько точек глобальной синхронизации, сколько всего тактов в конвейере, что равно знаменателю дроби в формуле (4).

Синхронизация с помощью атомарных функций `atomicCAS` и `atomicExch` обеспечивает возможности, аналогичные замкам в стандарте OpenMP. Этот способ межблочной синхронизации реализуется по локальной схеме — для каждой дуги орграфа параллельного бегущего счета (см. рис. 4, *б*) в отдельности. В конвейере по слоям необходимо столько точек локальной синхронизации, сколько счетных тактов в конвейере, что равно числителю дроби в формуле (4).

Результаты исследований производительности программы ПАУК на GPU Nvidia V100 16 GB SXM для двух механизмов синхронизации приведены в табл. 3. Результаты получены при $n_x^{loc} = 32$; $n_{hyp}^{loc} = 8$; $n_z^{loc} = 4$; $n_z = 800$; $n_w^{loc} = 4$.

Из табл. 3 следует, что реализация с помощью атомарных функций обеспечивает более высокие характеристики производительности, так как требует меньше накладных расходов на синхронизацию и меньше регистров на счетное ядро.

Зернистость алгоритма распараллеливания между блоками составляет $n_{hyp}^{loc} n_z^{loc}$ ячеек фазового пространства задачи.

Предлагаемый алгоритм распараллеливания между блоками на основе геометрической декомпозиции существенно отличается от аналогичных алгоритмов из работ [10—12]. На GPU реализуется полноценный KBA-алгоритм на основе двумерной декомпозиции, а не только распараллеливание по элементам одной гиперплоскости. Для межблочной синхронизации используется механизм, не требующий завершения счетного ядра на GPU.

Таблица 3

Производительность тестовой программы ПАУК для двух способов синхронизации и различного числа блоков

Механизм синхронизации	Общее число блоков	R, млрд яч./с
Кооперативные группы	320	18,1
Атомарные функции	320	21,5
	400	24,5

Распараллеливание между группами блоков

Для уменьшения нагрузки на подсистему памяти в алгоритм распараллеливания между блоками введен дополнительный уровень распараллеливания по порциям направлений полета частиц одного октанта. Этот уровень позволяет логически разделить сеть блоков на равные группы, между которыми не требуется взаимодействия по данным и управлению в рамках конвейера по слоям.

Сеть блоков имеет размеры $\frac{n_x}{n_x^{loc}} \times \frac{n_y + n_x^{loc} - 1}{n_{hyp}^{loc}} \times \frac{n_w^{GPU}}{n_w^{loc}}$, где n_w^{GPU} обозначает общее число направлений для распараллеливания на двух уровнях — между группами нитей внутри блока и между группами блоков. Один блок имеет размеры $n_x^{loc} \times n_w^{loc}$.

Из табл. 4 видно, что алгоритм распараллеливания на основе геометрической декомпозиции обеспечивает достаточно эффективную загрузку GPU. В режиме комбинированного распараллеливания между блоками (по геометрии и направлениям) достигнута эффективность вычислений 30,6 % от пиковой производительности при использовании всего 16 направлений полета частиц.

Зернистость алгоритма распараллеливания между группами блоков составляет $n_{hyp}^{loc} n_z^{loc} \left[\frac{n_z}{n_z^{loc}} + \right.$ $\left. + (D_{hyp} - 1) + \left\lfloor \frac{n_x^{loc} + n_{hyp}^{loc}}{n_{hyp}^{loc}} \right\rfloor (D_x - 1) \right]$ ячеек фазового пространства задачи.

Таблица 4

Производительность тестовой программы ПАУК на GPU Nvidia V100 16 GB SXM для алгоритмов распараллеливания между блоками ($n_x^{loc} = 32$; $n_{hyp}^{loc} = 8$; $n_w^{loc} = 4$; $n_z = 400$)

Алгоритм распараллеливания между блоками	Сеть блоков $D_x \times D_{hyp} \times n_w^{GPU}/n_w^{loc}$	n_w^{GPU}	E_{KBA}, %	R, млрд яч./с	FLOP Efficiency (Peak Double), %
По направлениям	$1 \times 1 \times 400$	1 600	100	38,9	45,4
По геометрии	$8 \times 50 \times 1$	4	82,6	24,5	–
Комбинированный	$4 \times 25 \times 4$	16	91,1	28,0	30,6

Распараллеливание между GPU

Для распараллеливания на уровне множества GPU предназначена модификация KBA-алгоритма. Исходная пространственная сетка разбивается по трем пространственным направлениям на геометрические подобласти. Декомпозиция выполняется регулярным образом, без перехлестов. На одном GPU рассчитывается одна геометрическая подобласть. Алгоритм реализуется в модели распределенной памяти с помощью средств стандарта MPI.

Из-за формата представления данных с выделением гиперплоскостей фиксированного размера были рассмотрены два способа разбиения по второму топологическому направлению:

1) декомпозиция по строкам: сначала выполняется разбиение сетки на геометрические подобласти, затем происходит смена формата в каждой подобласти (рис. 5, *а*). Число гиперплоскостей в задаче равно $N_y + P_2(n_x^{loc} - 1)$, где N_y — число строк; P_2 — число разбиений во втором топологическом направлении;

2) декомпозиция по гиперплоскостям: сначала происходит смена формата в задаче, а затем выполняется разбиение на подобласти (рис. 5, *б*). Число гиперплоскостей в задаче равно $N_y + n_x^{loc} - 1$.

Число вершин в графе, а значит, и объем вычислений меньше при декомпозиции по гиперплоскостям. Число гиперплоскостей, рассчитываемых в одном CUDA-блоке:

$$n_{hyp}^{loc} = \begin{cases} \dfrac{N_y + P_2(n_x^{loc} - 1)}{P_2 D_{hyp}} & \text{при декомпозиции по строкам;} \\ \dfrac{N_y + n_x^{loc} - 1}{P_2 D_{hyp}} & \text{при декомпозиции по гиперплоскостям.} \end{cases}$$

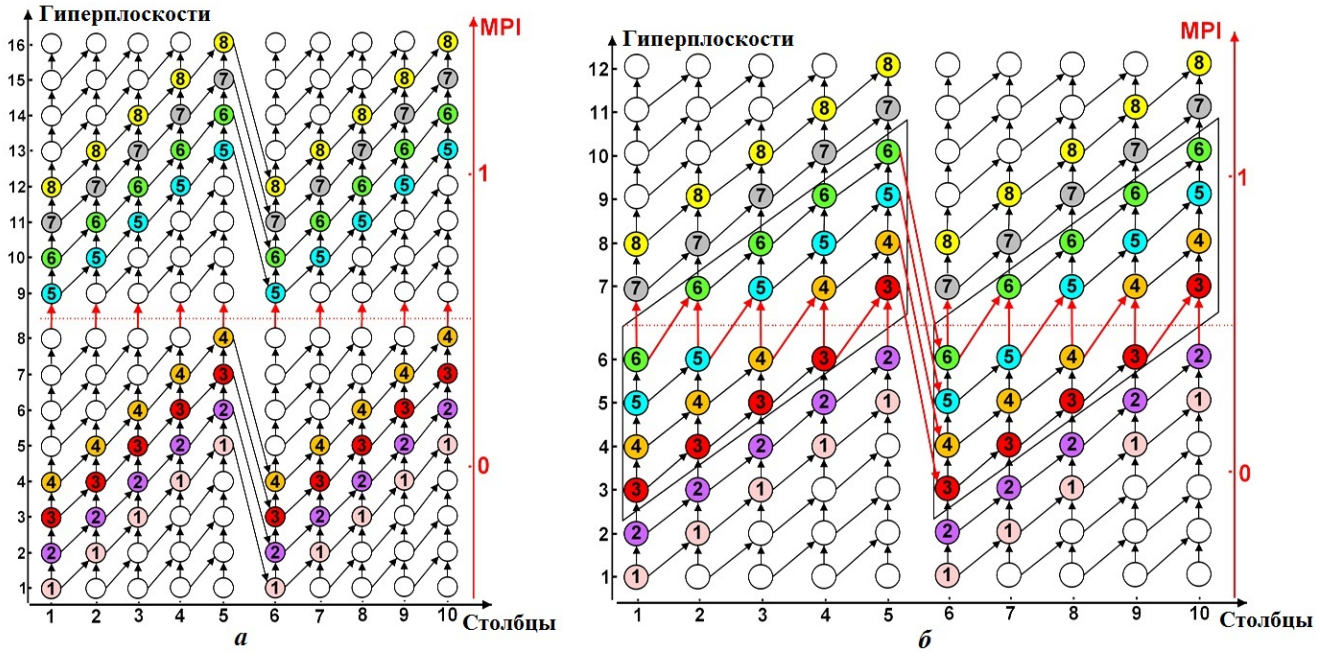

Рис. 5. Алгоритм бегущего счета при различных способах декомпозиции во втором топологическом направлении: *а* — по строкам; *б* — по гиперплоскостям

Число дуг в графе, которые соответствуют передаче потоков через MPI, значительно меньше при декомпозиции по строкам. На рис. 5, *а,б* такие дуги обозначены красным цветом. Объем данных (в байтах), передаваемых между двумя подобластями в рамках одного алгоритма параллельного бегущего счета:

$$V_{sweep} = \begin{cases} 8(n_x^{loc} D_x) n_z n_w^{GPU} & \text{при декомпозиции по строкам;} \\ 8\left[(2n_x^{loc} - 1)D_x + (D_x - 1)n_x^{loc}\right] n_z n_w^{GPU} & \text{при декомпозиции по гиперплоскостям.} \end{cases}$$

Нагрузка на коммуникационную подсистему при декомпозиции по гиперплоскостям практически в 3 раза больше, чем при декомпозиции по строкам. Кроме того, в силу необходимости представления данных в двух диагональных форматах (см. рис. 3) декомпозиция по гиперплоскостям приводит к частичному перехлесту подобластей. На рис. 5, *б* зона перехлеста обозначена штриховкой. Перехлест происходит из-за независимой декомпозиции орграфов для двух форматов представления данных: $i_{hyp} = i_x + i_y - 1$ и $i_{hyp} = n_x^{loc} - i_x + i_y$. Для каждой границы разбиения по гиперплоскостям потребуется дополнительный обмен информацией в объеме $V_{halo} = 8n_x^{loc} D_x (n_x^{loc} - 1) n_z$ байт. Этот обмен необходимо выполнить один раз на итерации по правой части решения уравнения переноса.

Еще одним недостатком разбиения по гиперплоскостям является усложнение графа параллельного бегущего счета и неполная загрузка GPU в случае $n_x > n_x^{loc}$ (см. рис. 5, *б*).

С учетом указанных обстоятельств для распараллеливания на рассматриваемом уровне архитектуры (множество GPU) выбрана декомпозиция по строкам, а не по гиперплоскостям.

Алгоритм параллельного бегущего счета по подобластям выполняется для всех слоев подобласти и для n_w^{GPU} направлений одного октанта. Конвейер организуется по направлениям углового октанта. В случае решения многогрупповой задачи количество тактов в конвейере может быть увеличено за счет групп, но не за счет слоев. Восемь (по числу угловых октантов) конвейеров обрабатываются в предопределенной последовательности и с пересечением, т. е. без точек глобальной синхронизации между ними.

Теоретическая эффективность алгоритма MPI-распараллеливания рассчитывается по формуле

$$E_{MPI} = \frac{8\dfrac{n_w^{oct}}{n_w^{GPU}}}{8\dfrac{n_w^{oct}}{n_w^{GPU}} + 4(P_1 - 1) + 4(P_2 - 1) + 2(P_3 - 1)} \cdot 100\,\%, \tag{5}$$

где n_w^{oct} — число всех направлений в октанте; P_1, P_2, P_3 — параметры пространственной декомпозиции по столбцам, строкам и слоям.

Анализ формулы (5) показывает, что алгоритм MPI-распараллеливания обеспечивает теоретическую (без учета коммуникационной составляющей) масштабируемость до 70 000 GPU с эффективностью выше 60 % для задач с уровнем дискретизации *32 группы и квадратура* S_{16}.

Зернистость алгоритма MPI-распараллеливания можно определить двумя способами:

– относительно объема вычислений, выполняемых одной CUDA-нитью, —

$$n_{hyp}^{loc} n_z^{loc} \left[\frac{n_z}{n_z^{loc}} + (D_{hyp} - 1) + \left[\frac{n_x^{loc} + n_{hyp}^{loc}}{n_{hyp}^{loc}} \right] (D_x - 1) \right] \text{ячеек.}$$

– относительно объема вычислений, выполняемых всеми CUDA-нитями за один вызов на GPU программы бегущего счета по ячейкам подобласти для n_w^{GPU} направлени, —

$$\frac{n_x \left(n_y + n_x^{loc} - 1 \right) n_z n_w^{GPU}}{E_{KBA}/100\,\%} \text{ячеек.}$$

Программная реализация MPI-обменов в тесте ПАУК выполнена по схеме прямой передачи информации между GPU, т. е. без промежуточного копирования в память универсального процессора.

В табл. 5 представлены результаты производительности и эффективности распараллеливания тестовой программы ПАУК в режиме Multi-GPU. Исследование выполнено методом слабой масштабируемости или методом увеличения задачи. Эффективность распараллеливания рассчитана по формуле

$$E_n = \frac{R_n}{nR_1} \cdot 100\,\%,$$

где n — число GPU, т. е. MPI-процессов; $R_n = 8nn_x n_y n_z n_w^{oct} n_{it}/t$ — производительность, или скорость счета на n GPU; t — время счета в секундах.

При тестировании использовались сервер с двумя GPU Nvidia V100 16 GB SXM и сервер с четырьмя GPU Nvidia A100 40 GB SXM из состава программно-аппаратного полигона НЦФМ.

Переход с V100 на A100 потребовал увеличить размеры сети блоков из-за увеличения значений параметров N_{SM} и b_{SM}. При смене GPU прирост фактической производительности теста ПАУК в 1,325 раза оказался больше прироста в 1,244 раза пиковой производительности, что объясняется влиянием подсистемы памяти и особенностью расчета метрики R. Доля счетных вершин в тесте на GPU типа V100 составляет 0,845, на A100 — 0,871.

Эффективность распараллеливания на два MPI-процесса выше на сервере с A100, чем на сервере с V100, за счет двукратного увеличения пропускной способности интерфейса NVLink.

Таблица 5

Масштабируемость тестовой программы ПАУК $\left(\boldsymbol{n_x^{loc} = 32;\ n_{hyp}^{loc} = 8;\ n_w^{loc} = 4;\ n_w^{oct}/n_w^{GPU} = 10}\right)$

Тип GPU	Сеть блоков	n_z	E_{KBA}, %	R_1, млрд яч./с	E_2, %	E_4, %
V100	$4 \times 25 \times 4$	400	91,1	28,0	92,6–95,9	–
A100	$5 \times 30 \times 5$	500	91,1	37,1	94,8–96,5	89,3–93,0

Удельная характеристика производительности

Характерным критерием качества при оценке производительности GPU-версии приложения является сравнение с производительностью CPU-версии. Для тестовой программы ПАУК такое сравнение выполнено с помощью удельной характеристики. Вычислительная эффективность — это производительность приложения на один Гфлоп/с пиковой производительности задействованных вычислительных устройств.

В табл. 6 приведены результаты, полученные на универсальных процессорах по версии ПАУК с распараллеливанием MPI+MPI-3 SHM+OpenMP и векторизацией по направлениям полета частиц [13]. При тестировании использовался сервер с двумя CPU Intel Broadwell с суммарной пиковой производительностью 1,165 Тфлоп/с и сервер с двумя CPU Intel Skylake с суммарной пиковой производительностью 3,456 Тфлоп/с.

Для GPU указаны два значения вычислительной эффективности, рассчитанные относительно счетного и общего числа вершин (в скобках) в графах бегущего счета. На GPU Nvidia и CPU Intel Skylake получены очень близкие значения удельной производительности, что позволяет утверждать об эффективной адаптации тестовой программы ПАУК к архитектуре GPU.

Таблица 6

Вычислительная эффективность тестовой программы ПАУК, (млн яч.)/Гфлоп

2 CPU Intel Broadwell	2 CPU Intel Skylake	1 GPU Nvidia V100	1 GPU Nvidia A100
5,75	4,34	3,57 (4,22)	3,81 (4,38)

Заключение

В работе представлен иерархический алгоритм распараллеливания вычислений при решении трехмерного уравнения переноса нейтронов на гибридных супер-ЭВМ с GPU. Алгоритм основан на выделении уровней архитектуры и применении принципа геометрической декомпозиции. Программная реализация выполнена на примере тестовой программы ПАУК с помощью технологии CUDA.

Задача эффективной загрузки GPU решена при минимальном использовании ресурса направлений — 16 из 51 200 нитей на GPU Nvidia V100 и 20 из 96 000 нитей на GPU Nvidia A100. Зернистость алгоритмов распараллеливания по пространственным переменным (в ячейках фазового пространства) составила:

- для одной CUDA-нити на всех уровнях одного GPU:
  - 1 при распараллеливании по ячейкам между CUDA-нитями одной группы (warp);
  - 8 при распараллеливании по фрагментам между CUDA-блоками;
- для одного GPU на уровне MPI-процессов — $1{,}8 \cdot 10^8$ $\left(4{,}2 \cdot 10^8\right)$ при распараллеливании по подобластям между различными GPU.

У вспомогательных алгоритмов распараллеливания по направлениям зернистость для одной CUDA-нити составила:

- 1 на уровне warp-ов одного блока;
- 3 512 (4 392) на уровне групп блоков.

Эффективность адаптации теста ПАУК к одному GPU подтверждена результатами профилирования и сравнением с производительностью CPU-версии программы. Для метрики *FLOP Efficiency (Peak Double)* получено значение 30,6 % от пиковой производительности GPU Nvidia V100. Для метрики *Вычислительная эффективность* (фактическая производительность приложения на 1 Гфлоп/с пиковой производительности) получены близкие результаты на GPU Nvidia V100, GPU Nvidia A100 и CPU Intel Skylake.

Масштабируемость тестовой программы ПАУК в режиме multi-GPU исследована методом умножения. В зависимости от способа декомпозиции эффективность распараллеливания составила 89,3—93,0 % для четырех GPU. Алгоритм MPI-распараллеливания обеспечивает теоретическую (без учета коммуникационной составляющей) масштабируемость до 70 000 GPU с эффективностью выше 60 % для задач с уровнем дискретизации *32 группы и квадратура* S_{16}.

Исследование выполнено в рамках научной программы Национального центра физики и математики (проект "Национальный центр исследования архитектур суперкомпьютеров").

Список литературы

1. Информация о CUDA. https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html. Informatsiya o CUDA. https://docs.nvidiA.com/cuda/cuda-toolkit-release-notes/index.html.
2. *Бочков А. И., Нуждин А. А.* Параллельный алгоритм решения трехмерного кинетического уравнения переноса. Программа ПАУК для тестирования многопроцессорных вычислительных систем // Межд. науч. конф. "Параллельные вычислительные технологии (ПАВТ'2008)": Тр. межд. науч. конф. (С.-Пб., 28 января — 1 февраля 2008 г.) Челябинск: ЮУрГУ, 2008. С. 39—47.
Bochkov A. I., Nuzhdin A. A. Parallelny algoritm resheniya trekhmernogo kineticheskogo uravneniya perenosa. Programma PAUK dlya testirovaniya mnogoprotsessornykh vychislitelnykh sistem // Mezhd. nauch. konf. "Parallelnye vychislitelnye tekhnologii (PAVT'2008)": Tr. mezhd. nauch. konf. (S.-Pb., 28 yanvarya — 1 fevralya 2008 g.) Chelyabinsk: YuUrGU, 2008.
3. *Lamport L.* The parallel execution of DO loops // Communications of the ACM. 1974. Vol. 17, No 2. P. 83—93.
4. *Evans T. M., Joubert W., Hamilton S. P., Johnson S. R., Turner J. A., Davidson G. G., Pandya T. M.* Three-dimensional discrete ordinates reactor assembly calculations on GPUs // ANS MC2015 — Joint Int. Conf. on Mathematics and Computation, Supercomputing in Nuclear Applications and the Monte Carlo Method. Nashville, Tennessee. April 19—23, 2015.
5. *Searles R., Chandrasekaran S., Joubert W., Hernandez O.* MPI + OpenACC: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems // Comput. Phys. Commun. 2019. Vol. 236. P. 176—187.
6. *Kunen A., Loffeld J., Black A., Chen R., Nowak P., Haut T., Bailey T., Brown P., Rennich S., Maginot P., Tagani B.* Porting 3D discrete ordinates sweep algorithm in Ardra to CUDA // Proc. Int. Conf. on Mathematics and Computational Methods Applied to Nuclear Science and Engineering. Portland, USA, 2019. P. 2585—2598.
7. The OpenACC API Specification for Parallel Programming. https://www.openacc.org/.
8. *Beckingsale D. A., Burmark J., Hornung R., Jones H., Killian W., Kunen A. J., Pearce O., Robinson P., Ryujin B. S., Scogland T. R. W.* RAJA: Portable performance for large-scale scientific fpplications // 2019 IEEE/ACM Int. Workshop on Performance, Portability and Productivity in HPC (P3HPC). DOI 10.1109/P3HPC49587.2019.00012.
9. *Zerr R. J., Baker R. S.* SNAP: SN (Discrete Ordinates) Application Proxy: Description. Tech. Rep. LA-UR-13-21070. Los Alamos National Laboratories. 2013.
10. *Deakin T., McIntosh-Smith S., Gaudin W.* Many-core acceleration of a discrete ordinates transport mini-app at extreme scale // 31st Int. Conf. ISC High Performance 2016. Frankfurt, Germany, Proceedings. Cham: Springer International Publishing, 2016. P. 429—448.
11. *Pennycook S., Mudalige G., Hammond S., Jarvis S.* Parallelising wavefront applications on general-purpose GPU devices // 26th UK Performance Engineering Workshop (UKPEW10). 2010. P. 111—118.
12. *Villa O., Johnson D. R., O'Connor M., Bolotin E., Nellans D., Luitjens J., Sakharnykh N., Wang P., Micikevicius P., Scuidero A., Keckler S. W., Dally W. J.* Scaling the power wall: a

path to exascale // Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis. IEEE Press, 2014. P. 830—841. https://doi.org/10.1109/SC.2014.73.

13. *Нуждин А. А.* Тестовая программа ПАУК как полигон для апробации алгоритмов и технологий параллельного программирования // Вопросы атомной науки и техники. Сер. Математическое моделирование физических процессов. 2020. Вып. 4. С. 48—61.
Nuzhdin A. A. Testovaya programma PAUK kak poligon dlya aprobatsii algoritmov i tekhnologiy parallelnogo programmirovaniya // Voprosy atomnoy nauki i tekhniki. Ser. Matematicheskoe modelirovanie fizicheskikh protsessov. 2020. Vyp. 4. S. 48—61.
14. *Koch K. R., Baker R. S., Alcouffe R. E.* Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor // Trans. of the Amer. Nuc. Soc. 1992. Vol. 65. P. 198.

Статья поступила в редакцию 16.03.22.