ТРЕХУРОВНЕВАЯ MPI+TBB+CUDA ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ БЛОЧНОГО ИТЕРАЦИОННОГО АЛГОРИТМА РЕШЕНИЯ СЛАУ ДЛЯ МЕЛКОБЛОЧНЫХ НЕСТРУКТУРИРОВАННЫХ РАЗРЕЖЕННЫХ МАТРИЦ

И.Н. Коньшин, Г.Б. Сушко, С.А. Харченко

Введение. В последнее время использование графических процессоров в качестве ускорителей в вычислительных системах гибридной архитектуры показало большой потенциал таких вычислительных систем для резкого повышения производительности вычислений. Гибридные архитектуры считаются одним из перспективных направлений в контексте создания суперкомпьютеров экзафлопсного класса, поскольку графические процессоры демонстрируют рекордные показатели производительности на ватт. С другой стороны, эффективное использование ускорителей на основе графических процессоров часто требуют от разработчиков программного обеспечения больших усилий, и не всегда результат подобных исследований приводит к реальному существенному ускорению приложений. В данной работе рассматривается задача об эффективной реализации на суперкомпьютерах гибридной архитектуры алгоритмов решения систем линейных алгебраических уравнений (СЛАУ) с плохообусловленными неструктурированными разреженными матрицами коэффициентов.

Сложность рассматриваемой задачи состоит в комбинации особенностей ее постановки. Для плотных матриц подобные прямые алгоритмы существуют — например, широко известный тест Linpack. Известны некоторые эффективные реализации алгоритмов решения СЛАУ на суперкомпьютерах гибридной архитектуры для структурированных разреженных матриц — как прямые, так и итерационные. Для неструктурированных разреженных хорошо обусловленных матриц также известны некоторые эффективные реализации итерационных алгоритмов, основанные на простейших предобусловливаниях типа метода Якоби. Для плохообусловленных неструктурированных разреженных матриц необходимо с одной стороны строить высококачественное предобусловливание для обеспечения сходимости к решению СЛАУ, а с другой стороны — обеспечить высокую эффективность вычислений на гибридной архитектуре на основных вычислительноемких этапах алгоритма.

Многоуровневая MPI+TBB+CUDA параллельная реализация предлагаемого алгоритма обусловлена трехуровневой организацией памяти гибридной архитектуры. На верхнем уровне распараллеливание происходит по распределенной памяти вычислительных узлов, используя межпроцессорные обмены с помощью MPI. На следующем уровне по общей памяти узла синхронизация вычислений происходит с помощью библиотеки Intel® TBB. На самом нижнем уровне распараллеливание происходит на языке программирования CUDA для использования всех возможностей работы со сложной иерархической памятью ускорителей на основе графических процессоров.

Работы выполнены ООО «ТЕСИС» по договору с ФГУП РФЯЦ-ВНИИЭФ (г. Саров), 2012 г.

Особенности архитектуры графических ускорителей. В последнее время ускорители на основе графических процессоров (GPGPU) все чаще применяются для решения задач общего назначения, накак не связанными с обработкой графических изображений. Будем для краткости в дальнейшем называть такие устройства термином ГПУ.

ГПУ ускорители обычно являются комплексом, включающим в себя следующие составляющие: набор кластеров потоковых процессоров, кэш-память, доступная всем потоковым процессорам, несколько контроллеров памяти, обеспечивающих доступ к ОЗУ устройства и коммуникационной сети. Вычислительные ядра выполняются на ГПУ ускорителе параллельно в несколько нитей, сгруппированных в блоки нитей, которые могут быть одно-, двух- или трехмерными. Блоки нитей, в свою очередь, сгруппированы в вычислительную сеть, которая может быть одно- или двумерной. Одна нить исполняется на одном потоковом процессоре, а один блок нитей — на одном потоковом мультипроцессоре. Нити выполняют одни и те же команды над различными данными, положение которых в памяти определяется нитью исходя из информации об идентификаторе блока нитей и идентификаторе нити внутри блока нитей. Имеется барьерный механизм синхронизаций нитей одного блока. Аппаратный механизм синхронизации нитей различных блоков нитей отсутствует.

В ГПУ ускорителе предусматривается несколько уровней памяти: регистровая, локальная, разделяемая память блока, глобальная, текстурная и константная память. Каждая нить имеет доступ к своей локальной памяти и к регистровой памяти. Все нити одного блока нитей имеют доступ к разделяемой памяти блока. Все нити, выполняемые на одном устройстве, имеют доступ к глобальной памяти на чтение и запись, а также к текстурной и константой памяти только на чтение.

Таким образом вычисления на ГПУ проводятся несколькими блоками нитей, причем нити синхронно выполняют единый набор команд, а последовательность выполнения блоков нитей определяется самим ГПУ автоматически. Это подразумевает использование не менее чем двухуровневого параллелизма, причем параллелизм на уровне нитей внутри блока должен быть по возможности однородным. Кроме того, для

максимальной эффективности работы ГПУ необходимо по возможности обеспечить максимальный объем вычислений с локальными данными нитей ГПУ при минимальной подкачке данных из глобальной памяти ГПУ. С точки зрения простейших операций линейной алгебры это означает, что наиболее эффективными на ГПУ должны быть операции типа BLAS3 такие, например, как умножение плотной матрицы на плотную матрицу.

Мелкоблочные разреженные матрицы и блочные итерационные алгоритмы. Моделирование задач математической физики часто подразумевает наличие в каждой ячейке расчетной сетки нескольких физических переменных, таких как скорость, давление, плотность, температура, и т. д. Использование в подобных случаях схем аппроксимации с совместным решением групп уравнений естественным образом приводит к совместным СЛАУ с внутренней мелкоблочной структурой: в этом случае с каждой ячейкой сетки ассоциируется набор неизвестных, а все уравнения связей между неизвестными задачи записываются в виде набора блочных связей между такими группами неизвестных. В результате возникают системы уравнений с мелкоблочными матрицами, в которых элемент связи есть плотная подматрица, внутренняя разреженность которой в вычислениях не учитывается. Большинство алгоритмов решения СЛАУ могут быть естественным образом переформулированы для мелкоблочных матриц с соответствующей заменой умножения элементов на произведение плотных подматриц малого размера, взятие обратного элемента на вычисление обратной к соответствующей плотной подматрице малого размера, и т. д. Введение мелкоблочных матриц в частности позволяет свести вычисление неполного треугольного разложения матрицы к набору операций типа BLAS3.

Предобусловленные итерационные алгоритмы решения СЛАУ включают в себя как правило следующие операции: операции с векторами типа DOT и DAXPY, ортогонализации векторов, умножение матрицы и транспонированной матрицы на вектор, решение треугольной системы уравнений с верхней и нижней треугольной матрицами. Для мелкоблочных матриц это есть операции типа BLAS1 и BLAS2. Для обеспечения использования в основном операций типа BLAS3 в итерационной схеме можно использовать так называемые блочные итерационные алгоритмы, в которых итерируется не один вектор, а сразу набор (блок) векторов, как, например, в блочном алгоритме сопряженных градиентов (BCG, Block Conjugate Gradient) [6].

Рассмотрим влияние размера блока на число блочных итераций в блочном итерационном алгоритме типа подпространства Крылова SOFGMRES [1] на примере задаче механики НДС «resh2_4», предоставленной РФЯЦ ВНИИЭФ (г. Саров): N = 569 217, число ненулевых элементов NZA = 37 826 212, эксперименты проводились для различных режимов параллельных вычислений по общей памяти ЦПУ. Строилось мелкоблочное предобусловливание с размером каждого мелкого блока 3x3, предобусловливание хранилось в формате «float». СЛАУ решалась с использованием предобусловленного алгоритма SOFGMRES с относительной точностью $\varepsilon = 10^{-9}$ по невязке. В таблицах ниже использованы следующие обозначения: «Ncol» - размер блока итерационной схемы, «Niter» - число итераций блочного предобусловленного алгоритма SOFGMRES для достижения заданной сходимости, «Nmvm» - полное число умножений матрицы на вектор в терминах векторных операций.

Таблица 1. Количество блочных итераций и полное число умножений матрицы на вектор при увеличении размера блока в блочной итерационной схеме SOFGMRES.

Ncol	1	2	3	4	5	6	7	8
Niter	535	356	277	246	204	193	177	165
Nmvm	535	712	831	984	1020	1158	1239	1320

Из Таблицы 1 видно, что полное число матрично-векторных операций растет медленно с ростом размера блока, в то время как число блочных итераций убывает достаточно быстро с ростом размера блока. Это означает, что имеется потенциальная возможность перекрыть рост арифметических затрат в блочном итерационном алгоритме увеличением производительности вычислений при реализации на гибридной архитектуре за счет использования операций типа BLAS3.

Двухуровневая MPI+TBB и трехуровневая MPI+TBB+CUDA организация параллельных вычислений. Большинство современных суперкомпьютерных вычислительных систем как правило имеют неоднородную архитектуру. С одной стороны, имеется набор вычислительных узлов с распределенной памятью, обмен данными между которыми может быть осуществлен по быстрой обменной сети. С другой стороны, каждый узел представляет собой многопроцессорный/многоядерный компьютер с общим доступом к оперативной памяти. Программная реализация вычислительных алгоритмов (включая алгоритмы решения СЛАУ) на компьютерах подобной архитектуры предполагает использование стандарта MPI при распараллеливании по распределенной памяти (по узлам вычислений), а по общей памяти узла распараллеливание по процессорам/ядрам естественно осуществлять на основе стандартов работы с потоками с встроенными механизмами динамической балансировки нагрузки, таких как OpenMP или Intel® Threading Building Blocks (TBB). При этом предполагается (Рисунок 1), что на каждом узле имеется только один MPI процесс, который порождает на этом узле нужное количество одновременно работающих потоков вычислений. Для таких вычислительных систем в работах [2-4] предложен параллельный итерационный алгоритм решения СЛАУ на основе неполного треугольного разложения второго порядка точности из работы [5].

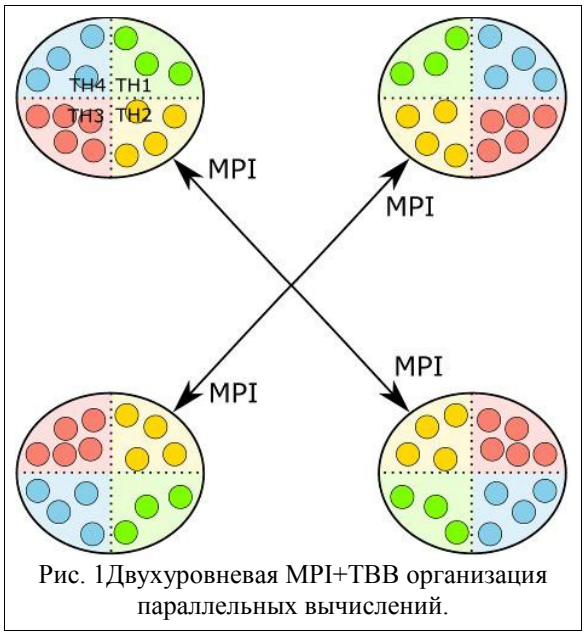

Рис. 1Двухуровневая MPI+TBB организация параллельных вычислений.

Предположим теперь, что кроме центрального процессора с набором нитей ЦПУ в каждом узле вычислительной системы имеется также один или несколько ГПУ ускорителей. Рассмотрим возможный вариант реализации на такой трехуровневой вычислительной системе MPI+TBB+CUDA итерационного алгоритма решения СЛАУ.

Прежде всего заметим, что при проведении вычислений на ГПУ необходим некоторый способ синхронизации вычислений между блоками нитей. В самом ГПУ такой механизм синхронизации отсутствует. В связи с этим предлагается использовать механизм синхронизации нитей ЦПУ для синхронизации блоков нитей ГПУ.

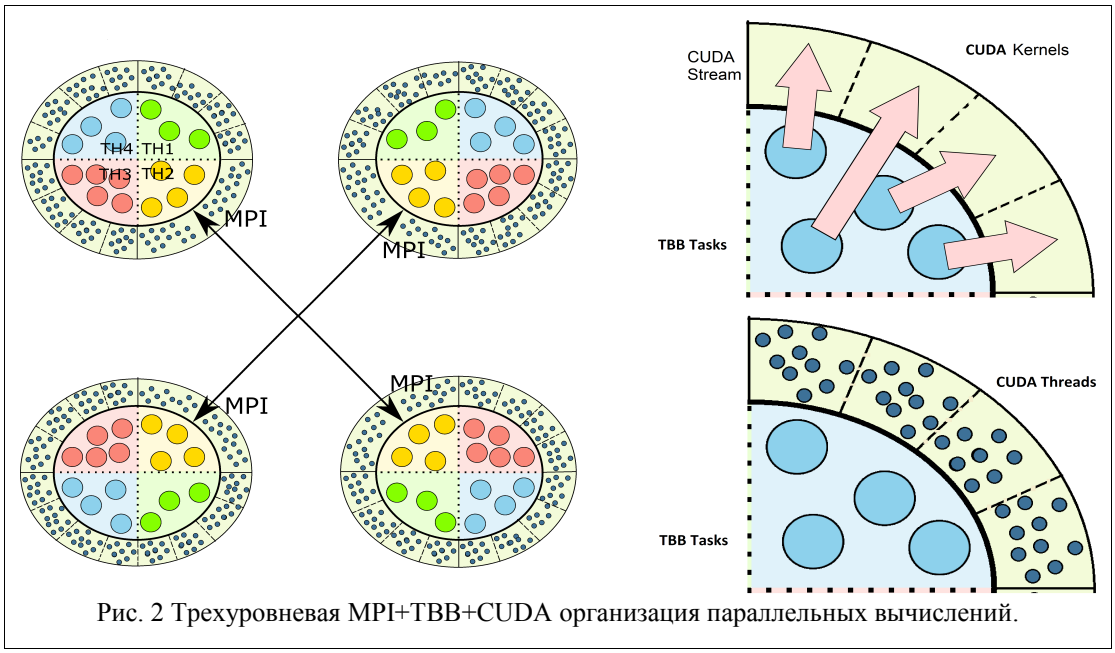

Рис. 2 Трехуровневая MPI+TBB+CUDA организация параллельных вычислений.

С каждым ГПУ ассоциируется набор потоков вычислений, их количество предполагается близким к числу мультипроцессоров в ГПУ. На ЦПУ запускается число нитей, равное этому числу, при этом осуществляется жесткая привязка — поток ГПУ — нить ЦПУ. С каждым потоком ГПУ ассоциируется большое число нитей ГПУ, соответствующее количеству нитей в мультипроцессоре ГПУ. Итерационный алгоритм решения СЛАУ распараллеливается на как будто бы СЛАУ решается в системе двухуровневой вычислительной системе с числом нитей ЦПУ, равным числу потоков вычислений на ГПУ. Синхронизации между нитями ЦПУ осуществляются как и ранее с использованием технологии Intel TBB. При этом, вычисления вместо нити ЦПУ выполняются потоком ГПУ. Схематически подобная трехуровневая MPI+TBB+CUDA организация параллельных вычислений изображена на Рисунке 2.

Для обеспечения эффективности подобной организация параллельных вычислений с участием ГПУ ускорителей необходимо обеспечить эффективность выполнения основных арифметических операций, ранее выполняемых одной нитью ЦПУ, потоком ГПУ на большом числе нитей ГПУ (число нитей ГПУ в одном мультипроцессоре ГПУ).

Формат хранения блока векторов. Традиционным форматом хранения блока векторов является способ «по столбцам». Такой формат хранения является естественным в контексте логики этого типа данных. С другой стороны, способ хранения «по столбцам» является очень плохим с точки зрения локальности работы с данными. Например, если нужно умножить элемент матрицы в позиции $\{i, j\}$ на набор из Ncol чисел в строке j для блока векторов Y, то придется из оперативной памяти подкачивать элементы, расположенные в памяти друг от друга на большом расстоянии, что не есть хорошо даже для центрального процессора, а для ГПУ не позволяет использовать быстрое совмещенное (coalesced) предчтение нитями данных из памяти ГПУ в разделяемую память линейными кусками фиксированной длины. Если же данные блока векторов хранятся «по строкам», то ситуация кардинально меняется, в этом случае для того же умножения необходимо использовать подряд лежащие Ncol чисел из блока векторов Y. При реализации блочного алгоритма все блоки векторов хранятся в формате по строкам, а все операции с матрицей, предобусловливанием и преобразованием блоков векторов (включая блочные ортогонализации) для векторных данных реализованы именно для этого локального формата хранения.

Реализация основных операций алгоритма решения СЛАУ на ГПУ.

Блочная операция AXPY. Блочная операция AXPY представляет собой прямое обобщение точечной операции AXPY (Level 1 BLAS из пакета Lapack) $y += a * x$, где a вместо скаляра в точечной версии представляет собой квадратную матрицу размера $m*m$, а блоки x и y имеют размерность $n*m$, где n – число неизвестных решаемой задачи, m – количество векторов в блоке (например, m=4,8). Существенным здесь является соотношение $n >> m$, которое, в отличие от общего случая, реализованного в операции GEMM (Level 3 BLAS из пакета Lapack), позволяет наиболее эффективным образом использовать различные механизмы распараллеливания по «младшей» и «старшей» размерности.

За счет использования разделяемой памяти (объявление __shared__) за одно считывание данных из глобальной памяти одновременно всеми нитями блока, после выполнения операции синхронизации (функция __syncthreads), эти данные становятся доступны для всех нитей блока и поэтому появляется возможность выполнить рассматриваемую операцию сразу для m слагаемых (с применением приема разворачивания цикла unrolling, и с использованием уже заранее считанных m элементов матрицы a). Таким образом наиболее узкое место этой операции, а именно, считывание данных, при такой реализации выполняется согласованно (coalesced).

Блочная операция DOT. Блочная операция DOT представляет собой прямое обобщение точечной операции DOT (Level 1 BLAS из пакета Lapack): $z = x * y$, где снова «z» вместо скаляра в точечной версии представляет собой квадратную матрицу размера $m*m$.

За счет размещения рабочих массивов в разделяемой памяти, за два считывания данных в эти массивы одновременно всеми нитями блока, после выполнения операции синхронизации, эти данные также становятся доступны для всех нитей блока и поэтому появляется возможность выполнить операцию подсчета частичной суммы сразу для m слагаемых (с применением приема разворачивания цикла unrolling). Далее $m*m$ первых нитей потока подсчитывают окончательный результат операции.

Таким образом, считывание данных при такой реализации также выполняется согласованно.

Для отдельного тестирования эффективности реализации операций AXPY и DOT в качестве модельной задачи была рассмотрена задача 64x64x64x4 x m, где m=4,8, для операций в одинарной (S) и двойной (D) точности. В Таблице 2 приведены результаты экспериментов на ГПУ для двух рассмотренных операций (AXPY и DOT). Рассматривались три реализации этих операций. Первая, через Level 1 BLAS точечную операцию AXPY с помощью соответствующего количества вызовов функции cublasSaxpy (соответственно, cublasDaxpy). Вторая, через Level 3 BLAS операцию GEMM с вызовом функции cublasSgemm (соответственно, cublasDgemm). В третьей строке приведены рассмотренные выше реализации функций Baxpy и Bdot.

Таблица 2. Увеличение производительности вычислений на ГПУ (Tesla C2070) по отношению к производительности одного ядра ЦПУ (Intel® Core i7 CPU 950 @ 3.07GHz) на задаче размерности n x m, где n=64x64x64x4, для m=4,8 правых частей при использовании одинарной (S) и двойной (D) точности.

		S(4)	S(8)	D(4)	D(8)
AXPY	cuBlas L1	6.9	5.8	3.7	3.1
	cuGEMM	2.7	7.6	1.9	5.2
	Baxpy	18.1	31.5	12.8	21.3
DOT	cuBlas L1	3.7	3.6	3.0	2.7
	cuGEMM	0.1	0.2	0.1	0.2

	Bdot	10.4	19.8	11.5	18.2

Результаты экспериментов показали достаточно высокую эффективность функций из Level 1 BLAS. Относительно низкая производительность блочной операции GEMM обусловлена тем, что эта функция реализует операцию умножения матриц произвольных размерностей, которая эффективно работает для матриц, у которых все 3 размерности достаточно большие, и это оказывается недостаточно эффективным для нашего конкретного случая блока векторов с небольшим количеством правых частей. Наоборот, в нашей реализации (функции Baxpy и Bdot) удалось добиться заметного преимущества за счет использования именно этого конкретного свойства рассматриваемых данных.

Алгоритм умножения разреженной матрицы на блок векторов (MVM). При реализации алгоритма умножения разреженной матрицы на вектор (MVM) общее количество нитей в потоке разбивалось на группы по $m*m$ нитей в каждой группе.

Для использования достаточно большого ресурса параллелизма, присущего этой операции, мы разбили все строки матрицы на набор из нескольких блочных строк (каждая соответствующая своей группе нитей), причем общее количество ненулевых мелких блоков для каждого потока выбиралось примерно одинаковым для получения наиболее сбалансированного блочного распределения (см. Рис. 3).

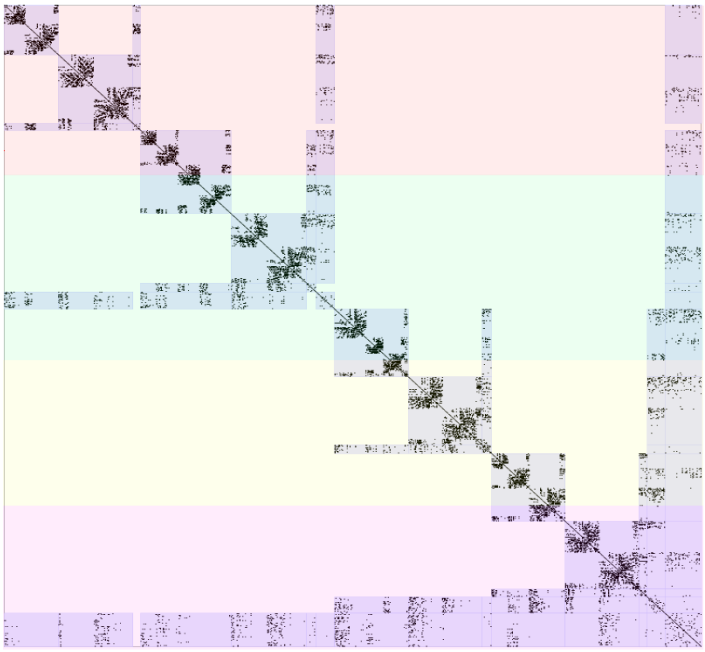

Рис. 3 Разбиение по блочным строкам при умножении матрицы на блок векторов.

Умножение на транспонированную матрицу MVM(T) реализовано аналогично, за единственным исключением использования косвенной адресации при нахождении адреса блока матрицы.

В качестве модельной задачи для проверки эффективности алгоритма умножения матрицы на блок векторов рассмотрим задачу аппроксимации некоторого разностного 7-точечного оператора на сетке 64x64x64 для 4 и 8 неизвестных функций на узел (т.е. размер мелкого блока m=4 и m=8, соответственно). Общее количество блоков при этом составляет n=262144, а количество ненулевых мелких блоков nzja=1810432. При этом общее количество неизвестных n4=1048576 и n8=2097152, а общее количество ненулевых элементов в матрице nza4=28966912 и nza8=115867648 для размеров мелкого блока m=4 и m=8, соответственно. Приведем результаты только для использования двойной точности (double) при хранении как коэффициентов матрицы, так и вектора неизвестных.

Таблица 3. Увеличение производительности вычислений на ГПУ для умножения на прямую MVM(N) и транспонированную MVM(T) матрицы по отношению к производительности одного ядра ЦПУ на задаче размерности n, где n=64x64x64xm, для m=4,8, где m – размер мелкого блока и количество правых частей при использовании двойной точности для хранения как матрицы так и правых частей.

	m=4	m=8
MVM(N)	6.4	31.2
MVM(T)	10.4	35.2

Из приведенной Таблицы 3 видно, что полученная производительность ГПУ более чем в 30 раз превышает производительность одного процессора ЦПУ для размера мелкого блока m=8 и числа правых частей 8. Сравнение результатов экспериментов для m=4 и m=8 показывает, что дополнительные преимущества применения большого размера мелкого блока (m=8) удается успешно использовать в текущей реализации операции MVM.

Несколько большая относительная производительность операции умножения на транспонированную матрицу по сравнению с умножением на прямую матрицу объясняется не большей абсолютной производительностью этой операции, а несколько большим замедлением работы ЦПУ по сравнению с ГПУ при использовании дополнительной косвенной адресации и нерегулярным обращением к коэффициентам матрицы.

Следует заметить, что применение библиотеки cuSparse (функция cusparseDcsrmm) показывает существенно меньшую производительность, примерно в 2 раза медленнее для m=4 и, в 8 раз медленнее для m=8.

Реализация базовых операций типа SOL. При реализации операции типа SOL (решение с нижне- и с верхнетреугольной матрицей L и U, соответственно) общее количество нитей в потоке разбивалось на группы по m*m нитей в каждой группе. При этом для диагонального блока требуется сложная организация вычислений ввиду зависимостей по данным и более сложной синхронизации нитей для одного потока.

Сначала рассмотрим основные зависимости по данным и последовательность действий для диагонального блока, предварительно упорядоченного по алгоритму вложенных сечений (ND, Nested Dissection). В этом случае определенный ресурс параллелизма имеется уже при использования естественного параллелизма вычислений по уровням бинарного дерева при распределении данных по блочным столбцам для L и блочным строкам для U, соответственно. Можно рассмотреть также более сложную комбинированную организацию параллельных вычислений (см. Рис. 4), которая позволяет также распараллелить вычисления с окаймлениями. При такой схеме организации вычислений непараллельный этап (заключительный этап 4 для матрицы L и начальный этап номер 0 для матрицы U) всего один, на всех остальных этапах группы нитей работают параллельно. Можно также видеть, что на этапе 1 для L и этапе 3 для U происходит «зацепление» вычислений при переходе от блочно-«плотного» типа разреженности к блочно-разреженному. На начальном 0-ом этапе для L последнем 4-ом этапе для U видно, что структура разреженности позволяет использовать гораздо большее количество групп процессоров, например 8, т.к. дополнительный ресурс параллелизма за счет разреженности типа ND на этих этапах оказался не востребованным.

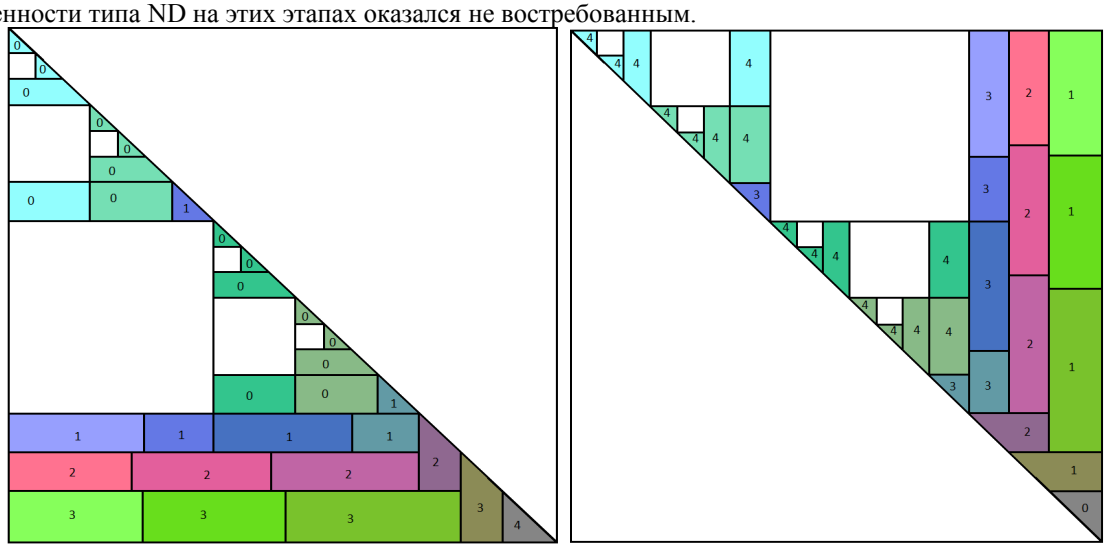

Рис. 4 Комбинированная схема зависимости по данным для решения систем с нижней треугольной и верхней треугольной матрицами, соответствующих разреженным неструктурированным диагональным блокам L и U, соответственно. Цифрами обозначена последовательность действий, причем операции над блоками, помеченными одинаковыми цифрами, могут параллельно и независимо выполняться различными группами нитей в потоке нитей ГПУ.

Приведем теперь результаты для использования одинарной точности (float) при хранении коэффициентов треугольного разложения и двойной точности (double) для хранения вектора неизвестных.

Таблица 4. Увеличение производительности вычислений на ГПУ для решения треугольных систем Sol_L и Sol_U по отношению к производительности оптимизированной версии для одного ядра ЦПУ при m=4,8.

	m=4	m=8
Sol_L	3.0	13.9
Sol_U	5.7	22.9

Заключение. Таким образом, в работе представлен трехуровневой MPI+TBB+CUDA параллельный блочный итерационный алгоритм решения СЛАУ для мелкоблочных матриц. Результаты численных экспериментов подтверждают эффективность предложенного алгоритма для набора тестовых задач.

Авторы выражают благодарность рецензентам, замечания которых помогли значительно улучшить изложение.

ЛИТЕРАТУРА:

1. С.А. Харченко, "Параллельная реализация итерационного алгоритма решения несимметричных систем линейных уравнений с частичным сохранением спектральной/сингулярной информации при явных рестартах" // "Вычислительные методы и программирование", 2010, т. 11, 373-381.
2. С.А. Харченко, "Влияние распараллеливания вычислений с поверхностными межпроцессорными границами на масштабируемость параллельного итерационного алгоритма решения систем линейных уравнений на примере уравнений вычислительной гидродинамики". Материалы международной научной конференции "Параллельные вычислительные технологии" (ПаВТ'2008), Санкт-Петербург, 28 января – 1 февраля 2008 г. Челябинск, Изд. ЮУрГУ, 2008, с. 494-499.
3. Г.Б. Сушко, С.А. Харченко, "Многопоточная параллельная реализация итерационного алгоритма решения систем линейных уравнений с динамическим распределением нагрузки по нитям вычислений". Труды международной научной конференции Параллельные вычислительные технологии (ПаВТ'2008), Санкт-Петербург, 28 января – 1 февраля 2008 г. Челябинск, Изд. ЮУрГУ, 2008, с. 452-457.
4. Г.Б. Сушко, С.А. Харченко, "Экспериментальное исследование на СКИФ МГУ “Чебышев” комбинированной MPI+threads реализации алгоритма решения систем линейных уравнений, возникающих во FlowVision при моделировании задач вычислительной гидродинамики". Материалы международной научной конференции "Параллельные вычислительные технологии" (ПаВТ'2009), Нижний Новгород, 30 марта – 3 апреля 2009 г., Челябинск: Изд. ЮУрГУ, 2009, с. 316-324.
5. I.E. Kaporin, High Quality Preconditioning of a General Symmetric Positive Definite Matrix Based on its $U^T U + U^T R + R^T U$ decomposition // Numer. Linear Algebra Appl. – 1998. – Vol. 5. – P. 483-509.
6. A. Greenbaum, Iterative Methods for Solving Linear Systems, SIAM, 1997.