Разработка базовых GPU ядер для новых блочных AMG алгоритмов для решения СЛАУ с явно вычисляемым базисом

И.В. Афанасьев, Ю.Ю. Потапов

Московский государственный университет имени М.В. Ломоносова

Основная цель данной работы – разработка базовых вычислительных ядер, позволяющих перенести часть вычислительных функций программного комплекса Flow Vision [1] с CPU на GPU. Для этого необходимо было реализовать некоторый набор базовых операций линейной алгебры плотных и разреженных матриц, с учетом специфики данных, используемых в пакете FlowVision. Основной причиной невозможности использования уже готовых библиотечных функций от компании NVidia [2][3] было то, что они не предоставляют достаточно эффективных алгоритмов параллельного вычисления нескольких задач небольшой размерности на одной видеокарте. В результате данной работы было получено ускорение от 2 до 20 раз по сравнению с уже упомянутыми библиотечными аналогами.

1. Введение

В настоящее время в суперкомпьютерных вычислениях наряду с вычислениями на центральных процессорах (CPU) важную роль для ускорения вычислений играет использование графических процессоров (GPU). Центральное место среди GPU занимают графические ускорители компании Nvidia с поддержкой технологии программирования CUDA. Это демонстрируется, например, тем, что данные графические процессоры входят в состав многих современных суперкомпьютеров из списков top500, green500, а так же top50, и вносят значительный вклад в производительность каждой из этих вычислительных машин. По этой причине вычислительно затратные программные комплексы, для работы которых необходима максимально доступная мощность суперкомпьютера, должны уметь производить вычисления не только на центральных процессорах, но и на графических.

Данная работа велась совместно с российской компанией «Тесис», разработавшей программный пакет FlowVision. FlowVision — программный комплекс, используемый для моделирования и расчета трехмерных сложных движений жидкости и газа, сопровождаемых дополнительными физическими явлениями, такими, как турбулентность, горение, контактные границы раздела, пористость среды, теплоперенос и.т.д. На данный момент программный пакет FlowVision поддерживает вычисления на кластерах из многоядерных центральных процессоров. Основная цель данной работы – перенести часть вычислительных функций пакета с центрального процессора на графические, без переписывания большого количества кода.

Для переноса части вычислений пакета FlowVision на GPU необходимо реализовать некоторый базовый набор операций линейной алгебры плотных и разреженных матриц. Далее приведен список данных операций:

1. Операция DAXPY - умножение блока из плотных векторов на мелкий блок, блочное обобщение операции первого уровня AXPY из BLAS.

$Y + X * A$ (1.1)

Здесь X, Y - блоки из плотных векторов, а A – мелкий блок одного из фиксированных размеров.

2. Операция DOT

Скалярное произведение двух блоков из плотных векторов.

$Y + X * Z$ (1.2)

X, Z – блоки из плотных векторов, Y – результат их скалярного произведения в виде мелкого блока фиксированного размера.

3. Умножение блочной разреженной матрицы и транспонированной к ней на блок из плотных векторов
   Операция умножения блочной разреженной матрицы (или транспонированной к ней) на блок из плотных векторов представляет собой одну из двух следующих операций, то есть обновления блока векторов Y со знаком «+» или «-».
   $Y + A \times X$ или $Y - A \times X$ (1.3)
   A – разреженная мелко-блочная матрица. Также умножение может производиться с учетом диагональных блоков матрицы или без.

4. (Не) полное треугольное разложение блочной разреженной матрицы
   Операция неполного треугольного разложения блочной разреженной матрицы представляет собой нахождение матриц, удовлетворяющих следующему отношению:
   $A \approx LU$ или $A = LU$ (1.4)
   где A, L, U – блочные разреженные матрицы с совпадающими профилями разреженности.

5. Решение СЛАУ
   Решение системы линейных уравнений с блочной разреженной верхней или нижней треугольной матрицей.
   $AX = B$(1.5)
   Где A — блочная разреженная матрица, имеющая треугольную форму, B, X — блоки плотных векторов.

Все приведенные операции обладают рядом особенностей. Размеры блока матрицы и блока векторов принимают одно из дискретного набора значений: $\{4, 8, 16\}$. Они могут принимать как совпадающие, так и различные значения, однако первый случай наиболее интересен (например, операции DOT и DAXPY будут использоваться в операции QR разложения только с совпадающими размерами). Тип данных с плавающей точкой может быть как float, так и double, типы данных аргументов одной операции всегда совпадает.

Для эффективной реализации приведенных выше операций важно учитывать специфику хранения данных в пакете FlowVision. Во-первых, существуют приведенные выше ограничения на размер мелкого блока матрицы, блока из векторов, и.т.д. Благодаря этому можно проводить оптимизации, связанные с фиксированным размером блока (реализация отдельных ядер для каждого фиксированного размера). Во-вторых, все матрицы и вектора, используемые в вычислениях пакета FlowVision, обычно имеют небольшую размерность, вследствие чего невыгодно использовать всю видеокарту для вычисления одной операции. Решение этой проблемы как раз составляет главную идею представленной работы и описано в следующем разделе.

2. Общие результаты и исследования

Из-за особенностей организации вычислений пакета FlowVision (все вектора и матрицы имеют небольшие размеры) появляется проблема эффективного использования видеокарты, так как последовательно запускать множество задач небольшой размерности, занимающих всю видеокарту, как это делается при классическом использовании видеокарты, крайне невыгодно.
В этом разделе описан основной подход, применявшийся для решения данной проблемы, а именно: выполнение каждой операции лишь на части ядер видеокарты, а не на всех. В результате этого подхода получена возможность эффективно запускать большое число параллельно работающих независимых операций на одной видеокарте, что и требуется от данных реализаций. Кроме того, при данном подходе появляется возможность эффективно использовать CPU и GPU одновременно.

Для эффективного запуска множества параллельных задач небольшой размерности на одной видеокарте необходима архитектура Kepler и выше, так как данные ускорители поддерживают технологию Hyper-Q. Данная технология необходима для того, чтобы можно было независимо запускать до 32 конкурирующих ядер в различных CUDA-потоках, что не было воз-

можным в более ранних архитектурах. Каждое ядро представляет собой одну из 6 приведенных выше операций. Запущенное ядро использует GPU следующем образом: оно запускается в конфигурации 1 блок, 1024 нити. Благодаря этому, каждая операция вычисляется в одном из потоков непрерывно, после чего за ней может последовать другая операция (возможно, также другого типа). Далее последует объяснение, как данные ядра размещаются на мультипроцессорах видеокарты.

Благодаря тому, что каждая из операций занимает 1024 нити, на каждом мультипроцессоре могут одновременно выполняться одна или две операции (для архитектуры Kepler доступно максимально 2048 нити на мультипроцессор, причем максимальное число нитей в блоке - 1024). На современных вычислительных GPU, используемых в суперкомпьютерных вычислениях, устанавливаются от 13 до 16 мультипроцессоров. Таким образом, при запуске двух задач на мультипроцессор на одной видеокарте можно запускать от 26 до 32 (в зависимости от модели видеокарты) параллельно работающих ядер. Благодаря технологии Hyper-Q, все запущенные таким образом ядра работают параллельно.

Исследование показало, что эффективнее размещать на каждом мультипроцессоре по две операции, так как загрузка вычислительных ядер мультипроцессора при таком подходе выше, что частично компенсирует потери из-за латентности доступа к памяти. Это подтверждается следующими диаграммами, демонстрирующими графики зависимости времени выполнения операций от числа параллельно запущенных на видеокарте задач (запускаются однотипные операции).

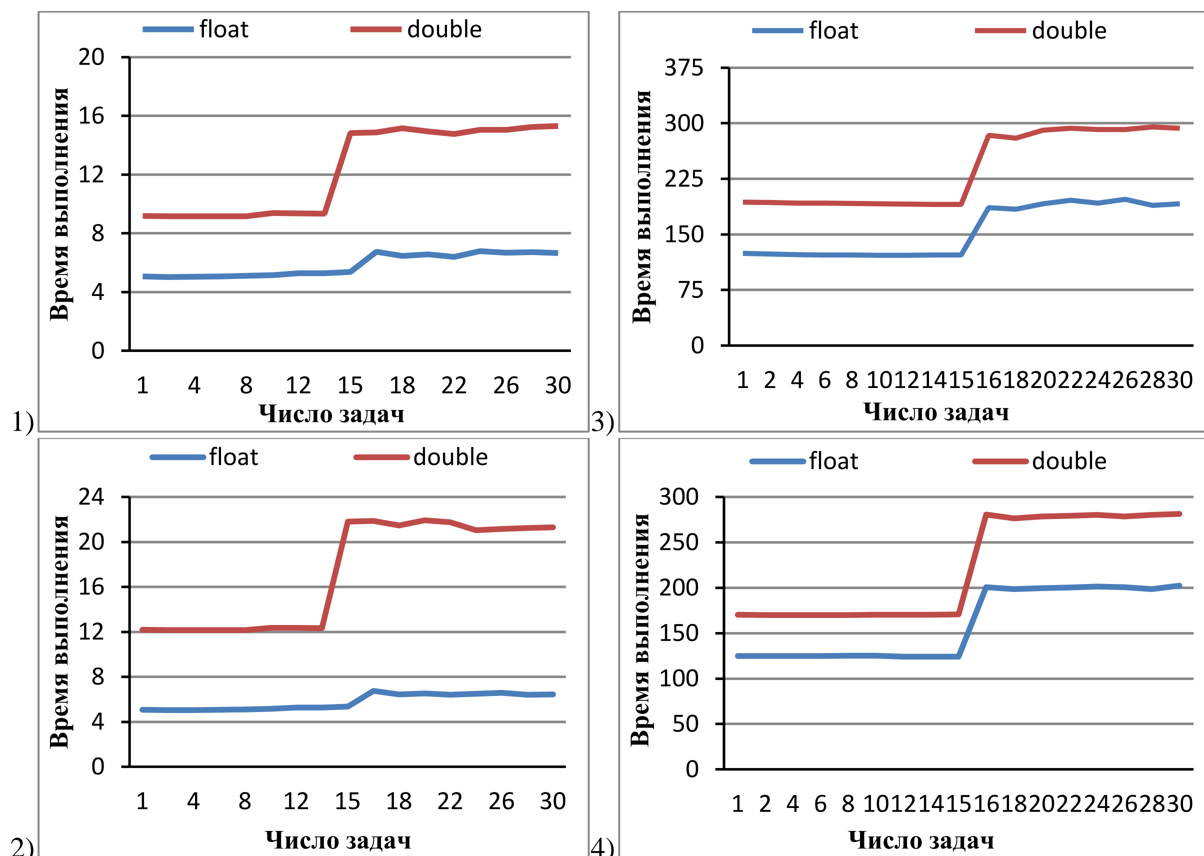

Рис. 2.1. Зависимость времени выполнения(ms) от числа задач при фиксированном размере задачи для операций 1) DAXPY, 2) DOT, 3) и 4) матрично-векторного умножения.

Данные графики приведены для видеокарты K40, оборудованной 15 мультипроцессорами. Таким образом, при описанном подходе на ней можно запустить от 1 до 30 задач, причем при запуске 16-ой задачи время выполнения увеличивается меньше, чем в два раза. Очевидно, это происходит из-за того, что на одном мультипроцессоре запускается уже не одна, а две задачи.

Кроме того, данный подход решает проблему одновременного использования CPU и GPU. Действительно, вычисления можно производить по следующей схеме: запускается некоторое количество CPU потоков (например, с помощью Intel TBB), некоторые из которых будут прикрепляться к ядрам CPU и проводить вычисления на них, в то время как другие будут заниматься управлением вычислениями на GPU (в основном запуском ядер и копированием данных). Эту схему демонстрирует следующая иллюстрация:

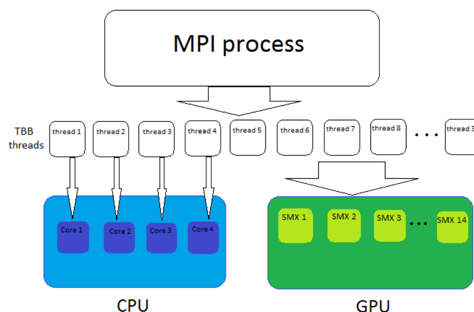

Рис. 2.2. Взаимодействие CPU и GPU задач через Intel TBB

Таким образом, например, для 4-ядерного центрального процессора и видеокарты K40 с 15 мультипроцессорами можно запустить 3 + 30 = 33 CPU потока. Три из них будут вычисляться на ядрах центрального процессора (одно оставшееся свободное ядро будет выполнять управления видеокартой), остальные 30 на видеокарте.

3. Результаты по операциям

3.1. Операция DAXPY

Операция DAXPY, как уже говорилось, представляет собой прямое обобщение точечной операции AXPY.

$$y += x * A \quad (3.1)$$

Где A – прямоугольный блок размера N_1 x N_2, N_1, N_1 $\in \{4, 8, 16\}$, а векторы x и y размера M x N_1 и M x N_2, где M >>N_1, N_2.

Операция реализована несколькими CUDA-ядрами для различных размеров мелкого блока. При совпадающих размерах получена наибольшая производительности из-за наиболее эффективного использования регистров и разделяемой памяти, а так же высокой занятости ядер графического процессора. Так же следует заметить, что наиболее интересны с практической точки зрения как раз случаи с квадратным мелким блоком размера 4x4, 8x8, 16x16.

Само ядро устроено следующим образом: для вычислений всегда используются 1024 нити, таким образом, при числе мультипроцессоров N можно параллельно запустить 2 * N таких ядер, как и описывалось в предыдущем разделе. Наиболее производительные ядра используют следующую стратегию вычислений: мелкий блок целиком помещается на регистры, а блок из векторов выгружается нитями из глобальной памяти с помощью двойного кэширования: сначала данные помещаются в разделяемую память, а затем на регистры. Умножение происходи только с использованием регистровых переменных. Так же, где возможно, используются операции __shfl() из архитектуры вычислений 3.0 для того, чтобы обойти разделяемую память и хранить данные только на регистрах.

Реализованные ядра можно сравнивать с библиотечной функцией DGEMM из cuBLAS, которая выполняет ту же операцию при правильном наборе параметров. Результаты сравнения их производительностей представлены на следующих диаграммах:

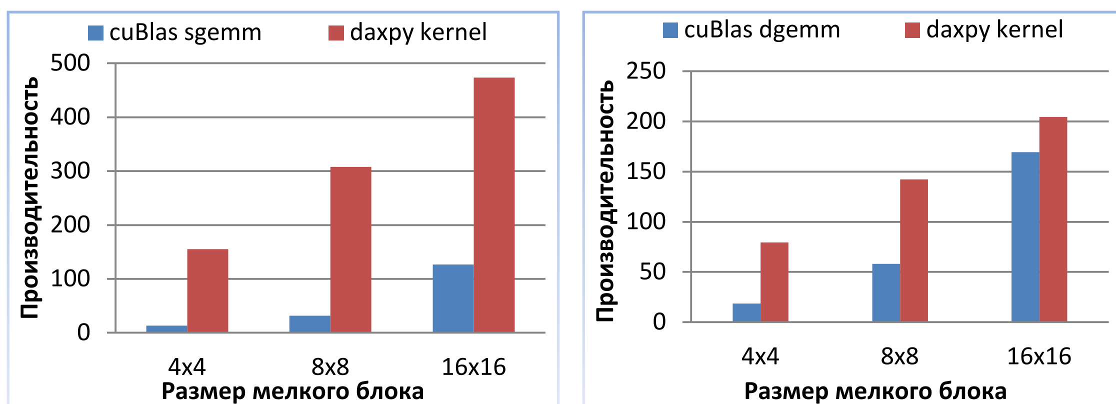

Рис. 3.1. Сравнение производительности(GFlops) cuBlas и ядра DAXPY. 30 задач, 10^6 элементов в зависимости от размера блока, точность **float** и **double**.

Из приведенных выше диаграмм видно, что реализованные ядра DAXPY обгоняют библиотечные реализации от 4 до 20 раз. Достигается, это, в основном за счет параллельного использования видеокарты для вычислений, а так же оптимизаций, связанных с фиксированным размером блока.

3.2. Операция DOT

Реализует скалярное произведение блоков векторов.

$$C = A^T * B \quad (3.2)$$

где A и B — блоки векторов, то есть матрицы размера MxN, где M > > N, а N — размер блока.

Результат C — блок размера NxN, такой, что C[i][j] — скалярное произведение i-го столбца матрицы A на j-й столбец матрицы B. В рамках исследования ядро запускается только на одном мультипроцессоре, занимая 1024 виртуальные нити, принадлежащие одному блоку. Так же, как и в операции DAXPY, используется двойное кэширование, сначала на быструю разделяемую память, потом на регистры. Все вычисления происходят на регистрах. Сравнение полученных результатов с операцией DGEMM из cuBLAS:

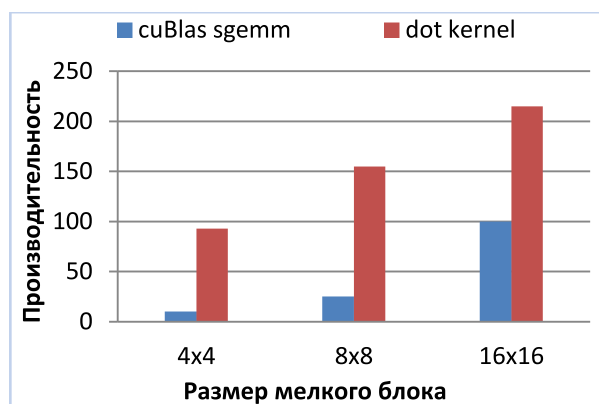

Рис. 4.1. Сравнение производительности(GFlops) cuBlas и ядра DOT. 30 задач, 10^6 элементов в зависимости от размера блока, точность **float**.

3.3. Блочное умножение на мелко блочную разреженную матрицу и транспонированную к ней

Умножение блочной разреженной матрицы на блок из плотных векторов (далее сокращенно MVM от англ. Matrix Vector Multiplication) представляет из себя одну из следующих двух операций, то есть обновление вектора Y со знаком «+» или «-» :

$$Y += A \times X \text{ или } Y -= A \times X \quad (3.3)$$

где матрица A – разреженная матрица в блочном CSR формате размера $M \times M$, X и Y – плотные вектора размера $M \times N_1$, а «$\times$ » - операция умножения с учетом или без учёта блоков на диагонали матрицы A.
Размер мелкого блока разреженной матрицы принимает одно из следующих фиксированных значений: $\{4 \times 4, 8 \times 8, 16 \times 16\}$. Размер блока из векторов также принимает одно из фиксированных значений: {4, 8, 16}. Наиболее интересны совпадающие размеры мелкого блока матрицы и блока из векторов.

С программной точки зрения операция умножения разреженной матрицы на вектор состоит из двух частей: производимой на центральном процессоре и на графическом ускорителе. Перед, непосредственно, умножением, на центральном процессоре происходит переупорядочивание строк матрицы таким образом, чтобы подряд шли строки с одинаковым числом блоков в строке. В результате этого становится удобно вызывать CUDA функцию умножения для группы строк с одинаковым числом блоков, что сильно повышает производительность.
Надо заметить, что, во-первых, перемещения данных при перестановке не происходит – вместо этого лишь переставляются индексы, для перестановки которых просто заводится еще один дополнительный массив. Во-вторых, такая перестановка нужна для матрицы только один раз перед ее записью на GPU, и далее с ней можно удобно и эффективно работать.
Работа, производимая центральным процессором, состоит из следующих действий:

1) Сначала к матрице добавляются две дополнительные структуры данных: массив структур, описывающий группы строк с одинаковыми длинами, и ссылки на новые индексы строк после перестановки.
2) Затем происходит перестановка индексов строк сортировкой длин с помощью стандартной функции quick sort на центральном процессоре. Затем, при необходимости вычислений на графическом ускорителе, все данные матрицы вместе с нововведенными структурами данных переносятся в память видеокарты.
3) После этого вычисления можно наиболее эффективно производить на графическом ускорителе.

Часть, выполняемая графическим процессором, состоит из следующих действий:

1) На графическом ускорителе запускается цикл по числу групп строк различной длины.
2) Для каждой группы строк вызывается __device__ функция, выполняющие умножение этих строк на вектор X. Для каждой строки выделяется группа нитей, с числом равным квадрату размера блока, однако, как уже говорилось – не более 1024 нитей на группу строк.
3) Блоки из каждой строки, умножаемые на соответствующие блоки из группы векторов, последовательно загружаются в разделяемую память и перемножаются. Благодаря удачно выбранному порядку циклов в умножении удалось использовать 100% пропускной способности разделяемой памяти, поэтому на регистрах хранится только промежуточная сумма. В конце обработки строки эта сумма сохраняется в глобальную память и группа нитей начинает обрабатывать следующую строку.

Для вычисления результата операции MVMT (умножения транспонированной разреженной матрицы на блок векторов) выполняются те же действия, но матрица предварительно переводится в формат CSC и вместо описания строк хранит описание столбцов матрицы. Это позволяет использовать то же ядро с минимальными модификациями.
Благодаря такому подходу удалось эффективно реализовать операцию MVM на GPU даже для небольших размеров входных матриц (а так же для сильно разреженных матриц). Далее приведено сравнение реализованных ядер с библиотечным аналогом – функцией cusparseSbsrmv из библиотеки cuSparse. Результаты сравнения их производительностей представлены на следующих диаграммах:

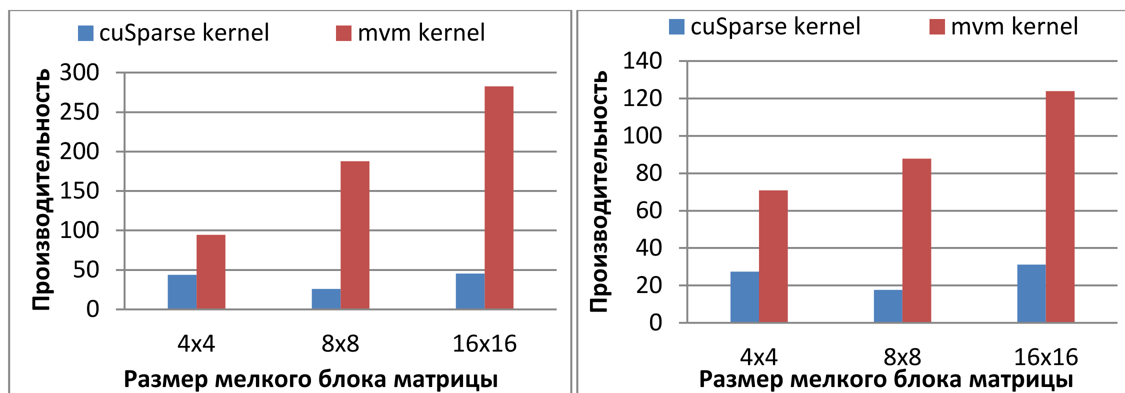

Рис. 3.3. Сравнение производительности(GFlops) cuBlas и ядра MVM. 30 задач, точность **float** и **double** соответственно.

Из приведенных выше диаграмм видно, что реализованные ядра MVM обгоняют библиотечные реализации от 2 до 6 раз.

3.4. Неполное треугольное разложение блочной разреженной матрицы

В данной операции для блочной разреженной матрицы A необходимо найти две такие матрицы L и U, что

$$A \approx LU \text{ или } A = LU \quad (3.4)$$

в случае неполного и полного треугольного разложения соответственно. Матрицы A, L, U устроены так же, как в операции матрично-векторного умножения.
Операция треугольного разложения, как на графическом, так и на центральном процессоре состоит из трех частей:

1) Подготовка данных о разреженности матрицы, их дополнение для дальнейшей обработки
2) Копирование данных (разреженности и блоков) на видеокарту (если хотим произвести вычисления на GPU, а не на CPU)
3) Запуск вычислительного ядра (или вычислительной функции на центральном процессоре)

Из приведенного списка видно, что первая часть для CPU и GPU одинаковая и уже была реализована разработчиками из компании «Тесис», поэтому в данной работе рассматриваться не будет. Так же важно заметить, что с вычислительной точки зрения она гораздо менее затратная, чем сами вычисления, и, кроме того, в данной части много ветвлений и операторов перехода и небольшие ресурсы параллелизма, поэтому выгоднее выполнять её на центральном процессоре.

Далее приведено описание схемы работы вычислительного ядра на графическом процессоре.

Вычислительное ядро запускается на 512 GPU-нитях (а не на 1024, как для предыдущих операций), так как оно использует значительно больше регистров, что ограничивает occupancy.

Для каждой строки матрицы выполняется:

1) Обновление текущей строки через предыдущие строки верхнего треугольника, а так же столбцы нижнего треугольника

   Данная функция вычисляется следующим образом: на каждую строку выделяется группа из всего множества нитей (так как нитей может быть сильно больше, чем необходимо для обработки одной строки при небольшом размере блока матрицы) – например 64 нити из 512 при обработке матрицы с блоком 4x4. Важно заметить, что эта часть вычислений в большинстве случаев наиболее затратная по времени выполнения.

2) Модификация диагональных блоков при необходимости (только в случае не полного разложения)

Производится быстро через __shfl инструкции.

3) Факторизация и обращение диагонального блока
   Здесь диагональный блок, представляющий собой плотную матрицу размера блока (NxN), сначала приводится к треугольному виду, а затем полученный треугольник обращается (в симметричном случае). В обоих случаях параллельно работать может только N нитей, в то время как остальные вычисления производить нельзя, что сильно замедляет вычисления при небольшом числе блоков в строке.
   Полученный блок передается в следующую функцию через разделяемую память, что позволяет уменьшить число обращений к более медленной глобальной памяти.

4) Умножение блоков данной строки на обращенный блок
   Умножение всех блоков строки на обращенный блок из предыдущей операции. В обоих случаях сделано так, что каждая пара умножаемых блоков хранится на регистрах.

5) Сохранение результата в нижнюю и верхнюю треугольные матрицы
   Сохранение результатов в глобальную память (блоков верхнего и нижнего треугольных факторов).

3.5. Решение СЛАУ

Данная операция представляет собой решение следующей системы:

$$A * \mathrm{X} = \mathrm{B} \ (3.5)$$

где A – мелко-блочная разреженная треугольная матрица, B – плотный блок векторов, матрицы A хранятся по столбцам, а матрицы B – по строкам.

В этой задаче требуется вычислить не решение системы линейных уравнений с блочной матрицей A, а найти решения для блока систем линейных уравнений. Матрицы этих систем сгруппированы поэлементно в блочную матрицу A, а правые части сгруппированы в блок векторов B. Так как матрица уже дана в треугольной форме, то реализуется обратный ход метода Гаусса решения системы линейных уравнений, с той лишь разницей, что вместо элемента матрицы и элемента правой части используются блоки.

1. Сначала матрица переводится из формата CSR в формат CSC для удобства и локальности обращений в память.

2. Блок нитей работает с последним, не обновлённым столбцом, при этом параллельно вычисляются новые значения правых частей. Доступ к памяти осуществляется блоками, реализуется кэширование сначала в разделяемую память, оттуда на регистры. Все вычисления проводятся на регистрах.

4. Заключение

В ходе данной работы были эффективно реализованы шесть базовых операций линейной алгебры из пакета FlowVision. Для каждого из реализованных ядер было проведено большое количество оптимизаций, исследованы различные их характеристики и свойства. Также для каждой операции была теоретически обоснована эффективность реализации. Из анализа эффективности с использованием профилировщика Visual Profiler было определено, что основным ограничением на производительность написанных ядер оказалась пропускная способность разделяемой памяти, которая используется в качестве КЭШа с быстрым доступом для хранения обрабатываемых элементов матрицы (обычно мелкого блока).

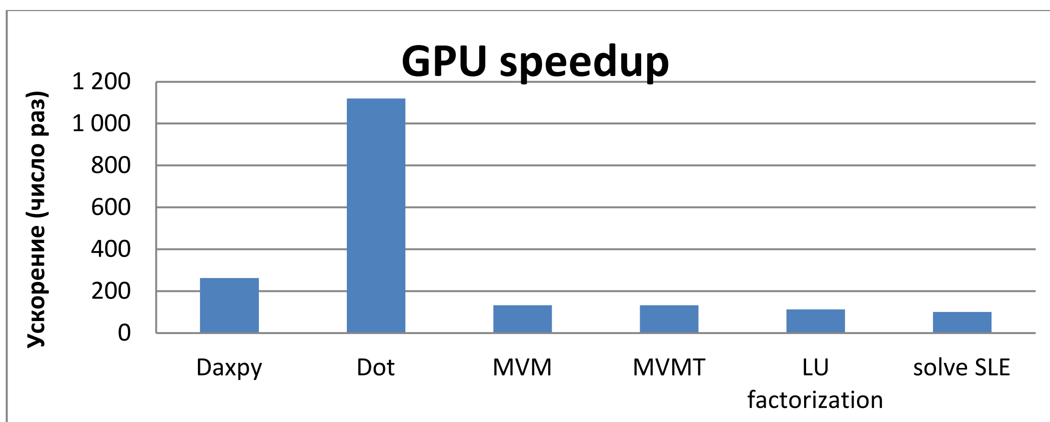

Рис. 4.1. Ускорение относительно одноядерных тестовых CPU аналогов данных операций.

Таким образом, в будущем эти ядра могут быть успешно использованы в программном пакете FlowVision компании «Тесис». Так же очень важно, что было получено ускорение в 2-20 раз относительно функций из программных пакетов cuBlas и cuSparse от компании NVidia, реализующих требуемые операции.

Работа выполнена/выполняется при финансовой поддержке Министерства образования и науки Российской Федерации, Соглашение №14.607.21.0006, уникальный идентификатор RFMEFI60714X0006.

Литература

1. Тесис, «Flow Vision,» [В Интернете]. Available: http://tesis.com.ru/software/flowvision
2. NVidia, «cuBlas,» [В Интернете]. Available: http://docs.nvidia.com/cuda/cublas
3. NVidia, «cuSparse,» [В Интернете]. Available: http://docs.nvidia.com/cuda/cusparse
4. Писсанецки С. Технология разреженных матриц = Sparse Matrix Technology. — М.: Мир, 1988