МИНОБРНАУКИ РОССИИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ»
Факультет управления и прикладной математики
Кафедра системного программирования

Исследование возможностей распараллеливания алгоритмов решения целочисленных линейных неравенств на GPU

Бакалаврский диплом

Направление подготовки 010400 «Прикладные математика и информатика»

Допущено к защите в ГЭК 20.06.2017

Зав.кафедрой	_____	д.физ.-мат.н., проф.	А.И. Аветисян
Обучающийся	_____		В.В. Швецова
Руководитель	_____	к.физ.-мат.н.	А.С. Камкин

Долгопрудный 2017

ОГЛАВЛЕНИЕ

Стр.

ГЛАВА 1

Введение

Актуальность

Данная работа является прикладным исследованием в пересечении областей смешанного целочисленного программирования (англ. mixed integer programming) и программирования в ограничениях (англ. constraint programming). Так как многие важные задачи сводятся к задачам разрешения ограничений (англ. constraint satisfaction problem), методам разрешения ограничений уделяется немало внимания [1], [2], [3].

Целочисленное и смешанное целочисленное программирование появилось в 50-60-х, когда выяснилось, что задачи смешанного целочисленного программирования часто возникают в различных практических задачах и стало актуальным их эффективное решение [4].

Впервые понятие общих ограничений, т.е не только линейных неравенств, было использовано Sutherland [5] в его проекте интерактивной системы Sketchpad, предоставляющей пользователю рабочий интерфейс для набросков или эскизов при помощи компьютерной графики. В 70-х это понятия появилось в контексте автоматизированного доказательства теорем и логического программирования, язык которого Prolog был разработанн Colmerauer [6] и Kowalski [7]. В 80-х задачи разрешения ограничений были отнесены к логическому программированию, что образовало логическое программирование в ограничения (англ. constraint logic programming) [8], [9], [10].

Задача смешанного программирования имеет вид:

$$c^* = \min\{c^T x | Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z}\ \forall j \in \mathbb{I}\}$$

$$X_{MIP} = \{x \in \mathbb{R}^n | Ax \leq b,\ x_j \in \mathbb{Z}\ \forall j \in \mathbb{I}\}$$

Решение x^* называется оптимальным, если $c^* = c^T x^*$.

Задача существования целочисленного решения системы является NP-полной. Таким образом, для реальных задач необходимо проделывать очень

большой объем вычислений, что не позволяет расширить довольно ограниченный круг задач, для решения которых используются методы целочисленного программирования. Актуальность и трудность проблематики делают целочисленное программирование одним из перспективных и интересных направлений в математическом программировании [11], [12].

Использование методов разрешения ограничений позволяет решать многие прикладные задачи, такие как планирование [13] и теория расписаний [14], поддержка принятия решений [15], обработка изображений [16], тестирование интегральных схем [17], формальная верификация и статический анализ [18], [19], [20].

Существуют методы построения тестов с помощью символического выполнения (англ. symbolic execution). Такие методы используют символическое описание проходимого во время выполнения теста пути по коду программы в виде набора предикатов. Это описание позволяет выбирать новые тестовые ситуации так, чтобы они покрывали другие пути и строить тесты с помощью техник разрешения ограничений [21], [22].

В литературе можно найти несколько хороших обзоров с описанием методов решения задач разрешения ограничений [23], [24], [25], [26], [27], а также статьи в энциклопедических сборниках [28], [29], [30].

Более подробную информацию по теме разрешения ограничений можно найти в монографиях [31], [32], [33], [34], [35], [36], [37], [38].

Линейные неравенства имеют большое самостоятельное значение, поскольку многие прикладные задачи, в том числе из области экономики [39], представляются в виде таких систем.

Нас интересует подраздел линейного программирования - целочисленное линейное программирование (англ. integer linear programming, ILP), в котором на все или некоторые (смешанное целочисленное программирование) переменные накладывается ограничение целочисленности.

GPU имеет тысячи более энергоэффективных ядер, созданных для выполнения нескольких задач одновременно. Представляется актуальным использовать возможности GPU для задачи разрешения ограничений. Таким образом направление вычислений эволюционирует от «централизованной

обработки данных» на центральном процессоре до «совместной обработки» на CPU и GPU. Всё чаще GPU применяется для решения вычислительных задач, выходящих за рамки первоначального предназначения этих устройств. В связи с чем в работе исследуются возможности графических карт применительно к задаче целочисленного программирования.

Технология GPGPU (general-purpose computing for graphics processing units) позволила увеличить на несколько порядков вычислительные возможности компьютеров, пропорционально уменьшив затраты на программное обеспечение. Она позволяет использовать ресурсы видеокарт для неграфических вычислений. Основным производителем видеокарт с аппаратной поддержкой GPGPU является NVIDIA с комплексом CUDA, которая обеспечила рост популярности GPGPU за счёт упрощения процесса создания программ, использующих возможности GPU. Ещё одной популярной реализацией техники GPGPU является OpenCL (open computing language) - фреймворк для написания программ на различных графических и центральных процессорах. Далее в работе рассматривается технология Cuda.

Постановка задачи

Цель работы - реализация алгоритма Branch-and-Cut с использованием подходов массивного параллелизма и возможностей Nvidia CUDA. Для достижения цели необходимо решить ряд подзадач. При составлении плана использовался план разработки ПО, представленный в [40]:

- Изучить предмет исследования - методы решения задач целочисленного программирования.
- Изучить инструмент исследования - технология Cuda.
- Выработать требования к входным данным и результатам.
- Реализовать алгоритм branch-and-Cut последовательно на CPU.
- Исследовать алгоритм на наиболее часто повторяющиеся вычисления.
- Исследовать возможности распараллеливания найденных участков на GPU.

- Создание параллельного алгоритма Branch-and-Cut, учитывая особенности CUDA.
- Детальное проектирование алгоритма.
- Кодирование, отладка и оптимизация.
- Тестирования.
- Интерпретация результатов.

Обзор существующих решений

Так как методы решения задач смешанного целочисленного программирования представляют большой коммерческий интерес, наибольшего прогресса в этой области коммерческие разработчики, включая Cplex от IBM ILOG, Lingo от Lindo Systems Inc. и Xpress от FICO. Но разработанный код скрыт правообладателями.

К методам целочисленного линейного программирования относят:

- метод отсечений (например, метод Гомори);
- приближённый метод (прямой алгоритм, предложенный Р.Д.Юнгом и Ф.Гловером)
- метод ветвей и границ (Branch-and-Bound);
- комбинированные методы (Branch-and-Cut или Cut-and-Branch);
- использование абсолютной унимодулярности матрицы (сильно ограничивает множество решаемых систем уравнений);
- эвристики:
  - табу-поиск - tabu-search [41];
  - алгоритм восхождение на вершину - hill climbing [42];
  - алгоритм имитации отжига [43];
  - муравьиный алгоритм [44];
  - нейронная сеть Хопфилда [45];

Недостатком эвристических методов является то, что если алгоритм не может найти решение, то невозможно определить, нет решения или есть, но алгоритм не смог его найти.

ГЛАВА 2

Методика решения задачи

Метод ветвей и границ

Метод ветвей и границ (англ. Branch-and-Bound) - это метод комбинаторной оптимизации для решения задач целочисленного программирования, то есть таких задач, решение которых целочисленно. Для метода ветвей и границ необходимы две процедуры: ветвление и нахождение оценок (границ). Процедура ветвления состоит в разбиении множества допустимых значений переменной x на подобласти (подмножества) меньших размеров. Процедуру можно рекурсивно применять к подобластям. Полученные подобласти образуют дерево, называемое деревом поиска или деревом ветвей и границ. Вершинами этого дерева являются построенные подобласти.

Метод использует решение системы линейных неравенств без наложения ограничения целочисленности - релаксационных задач, которые имеют вид:

$$\hat{c} = \min\{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n\}$$

$X_{LP} = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ - множество допустимых значений.

В приведённом ниже алгоритме Branch-and-Bound используются обозначения: L - множество активных задач, R - начальная задача.

[init] $L = \{R\}$, $\hat{c} = \infty$;

[abort] if $L = \emptyset$, return $x^* = \hat{x}$, $c^* = \hat{c}$;

[select] выбрать $Q \in L$, $L = L \backslash \{Q\}$;

[solve] решить Q_{relax} - задачу Q без ограничения целочисленности. Если нет решения, то $\bar{c} = \infty$. Иначе имеет оптимальное решение $\bar{x}$ и соответствующее значение $\bar{c}$;

[bound] if $\bar{c} \geq \hat{c}$, goto [abort];

[check] if $\bar{x}$ целочисленное, то $\hat{x} = \bar{x}$, $\hat{c} = \bar{c}$, и goto [abort];

[branch] разбить Q на подзадачи $Q = Q_1 \cup \cdots \cup Q_k$, $L = L \cup \{Q_1 \cup \cdots \cup Q_k\}$ и goto [abort].

Рис. 2.1 иллюстрирует, как происходит разбиение задачи в алгоритме на стадии [branch].

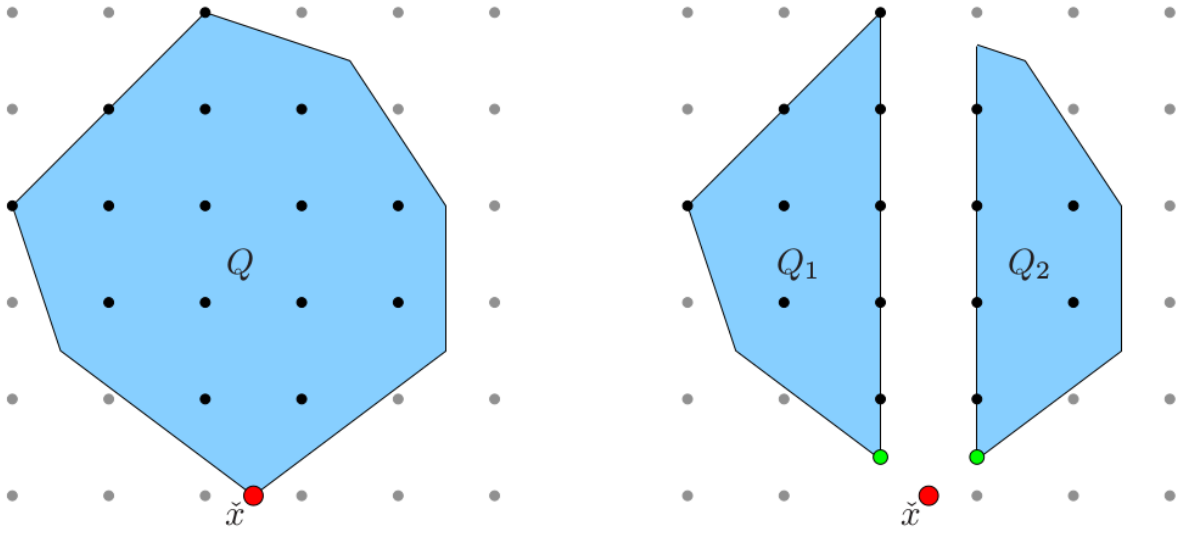

Рисунок 2.1 — Разбиение задачи Q на подзадачи $Q_1 \cup \cdots \cup Q_k$

Уточним, на какие задачи разбивается общая задача. Наиболее популярный метод состоит в выборе переменной, по которой мы ветвимся, а задачи $Q = Q_1 \cup Q_2$, где $Q_1 = Q \cap \{x_j \leq \lfloor \bar{x}_j \rfloor\}$ и $Q_2 = Q \cap \{x_j \geq \lceil \bar{x}_j \rceil\}$. Более сложное ветвление или ветвление на большее количество подзадач редко используется, тем не менее в некоторых задачах также может быть эффективным [46], [47], [48].

До сих пор остаётся открытым вопрос о стратегии выбора переменной для процедуры ветвления. К сожалению, нет оптимального метода и для выбора хорошей стратегии опираются на вычислительный эксперимент.

Опишем наиболее популярные методы ветвления:

- Ветвление по переменной, значение которой наиболее близко к целому значение или наиболее далеко от него:
  - стандартное правило, наиболее простое;
  - не всегда эффективно.
- Сильное ветвление:
  - каждая переменная проверяется на наибольшее влияние на целевую функцию, т.е. при ветвлении по которой, фукнция изменится максимально

  - метод включает в себя преждевременное решение некоторых задач без ограничения целочисленности и выбирает лучшее;
  - эффективно относительно размера дерева задач, но затратно относительно времени.
- Ветвление с учётом превдостоимости:
  - сохраняются треки каждой из переменных и выбирается та, изменение которой, возможно наибольшее, т.е. которая обладает наибольшей псевдостоимостью;
  - попытка оценить стоимость ветвления, основываясь на истории ветвления по данной переменной;
  - неэффективно в начале алгоритма.

Здесь приведены не все. Для более исчерпывающего исследования методов ветвления можно обратиться к [49], [50], [51].

Основные способы выбора узла в дереве задач, каждый из которых имеет свои преимущества:

- Обход дерева в глубину [52] всегда выбирает дочернюю подзадачу относительно нынешней.
- Выбор лучшей задачи (более подробно описан в параграфе 3.4).

Симплекс-метод

Решать релаксационную задачу в алгоритме Branch-and-Bound можно симплекс-методом.

Симплекс-метод - это алгоритм решения оптимизационной задачи линейного программирования, путём перебора вершин выпуклого многогранника в многомерном пространстве [53].

Прямой симплекс метод решает задачу:

$$c^T x \to \max,\ Ax \le b,\ x \ge 0,\ b \ge 0.$$

Двойственный симплекс-метод решает задачу:

$$c^T x \to \max,\ Ax \le b,\ x \ge 0,\ c \ge 0.$$

Алгритм симплекс-метода состоит из трёх частей: поиск строки, поиск столбца, трансформация матрицы:

1. Поиск строки - pivot_row:
   а) ищем первый отрицательный элемент, кроме нулевого, в нулевом столбце;
   б) если его нет, завершаем алгоритм.
2. Поиск столбца:
   а) не считая нулевого элемента, ищем отрицательные элементы в pivot_row и сохраняем колонну, которой принадлежит этот элемент и нормированную по модулю данного элемента, в некотором временном множестве P;
   б) выбираем лексикографически наименьшую колонну из множества P - pivot_col.
3. Трансформация матрицы - применяем гауссовы преобразования таким образом, чтобы pivot_row обнулилась, кроме элемента, соответствующего pivot_col. После преобразований этот элемент должен быть равен -1.

Метод отсечений

Метод Branch-and-Bound может быть усовершенствован методами отсечений. Отсечениями называют линейные неравенства, которые удовлетворяются целочисленными решениями задачи линейного программирования, но могут нарушаться решениями общей задачи оптимизации.

Метод отсечений приближает решение к целочисленному методом сведения исходной допустимой области к выпуклой оболочке её одпустимых целочисленных точек.

Таким образом в алгоритме Branch-and-Bound мы усиливаем релаксационную задачу. Отсечение добавляется на основе результата решения предыдущей релаксационной задачи.

Наибольший вклад в исследование области отсечений внёс Гомори [54], который доказал что целочисленная задача может быть решена за конеч-

ное число шагов с помощью одного подхода отсечений без ветвления [55]. К сожалению, предложенные им отсечения не были эффективными и медленно сходились, поэтому идея отсечений не рассматривалась много лет. Но спустя некоторое время работа Balas [56] 1996 г. показала что метод отсечений может быть более эффективным при соединении с методом Branch-and-Bound. Наиболее подробное описание методов можно найти в [57], [58], [59], [51].

Способы построения отсечений:

- mixed integer rounding cuts [60];
- gomory mixed integer [55];
- lift-and-project cuts [61], [62];
- lifted cover cuts [63], [64], [65];
- GUB cover cuts [66];
- complemented mixed integer rounding cuts [67];
- strong Chvátal-Gomory cuts [68], [69];
- flow cover cuts [70], [71].

В [72] подробно разобран пример решения двумерной задачи целочисленного программирования методом Branch-and-Bound и методом отсечений, проиллюстрированный на рис. 2.2.

Используемый метод

Решается задача линейного программирования в каноническом виде:

$$Ax \leq b$$

$$x \geq 0$$

На вход алгоритму подаётся матрица коэффициентов A размера $m \times n$. Сначала к матрице добавляется отрицательная единичная матрица.

$$\bar{A} = \begin{bmatrix} -E \\ A \end{bmatrix}$$

Так учитывается положительность переменных.

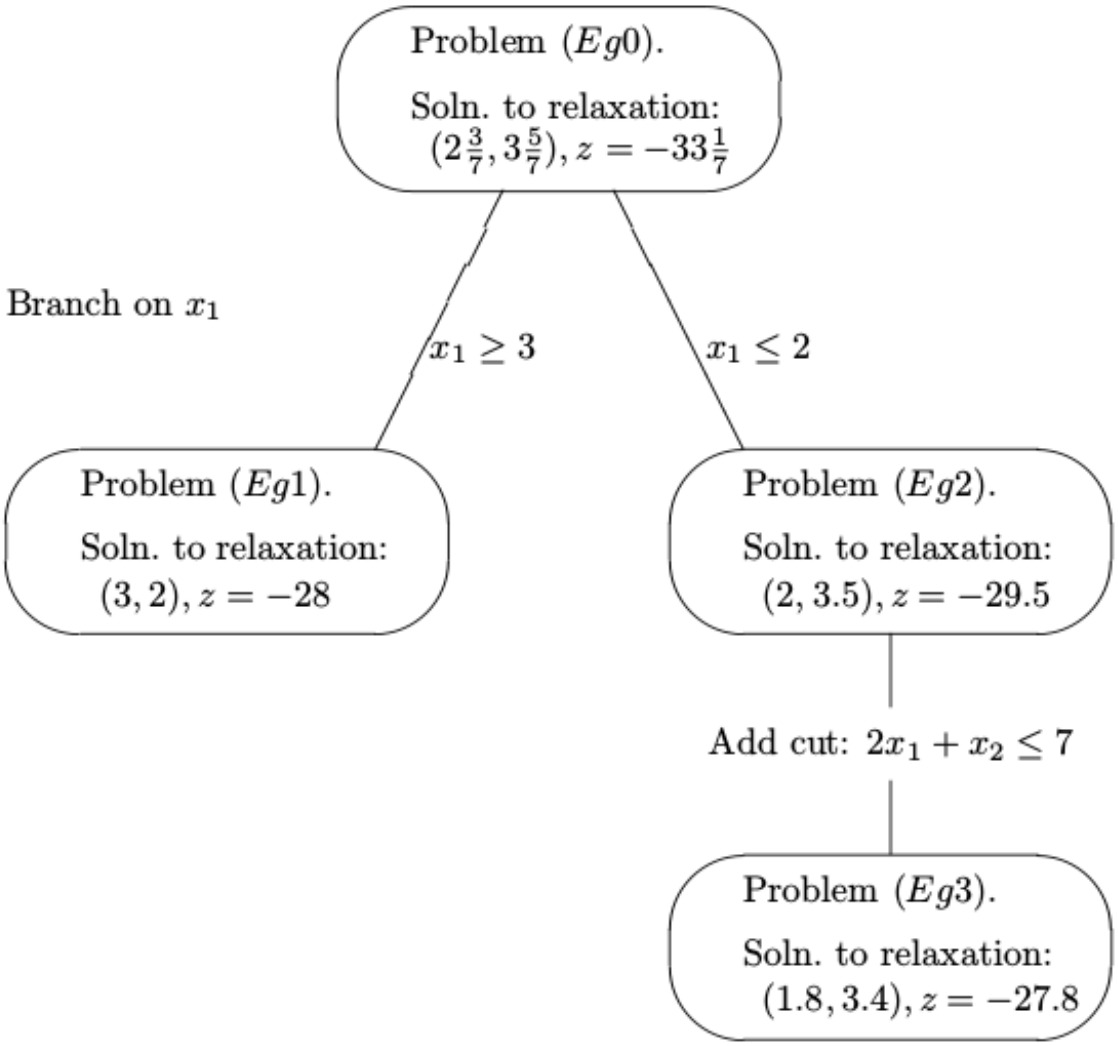

Рисунок 2.2 — Пример решения двумерной задачи методом Branch-and-Cut

Как известно, метод ветвления на основе псевдостоимости является наиболее эффективным относительно времени исполнения программы, но при этом может быть использован не сначала программы. В начале алгоритма используем метод наипростейшего ветвления. То есть относительно значения наименее близкого к целочисленному.

Выбор соответствующего отсечения совершался на основе результатов вычислений, приведённых в [73], таким образом был выбран метод MIR (англ. mixed integer rounding cuts).

На каждом шаге, кроме первого, решается одновременно две задачи,

которые являются дочерними относительно одного и того же родителя. Поэтому на каждой итерации необходимо выбирать уже решённую задачу, чтобы решать её дочерние.

Такой подход неэффективен, если реализовывать задачу последовательно. Но массивное распараллеливание даёт нам возможность решать такие задачи одновременно. Отсюда и нестандартный метод выбора следующей решаемой задачи - по максимальному количеству целых переменных.

Так как оптимальное решение после отсечения не является допустимым решением новой задачи линейного программирования, но является её двойственным допустимым решением, то для решения новой задачи выгоднее использовать двойственный симплекс-метод. Кроме того, так как правильное отсечение является неравенством, то удобнее использовать столбцовую форму записи и считать, что ограничения исходной задачи заданы в форме неравенств [74].

Внутри симплекс-метода выбор строки совершается по первой строке с наименьшим индексом. Таким образом, если добавить отсечение или ветвление в конец матрицы, то симплекс-метод сначала пройдёт все итерации, что первая матрица. Отсюда идея оптимизации алгоритма с помощью матрицы трансформации. Обозначим её через T, допустим вследствие симплекс-метода за k итераций матрица $\bar{A}$ изменилась:

$$\begin{bmatrix} -E \\ A \end{bmatrix} \times T = \begin{bmatrix} -T \\ A' \end{bmatrix}$$

Тогда при добавлении отсечений к матрице $\bar{A}$, обозначим их C, за те же k итераций изменения выглядят так:

$$\begin{bmatrix} -E \\ A \\ C \end{bmatrix} \times T = \begin{bmatrix} -T \\ A' \\ C' \end{bmatrix}$$

Используя эти свойства, после первого шага мы получаем матрицу трансформацию. И на следующих шагах перемножаем её с матрицами отсечений и запускаем в симплекс с $k + 1$ шага.

Реализованный алгоритм:

Input Матрица коэффициентов линейных ограничений.

Output Вектор целочисленных значений переменных или констатация, что таковых нет.

[simplex] симплекс-метод применяется к входной матрице, если решения нет - return нет целочисленного решения, иначе инициализируем матрицу трансформации;

[branch] ветвимся и определяем место решённой задачи в очереди, если нет переменных ветвления, то return целочисленное решение;

[order] если в очереди больше нет задач, то return нет целочисленного решения, следующая в очереди задача имеет максимальное число целых переменных;

[init] инициализируем дочерние задачи выбранной решённой;

[simplex] решаем задачи, используя симплекс-метода;

[delete] если одна из задач или обе не имеют решения, удаляем ветку дерева;

[cuts] если задача имеет решения и среди переменных решения есть нецелочисленные, проверяем задачу на возможность добавления отсечений, если таковые есть - снова решаем задачу;

[loop] переходим в пункту [branch]

Описанный выше алгоритм не решает задачу оптимизации, поэтому завершается, как только находится первое целочисленное решение. А целевая функция может быть любой.

ГЛАВА 3

Практическая реализация метода

язык программирования, технология CUDA, компилятор, количество строчек кода)) Текущая версия содержит столько-то строчек кода.

Сравнение GPU и CPU

Важно понимать различия между тем, как спроектированы CPU и GPU, которые изначально предназначены для разных типов вычислений. Поэтому прежде, чем перейти к CUDA, рассмотрим эти различия. Наиболее важные из них это модель управления нитями и отличия физической памяти.

CPU поддерживает выполнение сильно ограниченного числа совместно выполняющихся нитей. Серверы, содержащие 4 шестиядерных процессора, могут паралелльно исполнять лишь 24 нити параллельно (или 48 если ЦПУ поддерживет гиперпоточность). В это время наименьшее объединение потоков на видеокарте содержит 32 нити, что называется варпом. Современные карты NVIDIA поддерживает до 1536 активных нитей на один мультипроцессор [75]. Если видеокарта содержит 16 мультипроцессоров, то число может превышать 24 000 (Рис. 3.1).

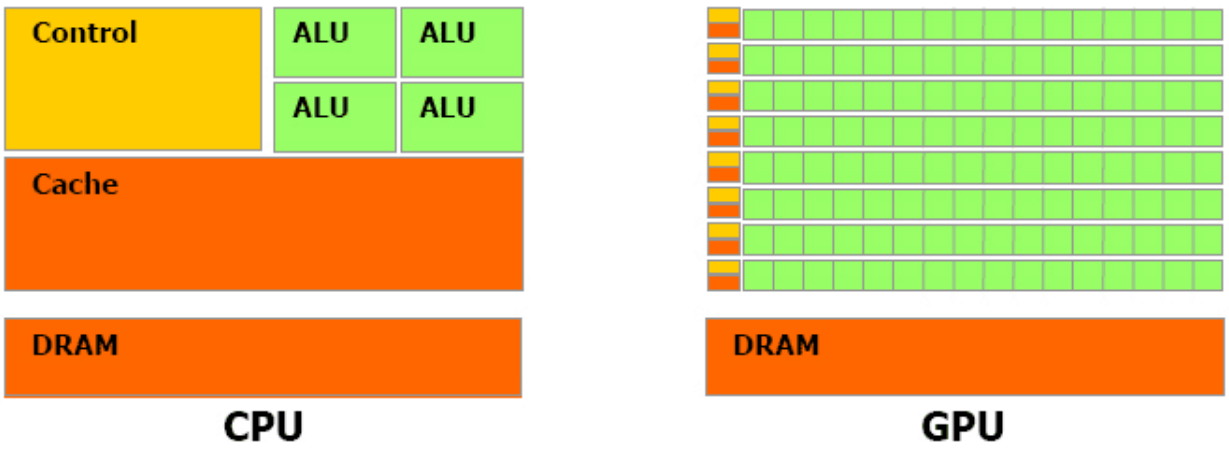

Рисунок 3.1 — Сравнение GPU и CPU

Многопоточность в графических процессорах реализована на аппарат-

ном уровне. Ядра GPU проектируются для большого числа параллельно выполняемых инструкций, в то время как ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью. Ядра CPU используют MIMD-архитектуру – множественный поток команд и данных. Каждое ядро работает отдельно от остальных, исполняя разные инструкции для разных процессов. GPU используют SIMD-архитектуру (одиночный поток команд, множество потоков данных) и специально рассчитанные на работу с ней контроллеры памяти. Ядра мультипроцессора в GPU исполняют одни и те же инструкции одновременно.

На CPU каждое переключение между потоками неизбежно ведёт к значительным временным задержкам продолжительностью в несколько сотен тактов. В то же время GPU легко переключается между нитями. Если происходит задержка на одном варпе, GPU начинает исполнение другого. Благодаря тому, что каждая нить содержит свои регистры, нет необходимости в перезаписи регистров при каждом переключении. Ресурсы распределены для каждой нити отдельно, пока не завершается её исполнение.

Память CPU и GPU, как правило, расположена отдельно и соединена шиной PCI Express.

В цетнральных процессорах большие количества транзисторов и площадь чипа идут на буферы команд, аппаратное предсказание ветвления и огромные объёмы начиповой кэш-памяти. Все эти аппаратные блоки нужны для ускорения исполнения немногочисленных потоков команд. Видеочипы тратят транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов. Вышеперечисленное не ускоряет выполнение отдельных потоков, но позволяет чипу обрабатывать несколько тысяч потоков, одновременно исполняющихся чипом и требующих высокой пропускной способности памяти [76].

CPU, будучи универсальным вычислительным устройством, эффективно справляется с целым спектром различных задач, в то время как предназначение графических процессоров гораздо более узконаправленное.

В задачах с множественными ветвлениями и переходами графический процессор не столь эффективен как центральный.

В связи с особенностями графических процессор они хорошо справляются с задачами, где требуется большое количество параллельных вычислений с большим количеством арифметических операций. Причем элементы данных должны быть независимы (GPU обладает плохим синхронизационным аппаратом) и работа над данными одинакова.

Такой стиль программирования является обычным для графических алгоритмов и многих научных задач, но требует специфического программирования. Зато такой подход позволяет увеличить количество исполнительных блоков за счёт их упрощения.

Технология CUDA

CUDA (Compute Unified Device Architecture) является архитектурой параллельных вычислений от NVIDIA, позволяющей существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров). Для реализации новой вычислительной парадигмы компания NVIDIA изобрела архитектуру параллельных вычислений CUDA, и обеспечивающую необходимую базу разработчикам ПО.

Программирование на CUDA включает в себя код на двух различных платформах совместно: host на одном или нескольких CPU и device на одном или нескольких NVIDIA GPU, поддерживающих CUDA.

Платформа параллельных вычислений CUDA обеспечивает набор расширений для языков C и C++, позволяющих выражать как параллелизм данных, так и параллелизм задач на уровне мелких и крупных структурных единиц. Всё это является несомненными преимуществами использования CUDA-технологии на ряду с доступностью.

Компания NVIDIA предоставляет показательные примеры кода на CUDA [77] в свободной доступе на английском языке и в продаже на русском, а также как проводить подробную оптимизацию кода [78]. В том числе предоставляются две платформы Nvidia Nsight Edition для разработчиков

в Eclipse и Microsoft Visual Studio. Дополнительную информацию также можно найти в [79], [80].

Перечислим основные характеристики CUDA:

- Унифицированное программно-аппаратное решение для параллельных вычислений на видеочипах NVIDIA.
- Большой набор поддерживаемых графических плат (от мобильных до мультичиповых).
- В качестве языка программирования используется расширенный вариант языка C.
- Поддерживает взаимодействие с графическими API OpenGL и DirectX.25
- Имеется поддержка 32- и 64-битных операционных систем: Windows XP, WindowsVista, Linux и MacOSX.
- Возможность разработки на низком уровне.
- CUDA обеспечивает доступ к быстрой разделяемой памяти, которая может быть использована для межпоточного взаимодействия.
- Нативный компилятор/отладчик: nvcc/cuda-gdb[linux/mac os], расширение к msvc/TotalView [win].

Команда разработчиков CUDA создала набор программных уровней для работы с GPU, которые отображены на Рис. 3.2. Как можно видеть, CUDA обеспечивает два API:

- высокоуровневый API: CUDA Runtime API;
- низкоуровневый API: CUDA Driver API;

Каждый вызов фукнции уровня Runtime состоит из более элементарных инструкций уровня Driver API. Стоит помнить, что два API взаимно исключают друг друга. При использовании одного, нельзя использовать функции другого уровня. Driver API сложнее и требует больше знаний о NVIDIA GPU, но при этом он более гибок и предоставляет весь контроль программисту.

Потоком в CUDA назвается базовый набор данных, который требуется обработать. В отличие от потоков CPU, переключение контекста между двумя потоками CUDA не является ресурсоёмкой операцией. 32 потока объединяются в один варп (англ. warp) - минимальный объём данных, обраба-

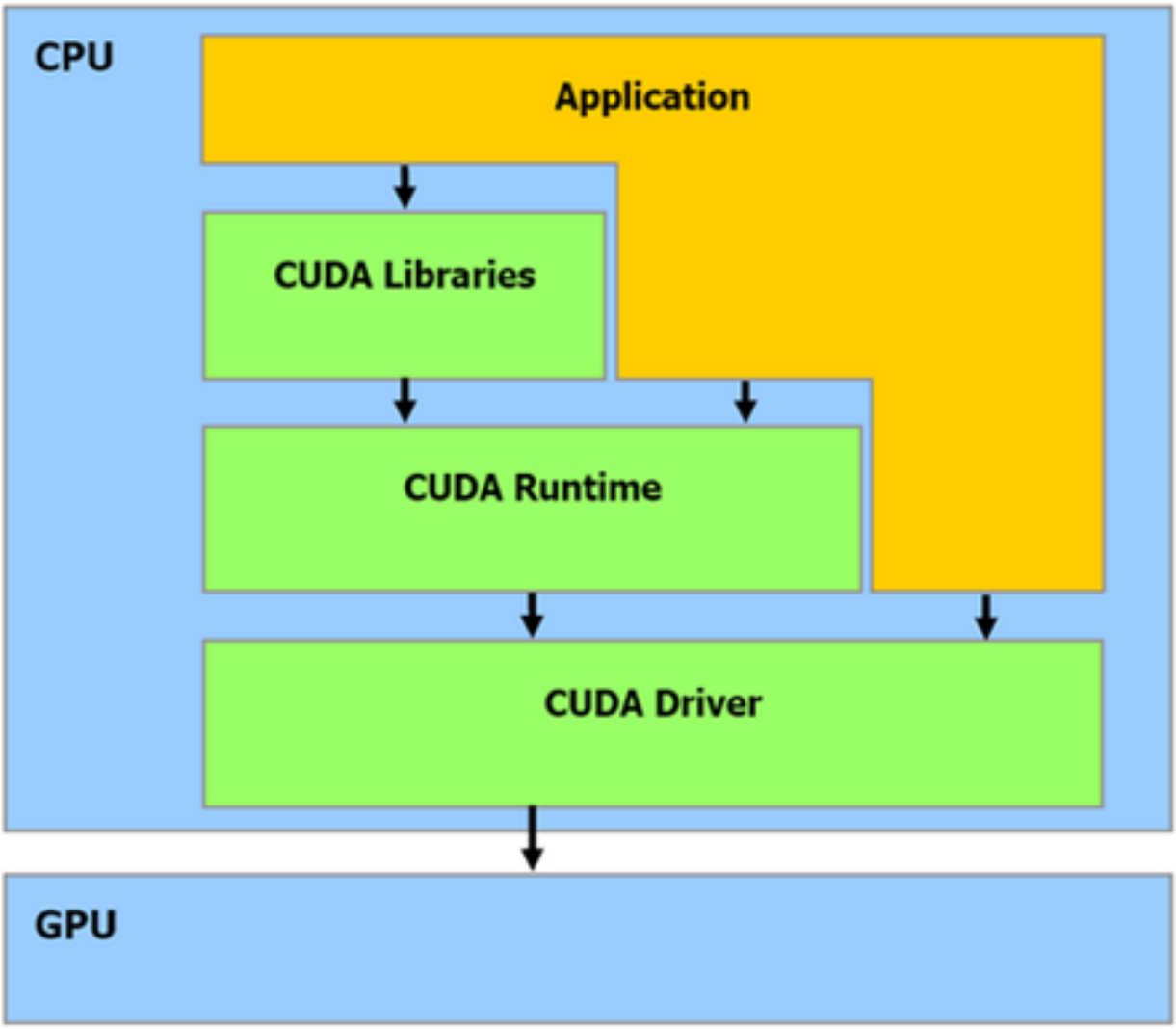

Рисунок 3.2 — Набор программных уровней для работы с GPU

тываемых SIMD-способом в мультипроцессорах CUDA. Вследствие этого, даже при ветвлении внутри программы все нити внутри одного warp'а исполняют сначала одну ветку, а затем другую.

Также потоки составляют блоки, которые в свою очередь формируют сетки. Если GPU имеет мало ресурсов, то он будет выполнять блоки последовательно.

Физический уровень - видеокарта:

- Streaming Multiprocessor (SM) - "процессор" на видеокарте.
- Bandwidth – внутренняя пропускная способность. Влияет на копирование dev-dev.
- PCI-express - шина общения с хостом. Влияет на скорость копирова-

ния host-dev.

Логический уровень - kernel (ядро) - функция, полностью вычисляющаяся на графическом устройстве (Рис. 3.3):

- Grid (геометрия сетки блоков нитей) – свойство запуска функции, определяющее количество запускаемых блоков вычислений. Важна как сама геометрия, так и максимизация количества.
- Thread block (блок нитей) - множество нитей, имеющих общую r/w память, со scope адресации внутри этого блока.
- Warp – множество одновременно исполняющихся нитей внутри одного cuda core. Имеет особый смысл в случае ветвлений: блок нитей бьется на две части, которые исполняются по сути последовательно. Также понятие warp сильно связано с выравниваниями в памяти.

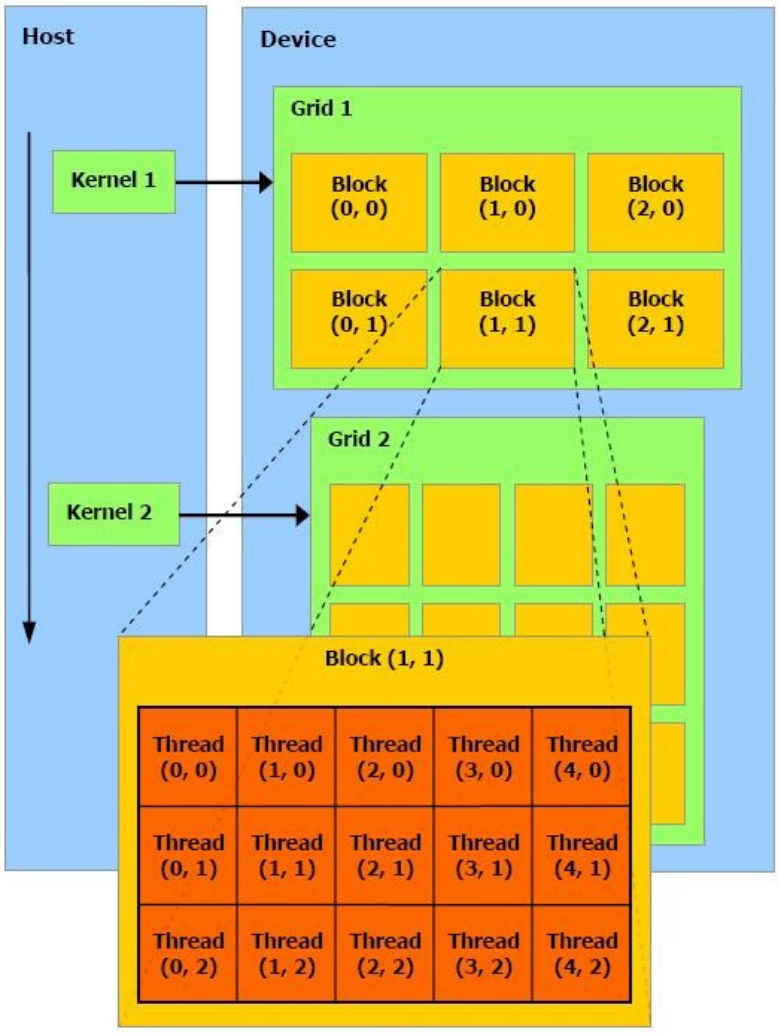

Рисунок 3.3 — Конструкция grid

Видеокарта может содержать несколько (от 2 до 2000) потоковых мультипроцессоров (streaming multiprocessor) (Рис. 3.4). Они в свою очередь содержат восемь вычислительных устройств и два суперфункциональных устройства SFU (Super Function Unit), где инструкции выполняются по принципу SIMD, что в данном случае означает применение одной инструкции ко всем потокам в варпе.

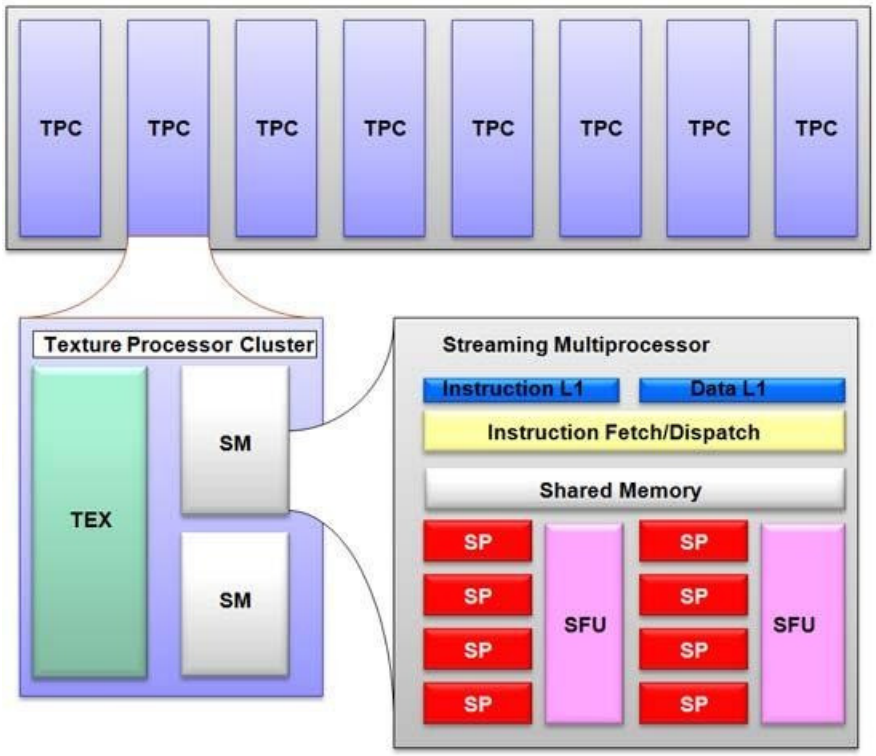

Рисунок 3.4 — Конструкция TPC и SM

Потоковые мультипроцессоры работают следующим образом: каждый такт начало конвейера выбирает варп, готовый к выполнению, и запускает выполнение инструкции. Чтобы инструкция применилась ко всем 32 потокам в варпе, концу конвейера потребуется четыре такта, но поскольку он работает на удвоенной частоте по сравнению с началом, потребуется только два такта (с точки зрения начала конвейера). Поэтому, чтобы начало конвейера не простаивало такт, а аппаратное обеспечение было максимально загружено, в идеальном случае можно чередовать инструкции каждый такт, классическая инструкция в один такт и инструкция для SFU в другой.

Мультипроцессор имеет 8192 регистра, которые общие для всех потоков всех блоков, активных на мультипроцессоре. Число активных блоков на мультипроцессор не может превышать восьми, а число активных варпов ограничено 24 (768 потоков).

Если распараллеливать задачу на CPU, то следить за потоками нужно самим. При распараллеливании на видеокартах известен только номер нити или блока. А также регулируется количество потоков, которые все исполняют одну последовательность инструкций. Запуск с хоста device-функции - это kernel. У этого ядра существует конфигурация. Внутри блока нити находятся физически близко и могут делить общую память. Адресация внутри сетки блоков происходит по идентификации блоков и нитей.

Эффективность распараллеливания на GPU также предъявляет требования к данным. Наибольшая производительность достигается на больших однотипных объёмах данных, над которыми производятся одинаковые операции. Таким образом используются тысячи или десятки тысяч параллельных потоков, чтобы GPU не простаивало.

Также желательно, чтобы память, используемая рядом расположенными нитями, располагалась последовательно. Некоторые шаблоны доступа к памяти позволяют аппаратным средствам объединять группы считываний или записи нескольких элементов данных в одну операцию. Данные, расположенные недостаточно близко, будут способствовать замедлению при использовании на CUDA.

Для использования данных на CUDA необходимо переместить их из памяти host на device через шину PCI Express. Такие перемещения затратны и должны быть минимизированы. Поэтому использование CUDA, если задействовано малое количество нитей, не даст выигрыша в такой ситуации.

Оптимизация программы CUDA также состоит в получении оптимального баланса между количеством блоков и их размером. Больше потоков на блок будут полезны для снижения задержек работы с памятью, но и число регистров, доступных на поток, уменьшается. Более того, блок из 512 потоков будет неэффективен, поскольку на мультипроцессоре может

быть активным только один блок, что приведёт к потере 256 потоков. Поэтому NVIDIA рекомендует использовать блоки по 128 или 256 потоков, что даёт оптимальный компромисс между снижением задержек и числом регистров для большинства ядер/kernel.

За формирование и компиляцию ядер отвечает CPU. Видеочип просто принимает уже скомпилированное ядро и создает его копии для каждого элемента данных. Каждое из ядер исполняется в своем собственном потоке.

Иерархия памяти

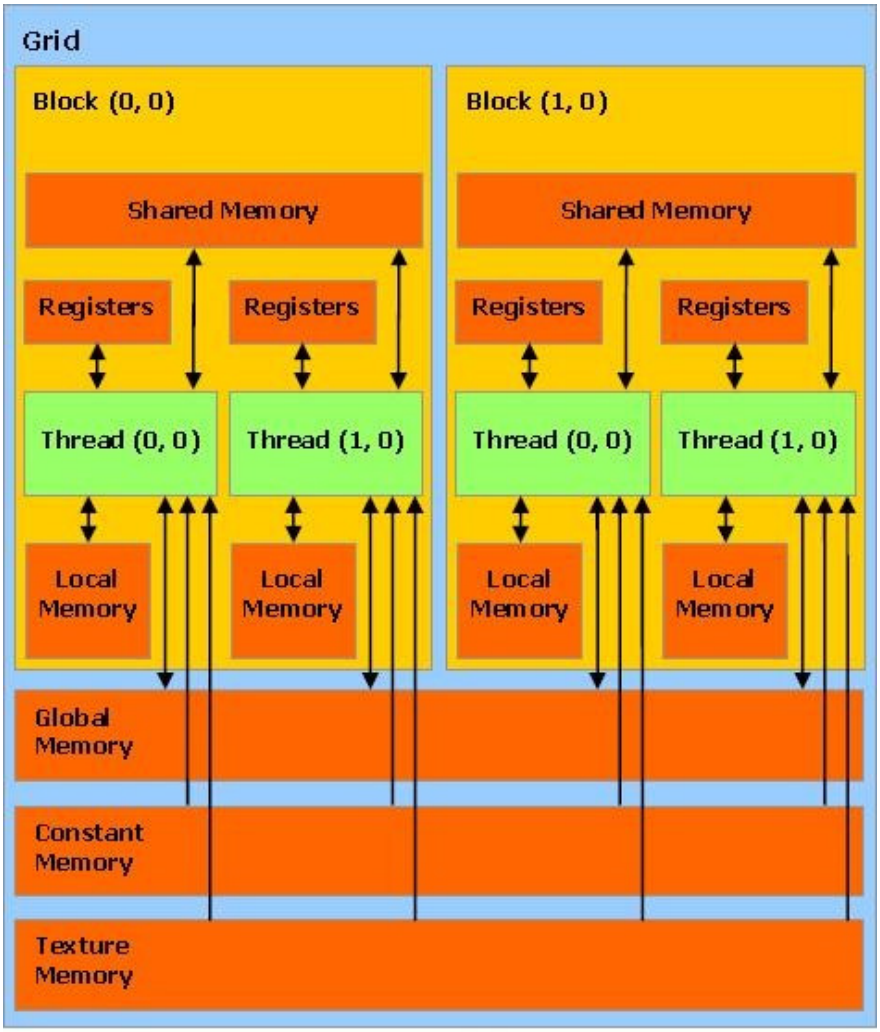

Рисунок 3.5 — Иерархия памяти CUDA

Вычислительные возможности начиная с 2.x:

- Глобальная память (чтение и запись):

  - медленная, но может быть кеширована;
  - последовательное и выровненное чтение по 64 или 128 байт для более быстрого использования. Иначе замедление в 10-100 раз;
  - плохая параллельная скорость доступа, если пользоваться ей напрямую;
  - адресация - прямая, по указателям;
  - видна всей сетке блоков.
- Текстурная память (только чтение) - кэш, оптимизированный для двумерного доступа.
  - более удобный метод доступа с геометрической точки зрения;
  - неизменяемая;
  - кэшируемая.
- Константная память:
  - содержит константы и аргументы функций ядра;
  - имеет особенные инструкции заполнения;
  - сравнительно быстрая, но её мало;
  - видна всей сетке блоков.
- Разделяемая память (48кБ на 1 SM):
  - быстрая, но подвержена конфликтам в банке памяти;
  - видна всем нитям внутри блока.
- Локальная память:
  - используется для любых данных, что не влезли в регистровую память;
  - часть глобальной памяти, поэтому медленная;
  - автоматическое выравнивание обращений;
  - видна одной нити.
- Регистровая память - 32768 32-битных регистра на 1 SM.

Константная память целиком кэшируется. Поэтому получается достаточно быстро. У каждой нити свои регистры. Локальные переменные могут попасть как в shared так и global. Блоки обрабатываются конвейерным образом. shared память очень быстрая. аллоцируется статически.

Два потока из разных блоков не могут обмениваться информацией

между собой во время выполнения. Пользоваться общей памятью не так просто. Общая память всё оправдана за исключением случаев, когда несколько потоков пытаются обратиться к одному банку памяти, вызывая конфликт.

Мультипроцессоры также могут обращаться к видеопамяти, но с меньшей пропускной способностью и большими задержками. Поэтому NVIDIA оснастила мультипроцессоры кэшем, хранящим константы и текстуры.

Доступ к глобальной памяти имеет свои трудности. Выравнивание доступа к памяти – самый сложный и самый важный аспект в программировании под Nvidia CUDA. Доступ к памяти называется coalesced в том случае, когда массовая операция доступа из одного warp к памяти укладывается в одну транзакцию.

Для обмена информацией внутри блока между потоками используется разделяемая память (англ. shared memory). PРазделяемая память устроена так, что доступная блоку память разбита на банки памяти с раздельными путями доступа. В случае, если две нити пытаются обратиться в один банк памяти одновременно, возникает “конфликт” и доступ они получают последовательно, а не параллельно. Максимальное количество нитей в блоке, пытающихся обратиться к одному банку, называется “глубиной конфликта доступа”.

Для интегрированных GPU использование нуль-копируемой памяти всегда даёт выигрыш, потому что эта память в любом случае физически разделяется с CPU. В тех случаях, когда входные и выходные буферы используются ровно один раз, повышение производительности будет наблюдаться и на дискретном GPU.

Наибольший выигрыш в том числе может быть достигнут за счёт асинхронизации копирования и исполнения ядра. Примеры, иллюстрирующие это, подробно описаны в [77] и [78].

Обозначенное угловыми скобками ядро также принимает необязательный аргумент - номер потока исполнения (англ. stream). Этот запуск ядра выполняется асинхронно и используется с фукнциями асинхронного копирования памяти. Они выполняются параллельно друг другу и паралельно

коду CPU. Поэтому не исключено, что последующий код CPU начнёт выполнятся ещё до того, как закончится копирование памяти или исполнение ядра. Для этого в CUDA существуют функции синхронизаци. Гарантируется лишь, что операции копирования и исполнения ядра, помещённые в один и тот же поток, будут выполняться последовательно. То есть поток ведёт себя как упорядоченная очередь задач для GPU. Подробнее об этом можно узнать в [81] и найти примеры в [77]

Программная реализация

Исследуем последовательный код на CPU на места, наиболее эффективно поддающиеся рапараллеливанию. Для этого обратимся к профайлеру gprof. Ниже приведены результаты исполнения только для функций, исполнение которых составляет больше 0.009 процентов от общего времени.

% time	cumul. seconds	self seconds	calls	self s/call	total s/call	name
99.67	13.21	13.21	22756	0.00	0.00	dualSimplex
0.38	13.26	0.05	22756	0.00	0.00	pivotColumn
0.08	13.27	0.01	13986754	0.00	0.00	cmp

Как видно из таблицы, в первую очередь необходимо оптимизировать метод трансформации матрицы. Во-вторых, оптимизировать произведение матриц.

Уже существуют известные оптимизации произведения матриц. Можно воспользоваться обучающим алгоритмом из [82], а также уже известной реализацией из библиотеки cuBLAS на GPU, которая есть аналог функции BLAS на CPU.

BLAS (Basic Linear Algebra Subprograms) - интерфейс, базовые подпрограммы линейной алгебры:

а) Векторные операции вида $y \Leftarrow \alpha x + y$ - операции скалярного произведения, взятия нормы вектора и др.

б) Операции матрица-вектор вида $y \Leftarrow \alpha Ax + \beta y$, решение $Tx = y$ для

x с треугольной матрицей T и др.

в) Операции матрица-матрица вида $C \Leftarrow \alpha AB + \beta C$, решение $B \Leftarrow \alpha T^{(}-1)B$ для треугольной матрицы T и др.

ГЛАВА 4

Анализ метода

Экспериментальный анализ корректности метода

Первоначально при выборке тестов учитывался так же факт того, что важен анализ как эффективности, так и корректности. В работе корректность алгоритма устанавливается при помощи сравнительных тестов в формате DIMACS, которые представляют собой КНФ формулу. Задача выполнимости сводится к нахождению булевого решения (а при добавлении ограничений $x_i <= 1$ и $x_i >= 0$ к нахождению целочисленного решения) некоторой системы линейных уравнений, составленных на основании КНФ формулы. Таким образом мы проверяем корректность алгоритма, зная выполнима формула или нет.

Экспериментальный анализ эффективности метода

Было проведено более подробное исследование эффективности выбранного метода. Оно состояло из трёх этапов:

1. Фиксируем количество ограничений и меняем количество переменных.
2. Фиксируем количество переменных и меняем количество ограничений.
3. Фиксируем отношение ограничений к переменным и меняем число переменных.

Симплекс-метод широко используется и хорошо работает на практике: многочисленные эксперименты подтверждают почти линейную по числу переменных оценку числа итераций. Однако можно показать, что на специальных примерах симплекс-метод (при некотором правиле выбора направляющего столбца и направляющей строки) работает экспоненциально долго. Первыми такой пример предложили Виктор Кли и Джордж Джеймс

Минти в 1972 г.:

$$\max(2^{n-1}x_1 + 2^{n-2}x_2 + 2x_{n-1} + x_n)$$

$$\begin{cases} x_1 & \leq & 5, \\ 4x_1 + x_2 & \leq & 25, \\ 8x_1 + 4x_2 + x_3 & \leq & 125, \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots & & \\ 2^n x_1 + 2^{n-1}x_2 + 2^{n-2}x_3 + ... + 4x_{n-1} + x_n & \leq & 5^n, \end{cases}$$

В этом случае используемый в работе симплекс-метод выполнит $2^n - 1$ итераций. Хотя для существуют другие правила выбора pivot_row и pivot_col, которые решают задачу за константное число операций, для них также существуют задачи, на которых алгоритм работает экспоненциально.

С другой стороны, алгоритмы решающие задачу линейного программирования за полиномиальное время существуют и, следовательно, она принадлежит классу P. Этот вопрос был решён в 1979 г. Л.Г.Хачияном, российским математиком. Он использовал метод эллипсоидов, разработанный для задач нелинейного программирования Н.З.Шором, Д.Б.Юдиным и А.С.Немировским. Хотя алгоритм Хачияна и его модификафии эффективны с теоретической точки зрени, практически конкурировать с симплекс-методом она, по крайней мере, пока, не могут.

Наложив ограничение целочисленности на все или часть переменных, мы получаем задачу целочисленного линейного программирования, которая оказывается намного более трудной, чем задача линейного программирования. Задача о совместности условий задачи целочисленного программирования принадлежит классу NP-полных.

Алгоритм Branch-and-Bound экспоненциально (2^t итераций) зависит от длины записи коэффициентов задачи целочисленного линейного программирования:

$$\max x_1$$

$$\begin{cases} (2^t + 1)x_1 = 2^t x_2, \\ 0 \leq x_2 \leq 2^t, \\ x_1, x_2 \in \mathbb{Z}. \end{cases}$$

Для установления несовместности условий задачи целочисленного линейного программирования

$$\max(x_1 + x_2 + ... + x_n)$$

$$\begin{cases} 2x_1 + 2x_2 + ... + 2x_n = n \\ x_j \in \mathbb{Z}, (j = 1, 2, .., n) \end{cases}$$

где n - нечётно, Branch-and-Cut требует $2^{\frac{n+1}{2}}$ итераций. Таким образом, трудоёмкость зависит от числа переменных.

Мерой исследования эффективности на практике было выбрано время. Сводка результатов приведена ниже в таблице:

Теоретический анализ метода

Определим понятие масштабируемости. Масштабируемость (англ. scalability) означает способность системы, сети или процесса справляться с увеличением рабочей нагрузки (увеличивать свою производительность) при добавлении ресурсов (обычно аппаратных).

Существует два показателя масштабируемости: сильная и слабая. Сильная масштабируемость показывает, как меняется время решения задачи с увеличением количества процессоров (или вычислительных узлов) при неизменном общем объёме задачи. Сильная масштабируемость определяется законом Амдала. Слабая масштабируемость показывает, как меняется время решения задачи с увеличением количества процессоров (узлов) при неизменном объёме задачи для одного процессора (или узла), и определяется законом Густафсона.

Закон Амдала может определить верхнюю грань ожидаемого ускорения за счёт увеличения количества процессоров с фиксированным объёмом

задачи. Он выглядит как:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

, где S - ускорение выполнения программы, которое по определению равно отношению времени вычисления программы на одном процессоре ко времени вычисления на нескольких процессорах, P - это доля исполнения части кода, которая может быть распараллелена, от общего времени исполнения всего кода, а N - это количество процессоров, на которые может быть распараллелена исследуемая часть кода. Из формулы очевидно следует, что чем больше N, тем больше ускорение.

Исключим N из формулы, устремив его к бесконечности. Отсюда

$$S = \frac{1}{1-P}$$

. В большинстве случаев программы не демонстрируют идеального ускорения. Но закон Амдала указывает на то, что для достижения ускорения стоит увеличивать распараллеливаемую часть. Если она маленькая, то программа не параллелизуема.

Закон Густафсона измеряет ускорение программы при увеличении количества процессоров и неизменном размере задачи на один процессор, т.е. объём задачи увеличивается пропорционально количеству процессоров. Он определяется формулой:

$$S = N + (1 - P(1 - N)$$

, где P и N принимают те же обозначения, что в формуле Амдала.

Цель такого подхода - за заданное время выполнить максимальный объем вычислений.

ГЛАВА 5

Результаты и выводы

Привести здесь отзывы профайлера.

Вывод должен отражать коэффициент распараллеливаемости. То есть без учета того, что GPU как правило медленнее CPU, оцениваем отношение шагов.

Направления дальнейших исследований

СПИСОК ЛИТЕРАТУРЫ

1. R. Dechter. Constraint processing. San Francisco: Morgan Kaufmann, 2003. 481 c.
2. Freuder E. C. Mackworth A. K. Constraint satisfaction: An emerging paradigm. Foundations of Artifficial Intelligence, 2006. 13 c.
3. E. Tsang. Foundations of Constraint Satisfaction. New York: Academic Press, 1993. 421 c.
4. Markowitz H. M., Manne A. S. On the solution of discrete programming problems. Econometrica 25, 195. 84 c.
5. Sutherland I. E. Sketchpad: A Man-Machine Graphical Communication System, PhD thesis. Massachusetts Institute of Technology, Lincoln Lab, 1963.
6. Colmerauer A. Total precedence relations. Journal of the ACM 17, 1970. 14 c.
7. Kowalski R. A. Predicate logic as programming language. Stockholm, Sweden: Sixth IFIP Congress, 1974. 569 c.
8. Jaffar J., Lassez J.-L. Constraint logic programming. Munich: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, 1987. 111 c.

9. The constraint logic programming language CHiP / H. Simonis M. Dincbas, P. van Hentenryck, A. Aggoun, T. Graf [и др.]. Tokyo: Proceedings of the International Conference on Fifth Generation Computer Systems, 1988. 693 с.
10. Colmerauer A. An introduction to Prolog III. Communications of the ACM 33, 1990. 69 с.
11. Г. Карманов В. Математическое программирование. Москва: Наука, 1986. 288 с.
12. L. Balinski M. Integer Programming: Methods, Uses, Computations. Management Science, 1965. 253 с.
13. Kautz H. Selman B. Planning as Satisfiability. Chichester: John Wiley and Sons, 1992. 359 с.
14. Barnier N. Brisset P. Graph coloring for air traffic flow management. Annals of Operations Research, 2004. 163 с.
15. Сараев А.Д. Щербина О.А. Системный анализ и современные информационные технологии // Труды Крымской академии наук. Симферополь: СОНАТ, 2006. 47 с.
16. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences, 1974. 95 с.
17. Hooker J.N. Yan H. Verifying logic circuits by Benders decomposition // Principles and Practice of Constraint Programming. Cambridge: MIT Press, 1994.
18. H. Collavizza M. Rueher, Hentenryck P. Van. CPBPV: A constraint programming framework for bounded program verification. Springer, Berlin, Heidelberg: Lecture Notes in Computer Science, 2008. 327 с.
19. Dick J., Faivre A. Automating the generation and sequencing of test cases from model-based specifications. Springer, Berlin, Heidelberg: Lecture Notes in Computer Science, 2005.
20. Flanagan Cormac. Automatic software model checking via constraint logic. Elsevier, 2004. 253 с.
21. A. Gotlieb B. Botella M. Rueher. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes,

1998. 53 c.

22. C. Boyapati S. Khurshid D. Marinov. Korat: automated testing based on Java predicates. Proc. of International Symposium on Software Testing and Analysis, 2002. 123 c.
23. V. Kumar. Algorithms for constraint satisfaction problems: A survey. AI Magazine, 1992. 32 c.
24. Dechter R. Frost D. Backtracking algorithms for constraint satisfaction problems. University of California: Department of Information and Computer Science, 1999.
25. R. Bartak. Theory and practice of constraint propagation // Proceedings of the Third Workshop on Constraint Programming for Decision and Control (CPDC-0I). Poland: Gliwice, 2001. 7 c.
26. P. Meseguer. Constraint satisfaction problems: an overview. AI Communications, 1989. 3 c.
27. Miguel I. Shen Q. Solution techniques for constraint satisfaction problems: advanced approaches. Artifficial Intelligence Review, 2001. 267 c.
28. R. Dechter. Constraint networks // Encyclopedia of Artifficial Intelligence (2nd edition). New York: Wiley and Sons, 1992. 276 c.
29. W. Hower. Constraint satisfaction. Algorithms and complexity analysis. Information Processing Letters, 1995. 171 c.
30. A.K. Mackworth. Constraint satisfaction // Encyclopedia of Artifficial Intelligence (second edition). New York: Wiley, 1992. 285 c.
31. R. Apt K. Principles of Constraint Programming. New York: Cambridge University Press, 2003. 407 c.
32. Fruehwirth T. Abdennadher S. Essentials of Constraint Programming. Springer, 2003. 144 c.
33. Marriott K. Stuckey P. J. Programming with Constraints: An Introduction. Cambridge: MIT Press, 1998. 483 c.
34. Rossi F. van Beek P. Walsh T. Handbook Of Constraint Programming. Elsevier, 2006. 978 c.
35. P. Van Hentenryck. Constraint Satisfaction in Logic Programming. Cambridge: MIT Press, 1989. 224 c.

36. Van Hentenryck P. Michel L. Deville Y. Numerica: A Modeling Language for Global Optimization. Cambridge: MIT Press, 1997. 210 c.
37. P. Van Hentenryck. The OPL Optimization Programming Language. Cambridge: MIT Press, 1999. 255 c.
38. Van Hentenryck P. Michel L. Constraint-Based Local Search. Cambridge: MIT Press, 2005. 442 c.
39. Канторович Л. В. Математические методы организации и планирования производства. Санкт-Петербург: изд. ЛГУ, 1939.
40. С. Макконнелл. Совершенный код. Русская Редакция, Microsoft Press, 2010.
41. F. Glover. Tabu search-Part II. ORSA Journal on computing, 1989.
42. Russell St. J. Norvig P. Artificial Intelligence: A Modern Approach (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, 2003. 111 c.
43. Kirkpatrick S. Gelatt Jr C. D. Vecchi M. P. Optimization by Simulated Annealing. Science, 1983. 671 c.
44. Colorni A. M. Dorigo et V. Maniezzo. Distributed Optimization by Ant Colonies, actes de la première conférence européenne sur la vie artificielle. Paris, France: Elsevier Publishing, 1991. 134 c.
45. Lau K.M. Chan S.M. Xu L. Comparison of the Hopfield scheme to the hybrid of Lagrange and transformation approaches for solving the travelling salesman problem. Proceedings of Intelligence in Neural and Biological Systems, 1995.
46. R. Borndoerfer C. E. Ferreira A. Martin. Decomposing matrices into blocks. SIAM Journal on Optimization, 1998. 236 c.
47. Clochard J. M., Naddef D. Using path inequalities in a branch-and-cut code for the symmetric traveling salesman problem. Proceedings on the Third IPCO Conference, 1993. 291 c.
48. Naddef D. Polyhedral theory and branch-and-cut algorithms for the symmetric TSP. Kluwer, 2002.
49. Land A., Powell S. Computer codes for problems of integer programming. Annals of Discrete Mathematics 5, 1979. 221 c.
50. Linderoth J. T., Savelsbergh M. W. P. A computational study of

search strategies for mixed integer programming. INFORMS Journal on Computing, 1999. 173 c.

51. Fügenschuh A., Martin A. Computational integer programming and cutting planes. Elsevier, 2005. 736 c.
52. Dakin R. J. A tree search algorithm for mixed integer programs. Computer Journal, 1965. 250 c.
53. Dantzig G. B. Maximization of a linear function of variables subject to linear inequalities. New York: Activity Analysis of Production and Allocation, 1951. 339 c.
54. Gomory R. E. Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Society 64, 1958. 275 c.
55. Gomory R. E. An algorithm for integer solutions to linear programming. New York: Recent Advances in Mathematical Programming, 1963. 269 c.
56. E. Balas S. Ceria G. Cornuéjols, Natraj N. Gomory cuts revisited. Operations Research Letters, 1996.
57. Klar A. Cutting planes in mixed integer programming. Technische Universität Berlin: master's thesis, 2006.
58. Wolter K. Implementation of cutting plane separators for mixed integer program. Technische Universität Berlin: master's thesis, 2006.
59. H. Marchand A. Martin R. Weismantel, Wolsey L. A. Cutting planes in integer and mixed integer programming. Discrete Applied Mathematics, 2002. 391 c.
60. Nemhauser G. L., Wolsey L. A. A recursive procedure to generate all cuts for 0-1 mixed integer programs. Mathematical Programming, 1990. 379 c.
61. E. Balas S. Ceria, Cornuéjols G. A lift-and-project cutting plane algorithm for mixed 0-1 programs. Mathematical Programming, 1993. 295 c.
62. E. Balas S. Ceria, Cornuéjols G. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. Management Science, 1996. 1229 c.
63. Balas E. Facets of the knapsack polytope. Mathematical Programming 8, 1975. 146 c.
64. Balas E., Zemel E. Lifting and complementing yields all the facets of positive zero-one programming polytopes. Rio de Janeiro: Proceedings of

the International Conference on Mathematical Programming, 1981. 13 c.
65. Martin A., Weismantel R. The intersection of knapsack polyhedra and extensions. Proceedings of the 6th IPCO Conference, 1998. 243 c.
66. Wolsey L. A. Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. Discrete Applied Mathematics 29, 1990. 251 c.
67. Marchand H., Wolsey L. A. Aggregation and mixed integer rounding to solve MIPs. Operations Research 49, 2001. 363 c.
68. Chvátal V. Edmonds polytopes and a hierarchy of combinatorial problems. Discrete Mathematics, 1973. 305 c.
69. Letchford A. N., Lodi A. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. Operations Research Letters 30, 2002. 74 c.
70. M. W. Padberg T. J. van Roy, Wolsey L. A. Valid inequalities for fixed charge problems. Operations Research, 1985. 842 c.
71. van Roy T. J., Wolsey L. A. Valid inequalities for mixed 0-1 programs. Discrete Applied Mathematics, 1986. 199 c.
72. Mitchell John E. Branch-and-Cut Algorithms for Combinatorial Optimization Problems. Troy, NY, USA: Mathematical Sciences Rensselaer Polytechnic Institute, 1999. 3 c.
73. Achterberg Tobias. Constraint Integer Programming. Technische Universität Berlin: doktor's thesis, 2007. 112 c.
74. Н.Ю.Золотых В.Н.Шевченко и. Линейное и целочисленное линейное программирование. Нижний Новгород: Изд. Нижегородского госуниверситета им. Н.И.Лобачевского, 2004. 119 с.
75. corp. Nvidia. CUDA C Programming guide. CUDA SDK, 2017. 212 c.
76. С.А.Полетаев. Параллельные вычисления на графических процессорах. Новосибирск: Ин-т систем информатики имени А. П. Ершова СО РАН, 2008. 271 с.
77. Sanders J. Kandrot E. CUDA by Example An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010.
78. corp. NVIDIA. CUDA C Best Practices Guide. CUDA SDK, 2014.
79. David B. Kirk Wen-mei W. Hwu. Programming Massively Parallel Processors A Hands-on Approach. Morgan Kaufmanm, 2012.

80. Боресков А.В. Харламов А.А. Основы работы с технологией CUDA. ДМК Пресс, 2010.
81. corp. Nvidia. CUDA C Programming guide. CUDA SDK, 2017.
82. corp. NVIDIA. CUDA C Best Practices Guide. CUDA SDK, 2014. 34 с.