

МИНОБРНАУКИ РОССИИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ»

Факультет управления и прикладной математики

Кафедра системного программирования

Исследование возможностей распараллеливания алгоритмов
разрешения ограничений на GPU

Магистерская диссертация

Направление 010100 Математика

Магистерская программа Вещественный, комплексный и
функциональный анализ

Допущено к защите в ГЭК 27.05.2015

Зав.кафедрой _____ д.физ.-мат.н., проф. Е.М. Семёнов

Обучающийся _____ С.М. Петров

Руководитель _____ д.физ.-мат.н., проф. Т.Я. Азизов

Долгопрудный 2017

ОГЛАВЛЕНИЕ

	Стр.
ГЛАВА 1 Введение	4
1.1 Актуальность	4
1.2 Постановка задачи	6
1.3 Обзор существующих решений	8
ГЛАВА 2 Методика решения задачи	10
2.1 Branch-and-Bound алгоритм	10
2.2 Симплекс-метод	13
2.3 Метод отсечений	14
2.4 Допущения используемого метода	16
2.5 Описание возможных методов	17
ГЛАВА 3 Практическая реализация	19
3.1 Сравнение GPU и CPU	19
3.2 Технология CUDA	21
3.3 Иерархия памяти	26
3.4 Программная реализация	28
ГЛАВА 4 Анализ	33
4.1 Исследование корректности	33
4.2 Исследование эффективности	33
4.3 Подробное исследование эффективности выбранного ме- тода	36
4.4 Теоретический анализ выбранного метода	37
ГЛАВА 5 Результаты и выводы	39
5.1 Дальнейшее исследование	39
СПИСОК ЛИТЕРАТУРЫ	39

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) в том случае, когда эти ограничения выражены системой линейных неравенств. Комбинирование подходов из обеих областей относится к области целочисленного программирования в ограничениях. Так как многие важные задачи сводятся к задачам разрешения ограничений (англ. constraint satisfaction problem), им уделяется немало внимания [1], [2], [3].

Целочисленное и смешанное целочисленное программирование появилось в поздних 50-х и ранних 60-х, когда выяснилось, что модель задачи смешанного целочисленного программирования очень эффективна в применении к практическим задачам [4].

Впервые понятие общих ограничений было использовано Sutherland[5] в его проекте интерактивной системы Sketchpad, позволяющей рисовать с помощью компьютера. В 70-х это понятия появилось в контексте автоматизированной проверки теорем и обработки языка, в первую очередь языка логического программирования, разработанного Colmerauer[6][7] и Kowalski[8]. В 80-х задачи разрешения ограничений были отнесены к логическому программированию, что образовало логическое программирование в ограничениях (англ. constraint logic programming) [9][10][11].

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 $c^* = c^T x^*$.

В общем случае проблема существования решений задачи разрешения ограничений может быть алгоритмически неразрешимой или NP-полной.

Программирование в ограничениях широко использует методы целочисленного программирования. Задача существования целочисленного решения системы является NP-полной, поэтому требует больших вычислительных затрат. Таким образом для реальных задач необходимо проделывать очень большой объем вычислений, что не позволяет расширить довольно ограниченный круг задач, для решения которых используются известные методы целочисленного программирования. Актуальность и трудность проблематики делают целочисленное программирование одним из перспективных и интересных направлений в математическом программировании [12][13].

Использование подходов и алгоритмов теории разрешения ограничений позволяет решать многие прикладные задачи в разработке программного обеспечения, такие как планирование [14] и теория расписаний [15], задачи проектирования экспертных систем и систем поддержки принятия решений [16], обработка изображений [17], задачи тестирования интегральных схем [18], формальная верификация и статический анализ. В свою очередь статический анализ широко используется в верификации свойств ПО, которое требует высокой надёжности, например в области медицины, ядерной физики или авиации. Многие примеры применимости, включая перечисленные, можно найти в [19]

Среди подходов формальной верификации существуют методы построения тестов с помощью символического выполнения (symbolic execution). Такие методы используют символическое описание проходимого во время выполнения теста пути по коду программы (или формальных проверяемых спецификаций) 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 [20], [21].

В литературе можно найти несколько хороших обзоров с описанием методов решения задач разрешения ограничений [22] [23] [24] [25] [26], а также статьи в энциклопедических сборниках [27], [28] [29].

Более подробную информацию по теме разрешения ограничений можно найти в монографиях [30], [1], [31], [32], [33], [34], [35], [36], [37].

Линейные неравенства имеют большое самостоятельное значение, поскольку многие математические модели представляются в виде таких систем. Особый интерес линейные неравенства стали представлять в связи с созданием математической экономики [38] и линейного программирования в середине XX века. Многие модели экономики имеют вид систем линейных неравенств или задач линейного программирования, которые можно считать частным случаем систем линейных неравенств. Основная задача линейного программирования - это нахождение экстремумов линейных функций при множестве линейных ограничений. Нас интересует подраздел линейного программирования - целочисленное линейное программирование (integer linear programming, ILP), в котором на все или некоторые (смешанное целочисленное программирование) переменные дополнительно накладывается ограничение целочисленности.

Постановка задачи

Расчет сложных хаотических систем вроде погоды, симуляции взаимодействий элементарных частиц в физике, моделирование на нано-уровне, data mining, криптографии очень требовательны к аппаратным ресурсам. При выполнении данных операций используют параллельные вычисления, распределяя нагрузку между ресурсами CPU и процессорами компьютеров кластера.

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.

Основные методы смешанного целочисленного программирования:

- LP relaxation;
- метод отсечений;
- Branch-and-Bound.

Основные методы программирования в ограничениях:

- domain propagation;
- symmetry handling;
- Branch-and-Bound.

Основные способы ветвления:

- Ветвление по переменной, значение которой наиболее близко к целому значению или наиболее далеко от него:
 - стандартное правило, наиболее простое;
 - слабое в вычислительном отношении.
- Сильное ветвление:
 - 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:
 - попытка оценить стоимость ветвления, основываясь на истории ветвления по данной переменной;
 - эффективно и дёшево, но очень слабо в начале алгоритма.

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

- Обход дерева в глубину [39] всегда выбирает дочернюю подзадачу относительно нынешней.
- Выбор лучшей 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, что затрудняет разработку новых идей.

Ранее уже были исследованы рассматриваемые подходы к решению разного рода задач. К ним относятся задача планирования химической промышленной [41], задача балансировки линии сборки [42], задача параллельного машинного планирования [43].

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);
- графический метод (если задача содержит две переменные);
- эвристики:
 - табу-поиск - tabu-search [44];
 - алгоритм восхождение на вершину - hill climbing [45];
 - алгоритм имитации отжига [46];
 - оперативная оптимизация поиска;
 - муравьиный алгоритм [47];
 - нейронная сеть Хопфилда [48];

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

В данной работе рассматривается алгоритм Branch-and-Cut. Во-первых, он не подразумевает ограничения по входным матрицам.

Branch-and-Bound алгоритм

Branch-and-Cut - это метод комбинаторной оптимизации для решения целочисленных задач линейного программирования, то есть таких задач, решение которых целочисленно. Техника метода использует решение системы линейных неравенств без наложения ограничения целочисленности - релаксационных задач, которые имеют вид:

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

$$X_{LP} = \{x \in \mathbb{R}^n | Ax \leq b\}$$

- множество допустимых значений.

Когда получено оптимальное решение, которое содержит не целочисленное значение одной из целевых переменных, может добавляться отсечение в виде линейного неравенства, которое исключает данное нецелочисленное значение, но включает все возможные целочисленные. Таким образом мы усиливаем релаксационную задачу. Отсечение добавляется на основе результата решения предыдущей релаксационной задачи. Такой подход называется методом отсечений.

После этого решается новая релаксационная задача, строится новое отсечение и т.д., пока либо не получится оптимальное решение, либо не будет выявлена несовместность условий получившейся задачи линейного программирования.

Branch-and-Cut алгоритм включает в себя Branch-and-Bound с использованием отсечений, чтобы сузить область поиска решений. В свою очередь Branch-and-Bound основывается на симплекс-методе, который подробно обсуждается в следующем параграфе. Делим задачу на несколько (как правило, две) подзадач и применяем симплекс-метод к получившимся двум.

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

[init] $L = \{R\}$, $\hat{c} = \infty$;
 [abort] if $L =$, return $x^* = \hat{x}$, $c^* = \hat{c}$;
 [select] выбрать $Q \in L$, $L = L \setminus \{Q\}$;
 [solve] решить Q_{relax} - задачу Q без ограничения целочисленности. Если нет решения, то $\bar{c} = \infty$. Иначе 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 $Q = Q_1 \cup \dots \cup Q_k$, $L = L \cup \{Q_1 \cup \dots \cup Q_k\}$ и goto [abort].

На Рис. 2.1 приведён пример решения двумерной задачи методом Branch-and-Cut. Рис. 2.2 иллюстрирует, как происходит разбиение задачи в алгоритме на стадии [branch].

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 [49] [50] [51]

До сих пор остаётся открытым вопрос о стратегии ветвления: в какой последовательности генерировать дочерние вершины и какую из дробных координат выбирать для генерации неравенств. К сожалению, здесь мало теоретических результатов и для выбора хорошей стратегии опираются на вычислительный эксперимент. К хорошим результатам, например, приводит выбор для ветвления вершины с максимальным значением верхней оценки. А для выбора дробной координаты считается разумным выбор той из них, которая приводит к максимальному уменьшению верхней оценки целевой функции.

Опишем наиболее популярные методы ветвления:

- выбирается переменная наиболее близкая по значению к 0.5;

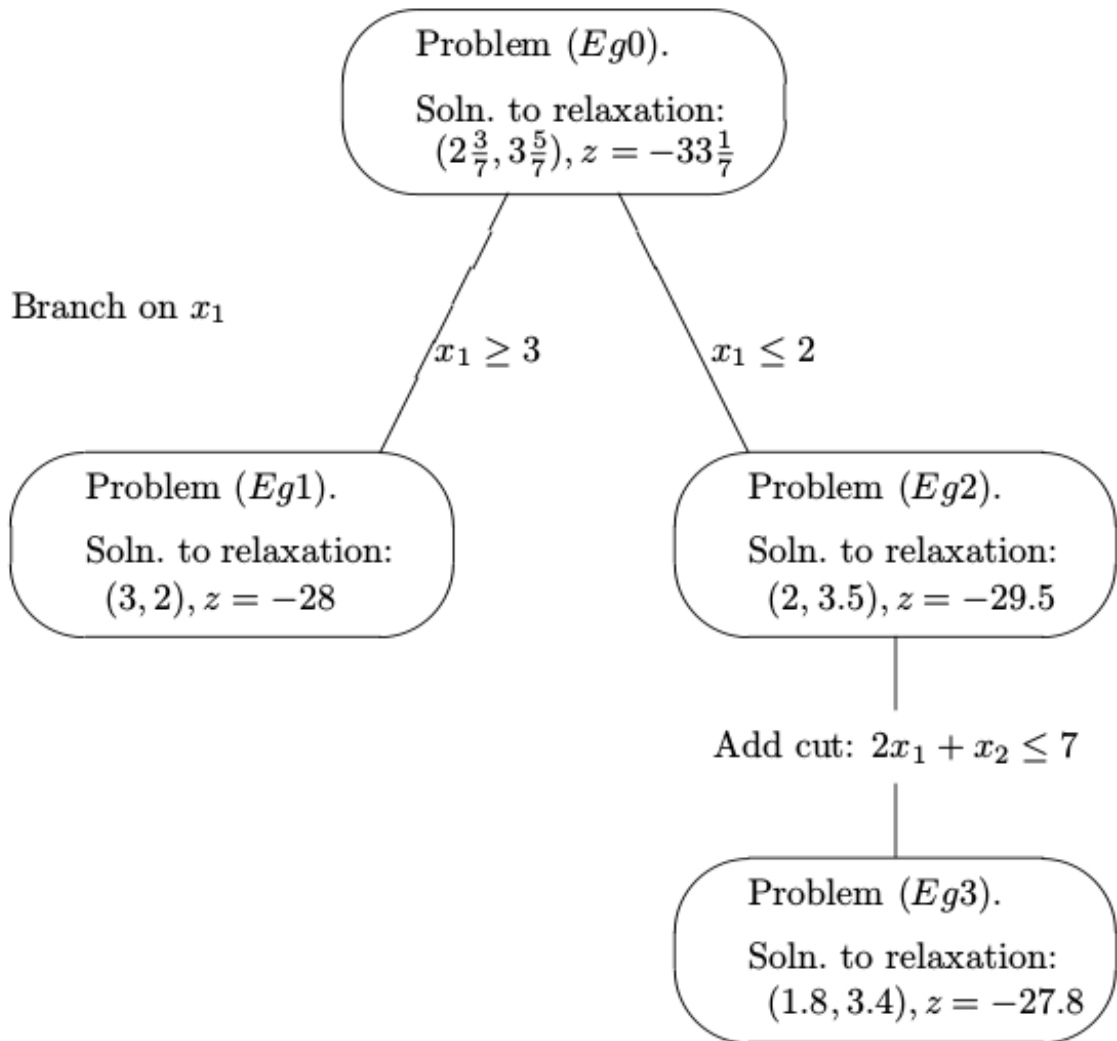


Рисунок 2.1 — Пример решения двумерной задачи методом Branch-and-Cut

- сохраняются треки каждой из переменных и выбирается та, изменение которой, возможно наибольшее, т.е. которая обладает наибольшей псевдостоимостью;
- каждая переменная проверяется на наибольшее влияние на целевую функцию, то при ветвлении по которой, функция изменится максимально.

Здесь приведены не все. Для более исчерпывающего исследования методов ветвления можно обратиться к [52] [53] [54].

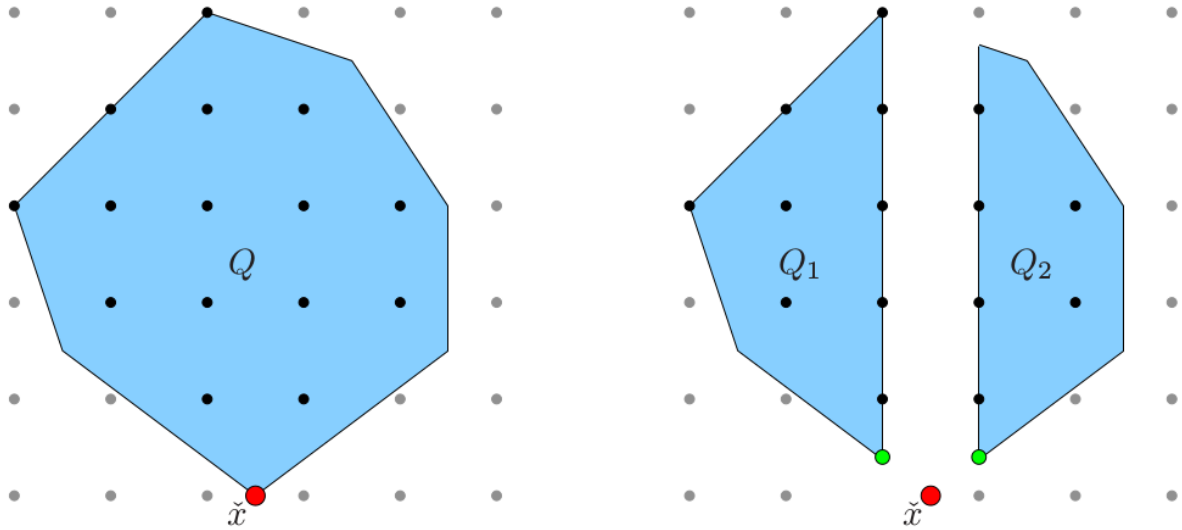


Рисунок 2.2 — Разбиение задачи Q на подзадачи $Q_1 \cup \dots \cup Q_k$

Симплекс-метод

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

Термин симплекс-метод был предложен Моцкиным и Данцигом[55]. Поясним происхождение этого термина.

Определение. Пусть $p_0, p_1, \dots, p_t \in \mathbb{R}^n$: векторы $q_1 = p_1 - p_0, q_2 = p_2 - p_0, \dots, q_t = p_t - p_0$ линейно независимы. Множество $\text{Conv} p_0, p_1, \dots, p_t$ называется t -мерным симплексом.

n -мерный симплекс можно описать как множество решений системы, состоящей из $n + 1$ неравенства. И наоборот, система неравенств определяет некоторый n -мерный симплекс. Таким образом, симплекс-метод можно описать как движение от одного симплекса к соседнему.

Определение. Вектор p лексикографически положителен $p \succ 0$, если его первая отличная от нуля компонента положительна. Вектор q лексикографически больше вектора p ($q \succ p$), если $q - p \succ 0$.

Прямой симплекс метод решает задачу:

$$c^T x \rightarrow \max, \quad Ax \leq b, \quad x \geq 0, \quad b \geq 0.$$

Так как оптимальное решение после правильного 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 [56].

Двойственный симплекс-метод решает задачу:

$$c^T x \rightarrow \max, \quad Ax \leq b, \quad x \geq 0, \quad c \geq 0.$$

Алгоритм состоит из трёх частей: поиск строки, поиск столбца, модернизация 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 наименьшую колонну из множества P - `pivot_col`.
3. Модернизация матрицы - применяем гауссовы преобразования таким образом, `pivot_row` обнулилась, кроме элемента, соответствующего `pivot_col`. После преобразований этот элемент должен быть равен -1.

Метод отсечений

Наибольший вклад в исследование этой области внёс Гомори [57] [58], который доказал что целочисленная задача может быть решена за конечное число шагов с помощью одного подхода отсечений без ветвления. К сожалению, предложенный им метод не был эффективным. Но спустя некоторое

время работа Balas [59] 1996 г. показала что метод отсечений может быть очень эффективным при соединении с методом Branch-and-Bound. Наиболее подробное описание методов можно найти в [60] [61] [62] [54].

Метод отсечений приближает решение к целочисленному. Пусть имеется оптимальное решение задачи линейного программирования, полученной из задачи целочисленного линейного программирования отбрасыванием требования целочисленности. Пусть также решение не является полностью целочисленным. Тогда добавим отсечение в виде линейного неравенства, которое исключает данное нецелочисленное значение, но включает все возможные целочисленные. После этого решается новая задача линейного программирования, строится новое отсечение и т.д., пока либо не получится оптимальное решение, либо не будет выявлена несовместность условий получившейся задачи линейного программирования.

В общем случае метод представляет собой систематическое введение дополнительных ограничений с целью сведения исходной допустимой области к выпуклой оболочке её допустимых целочисленных точек.

Метод отсечений для решения целочисленных задач а также способ построения отсечений, гарантирующий конечность процедуры, был впервые предложен Гомори в 1958 [58] и доказан им же в 1963[57]. К сожалению, предложенные им отсечения не были эффективными и медленно сходились, поэтому идея отсечений не рассматривалась много лет. Развитие теории многогранников привели к тому, что метод отсечений вновь привлёк внимание в 1980-х и широко используется в настоящее время. Наиболее известно применение branch-and-cut алгоритма в задаче коммивояжёра (traveling salesman problem). Такой подход оптимален в большинстве случаев по сравнению с другими методами.

Применение только отсечений не является эффективным методом решения оптимизационных задач. А branch-and-bound алгоритм может быть усовершенствован использованием отсечений, так как они существенно уменьшают размер получаемого дерева.

Отсечениями называют линейные неравенства, которые удовлетворяются целочисленными решениями задачи линейного программирования, но

могут нарушаться решениями общей задачи оптимизации.

Способы построения отсечений:

- mixed integer rounding cuts [63]
- gomory mixed integer [57]
- lift-and-project cuts [64] [65]
- lifted cover cuts [66][67][68]
- GUB cover cuts [69]
- complemented mixed integer rounding cuts [70]
- strong Chvátal-Gomory cuts [71] [72]
- flow cover cuts [73] [74]

Выбор соответствующего отсечения совершался на основе результатов вычислений, приведённых в [75]

Допущения используемого метода

Для этого используется алгоритм, который минимизирует функцию на дискретном множестве, описанном линейными неравенствами и состоящем из целых значений. Но для существующей задачи нет надобности минимизировать, поэтому алгоритм завершается, как только находится первое целочисленное решение. А целевая функция может быть любой.

Изначально данный алгоритм предназначался для нахождения оптимального решения и поэтому находятся все целочисленные решения. Для целевой задачи дипломной работы достаточно нахождение одного целочисленного решения и поэтому приведённый ниже алгоритм не является полным.

Как известно различные процессоры могут по-разному производить арифметические операции над переменными типа float или double. Такая же ситуация состоит и с разницей между GPU и CPU. Также необходимо учесть, что математика с использованием переменных с плавающей точкой не является сочетательной. Т.е. может быть так, что $(A + B) + C \neq A + (B + C)$. При распараллеливании, потенциально меняется порядок вычислений. В связи со всеми этими особенностями необходимо дополнитель-

но проверять сходятся ли результаты различных реализаций на GPU и CPU.

Описание возможных методов

Так как симплекс-метод играет ключевую роль в алгоритме Branch-and-Cut, начнём возможные реализации алгоритма именно с этой части.

При программной реализации алгоритмов линейного программирования одной из возникающих проблем является накопление ошибок округления.

После этого приступаем к алгоритму симплекса на GPU. Первый вопрос, который возникает при подходе к симплекс-методу на GPU, - где хранить матрицу и каких размеров она может быть.

Мы можем хранить матрицу в памяти графической карты, тем самым избегая постоянного копирования, но тогда поиск `pivot_row` и `pivot_col` проводя также на GPU. Это недостаток, так как такие задачи плохо распараллеливаются на GPU.

И как ещё один вариант, мы храним матрицу в памяти центрального процессора и каждый раз копируем из памяти CPU в GPU и наоборот. Графические карты поддерживают как синхронное, так и асинхронное копирование. Основным достоинством асинхронного копирования является то, что оно позволяет параллельное исполнение ядра (kernel), то есть мы имеем возможность обрабатывать уже скопированные данные во время копирования остальных данных. Но такое копирование медленнее синхронного и может оперировать только с закреплённой памятью.

В итоге мы имеем три возможные идеи, как реализовать параллельный алгоритм. Каждая идея имеет как свои плюсы, так и недостатки, поэтому в работе я реализовала все три алгоритма. Обобщим теоретические рассуждения в двух таблицах, содержащих достоинства каждого из подходов:

алгоритм с абсолютной реализацией на GPU:	алгоритм с частичной реализацией на GPU:
---	--

<ul style="list-style-type: none"> · нет необходимости копировать большие объёмы данных из памяти CPU в память GPU и наоборот 	<ul style="list-style-type: none"> · плохо распараллеливается на GPU функция поиска <code>pivot_row</code> и <code>pivot_col</code>, так как требуют лишних затрат для синхронизации алгоритма
использования синхронного копирования	использование асинхронного копирования
<ul style="list-style-type: none"> · быстрее · сильно выигрывает при копировании маленьких объёмов памяти 	<ul style="list-style-type: none"> · параллельная обработка уже скопированных данных

При прочих равных условиях преобразования таблицы, количество итераций, совершаемых действий и значения ячеек внутри таблицы на каждой итерации не зависят от реализации алгоритма.

ГЛАВА 3

Практическая реализация

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 в отличие физической памяти.

CPU поддерживает выполнение сильно ограниченного числа совместно выполняющихся нитей. Серверы, содержащие 4 шестиядерных процессора, могут параллельно исполнять лишь 24 нити параллельно (или 48 если ЦПУ поддерживает 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 [76]. Если видеокарта содержит 16 мультипроцессоров, то число может превышать 24 000.

В видеочипах NVIDIA, например, основной блок – это мультипроцессор с восемью-десятью ядрами и сотнями ALU в целом, несколькими тысячами регистров и небольшим количеством разделяемой общей памяти. Кроме того, видеокарта содержит быструю глобальную память с доступом к ней всех мультипроцессоров, локальную память в каждом мультипроцессоре, а также специальную память для констант (Рис. 3.1).

Многопоточность в графических процессорах также реализована и на аппаратном уровне. На CPU же её задействовать не целесообразно, так как 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 проис-

ходит 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.

Ядра CPU созданы для исполнения одно потока последовательных инструкций с максимальной производительностью, в то время как GPU проектируются для большого числа параллельно выполняемых инструкций. Видеоочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели.

Это отражается и в различии архитектур ядер. 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.

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

В универсальных 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. Видеоочипы тратят транзисторы на массивы исполнительных блоков, управляющие потоками блоков, 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. [77]

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 про-

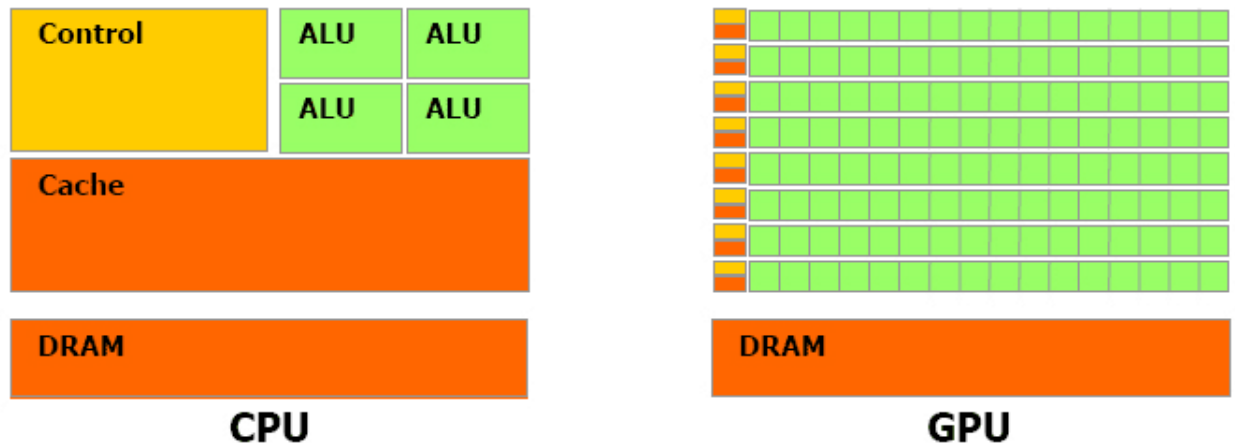


Рисунок 3.1 — Сравнение GPU и CPU

цессор 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

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. [78] [79] [80] [81]

CUDA (Compute Unified Device Architecture) – это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров). Для реализации новой вычислительной парадиг-

мы компания NVIDIA изобрела архитектуру параллельных вычислений CUDA, и обеспечивающую необходимую базу разработчикам ПО.

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 наряду с доступностью.

Компания NVIDIA предоставляет показательные примеры кода на CUDA [79] в свободной доступе на английском языке и в продаже на русском, а также две платформы Nvidia Nsight Edition для разработчиков в Eclipse и Microsoft Visual Studio.

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 функции уровня Runtime состоит из более элементар-

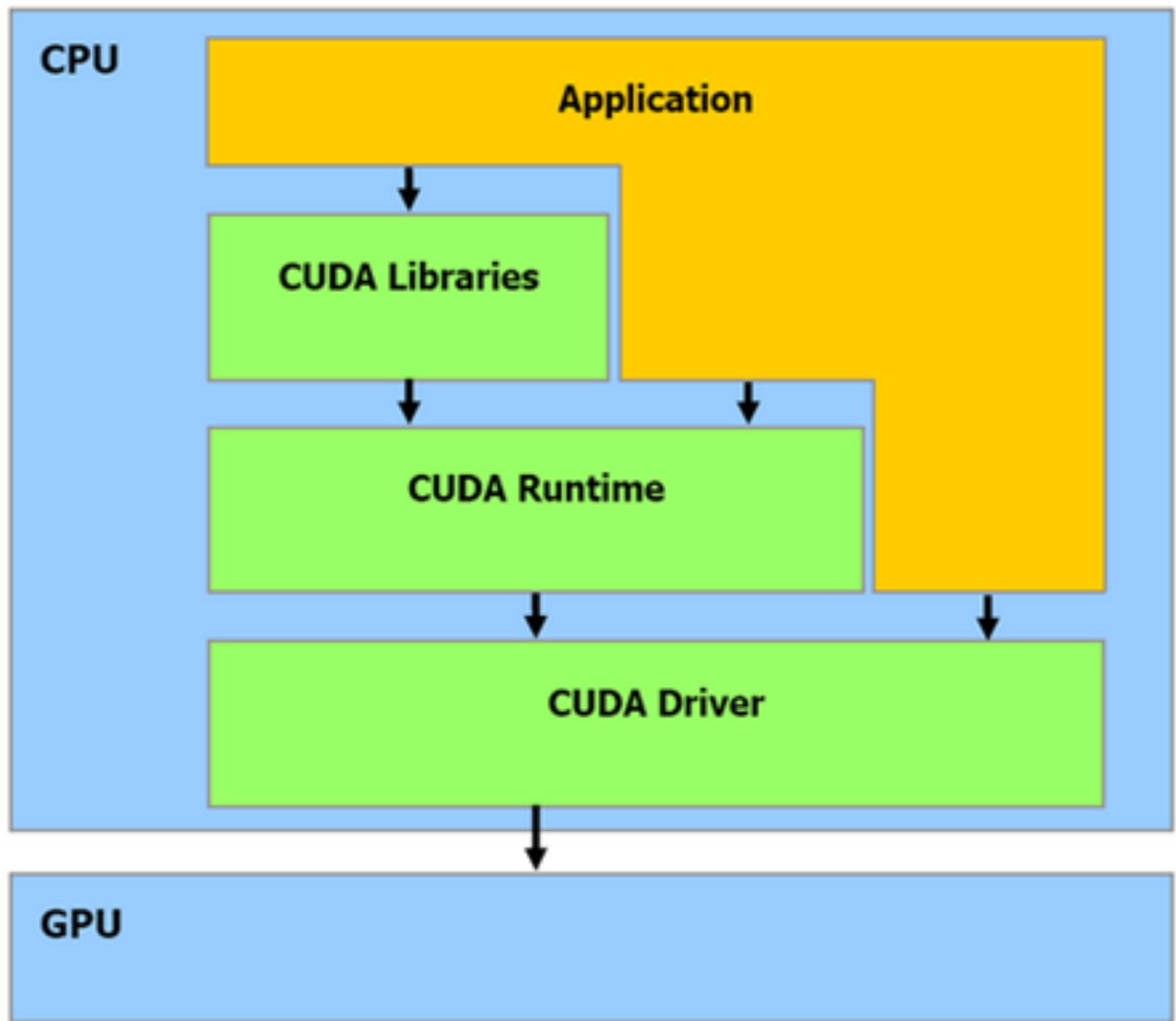


Рисунок 3.2 — Набор программных уровней для работы с GPU

ных 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 называется 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, обрабатываемых SIMD-способом в мультипроцессорах CUDA. Вследствие этого, даже при ветвлении внутри программы все нити внутри одного warp'a ис-

полняют 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 копирования 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 score 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.

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.

Потоковые мультипроцессоры работают следующим образом: каждый такт начало конвейера выбирает warp, 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 толь-

ко 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. Видеоочип просто принимает уже скомпилированное ядро и создает его копии для каждого элемента данных. Каждое из ядер выполняется в своем собственном потоке.

Иерархия памяти

Вычислительные возможности 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 (чтение только) - кэш, оптимизированный для двумерного доступа.
 - более удобный метод доступа с геометрической точки зрения;
 - неизменяемая;
 - кэшируемая.
- Константная память:
 - содержит константы и аргументы функций ядра;
 - имеет особые 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(по 32 штуки?). 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 (shared memory). РРазделяемая память устроена так,

что доступная блоку память разбита на банки памяти с отдельными 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 [79] и [78].

Обозначенное угловыми скобками ядро также принимает необязательный аргумент - номер потока исполнения (англ. *stream*). Этот запуск ядра выполняется асинхронно и используется с функциями асинхронного копирования памяти. Они выполняются параллельно друг другу и параллельно коду CPU. Поэтому не исключено, что последующий код CPU начнёт выполняться ещё до того, как закончится копирование памяти или исполнение ядра. Для этого в CUDA существуют функции синхронизации. 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 [82] и найти примеры в [79]

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 време-

ни.

%	cumul.	self		self	total	
time	seconds	seconds	calls	s/call	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
%	cumul.	self		self	total	
time	seconds	seconds	calls	Ks/call	Ks/call	name
84.66	1099.81	1099.81	23063	0.00	0.00	multip
15.17	1296.83	197.02	293997	0.00	0.00	dualSimplex
0.13	1298.58	1.75	23373	0.00	0.00	Matrix::Matrix(int, int, int)
0.07	1299.50	0.92	293999	0.00	0.00	pivotColumn
0.02	1299.77	0.27	197223184	0.00	0.00	cmp
0.01	1299.90	0.13	308	0.00	0.00	Matrix::operator=

Как видно из таблицы, в первую очередь необходимо оптимизировать произведение матриц. Можно как реализовать собственную функцию произведения матриц, так и воспользоваться 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.

Реализованный алгоритм:

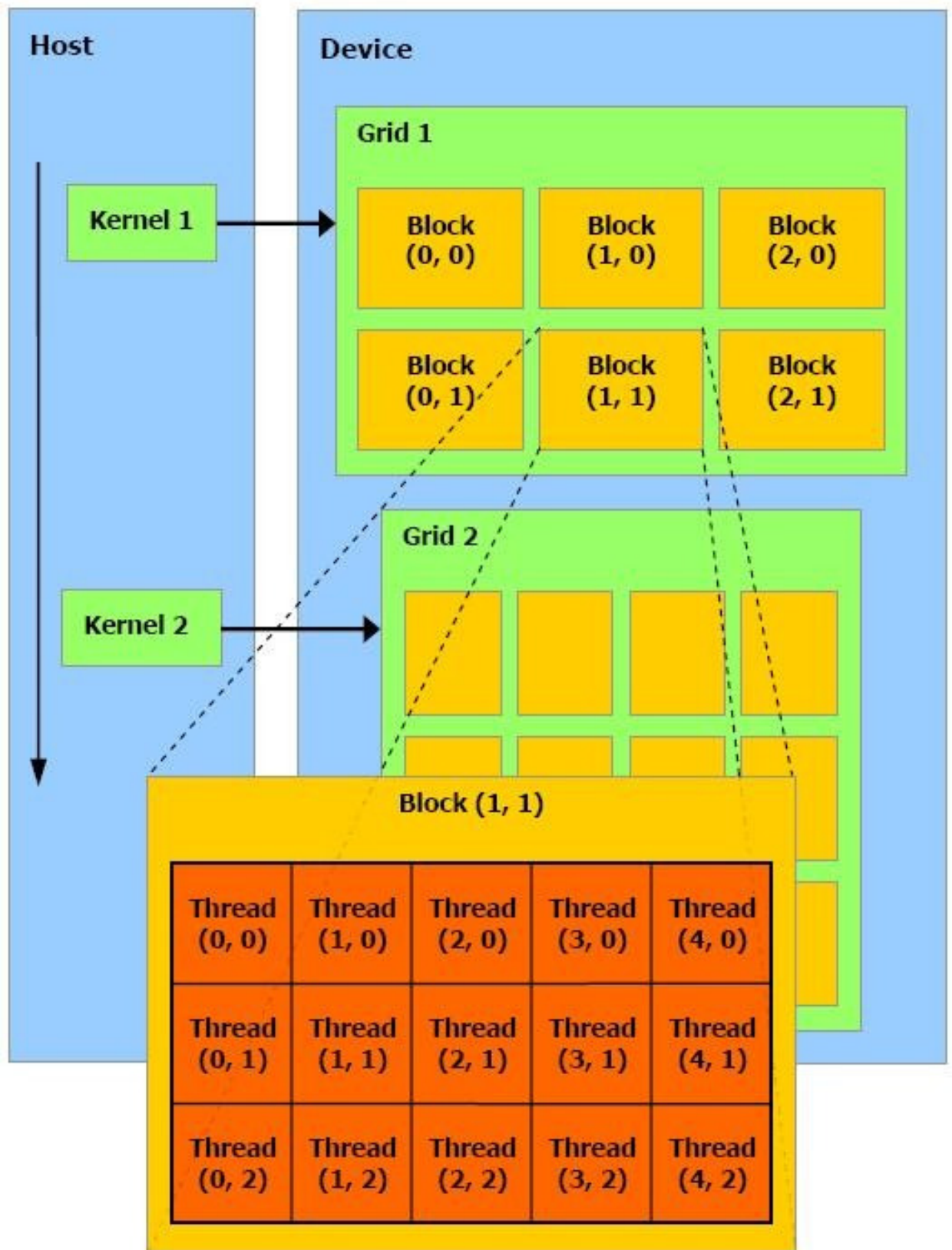


Рисунок 3.3 — Конструкция grid

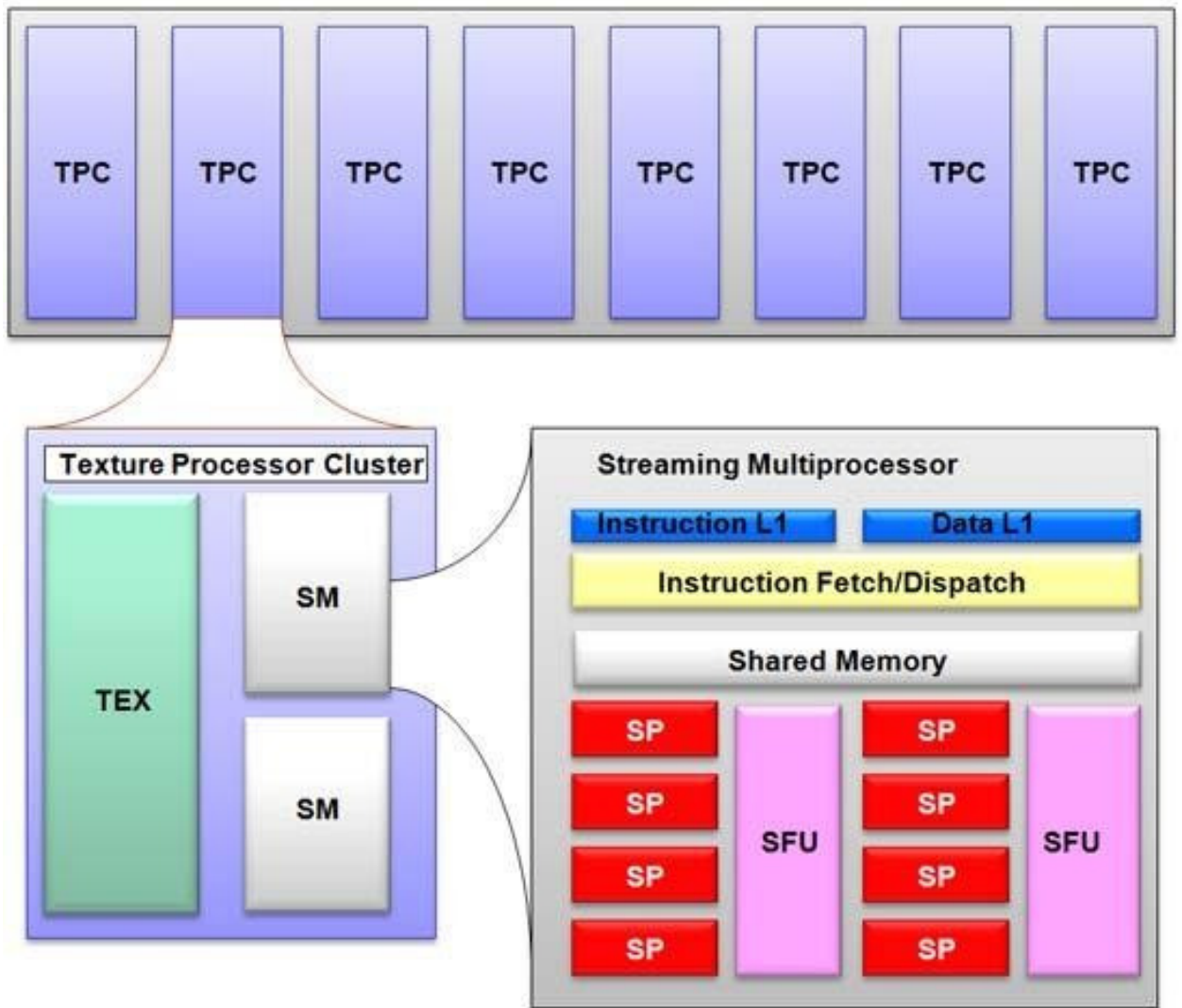


Рисунок 3.4 — Конструкция TPC и SM

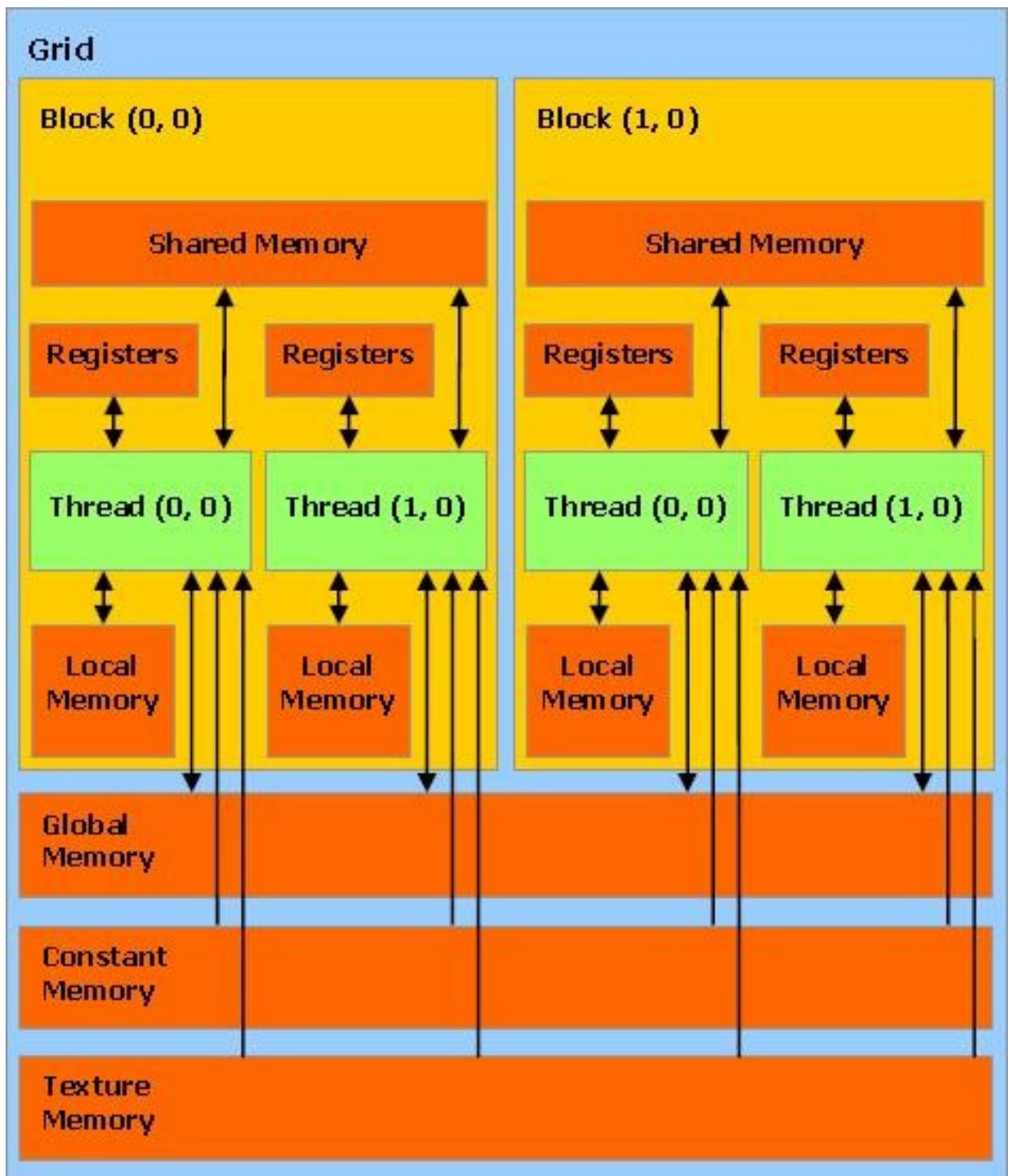


Рисунок 3.5 — Иерархия памяти CUDA

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 модификации эффективны с теоретической точки зрения, 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:

$$\begin{aligned} & \max x_1 \\ & \left\{ \begin{array}{l} (2^t + 1)x_1 = 2^t x_2, \\ 0 \leq x_2 \leq 2^t, \\ x_1, x_2 \in \mathbb{Z}. \end{array} \right. \end{aligned}$$

Для установления несовместности условий задачи целочисленного линейного программирования

$$\begin{aligned} & \max(x_1 + x_2 + \dots + x_n) \\ & \left\{ \begin{array}{l} 2x_1 + 2x_2 + \dots + 2x_n = n \\ x_j \in \mathbb{Z}, (j = 1, 2, \dots, n) \end{array} \right. \end{aligned}$$

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:

test name	rows	cols	iters	CPU, ms	dev, ms	async, ms	sync, ms
uf20-01.cnf	132	21	27	5	90	6	9
uf20-03.cnf	132	21	60	10	105	8	9
uf20-09.cnf	132	21	77	16	131	11	14
hole6.cnf	190	43	22	4	98	6	6
uuf50-01.cnf	304	51	456	327	566	74	285
uuf50-03.cnf	319	51	172	94	239	31	98
uf50-04.cnf	319	51	207	125	282	38	122
uuf50-02.cnf	319	51	265	180	374	52	167
uf50-01.cnf	319	51	317	214	407	53	191
uf50-05.cnf	319	51	465	360	643	80	338
hole8.cnf	368	73	29	17	103	12	20
uuf75-01.cnf	450	76	35	1037	1203	208	960
uf75-08.cnf	467	6	1181	1861	1912	342	1720
uuf75-02.cnf	469	76	909	1606	1592	276	1505
uuf75-04.cnf	470	76	1354	2264	2239	401	2106
uf75-01.cnf	476	76	511	773	805	151	729
uf75-04.cnf	476	76	782	1429	1443	242	1328
uf75-02.cnf	476	76	911	1880	1803	285	1620
flat30-100.cnf	481	91	61	55	121	22	54
BMS_k3_n100_m4...	487	101	908	1642	1526	315	1384
hole9.cnf	541	91	45	49	177	20	49
CBS_k3_n100_m4...	612	101	1784	5062	5285	807	4689
RTI_k3_n100_m42...	619	101	1628	4911	4860	738	4568
uuf100-01.cnf	622	101	1553	4509	4305	706	4183
uf100-01.cnf	631	101	2055	6233	5968	934	5756
CBS_k3_n100_m4...	650	101	1150	2855	2681	524	2621
hole10.cnf	626	111	44	65	171	22	61
uuf125-01.cnf	763	126	3275	14425	11439	1985	13429
uf125-01.cnf	789	126	3879	19345	17822	2879	18399

flat50-100.cnf	846	151	98	253	218	74	238
uuf150-01.cnf	913	151	3320	20960	17169	2902	20192
uuf175-01.cnf	1002	176	3235	21859	16009	3040	20210

Тесты в таблице отсортированы по количеству ячеек в матрице, т.е. $rows * cols$. Значения в колонке таковы:

- имя тестового файла;
- количество строк в матрице, т.е. размер колонки;
- количество колонок в матрице, т.е. размер строки;
- количество итераций в алгоритме симплекс-метода (одинаковое для всех реализаций);
- время, потраченное на симплекс-метод, реализованный абсолютно на CPU;
- время, потраченное на симплекс-метод, реализованный абсолютно на GPU;
- время, потраченное на симплекс-метод, реализованный частично на GPU и CPU при использовании асинхронного копирования из одной памяти в другую;
- время, потраченное на симплекс-метод, реализованный частично на GPU и CPU при использовании синхронного копирования из одной памяти в другую;

Исходя из полученных данных, считаю целесообразным из всех реализаций симплекс-метода при помощи GPU, использовать подход, частично использующий GPU и CPU и реализованный с использованием асинхронного копирования. Далее уделим внимание этой реализации алгоритма.

Подробное исследование эффективности выбранного 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.

Теоретический анализ выбранного 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 объём 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.

Дальнейшее исследование

СПИСОК ЛИТЕРАТУРЫ

1. R. Dechter. Constraint processing. San Francisco: Morgan Kaufmann, 2003. 481 с.
2. Freuder E. C. Mackworth A. K. Constraint satisfaction: An emerging paradigm. Foundations of Artificial Intelligence, 2006. 13 с.
3. E. Tsang. Foundations of Constraint Satisfaction. New York: Academic Press, 1993. 421 с.
4. Markowitz H. M., Manne A. S. On the solution of discrete programming problems. Econometrica 25, 195. 84 с.
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 с.
7. A. Colmerauer H. Kanoui R. Pasero, Roussel P. Un système de communication en français. Groupe Intelligence Artificielle, Université AixMarseille II, France, 1972.
8. Kowalski R. A. Predicate logic as programming language. Stockholm, Sweden: Sixth IFIP Congress, 1974. 569 с.

9. Jaffar J., Lassez J.-L. Constraint logic programming. Munich: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, 1987. 111 с.
10. 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.
11. Colmerauer A. An introduction to Prolog III. Communications of the ACM 33, 1990. 69 с.
12. Г. Карманов В. Математическое программирование. Москва: Наука, 1986. 288 с.
13. L. Balinski M. Integer Programming: Methods, Uses, Computations. Management Science, 1965. 253 с.
14. Kautz H. Selman B. Planning as Satisfiability. Chichester: John Wiley and Sons, 1992. 359 с.
15. Barnier N. Brisset P. Graph coloring for air traffic flow management. Annals of Operations Research, 2004. 163 с.
16. Сараев А.Д. Щербина О.А. Системный анализ и современные информационные технологии // Труды Крымской академии наук. Симферополь: СОНАТ, 2006. 47 с.
17. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences, 1974. 95 с.
18. Hooker J.N. Yan H. Verifying logic circuits by Benders decomposition // Principles and Practice of Constraint Programming. Cambridge: MIT Press, 1994.
19. Heipcke S. Applications of Optimization with Xpress-MP. Blisworth, U.K.: Dash Optimization, 2002.
20. A. Gotlieb B. Botella M. Rueher. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes, 1998. 53 с.
21. 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.
22. V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 1992. 32 c.
 23. Dechter R. Frost D. Backtracking algorithms for constraint satisfaction problems. University of California: Department of Information and Computer Science, 1999.
 24. R. Bartak. Theory and practice of constraint propagation // *Proceedings of the Third Workshop on Constraint Programming for Decision and Control (CPDC-01)*. Poland: Gliwice, 2001. 7 c.
 25. P. Meseguer. Constraint satisfaction problems: an overview. *AI Communications*, 1989. 3 c.
 26. Miguel I. Shen Q. Solution techniques for constraint satisfaction problems: advanced approaches. *Artificial Intelligence Review*, 2001. 267 c.
 27. R. Dechter. Constraint networks // *Encyclopedia of Artificial Intelligence* (2nd edition). New York: Wiley and Sons, 1992. 276 c.
 28. W. Hower. Constraint satisfaction. Algorithms and complexity analysis. *Information Processing Letters*, 1995. 171 c.
 29. A.K. Mackworth. Constraint satisfaction // *Encyclopedia of Artificial Intelligence* (second edition). New York: Wiley, 1992. 285 c.
 30. R. Apt K. Principles of Constraint Programming. New York: Cambridge University Press, 2003. 407 c.
 31. Fruehwirth T. Abdennadher S. *Essentials of Constraint Programming*. Springer, 2003. 144 c.
 32. Marriott K. Stuckey P. J. *Programming with Constraints: An Introduction*. Cambridge: MIT Press, 1998. 483 c.
 33. Rossi F. van Beek P. Walsh T. *Handbook Of Constraint Programming*. Elsevier, 2006. 978 c.
 34. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Cambridge: MIT Press, 1989. 224 c.
 35. Van Hentenryck P. Michel L. Deville Y. *Numerica: A Modeling Language for Global Optimization*. Cambridge: MIT Press, 1997. 210 c.
 36. P. Van Hentenryck. *The OPL Optimization Programming Language*.

- Cambridge: MIT Press, 1999. 255 с.
37. Van Hentenryck P. Michel L. Constraint-Based Local Search. Cambridge: MIT Press, 2005. 442 с.
 38. Канторович Л. В. Математические методы организации и планирования производства. Санкт-Петербург: изд. ЛГУ, 1939.
 39. Dakin R. J. A tree search algorithm for mixed integer programs. *Computer Journal*, 1965. 250 с.
 40. С. Макконнелл. Совершенный код. Русская Редакция, Microsoft Press, 2010.
 41. Timpe C. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum* 24, 2002. 431 с.
 42. Bockmayr A., Pizaruk N. Solving assembly line balancing problems by combining IP and CP. *Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.
 43. Jain V., Grossmann I. E. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing* 13, 2001. 258 с.
 44. F. Glover. Tabu search-Part II. *ORSA Journal on computing*, 1989.
 45. Russell St. J. Norvig P. *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, 2003. 111 с.
 46. Kirkpatrick S. Gelatt Jr C. D. Vecchi M. P. Optimization by Simulated Annealing. *Science*, 1983. 671 с.
 47. Colomi 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 с.
 48. 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.
 49. R. Borndorfer C. E. Ferreira A. Martin. Decomposing matrices into blocks. *SIAM Journal on Optimization*, 1998. 236 с.
 50. 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.
51. Naddef D. Polyhedral theory and branch-and-cut algorithms for the symmetric TSP. Kluwer, 2002.
 52. Land A., Powell S. Computer codes for problems of integer programming. *Annals of Discrete Mathematics* 5, 1979. 221 c.
 53. Linderoth J. T., Savelsbergh M. W. P. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 1999. 173 c.
 54. Fügenschuh A., Martin A. Computational integer programming and cutting planes. Elsevier, 2005. 736 c.
 55. 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.
 56. Н.Ю.Золотых В.Н.Шевченко и. Линейное и целочисленное линейное программирование. Нижний Новгород: Изд. Нижегородского госуниверситета им. Н.И.Лобачевского, 2004. 119 c.
 57. Gomory R. E. An algorithm for integer solutions to linear programming. New York: Recent Advances in Mathematical Programming, 1963. 269 c.
 58. Gomory R. E. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Society* 64, 1958. 275 c.
 59. E. Balas S. Ceria G. Cornuéjols, Natraj N. Gomory cuts revisited. *Operations Research Letters*, 1996.
 60. Klar A. Cutting planes in mixed integer programming. Technische Universität Berlin: master's thesis, 2006.
 61. Wolter K. Implementation of cutting plane separators for mixed integer program. Technische Universität Berlin: master's thesis, 2006.
 62. H. Marchand A. Martin R. Weismantel, Wolsey L. A. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 2002. 391 c.
 63. Nemhauser G. L., Wolsey L. A. A recursive procedure to generate all cuts for 0-1 mixed integer programs. *Mathematical Programming*, 1990. 379 c.

64. E. Balas S. Ceria, Cornuéjols G. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 1993. 295 c.
65. 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.
66. Balas E. Facets of the knapsack polytope. *Mathematical Programming* 8, 1975. 146 c.
67. 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.
68. Martin A., Weismantel R. The intersection of knapsack polyhedra and extensions. Proceedings of the 6th IPCO Conference, 1998. 243 c.
69. Wolsey L. A. Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. *Discrete Applied Mathematics* 29, 1990. 251 c.
70. Marchand H., Wolsey L. A. Aggregation and mixed integer rounding to solve MIPs. *Operations Research* 49, 2001. 363 c.
71. Chvátal V. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 1973. 305 c.
72. Letchford A. N., Lodi A. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters* 30, 2002. 74 c.
73. M. W. Padberg T. J. van Roy, Wolsey L. A. Valid inequalities for fixed charge problems. *Operations Research*, 1985. 842 c.
74. van Roy T. J., Wolsey L. A. Valid inequalities for mixed 0-1 programs. *Discrete Applied Mathematics*, 1986. 199 c.
75. Achterberg Tobias. Constraint Integer Programming. Technische Universität Berlin: doktor's thesis, 2007. 112 c.
76. corp. Nvidia. CUDA C Programming guide. CUDA SDK, 2017. 212 c.
77. С.А.Полетаев. Параллельные вычисления на графических процессорах. Новосибирск: Ин-т систем информатики имени А. П. Ершова СО РАН, 2008. 271 c.
78. corp. NVIDIA. CUDA C Best Practices Guide. CUDA SDK, 2014.
79. Sanders J. Kandrot E. CUDA by Example An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010.

80. David B. Kirk Wen-mei W. Hwu. Programming Massively Parallel Processors A Hands-on Approach. Morgan Kaufmann, 2012.
81. Боресков А.В. Харламов А.А. Основы работы с технологией CUDA. ДМК Пресс, 2010.
82. corp. Nvidia. CUDA C Programming guide. CUDA SDK, 2017.