МИНОБРНАУКИ РОССИИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНТСИТУТ»

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 ... 7

ГЛАВА 2 Методика решения задачи 9
2.1 Branch-and-Cut алгоритм ... 9
2.2 Симплекс-метод ... 11
2.3 Метод отсечений ... 11
2.4 Допущения используемого метода ... 13
2.5 Описание возможных методов ... 13

ГЛАВА 3 Практическая реализация 16
3.1 Сравнение GPU и CPU ... 16
3.2 Технология CUDA ... 18
3.3 Иерархия памяти ... 23
3.4 Программная реализация ... 25
3.4.1 Классы и методы ... 25
3.4.2 Произведение матриц ... 27
3.4.3 ... 28

ГЛАВА 4 Анализ 32
4.1 Исследование корректности ... 32
4.2 Исследование эффективности ... 32
4.3 Подробное исследование эффективности выбранного метода ... 35
4.4 Теоретический анализ выбранного метода ... 36

ГЛАВА 5 Результаты и выводы 37
5.1 Дальнейшее исследование ... 37
СПИСОК ЛИТЕРАТУРЫ ... 37

ГЛАВА 1

Введение

Актуальность

Данная работа является приклакладным исследованием в области целочисленного программирования, которая относится к более широкой области задач разрешения ограничений. Так как многие важные задачи сводятся к задачам разрешения ограничений (constraint satisfaction problem), им уделяется немало внимания [1], [2], [3]. Суть таких задач состоит в нахождении значений переменных, удовлетворяющих заданным ограничениям.

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 [4][5].

Использование подходов и алгоритмов теории разрешения ограничений позволяет решать многие прикладные задачи в разработке программного обеспечения, такие как планирование [6] и теория расписаний [7], задачи проектирования экспертных систем и систем поддержки принятия решений [8], обработка изображений [9], задачи тестирований интегральных схем [10], 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.

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 [11], [12].

В литературе можно найти несколько хороших обзоров с описанием методов решения задач разрешения ограничений, включая (Kumar, 1992) citekumar, (Dechter, Frost, 1999) [13], (Bartak, 2001) [14], [15], [16], а также статьи в энциклопедических сборниках (Dechter, 1992) [17], [18] и (Mackworth, 1992) [19].

Более подробную информацию по теме разрешения ограничений можно найти в монографиях [20], [1], [21], [22], [23], [24], [25], [26], [27].

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 [28] 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 фукнций при множестве линейных ограничений. Нас интересует подраздел линейного программирования - целочисленное линейное программирование (integer linear programming, ILP), в котором на все или некоторые переменные дополнительно накладывается ограничение целочисленности. Простейший метод решения задачи целочисленного программирования — сведение её к задаче линейного программирования с проверкой результата на целочисленность.

Задача линейного программирования может быть записана в виде:

$$\max \vec{c}^T \vec{x}$$

$$A\vec{x} \leq \vec{b}$$

, где $\vec{c}, \vec{x} \in \mathbb{R}^n, \vec{b} \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$

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.

В работе подробно рассматривается один из способов решения системы линейных неравенств общего вида Branch-and-Cut. А точнее находится случайное целочисленное решение системы или констатируется, что целочисленных решений системы не существует.

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 [29]:

- 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

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);

- использование абсолютной унимодулярности матрицы (сильно ограничивает множество решаемых систем уравнений);
- графический метод (если задача содержит две переменные);
- эвристики:
  - табу-поиск - tabu-search [30];
  - алгоритм восхождение на вершину - hill climbing [31];
  - алгоритм имитации отжига [32];
  - оперативная оптимизация поиска;
  - муравьиный алгоритм [33];
  - нейронная сеть Хопфилда [34];

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-Cut 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. 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. 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.

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.

1. Добавить исходную задачу в множество L - множество активных задач.
2. Инициализировать $x^* =$ и $v^* = -\infty$
3. Пока список L не пуст и не найдено целочисленное решение:
   а) выбираем проблемы и исключаем её из списка L;
   б) решаем выбранную проблему симплекс методом;
   в) если решения нет, то возвращаемся к 3., обозначим значение

функции v;

г) если $v \leq v^*$, то возвращаемся к 3.;

д) если решение целочисленное, то $v^* v$, $x^* x$ и возвращаемся к 3.;

е) по желанию добавляем отсечения или отсечение;

;) ветвимся, создаём на основе решения предыдущей задачи новые(как правило, две) и возвращаемся к 3.;

3. return найденное решение.

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 опираютс яна интуицию и 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.

Ветвиться можно несколькими способами. Некоторые из них реализуют ветвление по переменной, т.е. выбирается переменная x_i с нецелочисленным значением a_i и создаются две задачи, каждая из которых является добавлением к предыдущей ограничения $x_i \leqslant \lfloor a_i \rfloor$ и $x_i \geqslant \lceil a_i \rceil$. Например:

- выбирается переменная наиболее близкая по значению к 0.5;
- сохраняются треки каждой из переменных и выбирается та, изменение которой, возможно наибольшее;
- каждая переменная проверяется на наибольшее влияние на целевую функцию, то при ветвлении по которой, фукнция изменится максимально.

Многие задачи комбинаторной оптимизации могут быть представлены как смешанные целочисленные задачи линейного программирования. В этом случае применимы branch-and-cut методы, которые по сути есть комбинация метода отсечения и branch-and-bound алгоритма. Эти методы используют решения некоторой последовательности упрощённых целочиселнных задач. Метод отсечений приближает решение к целочисленному,

a branch-and-bound делит задачу на несколько других.

Также branch-and-cut используется для решения задачи линейного упорядочивания (linear ordering problem), максимального отсечения (maximum cut), планирования (scheduling), о рюкзаке (packing), нахождения максимального планарного подграфа и др.

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. Поясним происхождение этого термина.

Определение. Пусть $p_0, p_1, ..., p_t \in \mathbb{R}^n$: векторы $q_1 = p_1 - p_o, q_2 = p_2 - p_0, ..., q_t = p_t - p_0$ линейно независимы. Множество Conv$p_o, p_1, ..., p_t$ 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 лексикограически больше вектора p ($q \succ p$), если $q - p \succ 0$.

Метод отсечений

Пусть имеется оптимальное решение задачи линейного программирования, полученной из задачи цеолчисленного 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 сиситематическое введение дополнительных ограничений с целью сведения исходной допустимой области к выпуклой оболочке её одпустимых целочисленных точек.

Метод отсечений для решения целочисленных задач а также способ построеня отсечений, гарантирующий конечность процедуры, был впервые предложен Гомори (1963, опубликован в 1958). 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.

Так как оптимальное решение после правильного отсечения не является допустимым решением новой задачи линоейного 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. (с 119, В.Н.Шевченко и Н.Ю.Золотых).

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.

1. Добавить исходную задачу в множество L - множество активных задач.
2. Пока список L не пуст и не найдено целочисленное решение:
   а) выбираем проблемы и исключаем её из списка L;
   б) решаем выбранную проблему симплекс методом;
   в) если решения нет, то возвращаемся к пункту а), иначе проверяем решение на целочисленность;
   г) если решение целочисленное, переходим к 4;
   д) по желанию добавляем отсечения или отсечение;
   е) ветвимся, создаём на основе решения предыдущей задачи новые(как правило, две) и возвращаемся к началу пункта;
3. return найденное решение.

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.

В первую очередь реализуем весь алгоритм на CPU - то есть последовательно. Первый вопрос, на который следует ответить, - с какими элементами матрицы мы будем работать. Так как используемый симплекс-метод нецелочисленный, может возникнуть проблема округления чисел, что приведёт к неправильному решению даже самых простых задач. Поэтому для большей точности элемент представляем в виде дроби, то есть двух целых чисел. В остальном реализация на CPU была однозначной. Она 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.

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: · нет необходимости копировать большие объёмы данных из памяти CPU в память GPU и наоборот	алгоритм с частичной реализацией на GPU: · плохо распараллеливается на GPU функция поиска pivot_row и pivot_col, так как требуют лишних затрат для синхронизации алгоритма
использования синхронного копирования · быстрее · сильно выигрывает при копировании маленьких объёмов памяти	использование асинхронного копирования · параллельная обработка уже скопированных данных

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

ГЛАВА 3

Практическая реализация

язык программирования, технология CUDA, компилятор, количество строчек кода))

Сравнение GPU и CPU

[35] Рост частот универсальных процессоров упёрся в физические ограничения и высокое энергопотребление, и увеличение их производительности всё чаще происходит за счёт размещения нескольких ядер в одном чипе. Продаваемые сейчас процессоры чаще всего содержат лишь четыре ядра и они предназначены для обычных приложений. И они используют MIMD-архитектуру – множественный поток команд и данных. Каждое ядро работает отдельно от остальных, исполняя разные инструкции для разных процессов.

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.

Основное и наиболее существенное отличие в том, что 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. 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.

Ядра CPU созданы для исполнения одно потока последовательных инструкций с максимальной производительностью, в то время как GPU про-

ектируются для большого числа параллельно выполняемых инструкций. Видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели.

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. 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. Как показано на Рис. 3.1:

	CPU	GPU
тип ядер	MIMD ядра	SIMD ядра
число контроллеров памяти	нечастое использование	почти всегда несколько
пропускная способность	меньше	в разы больше
объём кэш-памяти	больше	128-256 кБ
использование кэш-памяти	для снижения задержек доступа к памяти	для хранения текстурных полосы про
многопоточность	1-2 потока на одно ядро	до 1024 потоков на каждый
доступ к памяти	случайный	последовательный

· последовательный поток инструкций · большое количество исполнительных блоков, кото
· высокая производительность одного потока команд
· доступ к памяти случайный
· ограниченное параллельное выполнение последовательного потока иснструкций

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.

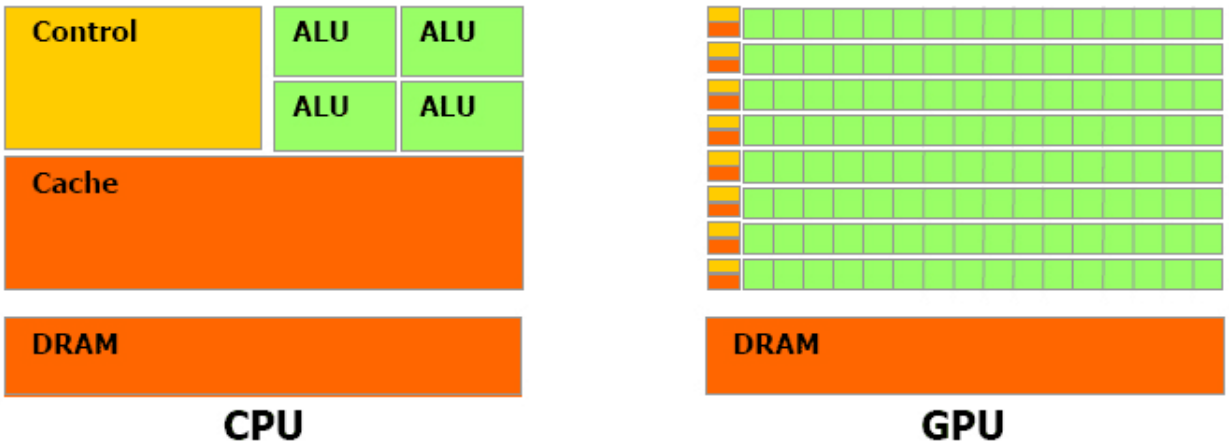

Рисунок 3.1 — Сравнение GPU и CPU

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.

Технология CUDA

[36] [37] [38] [39]

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

Платформа параллельных вычислений CUDA обеспечивает набор рас-

ширений 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 (CUDA Code Samples) 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.

команда разработчиков CUDA создала набор программных уровней для работы с GPU, которые отображены на Рис. 3.2.

Нативный компилятор/отладчик: nvcc/cuda-gdb[linux/mac os], расширение к msvc/TotalView [win].

Если распараллеливать задачу на CPU но 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. У нас в распоряжении миллион потоков и внутри device-функции известен только номер потока. Внутри блока нити находятся физически близко и могут де-

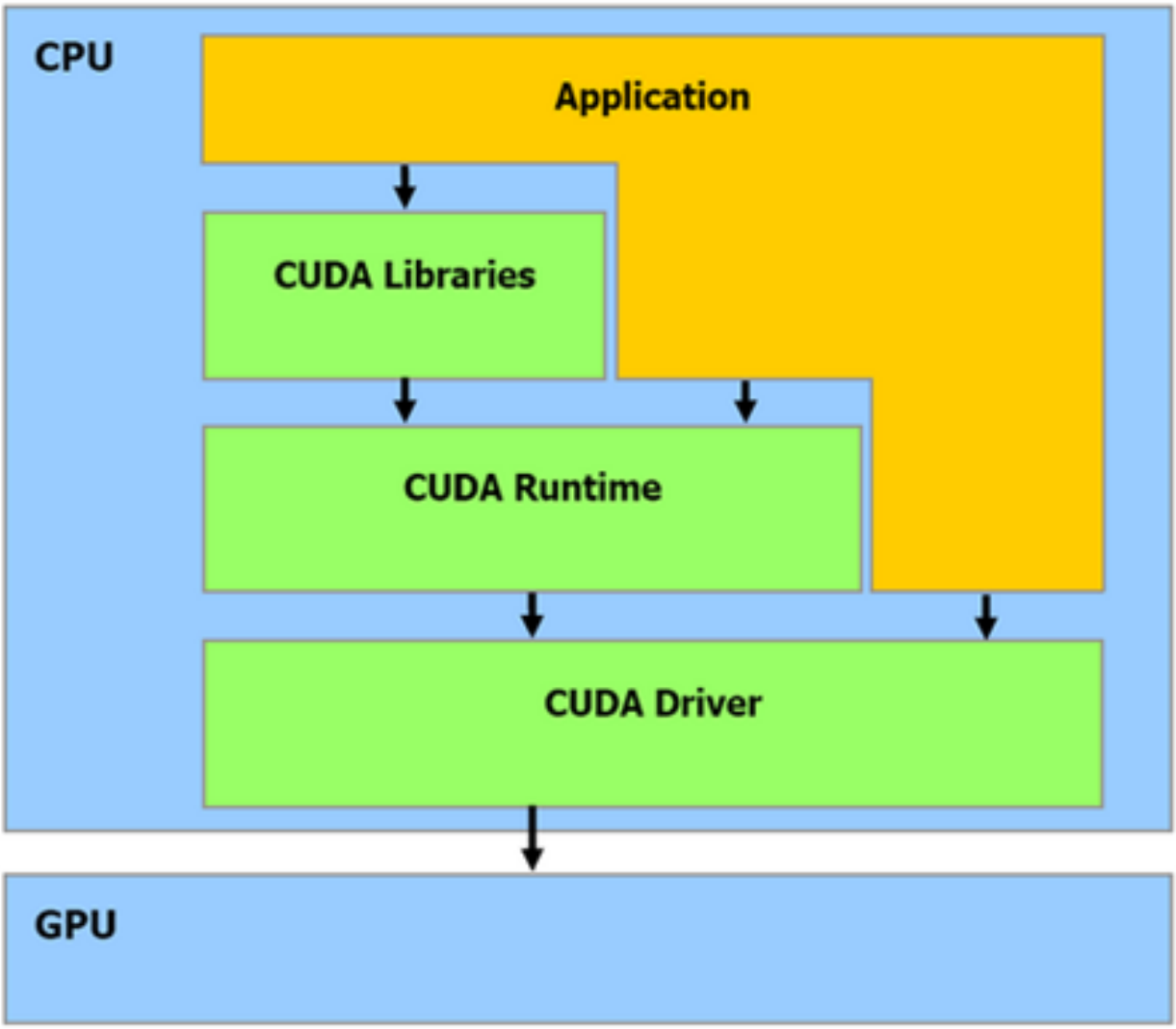

Рисунок 3.2 — Набор программных уровней для работы с GPU

лить общую память. Адресация внутри сетки блоков происходит по идентификации блоков и нитей. Может быть несколько измерений (до трёх). Вся забота думать о том, как делать конфигурацию. От этого много чего зависит.

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 вычисляю-

щаяся на графическом устройстве:

- 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.

Важно быть аккуратным с ветвлениями. Все нити сначала исполнят одну ветку, потом другую. Такой фокус вследствие SIMD. warp - множество нитей, которые прямо сейчас исполняются. Идеально если warp_size (ровно 32) совпадает с block_size.

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 - минимальный объём данных, обрабатываемых SIMD-способом в мультипроцессорах CUDA. Также потоки составляют блоки, которые в свою очередь формируют сетки. Если GPU имеет мало ресурсов, то он будет выполнять блоки последовательно.

Видеокарта может содержать несколько (от 2 до 2000) потоковых мультипроцессоров (streaming multiprocessor). 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 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).

Оптимизация программы CUDA, таким образом, 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

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 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 доста-

точно 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). 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.

Программная реализация

На данном этапе обратимся к плану представленному в главе 1:

Конструирование:

1. проверка выполнения условий, необходимых для успешного конструирования;
2. определение способов последующего тестирования кода;
3. проектирование и написание классов и методов;
4. создание и присвоение имен переменным и именованным константам;
5. выбор управляющих структур и организация блоков команд;
6. блочное тестирование, интеграционное тестирование и отладка собственного кода;
7. взаимный обзор кода и низкоуровневых программных структур членами группы;
8. «шлифовка» кода путем его тщательного форматирования и комментирования;
9. интеграция программных компонентов, созданных по отдельности;
10. оптимизация кода, направленная на повышение его быстродействия, и снижение степени использования ресурсов.

Условия и способы тестирования уже подробно обсуждались в предыдущих разделах. Обратимся теперь к третьему пункту.

Классы и методы

Основной класс - это класс матрицы, который называется Matrix. Остановимся на нём подробнее. Он содержит поля:

- int rows;
- int cols;
- double *numer;
- double *denom;
- int supply;

и методы:

- Matrix(int rows, int cols, int supply = 0);
- Matrix(char const *file_name, int supply = 0);
- Matrix(Matrix const &input, int supply = 0);
- Matrix(Matrix const &input, unsigned int flag, int supply = 0);
- Matrix();
- void freeHost();
- bool print(char const *filename) const;
- void gcd(int const n);
- void reinit(int rows, int cols, int supply = 0);
- void reinit(Matrix const &matrix1, Matrix const &matrix2);
- void pin_mem ();
- void unpin_mem ();
- Matrix &operator=(Matrix const &matrix);

Поля rows и cols оответствуют размерам матрицы - количеству строк и колонок соответственно. Поля numer и denom - массивы, хранящие числительное и знаменательное элементов матрицы.

Матрица представлена в виде одномерного массива колонок. Такая реализация выбрана вместо более распространённого представления в виде строк, потому что основные преобразования матрицы происходят при манипуляции колонок. То есть в первую очередь мы нормируем колонку. На следующем шаге мы нормируем ведущую строчку, домножая ведущую колонку на элемент и вычитая из текущей колонки. Что означает, что используемый метод симплекса можно представить в виде манипуляции колонками.

Последнее поле supply используется при добавлении элементов в матрицу. То есть в этой переменной хранится количество дополнительных свободных ячеек в массивах denom и numer. Это необходимо в алгоритме при добавлении отсечений к матрице.

Перейдём к описанию методов. Первые четыре метода - конструкторы, которые исполбзуют supply как необязательный параметр. Первый конструктор создаёт нулевую матрицу с данным количеством колонок и строк. Второй метод считывает матрицу из текстового файла. В данном файле

первые два считанных значения - количество колонок и строк в матрице. Далее попорядку считываются элементы матрицы в виде двух чисел - числитель и знаменатель. Но элементы считываются в построчном виде, по причине того что матрицы в C
C++ представляются в построчном виде.

Следующий конструктор принимает на вход матрицу - конструктор копирования. И последний конструктор помимо матрице как обязательный параметр принимает переменную $flag$. Дело в том, что реализация метода с асинхронным копированием использует закреплённую память, так как иначе это может привести ко множеству ошибок. При выделении закрпелённой памяти используется функция cudaHostAlloc, которая в свою очередь использует параметр flags, чтобы специфицировать использование выделяемой памяти.

Деструктор очищает выделенную динамически память. Но нужно запомнить, что при выделении закреплённой памяти нужно очищать выделенную память самостоятельно, для этого используется метод freeHost();

Метод print печатает матрицу в файл, имя которого принимается как входящий аргумент. Метод gcd сокращает элемент матрицы, используя алгоритм Евклида. Оба метода reinit перезаписывают матрицу. Первый метод выделяет памяти ровно столько, сколько требуется, а второй добавляет к существующей матрице дополнительные строчки.

Методы pin_mem() и unpin_mem() закрепляют и открепляют уже выделенную динамически память, которые изначально выделялась незакреплённой.

Для удобства также реализован оператор присваивания.

Произведение матриц

В приведённый алгоритм можно несколько модернизировать при использовании произведения матриц. Так как отсечения добавляются в конец таблицы, то первоначально алгоритм симплекса проходит все те же стадии, что и первоначальная матрица. В таком случае можно сохранить данные

преобразования таблицы в некоторую матрицу transformation. Таким образом не придётся много раз выполнять одну и ту же работу.

Используем всем известный алгоритм произведения матриц строка-столбец.

Отметим, что обозначенное угловыми скобками ядро также принимает необязательный аргумент stream. Этот запуск ядра выполняется асинхронно, как и две предыдущие операции копирования в память GPU и последующая операция копирования в память CPU. Строго говоря, не исключено, что мы выйдем из цикла ещё до того, как начнётся копирование памяти или исполнение ядра. Гарантируется лишь, что первая операция копирования, помещённая в поток, завершится до начала второй операции копирования. А вторая операция копирования завершится до начала исполнения ядра, а исполнение ядра закончится до начала третьей операции копирования. То есть, как и было сказано в начале главы, потом ведёт себя как упорядоченная очередь задач для GPU. По выходе из цикла for в очереди ещё могут находиться задания, которые GPU предстоит выполнить.

Асинхронное копирование имеет смысл, только если поиск пивота происходит на CPU. А эта часть плохо параллелиться, чтобы её производить на гпу, то есть снова асинхронно копировать туда сюда.

Обработка ошибок требует усовершенствования, так как при возникновении какой-либо программа вызывает аварийное завершение.

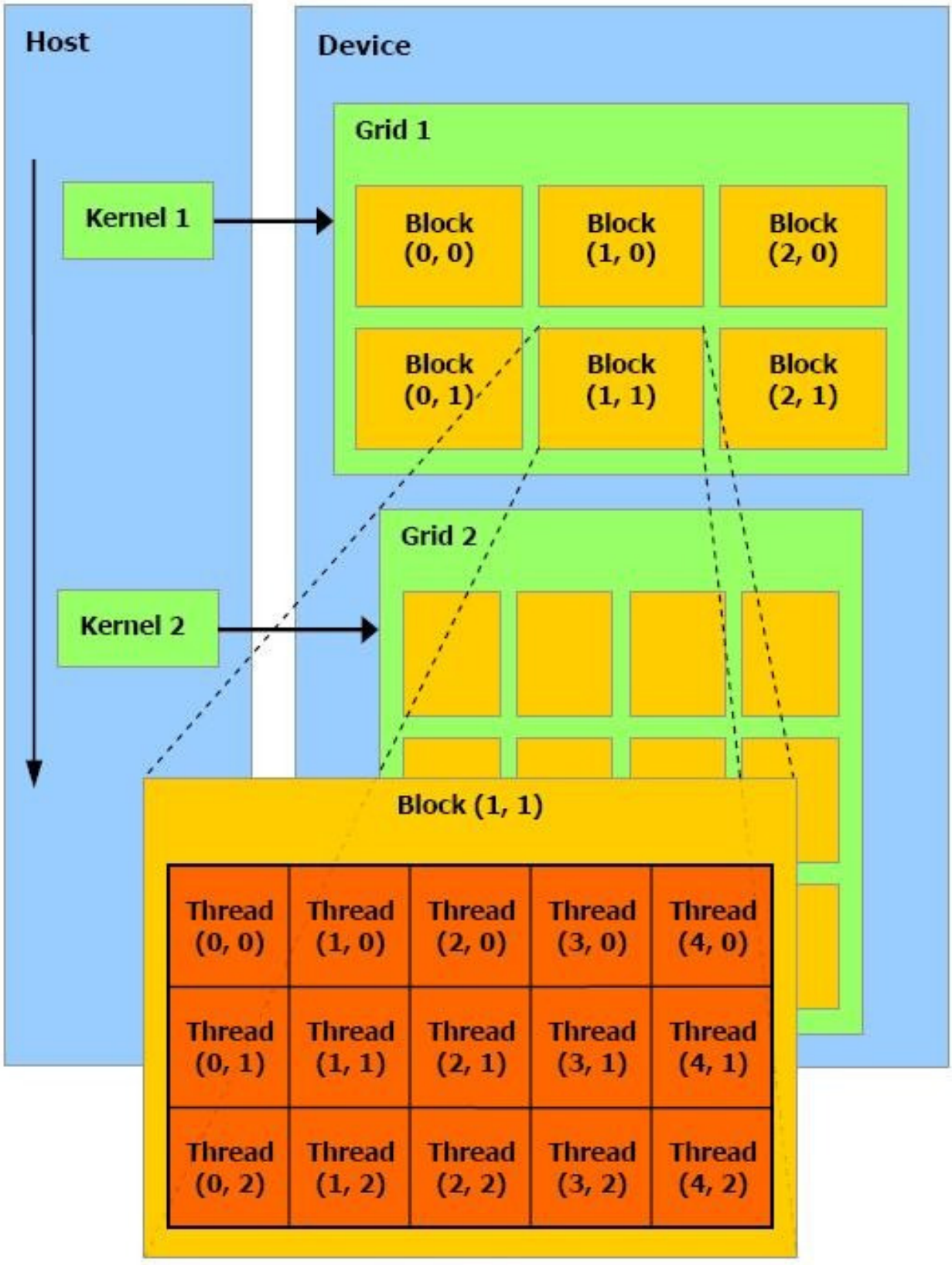

Рисунок 3.3 — Конструкция grid

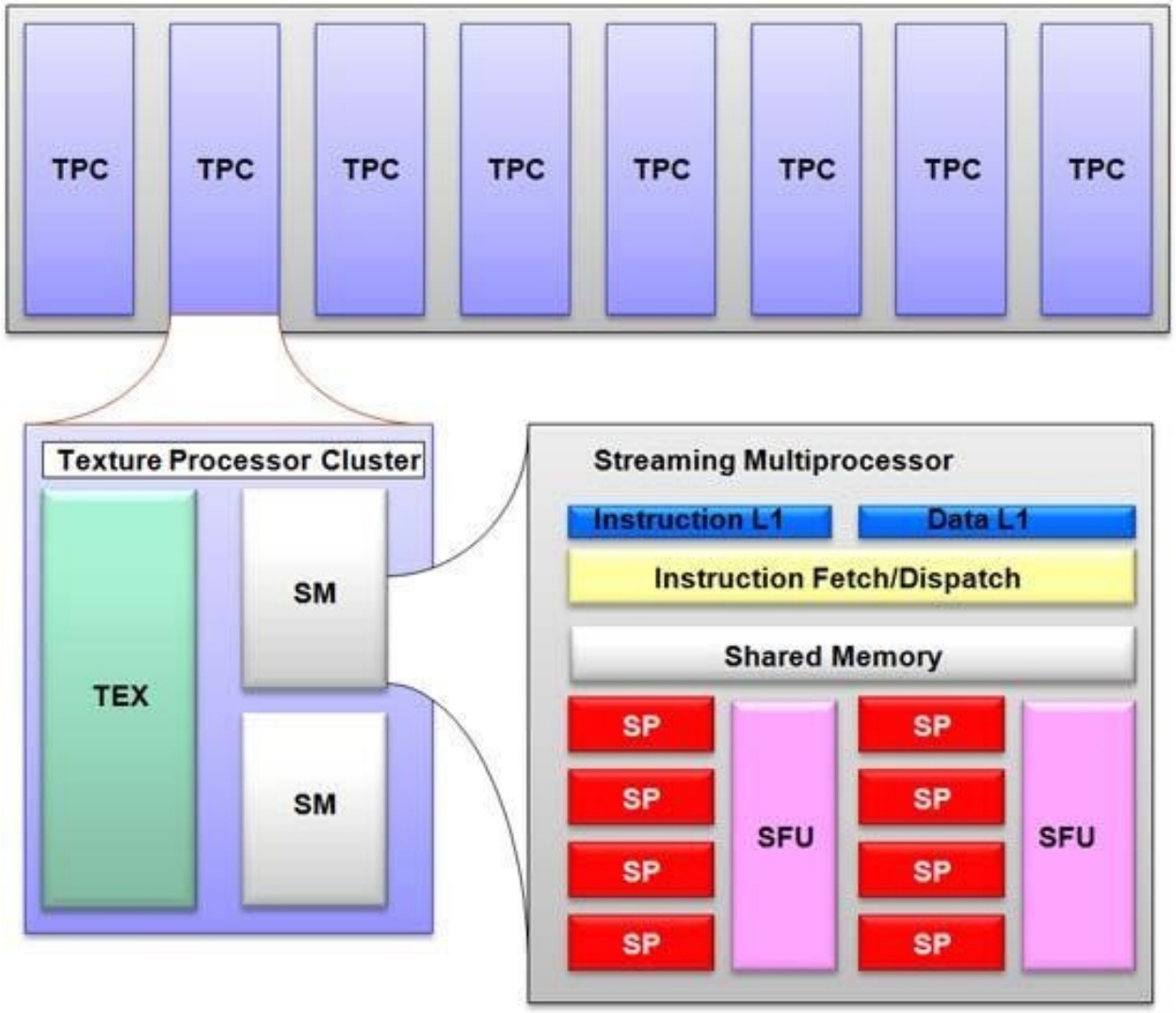

Рисунок 3.4 — Конструкция TCP и SM

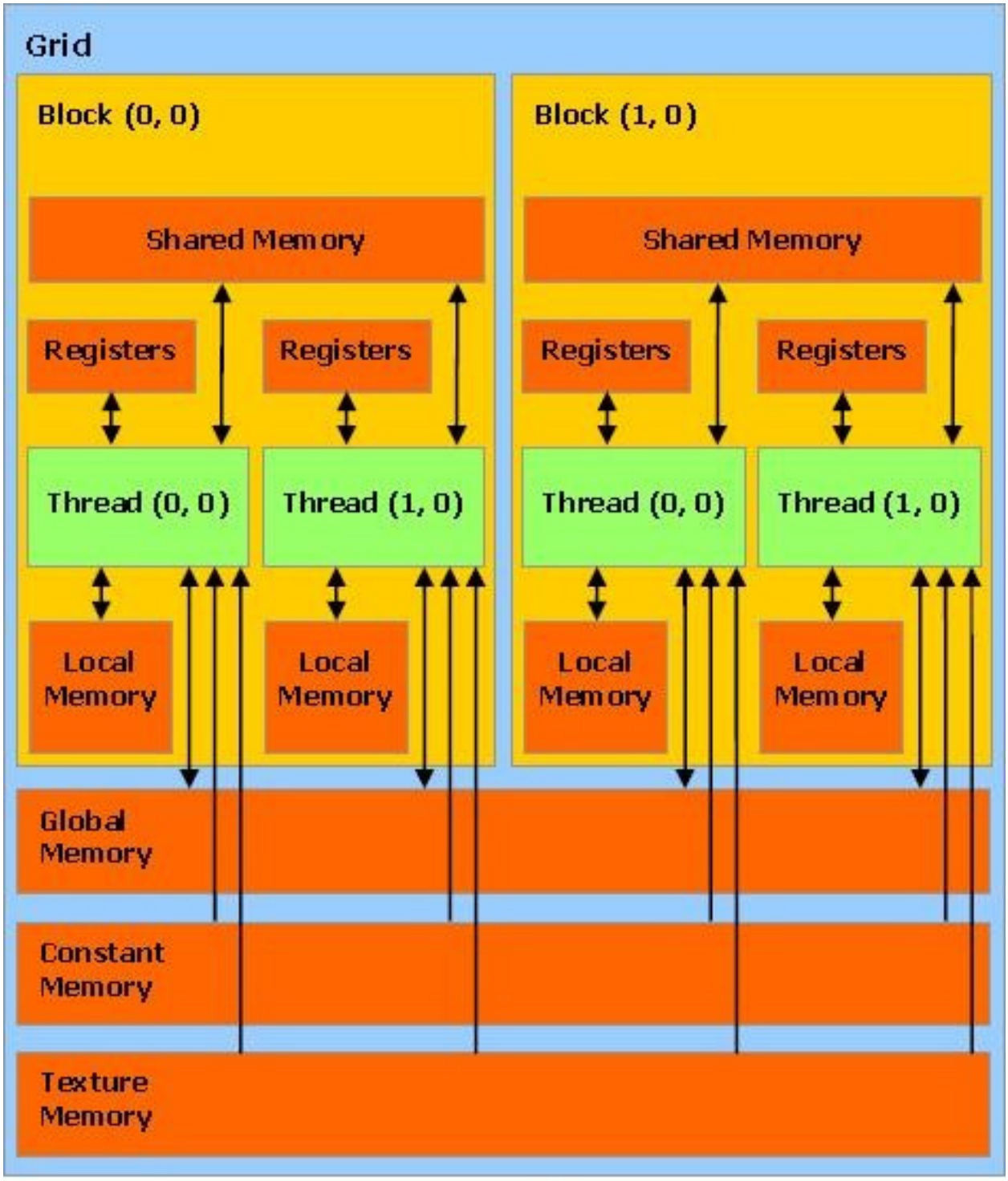

Рисунок 3.5 — Иерархия памяти CUDA

ГЛАВА 4

Анализ

Исследование корректности

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.

Исследование эффективности

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.

Упражнения из учебника (с.146 Шевченко Золотых)
Алгоритм Branch-and-Bound экспоненциально (2^t итераций) зависит от длины записи коэффициентов задачи целочисленного линейного программирования:

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

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:

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

закон Амдала и Густафсона

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 Artifficial Intelligence, 2006. 13 с.
3. E. Tsang. Foundations of Constraint Satisfaction. New York: Academic Press, 1993. 421 с.
4. Г. Карманов В. Математическое программирование. Москва: Наука, 1986. 288 с.
5. L. Balinski M. Integer Programming: Methods, Uses, Computations. Management Science, 1965. 253 с.
6. Kautz H. Selman B. Planning as Satisfiability. Chichester: John Wiley and Sons, 1992. 359 с.
7. Barnier N. Brisset P. Graph coloring for air traffic flow management. Annals of Operations Research, 2004. 163 с.
8. Сараев А.Д. Щербина О.А. Системный анализ и современные информационные технологии // Труды Крымской академии наук. Симферополь: СОНАТ, 2006. 47 с.
9. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences, 1974. 95 с.

10. Hooker J.N. Yan H. / V.A. Saraswat P. van Hentenryck (eds.). Verifying logic circuits by Benders decomposition // Principles and Practice of Constraint Programming. Cambridge: MIT Press, 1994.
11. A. Gotlieb B. Botella M. Rueher. Automatic test data generation using constraint solving techniques. ACM SIGSOFT Software Engineering Notes, 1998. 53 c.
12. 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.
13. Dechter R. Frost D. Backtracking algorithms for constraint satisfaction problems. University of California: Department of Information and Computer Science, 1999.
14. 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.
15. P. Meseguer. Constraint satisfaction problems: an overview. AI Communications, 1989. 3 c.
16. [Miguel I. Shen Q. Solution techniques for constraint satisfaction problems: advanced approaches. Artifficial Intelligence Review, 2001. 267 c.
17. R. Dechter. Constraint networks // Encyclopedia of Artifficial Intelligence (2nd edition). New York: Wiley and Sons, 1992. 276 c.
18. W. Hower. Constraint satisfaction. Algorithms and complexity analysis. Information Processing Letters, 1995. 171 c.
19. A.K. Mackworth. Constraint satisfaction // Encyclopedia of Artifficial Intelligence (second edition). New York: Wiley, 1992. 285 c.
20. R. Apt K. Principles of Constraint Programming. New York: Cambridge University Press, 2003. 407 c.
21. Fruehwirth T. Abdennadher S. Essentials of Constraint Programming. Springer, 2003. 144 c.
22. Marriott K. Stuckey P. J. Programming with Constraints: An Introduction. Cambridge: MIT Press, 1998. 483 c.
23. Rossi F. van Beek P. Walsh T. Handbook Of Constraint Programming.

Elsevier, 2006. 978 с.

24. P. Van Hentenryck. Constraint Satisfaction in Logic Programming. Cambridge: MIT Press, 1989. 224 с.
25. Van Hentenryck P. Michel L. Deville Y. Numerica: A Modeling Language for Global Optimization. Cambridge: MIT Press, 1997. 210 с.
26. P. Van Hentenryck. The OPL Optimization Programming Language. Cambridge: MIT Press, 1999. 255 с.
27. Van Hentenryck P. Michel L. Constraint-Based Local Search. Cambridge: MIT Press, 2005. 442 с.
28. Канторович Л. В. Математические методы организации и планирования производства. Санкт-Петербург: изд. ЛГУ, 1939.
29. С. Макконнелл. Совершенный код. Русская Редакция, Microsoft Press, 2010.
30. F. Glover. Tabu search-Part II. ORSA Journal on computing, 1989.
31. Russell St. J. Norvig P. Artificial Intelligence: A Modern Approach (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, 2003. 111 с.
32. Kirkpatrick S. Gelatt Jr C. D. Vecchi M. P. Optimization by Simulated Annealing. Science, 1983. 671 с.
33. 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 с.
34. 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.
35. С.А.Полетаев. Параллельные вычисления на графических процессорах. Новосибирск: Ин-т систем информатики имени А. П. Ершова СО РАН, 2008. 271 с.
36. corp. NVIDIA. CUDA C Best Practices Guide. CUDA SDK, 2014.
37. Sanders J. Kandrot E. CUDA by Example An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010.
38. David B. Kirk Wen-mei W. Hwu. Programming Massively Parallel

Processors A Hands-on Approach. Morgan Kaufmanm, 2012.

39. Боресков А.В. Харламов А.А. Основы работы с технологией CUDA. ДМК Пресс, 2010.