Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики

Компиляторы для глубоких нейросетевых моделей

Автоматическая оптимизация вывода нейросетевых моделей на уровне слоев в Apache TVM

При поддержке компании YADRO

Родимков Ю.А.

Содержание

ВВЕДЕНИЕ

Цель оптимизации программ

- ***Цель оптимизации программ*** – получение из работающего* варианта программы другого работающего* варианта, удовлетворяющего некоторым критериям

* принципиальный момент

Рассматриваемые критерии

Основными ***критериями*** при оптимизации программ являются:

- **Скорость работы**

  Миф: компьютеры работают все быстрее. Ничего делать не нужно ☺
  // Большие задачи; Real-time; Финансы и др. области

- **Объем используемой памяти**

  Миф: много дешевой памяти и будет еще больше. Ничего делать не нужно ☺
  // Большие задачи; ограничения на ускорителях

- **Объем места, занимаемого на диске**

  Выглядит сомнительным, но вспоминаем про инженерные расчеты
  и ограничения на кластерах

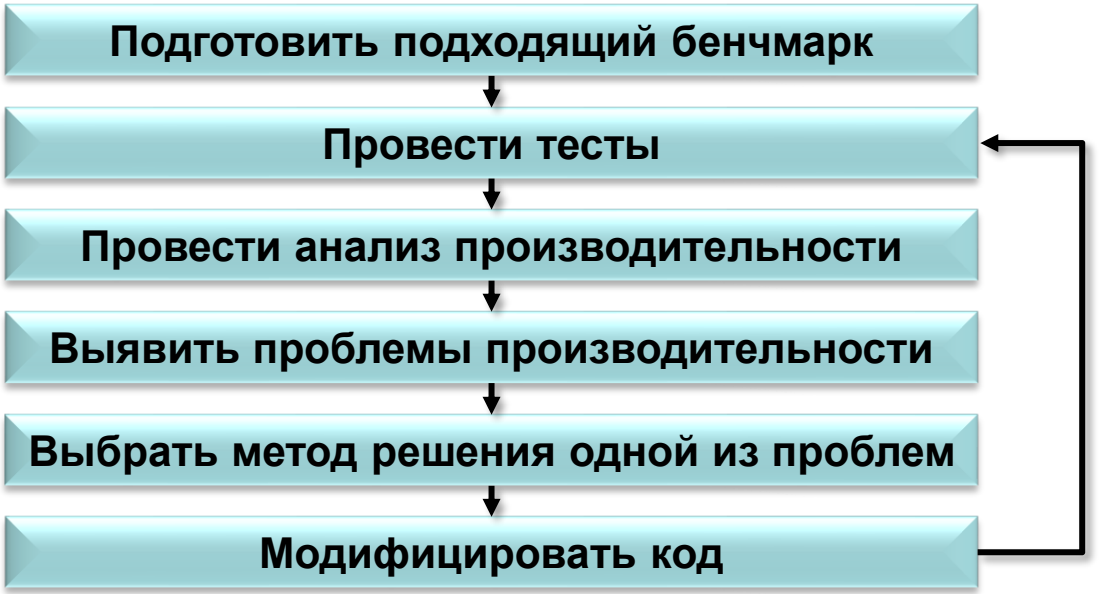
Правила оптимизации*

- Прежде чем оптимизировать программу, ***убеждаемся в ее работоспособности***
- Основной прирост производительности приходит от алгоритмической оптимизации и выбора структур данных, а не от «трюков». ***Ищем эффективные алгоритмы и подходящие СД***
- Оптимизация кода ≠ ассемблерная реализация. ***Улучшаем код для увеличения быстродействия в рамках ЯПВУ***. ***Используем возможности компилятора***
- ***Используем оптимизированные библиотеки***

* По материалам *К. Касперски. Техника оптимизации программ. Эффективное использование памяти. БХВ-Петербург, 2003.*

Жизненный цикл оптимизации

Предполагаем, что программа ***написана*** и ***работает***, но работает ***недостаточно быстро***



*Много **тонкостей***

*Тензорные компиляторы пытаются помочь программистам, решая часть задач оптимизации **автоматически / автоматизированно***

Терминология Apache TVM

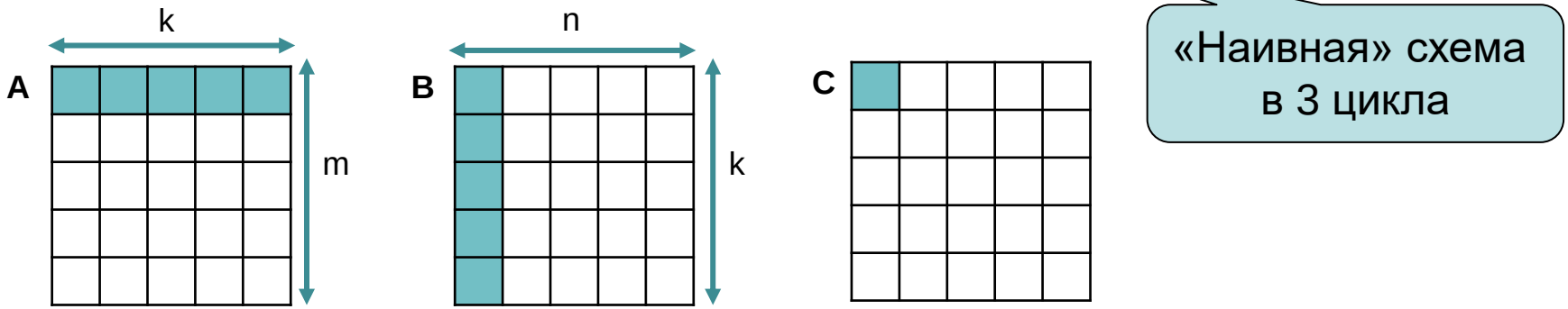
- ***Тензорное выражение (TE)*** описывает аппаратно-независимую реализацию операции
- ***План вычислений (schedule)*** – описание программно-аппаратных оптимизаций
- ***Тензорное внутреннее представление (TIR)*** – низкоуровневая аппаратно-зависимая программная реализация тензорного выражения
  - Генерируется на основании тензорного выражения и плана вычислений
  - Используется для компиляции нейросетевой модели в машинный код

- Примечание:
  - В контексте данной лекции **оптимизация слоев НЕ имеет отношения к обучению модели**. Речь идет про **время выполнения слоя!**

Пример оптимизации. Наивный алгоритм

- Пример умножения матриц из прошлой лекции:

```
k = te.reduce_axis((0, 1024), "k")           # ось редукции
A = te.placeholder((1024, 1024), name="A")   # пустой тензор
B = te.placeholder((1024, 1024), name="B")   # пустой тензор
C = te.compute((1024, 1024),                 # размер выходного тензора
      lambda m, n: te.sum(A[m, k] * B[k, n],# выражение для
                          axis=k))           # вычисления элементов
```



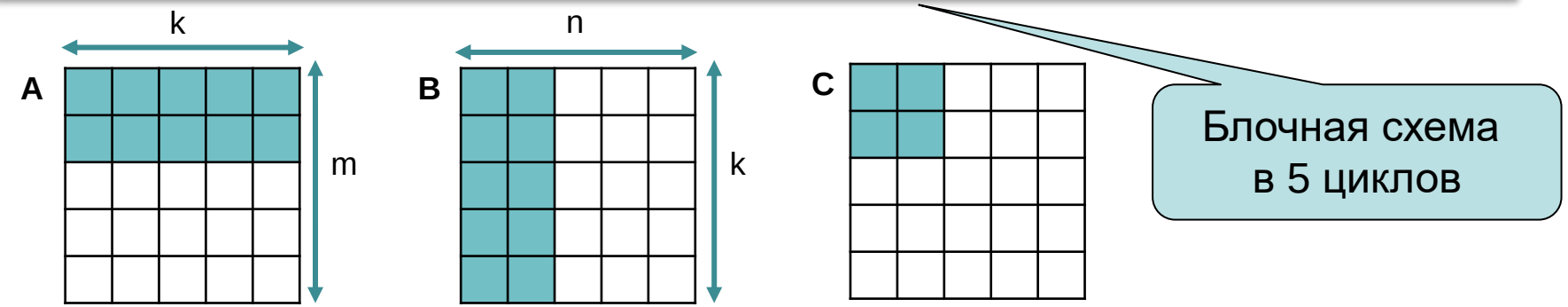
«Наивная» схема в 3 цикла

Пример оптимизации. Блочный алгоритм

- Преобразование «наивного» плана исполнения для алгоритма умножения матриц в блочный:

```
s = te.create_schedule(C.op)                  # создание объекта плана
y, x = s[C].op.axis                           # размеры результирующего тензора
k = s[C].op.reduce_axis[0]                    # размер цикла редукции

yo, xo, yi, xi = s[C].tile(y, x, 8, 8) # разбиение на блоки размера 8x8, редукция
s[C].reorder(yo, xo, k, yi, xi)               # переупорядочивание циклов
```



Проблема выбора и реализации планов

- Возникает ряд вопросов:
  - Почему мы берем размер блока, равный 8?
  - Подходит ли этот размер для текущей архитектуры? А для новой архитектуры?
  - Почему мы используем такой порядок циклов, и где другие оптимизации?
- ***Поиск оптимального или удовлетворяющего целям проекта плана – сложная, но критически важная задача для оптимизации производительности!***
- Для выбора качественного плана необходимо:
  - Знать алгоритмы и структуры данных
  - Знать особенности вычислительных архитектур
  - Проводить большое количество экспериментов и анализировать их результаты
  - Уметь профилировать программы и делать выводы по результатам профилировки

АВТОМАТИЧЕСКАЯ ОПТИМИЗАЦИЯ ВЫВОДА НЕЙРОСЕТЕВЫХ МОДЕЛЕЙ НА УРОВНЕ СЛОЕВ

Автоматическая оптимизация нейросетевых преобразований...

- Задача автоматической программной оптимизации нейросетевых преобразований сводится к математической задаче оптимизации:

$$f(x) \to \min_{x \in X},$$

где X – параметры различных реализаций оператора нейронной сети. Примеры:

  – Порядок циклов

  – Использование директив препроцессора (развертывание, векторизация циклов и т.д.)

- Функционал качества $f(x)$ отражает, насколько эффективно выполнена программная оптимизация преобразования. Примеры:
  - Время выполнения
  - Количество операций, выполненных в секунду (FLOPs)
- Оптимизация может выполняться с использованием случайного поиска, полного перебора, методов локальной или глобальной оптимизации

Автоматическая оптимизация нейросетевых преобразований

- ***Примечание:***
  - В тензорных компиляторах вместо минимизации времени исполнения одного прохода по сети может рассматриваться ***задача максимизации пропускной способности*** (например, за счет одновременной обработки нескольких проходов)
  - Увеличение пропускной способности ***не всегда*** предполагает минимизацию времени исполнения одного прохода

Предпосылки для эффективного решения задачи оптимизации

- ***Разнообразие методов оптимизации «черного ящика» (black-box optimization)***

- ***Относительно низкая стоимость эксперимента***
  - Компиляция и запуск тензорной программы занимает несколько секунд

- ***Большое количество похожих операторов***
  - Оптимизация одинаковых операторов для различных форм и конфигураций размещения входных данных
  - Априорная информация об эффективности какой-либо реализации позволяет использовать перенос знаний (transfer learning) при настройке операторов

Подходы в рамках Apache TVM

- Apache TVM содержит несколько поколений методов автоматической оптимизации тензорных выражений:
  - AutoTVM
  - Auto-scheduler
  - MetaSchedule
- ***Примечания:***
  - Эффект от оптимизации зависит от формата хранения данных, архитектуры нейронной сети и целевого устройства
  - Улучшение хорошо оптимизированных под целевую платформу слоев приведет к незначительному увеличению скорости вывода, либо не даст никакого эффекта
  - Автоматическая оптимизация операторов не всегда позволяет достигнуть лучших показателей производительности вследствие ограниченности области перебора или количества итераций оптимизации

AUTOTVM

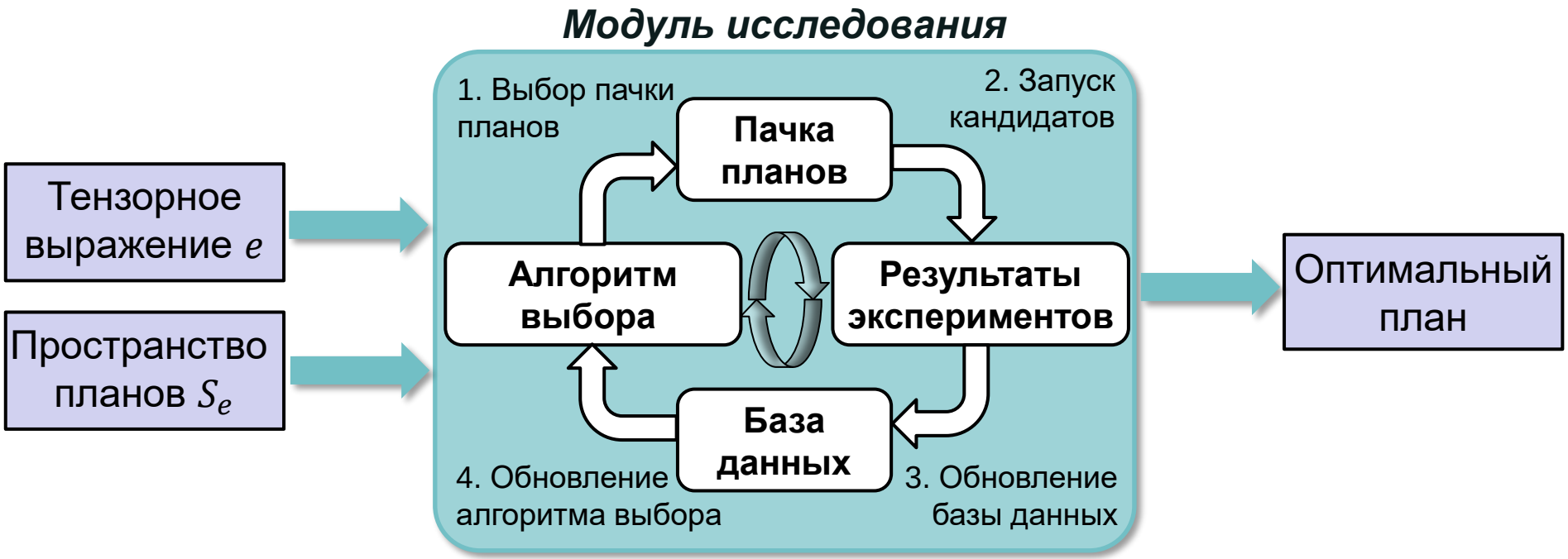
AutoTVM. Краткая информация

- AutoTVM – подход, предложенный в 2018 году
- AutoTVM перебирает различные варианты реализованных в Apache TVM планов вычислений для тензорных выражений – ***шаблонов***
- ***Шаблоны описывают пространство корректных планов вычислений*** тензорного выражения
- Шаблоны хранятся в виде набора файлов с программным кодов в Apache TVM

- ***Примечание:*** с практической точки зрения шаблон – функция, реализующая все возможные варианты оптимизаций через ветвления или через параметры, которые могут принимать фиксированное число значений

* Chen T. et al. Learning to optimize tensor programs // Advances in Neural Information Processing Systems. – 2018. – Т. 31.

AutoTVM. Схема работы



Алгоритм выбора. Модель затрат

- В AutoTVM для оценки производительности используется ***модель затрат – градиентный бустинг деревьев*** (XGBoost)
  - При обучении может использоваться функция потерь для восстановления регрессии или функция потерь для ранжирования
  - При обновлении базы данных модель затрат каждый раз обучается заново
- ***Входные данные модели затрат – признаки***
  - Информация о структуре циклов. Примеры:
    - Количество обращений к памяти
    - Коэффициент повторного использования данных
  - Аннотации циклов. Примеры:
    - Векторизация
    - Развертывание
    - Привязка потоков

Алгоритм выбора

- Модуль исследования может работать по одному из следующих принципов:
  - **Случайный поиск или полный перебор.** Выбор оптимального плана выполняется случайным или последовательным перебором из заранее реализованных планов вычислений
  - **Использование эволюционного поиска.** Используется скрещивание и мутация для создания следующей конфигурации. После генерации новой пачки она запускается на целевом устройстве и процедура повторяется
  - **Использование статической модели затрат.** На каждом шаге для проверки выбираются планы, оптимальные с точки зрения модели затрат

- ***Примечание:***
  - Когда модель затрат обучена, её можно применить для оптимизации нейронных сетей, имеющих схожую архитектуру, на аналогичном целевом устройстве

Параметры алгоритмов

Входные данные:
$e \leftarrow$ тензорное выражение
$S_e \leftarrow$ множество планов для тензорного выражения e
Выходные данные:
s^* – выбранный план вычислений

Общие параметры:
b – внутренний параметр AutoTVM, размер пачки планов для тестирования на итерации
Параметры, специфичные для эволюционного поиска:
k – внутренний параметр AutoTVM, количество лучших кандидатов с прошлой итерации которые будут использоваться на новой итерации
$\mathrm{PopulationSize}$ – размер популяции
Параметры, специфичные для модели затрат:
ε – внутренний параметр AutoTVM, количество случайно выбираемых планов для тестирования

Обозначения:
s_i – план вычислений, c_i – время выполнения, g – функция перевода плана вычислений в машинный код, f – функция определения времени вывода сети

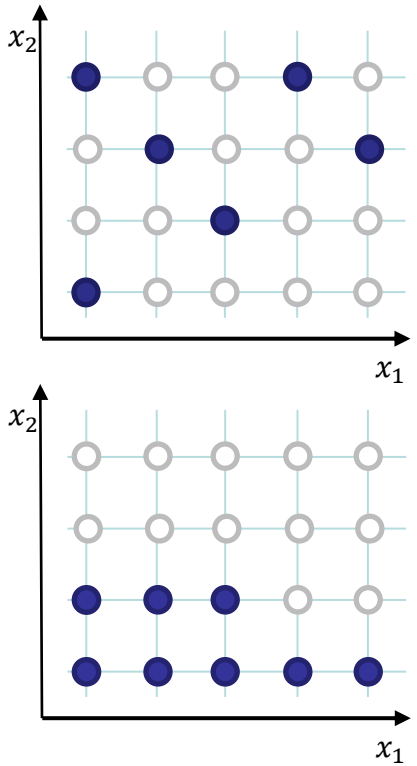
Алгоритм выбора. Случайный поиск и полный перебор

$D \leftarrow$ пустое множество результатов выполнения планов

while текущая итерация < максимальная итерация:
    If метод == случайный:
        $S \leftarrow$ сэмплирование b случайных кандидатов из S_e
    If метод == последовательный:
        $S \leftarrow$ выбор следующих b кандидатов из S_e
    Для каждого $s_i \in S$ выполнить запуск на целевом устройстве

    For $s_i \in S$:
        $c_i \leftarrow f\left(g(e, s_i)\right)$; $D \leftarrow D \cup \{(e, s_i, c_i)\}$; $S_e \leftarrow S_e \setminus s_i$

$s^* \leftarrow$ лучший план из множества D



Алгоритм выбора. Эволюционный поиск

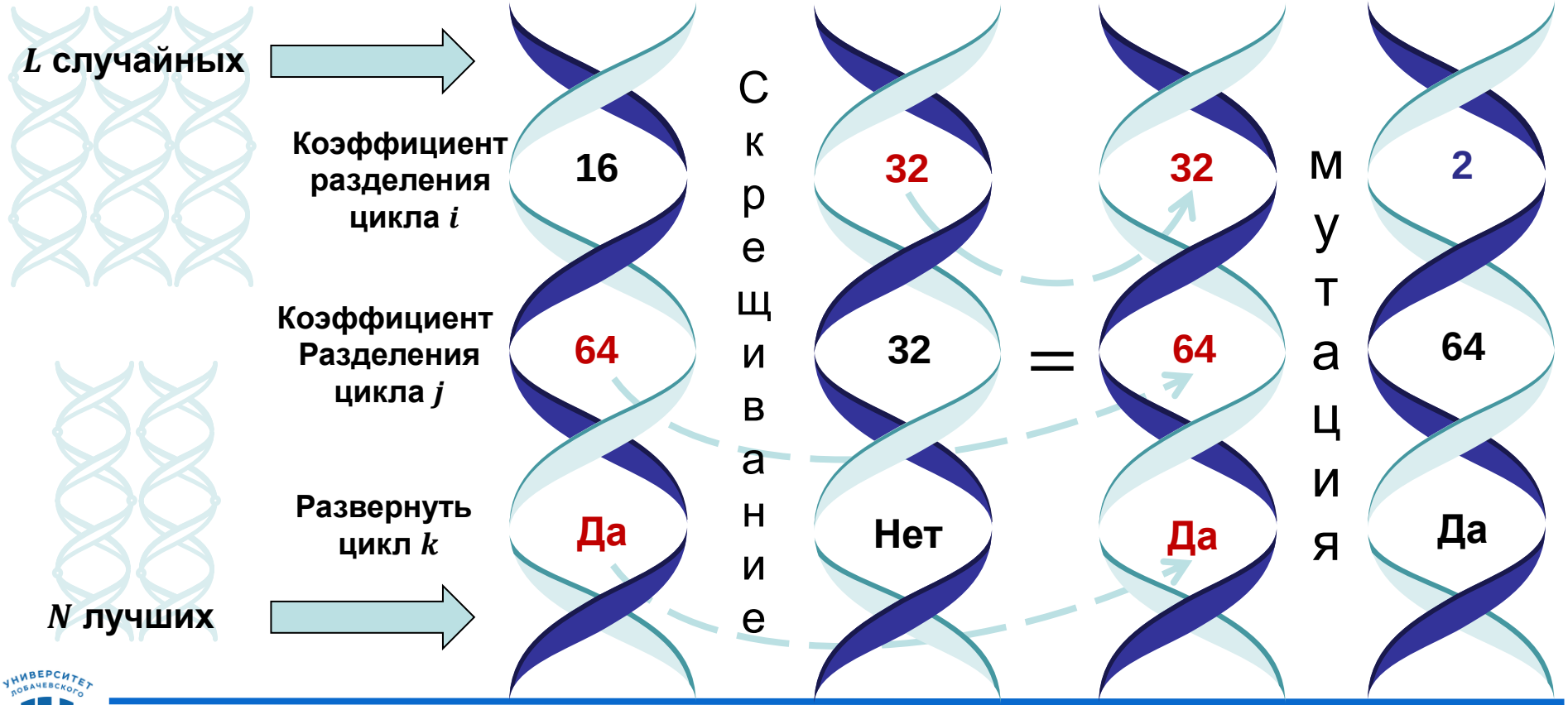
$D \leftarrow$ пустое множество результатов выполнения планов
Случайная инициализация популяции
while текущая итерация < максимальная итерация:
    If $|D| < PopulationSize$: $S \leftarrow$ случайный выбор b кандидатов из S_e
    Else:
        Селекция – $S \leftarrow$ случайный выбор $b - k$ кандидатов из S_e
                и объединение с k лучшими кандидатами с прошлой партии

        Скрещивание – генерация нового поколения через обмен параметров
                между планами

        Мутация – случайное изменение параметров плана с некоторой вероятностью
                (например, коэффициента разбиения цикла) в S

    Для каждого $s_i \in S$ выполнить запуск на целевом устройстве
    $c_i \leftarrow f\big(g(e_i, s_i)\big)$; $D \leftarrow D \cup \{(e_i, s_i, c_i)\}$; $S_e \leftarrow S_e \setminus s_i$
$s^* \leftarrow$ лучший план из множества D

Алгоритм выбора. Эволюционный поиск



Алгоритм выбора. Модель затрат

$D \leftarrow$ пустое множество результатов выполнения планов
while текущая итерация < максимальная итерация:
    Этап 1. $Q \leftarrow$ запуск алгоритма имитации отжига для отбора кандидатов из S_e на основании модели затрат $\hat{f}$

    Этап 2. $S \leftarrow$ выбор $(1-\varepsilon)b$ кандидатов с учетом качества и разнообразия из Q
    $S \leftarrow S \cup \{$случайное сэмплирование εb кандидатов$\}$

    Этап 3. Для каждого $s_i \in S$ выполнить запуск на целевом устройстве
    $c_i \leftarrow f\big(g(e, s_i)\big)$; $D \leftarrow D \cup \{(e, s_i, c_i)\}$; $S_e \leftarrow S_e \setminus s_i$

    Этап 4. Обновление модели затрат с помощью D
$s^* \leftarrow$ лучший план из множества D

Выбор кандидатов на каждой итерации алгоритма

- При выборе кандидатов для запуска учитывается ***качество*** и ***разнообразие***
- Множество кандидатов $S = \{s = (s_1, s_2, \dots, s_m), s_i - \text{признак}\}$
- Примеры признаков: порядок циклов, коэффициенты блоков
- AutoTVM выбирает множество кандидатов S, для которых достигается максимум следующего функционала:

$$L(S) = -\sum_{s \in S} \hat{f}\big(g(e,s)\big) + \alpha \sum_{j=1}^{m} \left| \cup_{s \in S} \{s_j\} \right|$$

  - Первое слагаемое побуждает выбирать кандидатов с низкими затратами по времени
  - Второе слагаемое подсчитывает количество различных значений каждого конкретного признака в множестве S

Пример

- Демонстрация написания шаблонов для умножения матриц с помощью AutoTVM lectures/sources/07_AutoTVM_GEMM.ipynb

AUTO-SCHEDULER

Auto-scheduler

- ***Можно ли автоматически оптимизировать тензорные выражения?***
  - Можно ли автоматически генерировать ***программные оптимизации*** и проверять их?
  - Можно ли автоматически генерировать ***пространство поиска*** на основании тензорного выражения?
  - Как ***распределить ресурсы*** при оптимизации ***между слоями***?
- В 2020 году предложено новое поколение автоматической оптимизации нейросетевых преобразований – Auto-scheduler (или Ansor), – призванное решить поставленные вопросы
- Auto-scheduler принимает на вход тензорное выражение и с помощью набора правил и эволюционного поиска автоматически генерирует реализации оператора

* Zheng L., et al. Ansor: Generating {High-Performance} tensor programs for deep learning //14th USENIX symposium on operating systems design and implementation (OSDI 20). – 2020. – С. 863-879.

Auto-scheduler vs. AutoTVM

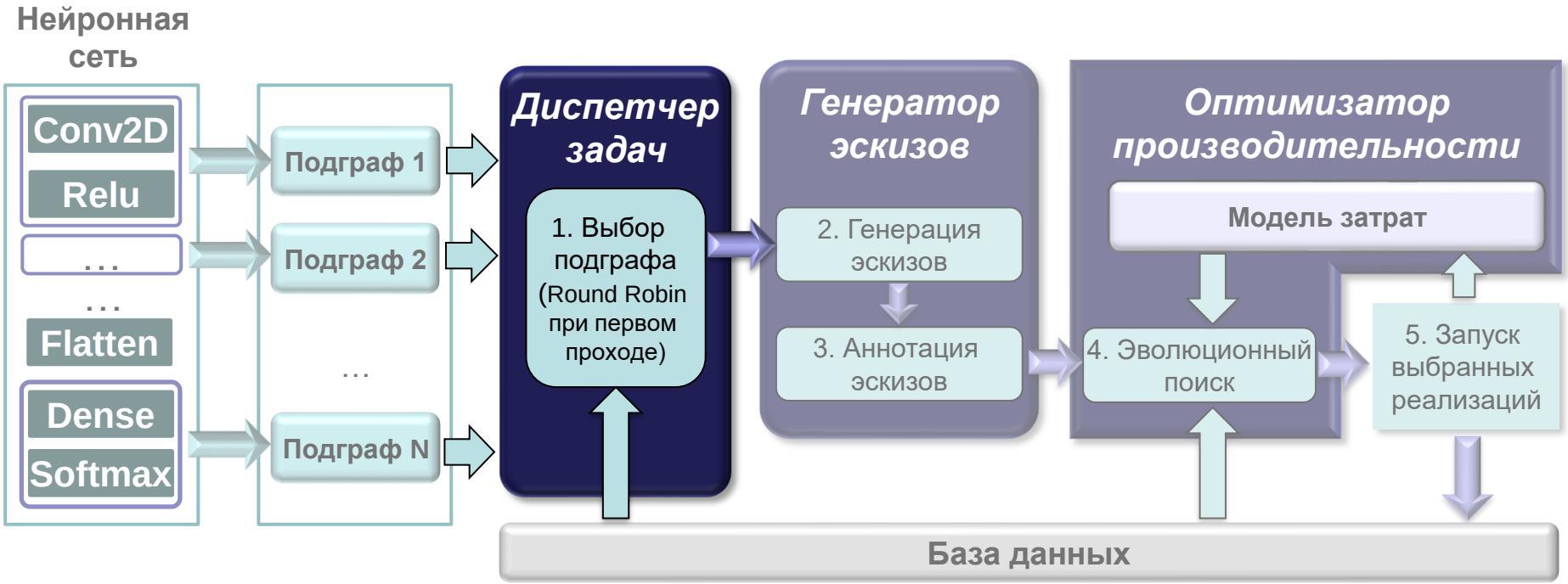
Общая схема	AutoTVM	Auto-scheduler
Шаг 1: Реализация тензорного выражения *Относительно простая часть*	Умножение матриц C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k))	Умножение матриц C = te.compute((M, N), lambda x, y: te.sum(A[x, k] * B[k, y], axis=k))
Шаг 2: Реализация плана вычислений *Относительно сложная часть*	***20-100 строк «хитрого» DSL-кода*** Задать пространство поиска cfg.define_split("tile_x", batch, num_outputs=4) cfg.define_split("tile_y", out_dim, num_outputs=4) … Применить конфигурации bx, tx, tx, xi = cfg["tile_x"].apply(s, c, c.op.axis[0]) by, ty, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])	***Не требуется***
Шаг 3: Запуск оптимизации	tuner.tune()	tuner.tune()

Auto-scheduler

- Модель глубокого обучения разбивается на небольшие подграфы – один или несколько последовательных операторов
- Auto-scheduler состоит из ***трех ключевых компонент***:
  - ***Генератор эскизов*** определяет высокоуровневые структуры программ – *эскизы* – и составляет миллиарды низкоуровневых вариантов реализаций – *аннотаций*
  - ***Оптимизатор производительности*** выбирает реализацию для тестирования с помощью эволюционного поиска и модели затрат
  - ***Диспетчер задач*** распределяет ресурсы при оптимизации между подграфами

Auto-scheduler. Схема работы

Auto-scheduler. Схема работы



Диспетчер задач

- Нейронную сеть можно разделить на множество независимых подграфов (например, conv2d + relu)
- Для некоторых подграфов затраты времени на их оптимизацию не приводят к значительному улучшению сквозной (полной) производительности
  - Подграф не является узким местом в производительности
  - Оптимизация приводит лишь к минимальному улучшению производительности подграфа
- ***Диспетчер задач*** – компонент, который оценивает важность подграфов с точки зрения их вклада в итоговую производительность и динамически распределяет временные ресурсы между подграфами

Диспетчер задач. Постановка задачи

- Обозначения:
  - n – количество подграфов
  - $t \in Z^n$ – вектор распределения времен, где t_i – количество раз, которое был запущен подграф i
  - g_i – минимальное полученное время на задаче i
  - $f(g_1(t), g_2(t), \dots, g_n(t))$ – время выполнения всего графа

- Постановка задачи, которая решается диспетчером задач:

$$\min \ f\big(g_1(t), g_2(t), \dots, g_n(t)\big)$$

Диспетчер задач. Схема решения задачи в частном случае

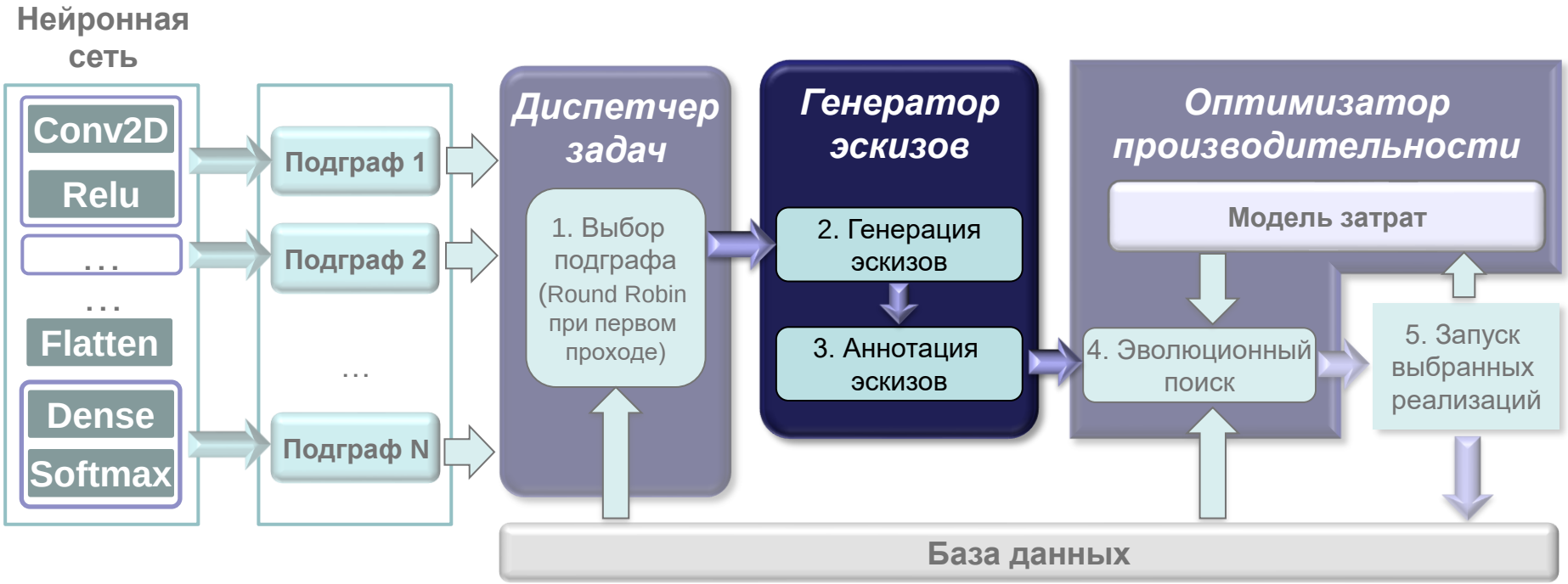
- Для простоты рассмотрим одну нейронную сеть с различными подграфами:

$$f\left(g_1(t), g_2(t), \dots, g_n(t)\right) = \sum_i g_i(t)$$

- В процессе оптимизации на текущем шаге выбирается подграф, который максимизирует модуль производной $f(g_1(t), g_2(t), \dots, g_n(t))$

$$i^* = \arg\max \left| \frac{df}{dt_i} \right|$$

Auto-scheduler. Схема работы



Генератор эскизов. Эскизы и аннотации

1. ***Создание эскизов*** – программных структур высокого уровня
   – Эскизы ***генерируются автоматически*** путем рекурсивного применения набора правил вывода к входному подграфу
   – Эскизы ***отражают структуру операторов***, ***но <u>не</u>*** определяют ***детали низкого уровня*** (размеры блоков и аннотации циклов)

2. ***Случайное аннотирование эскизов***
   – Созданные эскизы случайным образом аннотируются ***конкретными деталями низкого уровня*** (размеры блоков, распараллеливание, векторизация и развертывание циклов и прочее)

Генератор эскизов. Алгоритм генерации

1. Определим состояние $s = (S, i)$, где S – текущий эскиз, а i – индекс текущего узла в топологической сортировки подграфа
2. Процесс начинается с исходной наивной программы в виде эскиза S и индекса последнего узла i
3. Затем рекурсивно применяет набор правил вывода к текущему состоянию s для определения нового состояния $s' = (S', i')$
4. Правила применяются рекурсивно, при этом для каждого правила проверяется условие его применимости к текущему состоянию s
5. Процесс повторяется до тех пор, пока все узлы не будут обработаны ($i = 0$)

Генератор эскизов. Правила вывода…

- Некоторые правила вывода:

  Rule 1. Пропуск. Пропустить узел

  Rule 2. Встраивание (inline). Встроить узел

  Rule 3. Многоуровневое разбиение на блоки. Применить многоуровневое разбиение на блоки для повторного использования данных

  Rule 4. Многоуровневое разбиение на блоки с объединением. Разбиение на блоки и объединение с соседним узлом (например, с функцией активации)

  Rule 5. Добавление этапа кэширования

- ***Примечания:***
  - Для каждого правила имеется условие применимости, которое проверяется непосредственно перед его использованием
  - Имеется возможность добавления собственных правил вывода

Генератор эскизов. Правила вывода

- Правила 3, 4 и 5 касаются многоуровневого разбиения и слияния узлов подграфа, в которых данные используются повторно
- Правило 3 для ЦП использует структуру вида «SSRSRS», где «S» обозначает один уровень пространственных циклов по блокам, а «R» – один уровень цикла редукции
- Структура «SSRSRS» для матричного умножения преобразует исходный трехуровневый цикл с индексами (i, j, k), где i – номер строки, j – номер столбца, k – индекс элемента в скалярном произведении строки на столбец, в 10-уровневый цикл $(i_0, j_0, i_1, j_1, k_0, i_2, j_2, k_1, i_3, j_3)$

$$c_{i,j} = \sum a_{i,k} \cdot b_{k,j}$$

- ***Примечание:*** пространственный цикл – цикл, относящийся к данным (по i, j), цикл редукции – цикл по элементам в скалярном произведении (по k)

Генератор эскизов. Примеры эскизов…

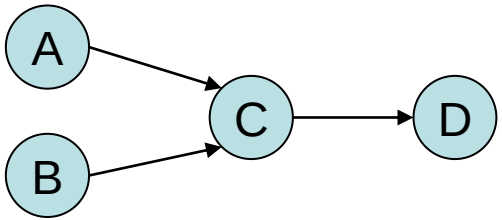
- Входная последовательность преобразований:

$$d_{i,j} = \max(0, c_{i,j})$$

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \cdot b_{k,j}$$

$$A = (a_{i,j}), B = (b_{i,j}) \in \mathbb{R}^{N \times N}$$

- Соответствующий граф:

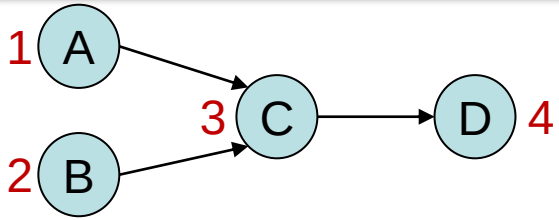


- Результат топологической сортировки вершин подграфа: (A, B, C, D)

Генератор эскизов. Примеры эскизов…

Программное описание подграфа

```
for i in range(512):
  for j in range(512):
   for k in range(512):
    C[i, j] += A[i, k] * B[k, j]
for i in range(512):
  for j in range(512):
    D[i, j] = max(C[i, j], 0.0)
```



Применяемые правила

$(S_0, i = 4)$
$\downarrow Rule\ 1 \downarrow$
$(S_1, i = 3)$
$\downarrow Rule\ 1 \downarrow$
$(S_2, i = 2)$
$\downarrow Rule\ 1 \downarrow$
$(S_3, i = 1)$
$\downarrow Rule\ 1 \downarrow$
эскиз

Сгенерированный эскиз (псевдокод)

```
for i0 in range(TILE_I0):
  for j0 in range(TILE_J0):
   for i1 in range(TILE_I1):
    for j1 in range(TILE_J1):
     for k0 in range(TILE_K0):
      for i2 in range(TILE_I2):
       for j2 in range(TILE_J2):
        for k1 in range(TILE_K1):
         for i3 in range(TILE_I3):
          for j3 in range(TILE_J3):
           C[...] += A[...] * B[...]
      for i4 in range(TILE_I2 * TILE_I3):
       for j4 in range(TILE_J2 * TILE_J3):
        D[...] = max(C[...], 0.0)
```

S_i – текущий эскиз, а i – индекс текущего узла в топологической сортировки подграфа

Генератор эскизов. Примеры эскизов…

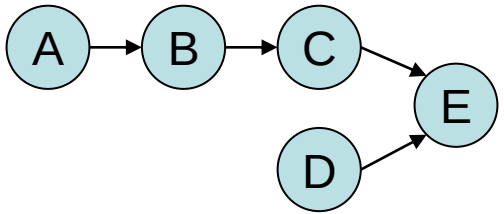
- Входная последовательность преобразований:

$$e_{i,j} = \sum_{k=1}^{N} c_{i,k} \cdot d_{k,j}, 0 \le i < 8, 0 \le j < 4$$

$$c_{i,k} = \begin{cases} b_{i,k}, & k < 400 \\ 0, & k \ge 400 \end{cases}, 0 \le k < 512$$

$$b_{i,l} = \max\{a_{i,l}, 0{,}0\}, 0 \le l < 400$$

- Соответствующий граф:

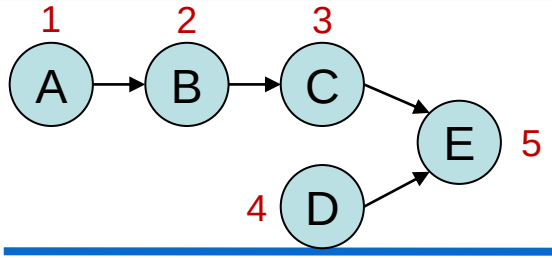


- Результат топологической сортировки вершин подграфа: (A,B,C,D,E)

Генератор эскизов. Примеры эскизов…

Программное описание подграфа

```
for i in range(8):
  for l in range(400):
    B[i, l] = max(A[i, l], 0.0)
    for i in range(8):
  for k in range(512):
    if k < 400 else 0
    C[i, k] = B[i, k]
for i in range(8):
  for j in range(4):
    for k in range(512):
      E[i, j] += C[i, k] * D[k, j]
```



Применяемые правила

$(S_0, i = 5)$
$\downarrow Rule\ 5 \downarrow$
$(S_1, i = 5)$
$\downarrow Rule\ 4 \downarrow$
$(S_2, i = 4)$
$\downarrow Rule\ 1 \downarrow$
$(S_3, i = 3)$
$\downarrow Rule\ 1 \downarrow$
$(S_4, i = 2)$
$\downarrow Rule\ 2 \downarrow$
$(S_5, i = 1)$
$\downarrow Rule\ 1 \downarrow$
эскиз

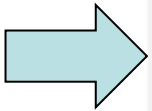
Сгенерированный эскиз (псевдокод)

```
for i in range(8):
  for k in range(512):
   C[i, j] = max(A[i,k], 0.0)if k<400 else 0
for i0 in range(TILE_I0):
  for j0 in range(TILE_J0):
   for i1 in range(TILE_I1):
    for j1 in range(TILE_J1):
     for k0 in range(TILE_K0):
      for i2 in range(TILE_I2):
       for j2 in range(TILE_J2):
        for k1 in range(TILE_K1):
         for i3 in range(TILE_I3):
          for j3 in range(TILE_J3):
           E_cache[...] += C[...] * D[...]
     for i4 in range(TILE_I2 * TILE_I3):
      for j4 in range(TILE_J2 * TILE_J3):
       E[...] = E_cache[...]
```

Генератор эскизов. Примеры аннотаций

Сгенерированный эскиз

```
for i0 in range(TILE_I0):
  for j0 in range(TILE_J0):
   for i1 in range(TILE_I1):
    for j1 in range(TILE_J1):
     for k0 in range(TILE_K0):
      for i2 in range(TILE_I2):
       for j2 in range(TILE_J2):
        for k1 in range(TILE_K1):
         for i3 in range(TILE_I3):
          for j3 in range(TILE_J3):
           C[...] += A[...] * B[...]
      for i4 in range(TILE_I2 * TILE_I3):
       for j4 in range(TILE_J2 * TILE_J3):
        D[...] = max(C[...], 0.0)
```



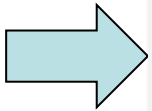
Аннотированные эскизы

```
parallel i0_j0_i1_j1 in range(256):
  for k.0 in range(32):
   for i.2 in range(16):
    unroll k.1 in range(16):
     unroll i.3 in range(4):
      vectorize j.3 in range(16):
       C[...] += A[...] * B[...]
  for i.4 in range(64):
   vectorize j.4 in range(16):
    D[...] = max(C[...], 0.0)
```

Генератор эскизов. Примеры аннотаций

Сгенерированный эскиз

```
for i0 in range(TILE_I0):
  for j0 in range(TILE_J0):
   for i1 in range(TILE_I1):
    for j1 in range(TILE_J1):
     for k0 in range(TILE_K0):
      for i2 in range(TILE_I2):
       for j2 in range(TILE_J2):
        for k1 in range(TILE_K1):
         for i3 in range(TILE_I3):
          for j3 in range(TILE_J3):
           C[...] += A[...] * B[...]
      for i4 in range(TILE_I2 * TILE_I3):
       for j4 in range(TILE_J2 * TILE_J3):
        D[...] = max(C[...], 0.0)
```



Аннотированные эскизы

```
parallel i2 in range(16):
  for j2 in range(128):
   for k1 in range(512):
    for i3 in range(32):
     vectorize j3 in range(4):
      C[...] += A[...] * B[...]
     parallel i4 in range(512):
  for j4 in range(512):
    D[...] = max(C[...], 0.0)
```

Auto-scheduler. Схема работы

Оптимизатор производительности

- Генерация и аннотация эскизов производится случайно, и количество всевозможных вариантов измеряется ***миллиардами***
- Для оптимизации используется процедура, основанная на эволюционном поиске и обучаемой модели затрат
  - Эволюционный поиск использует случайно выбранные программы, а также высококачественные программы из предыдущего измерения в качестве начальной популяции
  - Запускается эволюционный алгоритм со случайным применением скрещивания и мутаций для генерации следующего поколения
  - Вероятность выбора реализации на каждом шаге алгоритма пропорциональна ее качеству, которое предсказывается моделью затрат
  - После того как эволюционный алгоритм останавливается сгенерированная партия проверяется на целевом устройстве

Оптимизатор производительности. Эволюционный поиск. Мутации и скрещивания

- Случайная мутация коэффициента разделения цикла
- Случайные мутации распараллеливания, развертывания и векторизации циклов
- Случайная мутация местоположения вычислений:
  - Для узлов, которые не предполагают применение блочного алгоритма, положение вычислений меняется случайным образом (например, для функции активации после сверточного слоя)
- Скрещивание узлов подграфа:
  - Генерируется новая реализация, для которой правила генерации эскизов случайно выбирается из двух существующий реализаций

Оптимизатор производительности. Модель затрат

- Модель затрат – ***градиентный бустинг деревьев*** (Gradient Boosting Trees)
  - ***Вход*** – признаковое описание эксперимента
  - ***Выход*** – количественный показатель, отражающий производительность реализации
  - Модель обучается заново при каждом обновлении базы данных экспериментов
  - ***Функция потерь*** – взвешенная квадратичная ошибка (экспериментам, которые работают быстро, придается больший вес)
  - Примеры ***признаков*** для описания эксперимента:
    - Количество целочисленных и вещественных операций
    - Информация о возможности векторизации, развертывания, распараллеливания цикла (например, длина цикла)
    - Арифметическая интенсивность – отношение количества операций с плавающей точкой к числу байт памяти, которое необходимо загрузить или сохранить
    - Информацию о доступе к памяти (количество задействованных байт и количество задействованных кэш-линий, шаг доступа, количество переиспользований памяти)

Общий алгоритм работы Auto-scheduler

- Диспетчер задач начинает работу с вектора распределения времен $t = (0, 0, ..., 0)$ (ни один подграф не запускался) и использует алгоритм Round Robin для получения начальных значений времен
- Действия выполняются до достижения допустимого количества итераций:
  - Выбор подграфа для оптимизации с учетом эпсилон-жадной стратегии (epsilon-greedy algorithm)
    - Выбор рекомендованного алгоритмом подграфа с вероятностью $1 - \varepsilon$
    - Выбор случайного подграфа с вероятностью ε
  - Генерация новой пачки реализаций с помощью генерации и аннотации эскизов
  - Эволюционный поиск для выбора оптимальных реализаций из множества, построенных на предыдущем шаге, и лучших протестированных с учетом модели затрат
  - Запуск на целевом устройстве пачки реализаций
  - Обновление модели затрат

META-SCHEDULER

MetaSchedule

- ***MetaSchedule*** – третье поколение подходов для автоматической оптимизации нейросетевых преобразований, разработанное в 2022 году в Apache TVM
- ***Цель*** – разработать модульную систему, которая позволит ***объединить возможности использования других методов оптимизации слоев***
  - Заданный пользователем план
  - Оптимизация с помощью шаблонов (AutoTVM)
  - Оптимизация с помощью правил (Auto-scheduler)
- Внутри MetaSchedule заложен проблемно-ориентированный *вероятностный* язык для описания вариантов оптимизаций

* Shao J., et al. Tensor program optimization with probabilistic programs // Advances in Neural Information Processing Systems. – 2022. – Т. 35. – С. 35783-35796.

Обобщение ранних подходов

- MetaSchedule (также как и Auto-scheduler) принимает на вход тензорное выражение и генерирует TIR
- Данный подход является общим случаем ранних подходов тюнинга:
  - ***План, заданный пользователем.*** План вычислений — это особый случай программы MetaSchedule, где нет случайности. Оптимизации задаются по аналогии с AutoTVM
  - ***Оптимизация на основе шаблонов (AutoTVM).*** Это естественное представление шаблонов расписания AutoTVM. С помощью случайных переменных реализуются различные варианты реализаций плана
  - ***Оптимизация на основе правил вывода (Auto-scheduler).*** MetaSchedule также имеет набор правил и методов, позволяет автоматически генерировать различные реализации по аналогии с Auto-scheduler
- Решение, какой подход использовать, принимает пользователь

Обобщение ранних подходов

- MetaSchedule работает схожим образом с AutoTVM в режиме оптимизации на основе шаблонов и Auto-scheduler в режиме оптимизации без шаблонов
- MetaSchedule заимствует подходы из предыдущих методов, интегрируя лучшие идеи и учитывая прошлые технические ошибки

<table>
<tr><th>План, разработанный пользователем</th><th>Оптимизация на основе шаблона</th><th>Оптимизации на основе правил вывода</th></tr>
<tr><td><pre># коэффициенты разделения циклов
i_tiles = [16, 8, 8, 8]
j_tiles = [16, 8, 8, 8]
k_tiles = [256, 8]</pre></td><td><pre># генерация переменной для разделения циклов
i_tiles = sch.sample_perfect_tile(i, n=4)
j_tiles = sch.sample_perfect_tile(j, n=4)
k_tiles = sch.sample_perfect_tile(k, n=2)</pre></td><td rowspan="3"><pre>sch = tir.Schedule(func)
C = sch.get_block('C')

rule = MultiLevelTiling('SSRSRS')

sch = rule.apply(sch, C)[0]</pre></td></tr>
<tr><td colspan="2"><pre># разделение циклов по коэффициентам
i_0, i_1, i_2, i_3 = sch.split(loop=i, factors=i_tiles)
j_0, j_1, j_2, j_3 = sch.split(loop=j, factors=j_tiles)
k_0, k_1 = sch.split(loop=k, factors=k_tiles)</pre></td></tr>
<tr><td colspan="2"><pre>sch.reorder(...)</pre></td></tr>
</table>

Пользовательские оптимизации

- ***Пример*** – оптимизация алгоритма умножения матриц

- Для оптимизации используется ***разделение циклов***:
  - Разделим первый и второй цикл каждый на 4 цикла с фиксированным размером каждого цикла
  - Разделим внутренний цикл редукции на 2 цикла фиксированной длины

Исходный псевдокод

```
for i in range(N):,
    for j in range(N):
        for k in range(N):
            c[i, j] += a[i, k] * b[k, j]
```

Преобразования

```
i_0, i_1, i_2, i_3 = Split(loop=i, factors=[16, 8, 8, 8])
j_0, j_1, j_2, j_3 = Split(loop=j, factors=[16, 8, 8, 8])
k_0, k_1 = Split (loop=k, factors=[256, 8])
# в результате новая реализация умножения матриц через 10 циклов
```

Оптимизации на основе шаблонов

- Зададим коэффициенты разделения циклов с помощью ***случайных величин***
- При каждом запуске генерируется новое допустимое значение для коэффициентов разделения циклов

Исходный псевдокод

```
for i in range(N):,
    for j in range(N):
        for k in range(N):
            c[i, j] += a[i, k] * b[k, j]
```

Преобразования

```
i_tiles = sch.sample_perfect_tile(i, n=4)
j_tiles = sch.sample_perfect_tile(j, n=4)
k_tiles = sch.sample_perfect_tile(k, n=2)

i_0, i_1, i_2, i_3 = Split(loop=i, factors=i_tiles)
j_0, j_1, j_2, j_3 = Split(loop=j, factors=j_tiles)
k_0, k_1 = Split(loop=k, factors=k_tiles)
# в результате новая реализация умножения матриц через 10 циклов
```

Оптимизация на основе правил вывода

- В режиме оптимизации с помощью правил можно указать ***структуру циклов***
- В данном случае можно применить правило генерации блочного алгоритма с использованием шаблонов 'SSRSRS‘
- Автоматически выполняется аннотация новых циклов с помощью случайных величин

Исходный псевдокод

```
Block('C')
for i in range(N):,
    for j in range(N):
        for k in range(N):
            c[i, j] += a[i, k] * b[k, j]
```

Преобразования

```
C = sch.get_block('C')
rule = MultiLevelTiling('SSRSRS')
sch = rule.apply(sch, C)

# в результате новая реализация умножения матриц через 10 циклов
```

Демонстрация

- Демонстрация использования правил вывода для MetaSchedule
  lectures/sources/07_MultiLevelTiling_rule.ipynb

ЗАКЛЮЧЕНИЕ

Краткая информация о методах оптимизации слоев

	AutoTVM	Auto-scheduler	MetaSchedule
Входная информация	Множество планов – шаблоны	Тензорное выражение	
Принцип работы	Поиск оптимального плана вычислений среди описанных вариантов	Поиск оптимальной реализации с помощью набора правил и эволюционного алгоритма	Поиск оптимального плана вычислений среди описанных вариантов
Выходная информация	Оптимальный план вычислений	Оптимизированный TIR	

Заключение…

- Apache TVM содержит несколько подходов для автоматической оптимизации скорости работы нейросетевых преобразований под конкретное целевое устройство
  - ***Оптимизация на основании шаблонов*** – заранее реализованных вариантов планов исполнения операторов нейронной сети
  - ***Оптимизация на основании правил вывода*** для реализации преобразований
  - ***Гибридный подход***

Заключение

- ***Автоматическая оптимизация нейросетевых преобразований – это новый параметр для настройки модели!*** Возможности оптимизации слоев могут зависеть от качества базовой реализации, уровня оптимизации графа, формата хранения данных и прочего
- Для определения наилучших параметров требуется ***проведение большого числа испытаний***
- Возможно, что в будущем ***будут собраны большие базы знаний, специфичные для области тензорных компиляторов***, которые позволят за малое число испытаний находить хорошую конфигурацию запуска с помощью переноса знаний
- ***Методы автоматической оптимизации продолжают развиваться!***

Литература

1. Chen T., et al. Learning to optimize tensor programs // Advances in Neural Information Processing Systems. – 2018. – T. 31.
2. Zheng L., et al. Ansor: Generating High-Performance tensor programs for deep learning // 14th USENIX symposium on operating systems design and implementation (OSDI 20). – 2020. – P. 863-879.
3. Shao J., et al. Tensor program optimization with probabilistic programs // Advances in Neural Information Processing Systems. – 2022. – T. 35. – P. 35783-35796.

Авторский коллектив

- Мееров Иосиф Борисович, к.т.н., доцент, зав. каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Кустикова Валентина Дмитриевна, к.т.н., доцент каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Родимков Юрий Александрович, младший научный сотрудник каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Сысоев Александр Владимирович, к.т.н., доцент каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского