СБЕР

Автоматическая оптимизация ML используя Apache TVM Домашнее задание

Алексей Воронов

Руководитель направления в группе по разработке искусственного интеллекта

Задание

1. Взять/написать вычисление(matmul, conv или т.д.)
2. Превратить в шаблон и добавить трансформации
3. Выбрать некоторое исполняемое устройство(допустим CPU)
4. Уменьшить пространство поиска максимально сохранив производительность и уменьшить количество итераций/время тюнинга
5. Постараться объяснить как соотносятся трансформации, фильтры и архитектура устройства при полученном результате

- avoronov.icemist@gmail.com
- https://github.com/Icemist/neimark-it.tvm

Матричное умножение

```
def matmul(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A")
    B = te.placeholder((K, N), name="B")
    C = te.compute((M, N), lambda m, n:
        te.sum(A[m, k] * B[k, n], axis=k), name="C")
    s = te.create_schedule(C.op)
    return s, [A, B, C]
```

lower →

```
@main = primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"from_legacy_te_schedule": True, "global_symbol": "main",
"tir.noalias": True}
  buffers = {A: Buffer(A_2: Pointer(float32), float32, [1048576], []),
             B: Buffer(B_2: Pointer(float32), float32, [1048576], []),
             C: Buffer(C_2: Pointer(float32), float32, [1048576], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C}
  preflattened_buffer_map = {A_1: A_3: Buffer(A_2, float32, [1024, 1024], []),
B_1: B_3: Buffer(B_2, float32, [1024, 1024], []), C_1: C_3: Buffer(C_2, float32,
[1024, 1024], [])} {
  for (m: int32, 0, 1024) {
    for (n: int32, 0, 1024) {
      C[((m*1024) + n)] = 0f32
      for (k: int32, 0, 1024) {
        let cse_var_2: int32 = (m*1024)
        let cse_var_1: int32 = (cse_var_2 + n)
        C[cse_var_1] = (C[cse_var_1] + (A[(cse_var_2 + k)]*B[((k*1024) + n)]))
      }
    }
  }
}
```

Матричное умножение с трансформациями

```
def matmul_transforms(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A")
    B = te.placeholder((K, N), name="B")
    C = te.compute((M, N), lambda m, n:
        te.sum(A[m, k] * B[k, n], axis=k), name="C")
    s = te.create_schedule(C.op)
    # schedule
    m, n = C.op.axis
    mo, mi= s[C].split(m, factor=4)
    no, ni= s[C].split(n, factor=4)
    return s, [A, B, C]
```

lower

```
@main = primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"from_legacy_te_schedule": True, "global_symbol": "main",
"tir.noalias": True}
  buffers = {A: Buffer(A_2: Pointer(float32), float32, [1048576], []),  B:
Buffer(B_2: Pointer(float32), float32, [1048576], []),  C: Buffer(C_2:
Pointer(float32), float32, [1048576], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C}
  preflattened_buffer_map = {A_1: A_3: Buffer(A_2, float32, [1024, 1024], []),
B_1: B_3: Buffer(B_2, float32, [1024, 1024], []), C_1: C_3: Buffer(C_2, float32,
[1024, 1024], [])} {
  for (m.outer: int32, 0, 256) {
    for (m.inner: int32, 0, 4) {
      for (n.outer: int32, 0, 256) {
        for (n.inner: int32, 0, 4) {
          C[((((m.outer*4096) + (m.inner*1024)) + (n.outer*4)) + n.inner)] = 0f32
          for (k: int32, 0, 1024) {
            let cse_var_3: int32 = (n.outer*4)
            let cse_var_2: int32 = ((m.outer*4096) + (m.inner*1024))
            let cse_var_1: int32 = ((cse_var_2 + cse_var_3) + n.inner)
            C[cse_var_1] = (C[cse_var_1] + (A[(cse_var_2 + k)]*B[(((k*1024) +
cse_var_3) + n.inner)]))
          }      }     }   } }}
```

Шаблон матричного умножения

```
@autotvm.template("matmul_template")
def matmul_template(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A")
    B = te.placeholder((K, N), name="B")
    C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n],
axis=k), name="C")
    s = te.create_schedule(C.op)
    # schedule
    m, n = C.op.axis
    # config
    cfg = autotvm.get_config()
    candidates = [[8, 128], [16, 64], [32, 32], [64, 16], [128, 8]]
    cfg.define_split("tile_x", m, num_outputs=2, policy="candidate",
candidate=candidates)
    cfg.define_split("tile_y", n, num_outputs=2, policy="candidate",
candidate=candidates)
    return s, [A, B, C]
```

config_space →

```
ConfigSpace (len=25, range_length=25, space_map=
   0 tile_x: Split(policy=candidate, product=1024,
num_outputs=2) len=5
   1 tile_y: Split(policy=candidate, product=1024,
num_outputs=2) len=5
)
[('tile_x', [8, 128]), ('tile_y', [8, 128])],None,0
[('tile_x', [16, 64]), ('tile_y', [8, 128])],None,1
[('tile_x', [32, 32]), ('tile_y', [8, 128])],None,2
[('tile_x', [64, 16]), ('tile_y', [8, 128])],None,3
[('tile_x', [128, 8]), ('tile_y', [8, 128])],None,4
…
[('tile_x', [64, 16]), ('tile_y', [64, 16])],None,18
[('tile_x', [128, 8]), ('tile_y', [64, 16])],None,19
[('tile_x', [8, 128]), ('tile_y', [128, 8])],None,20
[('tile_x', [16, 64]), ('tile_y', [128, 8])],None,21
[('tile_x', [32, 32]), ('tile_y', [128, 8])],None,22
[('tile_x', [64, 16]), ('tile_y', [128, 8])],None,23
[('tile_x', [128, 8]), ('tile_y', [128, 8])],None,24
```

Шаблон матричного умножения с фильтром

```
@autotvm.template("matmul_template_with_filter")
def matmul_template_with_filter(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A")
    B = te.placeholder((K, N), name="B")
    C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n],
axis=k), name="C")
    s = te.create_schedule(C.op)
    # schedule
    m, n = C.op.axis
    # config
    cfg = autotvm.get_config()
    filter = lambda v: v.size[0] != 16 and v.size[1] != 16
    candidates = [[8, 128], [16, 64], [32, 32], [64, 16], [128, 8]]
    cfg.define_split("tile_x", m, num_outputs=2, policy="candidate",
        candidate=candidates, filter=filter)
    cfg.define_split("tile_y", n, num_outputs=2, policy="candidate",
        candidate=candidates, filter=filter)
    return s, [A, B, C]
```

config_space →

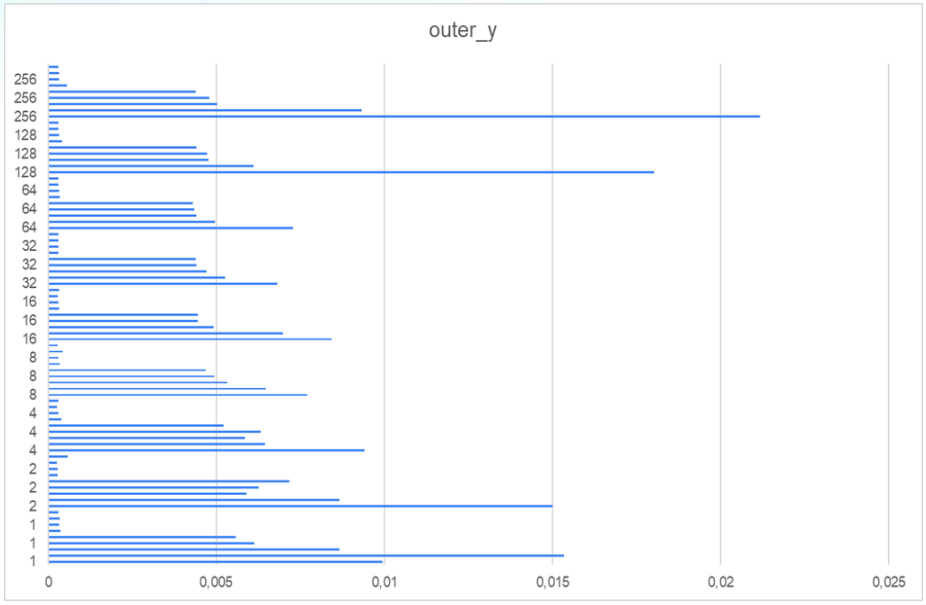
```
ConfigSpace (len=9, range_length=9, space_map=
   0 tile_x: Split(policy=candidate, product=1024,
num_outputs=2) len=3
   1 tile_y: Split(policy=candidate, product=1024,
num_outputs=2) len=3
)
[('tile_x', [8, 128]), ('tile_y', [8, 128])],None,0
[('tile_x', [32, 32]), ('tile_y', [8, 128])],None,1
[('tile_x', [128, 8]), ('tile_y', [8, 128])],None,2
[('tile_x', [8, 128]), ('tile_y', [32, 32])],None,3
[('tile_x', [32, 32]), ('tile_y', [32, 32])],None,4
[('tile_x', [128, 8]), ('tile_y', [32, 32])],None,5
[('tile_x', [8, 128]), ('tile_y', [128, 8])],None,6
[('tile_x', [32, 32]), ('tile_y', [128, 8])],None,7
[('tile_x', [128, 8]), ('tile_y', [128, 8])],None,8
```

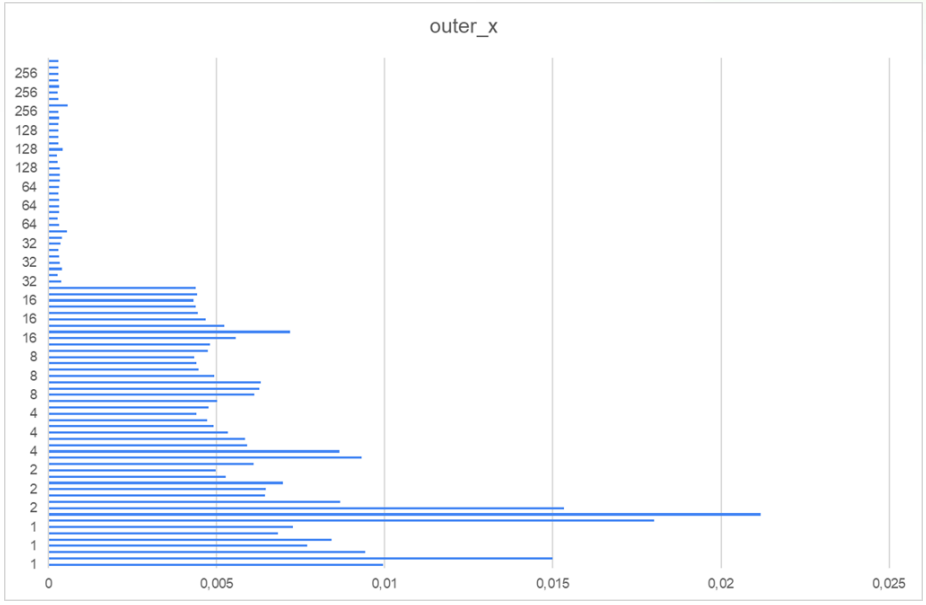
Пример без фильтра

```
@autotvm.template("matmul_template_wo_filter")
def matmul_template_wo_filter(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A", dtype=dtype)
    B = te.placeholder((K, N), name="B", dtype=dtype)
    C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k),
name="C")
    s = te.create_schedule(C.op)
    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]
    ##### define space #####
    cfg = autotvm.get_config()
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2)
    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)
    s[C].reorder(yo, xo, k, yi, xi)
    return s, [A, B, C]
```

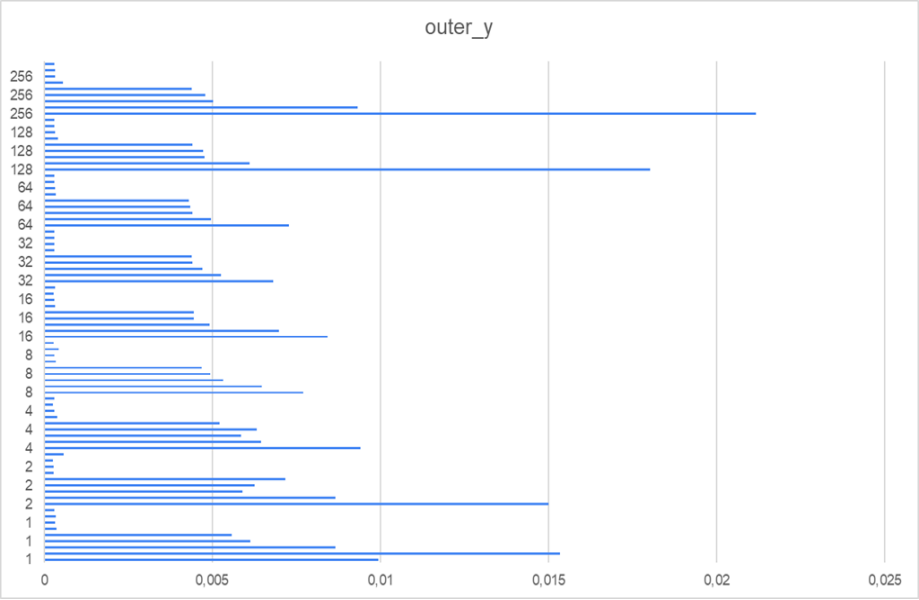
```
len 81
[Task  1/ 1]
Current/Best:
4.69/ 133.43 GFLOPS
Progress: (81/81)
51.07 s Done.
```

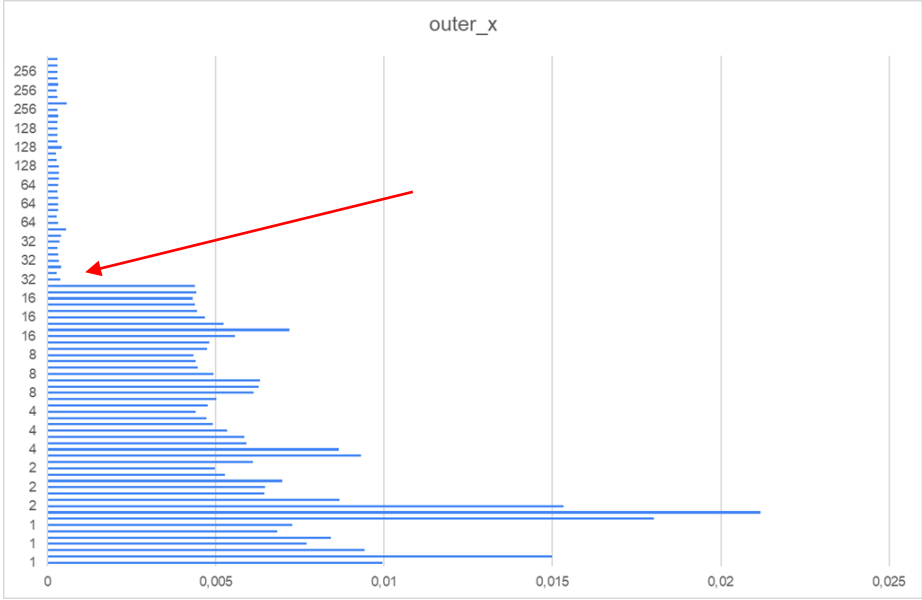
Пример без фильтра





Пример без фильтра



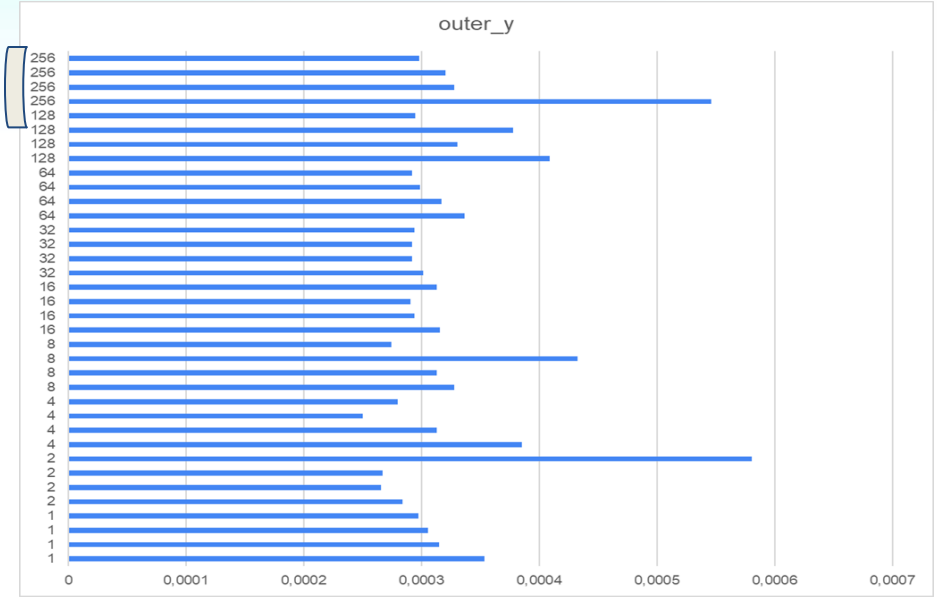


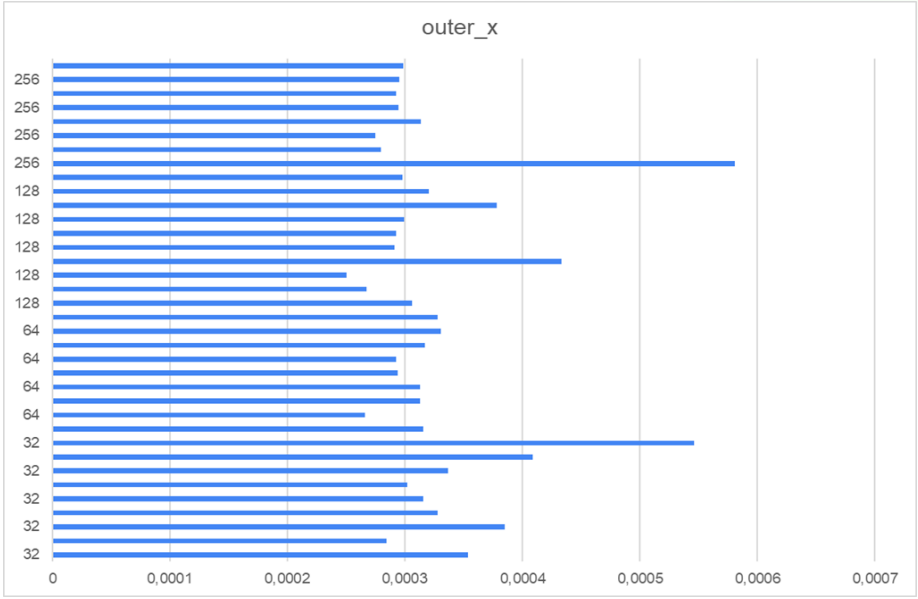
Пример с фильтром

```
@autotvm.template("matmul_template_with_filter")
def matmul_template_with_filter(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A", dtype=dtype)
    B = te.placeholder((K, N), name="B", dtype=dtype)
    C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k), name="C")
    s = te.create_schedule(C.op)
    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]
    ##### define space #####
    cfg = autotvm.get_config()
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2, filter= lambda v: v.size[1] >= 32)
    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)
    s[C].reorder(yo, xo, k, yi, xi)
    return s, [A, B, C]
```

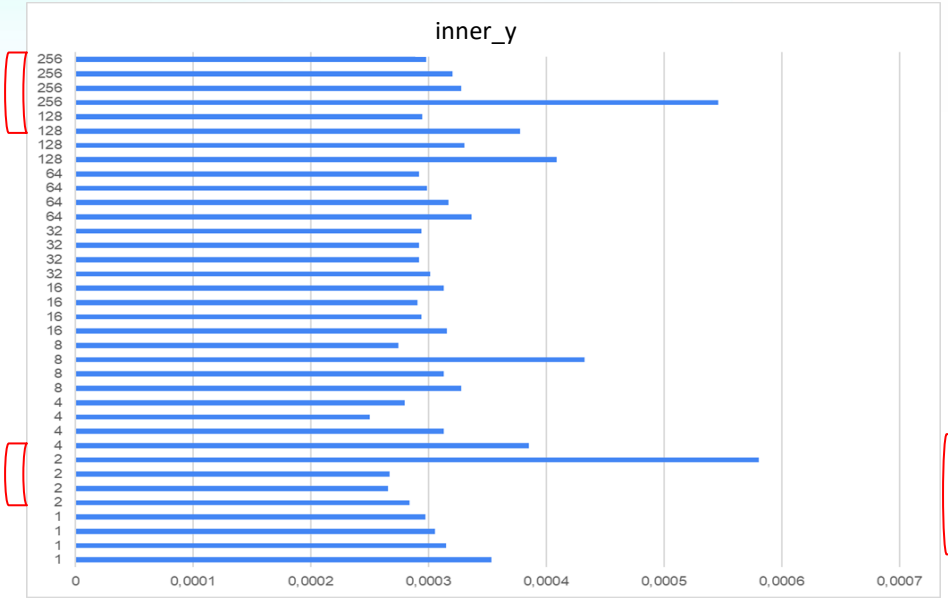
```
len 36
[Task  1/ 1]
Current/Best:
107.27/ 133.98 GFLOPS
Progress: (36/36)
10.73 s Done.
```

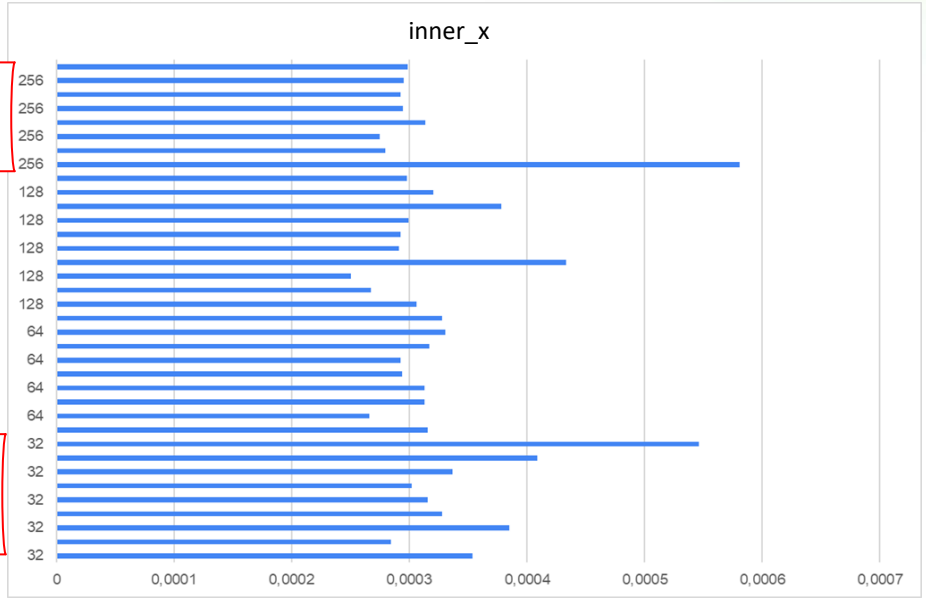
Пример с фильтром





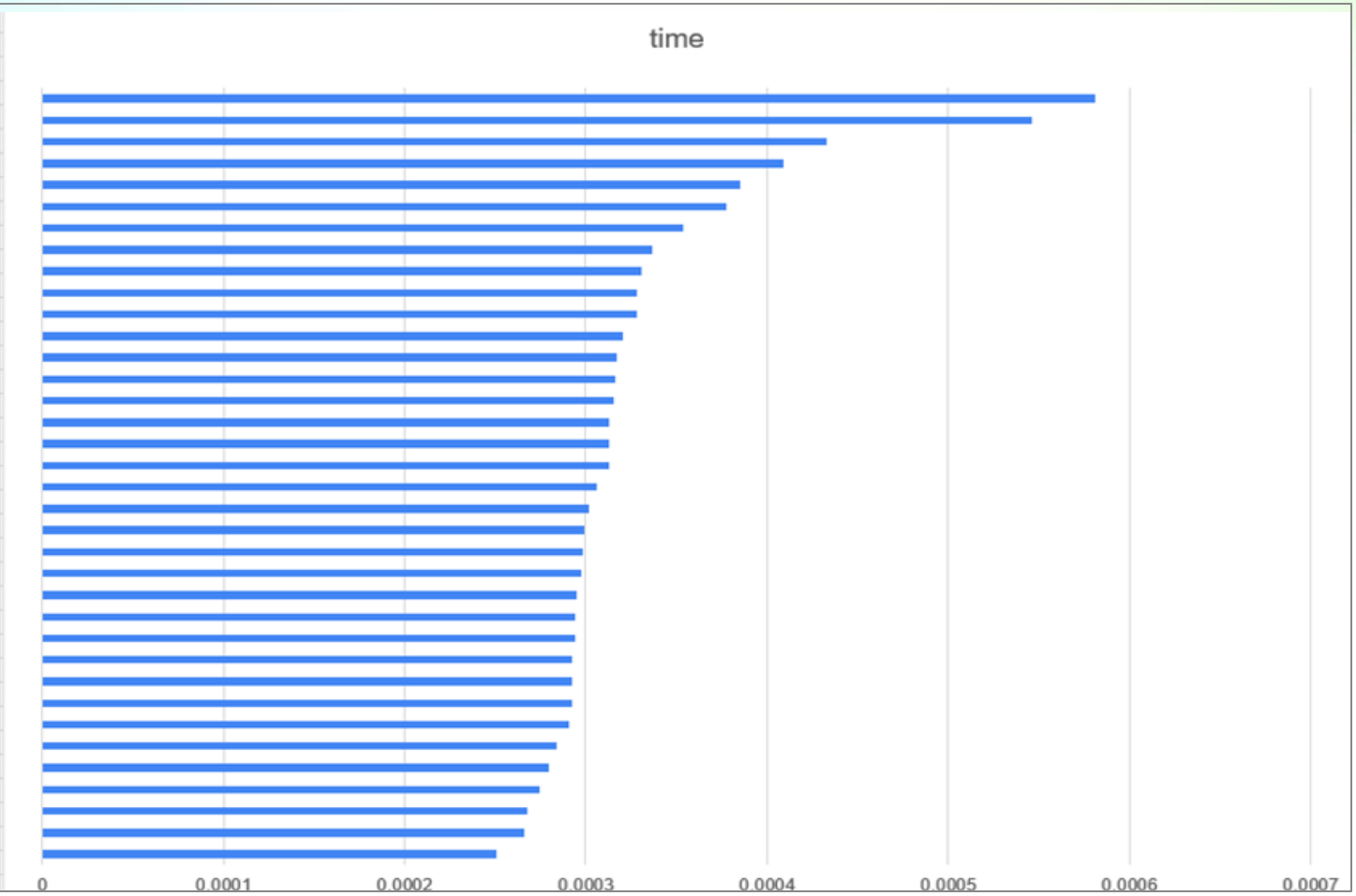
Пример с фильтром





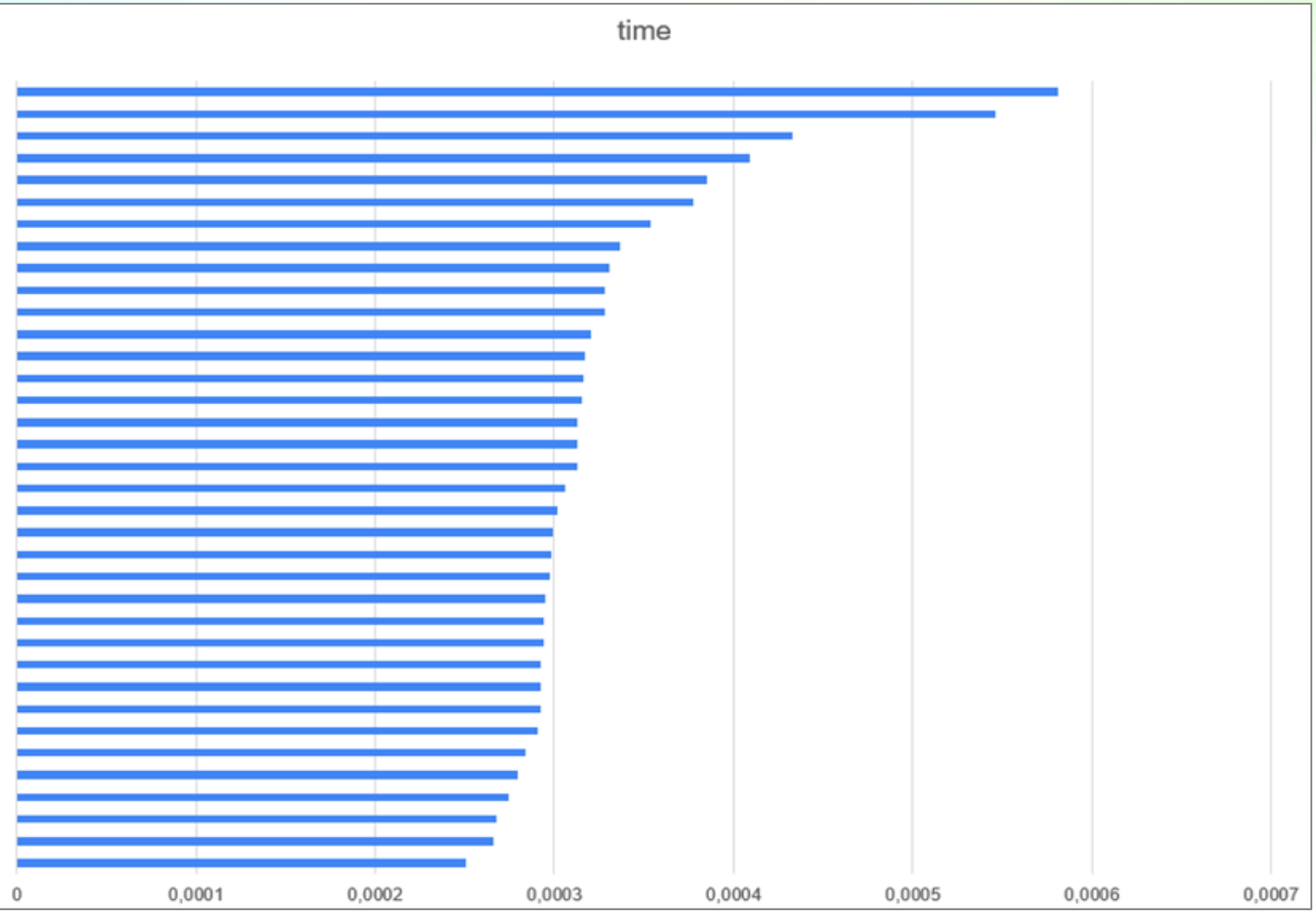
Пример с фильтром

time	outer_y	outer_x
0,00025	4	128
0,000266	2	64
0,000267	2	128
0,000275	8	256
0,00028	4	256
0,000284	2	32
0,000291	16	128
0,000292	32	128
0,000292	32	64
0,000292	64	256
0,000294	16	64
0,000294	32	256
0,000295	128	256
0,000298	1	256
0,000298	256	256
0,000299	64	128
0,000302	32	32
0,000306	1	128
0,000313	8	64
0,000313	4	64
0,000313	16	256
0,000315	1	64
0,000316	16	32
0,000317	64	64
0,000321	256	128
0,000328	8	32
0,000328	256	64
0,000331	128	64
0,000336	64	32
0,000354	1	32
0,000378	128	128
0,000385	4	32
0,000409	128	32
0,000433	8	128
0,000546	256	32
0,000581	2	256



Пример с фильтром

time	outer_y	outer_x
0,00025	4	128
0,000266	2	64
0,000267	2	128
0,000275	8	256
0,00028	4	256
0,000284	2	32
0,000291	16	128
0,000292	32	128
0,000292	32	64
0,000292	64	256
0,000294	16	64
0,000294	32	256
0,000295	128	256
0,000298	1	256
0,000298	256	256
0,000299	64	128
0,000302	32	32
0,000306	1	128
0,000313	8	64
0,000313	4	64
0,000313	16	256
0,000315	1	64
0,000316	16	32
0,000317	64	64
0,000321	256	128
0,000328	8	32
0,000328	256	64
0,000331	128	64
0,000336	64	32
0,000354	1	32
0,000378	128	128
0,000385	4	32
0,000409	128	32
0,000433	8	128
0,000546	256	32
0,000581	2	256

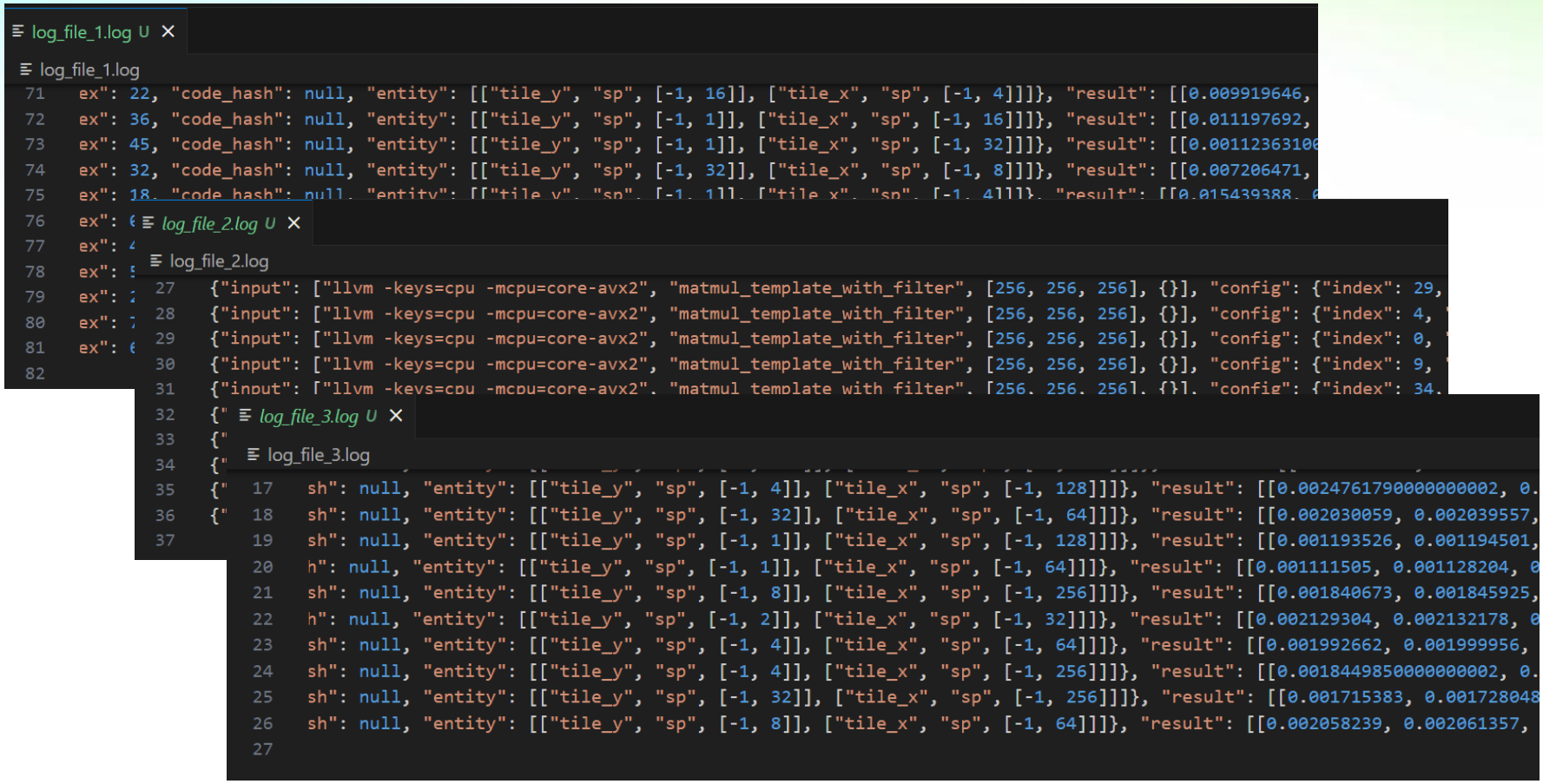


Пример с фильтром

```
@autotvm.template("matmul_template_with_multi_filter")
def matmul_template_with_multi_filter(N, M, K):
    k = te.reduce_axis((0, K), "k")
    A = te.placeholder((M, K), name="A", dtype=dtype)
    B = te.placeholder((K, N), name="B", dtype=dtype)
    C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k), name="C")
    s = te.create_schedule(C.op)
    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]
    ##### define space #####
    cfg = autotvm.get_config()
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2, filter= lambda v: v.size[1] >= 32)
    cfg.multi_filter(filter=lambda e: e["tile_x"].size[1] > e["tile_y"].size[1])
    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)
    s[C].reorder(yo, xo, k, yi, xi)
    return s, [A, B, C]
```

```
len 26
[Task  1/ 1]
Current/Best:
119.19/ 133.60 GFLOPS
Progress: (26/26)
8.35 s Done.
```

Логи тюнинга



Реальный случай

Данный фильтр был добавлен для Adreno GPU

topi.adreno:

- schedule_conv2d_NHWC
- schedule_conv2d_NCHWc_KCRSk
- schedule_conv2d_winograd
- schedule_depthwise_conv2d_NHWC_HWOI
- schedule_depthwise_conv2d_NCHWc_KCRSk

Например на mace_inceptionv3 моделе где на каждое раписание было 333 попытки:

- Среднее GFLOPS раписаний увеличился на 3%
- Общее время настройки расписаний уменьшилось на 18% (с 686 до 577 минут)
- На Adreno несколько тяжелых расписаний подряд (до 1 секунды) приводили к перезагрузке устройства. Этого удалось избежать.

Теория

Процессор AMD Ryzen 7 5800H представляет собой мобильный процессор, который предлагает высокую производительность для ноутбуков. Вот его основные характеристики:

- **Уровни кэш-памяти**:
  - Уровень 1 кэша (L1): 512 КБ. L1 кэш обычно разделяется между инструкциями и данными.
  - Уровень 2 кэша (L2): 4 МБ. L2 кэш используется для хранения данных и инструкций и обеспечивает более быстрый доступ к данным, чем уровень 3 кэша.
  - Уровень 3 кэша (L3): 16 МБ. L3 кэш представляет собой кэш большего размера.
- **Поддержка AVX2**: Процессор поддерживает набор команд AVX2 (Advanced Vector Extensions 2), который предоставляет расширенные инструкции SIMD (Single Instruction, Multiple Data.

Теория

Для того чтобы максимально эффективно использовать кэш размером 512 КБ, можно разделить циклы так, чтобы минимизировать количество кэш-промахов и максимально использовать локальность данных. Поскольку размер кэша ограничен, эффективное использование его пространства очень важно для обеспечения высокой производительности.

Проанализируем циклы:

Внешние циклы m и n итерируются от 0 до 256.

Внутренний цикл k также итерируется от 0 до 256.

Для оптимального использования кэша можно использовать технику разделения на блоки. Мы можем разделить матрицы A, B и C на блоки и выполнить вычисления на каждом блоке. При этом внутренние циклы будут итерироваться по блокам вместо всей матрицы

Для оптимизации матричного умножения на процессоре AMD Ryzen 7 5800H, можно использовать различные оптимизации разделения (split) блоков tile_y и tile_x

Теория

```
block_size = 32  # Выбираем размер блока

for (m_outer: int32, 0, 256, block_size) {
  for (n_outer: int32, 0, 256, block_size) {
    for (k_outer: int32, 0, 256, block_size) {
      for (m_inner: int32, 0, block_size) {
        for (n_inner: int32, 0, block_size) {
          for (k_inner: int32, 0, block_size) {
            let m_idx: int32 = m_outer + m_inner
            let n_idx: int32 = n_outer + n_inner
            let k_idx: int32 = k_outer + k_inner
            C[(m_idx * 256) + n_idx] = (C[(m_idx * 256) + n_idx] +
                                        (A[(m_idx * 256) + k_idx] * B[(k_idx * 256) + n_idx]))
          }
        }
      }
    }
  }
}
```

Теория

```
import tvm
from tvm import te
# Задаем размеры матриц
M,N,K = 256, 256, 256
# Объявляем тензоры для входных данных и выхода
A = te.placeholder((M, K), name="A", dtype="float32")
B = te.placeholder((K, N), name="B", dtype="float32")
k = te.reduce_axis((0, K), "k")
C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k), name="C")
# Создаем расписание
s = te.create_schedule(C.op)
# Разделение блоков для оптимизации кэша
block_size = 32
mo, mi = s[C].split(C.op.axis[0], factor=block_size)
no, ni = s[C].split(C.op.axis[1], factor=block_size)
ko, ki = s[C].split(k, factor=block_size)
s[C].reorder(mo, no, ko, mi, ni, ki)
# Собираем и оптимизируем программу
print(tvm.lower(s, [A, B, C]))
```

Теория

```
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(A: T.Buffer((256, 256), "float32"), B: T.Buffer((256, 256),
    "float32"), C: T.Buffer((256, 256), "float32")):
        T.func_attr({"from_legacy_te_schedule": T.bool(True), "global_symbol":
        "main", "tir.noalias": T.bool(True)})
        for m_outer, n_outer in T.grid(8, 8):
            C_1 = T.Buffer((65536,), data=C.data)
            for m_inner_init, n_inner_init in T.grid(32, 32):
                C_1[m_outer * 8192 + m_inner_init * 256 + n_outer * 32 +
                n_inner_init] = T.float32(0)
            for k_outer, m_inner, n_inner, k_inner in T.grid(8, 32, 32, 32):
                cse_var_3: T.int32 = n_outer * 32
                cse_var_2: T.int32 = m_outer * 8192 + m_inner * 256
                cse_var_1: T.int32 = cse_var_2 + cse_var_3 + n_inner
                A_1 = T.Buffer((65536,), data=A.data)
                B_1 = T.Buffer((65536,), data=B.data)
                C_1[cse_var_1] = C_1[cse_var_1] + A_1[cse_var_2 + k_outer * 32
                + k_inner] * B_1[k_outer * 8192 + k_inner * 256 + cse_var_3 +
                n_inner]
```

Теория

- Для перемножения двух матриц размером 256×256 (две матрицы размером 256×256 и одна матрица-результат такого же размера) :
- **Матрицы *A* и *B***:
  - Размер каждой матрицы составляет 256×256.
  - Каждый элемент матрицы имеет тип float32, который занимает 4 байта (32 бита).
  - Таким образом, общий размер каждой матрицы составляет 256×256×4 байт.
- **Матрица *C* (результат умножения)**:
  - Размер матрицы-результата также 256×256.
  - Каждый элемент матрицы имеет тип float32, который занимает 4 байта (32 бита).
  - Таким образом, общий размер матрицы-результата также составляет 256×256×4 байт.
- **Общее количество памяти**:
  - Для хранения матриц *A* и *B* требуется 256×256×4×2 байт.
  - Для хранения матрицы-результата *C* требуется еще 256×256×4 байт.
  - Общее количество памяти, необходимое для перемножения двух матриц 256×256, составляет 256×256×4×3 байт.
- **Итого: 786432 байт**

Теория

- Для вычисления общего объема данных, который будет использоваться в кэше за одну итерацию внешних циклов **при блочном вычислении**, мы должны учитывать размеры данных, которые будут обрабатываться в каждом блоке внешних циклов.
- **Матрицы A и B**:
  - Каждая матрица имеет размер 256x256 элементов.
  - Тип данных float32, каждый элемент занимает 4 байта.
  - Общий объем данных для каждой матрицы: 256×256×4байт = 524288.
- **Блок матрицы C**:
  - Размер блока - 32x32 элемента.
  - Тип данных float32, каждый элемент занимает 4 байта.
  - Общий объем данных для блока: 32×32×4байт = 4096.
- **Количество блоков**:
  - Для каждой матрицы мы разбиваем размер на блоки размером 32x32.
  - Общее количество блоков в каждом измерении: 256/32=8.
- **Итого**: **528384 байт**

Теория

512 килобайт L1 кеша равны **524288** байтам, что меньше **528384** при блоке равном 32.

Чтобы рассчитать количество кэш-промахов в данном коде, нужно учитывать, что кэш-промах происходит, когда данные, необходимые для выполнения операции, отсутствуют в кэше и приходится обращаться к памяти. Количество кэш-промахов зависит от того, как данные используются в алгоритме, и как они размещены в памяти.

В коде происходит операция матричного умножения, и для каждой итерации внешних циклов (матрицы A, B и C) нам необходимо получить данные из памяти. Поскольку блоки данных кэшируются и повторно используются, нам нужно знать, какие блоки данных будут использоваться на каждой итерации, чтобы определить количество кэш-промахов.

В данном случае, размер кэша L1 составляет 512 килобайт, что равно 524288 байтам. Это ровно достаточно для хранения матриц A и B целиком. Так как блоки матриц A и B постоянно перезаписываются и повторно используются внутри внутренних циклов, мы можем считать, что данные из них кэшируются и не вызывают кэш-промахов. Однако блоки матрицы C будут вызывать кэш-промахи, так как они заполняются заново на каждой итерации.

Количество кэш-промахов будет равно количеству блоков матрицы C, которые не помещаются в кэш полностью и требуют обращения к памяти. Для каждой итерации внешних циклов блоки матрицы C перезаписываются, поэтому количество кэш-промахов будет равно общему количеству блоков матрицы C.

Так как мы разделили матрицы на блоки размером 32x32 элемента, количество блоков матрицы C будет равно количеству блоков внутри каждого измерения, то есть 8×8=64. Таким образом, количество кэш-промахов в данном коде будет равно 64.

Теория

```
print(func.get_source())
```

```
for_begin_n.inner.preheader.6:                    ; preds = %for_begin_n.inner.preheader.6, %for_end_k.5
  %lsr.iv271 = phi float* [ %scevgep272, %for_begin_n.inner.preheader.6 ], [ %scevgep270, %for_end_k.5 ]
  %wide.load34.3207 = phi <8 x float> [ %.promoted206, %for_end_k.5 ], [ %175, %for_begin_n.inner.preheader.6 ]
  %wide.load34.2204 = phi <8 x float> [ %.promoted203, %for_end_k.5 ], [ %174, %for_begin_n.inner.preheader.6 ]
  %wide.load34.1201 = phi <8 x float> [ %.promoted200, %for_end_k.5 ], [ %173, %for_begin_n.inner.preheader.6 ]
  %wide.load34198 = phi <8 x float> [ %.promoted197, %for_end_k.5 ], [ %172, %for_begin_n.inner.preheader.6 ]
  %indvars.iv10.6 = phi i64 [ 0, %for_end_k.5 ], [ %indvars.iv.next11.6, %for_begin_n.inner.preheader.6 ]
  %lsr.iv271273 = bitcast float* %lsr.iv271 to <8 x float>*
  %scevgep277 = getelementptr float, float* %lsr.iv227, i64 %indvars.iv10.6
  %171 = load float, float* %scevgep277, align 4, !tbaa !114
  %broadcast.splatinsert35 = insertelement <8 x float> undef, float %171, i32 0
  %broadcast.splat36 = shufflevector <8 x float> %broadcast.splatinsert35, <8 x float> undef, <8 x i32> zeroinitializer
  %scevgep276 = getelementptr <8 x float>, <8 x float>* %lsr.iv271273, i64 -3
  %wide.load33 = load <8 x float>, <8 x float>* %scevgep276, align 64, !tbaa !116
  %172 = call <8 x float> @llvm.fmuladd.v8f32(<8 x float> %broadcast.splat36, <8 x float> %wide.load33, <8 x float> %wide.lo
  %scevgep275 = getelementptr <8 x float>, <8 x float>* %lsr.iv271273, i64 -2
  %wide.load33.1 = load <8 x float>, <8 x float>* %scevgep275, align 32, !tbaa !116
  %173 = call <8 x float> @llvm.fmuladd.v8f32(<8 x float> %broadcast.splat36, <8 x float> %wide.load33.1, <8 x float> %wide.
  %scevgep274 = getelementptr <8 x float>, <8 x float>* %lsr.iv271273, i64 -1
  %wide.load33.2 = load <8 x float>, <8 x float>* %scevgep274, align 64, !tbaa !116
  %174 = call <8 x float> @llvm.fmuladd.v8f32(<8 x float> %broadcast.splat36, <8 x float> %wide.load33.2, <8 x float> %wide.
  %wide.load33.3 = load <8 x float>, <8 x float>* %lsr.iv271273, align 32, !tbaa !116
  %175 = call <8 x float> @llvm.fmuladd.v8f32(<8 x float> %broadcast.splat36, <8 x float> %wide.load33.3, <8 x float> %wide.
  %indvars.iv.next11.6 = add nuw nsw i64 %indvars.iv10.6, 1
  %scevgep272 = getelementptr float, float* %lsr.iv271, i64 256
  %exitcond12.6 = icmp eq i64 %indvars.iv.next11.6, 256
  br i1 %exitcond12.6, label %for_end_k.6, label %for_begin_n.inner.preheader.6, !prof !51
```

ДЗ 1

```python
@autotvm.template("matmul_brute_force_reduced_")
def matmul_brute_force(N, L, M, candidates=None):
    A = te.placeholder((N, L), name="A", dtype="float32")
    B = te.placeholder((L, M), name="B", dtype="float32")

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)

    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    ##### define space begin #####
    cfg = autotvm.get_config()
    filter = lambda v: v.size[0] != 1 and v.size[1] != 1
    multi_filter = lambda e: 16 <= (e["tile_x"].size[1] + e["tile_y"].size[1]) < 128
    cfg.multi_filter(multi_filter)

    cfg.define_split("tile_y", y, num_outputs=2, filter=filter)
    cfg.define_split("tile_x", x, num_outputs=2, filter=filter)
    ##### define space end #####

    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

ДЗ 1

```
import logging
import sys
# logging config (for printing tuning log to the screen)
logging.getLogger("autotvm").setLevel(logging.DEBUG)
logging.getLogger("autotvm").addHandler(logging.StreamHandler(sys.stdout))
```

```
No: 12  GFLOPS: 22.01/22.01     result: MeasureResult(costs=(0.011660069, 0.011842209999999999, 0.011887309, 0.01214783899999
9999, 0.01220982, 0.012457519, 0.012564019999999999, 0.012779699), error_no=MeasureErrorNo.NO_ERROR, all_cost=0.5493397712707
52, timestamp=1713769588.1319046)       [('tile_y', [-1, 32]), ('tile_x', [-1, 64])],None,49
```

```
!cat /proc/cpuinfo | grep flags
```

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscal
l nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid tsc_known_freq pni pclmulqdq
ssse3 fma cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a
misalignsse 3dnowprefetch osvw topoext ssbd ibrs ibpb stibp vmmcall fsgsbase tsc_adjust bmi1 avx2 smep bmi2 rdseed adx smap c
lflushopt clwb sha_ni xsaveopt xsavec xgetbv1 clzero xsaveerptr arat npt nrip_save umip rdpid
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscal
l nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid tsc_known_freq pni pclmulqdq
ssse3 fma cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a
misalignsse 3dnowprefetch osvw topoext ssbd ibrs ibpb stibp vmmcall fsgsbase tsc_adjust bmi1 avx2 smep bmi2 rdseed adx smap c
lflushopt clwb sha_ni xsaveopt xsavec xgetbv1 clzero xsaveerptr arat npt nrip_save umip rdpid
```

ДЗ 2

```python
@autotvm.template("filtered_c_template")
def filtered_c_template(M, N, K, dtype):
    A = te.placeholder((M, K), name='A', dtype=dtype)
    B = te.placeholder((K, N), name='B', dtype=dtype)

    k = te.reduce_axis((0, K), name='k')
    C = te.compute((M, N),
                   lambda i, j: te.sum(
                       A[i, k] * B[k, j], axis=k
                       ), name='C')

    s = te.create_schedule(C.op)

    i, j = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    candidates = [[1, 1024], [2, 512], [4, 256], [8, 128], [16, 64], [32, 32], [64, 16], [128, 8],

    cfg = autotvm.get_config()
    cfg.multi_filter(filter=filter)
    cfg.define_split("tile_y", i, num_outputs=2, policy="candidate", candidate=candidates)
    cfg.define_split("tile_x", j, num_outputs=2, policy="candidate", candidate=candidates)

    yo, yi = cfg["tile_y"].apply(s, C, i)
    xo, xi = cfg["tile_x"].apply(s, C, j)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

ДЗ 2

```
!lscpu | grep L
```

```
Byte Order:                      Little Endian
L1d cache:                       32 KiB (1 instance)
L1i cache:                       32 KiB (1 instance)
L2 cache:                        256 KiB (1 instance)
L3 cache:                        55 MiB (1 instance)
Vulnerability L1tf:              Mitigation; PTE Inversion
```

```
def filter(e):
    max_bx = e["tile_x"].size[0] <= e["tile_x"].size[1]
    inline_cache = e["tile_y"].size[1] <= e["tile_x"].size[1]
    cache_kbytes = 32
    constrains = 4 * (e["tile_x"].size[1] + e["tile_y"].size[1] + e["tile_x"].
    size[1] * e["tile_y"].size[1]) <= cache_kbytes * 1024
    return max_bx and constrains
```

ДЗ 2

```
@main = primfn(A_1: handle, B_1: handle, C_1: handle) -> ()
  attr = {"from_legacy_te_schedule": True, "global_symbol": "main", "tir.noalias": True}
  buffers = {A: Buffer(A_2: Pointer(float32), float32, [1024, 1024], []),
             B: Buffer(B_2: Pointer(float32), float32, [1024, 1024], []),
             C: Buffer(C_2: Pointer(float32), float32, [1024, 1024], [])}
  buffer_map = {A_1: A, B_1: B, C_1: C} {
  for (i.outer: int32, 0, 1024) {
    for (j.inner.init: int32, 0, 1024) {
      C_3: Buffer(C_2, float32, [1048576], [])[((i.outer*1024) + j.inner.init)] = 0f32
    }
    for (k: int32, 0, 1024) {
      for (j.inner: int32, 0, 1024) {
        let cse_var_2: int32 = (i.outer*1024)
        let cse_var_1: int32 = (cse_var_2 + j.inner)
        C_3[cse_var_1] = (C_3[cse_var_1] + (A_3: Buffer(A_2, float32, [1048576], [])[(cse_var_2 + k)]*B_3: Buffer(B_2, float3
2, [1048576], [])[((k*1024) + j.inner)]))
      }
    }
  }
}
```

```
func 245.2255676
```

Io, Ii - e["tile_y"].size[0], e["tile_y"].size[1]

Jo, Ji - e["tile_x"].size[0], e["tile_x"].size[1]

- Базовый оптимизация - увеличение размеров блоков, т.е Ji и Ii должны быть больше, чем обратные им Jo и Io соответственно. (Io <= Ii, Jo <= Ji)
- Причем желательно, чтобы Ji >= Ii, т.к это длина последовательных элементов в памяти и к ним обращение будет оптимальнее.
- Для ограничения размеров блоков сверху выберем одну итерацию k, мы хотим чтобы блоки матриц A, B, C, с которыми мы работаем на этой итерации содержались в кеше. Их размеры соответственно: Ii, Ji, Ii*Ji. Т.к кеш у этой машины 32 килобайта, и мы работаем с float32 (4 байта), то получается соотношение 4 * (Ii + Ji + IiJi) <= 32 * 1024*
- Чтобы сильно не сужать пространство поиска уберем не самое важное условие Io <= Ii

При проведении исследований в таргет не включались векторные расширения AVX для чистоты экспериментов