Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 14

Triton: програмування GPU з відкритим вихідним кодом для нейронних мереж

Ця лекція буде присвячена мові програмування Triton 1.0, мові програмування з відкритим вихідним кодом, схожій на Python, яка дає змогу дослідникам без досвіду роботи з CUDA писати високоефективний код GPU — у більшості випадків на рівні з тим, що може створити експерт. Triton дозволяє досягти максимальної продуктивності апаратного забезпечення з відносно невеликими зусиллями; наприклад, його можна використовувати для написання ядер множення матриць FP16, які відповідають продуктивності cuBLAS — чого не можуть зробити багато програмістів GPU — менш ніж за 25 рядків коду.

Багато дослідників останні пару років вже використовували її для створення ядер, які в 2 рази ефективніші, ніж еквівалентні реалізації Torch, і використання Triton зробить програмування GPU доступнішим для всіх.

Triton — це мова та компілятор для паралельного програмування. Він спрямований на забезпечення середовища програмування на основі Python для продуктивного написання спеціальних обчислювальних ядер DNN, здатних працювати з максимальною пропускною здатністю на сучасному обладнанні GPU.

Інсталяція

Ви можете встановити останню стабільну версію Triton з pip:

```
pip install triton
```

Бінарні “колеса” доступні для CPython 3.6-3.9 і PyPy 3.6-3.7.

І останній вечірній випуск “новин”:

```
pip install -U --pre triton
```

Інсталяція з ісходників

Ви можете інсталювати пакет Python з джерела, виконавши такі команди:

```
git clone
https://github.com/openai/triton.git;
cd triton/python;
pip install cmake; # build time dependency
pip install -e .
```

Зауважте, що якщо **`llvm-11`** відсутній у вашій системі, сценарій **`setup.py`** завантажить офіційне посилання на статичні бібліотеки **`LLVM11`**.

Потім ви можете перевірити свою інсталяцію, запустивши модульні тести:

```
pip install -r requirements-test.txt
pytest -vs test/unit/
```

і контрольні показники

```
cd bench/
python -m run --with-plots --result-dir
/tmp/triton-bench
```

Нижче наведена галерея навчальних посібників для написання різних основних операцій з Triton. Рекомендується ознайомитися з туторіалами по порядку, починаючи з найпростішого.

Векторне додавання

Зараз ми напишемо просте векторне додавання за допомогою Triton і дізнаєтеся про:

- Основну модель програмування Triton
- Декоратор `triton.jit`, який використовується для визначення ядер Triton.
- Найкращі методи перевірки та порівняння ваших користувальницьких операцій із власними довідковими реалізаціями

Обчислювальне ядро

```
import torch
import triton
import triton.language as tl


@triton.jit
def add_kernel(
    x_ptr,  # *Pointer* to first input vector
    y_ptr,  # *Pointer* to second input vector
    output_ptr,  # *Pointer* to output vector
    n_elements,  # Size of the vector
    **meta,  # Optional meta-parameters for the kernel
):
    BLOCK_SIZE = meta['BLOCK_SIZE']  # How many inputs
each program should process
    # There are multiple 'program's processing
different data. We identify which program
    # we are here
```

```
    pid = tl.program_id(axis=0)  # We use a 1D launch
grid so axis is 0
    # This program will process inputs that are offset
from the initial data.
    # for instance, if you had a vector of length 256
and block_size of 64, the programs
    # would each access the elements [0:64, 64:128,
128:192, 192:256].
    # Note that offsets is a list of pointers
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against
out-of-bounds accesses
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extar
elements in case the input is not a
    # multiple of the block size
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
```

```
output = x + y
# Write x + y back to DRAM
tl.store(output_ptr + offsets, output, mask=mask)
```

Давайте також оголосимо допоміжну функцію, щоб (1) виділити тензор z і (2) поставити в чергу вищезгадане ядро з відповідними розмірами сітки/блоку.

```
def add(x: torch.Tensor, y: torch.Tensor):
    # We need to preallocate the output
    output = torch.empty_like(x)
    assert x.is_cuda and y.is_cuda and output.is_cuda
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel
instances that run in parallel.
    # It is analogous to CUDA launch grids. It can be
either Tuple[int], or Callable(metaparameters) ->
Tuple[int]
    # In this case, we use a 1D grid where the size is
the number of blocks
    grid = lambda meta: (triton.cdiv(n_elements,
meta['BLOCK_SIZE']),)
    # NOTE:
    #  - each torch.tensor object is implicitly
```

```
converted into a pointer to its first element.
    #  - `triton.jit`'ed functions can be index with a
launch grid to obtain a callable GPU kernel
    #  - don't forget to pass meta-parameters as
keywords arguments
    pgm = add_kernel[grid](x, y, output, n_elements,
BLOCK_SIZE=1024)
    # We return a handle to z but, since
`torch.cuda.synchronize()` hasn't been called, the
kernel is still
    # running asynchronously at this point.
    return output
```

Тепер ми можемо використовувати наведену вище функцію, щоб обчислити поелементну суму двох об’єктів **`torch.tensor`** і перевірити її правильність:

```
torch.manual_seed(0)
size = 98432
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
output_torch = x + y
output_triton = add(x, y)
print(output_torch)
print(output_triton)
print(
    f'The maximum difference between torch and triton is '
    f'{torch.max(torch.abs(output_torch - output_triton))}'
)
```

```
torch.manual_seed(0)
size = 98432
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
output_torch = x + y
output_triton = add(x, y)
print(output_torch)
print(output_triton)
print(
    f'The maximum difference between torch and triton
is '
    f'{torch.max(torch.abs(output_torch -
output_triton))}'
)
```

На виході будем мати:

```
tensor([1.3713, 1.3076, 0.4940,   ..., 0.6724, 1.2141,
0.9733], device='cuda:0')
tensor([1.3713, 1.3076, 0.4940,  ..., 0.6724, 1.2141,
0.9733], device='cuda:0')
The maximum difference between torch and triton is 0.0
```

Порівняння

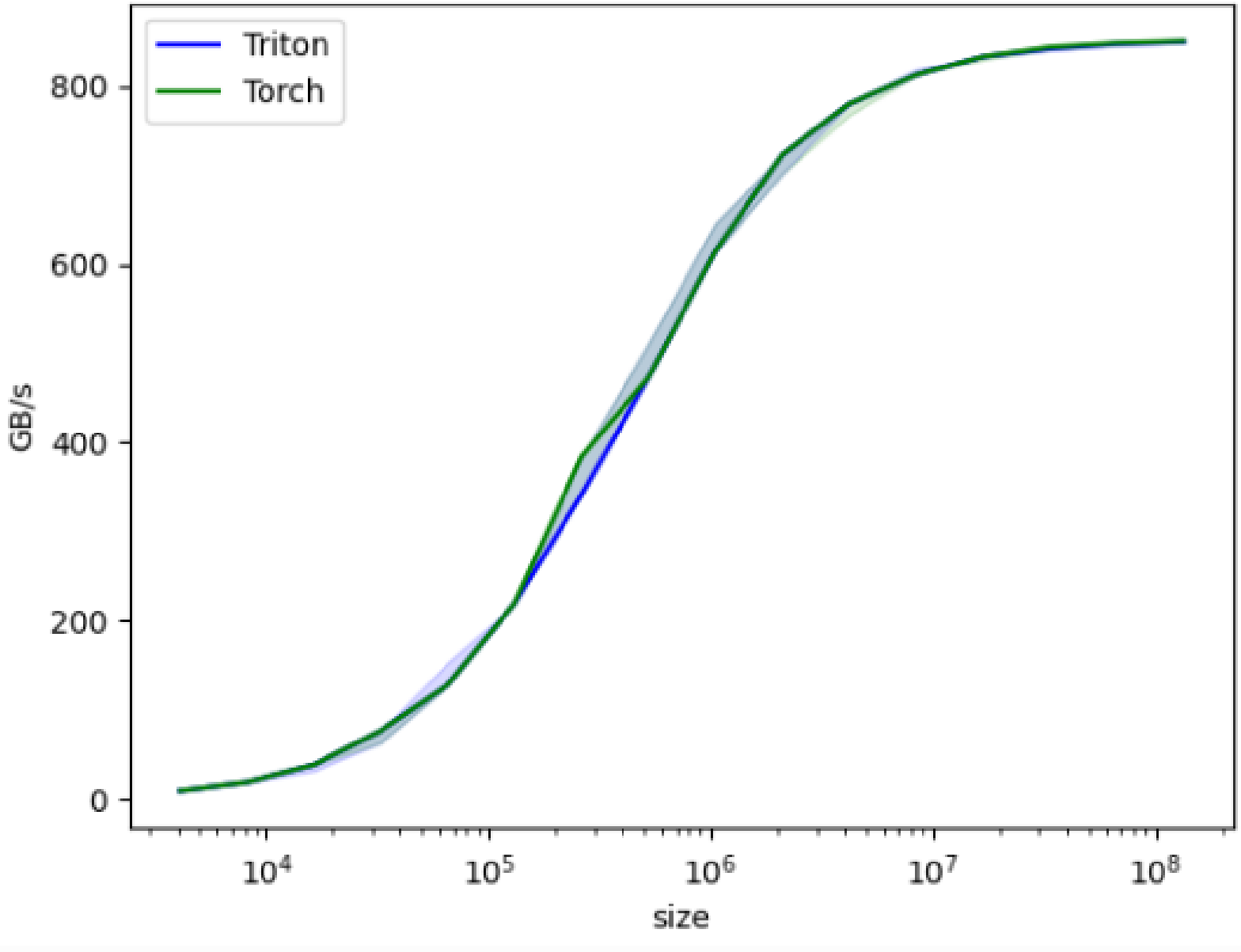
Тепер ми можемо порівняти нашу спеціальну операцію з векторами зростаючих розмірів, щоб зрозуміти, як вона працює відносно PyTorch. Щоб спростити задачу, у Triton є набір вбудованих утиліт, які дозволяють нам стислий графік роботи наших користувацьких операцій для різних розмірів проблем.

```
@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['size'],  # argument names to use as
an x-axis for the plot
        x_vals=[
            2 ** i for i in range(12, 28, 1)
        ],  # different possible values for `x_name`
        x_log=True,  # x axis is logarithmic
        line_arg='provider',  # argument name whose
value corresponds to a different line in the plot
        line_vals=['triton', 'torch'],  # possible
values for `line_arg`
        line_names=['Triton', 'Torch'],  # label name
for the lines
        styles=[('blue', '-'), ('green', '-')],  # line
styles
        ylabel='GB/s',  # label name for the y-axis
        plot_name='vector-add-performance',  # name for
the plot. Used also as a file name for saving the plot.
```

```
        args={},  # values for function arguments not
in `x_names` and `y_name`
    )
)
def benchmark(size, provider):
    x = torch.rand(size, device='cuda',
dtype=torch.float32)
    y = torch.rand(size, device='cuda',
dtype=torch.float32)
    if provider == 'torch':
        ms, min_ms, max_ms =
triton.testing.do_bench(lambda: x + y)
    if provider == 'triton':
        ms, min_ms, max_ms =
triton.testing.do_bench(lambda: add(x, y))
    gbps = lambda ms: 12 * size / ms * 1e-6
    return gbps(ms), gbps(max_ms), gbps(min_ms)
```

Тепер ми можемо запустити описану вище функцію. Передайте **`print_data=True`**, щоб побачити номер продуктивності, **`show_plots=True`**, щоб побудувати їх, та/або **`save_path='/path/to/results/'`**, щоб зберегти їх на диск разом із необробленими даними в форматі CSV

```
benchmark.run(print_data=True, show_plots=True)
```


Triton
Torch
800
600
400
200
0
GB/s
10^4
10^5
10^6
10^7
10^8
size

На виході маємо:

```
vector-add-performance:
            size      Triton       Torch
0         4096.0    9.600000    9.600000
1         8192.0   19.200000   19.200000
2        16384.0   38.400001   38.400001
3        32768.0   76.800002   76.800002
4        65536.0  127.999995  127.999995
5       131072.0  219.428568  219.428568
6       262144.0  341.333321  384.000001
7       524288.0  472.615390  472.615390
8      1048576.0  614.400016  614.400016
9      2097152.0  722.823517  722.823517
10     4194304.0  780.190482  780.190482
11     8388608.0  812.429770  812.429770
12    16777216.0  833.084721  833.084721
13    33554432.0  842.004273  843.811163
14    67108864.0  847.448255  848.362445
15   134217728.0  849.737435  850.656574
```

Total running time of the script: (1 minutes 52.411 seconds)

Вхідний Pyton-код додавання вектора:
https://triton-lang.org/_downloads/62d97d49a32414049819dd8bb8378080/01-vector-add.py

Посилання на Jupyter-ноутбук
https://triton-lang.org/_downloads/f191ee1e78dc52eb5f7cba88f71cef2f/01-vector-add.ipynb
Jupyter-ноутбук - це середовище розробки, де одразу можна бачити результат виконання коду та його окремих фрагментів.

Fused Softmax

У цій лекції ви напишете об’єднану операцію **`softmax`**, яка значно швидше, ніж рідна операція **`PyTorch`** для певного класу матриць: тих, чиї рядки можуть поміститися в SRAM графічного процесора. Ви дізнаєтесь про:

- Переваги злиття ядра для операцій, пов’язаних із пропускною здатністю.
- Оператори скорочення в Triton.

Спеціальні ядра графічного процесора для поелементних доповнень є цінними в освіті, але на практиці вони не заведуть далеко. Натомість розглянемо випадок простої (числово стабілізованої) операції **`softmax`**:

```python
import torch


@torch.jit.script
def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch

    We subtract the maximum element in order to avoid overflows. Softmax is invariant to
    this shift.
    """
    # read  MN elements ; write M  elements
    x_max = x.max(dim=1)[0]
    # read MN + M elements ; write MN elements
    z = x - x_max[:, None]
    # read  MN elements ; write MN elements
    numerator = torch.exp(z)
    # read  MN elements ; write M  elements
    denominator = numerator.sum(dim=1)
    # read MN + M elements ; write MN elements
    ret = numerator / denominator[:, None]
    # in total: read 5MN + 2M elements ; wrote 3MN + 2M elements
    return ret
```

При наївній реалізації в PyTorch обчислення `y = naive_softmax(x)` для

$$x \in R^{M \times N}$$

вимагає читання

$$5MN + 2M$$

елементів із DRAM і зворотного запису

$$3MN + 2M$$

елементів.

Це, очевидно, марнотратство; ми б воліли мати користувацьке «з'єднане» ядро, яке читає `X` лише один раз і виконує всі необхідні обчислення на чіпі.

Для цього знадобиться читання та записування лише **MN** байтів, тому ми могли б очікувати теоретичного прискорення в **~4x** (тобто

$$(8MN + 4M)/2MN$$

).
Прапорці **torch.jit.script** мають на меті виконати цей вид «злиття ядра» автоматично, але, як ми побачимо пізніше, він ще далекий від ідеалу.

Наше ядро **`softmax`** працює наступним чином: кожна програма завантажує рядок вхідної матриці **`X`**, нормалізує її і записує результат у вихідний **`Y`**. Зауважте, що одне важливе обмеження Triton полягає в тому, що кожен блок повинен мати число в степені двох. елементів, тому нам потрібно внутрішньо «заповнювати» кожен рядок і належним чином охороняти операції з пам’яттю, якщо ми хочемо обробляти будь-які можливі форми введення:

```python
import triton
import triton.language as tl


@triton.jit
def softmax_kernel(
    output_ptr, input_ptr, input_row_stride, output_row_stride, n_cols, **meta
):
    # The rows of the softmax are independent, so we parallelize across those
    row_idx = tl.program_id(0)
    BLOCK_SIZE = meta['BLOCK_SIZE']
    # The stride represents how much we need to increase the pointer to advance 1 row
    row_start_ptr = input_ptr + row_idx * input_row_stride
    # The block size is the next power of two greater than n_cols, so we can fit each
    # row in a single block
    col_offsets = tl.arange(0, BLOCK_SIZE)
    input_ptrs = row_start_ptr + col_offsets
    # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-float('inf'))
    # Substract maximum for numerical stability
    row_minus_max = row - tl.max(row, axis=0)
    # Note that exponentials in Triton are fast but approximate (i.e., think __expf in CUDA)
    numerator = tl.exp(row_minus_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator
    # Write back output to DRAM
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets
    tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)
```

Ми можемо створити допоміжну функцію, яка ставить в чергу ядро та його (мета-)аргументи для будь-якого заданого тензора введення.

```python
def softmax(x):
    n_rows, n_cols = x.shape
    # The block size is the smallest power of two greater than the number of columns in `x`
    BLOCK_SIZE = triton.next_power_of_2(n_cols)
    # Another trick we can use is to ask the compiler to use more threads per row by
    # increasing the number of warps (`num_warps`) over which each row is distributed.
    # You will see in the next tutorial how to auto-tune this value in a more natural
    # way so you don't have to come up with manual heuristics yourself.
    num_warps = 4
    if BLOCK_SIZE >= 2048:
        num_warps = 8
    if BLOCK_SIZE >= 4096:
        num_warps = 16
    # Allocate output
    y = torch.empty_like(x)
    # Enqueue kernel. The 1D launch grid is simple: we have one kernel instance per row o
    # f the input matrix
    softmax_kernel[(n_rows,)](
        y,
        x,
        x.stride(0),
        y.stride(0),
        n_cols,
        num_warps=num_warps,
        BLOCK_SIZE=BLOCK_SIZE,
    )
    return y
```

Ми переконаємося, що ми тестуємо наше ядро на матриці з неправильною кількістю рядків і стовпців. Це дозволить нам перевірити, чи працює наш механізм заповнення.

```
torch.manual_seed(0)
x = torch.randn(1823, 781, device='cuda')
y_triton = softmax(x)
y_torch = torch.softmax(x, axis=1)
print(torch.allclose(y_triton, y_torch))
```

Out:

```
True
```

Як і очікувалося, результати ідентичні.

Тут ми порівняємо нашу операцію як функцію кількості стовпців у вхідній матриці – припускаючи, що 4096 рядків. Потім ми порівняємо його продуктивність з (1) torch.softmax і (2) naive_softmax, визначеними вище.

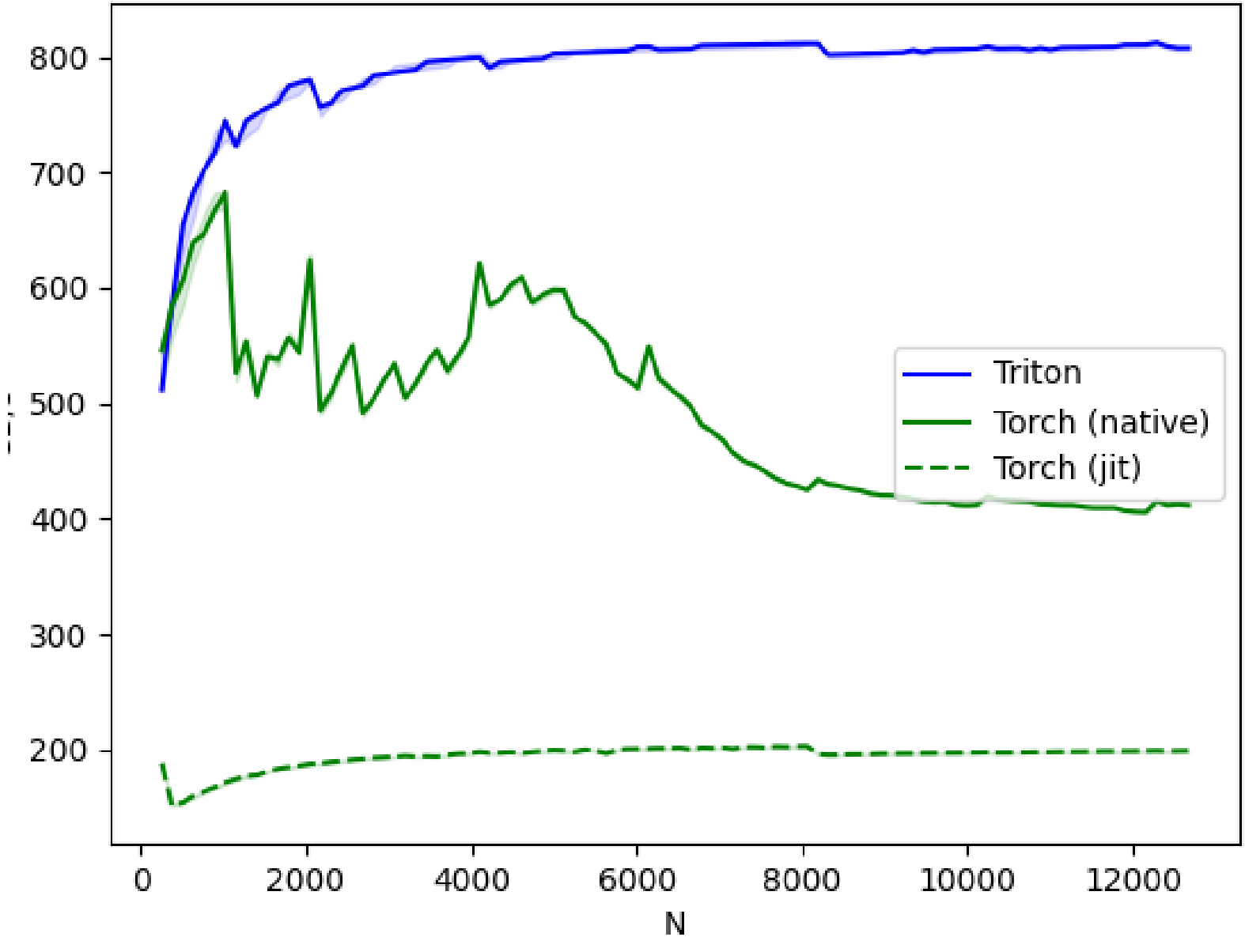
```
@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['N'],  # argument names to
use as an x-axis for the plot
        x_vals=[
            128 * i for i in range(2, 100)
        ],  # different possible values for
`x_name`
        line_arg='provider',  # argument name
whose value corresponds to a different line in
the plot
        line_vals=[
```

```
            'triton',
            'torch-native',
            'torch-jit',
        ],  # possible values for `line_arg``
        line_names=[
            "Triton",
            "Torch (native)",
            "Torch (jit)",
        ],  # label name for the lines
        styles=[('blue', '-'), ('green', '-'),
('green', '--')],  # line styles
        ylabel="GB/s",  # label name for the
y-axis
        plot_name="softmax-performance",  #
name for the plot. Used also as a file name
for saving the plot.
```

```
        args={'M': 4096},  # values for
function arguments not in `x_names` and
`y_name`
    )
)
def benchmark(M, N, provider):
    x = torch.randn(M, N, device='cuda',
dtype=torch.float32)
    if provider == 'torch-native':
        ms, min_ms, max_ms =
triton.testing.do_bench(lambda:
torch.softmax(x, axis=-1))
    if provider == 'triton':
        ms, min_ms, max_ms =
triton.testing.do_bench(lambda: softmax(x))
    if provider == 'torch-jit':
```

```
        ms, min_ms, max_ms =
triton.testing.do_bench(lambda:
naive_softmax(x))
    gbps = lambda ms: 2 * x.nelement() *
x.element_size() * 1e-9 / (ms * 1e-3)
    return gbps(ms), gbps(max_ms),
gbps(min_ms)


benchmark.run(show_plots=True,
print_data=True)
```


800
700
600
500
400
300
200
0
2000
4000
6000
8000
10000
12000
N
Triton
Torch (native)
Torch (jit)

Out:

```
softmax-performance:
          N      Triton  Torch (native)  Torch (jit)
0     256.0  512.000001      546.133347   188.321838
1     384.0  585.142862      585.142862   151.703707
2     512.0  655.360017      606.814814   154.566038
3     640.0  682.666684      640.000002   160.000000
4     768.0  702.171410      646.736871   163.839992
..      ...         ...             ...          ...
93  12160.0  810.666687      405.755985   199.038365
94  12288.0  812.429770      415.661740   199.197579
95  12416.0  809.189387      412.149375   198.854847
96  12544.0  807.661970      412.971190   199.012395
97  12672.0  807.776923      412.097543   199.167004

[98 rows x 4 columns]
```

На наведеному вище сюжеті ми бачимо, що:
Triton в 4 рази швидше, ніж **`Torch JIT`**. Це підтверджує наші підозри, що **`Torch JIT`** не виконує тут жодного синтезу.
Triton помітно швидше, ніж **`torch.softmax`** – крім того, що його легше читати, розуміти та підтримувати. Однак зауважте, що операція **`softmax PyTorch`** є більш загальною і працюватиме з тензорами будь-якої форми.
Загальний час виконання сценарію: (3 хвилини 26,243 секунди).

https://triton-lang.org/_downloads/d91442ac2982c4e0cc3ab0f43534afbc/02-fused-softmax.py

https://triton-lang.org/_downloads/034d953b6214fedce6ea03803c712b89/02-fused-softmax.ipynb

Множення матриць

У цій частині лекції ми напишемо 25-рядкове високопродуктивне ядро множення матриці FP16, яке досягає продуктивності на рівні з cuBLAS. Ви дізнаєтеся конкретно про:

- Матричні множення на рівні блоків
- Багатовимірну арифметика вказівника
- Переупорядкування програми для покращення частоти звернення до кешу другого рівня
- Автоматична настройка продуктивності

Матричні множення є ключовим будівельним блоком більшості сучасних високопродуктивних обчислювальних систем. Їх, як відомо, важко оптимізувати, тому їх реалізацію зазвичай виконують самі виробники обладнання як частину так званих «бібліотек ядра» (наприклад, cuBLAS). На жаль, ці бібліотеки часто є власністю, і їх неможливо легко налаштувати, щоб задовольнити потреби сучасних робочих навантажень глибокого навчання (наприклад, об'єднані функції активації). У цьому підручнику ви дізнаєтеся, як самостійно реалізувати ефективне множення матриці за допомогою Triton, легко налаштувати та розширити.

Грубо кажучи, ядро, яке ми напишемо, реалізує наступний заблокований алгоритм множення `(M, K)` на `(K, N)` матрицю:

```
# do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N),
dtype=float32)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc;
```

де кожна ітерація подвійно вкладеного циклу **`for`** виконується спеціальним екземпляром програми Triton.

Насправді, вищезазначений алгоритм досить простий для реалізації в Triton. Основна складність пов'язана з обчисленням місць пам'яті, в яких блоки **A** і **B** повинні бути прочитані у внутрішньому циклі. Для цього нам потрібна багатовимірна арифметика вказівників.

Для великого **2D** тензора **X** розташування пам’яті **X[i, j]** визначається як **&X[i, j] = X + i*stride_xi + j*stride_xj**. Отже, блоки покажчиків для **A[m : m+BLOCK_SIZE_M, k:k+BLOCK_SIZE_K]** і **B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]** можуть бути визначені у псевдокоді як:

```
&A[m : m+BLOCK_SIZE_M, k:k+BLOCK_SIZE_K] =
a_ptr    +    (m    :    m+BLOCK_SIZE_M)[:,
None]*A.stride(0)  +  (k  :  k+BLOCK_SIZE_K)
[None, :]*A.stride(1);
&B[k : k+BLOCK_SIZE_K, n:n+BLOCK_SIZE_N] =
b_ptr + (k : k+BLOCK_SIZE_K)[:,
None]*B.stride(0) + (n : n+BLOCK_SIZE_N)
[None, :]*B.stride(1);
```

Це означає, що покажчики на блоки **A** і **B** можна ініціалізувати (тобто **k=0**) у Triton як:

```
offs_am   =   pid_m   *   BLOCK_SIZE_M   +
tl.arange(0, BLOCK_SIZE_M)
offs_bn = pid_n * BLOCK_SIZE_N +
tl.arange(0, BLOCK_SIZE_N)
offs_k = tl.arange(0, BLOCK_SIZE_K)
a_ptrs = a_ptr + (offs_am[:,
None]*stride_am + offs_k
[None, :]*stride_ak)
b_ptrs = b_ptr + (offs_k [:,
None]*stride_bk +
offs_bn[None, :]*stride_bn)
```

Як згадувалося вище, кожен екземпляр програми обчислює **[BLOCK_SIZE_M, BLOCK_SIZE_N]** блок **C**. Важливо пам’ятати, що порядок обчислення цих блоків має значення, оскільки він впливає на швидкість звернення до кешу другого рівня нашої програми. і, на жаль, простий рядковий порядок

```
pid = triton.program_id(0);
grid_m = (M + BLOCK_SIZE_M - 1) //
BLOCK_SIZE_M;
grid_n = (N + BLOCK_SIZE_N - 1) //
BLOCK_SIZE_N;
pid_m = pid / grid_n;
pid_n = pid % grid_n;
```

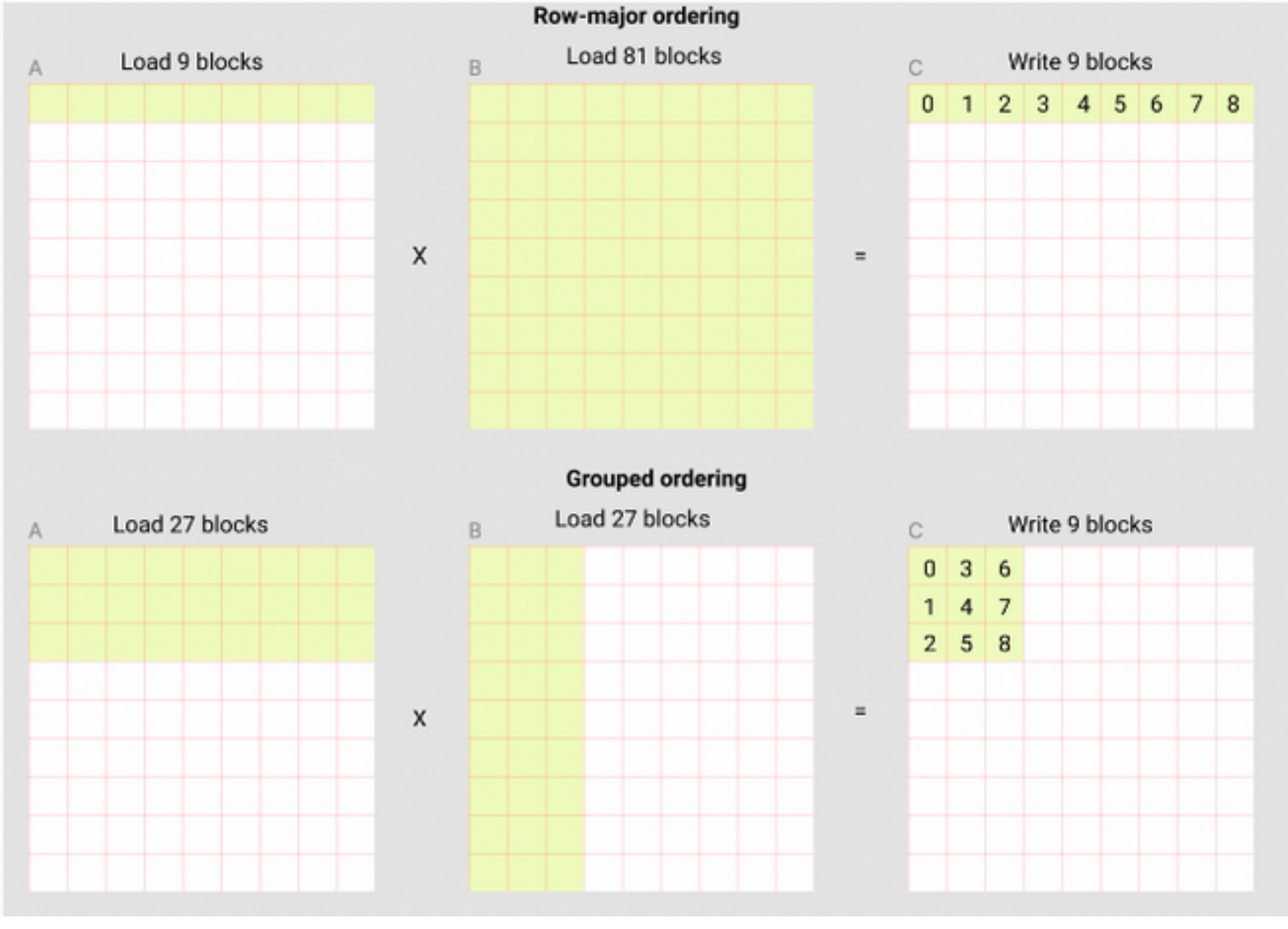
просто не збирається його різати.

Одним з можливих рішень є запуск блоків у порядку, який сприяє повторному використанню даних. Це можна зробити шляхом «супергрупування» блоків у групах рядків **GROUP_M** перед переходом до наступного стовпця:

```
# program ID
pid = tl.program_id(axis=0)
# number of program ids along the M axis
num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
# number of programs ids along the N axis
num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
# number of programs in group
num_pid_in_group = GROUP_SIZE_M *
num_pid_n
# id of the group this program is in
group_id = pid // num_pid_in_group
```

```
# row-id of the first program in the group
first_pid_m = group_id * GROUP_SIZE_M
# if `num_pid_m` isn't divisible by
`GROUP_SIZE_M`, the last group is smaller
group_size_m = min(num_pid_m -
first_pid_m, GROUP_SIZE_M)
# *within groups*, programs are ordered in
a column-major order
# row-id of the program in the *launch
grid*
pid_m = first_pid_m + (pid % group_size_m)
# col-id of the program in the *launch
grid*
pid_n = (pid % num_pid_in_group) //
group_size_m
```

Наприклад, у наступному matmul, де кожна матриця складається з 9 блоків на 9 блоків, ми бачимо, що якщо ми обчислюємо вихід у порядку основних рядків, нам потрібно завантажити 90 блоків у SRAM, щоб обчислити перші 9 вихідних блоків, але якщо ми робимо це в згрупованому порядку, нам потрібно лише завантажити 54 блоки.


Row-major ordering
A
Load 9 blocks
B
Load 81 blocks
C
Write 9 blocks
0 1 2 3 4 5 6 7 8
X
=
Grouped ordering
A
Load 27 blocks
B
Load 27 blocks
C
Write 9 blocks
0 3 6
1 4 7
2 5 8
X
=

На практиці це може підвищити продуктивність нашого ядра множення матриці більш ніж на 10% на деякій апаратній архітектурі (наприклад, від 220 до 245 TFLOPS на A100).

Кінцевий результат

```
import torch
import triton
import triton.language as tl

# %
# :code:`triton.jit`'ed functions can be auto-tuned by
using the `triton.autotune`
# decorator, which consumes:
#     - A list of :code:`triton.Config` objects that
define different configurations of
#           meta-parameters (e.g., BLOCK_SIZE_M) and
compilation options (e.g., num_warps) to try
#     - An autotuning *key* whose change in values will
trigger evaluation of all the
#       provided configs

@triton.autotune(
    configs=[
```

```
                    triton.Config({'BLOCK_SIZE_M':   128,
'BLOCK_SIZE_N':      256,      'BLOCK_SIZE_K':      32,
'GROUP_SIZE_M': 8}, num_stages=3, num_warps=8),
                    triton.Config({'BLOCK_SIZE_M':   256,
'BLOCK_SIZE_N':      128,      'BLOCK_SIZE_K':      32,
'GROUP_SIZE_M': 8}, num_stages=3, num_warps=8),
                    triton.Config({'BLOCK_SIZE_M':   256,
'BLOCK_SIZE_N':       64,          'BLOCK_SIZE_K':     32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                   triton.Config({'BLOCK_SIZE_M':   64   ,
'BLOCK_SIZE_N':      256,      'BLOCK_SIZE_K':      32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                    triton.Config({'BLOCK_SIZE_M':   128,
'BLOCK_SIZE_N':      128,      'BLOCK_SIZE_K':      32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                    triton.Config({'BLOCK_SIZE_M':   128,
'BLOCK_SIZE_N':       64     ,     'BLOCK_SIZE_K':     32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                   triton.Config({'BLOCK_SIZE_M':   64   ,
```

```
'BLOCK_SIZE_N':      128,      'BLOCK_SIZE_K':      32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                    triton.Config({'BLOCK_SIZE_M':   128,
'BLOCK_SIZE_N':     32     ,     'BLOCK_SIZE_K':     32,
'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
                  triton.Config({'BLOCK_SIZE_M':   64   ,
'BLOCK_SIZE_N':     32     ,     'BLOCK_SIZE_K':     32,
'GROUP_SIZE_M': 8}, num_stages=5, num_warps=2),
                  triton.Config({'BLOCK_SIZE_M':   32   ,
'BLOCK_SIZE_N':     64     ,     'BLOCK_SIZE_K':     32,
'GROUP_SIZE_M': 8}, num_stages=5, num_warps=2),
    ],
    key=['M', 'N', 'K'],
)
# %
# We can now define our kernel as normal, using all the
techniques presented above
@triton.jit
def matmul_kernel(
```

```
    # Pointers to matrices
    a_ptr, b_ptr, c_ptr,
    # Matrix dimensions
    M, N, K,
       # The stride variables represent how much to
increase the ptr by when moving by 1
    # element in a particular dimension. E.g. stride_am
is how much to increase a_ptr
    # by to get the element one row down (A has M rows)
    stride_am, stride_ak,
    stride_bk, stride_bn,
    stride_cm, stride_cn,
    # Meta-parameters
    **meta,
):
    """Kernel for computing the matmul C = A x B.
     A has shape (M, K), B has shape (K, N) and C has
shape (M, N)
    """
```

```
    # extract meta-parameters
    BLOCK_SIZE_M = meta['BLOCK_SIZE_M']
    BLOCK_SIZE_N = meta['BLOCK_SIZE_N']
    BLOCK_SIZE_K = meta['BLOCK_SIZE_K']
    GROUP_SIZE_M = 8

                                                                #
-------------------------------------------------------------
----
    # Map program ids `pid` to the block of C it should
compute.
    # This is done in a grouped ordering to promote L2
data reuse
     # See above `L2 Cache Optimizations` section for
details
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
```

```
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
        group_size_m  =  min(num_pid_m  -  first_pid_m,
GROUP_SIZE_M)
    pid_m = first_pid_m + (pid % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

                                                        #
-------------------------------------------------------
---
    # Create pointers for the first blocks of A and B.
    # We will advance this pointer as we move in the K
direction
    # and accumulate
    # a_ptrs is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K]
pointers
    # b_ptrs is a block of [BLOCK_SIZE_K, BLOCK_SIZE_n]
pointers
      #  see  above  `Pointer  Arithmetics`  section  for
```

```
details
      offs_am = pid_m * BLOCK_SIZE_M + tl.arange(0,
BLOCK_SIZE_M)
      offs_bn = pid_n * BLOCK_SIZE_N + tl.arange(0,
BLOCK_SIZE_N)
    offs_k = tl.arange(0, BLOCK_SIZE_K)
      a_ptrs = a_ptr + (offs_am[:, None]*stride_am +
offs_k [None, :]*stride_ak)
      b_ptrs = b_ptr + (offs_k [:, None]*stride_bk +
offs_bn[None, :]*stride_bn)

                                                    #
----------------------------------------------------
----
    # Iterate to compute a block of the C matrix
          #  We  accumulate  into  a  `[BLOCK_SIZE_M,
BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
      # `accumulator` will be converted back to fp16
```

```
after the loop
                accumulator    =    tl.zeros((BLOCK_SIZE_M,
BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, K, BLOCK_SIZE_K):
          # Note that for simplicity, we don't apply a
mask here.
          # This means that if K is not a multiple of
BLOCK_SIZE_K,
           # this will access out-of-bounds memory and
produce an
        # error or (worse!) incorrect results.
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        # We accumulate along the K dimension
        accumulator += tl.dot(a, b)
        # Advance the ptrs to the next K block
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # you can fuse arbitrary activation functions here
```

```
    # while the accumulator is still in FP32 !
    if meta['ACTIVATION']:
        accumulator = meta['ACTIVATION'](accumulator)
    c = accumulator.to(tl.float16)

                                                                #
-------------------------------------------------------------
----
    # Write back the block of the output matrix C
      offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0,
BLOCK_SIZE_M)
      offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0,
BLOCK_SIZE_N)
     c_ptrs = c_ptr + stride_cm * offs_cm[:, None] +
stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :]
< N)
    tl.store(c_ptrs, c, mask=c_mask)
```

```
# we can fuse `leaky_relu` by providing it as an
`ACTIVATION` meta-parameter in `_matmul`
@triton.jit
def leaky_relu(x):
    return tl.where(x >= 0, x, 0.01 * x)
```

Наступна лекція буде присвячена прикладам програмування інтеллектуальних методів на базі проблемно-орієнтованої мови Triton.