lecture_06.py

```
import time
from typing import Callable
import torch
import torch.nn as nn
from torch.profiler import ProfilerActivity
from torch.utils.cpp_extension import load_inline
import triton
import triton.language as tl
from execute_util import text, link, image
from file_util import ensure_directory_exists
from lecture_util import article_link
from torch_util import get_device
from lecture_06_utils import check_equal, check_equal2, get_local_url, round1, mean
import os

def main():
    announcements()
```

Last lecture: high-level overview of GPUs and performance
This lecture: benchmarking/profiling + write kernels

```
    if not torch.cuda.is_available():
        text("You should run this lecture on a GPU to get the full experience.")

    review_of_gpus()
    benchmarking_and_profiling()  # Important for understanding!

    kernel_fusion_motivation()
    cuda_kernels()  # Write kernels in CUDA/C++
    triton_kernels()  # Write kernels in Python
    pytorch_compilation()  # Don't write kernels at all?

    # More advanced computations
    triton_softmax_main()
```

Summary

Gap between the programming model (PyTorch, Triton, PTX) and hardware => performance mysteries

Benchmarking for understanding scaling
Profiling for understanding internals of PyTorch functions (bottoms out with kernels)
Looking at PTX assembly to understand internals of CUDA kernels

5 ways to write a function: manual, PyTorch, compiled, CUDA, Triton
GeLU (element-wise), softmax (row-wise), matmul (complex aggregation)

Key principle: organize computation to minimize reads/writes
Key ideas: kernel fusion (warehouse/factory analogy), tiling (shared memory)
Automatic compilers (Triton, torch.compile) will get better over time

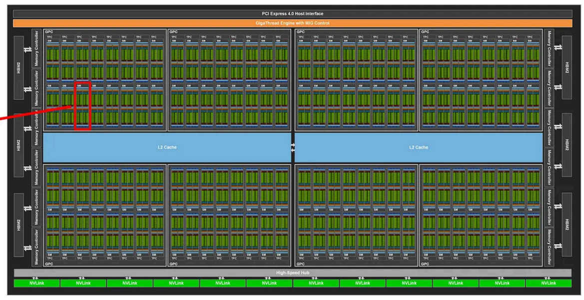
```
    further_reading()


def announcements():
```

Assignment 1 leaderboard [Leaderboard]
Assignment 2 is out [A2]

```
def review_of_gpus():
```

Hardware


SM


GA100 Full GPU with 128 SMs

Compute: streaming multiprocessors (SMs) [A100: 108]

Memory:

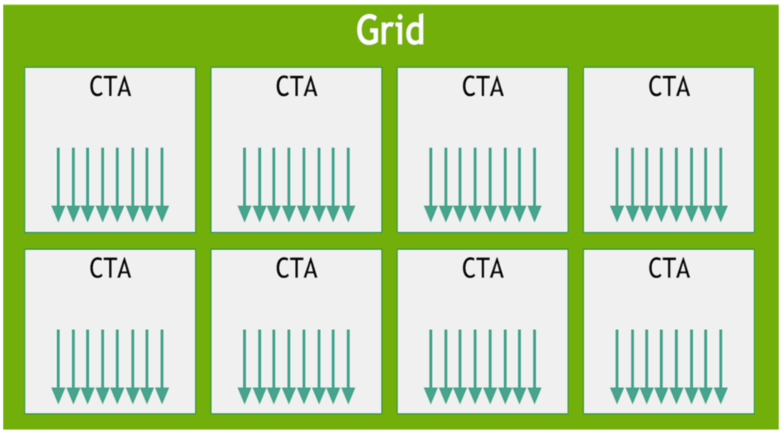
- DRAM [A100: 80GB] - big, slow
- L2 cache [A100: 40MB]
- L1 cache [A100: 192KB per SM] - small, fast

You can look at the specs on your actual GPU.

`print_gpu_specs()`

Basic structure: run f(i) for all i = 0, ..., N-1

Execution model

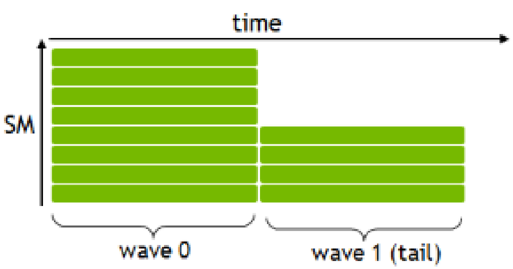


- *Thread*: process individual index (i.e., f(i))
- *Thread block* (a.k.a. concurrent thread arrays): scheduled on a single SM
- *Grid*: collection of thread blocks

Why thread blocks? Shared memory.

- Intuition: group f(i)'s that read similar data together
- Threads within a thread block have shared memory (as fast as L1 cache) [A100: 164KB]
- Can synchronize threads (for reading/writing) within a block (but not across blocks)

Hardware and execution interact.



Thread blocks scheduled onto SMs in waves.
Problem: last wave has fewer thread blocks, leaving some SMs idle (low occupancy).
Wave quantization: make number of thread blocks divide # SMs.
Rule of thumb: number of thread blocks should be >= 4x # SMs
Challenge: some aspects of hardware are hidden from the execution model (e.g., scheduling, # SMs).

Arithmetic intensity: # FLOPs / # bytes

- If high, operation is compute-bound (good)
- If low, operation is memory-bound (bad)

General rule: matrix multiplication is compute-bound, everything else is memory-bound

```
def benchmarking_and_profiling():
```

IMPORTANT: benchmark/profile your code!

You can read spec sheets (marketing material) and papers
...but performance depends on your library version, your hardware, your workload
...so there is no substitute for benchmarking/profiling your code.

Example computation: running forward/backward passes on an MLP.

```
    run_mlp(dim=128, num_layers=16, batch_size=128, num_steps=5)

    benchmarking()       # How long does it take?
    profiling()          # Where time is being spent?
```

Every time you make a change, benchmark/profile!

```
class MLP(nn.Module):
    """Simple MLP: linear -> GeLU -> linear -> GeLU -> ... -> linear -> GeLU"""
    def __init__(self, dim: int, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(dim, dim) for _ in range(num_layers)])

    def forward(self, x: torch.Tensor):
        for layer in self.layers:
            x = layer(x)
            x = torch.nn.functional.gelu(x)
        return x


def run_mlp(dim: int, num_layers: int, batch_size: int, num_steps: int) -> Callable:
    # Define a model (with random weights)
    model = MLP(dim, num_layers).to(get_device())

    # Define an input (random)
    x = torch.randn(batch_size, dim, device=get_device())

    def run():
        # Run the model `num_steps` times (note: no optimizer updates)
        for step in range(num_steps):
            # Forward
            y = model(x).mean()

            # Backward
            y.backward()

    return run


def run_operation1(dim: int, operation: Callable) -> Callable:
    # Setup: create one random dim x dim matrices
```

```
    x = torch.randn(dim, dim, device=get_device())
    # Return a function to perform the operation
    return lambda : operation(x)


def run_operation2(dim: int, operation: Callable) -> Callable:
    # Setup: create two random dim x dim matrices
    x = torch.randn(dim, dim, device=get_device())
    y = torch.randn(dim, dim, device=get_device())
    # Return a function to perform the operation
    return lambda : operation(x, y)


def benchmarking():
```

Benchmarking measures the wall-clock time of performing some operation.

It only gives you end-to-end time, not where time is spent (profiling).

It is still useful for:

- comparing different implementations (which is faster?), and
- understanding how performance scales (e.g., with dimension).

Let's define a convenient function for benchmarking an arbitrary function.

```
    benchmark("sleep", lambda : time.sleep(50 / 1000))
```

Benchmarking matrix multiplication

First, let us benchmark matrix multiplication of square matrices.

```
    if torch.cuda.is_available():
        dims = (1024, 2048, 4096, 8192, 16384)  # @inspect dims
    else:
        dims = (1024, 2048)  # @inspect dims

    matmul_results = []
    for dim in dims:
        # @ inspect dim
        result = benchmark(f"matmul(dim={dim})", run_operation2(dim=dim, operation=lambda a, b: a @ b))
        matmul_results.append((dim, result))  # @inspect matmul_results
```

Let us benchmark our MLP!

```
    dim = 256  # @inspect dim
    num_layers = 4  # @inspect num_layers
    batch_size = 256  # @inspect batch_size
    num_steps = 2  # @inspect num_steps

    mlp_base = benchmark("run_mlp", run_mlp(dim=dim, num_layers=num_layers, batch_size=batch_size, num_steps=num_steps))  # @inspect mlp_base
```

Scale the number of steps.

```
    step_results = []
    for scale in (2, 3, 4, 5):
        result = benchmark(f"run_mlp({scale}x num_steps)",
                           run_mlp(dim=dim, num_layers=num_layers,
                                   batch_size=batch_size, num_steps=scale * num_steps)) # @inspect result, @inspect scale, @inspect num_steps
        step_results.append((scale, result))  # @inspect step_results
```

Scale the number of layers.

```
    layer_results = []
    for scale in (2, 3, 4, 5):
        result = benchmark(f"run_mlp({scale}x num_layers)",
                           run_mlp(dim=dim, num_layers=scale * num_layers,
```

```
                                 batch_size=batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect num_layers, @inspect num_steps
        layer_results.append((scale, result))  # @inspect layer_results

```

Scale the batch size.

```
    batch_results = []
    for scale in (2, 3, 4, 5):
        result = benchmark(f"run_mlp({scale}x batch_size)",
                           run_mlp(dim=dim, num_layers=num_layers,
                                   batch_size=scale * batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect num_layers, @inspect num_steps
        batch_results.append((scale, result))  # @inspect batch_results

```

Scale the dimension.

```
    dim_results = []
    for scale in (2, 3, 4, 5):
        result = benchmark(f"run_mlp({scale}x dim)",
                           run_mlp(dim=scale * dim, num_layers=num_layers,
                                   batch_size=batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect num_layers, @inspect num_steps
        dim_results.append((scale, result))  # @inspect dim_results

```

The timings are not always predictable due to the non-homogenous nature of CUDA kernels, hardware, etc.

You can also use `torch.utils.benchmark`, which provides more amenities.
https://pytorch.org/tutorials/recipes/recipes/benchmark.html
We did not use this to make benchmarking more transparent.

```
def benchmark(description: str, run: Callable, num_warmups: int = 1, num_trials: int = 3):
    """Benchmark `func` by running it `num_trials`, and return all the times."""
    # Warmup: first times might be slower due to compilation, things not cached.
    # Since we will run the kernel multiple times, the timing that matters is steady state.
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA threads to finish (important!)

    # Time it for real now!
    times: list[float] = [] # @inspect times, @inspect description
    for trial in range(num_trials):  # Do it multiple times to capture variance
        start_time = time.time()

        run()  # Actually perform computation
        if torch.cuda.is_available():
            torch.cuda.synchronize()  # Wait for CUDA threads to finish (important!)

        end_time = time.time()
        times.append((end_time - start_time) * 1000) # @inspect times

    mean_time = mean(times) # @inspect mean_time
    return mean_time


def profiling():
```

While benchmarking looks at end-to-end time, profiling looks at where time is spent.
Obvious: profiling helps you understand where time is being spent.
Deeper: profiling helps you understand (what is being called).

PyTorch has a nice built-in profiler https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

Let's profile some code to see what is going on under the hood.

```
    sleep_function = lambda : time.sleep(50 / 1000)
    sleep_profile = profile("sleep", sleep_function)
```

sleep

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
cudaDeviceSynchronize	100.00%	11.610us	100.00%	11.610us	5.805us	2

Self CPU time total: 11.610us

Let's start with some basic operations.

```
add_function = lambda a, b: a + b
add_profile = profile("add", run_operation2(dim=2048, operation=add_function))
```

add

Name	Self CPU %	Self CPU	CPU total ⁝
aten::add	98.02%	1.392ms	99.38⁝
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...	0.00%	0.000us	
cudaLaunchKernel	1.37%	19.392us	1.37⁝
cudaDeviceSynchronize	0.62%	8.734us	0.62⁝

Self CPU time total: 1.420msSelf CUDA time total: 17.119us

```
matmul_function = lambda a, b: a @ b
matmul_profile = profile("matmul", run_operation2(dim=2048, operation=matmul_function))
```

matmul

Name	Self CPU %	Self CPU	CPU total ⁝			
aten::matmul	2.29%	7.520us	97.24⁝			
aten::mm	90.14%	295.387us	94.94⁝			
void cutlass::Kernel2(cutlass_80...	0.00%	0.000us	0.00%	0.000us	0.000us	342.62⁝
cudaDeviceGetAttribute	0.21%	0.690us	0.21⁝			
cuLaunchKernel	4.59%	15.037us	4.59⁝			
cudaDeviceSynchronize	2.76%	9.051us	2.76⁝			

Self CPU time total: 327.685usSelf CUDA time total: 342.620us

```
matmul_function_128 = lambda a, b: a @ b
```

```
matmul_profile_128 = profile("matmul(dim=128)", run_operation2(dim=128, operation=matmul_function_128))
```

matmul(dim=128)

Name	Self CPU %	Self CPU	CPU total %
aten::matmul	1.17%	4.912us	98.24%
aten::mm	42.40%	178.581us	97.07%
sm80_xmma_gemm_f32f32_f32f32_f32_nn_n_tilesize32x32x8_stage3_warpsize1x2x1_ff...	0.00%	0.000us	
cudaFuncGetAttributes	0.96%	4.023us	0.96%
cudaLaunchKernelExC	53.71%	226.207us	53.71%
cudaDeviceSynchronize	1.76%	7.413us	1.76%

Self CPU time total: 421.136usSelf CUDA time total: 4.992us

Observations

- You can see what CUDA kernels are actually being called.
- Different CUDA kernels are invoked depending on the tensor dimensions.

Name of CUDA kernel tells us something about the implementation.

Example: cutlass_80_simt_sgemm_256x128_8x4_nn_align1

- cutlass: NVIDIA's CUDA library for linear algebra
- 256x128: tile size

Let's now look at some composite operations.

```
cdist_function = lambda a, b: torch.cdist(a, b)
cdist_profile = profile("cdist", run_operation2(dim=2048, operation=cdist_function))
```

cdist

Name	Self CPU %	Self CPU	CPU total
aten::cdist	1.38%	27.430us	99.62
aten::_euclidean_dist	2.92%	58.128us	97.28
aten::matmul	0.10%	1.961us	2.51
aten::mm	1.92%	38.220us	2.41
sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize128x128x8_stage3_warpsize2x2x1_...	0.00%	0.000us	
aten::cat	0.88%	17.459us	1.33
void at::native::(anonymous namespace)::CatArrayBatchedCopy_aligned16_contig<...	0.00%	0.000us	
aten::pow	72.92%	1.451ms	84.00
void at::native::vectorized_elementwise_kernel<4, at::native::(anonymous name...	0.00%	0.000us	
aten::sum	1.22%	24.211us	1.77

Self CPU time total: 1.990msSelf CUDA time total: 440.121us

```python
gelu_function = lambda a, b: torch.nn.functional.gelu(a + b)
gelu_profile = profile("gelu", run_operation2(dim=2048, operation=gelu_function))
```

gelu

Name	Self CPU %	Self CPU	CPU total
aten::add	86.27%	1.422ms	98.31
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...	0.00%	0.000us	
aten::gelu	0.74%	12.250us	1.13
void at::native::vectorized_elementwise_kernel<4, at::native::GeluCUDAKernelI...	0.00%	0.000us	
cudaLaunchKernel	12.42%	204.811us	12.42
cudaDeviceSynchronize	0.56%	9.236us	0.56

Self CPU time total: 1.648msSelf CUDA time total: 27.360us

```python
softmax_function = lambda a, b: torch.nn.functional.softmax(a + b, dim=-1)
softmax_profile = profile("softmax", run_operation2(dim=2048, operation=softmax_function))
```

softmax

Name	Self CPU %	Self CPU	CPU total
aten::softmax	0.70%	11.487us	1.85
aten::_softmax	0.73%	11.951us	1.15
void at::native::(anonymous namespace)::cunn_SoftMaxForwardSmem<4, float, flo...	0.00%	0.000us	
aten::add	87.82%	1.434ms	97.71
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...	0.00%	0.000us	
cudaLaunchKernel	10.31%	168.454us	10.31
cudaDeviceSynchronize	0.43%	7.097us	0.43

Self CPU time total: 1.633msSelf CUDA time total: 38.719us

Now let's profile our MLP.

We will also visualize our stack trace using a flame graph, which reveals where time is being spent.

```
if torch.cuda.is_available():
    mlp_profile = profile("mlp", run_mlp(dim=2048, num_layers=64, batch_size=1024, num_steps=2), with_stack=True)
else:
    mlp_profile = profile("mlp", run_mlp(dim=128, num_layers=16, batch_size=128, num_steps=2), with_stack=True)
```

mlp

Name	Self CPU %	Self CPU	CPU total
autograd::engine::evaluate_function: AddmmBackward0	1.72%	672.089us	16.79
AddmmBackward0	1.32%	517.317us	10.87
aten::mm	5.47%	2.141ms	7.96
aten::linear	0.55%	217.124us	12.51
aten::addmm	7.09%	2.773ms	10.67
sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize128x128x8_stage3_warpsize2x2x1_...	0.00%	0.000us	
autograd::engine::evaluate_function: torch::autograd::AccumulateGrad	1.34%	522.511us	5.72
torch::autograd::AccumulateGrad	0.88%	342.816us	4.38

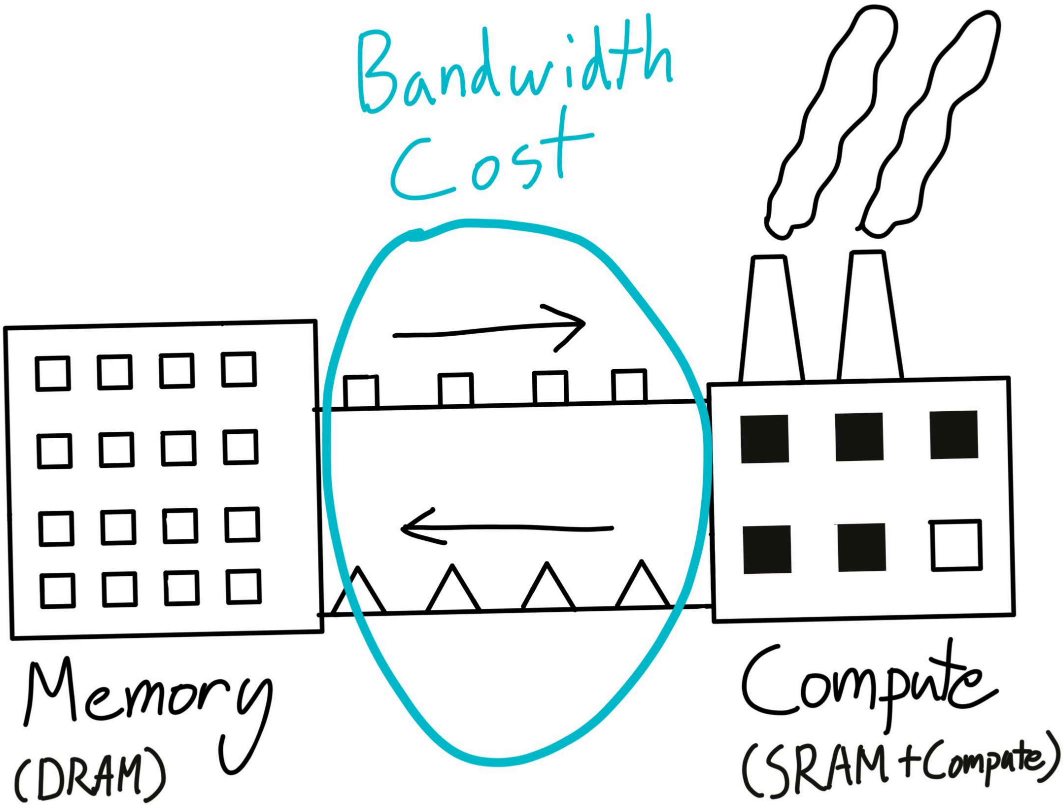
Self CPU time total: 39.129msSelf CUDA time total: 73.598ms

```
def profile(description: str, run: Callable, num_warmups: int = 1, with_stack: bool = False):
    # Warmup
    for _ in range(num_warmups):
```

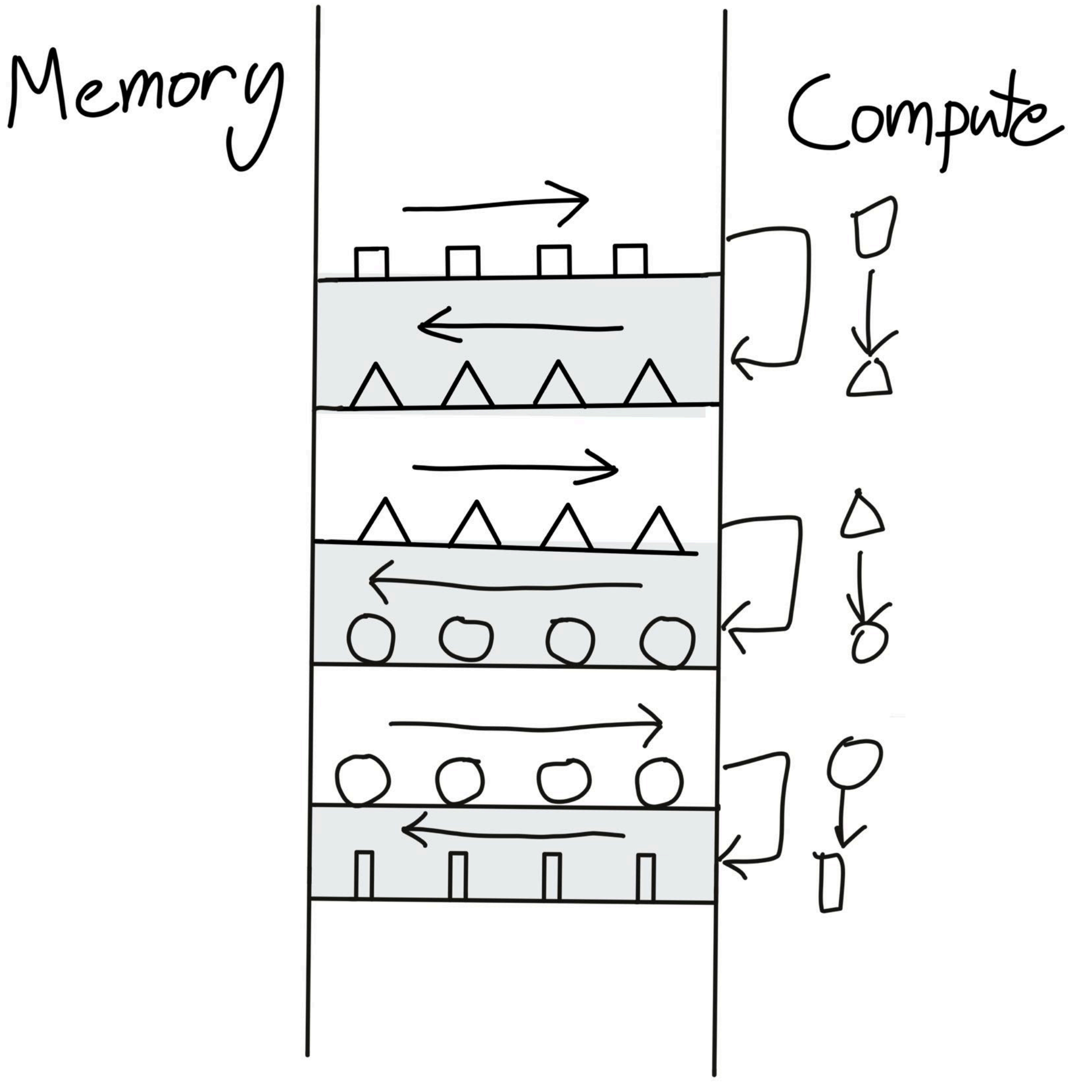
```
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA threads to finish (important!)

    # Run the code with the profiler
    with torch.profiler.profile(
            activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
            # Output stack trace for visualization
            with_stack=with_stack,
            # Needed to export stack trace for visualization
            experimental_config=torch._C._profiler._ExperimentalConfig(verbose=True)) as prof:
        run()
        if torch.cuda.is_available():
            torch.cuda.synchronize()  # Wait for CUDA threads to finish (important!)

    # Print out table
    table = prof.key_averages().table(sort_by="cuda_time_total",
                                      max_name_column_width=80,
                                      row_limit=10)
    #text(f"## {description}")
    #text(table, verbatim=True)

    # Write stack trace visualization
    if with_stack:
        text_path = f"var/stacks_{description}.txt"
        svg_path = f"var/stacks_{description}.svg"
        prof.export_stacks(text_path, "self_cuda_time_total")

    return table

def kernel_fusion_motivation():
```

Horace He's blog post [Article]

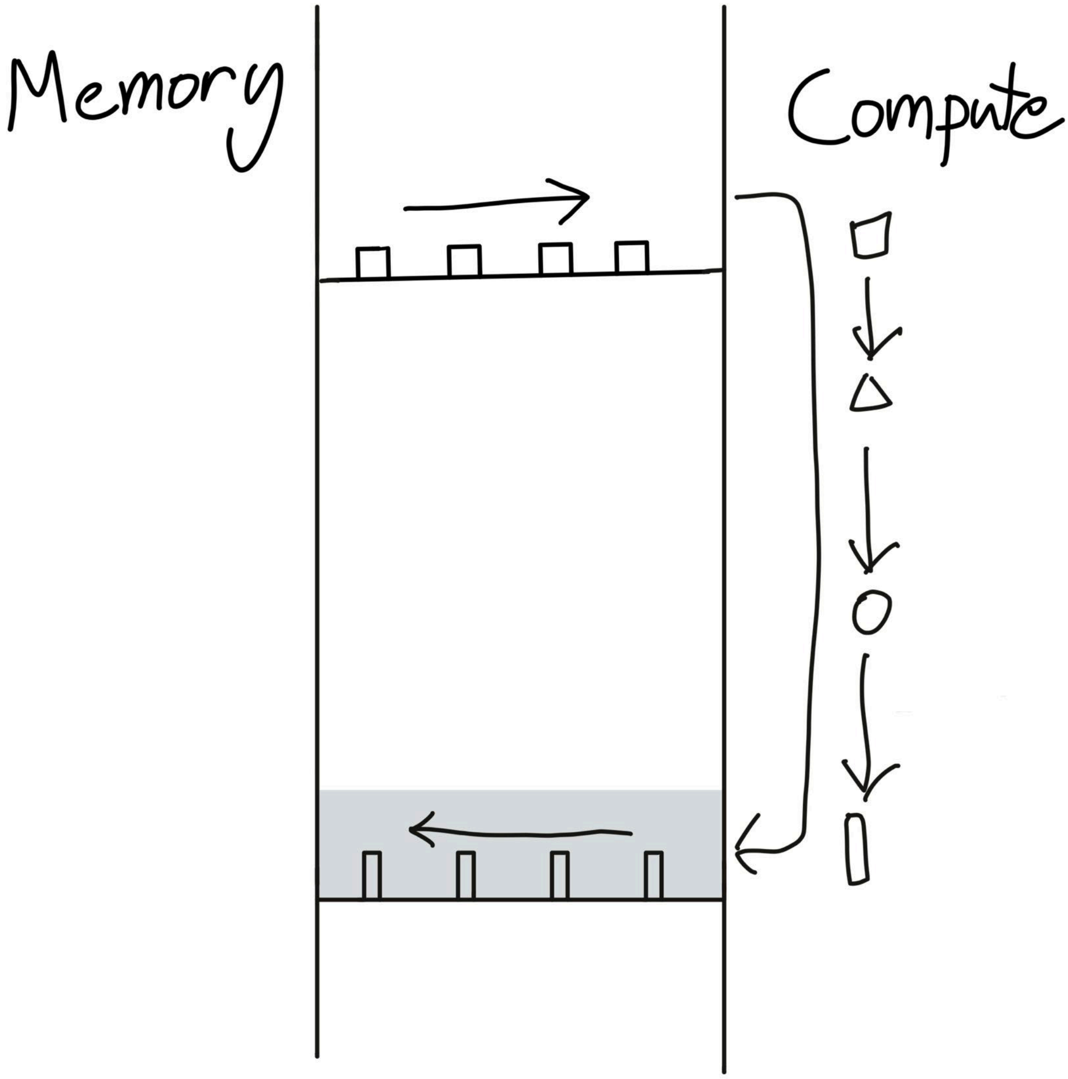
Analogy: warehouse : DRAM :: factory : SRAM



Each operation needs to read/compute/write:



If we *fuse* the operations, only need to read/write once:



To see the effect of fusion, let's consider the GeLU activation function.
https://pytorch.org/docs/stable/generated/torch.nn.GELU.html

Let's consider two ways to compute GeLU:
```
x = torch.tensor([1.])  # @inspect x
```

1. The default PyTorch implementation (fused):
```
y1 = pytorch_gelu(x)  # @inspect y1
```

2. We can also write our own by hand (not fused):
```
y2 = manual_gelu(x)  # @inspect y2

# Check that the implementations match
assert torch.allclose(y1, y2)

# Check more systematically
check_equal(pytorch_gelu, manual_gelu)
```

Let's benchmark.
```
manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
if manual_time is not None and pytorch_time is not None:
```
The fused version is significantly faster: 8.15 ms, 1.13 ms
```
else:
    text("Could not compare times - benchmark results were None")
```

Let's look under the hood.
```
manual_gelu_profile = profile("manual_gelu", run_operation1(dim=16384, operation=manual_gelu))
```

manual_gelu

Name	Self CPU %	Self CPU	CPU total
aten::mul	15.19%	1.479ms	26.01
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<f...	0.00%	0.000us	
aten::add	0.13%	12.473us	0.21
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...	0.00%	0.000us	
aten::tanh	0.07%	6.961us	0.12
void at::native::vectorized_elementwise_kernel<4, at::native::tanh_kernel_cud...	0.00%	0.000us	
cudaLaunchKernel	10.95%	1.066ms	10.95
cudaDeviceSynchronize	73.66%	7.171ms	73.66

Self CPU time total: 9.735msSelf CUDA time total: 7.669ms

```
pytorch_gelu_profile = profile("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu))
```

pytorch_gelu

Name	Self CPU %	Self CPU	CPU total
aten::gelu	71.12%	1.436ms	84.28
void at::native::vectorized_elementwise_kernel<4, at::native::GeluCUDAKernelI...	0.00%	0.000us	
cudaLaunchKernel	13.16%	265.687us	13.16
cudaDeviceSynchronize	15.72%	317.405us	15.72

Self CPU time total: 2.019msSelf CUDA time total: 701.560us

The PyTorch just calls one kernel whereas the others are atomic (remember the warehouse/factory)

Look at Nsight profiler for MLP

```
def cuda_kernels():
```

Now let's open the box to understand what's going on inside a CUDA kernel by writing our own.

Let's write the GeLU function in CUDA.

```
cuda_gelu = create_cuda_gelu() # @inspect cuda_gelu
x = manual_gelu # @inspect x
```

Check correctness of our implementation.

```
if cuda_gelu is not None:
    check_equal(cuda_gelu, manual_gelu)
```

Benchmark our CUDA version.

```
pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
if cuda_gelu is not None:
    cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu)) # @inspect cuda_time
    cuda_gelu_profile = profile("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu))
```

cuda_gelu

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self C
gelu_kernel(float*, float*, int)	0.00%	0.000us	0.00%	0.000us	0.000us	1.66
aten::empty_like	0.20%	6.209us	46.52%	1.428ms	1.428ms	0.00
aten::empty_strided	46.31%	1.422ms	46.31%	1.422ms	1.422ms	0.00
cudaLaunchKernel	8.93%	274.078us	8.93%	274.078us	274.078us	0.00
cudaDeviceSynchronize	44.56%	1.368ms	44.56%	1.368ms	683.944us	0.00

Self CPU time total: 3.070msSelf CUDA time total: 1.664ms

Our CUDA implementation is faster than manual, but not as good as PyTorch.

Elementwise operations are easy in CUDA (though you can still be smarter).
But most interesting operations (e.g., matmul, softmax, RMSNorm) require reading multiple values.
For that, you have to think about managing shared memory, etc.

```
def create_cuda_gelu():
```

CUDA is an extension of C/C++ with APIs for managing GPUs.

Simplified picture: write f(i), CUDA kernel computes f(i) for all i.

Grid: collection of thread blocks: numBlocks = (2, 4), blockDim = (1, 8)
Thread block: collection of threads: blockIdx = (0, 1)
Thread: single unit of operation: threadIdx = (0, 3).

You write code that a thread execute, using (blockIdx, blockDim, threadIdx) to determine what to do.

Set CUDA_LAUNCH_BLOCKING so that if there are errors, CUDA will tell you what went wrong.

```
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"
```

The `load_inline` function makes it convenient to write CUDA code and bind it to a Python module for immediate use.

```
# CUDA code: has the full logic
cuda_gelu_src = open("gelu.cu").read()
```

```
    #include <math.h>#include <torch/extension.h>#include <c10/cuda/CUDAException.h>global void gelu_kernel(float* in, fl

    // Get the index into the tensor

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < num_elements) {  // To handle the case when n < numBlocks * blockDim

        // Do the actual computation

        out[i] = 0.5 * in[i] * (1.0 + tanh(0.79788456 * (in[i] + 0.044715 * in[i] * in[i] * in[i])));

    }

    }inline unsigned int cdiv(unsigned int a, unsigned int b) {

    // Compute ceil(a / b)

    return (a + b - 1) / b;

    }torch::Tensor gelu(torch::Tensor x) {

    TORCH_CHECK(x.device().is_cuda());

    TORCH_CHECK(x.is_contiguous());

    // Allocate empty tensor

    torch::Tensor y = torch::empty_like(x);

    // Determine grid (elements divided into blocks)

    int num_elements = x.numel();

    int block_size = 1024;  // Number of threads

    int num_blocks = cdiv(num_elements, block_size);

    // Launch the kernel

    gelu_kernel<<<num_blocks, block_size>>>(x.data_ptr<float>(), y.data_ptr<float>(), num_elements);

    C10_CUDA_KERNEL_LAUNCH_CHECK();  // Catch errors immediately

    return y;

    }

    # C++ code: defines the gelu function
    cpp_gelu_src = "torch::Tensor gelu(torch::Tensor x);"

```

Compile the CUDA code and bind it to a Python module.

```
    ensure_directory_exists("var/cuda_gelu")
    if not torch.cuda.is_available():
        return None
    module = load_inline(
        cuda_sources=[cuda_gelu_src],
        cpp_sources=[cpp_gelu_src],
        functions=["gelu"],
        extra_cflags=["-O2"],
        verbose=True,
        name="inline_gelu",
        build_directory="var/cuda_gelu",
    )

    cuda_gelu = getattr(module, "gelu")
    return cuda_gelu


def triton_kernels():
    triton_introduction()
    triton_gelu_main()


def triton_introduction():
```

Developed by OpenAI in 2021

https://openai.com/research/triton

Make GPU programming more accessible

- Write in Python
- Think about thread blocks rather than threads

What does Triton offer?

	CUDA	Triton
• Memory coalescing (transfer from DRAM)	manual	automatic
• Shared memory management	manual	automatic
• Scheduling within SMs	manual	automatic
• Scheduling across SMs	manual	manual

Compiler does more work, can actually outperform PyTorch implementations!

```
def triton_gelu_main():
    if not torch.cuda.is_available():
        return
```

One big advantage of Triton is that you can step through the Python code.

Let's step through a Triton kernel.

```
x = torch.randn(8192, device=get_device())
y1 = triton_gelu(x)

print_ptx_main()  # Look at the generated instructions
```

Check that it's correct.

```
check_equal(triton_gelu, manual_gelu)
```

Let's now benchmark it compared to the PyTorch and CUDA implementations.

Remember to set TRITON_INTERPRET=0 for good performance.

```
manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=create_cuda_gelu())) # @inspect cuda_time
triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspect triton_time

triton_gelu_profile = profile("triton_gelu", run_operation1(dim=16384, operation=triton_gelu))
```

triton_gelu

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA
triton_gelu_kernel	0.00%	0.000us	0.00%	0.000us	0.000us	705.240us	100.0
aten::empty_like	0.32%	5.629us	72.49%	1.267ms	1.267ms	0.000us	0.0
aten::empty_strided	72.16%	1.261ms	72.16%	1.261ms	1.261ms	0.000us	0.0
cuLaunchKernel	16.46%	287.632us	16.46%	287.632us	287.632us	0.000us	0.0
cudaDeviceSynchronize	11.05%	193.144us	11.05%	193.144us	96.572us	0.000us	0.0

Self CPU time total: 1.747msSelf CUDA time total: 705.240us

Our Triton implementation (triton_gelu):

- is almost as good as the PyTorch implementation (pytorch_gelu).
- is actually slower than our naive CUDA implementation (cuda_gelu).

Triton operates on blocks, CUDA operates on threads.
Blocks allows Triton compiler to do other optimizations (e.g., thread coarsening).

Everything is way faster than the manual implementation (manual_gelu).

```python
def triton_gelu(x: torch.Tensor):
    assert x.is_cuda
    assert x.is_contiguous()

    # Allocate output tensor
    y = torch.empty_like(x)

    # Determine grid (elements divided into blocks)
    num_elements = x.numel()
    block_size = 1024  # Number of threads
    num_blocks = triton.cdiv(num_elements, block_size)

    triton_gelu_kernel[(num_blocks,)](x, y, num_elements, BLOCK_SIZE=block_size)

    return y


@triton.jit
def triton_gelu_kernel(x_ptr, y_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    # Input is at `x_ptr` and output is at `y_ptr`
    #     |        Block 0            |          Block 1          |      ...      |
    #                            BLOCK_SIZE                                 num_elements

    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE

    # Indices where this thread block should operate
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    # Handle boundary
    mask = offsets < num_elements
```

```
    # Read
    x = tl.load(x_ptr + offsets, mask=mask)

    # Approx gelu is 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
    # Compute (tl.tanh doesn't exist, use tanh(a) = (exp(2a) - 1) / (exp(2a) + 1)
    a = 0.79788456 * (x + 0.044715 * x * x * x)
    exp = tl.exp(2 * a)
    tanh = (exp - 1) / (exp + 1)
    y = 0.5 * x * (1 + tanh)

    # Store
    tl.store(y_ptr + offsets, y, mask=mask)


def print_ptx_main():
```

PTX (parallel thread execution) is like an assembly language for GPUs.

We can see the PTX code generated by Triton.
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

```
    ptx = print_ptx("triton_gelu", triton_gelu_kernel)
```

```
//// Generated by LLVM NVPTX Back-End//.version 8.4.target sm_90a.address_size 64

// .globl       triton_gelu_kernel      // -- Begin function triton_gelu_kernel

                                    // @triton_gelu_kernel

.visible .entry triton_gelu_kernel(

.param .u64 .ptr .global .align 1 triton_gelu_kernel_param_0,

.param .u64 .ptr .global .align 1 triton_gelu_kernel_param_1,

.param .u32 triton_gelu_kernel_param_2

).reqntid 128, 1, 1{

.reg .pred      %p<5>;

.reg .b32       %r<49>;

.reg .f32       %f<113>;

.reg .b64       %rd<8>;

.loc    1 552 0                         // lecture_06.py:552:0

$L__func_begin0:

.loc    1 552 0                         // lecture_06.py:552:0

// %bb.0:

ld.param.u64    %rd5, [triton_gelu_kernel_param_0];

ld.param.u64    %rd6, [triton_gelu_kernel_param_1];

$L__tmp0:

.loc    1 557 24                        // lecture_06.py:557:24

// begin inline asm

mov.u32 %r1, %ctaid.x;

// end inline asm

.loc    1 558 24                        // lecture_06.py:558:24

shl.b32         %r42, %r1, 10;

ld.param.u32    %r43, [triton_gelu_kernel_param_2];

.loc    1 561 41                        // lecture_06.py:561:41

mov.u32         %r44, %tid.x;

shl.b32         %r45, %r44, 2;

and.b32         %r46, %r45, 508;

.loc    1 561 28                        // lecture_06.py:561:28

or.b32          %r47, %r42, %r46;

or.b32          %r48, %r47, 512;

.loc    1 564 21                        // lecture_06.py:564:21

setp.lt.s32     %p1, %r47, %r43;

setp.lt.s32     %p2, %r48, %r43;

.loc    1 567 24                        // lecture_06.py:567:24

mul.wide.s32    %rd7, %r47, 4;

add.s64         %rd1, %rd5, %rd7;

add.s64         %rd2, %rd1, 2048;

.loc    1 567 16                        // lecture_06.py:567:16

// begin inline asm

mov.u32 %r2, 0x0;

mov.u32 %r3, 0x0;

mov.u32 %r4, 0x0;

mov.u32 %r5, 0x0;

@%p1 ld.global.v4.b32 { %r2, %r3, %r4, %r5 }, [ %rd1 + 0 ];
```

```
// end inline asm
mov.b32         %f17, %r2;
mov.b32         %f18, %r3;
mov.b32         %f19, %r4;
mov.b32         %f20, %r5;
// begin inline asm
mov.u32 %r6, 0x0;
mov.u32 %r7, 0x0;
mov.u32 %r8, 0x0;
mov.u32 %r9, 0x0;
@%p2 ld.global.v4.b32 { %r6, %r7, %r8, %r9 }, [ %rd2 + 0 ];
// end inline asm
mov.b32         %f21, %r6;
mov.b32         %f22, %r7;
mov.b32         %f23, %r8;
mov.b32         %f24, %r9;
.loc    1 571 37                        // lecture_06.py:571:37
mul.f32         %f25, %f17, 0f3D372713;
mul.f32         %f26, %f18, 0f3D372713;
mul.f32         %f27, %f19, 0f3D372713;
mul.f32         %f28, %f20, 0f3D372713;
mul.f32         %f29, %f21, 0f3D372713;
mul.f32         %f30, %f22, 0f3D372713;
mul.f32         %f31, %f23, 0f3D372713;
mul.f32         %f32, %f24, 0f3D372713;
.loc    1 571 41                        // lecture_06.py:571:41
mul.f32         %f33, %f25, %f17;
mul.f32         %f34, %f26, %f18;
mul.f32         %f35, %f27, %f19;
mul.f32         %f36, %f28, %f20;
mul.f32         %f37, %f29, %f21;
mul.f32         %f38, %f30, %f22;
mul.f32         %f39, %f31, %f23;
mul.f32         %f40, %f32, %f24;
.loc    1 571 26                        // lecture_06.py:571:26
fma.rn.f32      %f41, %f33, %f17, %f17;
fma.rn.f32      %f42, %f34, %f18, %f18;
fma.rn.f32      %f43, %f35, %f19, %f19;
fma.rn.f32      %f44, %f36, %f20, %f20;
fma.rn.f32      %f45, %f37, %f21, %f21;
fma.rn.f32      %f46, %f38, %f22, %f22;
fma.rn.f32      %f47, %f39, %f23, %f23;
fma.rn.f32      %f48, %f40, %f24, %f24;
.loc    1 571 22                        // lecture_06.py:571:22
mul.f32         %f49, %f41, 0f3F4C422A;
mul.f32         %f50, %f42, 0f3F4C422A;
mul.f32         %f51, %f43, 0f3F4C422A;
mul.f32         %f52, %f44, 0f3F4C422A;
```

```
mul.f32         %f53, %f45, 0f3F4C422A;
mul.f32         %f54, %f46, 0f3F4C422A;
mul.f32         %f55, %f47, 0f3F4C422A;
mul.f32         %f56, %f48, 0f3F4C422A;
.loc    1 572 21                        // lecture_06.py:572:21
fma.rn.f32      %f57, %f41, 0f3F4C422A, %f49;
fma.rn.f32      %f58, %f42, 0f3F4C422A, %f50;
fma.rn.f32      %f59, %f43, 0f3F4C422A, %f51;
fma.rn.f32      %f60, %f44, 0f3F4C422A, %f52;
fma.rn.f32      %f61, %f45, 0f3F4C422A, %f53;
fma.rn.f32      %f62, %f46, 0f3F4C422A, %f54;
fma.rn.f32      %f63, %f47, 0f3F4C422A, %f55;
fma.rn.f32      %f64, %f48, 0f3F4C422A, %f56;
.loc    1 572 17                        // lecture_06.py:572:17
mul.f32         %f2, %f57, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f1, %f2;
// end inline asm
mul.f32         %f4, %f58, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f3, %f4;
// end inline asm
mul.f32         %f6, %f59, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f5, %f6;
// end inline asm
mul.f32         %f8, %f60, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f7, %f8;
// end inline asm
mul.f32         %f10, %f61, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f9, %f10;
// end inline asm
mul.f32         %f12, %f62, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f11, %f12;
// end inline asm
mul.f32         %f14, %f63, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f13, %f14;
// end inline asm
mul.f32         %f16, %f64, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f15, %f16;
// end inline asm
.loc    1 573 18                        // lecture_06.py:573:18
add.f32         %f65, %f1, 0fBF800000;
```

```
add.f32         %f65, %f1, 0fBF800000;
add.f32         %f66, %f3, 0fBF800000;
add.f32         %f67, %f5, 0fBF800000;
add.f32         %f68, %f7, 0fBF800000;
add.f32         %f69, %f9, 0fBF800000;
add.f32         %f70, %f11, 0fBF800000;
add.f32         %f71, %f13, 0fBF800000;
add.f32         %f72, %f15, 0fBF800000;
.loc    1 573 30                        // lecture_06.py:573:30
add.f32         %f73, %f1, 0f3F800000;
add.f32         %f74, %f3, 0f3F800000;
add.f32         %f75, %f5, 0f3F800000;
add.f32         %f76, %f7, 0f3F800000;
add.f32         %f77, %f9, 0f3F800000;
add.f32         %f78, %f11, 0f3F800000;
add.f32         %f79, %f13, 0f3F800000;
add.f32         %f80, %f15, 0f3F800000;
.loc    1 573 24                        // lecture_06.py:573:24
mov.b32         %r11, %f65;
mov.b32         %r12, %f73;
// begin inline asm
div.full.f32 %r10, %r11, %r12;
// end inline asm
mov.b32         %f81, %r10;
mov.b32         %r14, %f66;
mov.b32         %r15, %f74;
// begin inline asm
div.full.f32 %r13, %r14, %r15;
// end inline asm
mov.b32         %f82, %r13;
mov.b32         %r17, %f67;
mov.b32         %r18, %f75;
// begin inline asm
div.full.f32 %r16, %r17, %r18;
// end inline asm
mov.b32         %f83, %r16;
mov.b32         %r20, %f68;
mov.b32         %r21, %f76;
// begin inline asm
div.full.f32 %r19, %r20, %r21;
// end inline asm
mov.b32         %f84, %r19;
mov.b32         %r23, %f69;
mov.b32         %r24, %f77;
// begin inline asm
div.full.f32 %r22, %r23, %r24;
// end inline asm
mov.b32         %f85, %r22;
```

```
mov.b32         %r26, %f70;
mov.b32         %r27, %f78;
// begin inline asm
div.full.f32 %r25, %r26, %r27;
// end inline asm
mov.b32         %f86, %r25;
mov.b32         %r29, %f71;
mov.b32         %r30, %f79;
// begin inline asm
div.full.f32 %r28, %r29, %r30;
// end inline asm
mov.b32         %f87, %r28;
mov.b32         %r32, %f72;
mov.b32         %r33, %f80;
// begin inline asm
div.full.f32 %r31, %r32, %r33;
// end inline asm
mov.b32         %f88, %r31;
.loc    1 574 14                        // lecture_06.py:574:14
mul.f32         %f89, %f17, 0f3F000000;
mul.f32         %f90, %f18, 0f3F000000;
mul.f32         %f91, %f19, 0f3F000000;
mul.f32         %f92, %f20, 0f3F000000;
mul.f32         %f93, %f21, 0f3F000000;
mul.f32         %f94, %f22, 0f3F000000;
mul.f32         %f95, %f23, 0f3F000000;
mul.f32         %f96, %f24, 0f3F000000;
.loc    1 574 23                        // lecture_06.py:574:23
add.f32         %f97, %f81, 0f3F800000;
add.f32         %f98, %f82, 0f3F800000;
add.f32         %f99, %f83, 0f3F800000;
add.f32         %f100, %f84, 0f3F800000;
add.f32         %f101, %f85, 0f3F800000;
add.f32         %f102, %f86, 0f3F800000;
add.f32         %f103, %f87, 0f3F800000;
add.f32         %f104, %f88, 0f3F800000;
.loc    1 574 19                        // lecture_06.py:574:19
mul.f32         %f105, %f89, %f97;
mul.f32         %f106, %f90, %f98;
mul.f32         %f107, %f91, %f99;
mul.f32         %f108, %f92, %f100;
mul.f32         %f109, %f93, %f101;
mul.f32         %f110, %f94, %f102;
mul.f32         %f111, %f95, %f103;
mul.f32         %f112, %f96, %f104;
.loc    1 577 21                        // lecture_06.py:577:21
add.s64         %rd3, %rd6, %rd7;
add.s64         %rd4, %rd3, 2048;
```

```
.loc    1 577 30                        // lecture_06.py:577:30
mov.b32         %r34, %f105;
mov.b32         %r35, %f106;
mov.b32         %r36, %f107;
mov.b32         %r37, %f108;
// begin inline asm
@%p1 st.global.v4.b32 [ %rd3 + 0 ], { %r34, %r35, %r36, %r37 };
// end inline asm
mov.b32         %r38, %f109;
mov.b32         %r39, %f110;
mov.b32         %r40, %f111;
mov.b32         %r41, %f112;
// begin inline asm
@%p2 st.global.v4.b32 [ %rd4 + 0 ], { %r38, %r39, %r40, %r41 };
// end inline asm
.loc    1 577 4                         // lecture_06.py:577:4
ret;
$L__tmp1:$L__func_end0:
                                    // -- End function
}
.file   1 "/home/c-thashim/2025/spring2025-lectures/lecture_06.py"
.section        .debug_abbrev
{
.b8 1                                   // Abbreviation Code.b8 17                              // DW_TAG_compile_
}
.section        .debug_info
{
.b32 76                                 // Length of Unit.b8 2                              // DWARF version num
}
.section        .debug_macinfo  {       }
```

Observations:

- ld.global.* and st.global.* reads and writes from global memory
- %ctaid.x is block index, %tid.x is thread index
- %f* are floating point registers, %r* are integer registers
- One thread processes 8 elements at the same time (thread coarsening)

```python
def print_ptx(name: str, kernel):
    if os.environ.get("TRITON_INTERPRET") == "1":
        text("PTX is not generated when in interpret mode.")
        return

    """Print out the PTX code generated by Triton for the given `kernel`."""
    ptx_path = f"var/{name}-ptx.txt"
```

Let's go poke around at the PTX code.
https://github.com/stanford-cs336/spring2025-lectures/blob/main/var/triton_softmax-ptx.txt

```python
    with open(ptx_path, "w") as f:
        return list(kernel.cache[0].values())[0].asm["ptx"]
```

```
def pytorch_compilation():
```

So far, we have seen three ways to write GeLU:

- Use the default PyTorch function
- Write it in Python manual_gelu
- Write it in CUDA create_cuda_gelu
- Write it in Triton triton_gelu

- Write it in Python and compile it into Triton

```
compiled_gelu = torch.compile(manual_gelu)
```

Check correctness of our implementation.

```
check_equal(compiled_gelu, manual_gelu)

if not torch.cuda.is_available():
    return
```

Let's benchmark and profile it!

```
manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=create_cuda_gelu())) # @inspect cuda_time
triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspect triton_time
compiled_time = benchmark("compiled_gelu", run_operation1(dim=16384, operation=compiled_gelu)) # @inspect compiled_time
```

Let's look under the hood

```
compiled_gelu_profile = profile("compiled_gelu", run_operation1(dim=16384, operation=compiled_gelu))
```

compiled_gelu

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA
Torch-Compiled Region: 0/1	77.92%	1.934ms	91.35%	2.268ms	2.268ms	0.000us
triton_poi_fused_add_mul_tanh_0	1.84%	45.795us	13.43%	333.455us	333.455us	707.261us
triton_poi_fused_add_mul_tanh_0	0.00%	0.000us	0.00%	0.000us	0.000us	707.261us
TorchDynamo Cache Lookup	0.62%	15.371us	0.62%	15.371us	15.371us	0.000us
cuLaunchKernel	11.59%	287.660us	11.59%	287.660us	287.660us	0.000us
cudaDeviceSynchronize	8.03%	199.299us	8.03%	199.299us	99.649us	0.000us

Self CPU time total: 2.482msSelf CUDA time total: 707.261us

```
def triton_softmax_main():
```

So far, we've looked at elementwise operations in Triton (e.g., GeLU).
Now let us look at operations that aggregate over multiple values.

We will roughly follow the Triton fused softmax tutorial: https://triton-lang.org/main/getting-started/tutorials/02-fused-softmax.html

Recall the softmax operation is used in attention and generating probabilities.
Normalize each row of a matrix:

```
[A1 A2 A3]   =>   [A1/A A2/A A3/A]
[B1 B2 B3]   =>   [B1/B B2/B B3/B]
```

Let's first start with the naive implementation and keep track of reads/writes.

```
x = torch.tensor([
    [5., 5, 5],
    [0, 0, 100],
], device=get_device())
y1 = manual_softmax(x) # @inspect y1

if not torch.cuda.is_available():
    return
```

Now let us write the Triton kernel.

```
y2 = triton_softmax(x)
assert torch.allclose(y1, y2)
```

Check our implementations are correct.

```
check_equal2(pytorch_softmax, manual_softmax)
check_equal2(pytorch_softmax, triton_softmax)

compiled_softmax = torch.compile(manual_softmax)
```

Now let's benchmark everything.

```
manual_time = benchmark("manual_softmax", run_operation1(dim=16384, operation=manual_softmax)) # @inspect manual_time
compiled_time = benchmark("compiled_softmax", run_operation1(dim=16384, operation=compiled_softmax)) # @inspect compiled_time
```

```
pytorch_time = benchmark("pytorch_softmax", run_operation1(dim=16384, operation=pytorch_softmax)) # @inspect pytorch_time
triton_time = benchmark("triton_softmax", run_operation1(dim=16384, operation=triton_softmax)) # @inspect triton_time
```

Look under the hood using the profiler.

```
manual_softmax_profile = profile("manual_softmax", run_operation1(dim=16384, operation=manual_softmax))
```

manual_softmax

Name	Self CPU %	Self CPU	CPU total
aten::div	0.28%	8.466us	0.43
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocas...	0.00%	0.000us	
aten::sub	0.44%	13.364us	0.70
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocas...	0.00%	0.000us	
aten::exp	0.25%	7.610us	0.41
void at::native::vectorized_elementwise_kernel<4, at::native::exp_kernel_cuda...	0.00%	0.000us	
aten::max	10.71%	328.962us	19.95
void at::native::reduce_kernel<512, 1, at::native::ReduceOp<float, at::native...	0.00%	0.000us	
aten::sum	0.38%	11.798us	0.67
void at::native::reduce_kernel<512, 1, at::native::ReduceOp<float, at::native...	0.00%	0.000us	

Self CPU time total: 3.071msSelf CUDA time total: 3.258ms

```
compiled_softmax_profile = profile("compiled_softmax", run_operation1(dim=16384, operation=compiled_softmax))
```

compiled_softmax

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self
Torch-Compiled Region: 1/0	56.77%	769.985us	78.99%	1.071ms	1.071ms	0.
triton_red_fused_div_exp_max_sub_sum_0	3.20%	43.382us	22.22%	301.366us	301.366us	730.
triton_red_fused_div_exp_max_sub_sum_0	0.00%	0.000us	0.00%	0.000us	0.000us	730.
TorchDynamo Cache Lookup	0.53%	7.239us	0.53%	7.239us	7.239us	0.
cuLaunchKernel	19.02%	257.984us	19.02%	257.984us	257.984us	0.
cudaDeviceSynchronize	20.48%	277.800us	20.48%	277.800us	138.900us	0.

Self CPU time total: 1.356msSelf CUDA time total: 730.770us

```
pytorch_softmax_profile = profile("pytorch_softmax", run_operation1(dim=16384, operation=pytorch_softmax))
```

pytorch_softmax

Name	Self CPU %	Self CPU	CPU total %
aten::softmax	0.47%	5.061us	28.87
aten::_softmax	13.44%	145.534us	28.40
void at::native::(anonymous namespace)::cunn_SoftMaxForward<4, float, float, ...	0.00%	0.000us	
cudaLaunchKernel	14.96%	161.969us	14.96
cudaDeviceSynchronize	71.13%	770.228us	71.13

Self CPU time total: 1.083msSelf CUDA time total: 1.137ms

```
triton_softmax_profile = profile("triton_softmax", run_operation1(dim=16384, operation=triton_softmax))
```

triton_softmax

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA
triton_softmax_kernel	0.00%	0.000us	0.00%	0.000us	0.000us	705.462us	100.0
aten::empty_like	0.23%	4.238us	77.57%	1.409ms	1.409ms	0.000us	0.0
aten::empty_strided	77.34%	1.405ms	77.34%	1.405ms	1.405ms	0.000us	0.0
cuLaunchKernel	8.02%	145.738us	8.02%	145.738us	145.738us	0.000us	0.0
cudaDeviceSynchronize	14.41%	261.640us	14.41%	261.640us	130.820us	0.000us	0.0

Self CPU time total: 1.816msSelf CUDA time total: 705.462us

Let's end by looking at the PTX code.

```
ptx = print_ptx("triton_softmax", triton_softmax_kernel)
```

```
//// Generated by LLVM NVPTX Back-End//.version 8.4.target sm_90a.address_size 64

// .globl       triton_softmax_kernel   // -- Begin function triton_softmax_kernel
                                    // @triton_softmax_kernel
.visible .entry triton_softmax_kernel(

.param .u64 .ptr .global .align 1 triton_softmax_kernel_param_0,
.param .u64 .ptr .global .align 1 triton_softmax_kernel_param_1,
.param .u32 triton_softmax_kernel_param_2,
.param .u32 triton_softmax_kernel_param_3,
.param .u32 triton_softmax_kernel_param_4
).reqntid 128, 1, 1{

.reg .pred      %p<5>;
.reg .b32       %r<22>;
.reg .f32       %f<13>;
.reg .b64       %rd<10>;
.loc    1 741 0                         // lecture_06.py:741:0
$L__func_begin0:

.loc    1 741 0                         // lecture_06.py:741:0
// %bb.0:

ld.param.u64    %rd3, [triton_softmax_kernel_param_0];
ld.param.u64    %rd4, [triton_softmax_kernel_param_1];
$L__tmp0:

.loc    1 745 28                        // lecture_06.py:745:28
// begin inline asm
mov.u32 %r1, %ctaid.x;
// end inline asm
ld.param.u32    %r8, [triton_softmax_kernel_param_2];
.loc    1 746 31                        // lecture_06.py:746:31
mov.u32         %r9, %tid.x;
and.b32         %r10, %r9, 3;
ld.param.u32    %r11, [triton_softmax_kernel_param_3];
.loc    1 749 36                        // lecture_06.py:749:36
mul.lo.s32      %r12, %r1, %r8;
ld.param.u32    %r13, [triton_softmax_kernel_param_4];
.loc    1 749 26                        // lecture_06.py:749:26
mul.wide.s32    %rd5, %r12, 4;
add.s64         %rd6, %rd3, %rd5;
.loc    1 750 27                        // lecture_06.py:750:27
mul.wide.u32    %rd7, %r10, 4;
add.s64         %rd1, %rd6, %rd7;
.loc    1 751 47                        // lecture_06.py:751:47
setp.lt.s32     %p1, %r10, %r13;
mov.b32         %r3, -8388608;
.loc    1 751 20                        // lecture_06.py:751:20
// begin inline asm
mov.u32 %r2, 0x0;
@%p1 ld.global.b32 { %r2 }, [ %rd1 + 0 ];
@!%p1 mov.u32 %r2, %r3;
```

```
// end inline asm
mov.b32         %f3, %r2;
$L__tmp1:
.loc    2 184 40                        // standard.py:184:40
shfl.sync.bfly.b32      %r14, %r2, 2, 31, -1;
mov.b32         %f4, %r14;
.loc    2 163 27                        // standard.py:163:27
max.f32         %f5, %f3, %f4;
.loc    2 184 40                        // standard.py:184:40
mov.b32         %r15, %f5;
shfl.sync.bfly.b32      %r16, %r15, 1, 31, -1;
mov.b32         %f6, %r16;
.loc    2 163 27                        // standard.py:163:27
max.f32         %f7, %f5, %f6;
$L__tmp2:
.loc    1 754 20                        // lecture_06.py:754:20
sub.f32         %f8, %f3, %f7;
.loc    1 755 23                        // lecture_06.py:755:23
mul.f32         %f2, %f8, 0f3FB8AA3B;
// begin inline asm
ex2.approx.f32 %f1, %f2;
// end inline asm
$L__tmp3:
.loc    2 267 36                        // standard.py:267:36
mov.b32         %r5, %f1;
shfl.sync.bfly.b32      %r17, %r5, 2, 31, -1;
mov.b32         %f9, %r17;
.loc    2 256 15                        // standard.py:256:15
add.f32         %f10, %f1, %f9;
.loc    2 267 36                        // standard.py:267:36
mov.b32         %r18, %f10;
shfl.sync.bfly.b32      %r19, %r18, 1, 31, -1;
mov.b32         %f11, %r19;
.loc    2 256 15                        // standard.py:256:15
add.f32         %f12, %f10, %f11;
$L__tmp4:
.loc    1 757 24                        // lecture_06.py:757:24
mov.b32         %r6, %f12;
// begin inline asm
div.full.f32 %r7, %r5, %r6;
// end inline asm
.loc    1 760 36                        // lecture_06.py:760:36
mul.lo.s32      %r20, %r1, %r11;
.loc    1 760 26                        // lecture_06.py:760:26
mul.wide.s32    %rd8, %r20, 4;
add.s64         %rd9, %rd4, %rd8;
.loc    1 761 27                        // lecture_06.py:761:27
```

```
add.s64         %rd2, %rd9, %rd7;

.loc    1 762 21                        // lecture_06.py:762:21

and.b32         %r21, %r9, 124;

setp.eq.s32     %p4, %r21, 0;

and.pred        %p3, %p4, %p1;

// begin inline asm

@%p3 st.global.b32 [ %rd2 + 0 ], { %r7 };

// end inline asm

.loc    1 762 4                         // lecture_06.py:762:4

ret;

$L__tmp5:$L__func_end0:

                                    // -- End function

}

.file   1 "/home/c-thashim/2025/spring2025-lectures/lecture_06.py"

.file   2 "/home/c-thashim/2025/spring2025-lectures/.venv/lib/python3.10/site-packages/triton/language/standard.py"

.section        .debug_abbrev

{

.b8 1                               // Abbreviation Code.b8 17                              // DW_TAG_compile_

}

.section        .debug_info

{

.b32 173                            // Length of Unit.b8 2                                 // DWARF version num

}

.section        .debug_macinfo  {       }
```

```python


def manual_softmax(x: torch.Tensor):
    # M: number of rows, N: number of columns
    M, N = x.shape

    # Compute the max of each row (MN reads, M writes)
    x_max = x.max(dim=1)[0]

    # Subtract off the max (MN + M reads, MN writes)
    x = x - x_max[:, None]

    # Exponentiate (MN reads, MN writes)
    numerator = torch.exp(x)

    # Compute normalization constant (MN reads, M writes)
    denominator = numerator.sum(dim=1)

    # Normalize (MN reads, MN writes)
    y = numerator / denominator[:, None]

    # Total: 5MN + M reads, 3MN + 2M writes
    # In principle, should have MN reads, MN writes (speedup of 4x!)
    return y


def triton_softmax(x: torch.Tensor):
    # Allocate output tensor
    y = torch.empty_like(x)

    # Determine grid
```

```
    M, N = x.shape                          # Number of rows x number of columns
    block_size = triton.next_power_of_2(N)  # Each block contains all the columns
    num_blocks = M                          # Each block is a row

    # Launch kernel
    triton_softmax_kernel[(M,)](
        x_ptr=x, y_ptr=y,
        x_row_stride=x.stride(0), y_row_stride=y.stride(0),
        num_cols=N, BLOCK_SIZE=block_size
    )

    return y


@triton.jit
def triton_softmax_kernel(x_ptr, y_ptr, x_row_stride, y_row_stride, num_cols, BLOCK_SIZE: tl.constexpr):
    assert num_cols <= BLOCK_SIZE

    # Process each row independently
    row_idx = tl.program_id(0)
    col_offsets = tl.arange(0, BLOCK_SIZE)

    # Read from global memory
    x_start_ptr = x_ptr + row_idx * x_row_stride
    x_ptrs = x_start_ptr + col_offsets
    x_row = tl.load(x_ptrs, mask=col_offsets < num_cols, other=float("-inf"))

    # Compute
    x_row = x_row - tl.max(x_row, axis=0)
    numerator = tl.exp(x_row)
    denominator = tl.sum(numerator, axis=0)
    y_row = numerator / denominator

    # Write back to global memory
    y_start_ptr = y_ptr + row_idx * y_row_stride
    y_ptrs = y_start_ptr + col_offsets
    tl.store(y_ptrs, y_row, mask=col_offsets < num_cols)


def triton_matmul_main():
    text("Matrix multipliction is perhaps the most optimized algorithm ever.")

    text("If you write matrix multiplication in CUDA, there's all sorts of crazy things you have to do.")
    link("https://github.com/openai/blocksparse/blob/master/src/matmul_op_gpu.cu")

    text("It's much easier in Triton.")
    link("https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html")

    text("       k                  j                    ", verbatim=True)
    text("  [ A1 A2 A3 ]       [ B1 B2 B3 ]   [ C1 C2 C3 ]", verbatim=True)
    text("i [ A4 A5 A6 ]  *  k [ B4 B5 B6 ] = [ C4 C5 C6 ]", verbatim=True)
    text("  [ A7 A8 A9 ]       [ B7 B8 B9 ]   [ C7 C8 C9 ]", verbatim=True)

    text("Naively: need MKN reads, MN writes")

    text("Computing C4 and C5 both need A4, A5, A6.")
    text("Can we read A4, A5, A6 from DRAM once to compute both?")
    text("Answer: yes, using shared memory!")

    text("## Tiling (leveraging shared memory)")

    text("Recall that shared memory is:")
```

```
    text("- fast (10x faster) and small(~100KB)")
    text("- shared between all the threads in a block.")
    image("https://miro.medium.com/v2/resize:fit:2000/format:webp/1*6xoBKi5kL2dZpivFe1-zgw.jpeg")

    text("Trivial: for small matrices, load all of A and B into shared memory, then could compute C.")
    text("Now we get MK + KN reads, MN writes")

    text("But what if we have big matrices...")

    image("https://www.researchgate.net/profile/Axel-Huebl/publication/320499173/figure/fig1/AS:614298980196359@1523471698396/Performance-critical-A-B-part-of-the-GEMM-using-a-tiling-strategy-A-thread-iterates.png", width=0.5)
    text("Key idea: divide the matrix into blocks.")
    text("For each block of A and block of B:")
    text("- load into shared memory,")
    text("- do mini-matrix multiplication,")
    text("- write the partial sum.")

    text("Animation of tiled matrix multiplication "), link("https://youtu.be/aMvCEEBIBto")

    text("## Leveraging L2 cache")

    text("Two ways of computing 9 elements of a matrix:")
    image("https://triton-lang.org/main/_images/grouped_vs_row_major_ordering.png", width=0.5)
    text("1. Loads 9 + 81 = 90 blocks")
    text("1. Loads 27 + 27 = 54 blocks")

    text("Process the blocks in an order that minimizes the reads.")

    text("Why write your own kernel for matrix multiplication (e.g., A @ B)?")
    text("Answer: fusion with another operation (e.g., gelu(A @ B))")

    if not torch.cuda.is_available():
        return
    text("Let's try it!")
    benchmark("pytorch_matmul", run_operation2(dim=16384, operation=torch.matmul))
    benchmark("triton_matmul", run_operation2(dim=16384, operation=triton_matmul))

    # Not working for some reason
    #print_ptx("triton_matmul", triton_matmul_kernel)


def further_reading():
```

Horace He's blog post [Article]

CUDA MODE Lecture 1: how to profile CUDA kernels in PyTorch [Video]
CUDA MODE Lecture 2: Chapters 1-3 of PPMP book [Video]
CUDA MODE Lecture 3: Getting started with CUDA for Python Programmers [Video]
CUDA MODE Lecture 4: Compute and memory basics [Video]
CUDA MODE Lecture 8: CUDA performance checklist [Video]

HetSys Course: Lecture 1: Programming heterogenous computing systems with GPUs [Video]
HetSys Course: Lecture 2: SIMD processing and GPUs [Video]
HetSys Course: Lecture 3: GPU Software Hierarchy [Video]
HetSys Course: Lecture 4: GPU Memory Hierarchy [Video]
HetSys Course: Lecture 5: GPU performance considerations [Video]

[A100 GPU with NVIDIA Ampere Architecture]
[NVIDIA Deep Learning Performance Guide]
[GPU Puzzles]
[Triton Paper]
[PyTorch 2.0 Acceleration]

```
############################################################
```

```python

def print_gpu_specs():
    num_devices = torch.cuda.device_count()  # @inspect num_devices
```

8 devices

```python
    for i in range(num_devices):
        properties = torch.cuda.get_device_properties(i)  # @inspect properties
```

7: _CudaDeviceProperties(name='NVIDIA H100 80GB HBM3', major=9, minor=0, total_memory=81090MB, multi_processor_count=132, uuid=62f395b0-f63d-2a9d-d202-53f798ada4f4, L2_cache_size=50MB)

```python


def pytorch_softmax(x: torch.Tensor):
    return torch.nn.functional.softmax(x, dim=-1)


def pytorch_gelu(x: torch.Tensor):
    # Use the tanh approximation to match our implementation
    return torch.nn.functional.gelu(x, approximate="tanh")


def manual_gelu(x: torch.Tensor):
    return 0.5 * x * (1 + torch.tanh(0.79788456 * (x + 0.044715 * x * x * x)))




if __name__ == "__main__":
    main()
```