



lecture_06.py

```

1 import time
2 from typing import Callable
3 import torch
4 import torch.nn as nn
5 from torch.profiler import ProfilerActivity
6 from torch.utils.cpp_extension import load_inline
7 import triton
8 import triton.language as tl
9 from execute_util import text, link, image
10 from file_util import ensure_directory_exists
11 from lecture_util import article_link
12 from torch_util import get_device
13 from lecture_06_utils import check_equal, check_equal2, get_local_url, round1, mean
14 import os
15
16 def main():
17     announcements()
18
19     Last lecture: high-level overview of GPUs and performance
20     This lecture: benchmarking/profiling + write kernels
21
22     if not torch.cuda.is_available():
23         text("You should run this lecture on a GPU to get the full experience.")
24
25     review_of_gpus()
26     benchmarking_and_profiling() # Important for understanding!
27
28     kernel_fusion_motivation()
29     cuda_kernels() # Write kernels in CUDA/C++
30     triton_kernels() # Write kernels in Python
31     pytorch_compilation() # Don't write kernels at all?
32
33     # More advanced computations
34     triton_softmax_main()
35
36 Summary
37
38 Gap between the programming model (PyTorch, Triton, PTX) and hardware => performance mysteries
39
40 Benchmarking for understanding scaling
41 Profiling for understanding internals of PyTorch functions (bottoms out with kernels)
42 Looking at PTX assembly to understand internals of CUDA kernels
43
44 5 ways to write a function: manual, PyTorch, compiled, CUDA, Triton
45 GeLU (element-wise), softmax (row-wise), matmul (complex aggregation)
46
47 Key principle: organize computation to minimize reads/writes
48 Key ideas: kernel fusion (warehouse/factory analogy), tiling (shared memory)
49 Automatic compilers (Triton, torch.compile) will get better over time
50
51 further_reading()
52
53
54 def announcements():
55     Assignment 1 leaderboard [Leaderboard]
56     Assignment 2 is out [A2]
57
58
59 def review_of_gpus():

```

60

Hardware

61



GA100 Full GPU with 128 SMs

SM

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

63 Memory:

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

68 You can look at the specs on your actual GPU.

69 `print_gpu_specs()`

70

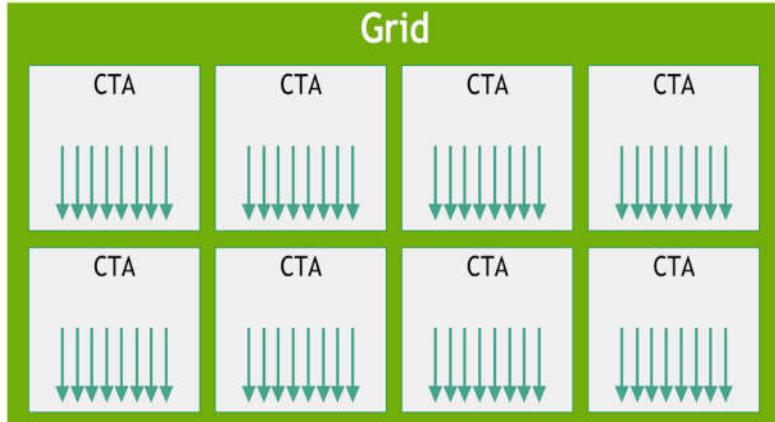
71 Basic structure: run $f(i)$ for all $i = 0, \dots, N-1$

72

73

Execution model

74



75

- *Thread*: process individual index (i.e., $f(i)$)

76 • *Thread block* (a.k.a. concurrent thread arrays): scheduled on a single SM77 • *Grid*: collection of thread blocks

78

79 Why thread blocks? Shared memory.

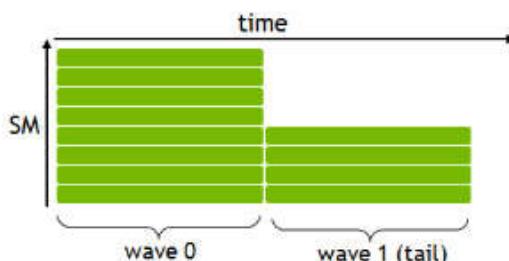
- Intuition: group $f(i)$'s that read similar data together
- Threads within a thread block have shared memory (as fast as L1 cache) [A100: 164KB]
- Can synchronize threads (for reading/writing) within a block (but not across blocks)

83

84

Hardware and execution interact.

85



```

86     Thread blocks scheduled onto SMs in waves.
87     Problem: last wave has fewer thread blocks, leaving some SMs idle (low occupancy).
88     Wave quantization: make number of thread blocks divide # SMs.
89     Rule of thumb: number of thread blocks should be >= 4x # SMs.
90     Challenge: some aspects of hardware are hidden from the execution model (e.g., scheduling, # SMs).
91
92

```

Arithmetic intensity: # FLOPs / # bytes

```

93     • If high, operation is compute-bound (good)
94     • If low, operation is memory-bound (bad)
95     General rule: matrix multiplication is compute-bound, everything else is memory-bound
96
97
98 def benchmarking_and_profiling():
99     IMPORTANT: benchmark/profile your code!
100
101    You can read spec sheets (marketing material) and papers
102    ...but performance depends on your library version, your hardware, your workload
103    ...so there is no substitute for benchmarking/profiling your code.
104
105    Example computation: running forward/backward passes on an MLP.
106    run_mlp(dim=128, num_layers=16, batch_size=128, num_steps=5)
107
108    benchmarking()      # How long does it take?
109    profiling()        # Where time is being spent?
110
111    Every time you make a change, benchmark/profile!
112
113
114 class MLP(nn.Module):
115     """Simple MLP: linear -> GeLU -> linear -> GeLU -> ... -> linear -> GeLU"""
116     def __init__(self, dim: int, num_layers: int):
117         super().__init__()
118         self.layers = nn.ModuleList([nn.Linear(dim, dim) for _ in range(num_layers)])
119
120     def forward(self, x: torch.Tensor):
121         for layer in self.layers:
122             x = layer(x)
123             x = torch.nn.functional.gelu(x)
124
125         return x
126
127 def run_mlp(dim: int, num_layers: int, batch_size: int, num_steps: int) -> Callable:
128     # Define a model (with random weights)
129     model = MLP(dim, num_layers).to(get_device())
130
131     # Define an input (random)
132     x = torch.randn(batch_size, dim, device=get_device())
133
134     def run():
135         # Run the model `num_steps` times (note: no optimizer updates)
136         for step in range(num_steps):
137             # Forward
138             y = model(x).mean()
139
140             # Backward
141             y.backward()
142
143     return run
144
145
146 def run_operation1(dim: int, operation: Callable) -> Callable:
147     # Setup: create one random dim x dim matrices

```

```

148     x = torch.randn(dim, dim, device=get_device())
149     # Return a function to perform the operation
150     return lambda : operation(x)
151
152
153 def run_operation2(dim: int, operation: Callable) -> Callable:
154     # Setup: create two random dim x dim matrices
155     x = torch.randn(dim, dim, device=get_device())
156     y = torch.randn(dim, dim, device=get_device())
157     # Return a function to perform the operation
158     return lambda : operation(x, y)
159
160
161 def benchmarking():
162     Benchmarking measures the wall-clock time of performing some operation.
163
164     It only gives you end-to-end time, not where time is spent (profiling).
165
166     It is still useful for:
167     • comparing different implementations (which is faster?), and
168     • understanding how performance scales (e.g., with dimension).
169
170     Let's define a convenient function for benchmarking an arbitrary function.
171     benchmark("sleep", lambda : time.sleep(50 / 1000))
172
173

```

Benchmarking matrix multiplication

```

174 First, let us benchmark matrix multiplication of square matrices.
175 if torch.cuda.is_available():
176     dims = (1024, 2048, 4096, 8192, 16384) # @inspect dims
177 else:
178     dims = (1024, 2048) # @inspect dims
179
180 matmul_results = []
181 for dim in dims:
182     # @ inspect dim
183     result = benchmark(f"matmul(dim={dim})", run_operation2(dim=dim, operation=lambda a, b: a @ b))
184     matmul_results.append((dim, result)) # @inspect matmul_results
185
186 Let us benchmark our MLP!
187 dim = 256 # @inspect dim
188 num_layers = 4 # @inspect num_layers
189 batch_size = 256 # @inspect batch_size
190 num_steps = 2 # @inspect num_steps
191
192 mlp_base = benchmark("run_mlp", run_mlp(dim=dim, num_layers=num_layers, batch_size=batch_size, num_steps=num_steps)) # @inspect
mlp_base
193
194
195 Scale the number of steps.
196 step_results = []
197 for scale in (2, 3, 4, 5):
198     result = benchmark(f"run_mlp({scale}x num_steps)",
199                         run_mlp(dim=dim, num_layers=num_layers,
200                                 batch_size=batch_size, num_steps=scale * num_steps)) # @inspect result, @inspect scale, @inspect
num_steps
201     step_results.append((scale, result)) # @inspect step_results
202
203 Scale the number of layers.
204 layer_results = []
205 for scale in (2, 3, 4, 5):
206     result = benchmark(f"run_mlp({scale}x num_layers)",
207                         run_mlp(dim=dim, num_layers=scale * num_layers,
208                                 batch_size=batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect
num_layers
209     layer_results.append((scale, result)) # @inspect layer_results
210

```

```

208         batch_size=batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect num_layers,
209         @inspect num_steps
210
211     Scale the batch size.
212     batch_results = []
213     for scale in (2, 3, 4, 5):
214         result = benchmark(f"run_mlp({{scale}}x batch_size)",
215                             run_mlp(dim=dim, num_layers=num_layers,
216                                     batch_size=scale * batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect
217         batch_results.append((scale, result)) # @inspect batch_results
218
219     Scale the dimension.
220     dim_results = []
221     for scale in (2, 3, 4, 5):
222         result = benchmark(f"run_mlp({{scale}}x dim)",
223                             run_mlp(dim=scale * dim, num_layers=num_layers,
224                                     batch_size=batch_size, num_steps=num_steps)) # @inspect result, @inspect scale, @inspect num_layers,
225         dim_results.append((scale, result)) # @inspect dim_results
226
227     The timings are not always predictable due to the non-homogenous nature of CUDA kernels, hardware, etc.
228
229     You can also use torch.utils.benchmark, which provides more amenities.
230     https://pytorch.org/tutorials/recipes/recipes/benchmark.html
231     We did not use this to make benchmarking more transparent.
232
233
234 def benchmark(description: str, run: Callable, num_warmups: int = 1, num_trials: int = 3):
235     """Benchmark `func` by running it `num_trials`, and return all the times."""
236     # Warmup: first times might be slower due to compilation, things not cached.
237     # Since we will run the kernel multiple times, the timing that matters is steady state.
238     for _ in range(num_warmups):
239         run()
240     if torch.cuda.is_available():
241         torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
242
243     # Time it for real now!
244     times: List[float] = [] # @inspect times, @inspect description
245     for trial in range(num_trials): # Do it multiple times to capture variance
246         start_time = time.time()
247
248         run() # Actually perform computation
249         if torch.cuda.is_available():
250             torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
251
252         end_time = time.time()
253         times.append((end_time - start_time) * 1000) # @inspect times
254
255     mean_time = mean(times) # @inspect mean_time
256     return mean_time
257
258
259 def profiling():
260     While benchmarking looks at end-to-end time, profiling looks at where time is spent.
261     Obvious: profiling helps you understand where time is being spent.
262     Deeper: profiling helps you understand (what is being called).
263
264     PyTorch has a nice built-in profiler https://pytorch.org/tutorials/recipes/recipes/profiler\_recipe.html
265
266     Let's profile some code to see what is going on under the hood.
267     sleep_function = lambda : time.sleep(50 / 1000)
268     sleep_profile = profile("sleep", sleep_function)

```

269

sleep

270

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

271

272

273 Let's start with some basic operations.

274 add_function = **lambda** a, b: a + b
275 add_profile = profile("add", run_operation2(dim=2048, operation=add_function))276 **add**

277

Name	Self CPU %	Self CPU	CPU total %	CPU to
aten::add	98.02%	1.392ms	99.38%	1.41
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...>	0.00%	0.000us	0.00%	
cudaLaunchKernel	1.37%	19.392us	1.37%	19.39
cudaDeviceSynchronize	0.62%	8.734us	0.62%	8.73

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

278

279 matmul_function = **lambda** a, b: a @ b
280 matmul_profile = profile("matmul", run_operation2(dim=2048, operation=matmul_function))281 **matmul**

282

Name	Self CPU %	Self CPU	CPU total %	CPU to
aten::matmul	2.29%	7.520us	97.24%	318.63
aten::mm	90.14%	295.387us	94.94%	311.11
void cutlass::Kernel2(cutlass_80...)	0.00%	0.000us	0.00%	0.000us
cudaDeviceGetAttribute	0.21%	0.690us	0.21%	0.69
cuLaunchKernel	4.59%	15.037us	4.59%	15.03
cudaDeviceSynchronize	2.76%	9.051us	2.76%	9.05

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

283

284 matmul_function_128 = **lambda** a, b: a @ b

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

matmul(dim=128)

```
287
```

Name	Self CPU %	Self CPU	CPU total %	CPU to
aten::matmul	1.17%	4.912us	98.24%	413.72
aten::mm	42.40%	178.581us	97.07%	408.81
sm80_xmma_gemm_f32f32_f32f32_f32_nn_n_tilesize32x32x8_stage3_warpsize1x2x1_ff...	0.00%	0.000us	0.00%	
cudaFuncGetAttributes	0.96%	4.023us	0.96%	4.02
cudaLaunchKernelExC	53.71%	226.207us	53.71%	226.20
cudaDeviceSynchronize	1.76%	7.413us	1.76%	7.41

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

```
288
```

Observations

```
289
```

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

```
290
```

293 Name of CUDA kernel tells us something about the implementation.

```
294
```

Example: cutlass_80_simt_sgemm_256x128_8x4_nn_align1

```
295
```

- cutlass: NVIDIA's CUDA library for linear algebra

```
296
```

- 256x128: tile size

```
297
```

298 Let's now look at some composite operations.

```
299
```

cdist_function = `lambda a, b: torch.cdist(a, b)`

```
300
```

cdist_profile = profile("cdist", run_operation2(dim=2048, operation=cdist_function))

```
301
```

cdist

302

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::cdist	1.38%	27.430us	99.62%	1.98
	aten::_euclidean_dist	2.92%	58.128us	97.28%	1.93
	aten::matmul	0.10%	1.961us	2.51%	49.89
	aten::mm	1.92%	38.220us	2.41%	47.93
	sm80_xmma_gemm_f32f32_f32f32_f32_tn_n_tilesize128x128x8_stage3_warpsize2x2x1_...	0.00%	0.000us	0.00%	
	aten::cat	0.88%	17.459us	1.33%	26.45
	void at::native::(anonymous namespace)::CatArrayBatchedCopy_aligned16_contig<...	0.00%	0.000us	0.00%	
	aten::pow	72.92%	1.451ms	84.00%	1.67
	void at::native::vectorized_elementwise_kernel<4, at::native::(anonymous name...	0.00%	0.000us	0.00%	
	aten::sum	1.22%	24.211us	1.77%	35.27

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

303

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

gelu

307

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::add	86.27%	1.422ms	98.31%	1.62
	void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...	0.00%	0.000us	0.00%	
	aten::gelu	0.74%	12.250us	1.13%	18.67
	void at::native::vectorized_elementwise_kernel<4, at::native::GeluCUDAKernelI...	0.00%	0.000us	0.00%	
	cudaLaunchKernel	12.42%	204.811us	12.42%	204.81
	cudaDeviceSynchronize	0.56%	9.236us	0.56%	9.23

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

308

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

softmax

312

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::softmax	0.70%	11.487us	1.85%	30.25
	aten::_softmax	0.73%	11.951us	1.15%	18.76
void at::native::(anonymous namespace)::cunn_SoftMaxForwardSmem<4, float, flo...		0.00%	0.000us	0.00%	
	aten::add	87.82%	1.434ms	97.71%	1.59
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add...		0.00%	0.000us	0.00%	
	cudaLaunchKernel	10.31%	168.454us	10.31%	168.45
	cudaDeviceSynchronize	0.43%	7.097us	0.43%	7.09

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

313

Now let's profile our MLP.

314

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

315

```
if torch.cuda.is_available():
```

316

```
    mlp_profile = profile("mlp", run_mlp(dim=2048, num_layers=64, batch_size=1024, num_steps=2), with_stack=True)
```

317

```
else:
```

318

```
    mlp_profile = profile("mlp", run_mlp(dim=128, num_layers=16, batch_size=128, num_steps=2), with_stack=True)
```

319

mlp

320

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	autograd::engine::evaluate_function: AddmmBackward0	1.72%	672.089us	16.79%	6.57
	AddmmBackward0	1.32%	517.317us	10.87%	4.25
	aten::mm	5.47%	2.141ms	7.96%	3.11
	aten::linear	0.55%	217.124us	12.51%	4.89
	aten::addmm	7.09%	2.773ms	10.67%	4.17
sm80_xmma_gemm_f32f32_f32f32_f32_th_n_tilesize128x128x8_stage3_warpsize2x2x1_...		0.00%	0.000us	0.00%	
autograd::engine::evaluate_function: torch::autograd::AccumulateGrad		1.34%	522.511us	5.72%	2.23
	torch::autograd::AccumulateGrad	0.88%	342.816us	4.38%	1.71

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

321

322

323

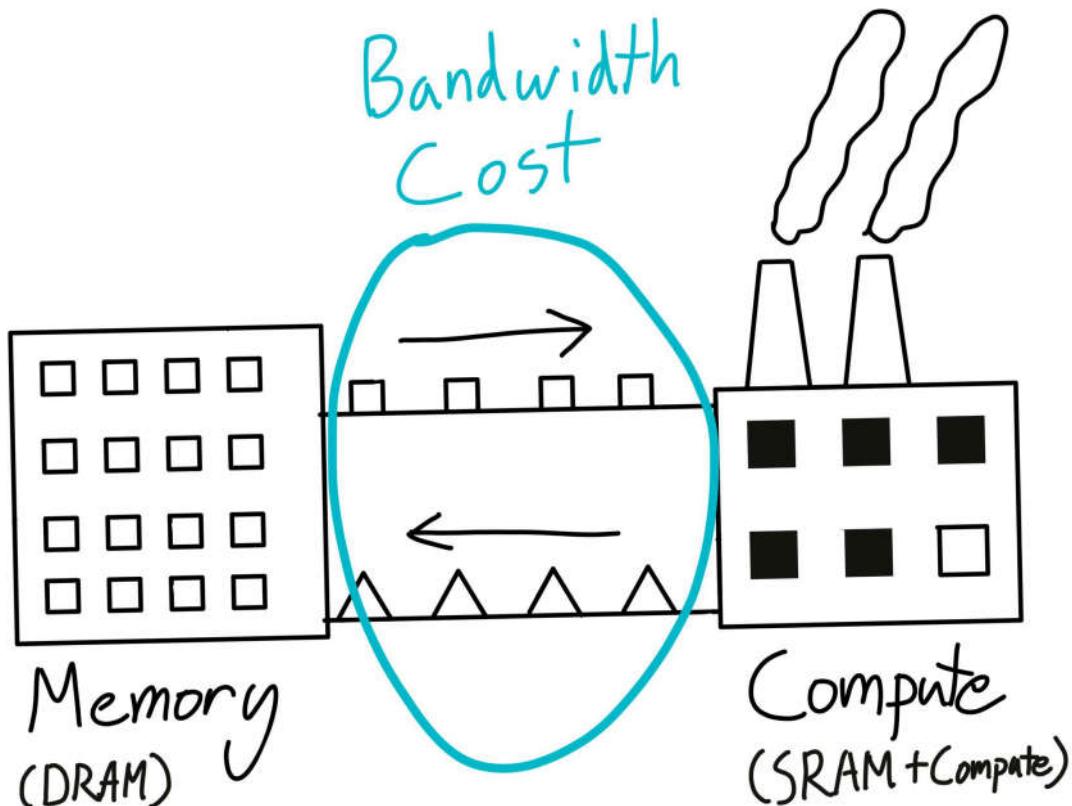
```
def profile(description: str, run: Callable, num_warmups: int = 1, with_stack: bool = False):
    # Warmup
```

```
    for _ in range(num_warmups):
```

```

327     run()
328     if torch.cuda.is_available():
329         torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
330
331     # Run the code with the profiler
332     with torch.profiler.profile(
333         activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
334         # Output stack trace for visualization
335         with_stack=with_stack,
336         # Needed to export stack trace for visualization
337         experimental_config=torch._C._profiler._ExperimentalConfig(verbose=True)) as prof:
338         run()
339         if torch.cuda.is_available():
340             torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
341
342     # Print out table
343     table = prof.key_averages().table(sort_by="cuda_time_total",
344                                         max_name_column_width=80,
345                                         row_limit=10)
346     #text(f"## {description}")
347     #text(table, verbatim=True)
348
349     # Write stack trace visualization
350     if with_stack:
351         text_path = f"var/stacks_{description}.txt"
352         svg_path = f"var/stacks_{description}.svg"
353         prof.export_stacks(text_path, "self_cuda_time_total")
354
355     return table
356
357 def kernel_fusion_motivation():
358     Horace He's blog post \[Article\]
359
360     Analogy: warehouse : DRAM :: factory : SRAM
361

```

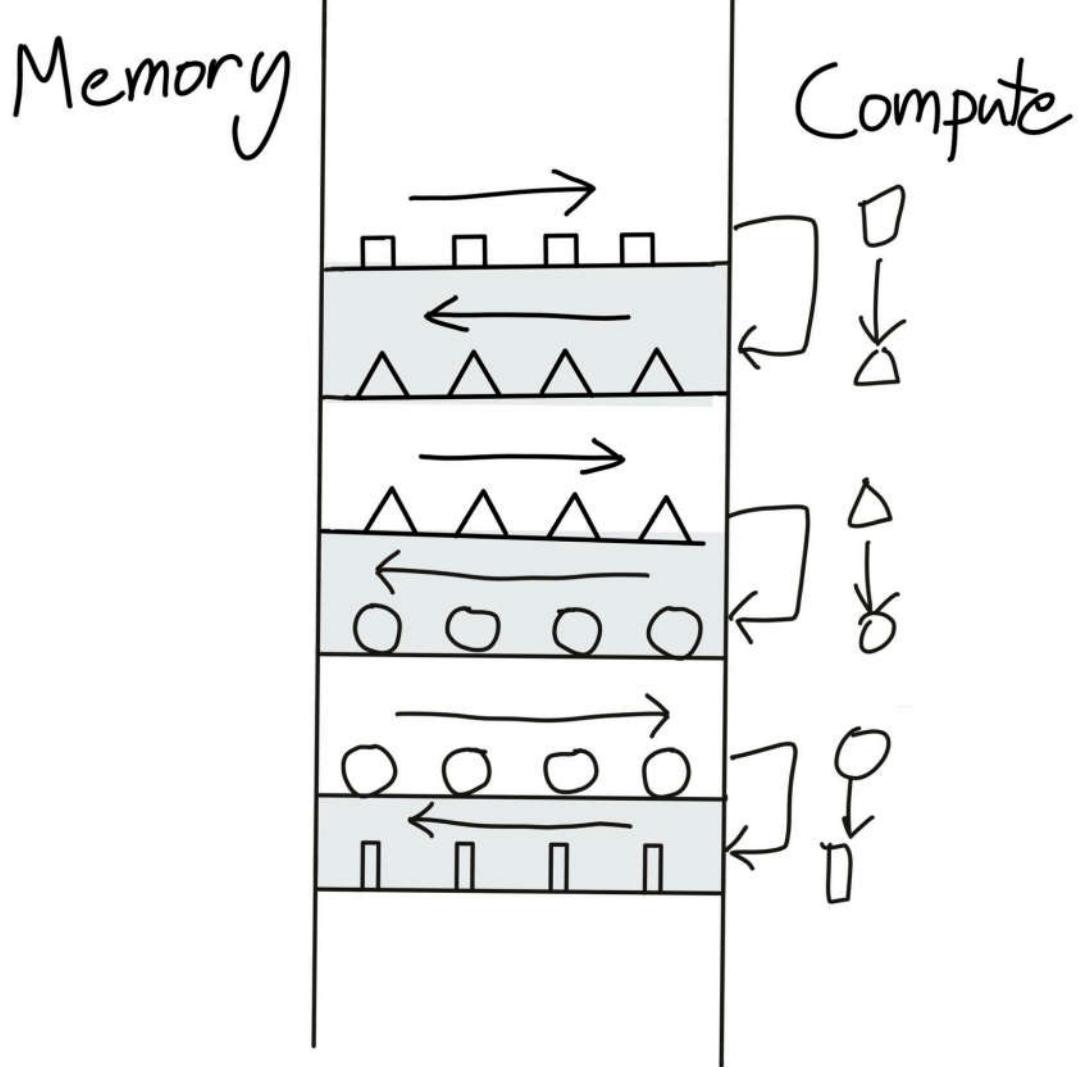


```

362
363     Each operation needs to read/compute/write:

```

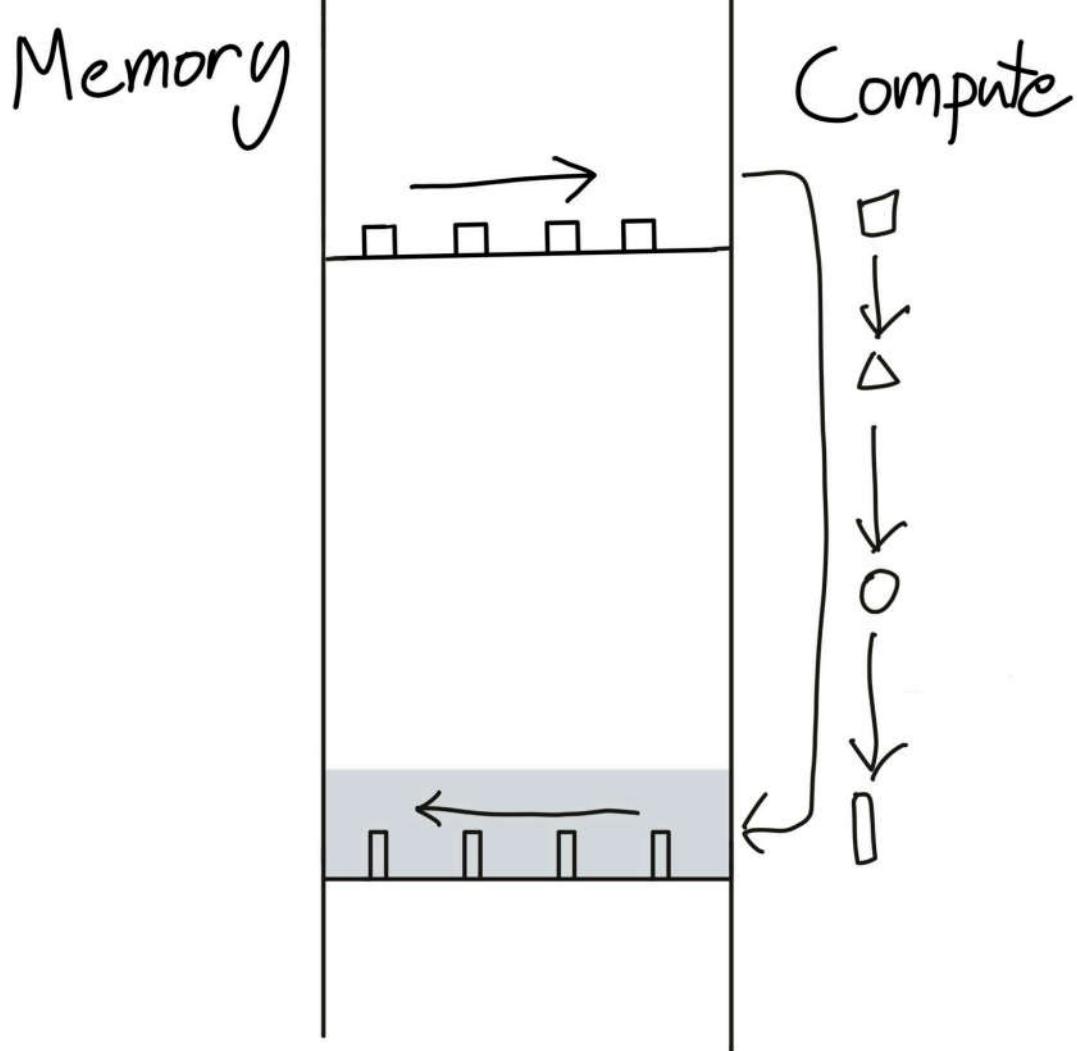
364



365

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

367



368

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

371

372 Let's consider two ways to compute GeLU:

```
373 x = torch.tensor([1.]) # @inspect x
```

374

375 1. The default PyTorch implementation (fused):

```
376 y1 = pytorch_gelu(x) # @inspect y1
```

377

378 2. We can also write our own by hand (not fused):

```
379 y2 = manual_gelu(x) # @inspect y2
```

380

```
381 # Check that the implementations match
```

```
382 assert torch.allclose(y1, y2)
```

383

```
384 # Check more systematically
```

```
385 check_equal(pytorch_gelu, manual_gelu)
```

386

387 Let's benchmark.

```
388 manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
```

```
389 pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
```

```
390 if manual_time is not None and pytorch_time is not None:
```

391 The fused version is significantly faster: 8.15 ms, 1.13 ms

```
392 else:
```

```
393     text("Could not compare times - benchmark results were None")
```

394

395 Let's look under the hood.

```
396 manual_gelu_profile = profile("manual_gelu", run_operation1(dim=16384, operation=manual_gelu))
```

397

manual_gelu

398

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::mul	15.19%	1.479ms	26.01%	2.53
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<f...>::operator()>(TensorList const&, TensorList const&, Tensor const&, Scalar const&) [at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<f...>::operator()>]		0.00%	0.000us	0.00%	
	aten::add	0.13%	12.473us	0.21%	20.08
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor<_add...>::operator()>(TensorList const&, TensorList const&, Tensor const&, Scalar const&) [at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor<_add...>::operator()>]		0.00%	0.000us	0.00%	
	aten::tanh	0.07%	6.961us	0.12%	11.58
void at::native::vectorized_elementwise_kernel<4, at::native::tanh_kernel_cud...>(TensorList const&, TensorList const&, Tensor const&, Scalar const&) [at::native::vectorized_elementwise_kernel<4, at::native::tanh_kernel_cud...>]		0.00%	0.000us	0.00%	
	cudaLaunchKernel	10.95%	1.066ms	10.95%	1.06
	cudaDeviceSynchronize	73.66%	7.171ms	73.66%	7.17

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

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

400 pytorch_gelu

401

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::gelu	71.12%	1.436ms	84.28%	1.70
void at::native::vectorized_elementwise_kernel<4, at::native::GeluCUDAKernelI...>(TensorList const&, TensorList const&, Tensor const&, Scalar const&) [at::native::vectorized_elementwise_kernel<4, at::native::GeluCUDAKernelI...>]		0.00%	0.000us	0.00%	
	cudaLaunchKernel	13.16%	265.687us	13.16%	265.68
	cudaDeviceSynchronize	15.72%	317.405us	15.72%	317.40

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

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

403

404 Look at Nsight profiler for MLP

405

406

407 def cuda_kernels():

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

409

410 Let's write the GeLU function in CUDA.

411 cuda_gelu = create_cuda_gelu() # @inspect cuda_gelu

412 x = manual_gelu # @inspect x

413

414 Check correctness of our implementation.

415 if cuda_gelu is not None:

416 check_equal(cuda_gelu, manual_gelu)

417

418 Benchmark our CUDA version.

419 pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time

420 manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time

421 if cuda_gelu is not None:

422 cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu)) # @inspect cuda_time

423 cuda_gelu_profile = profile("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu))

424

cuda_gelu

425

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self C
gelu_kernel(float*, float*, int)	0.00%	0.000us	0.00%	0.000us	0.000us	1.664ms	10
aten::empty_like	0.20%	6.209us	46.52%	1.428ms	1.428ms	0.000us	
aten::empty_strided	46.31%	1.422ms	46.31%	1.422ms	1.422ms	0.000us	
cudaLaunchKernel	8.93%	274.078us	8.93%	274.078us	274.078us	0.000us	
cudaDeviceSynchronize	44.56%	1.368ms	44.56%	1.368ms	683.944us	0.000us	

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

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

427

428 Elementwise operations are easy in CUDA (though you can still be smarter).

429 But most interesting operations (e.g., matmul, softmax, RMSNorm) require reading multiple values.

430 For that, you have to think about managing shared memory, etc.

431

432

433 `def create_cuda_gelu():`

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

435

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

437

438

439 Grid: collection of thread blocks: numBlocks = (2, 4), blockDim = (1, 8)

440 Thread block: collection of threads: blockIdx = (0, 1)

441 Thread: single unit of operation: threadIdx = (0, 3).

442

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

444

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

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

447

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

449

450 # CUDA code: has the full logic

451 `cuda_gelu_src = open("gelu.cu").read()`

```

452     #include <math.h>#include <torch/extension.h>#include <c10/cuda/CUDAException.h>global void gelu_kernel(float* in, float* out, in
// 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;
}
453
454 # C++ code: defines the gelu function
455 cpp_gelu_src = "torch::Tensor gelu(torch::Tensor x);"
456
457 Compile the CUDA code and bind it to a Python module.
458 ensure_directory_exists("var/cuda_gelu")
459 if not torch.cuda.is_available():
460     return None
461 module = load_inline(
462     cuda_sources=[cuda_gelu_src],
463     cpp_sources=[cpp_gelu_src],
464     functions=["gelu"],
465     extra_cflags=["-O2"],
466     verbose=True,
467     name="inline_gelu",
468     build_directory="var/cuda_gelu",
469 )
470
471 cuda_gelu = getattr(module, "gelu")
472 return cuda_gelu
473
474
475 def triton_kernels():
476     triton_introduction()
477     triton_gelu_main()
478
479
480 def triton_introduction():
481     Developed by OpenAI in 2021
482     https://openai.com/research/triton
483

```

```

484     Make GPU programming more accessible
485     • Write in Python
486     • Think about thread blocks rather than threads
487
488     What does Triton offer?
489
490             CUDA      Triton
491
492     • Memory coalescing (transfer from DRAM)      manual      automatic
493
494     • Shared memory management                  manual      automatic
495
496     • Scheduling within SMs                      manual      automatic
497
498     • Scheduling across SMs                     manual      manual
499
500
501
502     Compiler does more work, can actually outperform PyTorch implementations!
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521

```

484 Make GPU programming more accessible
485 • Write in Python
486 • Think about thread blocks rather than threads
487
488 What does Triton offer?
489
490 CUDA Triton
491
492 • Memory coalescing (transfer from DRAM) manual automatic
493
494 • Shared memory management manual automatic
495
496 • Scheduling within SMs manual automatic
497
498 • Scheduling across SMs manual manual
499
500
501
502 One big advantage of Triton is that you can step through the Python code.
503
504 Let's step through a Triton kernel.
505 x = torch.randn(8192, device=get_device())
506 y1 = triton_gelu(x)
507
508 print_ptx_main() # Look at the generated instructions
509
510 Check that it's correct.
511 check_equal(triton_gelu, manual_gelu)
512
513 Let's now benchmark it compared to the PyTorch and CUDA implementations.
514 Remember to set TRITON_INTERPRET=0 for good performance.
515 manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
516 pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
517 cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=create_cuda_gelu())) # @inspect cuda_time
518 triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspect triton_time
519
520 triton_gelu_profile = profile("triton_gelu", run_operation1(dim=16384, operation=triton_gelu))
521 **triton_gelu**

522

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA t
triton_gelu_kernel	0.00%	0.000us	0.00%	0.000us	0.000us	705.240us	100.00%	705.2
aten::empty_like	0.32%	5.629us	72.49%	1.267ms	1.267ms	0.000us	0.00%	0.0
aten::empty_strided	72.16%	1.261ms	72.16%	1.261ms	1.261ms	0.000us	0.00%	0.0
cuLaunchKernel	16.46%	287.632us	16.46%	287.632us	287.632us	0.000us	0.00%	0.0
cudaDeviceSynchronize	11.05%	193.144us	11.05%	193.144us	96.572us	0.000us	0.00%	0.0

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

523

524 Our Triton implementation (triton_gelu):

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

527

528 Triton operates on blocks, CUDA operates on threads.

529 Blocks allows Triton compiler to do other optimizations (e.g., thread coarsening).

530

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

532

533

```

534 def triton_gelu(x: torch.Tensor):
535     assert x.is_cuda
536     assert x.is_contiguous()
537
538     # Allocate output tensor
539     y = torch.empty_like(x)
540
541     # Determine grid (elements divided into blocks)
542     num_elements = x.numel()
543     block_size = 1024 # Number of threads
544     num_blocks = triton.cdiv(num_elements, block_size)
545
546     triton_gelu_kernel[(num_blocks,)](x, y, num_elements, BLOCK_SIZE=block_size)
547
548     return y
549
550
551 @triton.jit
552 def triton_gelu_kernel(x_ptr, y_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
553     # Input is at `x_ptr` and output is at `y_ptr`
554     # | Block 0 | Block 1 | ... | num_elements
555     #           BLOCK_SIZE
556
557     pid = tl.program_id(axis=0)
558     block_start = pid * BLOCK_SIZE
559
560     # Indices where this thread block should operate
561     offsets = block_start + tl.arange(0, BLOCK_SIZE)
562
563     # Handle boundary
564     mask = offsets < num_elements

```

```
565
566     # Read
567     x = tl.load(x_ptr + offsets, mask=mask)
568
569     # Approx gelu is 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
570     # Compute (tl.tanh doesn't exist, use tanh(a) = (exp(2a) - 1) / (exp(2a) + 1)
571     a = 0.79788456 * (x + 0.044715 * x * x * x)
572     exp = tl.exp(2 * a)
573     tanh = (exp - 1) / (exp + 1)
574     y = 0.5 * x * (1 + tanh)
575
576     # Store
577     tl.store(y_ptr + offsets, y, mask=mask)
578
579
580 def print_ptx_main():
581     PTX (parallel thread execution) is like an assembly language for GPUs.
582
583     We can see the PTX code generated by Triton.
584     https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
585
586     ptx = print_ptx("triton_gelu", triton_gelu_kernel)
```

```

587     //// 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.t32      %t65, %t1, 0tBF8000000;
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_unit.b8 0
}
.section .debug_info
{
    .b32 76         // Length of Unit.b8 2          // DWARF version number.b8 0.b32
}
.section .debug_macinfo { }

588
589 Observations:
590 • ld.global.* and st.global.* reads and writes from global memory
591 • %ctaid.x is block index, %tid.x is thread index
592 • %f* are floating point registers, %r* are integer registers
593 • One thread processes 8 elements at the same time (thread coarsening)
594
595
596 def print_ptx(name: str, kernel):
597     if os.environ.get("TRITON_INTERPRET") == "1":
598         text("PTX is not generated when in interpret mode.")
599         return
600
601     """Print out the PTX code generated by Triton for the given `kernel`."""
602     ptx_path = f"var/{name}-ptx.txt"
603
604     Let's go poke around at the PTX code.
605     https://github.com/stanford-cs336/spring2025-lectures/blob/main/var/triton\_softmax-ptx.txt
606
607     with open(ptx_path, "w") as f:
608         return list(kernel.cache[0].values())[0].asm["ptx"]
609

```

```
610
611
612 def pytorch_compilation():
613     So far, we have seen three ways to write GeLU:
614     • Use the default PyTorch function
615     • Write it in Python manual_gelu
616     • Write it in CUDA create_cuda_gelu
617     • Write it in Triton triton_gelu
618
619     • Write it in Python and compile it into Triton
620     compiled_gelu = torch.compile(manual_gelu)
621
622     Check correctness of our implementation.
623     check_equal(compiled_gelu, manual_gelu)
624
625     if not torch.cuda.is_available():
626         return
627
628     Let's benchmark and profile it!
629     manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspect manual_time
630     pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @inspect pytorch_time
631     cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=create_cuda_gelu())) # @inspect cuda_time
632     triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspect triton_time
633     compiled_time = benchmark("compiled_gelu", run_operation1(dim=16384, operation=compiled_gelu)) # @inspect compiled_time
634
635     Let's look under the hood
636     compiled_gelu_profile = profile("compiled_gelu", run_operation1(dim=16384, operation=compiled_gelu))
637 compiled_gelu
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %
<hr/>							
Torch-Compiled Region: 0/1	77.92%	1.934ms	91.35%	2.268ms	2.268ms	0.000us	0.00%
triton_poi_fused_add_mul_tanh_0	1.84%	45.795us	13.43%	333.455us	333.455us	707.261us	100.00%
triton_poi_fused_add_mul_tanh_0	0.00%	0.000us	0.00%	0.000us	0.000us	707.261us	100.00%
TorchDynamo Cache Lookup	0.62%	15.371us	0.62%	15.371us	15.371us	0.000us	0.00%
cuLaunchKernel	11.59%	287.660us	11.59%	287.660us	287.660us	0.000us	0.00%
cudaDeviceSynchronize	8.03%	199.299us	8.03%	199.299us	99.649us	0.000us	0.00%

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

```

639
640
641 def triton_softmax_main():
642     So far, we've looked at elementwise operations in Triton (e.g., GeLU).
643     Now let us look at operations that aggregate over multiple values.
644
645     We will roughly follow the Triton fused softmax tutorial: https://triton-lang.org/main/getting-started/tutorials/02-fused-softmax.html
646
647     Recall the softmax operation is used in attention and generating probabilities.
648     Normalize each row of a matrix:
649     [A1 A2 A3]  =>  [A1/A A2/A A3/A]
650     [B1 B2 B3]  =>  [B1/B B2/B B3/B]
651
652     Let's first start with the naive implementation and keep track of reads/writes.
653     x = torch.tensor([
654         [5., 5, 5],
655         [0, 0, 100],
656     ], device=get_device())
657     y1 = manual_softmax(x) # @inspect y1
658
659     if not torch.cuda.is_available():
660         return
661
662     Now let us write the Triton kernel.
663     y2 = triton_softmax(x)
664     assert torch.allclose(y1, y2)
665
666     Check our implementations are correct.
667     check_equal2(pytorch_softmax, manual_softmax)
668     check_equal2(pytorch_softmax, triton_softmax)
669
670     compiled_softmax = torch.compile(manual_softmax)
671
672     Now let's benchmark everything.
673     manual_time = benchmark("manual_softmax", run_operation1(dim=16384, operation=manual_softmax)) # @inspect manual_time
674     compiled_time = benchmark("compiled_softmax", run_operation1(dim=16384, operation=compiled_softmax)) # @inspect compiled_time
675     pytorch_time = benchmark("pytorch_softmax", run_operation1(dim=16384, operation=pytorch_softmax)) # @inspect pytorch_time
676     triton_time = benchmark("triton_softmax", run_operation1(dim=16384, operation=triton_softmax)) # @inspect triton_time

```

677

678 Look under the hood using the profiler.

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

681

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::div	0.28%	8.466us	0.43%	13.24
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocas...>		0.00%	0.000us		0.00%
	aten::sub	0.44%	13.364us	0.70%	21.60
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocas...>		0.00%	0.000us		0.00%
	aten::exp	0.25%	7.610us	0.41%	12.59
void at::native::vectorized_elementwise_kernel<4, at::native::exp_kernel_cuda...>		0.00%	0.000us		0.00%
	aten::max	10.71%	328.962us	19.95%	612.83
void at::native::reduce_kernel<512, 1, at::native::ReduceOp<float, at::native...>		0.00%	0.000us		0.00%
	aten::sum	0.38%	11.798us	0.67%	20.68
void at::native::reduce_kernel<512, 1, at::native::ReduceOp<float, at::native...>		0.00%	0.000us		0.00%

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

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

683

	Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self
	Torch-Compiled Region: 1/0	56.77%	769.985us	78.99%	1.071ms	1.071ms	0.000us	
	triton_red_fused_div_exp_max_sub_sum_0	3.20%	43.382us	22.22%	301.366us	301.366us	730.770us	
	triton_red_fused_div_exp_max_sub_sum_0	0.00%	0.000us	0.00%	0.000us	0.000us	730.770us	
	TorchDynamo Cache Lookup	0.53%	7.239us	0.53%	7.239us	7.239us	0.000us	
	cuLaunchKernel	19.02%	257.984us	19.02%	257.984us	257.984us	0.000us	
	cudaDeviceSynchronize	20.48%	277.800us	20.48%	277.800us	138.900us	0.000us	

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

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

686

pytorch_softmax

687

	Name	Self CPU %	Self CPU	CPU total %	CPU to
	aten::softmax	0.47%	5.061us	28.87%	312.56
	aten::_softmax	13.44%	145.534us	28.40%	307.50
	void at::native::(anonymous namespace)::cunn_SoftMaxForward<4, float, float, ...>	0.00%	0.000us	0.00%	
	cudaLaunchKernel	14.96%	161.969us	14.96%	161.96
	cudaDeviceSynchronize	71.13%	770.228us	71.13%	770.22

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

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

triton_softmax

690

	Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA t
	triton_softmax_kernel	0.00%	0.000us	0.00%	0.000us	0.000us	705.462us	100.00%	705.4
	aten::empty_like	0.23%	4.238us	77.57%	1.409ms	1.409ms	0.000us	0.00%	0.0
	aten::empty_strided	77.34%	1.405ms	77.34%	1.405ms	1.405ms	0.000us	0.00%	0.0
	cuLaunchKernel	8.02%	145.738us	8.02%	145.738us	145.738us	0.000us	0.00%	0.0
	cudaDeviceSynchronize	14.41%	261.640us	14.41%	261.640us	130.820us	0.000us	0.00%	0.0

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

691

692 Let's end by looking at the PTX code.

693 ptx = print_ptx("triton_softmax", triton_softmax_kernel)

```

694     //// 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_unit.b8 1

}
.section .debug_info
{
.b32 173                // Length of Unit.b8 2           // DWARF version number.b8 0.b32

}
.section .debug_macinfo { }

695
696
697 def manual_softmax(x: torch.Tensor):
698     # M: number of rows, N: number of columns
699     M, N = x.shape
700
701     # Compute the max of each row (MN reads, M writes)
702     x_max = x.max(dim=1)[0]
703
704     # Subtract off the max (MN + M reads, MN writes)
705     x = x - x_max[:, None]
706
707     # Exponentiate (MN reads, MN writes)
708     numerator = torch.exp(x)
709
710     # Compute normalization constant (MN reads, M writes)
711     denominator = numerator.sum(dim=1)
712
713     # Normalize (MN reads, MN writes)
714     y = numerator / denominator[:, None]
715
716     # Total: 5MN + M reads, 3MN + 2M writes
717     # In principle, should have MN reads, MN writes (speedup of 4x!)
718     return y
719
720
721 def triton_softmax(x: torch.Tensor):
722     # Allocate output tensor
723     y = torch.empty_like(x)
724
725     # Determine grid

```

```

726     M, N = x.shape                      # Number of rows x number of columns
727     block_size = triton.next_power_of_2(N) # Each block contains all the columns
728     num_blocks = M                        # Each block is a row
729
730     # Launch kernel
731     triton_softmax_kernel[(M,)](
732         x_ptr=x, y_ptr=y,
733         x_row_stride=x.stride(0), y_row_stride=y.stride(0),
734         num_cols=N, BLOCK_SIZE=block_size
735     )
736
737     return y
738
739
740 @triton.jit
741 def triton_softmax_kernel(x_ptr, y_ptr, x_row_stride, y_row_stride, num_cols, BLOCK_SIZE: tl.constexpr):
742     assert num_cols <= BLOCK_SIZE
743
744     # Process each row independently
745     row_idx = tl.program_id(0)
746     col_offsets = tl.arange(0, BLOCK_SIZE)
747
748     # Read from global memory
749     x_start_ptr = x_ptr + row_idx * x_row_stride
750     x_ptrs = x_start_ptr + col_offsets
751     x_row = tl.load(x_ptrs, mask=col_offsets < num_cols, other=float("-inf"))
752
753     # Compute
754     x_row = x_row - tl.max(x_row, axis=0)
755     numerator = tl.exp(x_row)
756     denominator = tl.sum(numerator, axis=0)
757     y_row = numerator / denominator
758
759     # Write back to global memory
760     y_start_ptr = y_ptr + row_idx * y_row_stride
761     y_ptrs = y_start_ptr + col_offsets
762     tl.store(y_ptrs, y_row, mask=col_offsets < num_cols)
763
764
765 def triton_matmul_main():
766     text("Matrix multiplication is perhaps the most optimized algorithm ever.")
767
768     text("If you write matrix multiplication in CUDA, there's all sorts of crazy things you have to do.")
769     link("https://github.com/openai/blocksparse/blob/master/src/matmul_op_gpu.cu")
770
771     text("It's much easier in Triton.")
772     link("https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html")
773
774     text("      k          j          ", verbatim=True)
775     text(" [ A1 A2 A3 ]   [ B1 B2 B3 ]   [ C1 C2 C3 ]", verbatim=True)
776     text("i [ A4 A5 A6 ] * k [ B4 B5 B6 ] = [ C4 C5 C6 ]", verbatim=True)
777     text(" [ A7 A8 A9 ]   [ B7 B8 B9 ]   [ C7 C8 C9 ]", verbatim=True)
778
779     text("Naively: need MKN reads, MN writes")
780
781     text("Computing C4 and C5 both need A4, A5, A6.")
782     text("Can we read A4, A5, A6 from DRAM once to compute both?")
783     text("Answer: yes, using shared memory!")
784
785     text("## Tiling (leveraging shared memory)")
786
787     text("Recall that shared memory is:")

```

```

788     text("- fast (10x faster) and small(~100KB)")
789     text("- shared between all the threads in a block.")
790     image("https://miro.medium.com/v2/resize:fit:2000/format:webp/1*6xoBK5kL2dZpivFe1-zgw.jpeg")
791
792     text("Trivial: for small matrices, load all of A and B into shared memory, then could compute C.")
793     text("Now we get MK + KN reads, MN writes")
794
795     text("But what if we have big matrices...")
796
797     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)
798     text("Key idea: divide the matrix into blocks.")
799     text("For each block of A and block of B:")
800     text("- load into shared memory,")
801     text("- do mini-matrix multiplication,")
802     text("- write the partial sum.")
803
804     text("Animation of tiled matrix multiplication "), link("https://youtu.be/aMvCEEBIBto")
805
806     text("## Leveraging L2 cache")
807
808     text("Two ways of computing 9 elements of a matrix:")
809     image("https://triton-lang.org/main/_images/grouped_vs_row_major_ordering.png", width=0.5)
810     text("1. Loads 9 + 81 = 90 blocks")
811     text("1. Loads 27 + 27 = 54 blocks")
812
813     text("Process the blocks in an order that minimizes the reads.")
814
815     text("Why write your own kernel for matrix multiplication (e.g., A @ B)?")
816     text("Answer: fusion with another operation (e.g., gelu(A @ B))")
817
818     if not torch.cuda.is_available():
819         return
820     text("Let's try it!")
821     benchmark("pytorch_matmul", run_operation2(dim=16384, operation=torch.matmul))
822     benchmark("triton_matmul", run_operation2(dim=16384, operation=triton_matmul))
823
824     # Not working for some reason
825     #print_ptx("triton_matmul", triton_matmul_kernel)
826
827
828 def further_reading():
829     Horace He's blog post [Article]
830
831     CUDA MODE Lecture 1: how to profile CUDA kernels in PyTorch [Video]
832     CUDA MODE Lecture 2: Chapters 1-3 of PPMP book [Video]
833     CUDA MODE Lecture 3: Getting started with CUDA for Python Programmers [Video]
834     CUDA MODE Lecture 4: Compute and memory basics [Video]
835     CUDA MODE Lecture 8: CUDA performance checklist [Video]
836
837     HetSys Course: Lecture 1: Programming heterogenous computing systems with GPUs [Video]
838     HetSys Course: Lecture 2: SIMD processing and GPUs [Video]
839     HetSys Course: Lecture 3: GPU Software Hierarchy [Video]
840     HetSys Course: Lecture 4: GPU Memory Hierarchy [Video]
841     HetSys Course: Lecture 5: GPU performance considerations [Video]
842
843     [A100 GPU with NVIDIA Ampere Architecture]
844     [NVIDIA Deep Learning Performance Guide]
845     [GPU Puzzles]
846     [Triton Paper]
847     [PyTorch 2.0 Acceleration]
848
849 #####

```

```
850
851 def print_gpu_specs():
852     num_devices = torch.cuda.device_count() # @inspect num_devices
853     8 devices
854     for i in range(num_devices):
855         properties = torch.cuda.get_device_properties(i) # @inspect properties
856         7: _CudaDeviceProperties(name='NVIDIA H100 80GB HBM3', major=9, minor=0, total_memory=81090MB,
857         multi_processor_count=132, uid=62f395b0-f63d-2a9d-d202-53f798ada4f4, L2_cache_size=50MB)
858
859 def pytorch_softmax(x: torch.Tensor):
860     return torch.nn.functional.softmax(x, dim=-1)
861
862
863 def pytorch_gelu(x: torch.Tensor):
864     # Use the tanh approximation to match our implementation
865     return torch.nn.functional.gelu(x, approximate="tanh")
866
867
868 def manual_gelu(x: torch.Tensor):
869     return 0.5 * x * (1 + torch.tanh(0.79788456 * (x + 0.044715 * x * x * x)))
870
871
872
873
874 if __name__ == "__main__":
875     main()
```