Thanks Ajinkya Tejankar, Mohsin Iqbal, Sujit Ahirrao, and Krishna Gupta for the notes!

After the event, the note will be used to:
1. Create a summary and key takeaways for each talk.
2. Curate a list of resources for people to learn about GPU optimization, recommended by the speakers and other attendees.
    a. A roadmap would be nice.

The materials will be posted on our shared GitHub repo when done.

Crash course to GPU optimization shared notes [Slides]

1. PyTorch
    a. Needs to support different dtypes, layouts, and devices so the kernels are general
    b. Eager execution allows easy debugging but this trades off performance
    c. Models getting converged to eager may not be the best performance
2. Pointwise ops
    a. Every element assigned to a single thread that run in parallel
3. Memory hierarchy
    a. Load data in shared memory and apply relu
    b. Done differently in Triton
4. Eager execution can lead to unnecessary memory accesses if kernels are repeated
5. GPU mem bandwidth is the bottleneck not FLOPs
6. Operations / no of bytes accessed = arithmetic intensity
7. Repeated calls to a kernel can be fused together - torch.compile (generates Triton) - PyTorch 2 paper for more information
8. FP32 to FP16 or BF16 improves the perf significantly
9. torch.set_float32_matmul_precision('high') => use tensor cores => talk by Nvidia about tensor cores on CUDA MODE
10. Most of the time may be spent on figuring out which GPU kernel to use because 1.a above
11. CUDA kernels are asyc so queue them up -> CUDA graphs "reduce-overhead" in torch.compile
12. Quantization helps compute bound but also mem bound kernels as it reduces the number of bytes accessed in the arithmetic intensity calculation
13. GPT fast - weight only quantization
14. Int8 is ambiguous - quantize optimizers? Gradients? Not applied over all the model only the linear layers. W8A16 -> Int 8 weights.
15. Bit packing: Pack 2 int4s into a single int8
16. Compute bound problems: become better at math
17. Why compiler couldn't have figured out FlashAttention? Q by a reviewer Compilers are good at fusing kernels but not math of the operations

18. Online softmax paper explains the FlashAttention better
19. Learn the basics of CUDA - Programming Massively Parallel Processors: A Hands-on Approach - helps with compute bound kernels
20. `load_inline` function in `cpp_extension` in pytorch
21. Nvidia provides a profiler: `ncu` - good supplement for reading the above book
22. Write kernels!! Good content on the cuda-mode and join for writing custom kernels
23. Karpathy - building in raw cuda
24. Reach out to Mark for shipping your hand written cuda kernels (he'll help with release)
25. Learning through mentorship is great since public docs are not great at the moment
26. Quantization is not possible through torch.compile
27. How to make PyTorch models faster: Fuse more, use tensor cores, reduce overhead, quantize, use a custom kernel (all in order)
28. How's execute torch different from torch.compile? Focused on more constrained devices. However, dynamo (a part of the compile subsystem) is shared.
29. How does PyTorch treat GPUs other than Nvidia's? Triton provides backends that work on Intel, AMD GPUs so PyTorch just generates Triton. Hierarchical IR and Code gen.
30. What do you think about 1 bit quantization? Eval does not scale. Bit packing can help.
31. Common pitfalls of running GPUs?
    a. Eager - Profile first to figure out the real bottlenecks
    b. Compile - Enable first 3 things on 27 point

Relevant resources

- [Programming Massively Parallel Processors: A Hands-on Approach]()
- CUDA Mode: [discord.gg/cudamode]()
- Native PyTorch library for quantization and sparsity: [https://github.com/pytorch/ao]()
- Learn Triton: [https://github.com/cuda-mode/triton-index/]()

LLM Serving optimization

1. Focusing on server-based systems not edge-end user latencies are important
2. Multi-functional accurate models are large - deployment and optimization is a challenge
3. Many models, very big models, new operators (optimization becomes a moving target)
4. Goal: SoTA performance for LLMs for production deployments
5. Fast forward pass is very important. Also, important intelligent batching
6. Other techniques like kv cache optimization for improved GPU workload
7. Quantization
    a. As long as you can preserve accuracy, lower bit-width precisions are great
        i. Lesser memory, higher throughput comms between GPUs, faster computation (all-round win)
    b. Post-training quantization is the most common
    c. TensorRT model optimizer offers a bunch of techniques
        i. PTQ (post-training quantization) and QAT (quantization-aware training)

        ii. SmoothQuant and INT4 AWQ don't lead to too much drop in acc (MMLU)
8. LLM request has two phases
    a. Prefill: process the prompt, generate the first token, and init the kv cache. Called only once for a request. Lots of parallel operations across tokens.
    b. Generate: starts from prior state (kv cache) and generates the next token, updating the kv cache. Called in a loop for each request. Lot of memory bound operations.
    c. Attention is complex - features like GQA and Speculative Decoding increase math:data movement ratio (arithmetic intensity)
    d. TRT-LLMs fastest implementations use hand tuned custom cuda kernels
9. Traditional Request Scheduling (static batching)
    a. Accumulate, batch, forward
    b. Request as an atomic operation is great for fixed length inputs however for tasks like completion where outputs differ in length this is not great. (image vs. chat)
    c. Largest completion in a batch can stall the smallest completion. Padding also wastes computation.
10. LLM Request Properties
    a. Multiple forward passes and the number is unknown a priori
    b. Online setting, request arrival time is priori
    c. In flight batching
        i. On EOS, Max tokens reached, stop phrase -> send response and evict
        ii. Process new work - next iteration of LLM
            1. Prompt phase goes to prefill
            2. Prefill goes to generate
            3. Generate keeps generating
        iii. Transformer ops
            1. Token parallel - Matmul, LayerNorm
            2. Sequence parallel - MHA
            3. Tokens across above two types are concatenated in in-flight batching to improve memory bound (makes it more compute intensive)
    d. Paged KV Cache
        i. Contiguous KV Cache leads to wasted allocation of memory since all KV cache memory is contiguous
        ii. Instead think of memory as a linked list of pages - reduces memory unused memory - lazy memory allocation - increases complexity of attention kernel
        iii. Allows sharing of KV cache between requests! E.g. system prompt kv cache blocks are part of the linked list of different requests!
    e. Speculative Decoding
        i. Instead of generating a single token as in regular autoregressive generation, generate many tokens
        ii. Evaluate if draft tokens are valid in the same time as a single token is generated

            iii. Speculates that speculative decoding will be used everywhere ;)
            iv. Turns latency problem into throughput problem where GPUs are great
    f. Time to first token vs time between token. Which is important? Time between since time to first is easily optimized.
    g. Online vs batch inference. Which is common? Online is important, but the idea is to turn online into batch inference.
    h. Any specific techniques for streaming mode? Not much. Stream out tokens as they are generated. Since everything is async anyway.
    i. Quantization sounds too good to be true. Any caveats? PTQ is model dependent.
    j. Good intro paper for changing workload? Orca paper. Link in the discord.
    k. Many LLM inference services. Which one to use? Each is optimized for a specific use cases so explore.
    l. What are the questions ppl should be asking when evaluating inference services? Clarity of Quality of Service (latency, throughput, acc) for your use case
    m. Now way to avoid multi-gpu since models keep getting bigger. For many cases, single GPU use case is just fine.

Relevant resources

- Decoding Speculative Decoding
- Accelerating Large Language Model Decoding with Speculative Sampling
- Efficient Memory Management for Large Language Model Serving with PagedAttention

Block-based optimization with Triton [Slides]

1. CUDA - all sorts of things can be done on GPUs but since it allows anything to be done it creates problems and hampers productivity.
    a. First few months of support are okay
    b. Supporting different gpus becomes problem
    c. Opaque to researchers - cannot read CUDA code - reading tensor core code requires proficiency - becomes a black box - slows down research
    d. Addressed with Graph Compilers - better for research
        i. Walking a tree, linked lists in PyTorch are very slow
        ii. Control flow becomes complicated with graph operators
        iii. Code gen from graph compilers is a very difficult problem - this gives rise FlashAttention like custom CUDA kernels
        iv. Simplicity at the cost of flexibility
2. Triton - more low level than graph compilers but much easier to work with than CUDA
    a. Can write algorithms out of scope of graph compilers - trees, linked lists, radix sort
    b. Code still remains readable/modifiable by researchers
    c. Performance is portable across different vendors
    d. Less expressive than CUDA not as fast
3. Triton Machine Model

    a. DRAM, L1 and L2 cache, Cores, Memory Controllers - Von Neumann Basic
4. Programming Model
    a. Tensors are defined in SRAM and modified using torch like operators
    b. Embedded in Python and Just-in-Time compiled
    c. Tensor of pointers!
    d. Powers of 2 - shapes of tensors!?
5. Vector addition
    a. Each program gets a different slice to the input with tl.program_id
6. Softmax
    a. Entirely fused kernels in less than 10 lines
    b. Load the data only once unlike PyTorch eager mode
7. Why blocked program representation?
    a. Peephole optimization
    b. SRAM allocation
    c. Automatic vectorization - Need to issue big enough loads to keep the memory bandwidth busy
    d. Compiler allocates shared mem in addition to registers
    e. Lot of value in researchers doing kernel developement!
    f. Technical debt manageable
8. Challenges of building kernels at OpenAI scale? Reliability vs agility of the code base
9. Tricks for single GPU? Consumer GPUs have restriction on tensor cores. Go out of your way to use 16bit tensor cores. Not a priority of OpenAI, but TinyGrad focuses on it.
10. Model performance can change after optimizations? Kernel output shouldn't change with reference non-optimized implementation. Power of 2 inputs.
11. Surprising kernels built on top of Triton? Sorting kernel. Hypercubes.
12. Why block based? Grew out of dissertation.

Relevant resources

- https://openai.com/index/triton/
-

Scaling data workloads on GPUs

1. Transactional databases - not gpu friendly - row oriented - CSV
2. Analytics datasets - gpu friendly - column oriented - Parquet, Apache Arrow. Apache Arrow is everywhere today. It makes it easy to move data across multiple data platforms.
3. Nvidia Rapids contains many libraries for gpu processing: cuPy, cuDF, cuML, cuGraph
4. Benchmark showing prformance boost moving from CPU to GPU, can be up to 100x times faster. The speed up is more with larger workloads.
5. Data processing on CPUs eventually hits a wall.
6. GPUs are fast for data processing because many data processing jobs are naturally parallelizable and GPUs have many cores.

7. What to do depending on where your job bottlenecks: memory bound, latency bound, or compute bound. Figure out where the bottleneck is by using profiling tools.

Relevant resources

Overall