THE CYPRUS INSTITUTE
RESEARCH•TECHNOLOGY•INNOVATION

EURO CC

REINVENTING HPC
ISC High Performance

Optimising Llama2

Enhancing Efficiency and Scalability of Large Language Models with PyTorch

Chris Stylianou

Research Engineer

CaSToRC

eurocc.cyi.ac.cy

A Bit About Myself

- Research Engineer at CaSToRC
- Collaborations Task Leader for EuroCC2
- Area of expertise in High Performance Computing (HPC)
- PhD Candidate, EPCC
- Contact & Info:
  - Email: c.stylianou@cyi.ac.cy
  - Website: cstyl.github.io

- **Generative AI** use cases have **exploded** in popularity recently!
- **Text generation** one particularly popular area.
  - ChatGPT, Llama, vLLM etc.
- PyTorch is a popular **open-source** ML/AI library widely used in the area of AI
  - **Ease of Use** and Flexibility
  - Strong community and industry support
  - **Integration** with Python ecosystem
  - Accelerated computing (**GPU support**)
  - Educational resources
- How fast we can run transformer **inference** with only pure, native PyTorch?

Logistics

- Training GitHub Repo to be available at:
  - https://github.com/CaSToRC-CyI/isc2024-tutorial/tree/main/gpt-fast
  - Follow `README.md` for every part in the slides.
- Complete tutorial:
  - https://github.com/pytorch-labs/gpt-fast
- To download Llama2 7B parameters model, go to
  - https://huggingface.co/meta-llama/Llama-2-7b
    - Stored in HF Transformer format
  - Shared directory with processed weights available for the training!
- Hardware needed: GPUs with BF16 & INT8 precision
  - E.g., A100 NVIDIA GPUs 40GB

Environment Setup

```
# Connect to GWDG
$ ssh -i /path/to/ssh-key <username>@glogin.hlrn.de
# Shared Project Path
$ export SHARED_PATH=/mnt/lustre-emmy-ssd/projects/isc2024_accel_genai_pytorch
# Make a local copy of the code
$ git clone https://github.com/CaSToRC-CyI/isc2024-tutorial.git
# Activate environment
$ module load python/3.9.16
$ source $SHARED_PATH/torch/bin/activate
# Code for the session
$ cd isc2024-tutorial/gpt-fast
```

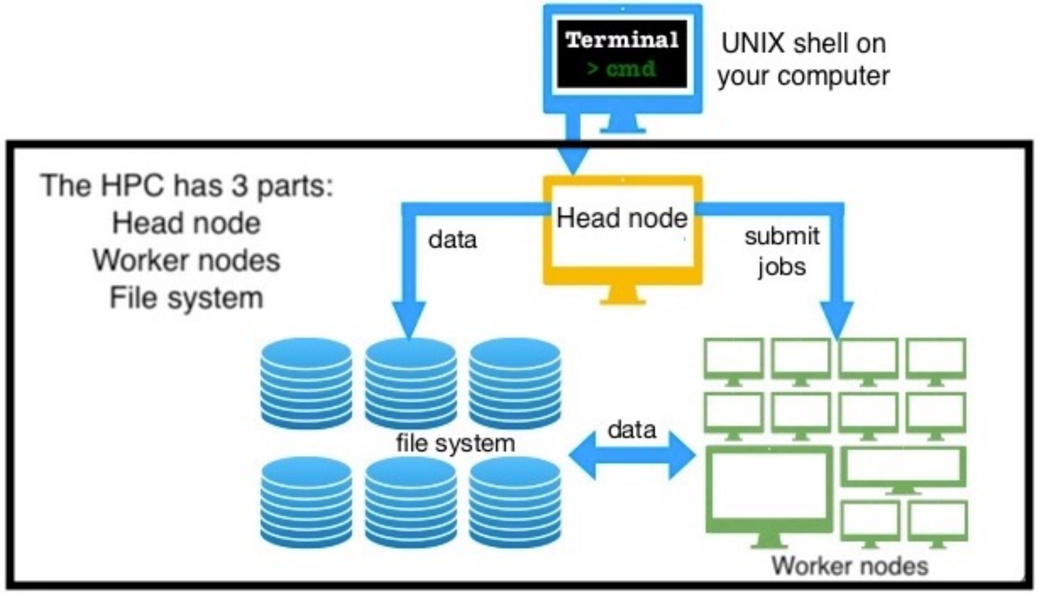


Figure 1: High-level overview of a Supercomputer [1].

A Note on the Scheduler Settings

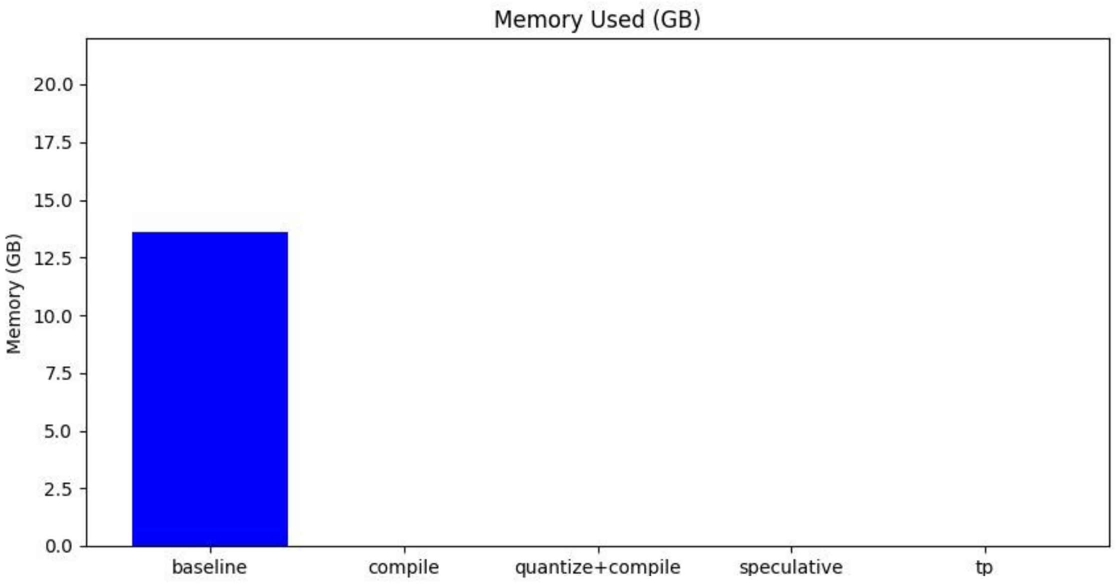
```
#!/bin/bash -l

#SBATCH --job-name=gpt-fast-baseline
#SBATCH --nodes=1                   # Request 1 Node
#SBATCH --ntasks-per-node=1         # Request 1 Process
#SBATCH -G A100:1                   # Request 1 GPU
#SBATCH --cpus-per-task=16          # Request 12 threads (not used, but good for locality)
#SBATCH --partition=grete           # Request on Cluster with A100 NVIDIA GPUs
#SBATCH --time=01:00:00             # Request access for 1 hour
#SBATCH --exclusive                 # Request exclusive access on whole node
#SBATCH --output=%x.out
# Setup paths and Environment
SHARED_PATH=/mnt/lustre-emmy-ssd/projects/isc2024_accel_genai_pytorch
EXEC_PATH=$SHARED_PATH/isc2024-tutorial/gpt-fast   # Path to Code
CHECKPOINTS_PATH=$SHARED_PATH/gpt-fast-checkpoints # Path to Weights
module load python/3.9.16
source $SHARED_PATH/torch/bin/activate

export DEVICE=cuda
export MODEL_REPO=meta-llama/Llama-2-7b
python $EXEC_PATH/generate.py --checkpoint_path $CHECKPOINTS_PATH/$MODEL_REPO/model.pth --prompt “Hello, my name is”
```

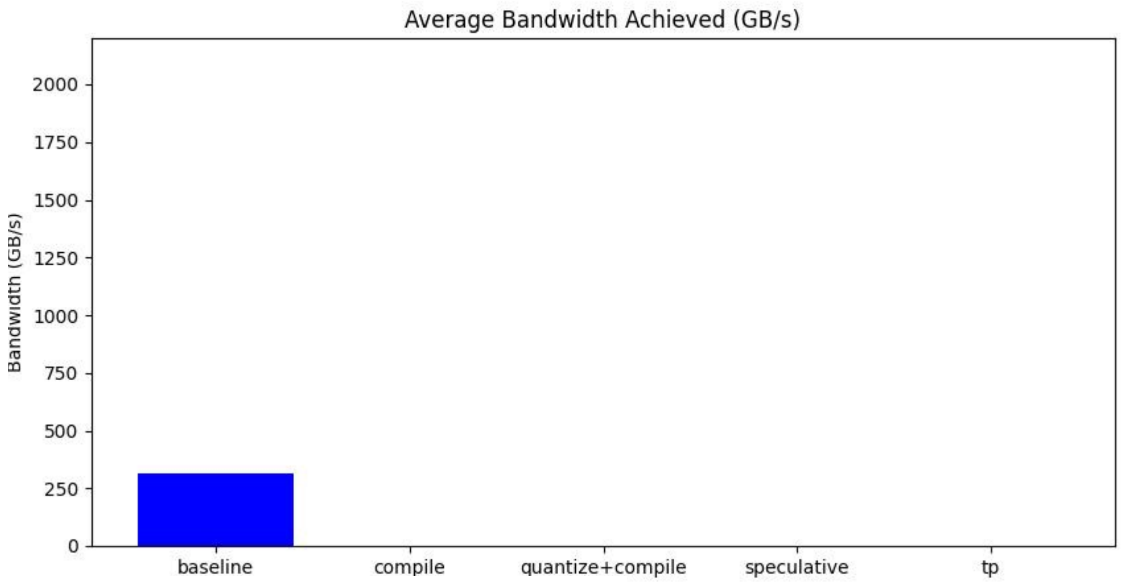
- **Llama2** (Meta) model
  - **7B** Parameters (~13GB)
  - **BF16** Precision used for weights
- Inference
  - Prompt: “`Hello, my name is`”
  - Generate **5 samples**, up to **200 tokens**.
  - **Batch Size = 1**
- Resources configuration:
  - # of GPUs (Nodes): 1 (1)
  - CUDA Version: 11.8.0
  - PyTorch Version: 2.3

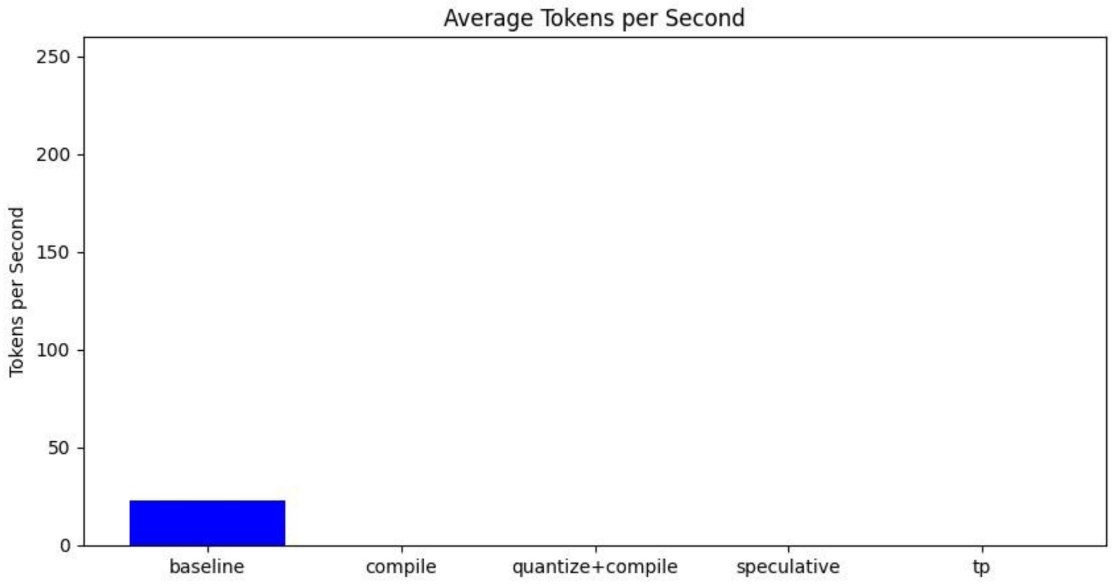
“Hello, my name is [Name], and I am a [insert activity or hobby here] enthusiast. I have been involved in this activity for [insert number of years] years, and I can tell you that it has brought me a great deal of joy and fulfillment.”

Original Model:
https://github.com/meta-llama/llama/tree/main

Running Baseline Version (23.17 tokens/s)







The Problem with Baseline Model

- Modern GPUs are **extremely efficient**, with a lot compute power.
- Deep learning computations run **entirely on GPU**
  - CPU acts as the **orchestrator**, telling the GPU what to do next.
- Complex algorithms often require **multiple GPU operations**
  → **Multiple kernel launches**
- Each kernel launch is associated with **some overhead**.
- When **kernels** sent **individually** and **complete quickly**:
  - CPU tries to keep up with the GPU (**CPU-bound**)
  - **Combined overhead** becomes noticeable, drop in overall performance!
- Solutions:
  - Rewrite in C++, raw CUDA?
  - Send more work to GPU at once?

A Compiler for PyTorch Models

- **Purpose of torch.compile()**: [Requires PyTorch >= 2.0]
  - **Arbitrary Python functions** can be optimised by passing the callable to `torch.compile()`
  - **JIT** Compilation, via minimal code changes.
- **Usage Example**:

```python
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b
opt_foo1 = torch.compile(foo)
print(opt_foo1(torch.randn(10, 10), torch.randn(10, 10)))
```

- **Key Considerations**:
  - Can also be used for `torch.nn.Module`
    - Compiles the model into optimised kernels during execution!
    - Useful if optimised model is used several times (Only have to pay once!)

A Compiler for PyTorch Models – LLM Case

- Speedup achieved via reducing **Python overhead** and **GPU read/writes**.
- `torch.compile` optimizes computational graphs in PyTorch to **reduce CPU overhead**.
  - Allows for larger model components to be **compiled into a single optimized section**.
- **Usage Example**:

```
torch.compile(decode_one_token, mode="reduce-overhead",
                    fullgraph=True)
```

- Available **Modes**:
  - *Default*: For large models, **low compile time**, no extra memory
  - *Reduce-overhead*: **Reduces framework overhead**, uses extra memory, good for **small batches**
  - *Max-autotune*: Produces the fastest model but takes a **very long time to compile**.
- `fullgraph=True` :
  - Minimizes the frequency and impact of "graph breaks"
    - i.e portions that cannot be compiled.
  - Ensures maximum potential utilization of `torch.compile`.

- A common optimisation trick for speeding up **transformer inference**.
  - Activations computed for the previous tokens are cached.
- As more tokens are generated, the “logical length” of the kv-cache grows.
  - Rellocating and copying every time the cache grows!
- Due to this dynamism `torch.compile` **less efficient**
  - i.e. need to recompile.

- Use **“static”** kv-cache:
  - Statically allocate the maximum size of kv-cache
  - **Mask out the unused values** in the attention portion

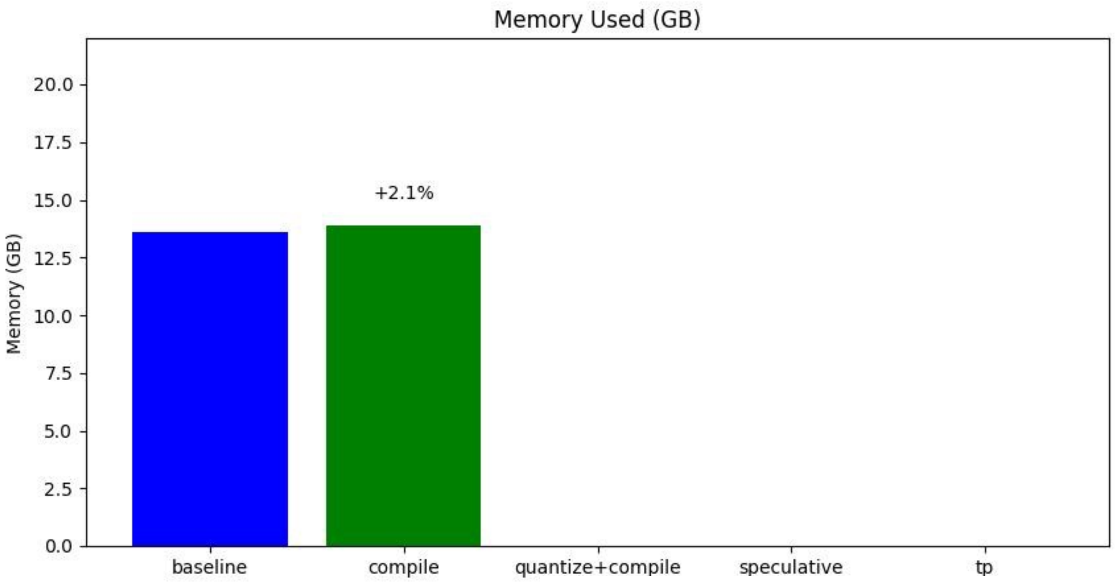
```
with torch.device(device):
    model.setup_caches(max_batch_size=1, max_seq_length=max_seq_length)
```

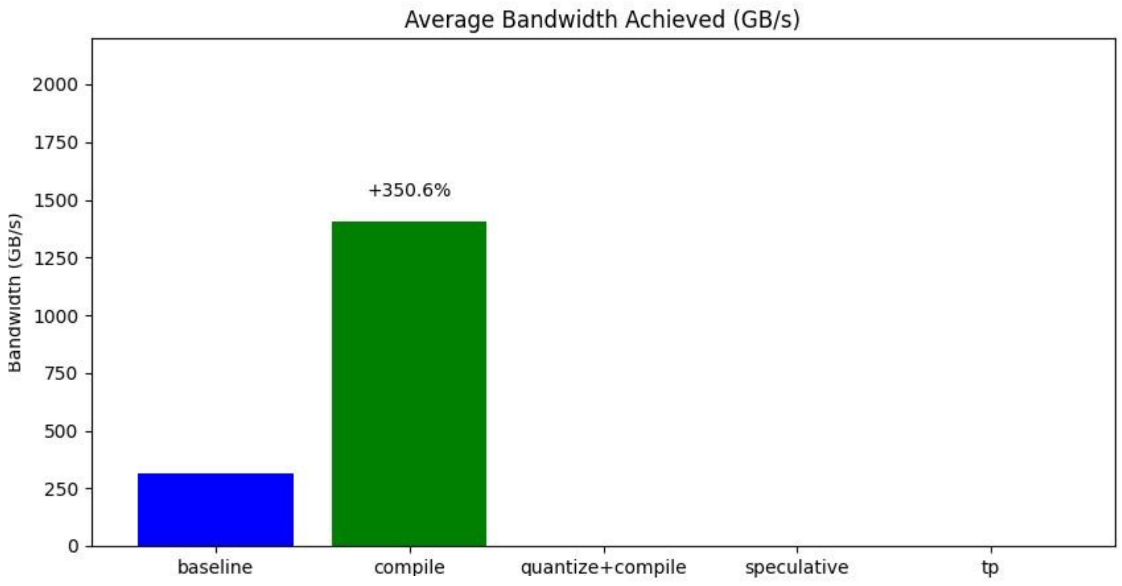
- Two-phase compilation:
  1. The prefill, where the **entire prompt is processed**
     - more **dynamic**, due to variable prompt length

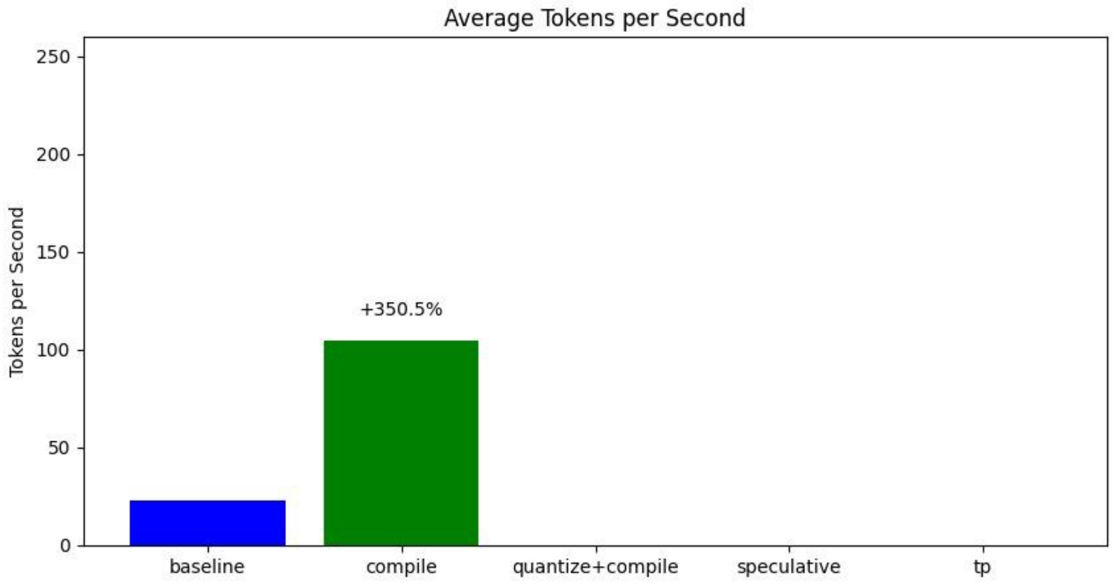
     ```
     prefill = torch.compile(prefill, dynamic=True,
     fullgraph=True)
     ```
  2. Decoding, where each token is generated (kv-cache static)

     ```
     decode_one_token = torch.compile(decode_one_token,
                                      mode="reduce-overhead",
                                             fullgraph=True)
     ```

Running Compile Version (104.37 tokens/s)







- Largest bottleneck now is the cost of loading weights **from GPU global memory to registers**.
  - Happens in each forward pass.
- **Percentage of memory bandwidth** used during inference given by **Model Bandwidth Utilisation (MBU)**:

$$MBU = \frac{\#\,Params\ *\ {}^{bytes}/_{param}\ *\ {}^{tokens}/_{second}}{Memory\ Bandwidth}$$

- In our case, (7B Params, each FP16, and 104 tokens/s):

$$MBU = \frac{7B\ *2\ *104}{1.6\,TB} = 93\%$$

  - Very close to the theoretical limit!
- We can change how many **bytes each parameter is stored in**!

- Quantize only the weights, **computation still done in BF16**.
  - Easy to apply with little to no loss of accuracy.
- Done once, **offline**!
  - Results in reduced memory footprint and faster execution on hardware.
- Here applied per-channel.
- Example:

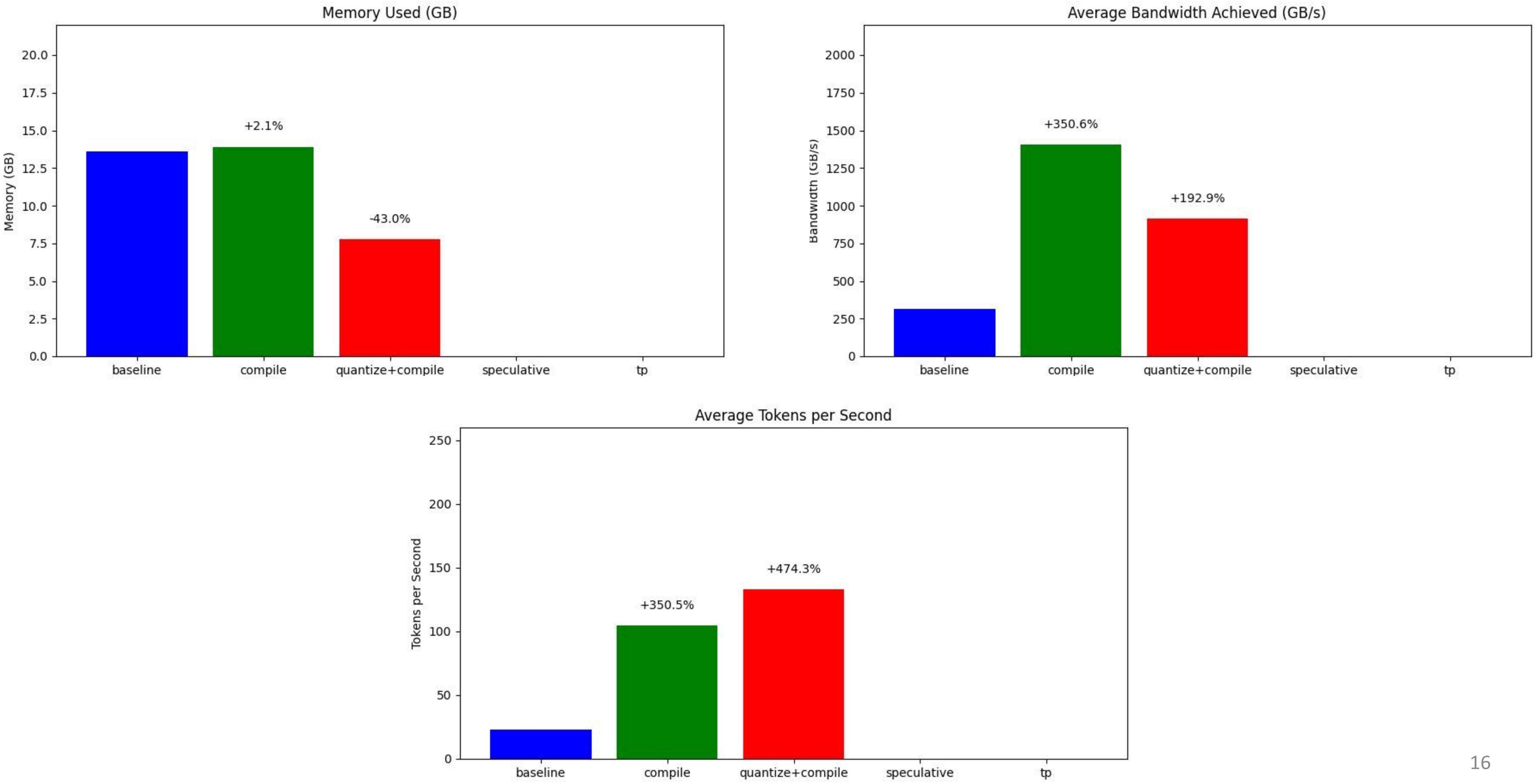
$$X_q = round\left(\frac{127}{\max|X|} * X\right)$$

maps FP16 values into [-127, 127] 8-bit integers!

- Quantized Matrix Multiplication becomes:

```
x: bf16[1, K]
weight: int8[K, N]
@torch.compile
def int8_mm(x, weight):
    return F.linear(x, weight.to(torch.bfloat16))
```

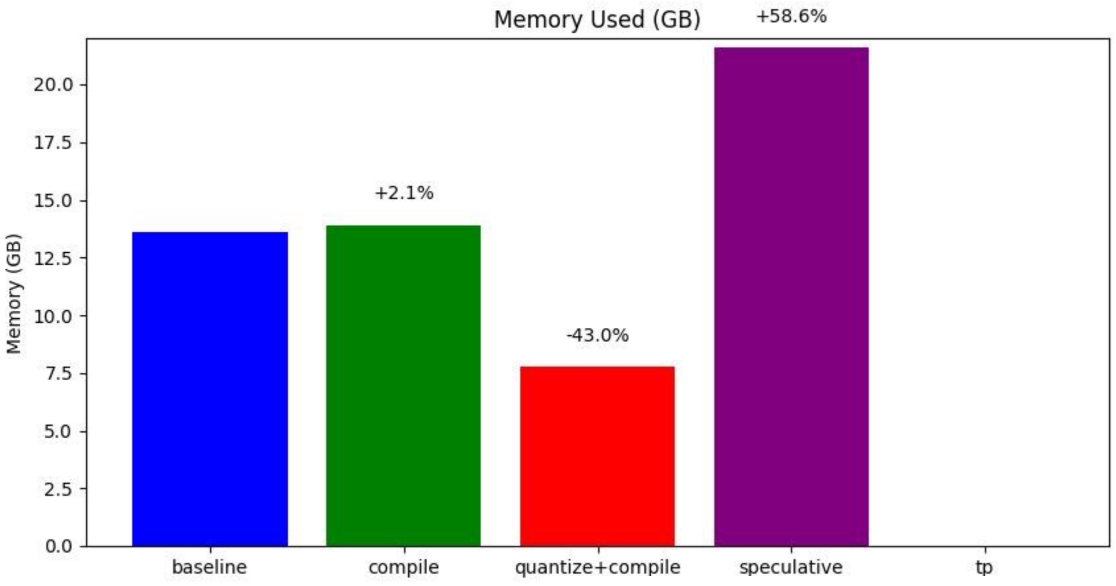
Running Quantized Version (133.07 tokens/s)

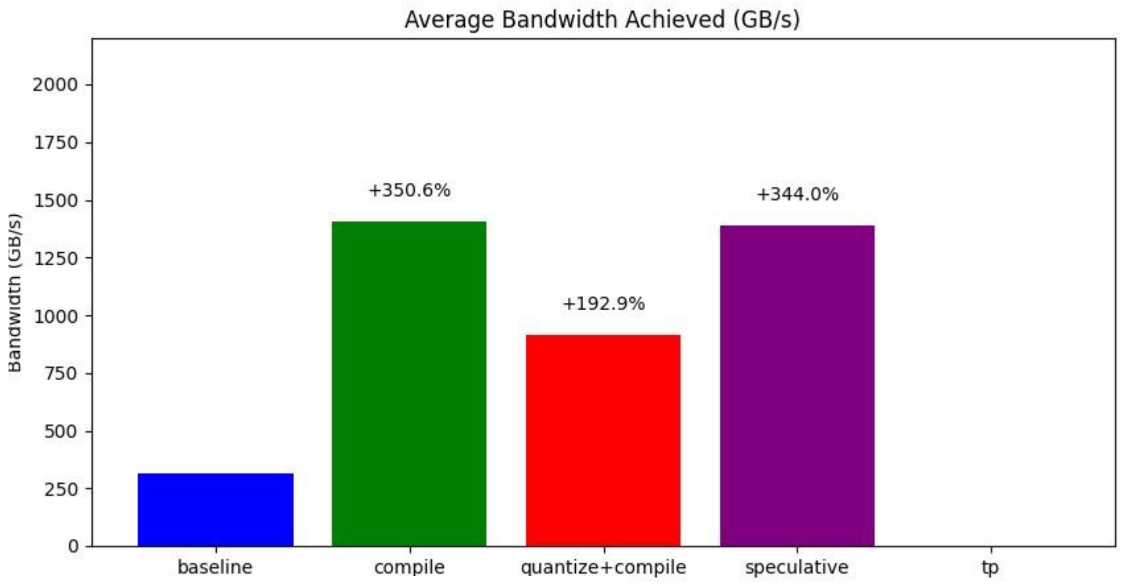


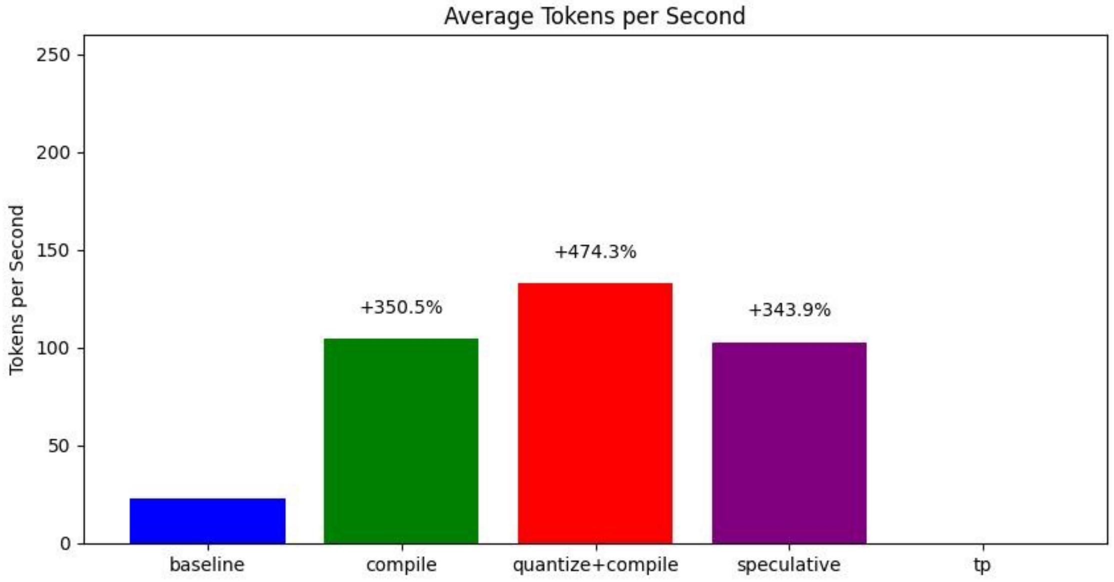
Speculative Decoding

- For every token generated, weights have to be loaded over and over.
  - **Strict serial dependency** in autoregressive generation.
- **Speculative decoding** breaks this dependency!
- Main Idea:
  - Larger model, which we want to use for inference (**Verifier Model**)
  - Smaller model, able to generate text much faster (**Draft Model**)
    - But less accurate!
  - Generate **N tokens using the cheaper** draft model, then process all of them in parallel using the verifier model
    - Those **not matching**, discard and **regenerate with Verifier Model**.
- Speculative decoding **does not change the quality** of the output.
- Around 50 lines of code implementation.
- **Runtime performance varies** depending on the generated text.

Running Speculative Version (102.85 tokens/s)





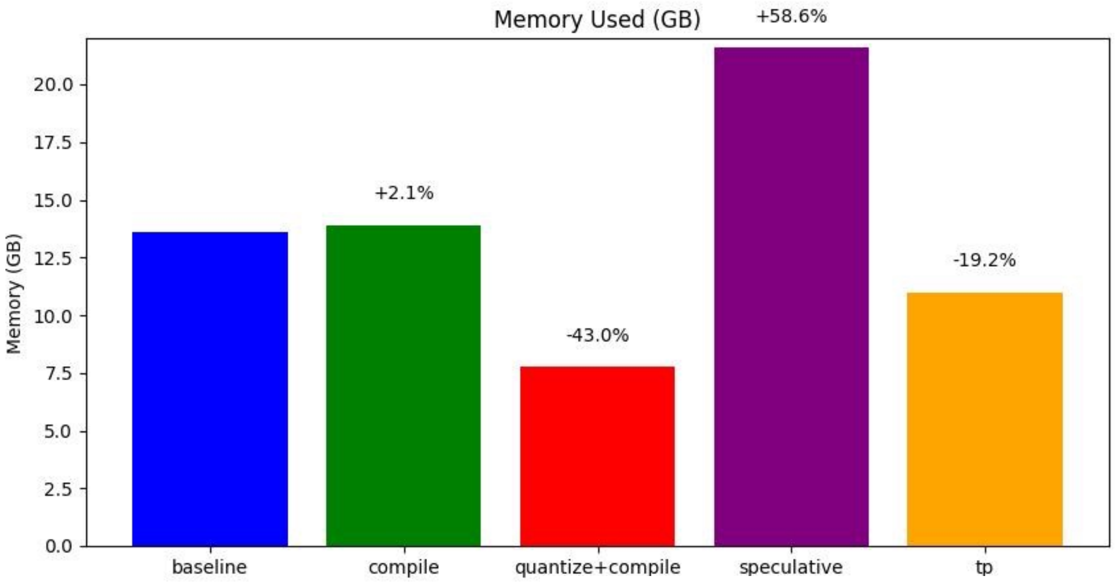


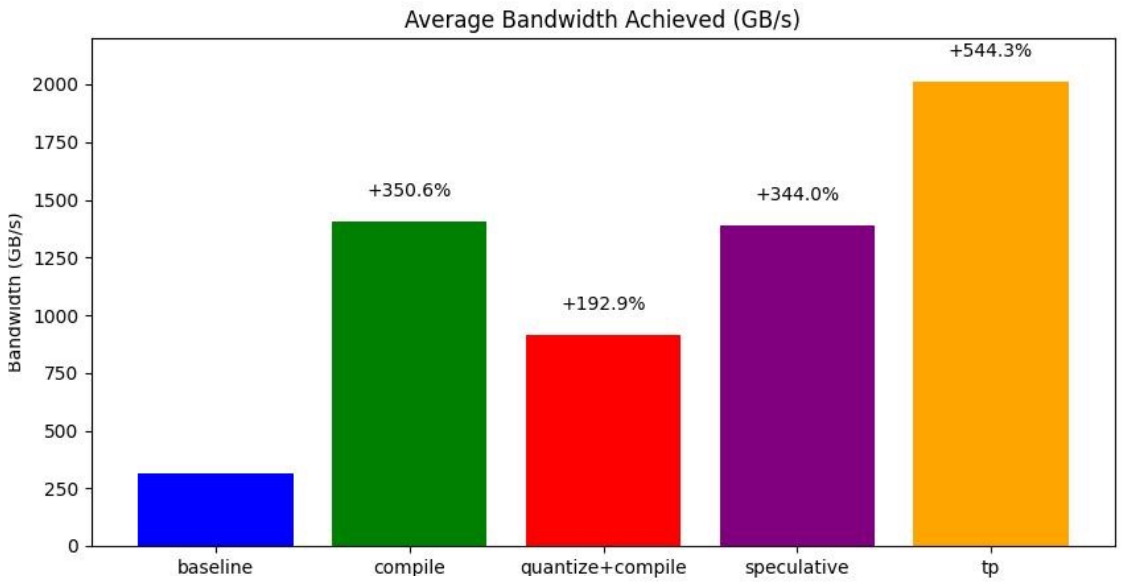
Tensor Parallelism

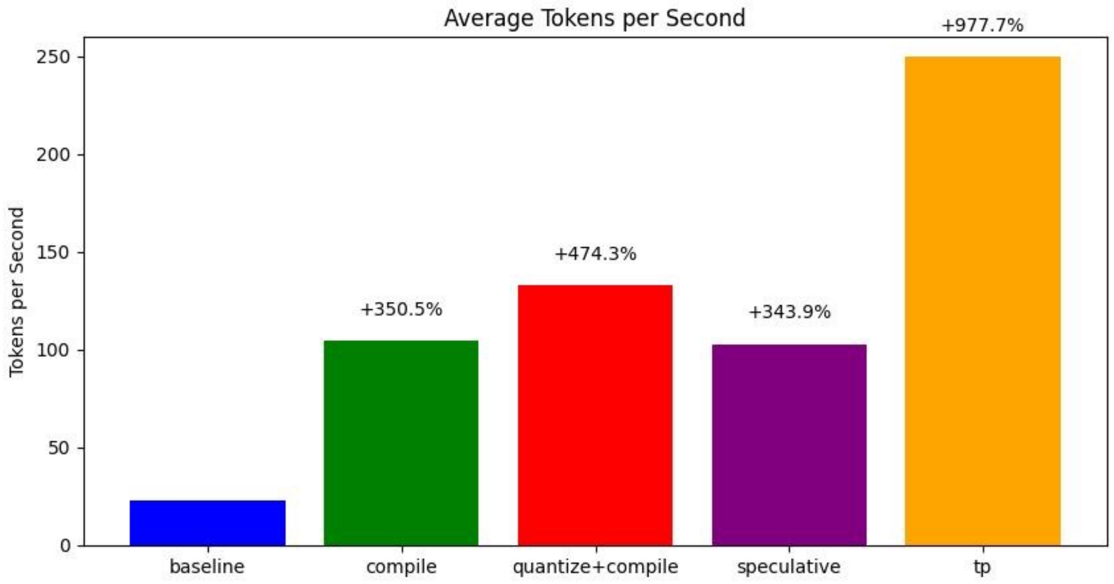
- So far, only one GPU was used!
- Running on more GPUs gives us access to more memory bandwidth.
- Parallelisation strategy is to **split the processing of one token across multiple devices**
  - Tensor Parallelism
- PyTorch supports this, although low-level API currently.
  - 150 lines of code, no model changes!
- Main idea:

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = F.silu(self.c_fc1(x)) * self.c_fc2(x)
    x = self.c_proj(x)
    x = collectives.all_reduce(x, "sum",
                           list(range(LOCAL_WORLD_SIZE)))
    return x
```

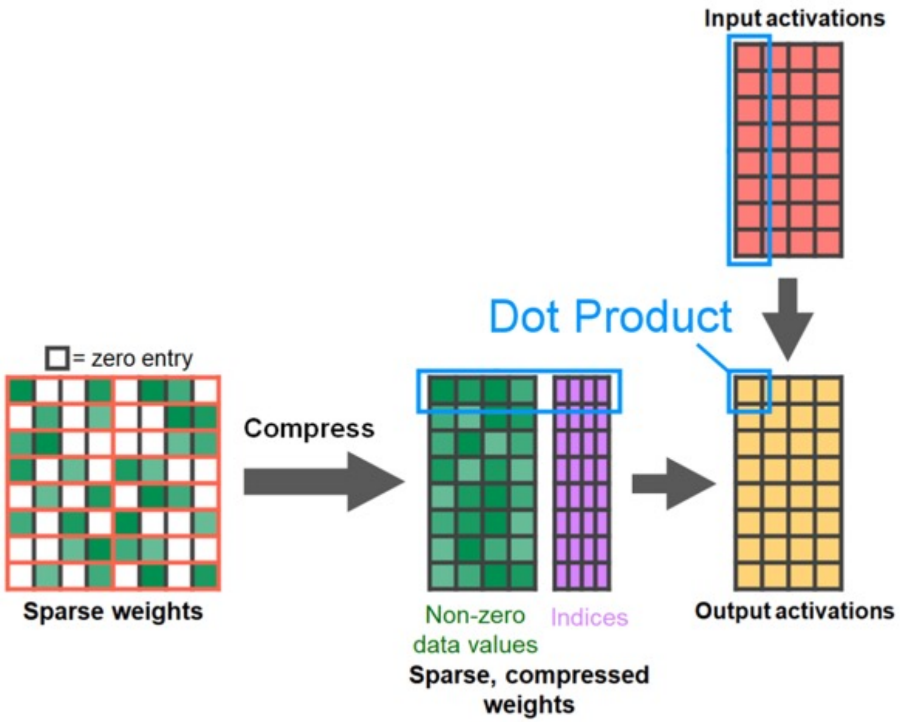
Running Tensor Parallelism Version (250 tokens/s)







- Memory efficient attention implementations
  - Scaled dot product attention (SPDA)
- Semi-structured (2:4) Sparsity
  - Good for Sparsification/Prunning



From developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt

Summary

- Using native PyTorch offers ease of use without sacrificing performance.
- The code for optimisations is around 900 lines.
  - torch.compile
  - Quantization (BF16 to INT8)
  - Speculative Decoding (Expert 7B 16BF, Draft 7B INT8)
  - Tensor Parallelism
- From:
  - **23.17 tokens/second to 250**!
  - 1 GPU to 4 GPUS.
  - Overhead-bound to memory bandwidth bound.

Acknowledgements

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101101903.

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

References

1. https://gu-eresearch.github.io/hpcWorkshop/content/12-logOntoHPC.html