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/EuroCC2PyTorch
- 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 to be used: A100 NVIDIA GPUs 40GB
  - Available are 5 nodes, with 4 GPUs each.
  - For accurate timings `--exclusive` access will be given for each run (i.e 5 users can run at the same time).

Environment Setup

```
# Connect to Cyclone
$ ssh <username>@front02.hpcf.cyi.ac.cy
# Project path containing code/data
$ export $PROJ_PATH=/nvme/scratch/edu20
# Checkpoints contain the Llama2 7B weights
$ ls $PROJ_PATH/gpt-fast
  gpt-fast gpt-fast-checkpoints
# Make a local copy of the code
$ cp -R $PROJ_PATH/gpt-fast .
# Setup conda and activate environment
$ module load Anaconda/2023.03-1
$ conda init
$ conda activate $PROJ_PATH/envs/pytorch
# Code for the session
$ cd gpt-fast
```

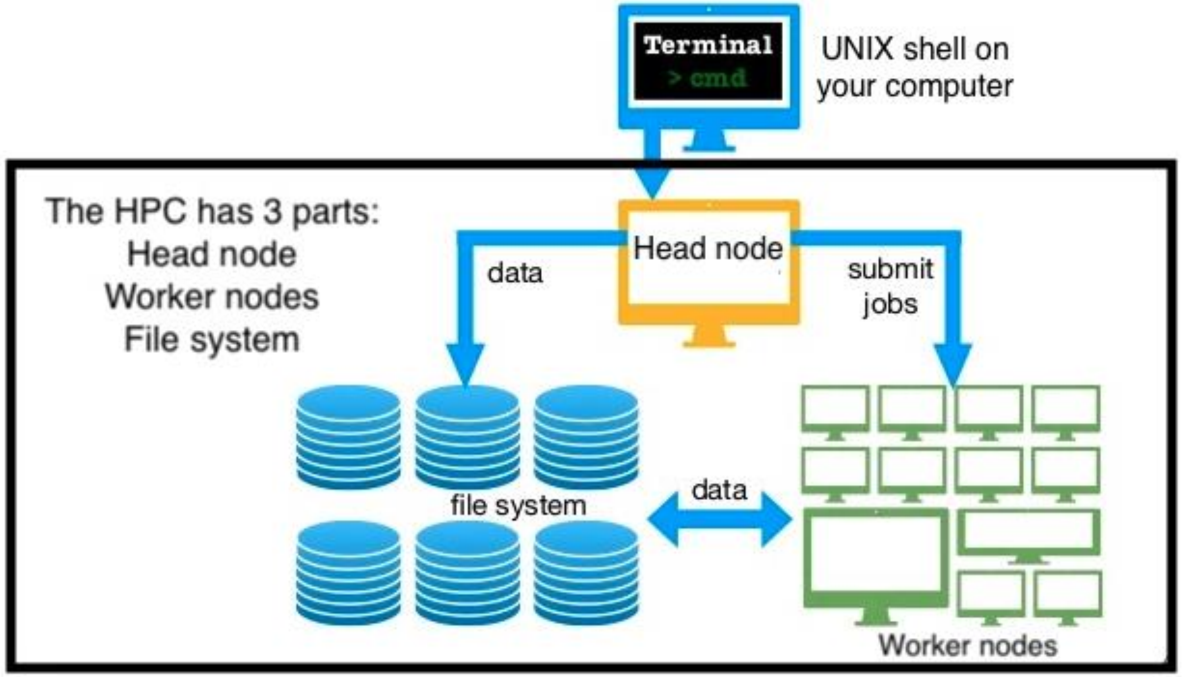


Figure 1: High-level overview of a Supercomputer [1].

Establishing a Baseline

- **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: 2.2

“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

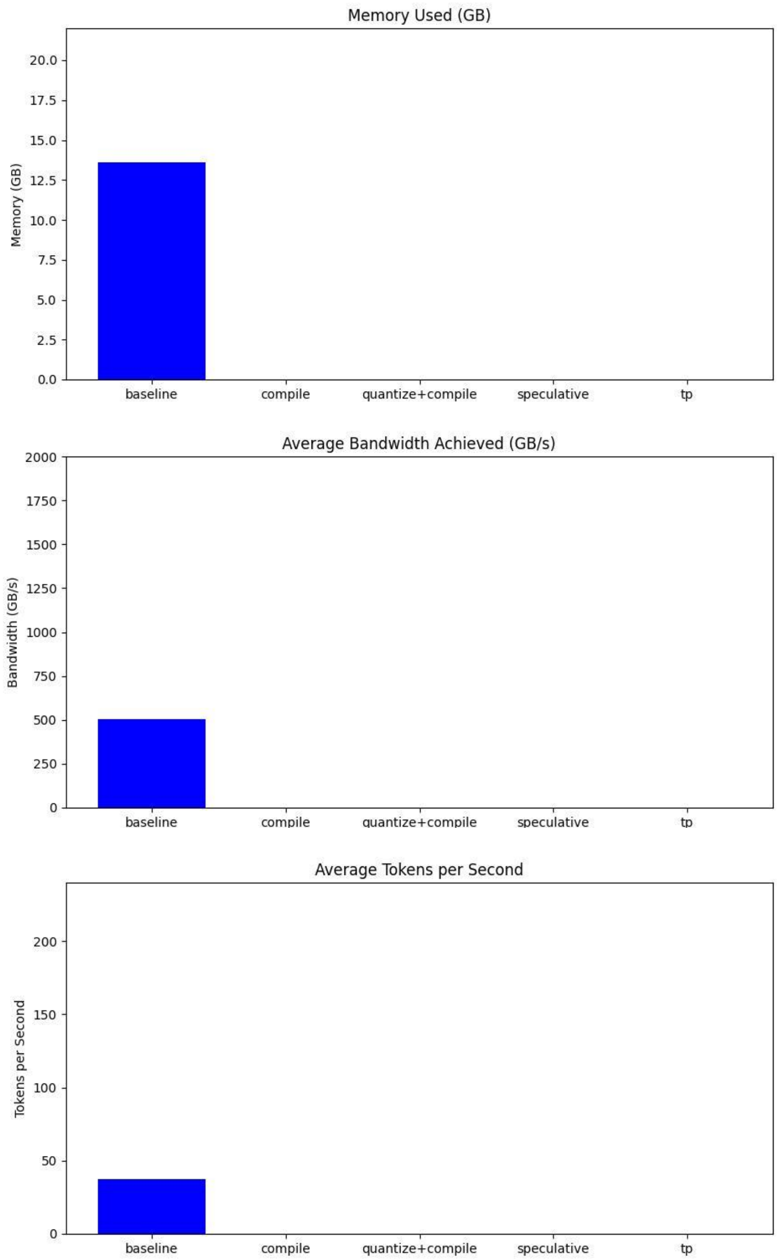
A Note on the Scheduler Settings

```
#!/bin/bash –l

#SBATCH --job-name=gpt-fast-baseline
#SBATCH --account=edu20
#SBATCH --nodes=1                    # Request 1 Node
#SBATCH --ntasks-per-node=1          # Request 1 Process
#SBATCH --gpus=1                     # Request 1 GPU
#SBATCH --cpus-per-task=12           # Request 12 threads (not used, but good for locality)
#SBATCH --partition=a100          # Request on Cluster with A100 NVIDIA GPUs
#SBATCH --time=01:00:00           # Request 1 process
#SBATCH --exclusive               # Request exclusive access on whole node
#SBATCH --exclude=sim02
#SBATCH --output=%x-%j.out
# Setup paths
TRAINING_PATH=/nvme/scratch/edu20
CHECKPOINTS_PATH=$TRAINING_PATH/gpt-fast-checkpoints # Path to Weights
# Setup environment (Shared across all edu20)
module load Anaconda3/2023.03-1
module load CUDA/11.8.0
conda activate $TRAINING_PATH/envs/pytorch
```

Running Baseline Version

```
# Go to Code Repository
$ cd $HOME/gpt-fast
$ sbatch scripts/run.baseline.slurm –reservation=edu20
  Submitted batch job <JOBID>
# Check your job status
$ squeue --user=$USER
  JOBID  PARTITION NAME     USER ST TIME …
 <JOBID>      a100 gpt-fast cstyl R 0:01 …
# Look for the output file
$ ls gpt-fast-baseline
  gpt-fast-baseline-<JOBID>.out
# Plot the results
$ python plots/extract_and_plot.py gpt-fast-baseline- <JOBID>.out
```



The Problem with Baseline Model

- Modern GPUs are **extremely efficient**, completing operations in **microseconds**
  - including tasks like **kernel executions** and **memory transfers**.
- Despite their speed, the process of submitting these operations to the GPU introduces **additional overhead**.
- Complex algorithms often require **multiple GPU operations** to complete a single task.
- When GPU **operations** are **sent individually** and **complete quickly**, the **combined overhead** from each operation can lead to a **noticeable** drop in overall performance.
  - CPU acts as an orchestrator and tries to keep up with the overhead!
  - 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**:

```
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 optimize `torch.nn.Module`
  - Compiles the optimised model/routine into optimised kernels as it executes
    - If structure of the model remains constant (i.e. no recompilation needed) overhead only paid once.
    - Useful if optimised model/routine is used several times.

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 models
  - *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 Compiler for PyTorch Models – KV Caching

- 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_lenght)
```

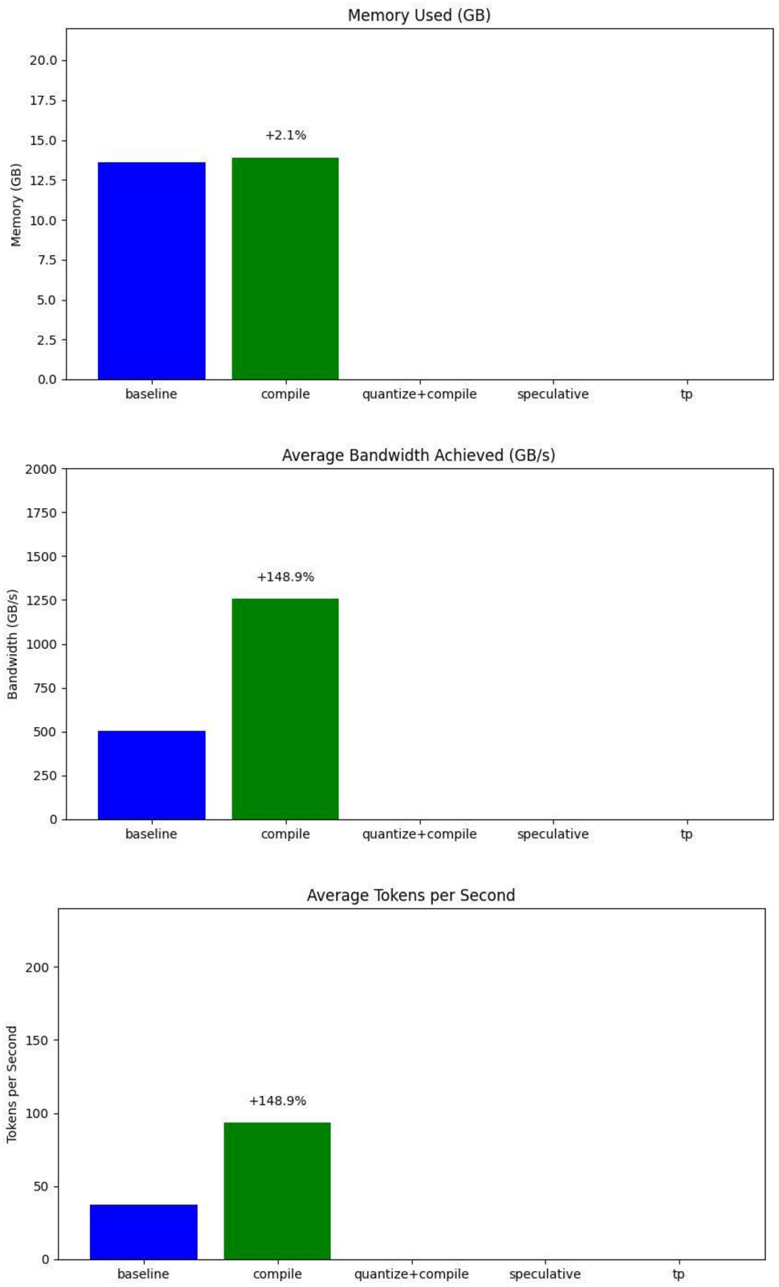
- Two-phase compilation:
  1. The prefill, where the entire prompt is processed
     - more dynamic, due to variable promtp length

     ```
     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

```
# Go to Code Repository
$ cd $HOME/gpt-fast
$ sbatch scripts/run.compile.slurm
  Submitted batch job <JOBID>
# Check your job status
$ squeue --user=$USER
  JOBID   PARTITION NAME     USER ST TIME …
  <JOBID>      a100 gpt-fast cstyl R 0:01 …
# Look for the output file
$ ls gpt-fast-compile
  gpt-fast-compile-<JOBID>.out
# Plot the results
$ python plots/extract_and_plot.py gpt-fast-baseline-488363.out gpt-fast-compile-<JOBID>.out
```



Accelerating Models with Reduced Precision Operations

- 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 Bandiwdth Utilisation (MBU)**:

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

- In our case, (7B Params, each FP16, and 95 tokens/s):

$$MBU = \frac{7B\ *2\ *95}{2\ TB} = 63\%$$

- We can change how many bytes each parameter is stored in!

Accelerating Models with Reduced Precision – From BF16 to INT8

- 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:

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

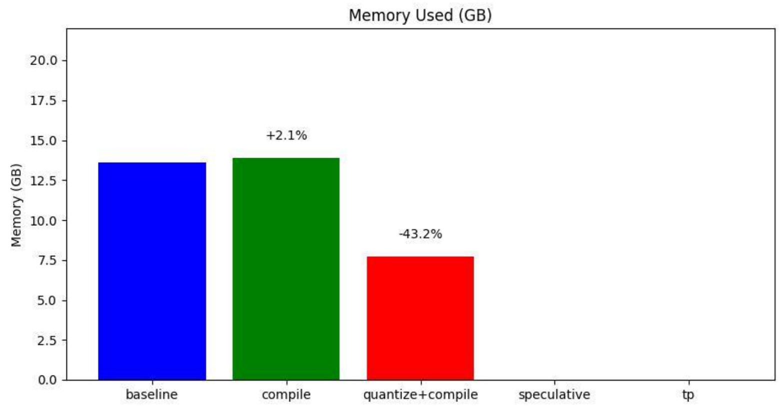
maps FP16 values into [-127, 127] 8-bit integers!

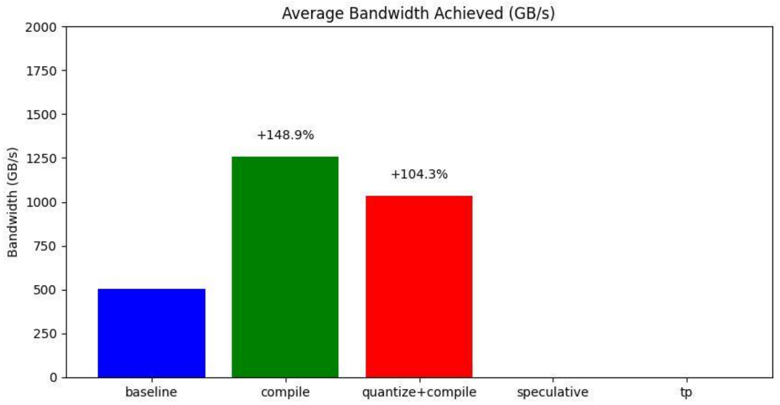
- Quantized Matrix Multiplication becomes:

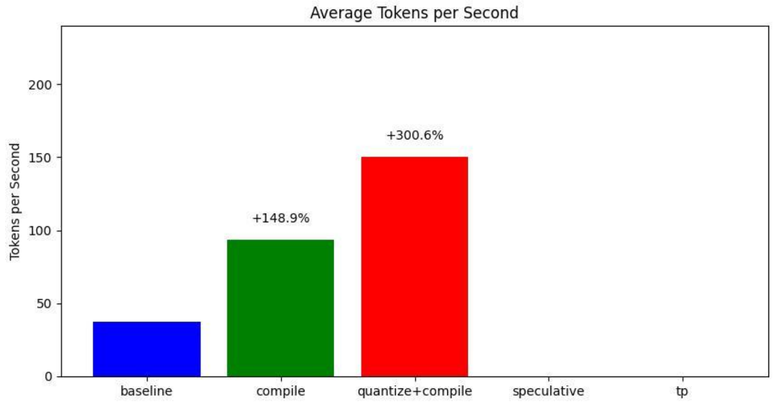
```
x: bf16[1, K]
weight: int8[K, N]
@torch.compile
def int8_mm(x, weight):
    return F.linear(x, weight.to(torch.bfloat16))
```

Running Quantized Version

```
# Go to Code Repository
$ cd $HOME/gpt-fast
$ sbatch scripts/run.quantize.slurm
  Submitted batch job <JOBID>
# Check your job status
$ squeue --user=$USER
  JOBID   PARTITION NAME      USER ST TIME …
  <JOBID>      a100 gpt-fast cstyl R 0:01 …
# Look for the output file
$ ls gpt-fast-quantize
  gpt-fast-quantize-<JOBID>.out
# Plot the results
$ python plots/extract_and_plot.py gpt-
fast-baseline-488363.out gpt-fast-compile-
488364.out gpt-fast-quantize-<JOBID>.out
```





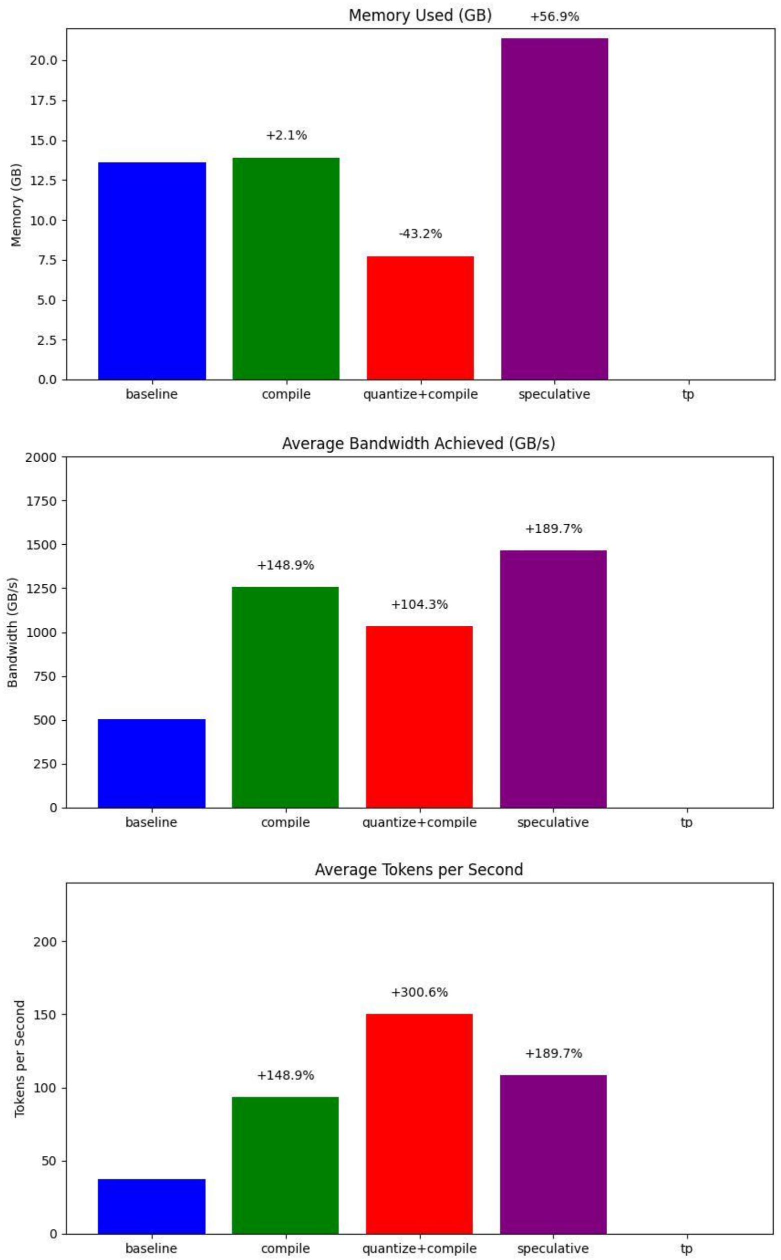


Speculative Decoding

- 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 more inaccurate!
  - 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

```
# Go to Code Repository
$ cd $HOME/gpt-fast
$ sbatch scripts/run.speculative.slurm
  Submitted batch job <JOBID>
# Check your job status
$ squeue --user=$USER
  JOBID   PARTITION NAME     USER ST TIME …
  <JOBID>      a100 gpt-fast cstyl R 0:01 …
# Look for the output file
$ ls gpt-fast-speculative
  gpt-fast-speculative-<JOBID>.out
# Plot the results
$ python plots/extract_and_plot.py gpt-fast-baseline-488363.out gpt-fast-compile-488364.out gpt-fast-quantize-488365.out gpt-fast-speculative-<JOBID>.out
```

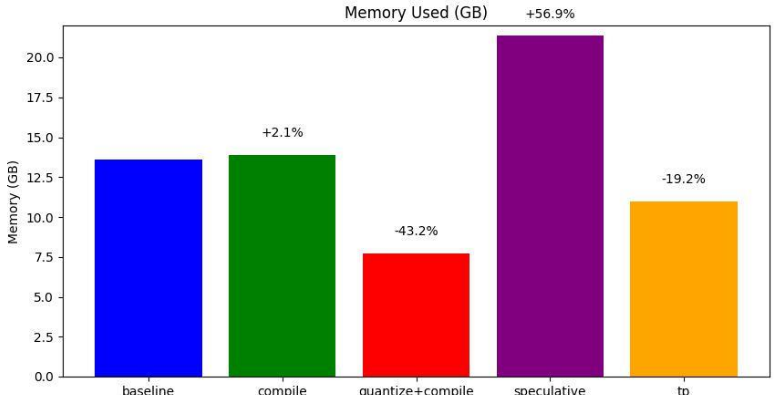


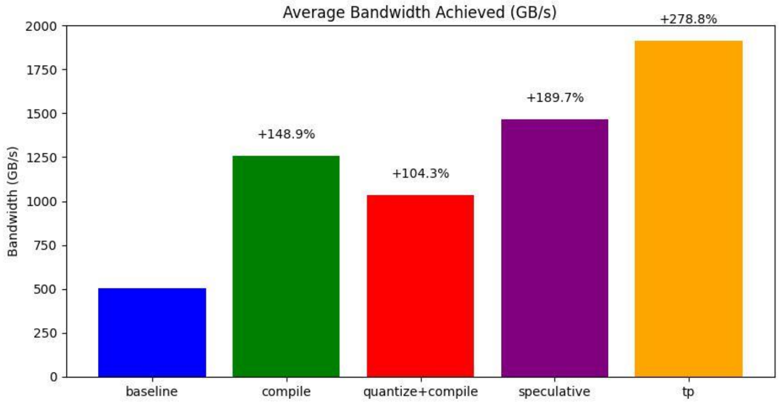
- So far, only one GPU was used!
- Running on more GPUs gives us access to more memory bandwidth.
- Parallelisation strategy is to **split the processing of one token across multiple devices**
  - Tensor Parallelism
- PyTorch supports this, although low-level API currently.
  - 150 lines of code, no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

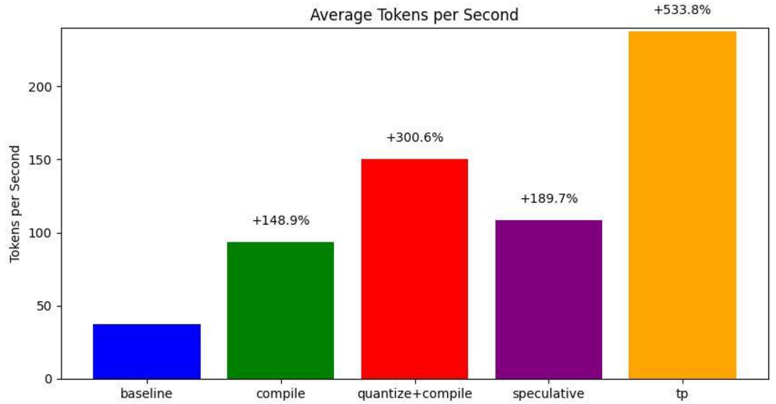
```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = F.silu(self.c_fc1(x)) * self.c_fc2(x)
    x = self.c_proj(x)
    x = collectives.all_reduce(x, “sum”,
                                   list(range(LOCAL_WORLD_SIZE)))
    return x
```

Running Tensor Parallelism Version

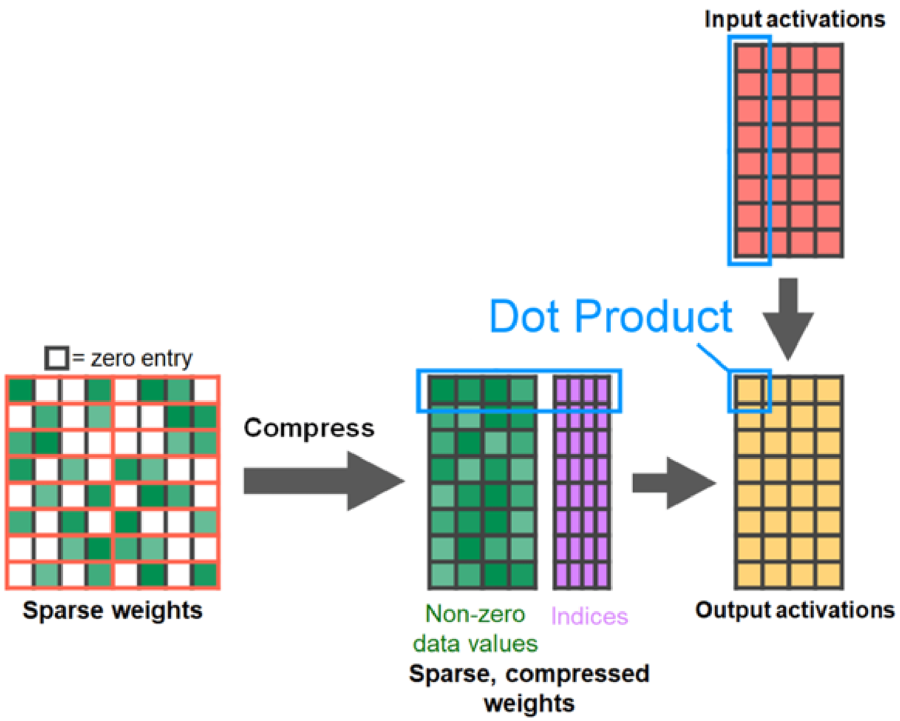
```
# Go to Code Repository
$ cd $HOME/gpt-fast
$ sbatch scripts/run.tp.slurm
  Submitted batch job <JOBID>
# Check your job status
$ squeue --user=$USER
  JOBID  PARTITION NAME     USER ST TIME …
 <JOBID>       a100 gpt-fast cstyl R 0:01 …
# Look for the output file
$ ls gpt-fast-tp
  gpt-fast-tp-<JOBID>.out
# Plot the results
$ python plots/extract_and_plot.py gpt-fast-baseline-488363.out gpt-fast-compile-488364.out gpt-fast-quantize-488365.out gpt-fast-speculative-488366.out gpt-fast-tp-488367.out
```







- 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:
  - **37 tokens/second to 237**!
  - 1 GPU to multiple.
  - 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.

The hands-on sessions have been carried out on the SimEA Partition on Cyclone HPC system at The Cyprus Institute, funded by European Union’s Horizon 2020 research and innovation program (*grant agreement no. 810660, SimEA*).

References

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