Improving Inference With MedSAM Lite

Greg Cheung

Introduction

The MedSAM Lite model makes use of several different pytorch models to segment images. Because of this, the model can induce a significant amount of overhead when computing inference on images. In order to perform inference at a faster rate, some well documented methods were tried out. These methods were inspired from the following 3 part article from pytorch: https://pytorch.org/blog/accelerating-generative-ai/. The main methods tested were the following:

- Torch.compile
- GPU quantization
- Scaled dot product attention
- Semi-structured sparsity
- Nested tensors

Using a slightly MedSAM lite model, each of these methods were tested on the 10 test .npz files, just like on the github instructions. The data is in the directory data/npz/MedSAM_test/CT_Abd. Using emperical data in the form of runtime, we can get an idea of whether or not any of these methods can speed up inference of the MedSAM lite model, and by how much.

The Model Interface

In order to provide easy testing, MedSAM lite was given its own API. First, the model was trained just as described in the github. Once the model was fully trained, the function MedSAM_infer_npz from module inference_3D.py was used as a main framework for the API. This module was reworked into a single class, MedSAM_Interface, which acts as our main API. This interface functions almost identically to MedSAM lite, except now we can change the model, get inference, and save overlays all in one object. The python interface was then used by Django to create a simple to use web browser interface. The choice of Django was somewhat arbitrary, so in theory, any other full stack web development framework should work similar. More information about the interface can be found at https://github.com/gc9340/MedSAM_Lite_interface/tree/main

Testing Acceleration Methods

In order to see how well each acceleration method worked, a **batch inference time** was measured. This batch inference time tracked the time it took to complete inference on all 10 files using 4 workers in parallel. In order to reduce the appearance of any anomalies in computing time, the average time over 3 inferences was used for all metrics. (30 total files) was used as a basic metric. On the hardware side, the graphics card used to run the inference was an NVIDIA GeForce RTX 4050 Laptop GPU, and the operating system was Windows 11. The metrics for all acceleration techniques are given in the table below.

Acceleration Technique	**Inference Time In Seconds**(average over 3 runs)
None	81.18
Torch.compile	79.22
Quantization (weight only)	90.53
SDPA	74.44
Semi-structured sparsity	N/A (Not supported on Windows)
Nested Tensors	N/A (Use with SDPA)

One by one, each of these methods will be analyzed throughout the report. The only exception is semi-structured sparsity, which is currently lacking support on Windows.

Torch.compile

Torch.compile by far was the easiest implementation of a potential acceleration method. When initializing the model, calling Torch.compile just once will optimize matrix multiplications in the model. In code, the difference is as simple as adding one line.

```
medsam_lite_model = MedSAM_Lite( ...

# initialize the model, then add this line.
# max-autotune makes the maximum optimization for matrix operations
medsam_lite_model = torch.compile(medsam_lite_model, mode = "max-autotune")

# now connect the device and start processing
medsam_lite_model.to(device)
medsam_lite_model.eval()

...
```

Overall, compiling was marginal at best in terms of improvements for inference speed. Many different options were used when trying to minimize inference time caused by compile, but none caused any significant decrease. On average, it saved about 2.5% on time, which is not negligible, but not outstanding either. It can have a chain reaction, and cause other acceleration techniques to further accelerate. So all techniques from here on will be tried with and without compile as well. All of the times above, however, are the times reported using compile in conjunction with the other technique since that was the faster. It is of note that compiling works best on larger models,

GPU Quantization

Using torchao, we are able to test different kinds of quantization. The way these were implemented can be found on the torchao website: https://pypi.org/project/torchao-nightly/2024.4.10/. The two main options are dynamic quantization, and weight only quantization. Dynamic quantization was not able to be tested due to an error that could not be resolved. Specifically, when trying to apply dynamic quantization, the error that gets thrown is "RuntimeError: _int_mm_out_cuda not compiled for this platform". Similarly to torch.compile, quantization is very easy to implement.

```
from torchao.quantization import quant_api
medsam_lite_model = MedSAM_Lite( ...

# the following is a recommended flag for optimal use
torch._inductor.config.use_mixed_mm = True


# one line change for int8 weight only quantization
quant_api.change_linear_weights_to_int8_woqtensors(medsam_lite_model)

medsam_lite_model = torch.compile(medsam_lite_model, mode = "max-autotune")
```

```

medsam_lite_model.to(device)
medsam_lite_model.eval()

...
```

Weight only quantization significantly slowed inference down, possibly due to the overhead incurred not being worth the small amount of weight loading being sped up. Since MedSAM lite doesn't have much overhead, some techniques like quantization can have a negative affect on runtime. Quantization was used both with, and without compiling, but the results were similar. Compiling had a slight edge, and the time in the table (90.53 seconds) is the time it took with compiling.

Scaled Dot Product Attention

The scaled dot product attention (SDPA) method could be used instead of manually computing it using matrix multiplation and softmax. In order to implement this, the Attention class from tiny_vit_sam.py had to be modified. Specifically, in the forward method, the calculation was replaced by torch.nn.functional.scaled_dot_product_attention. Below you can see the old way it was calculated. Specifically, lines 21 - 27

```
...

class Attention(torch.nn.Module):
...
    def forward(self, x):  # x (B,N,C)
        B, N, _ = x.shape

        # Normalization
        x = self.norm(x)

        qkv = self.qkv(x)
        # (B, N, num_heads, d)
        q, k, v = qkv.view(B, N, self.num_heads, -
                           1).split([self.key_dim, self.key_dim, self.d], dim=3)
        # (B, num_heads, N, d)
        q = q.permute(0, 2, 1, 3)
        k = k.permute(0, 2, 1, 3)
        v = v.permute(0, 2, 1, 3)

        # Old way of calculating the SDPA
        attn = (
                    (q @ k.transpose(-2, -1)) * self.scale
                    +
                    (self.attention_biases[:, self.attention_bias_idxs]
                        if self.training else self.ab)
                )
        attn = attn.softmax(dim=-1)
        x = (attn @ v).transpose(1, 2).reshape(B, N, self.dh)



        x = self.proj(x)
        return x

...
```

On the next page, there is the implementation of the pytorch SDPA method.

```python
import torch.nn.functional as F
...

class Attention(torch.nn.Module):
...
    def forward(self, x):  # x (B,N,C)
        B, N, _ = x.shape

        # Normalization
        x = self.norm(x)

        qkv = self.qkv(x)
        # (B, N, num_heads, d)
        q, k, v = qkv.view(B, N, self.num_heads, -
                           1).split([self.key_dim, self.key_dim, self.d], dim=3)
        # (B, num_heads, N, d)
        q = q.permute(0, 2, 1, 3)
        k = k.permute(0, 2, 1, 3)
        v = v.permute(0, 2, 1, 3)

        # one line way of calculating the SDPA
        sdpa = F.scaled_dot_product_attention(q, k, v, attn_mask = self.ab, scale = self.
    scale)


        x = sdpa.transpose(1,2).reshape(B, N, self.dh)
        x = self.proj(x)
        return x

...
```

This was the first significant improvement in runtime for inference. The built in efficiency of this method improves the run time by about 8.3% (81.18 seconds to 74.44 seconds). This was a relatively simple fix that made a drastic improvement, and can become more common as pytorch further implement solutions.

Nested Tensors

Nested tensors can be useful for accelerating inference when the data becomes heavily padded. This extra padding usually has redundant information that does not need to be carried over in memory or processed. Nested tensors allow us to avoid processing unnecessary padding, while also giving us the ability to process tensors of different sizes simultaneously.

Similar to the SDPA, nested tensors would have to be implemented to work in tiny_vit_sam.py, or whichever library is used for the Attention. Implementing this in its entirety is somewhat difficult due to issues with shaping and viewing different sized tensors. So instead, we can use a proof of concept to see how effective it would be to implement. We will implement nested tensors in the Attention.forward method, and compare how well it does against the regular method in tiny_vit_sam. We will run Attention.forward 1000 times for each input, and track the average run time on 1000 iterations is. On the next page is an implementation of nested tensors and the equivalent regular tensor in the TinyVitBlock.forward method.

```
...
class Attention(torch.nn.Module):
...
    def forward(self, x):  # x (B,N,C)
            ...
            x = x.view(B, H, W, C)
            pad_b = (self.window_size - H % self.window_size) % self.window_size
            pad_r = (self.window_size - W %
                     self.window_size) % self.window_size
            padding = pad_b > 0 or pad_r > 0
            nH = (H + pad_b)//self.window_size
            nW = (W + pad_r) // self.window_size
            # nested tensors
            split_list = [i * self.window_size for i in range(1, nH)]
            first_tensors = torch.tensor_split(x, split_list, dim = 1)
            split_list = [i * self.window_size for i in range(1, nW)]
            jagged_tensors = [st.reshape(st.shape[1]*st.shape[2], st.shape[3]) for ft in
    first_tensors\
                   for st in torch.tensor_split(ft, split_list, dim = 2)]
            # padded tensors
            if padding:
                x = F.pad(x, (0, 0, 0, pad_r, 0, pad_b))
            x = x.view(B, nH, self.window_size, nW, self.window_size, C).transpose(2, 3).
    reshape(B * nH * nW, self.window_size * self.window_size, C)
            # jagged_tensors == x without padding
            # time the nested tensors
            time_nested_start = time()
            for _ in range(1000):
                new_x = self.attn(jagged_tensors)
            time_nested_stop = time()
            print("Nested: ", time_nested_stop - time_nested_start)
            # time the padded tensors
            time_padded_start = time()
            for _ in range(1000):
                new_x = self.attn(x)
            time_padded_stop = time()
            print("Nested: ", time_nested_stop - time_nested_start)
            ...
...
```

These both end up passing in their respected tensors into the Attention.forward method. Since the rest of the code could not be tested with nested tensors, each file was run for 3 minutes total. The average of all times was recorded and can be found in the table below.

Acceleration Method	**Attention.forward Runtime** (average 1000 runs)
None	2.31 seconds
Nested Tensors	14.49 seconds

This is much slower than the usual method. However, it should be noted that nested tensors are said to work incredibly well in conjuction with pytorch's sdpa method (see https://pytorch.org/tutorials/intermediate/

scaled_dot_product_attention_tutorial.html for further information). Below are the runtimes using the regular method, just sdpa, and spda with nested tensors.

Acceleration Method	**Attention.forward Runtime** (average 1000 runs)
None	2.31 seconds
pytorch SDPA	1.25 seconds
pytorch SDPA and Nested Tensors	1.16 seconds

We can see that now, the nested tensors have tremendous benefit when used in conjunction with the sdpa method. Although, for data that did not require too much padding (or none at all), nested tensors performed slightly worse. It also seems like implementing nested tensors could be very laborious to do, and it is currently labeled as a prototype feature. So getting it to work with MedSAM would likely come with some uncharted territory. While implementations for nested tensors continues to develop, the SDPA method is a nice alternative until then.

Conclusion

Overall, these methods tested proved to have some benefit on model inference. Specifically, the SDPA and nested tensors seem so have a great impact, despite the hardware and software environments not being ideal. In the interface, only compile and the SDPA method are used due to not being able to fully implement other acceleration methods yet. If the hardware running the code is built for the compile method, it is suspected that would greatly improve inference time even further. Nested tensors could also be implemented for just a few methods if those methods are used frequently, and can benefit from unpadded data. However, it may be better to wait until pytorch can implement more of the nested tensor features.