Project Team 10 Report

Topic: CUDA Stable Diffusion Algorithm

Introduction:

In this project, we modify the existing open-source implementations of the diffusion models [1], which uses high level APIs(pytorch). We substitute the CrossAttention function into our own version. And then we use my_scribble2image.py (originally gradio_scribble2image.py) to check if our function works. At the end, we profile our CrossAttention function on Nsight Compute to compare our performance with the original CrossAttention.

Motivation:

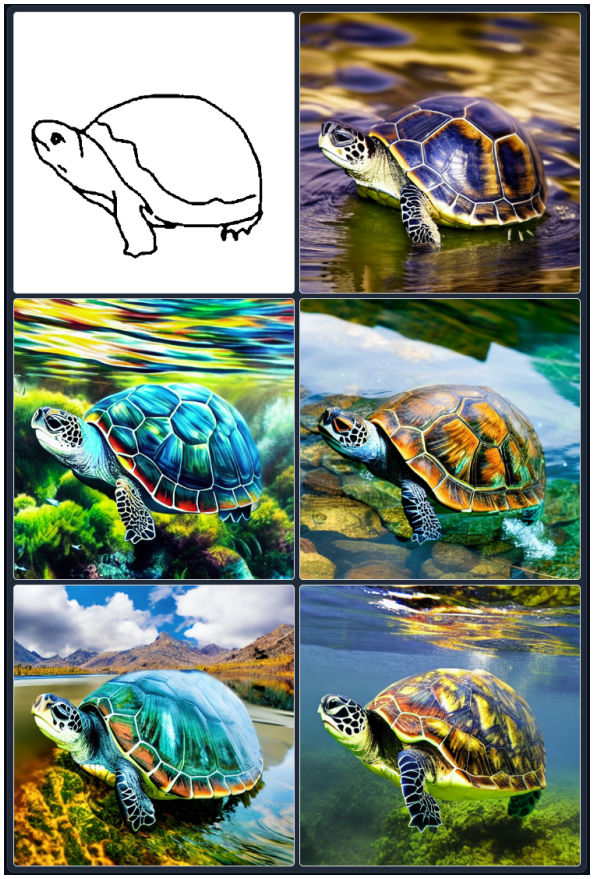
- [2] has shown that CUDA based implementations of the MLP networks can achieve approximately 1OOM improvement in training and inference times compared to their high level (tensorflow based) implementations.
- The frame generation latency scales with the resolution of the frame, which makes matter even worse for higher resolution images.

Hence, it might make sense to accelerate the stable diffusion algorithm using GPU programming languages. We try to implement the state of the art stable discussion algorithm [1] in GPU programming languages (CUDA) where “hardware-aware” fusion of certain layers of diffusion models can be performed, which can potentially exploit more and more on-chip data reuse for intermediate outputs and avoid more expensive off-chip DRAM accesses.

Background and Related Work

- Previous CPU/FPGA/GPU results, including citations if applicable.
  https://github.com/lllyasviel/ControlNet
  The author provided several ControlNet models that add conditional control to Text-to-Image Diffusion Models. We chose Scribbles as our implementation. The

following turtles is one showcased result.



- What's the important thing to measure overall (end-to-end execution on system, disk-to-disk, throughput, etc.)?
  Our goal is to reduce the back-and-forth process from python to CUDA in the original model, hence the most important thing to measure is going to be end-to-end speed.

- Explanation of any specific algorithms to be parallelized (pseudo-code) and estimate of what fraction of time they take; it's fine if you found these by profiling CPU code; if there are too many, explain that.

Baseline

- What's the baseline design against which you optimized?

  This is the diagram of an Attention block(From [3]). It consists of several multiple matrix multiplications, some matrix additions, and some reshape. In pytorch, this is done by functions such as nn.Linear, torch.einsum, rearrange, etc. We try to optimize these functions by implementing them in CUDA.
  The design of attention block is as follows:

  1. 3 matrix multiplications to get query, key, value.
  2. Some rearrangement for later calculation.
  3. Batch matrix multiplication on query and key.
  4. Softmax on the previous result.
  5. Another batch matrix multiplication on previous result and value.
  6. Rearrangement into desired shape.
  7. Matrix multiplication to get output.

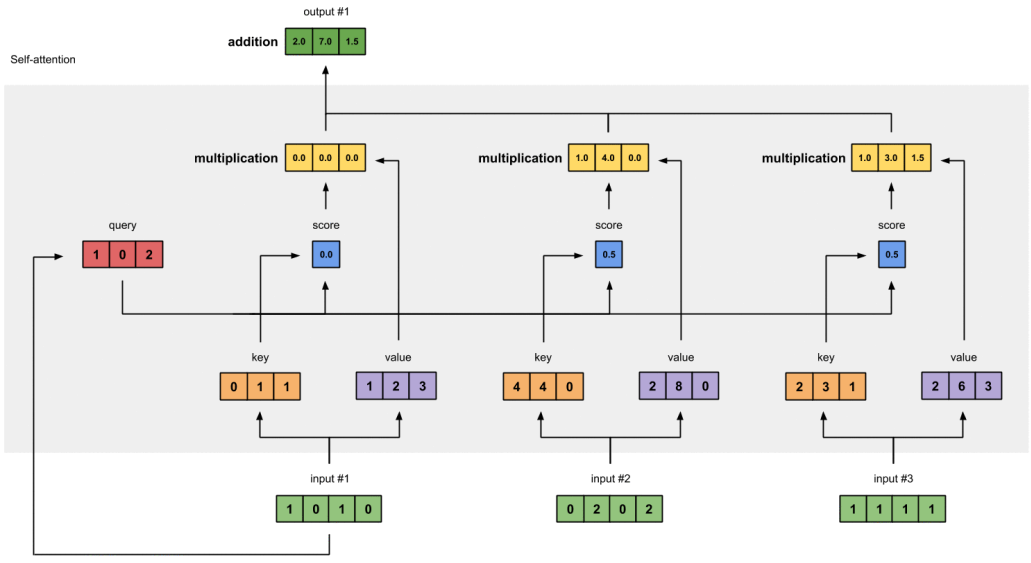


Figure 1

- Did you have to reimplement from scratch, or were you able to make use of an available software system?

  Our plan was to replace the attention block from attention.py in CrossAttention and plug it back into it. It is copied to our repository at crossattn_copied.py. However, although being able to reproduce the results using our CUDA

implementation, the entire project is too big to fit in nsight. Hence, the final result of optimization only shows the block itself.

- Did you need to do any work to connect existing software to CUDA so that you could optimize? Were there/are there overheads for such connections?

  To implement our own CUDA kernel we have to first convert the Pytorch code to PyTorch C++. We used nv-nsight-cu-cli command to compare the original Python CrossAttention function and the equivalent one using PyTorch C++ API. Since RAI couldn't handle the whole model in nsight, even if there is overhead for moving between python and C++, we wouldn't be able to see it. Just for comparison, the two versions of the original design are shown below. We only show the functions that we do end up implementing.

ID	Iss	Function Name	De	Pr	De	Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread]
0	**1**	**volta_sgemm_128x64_tn**	...	...	...	**384, 64, 1**	**128, 1, 1**	**136,120**	**120.19**	**72.57**	**36.72**	**122**
5	1	volta_sgemm_128x64_tn	...	...	...	128, 64, 8	128, 1, 1	53,661	53.34	63.14	46.53	122
11	1	volta_sgemm_128x64_tn	...	...	...	384, 64, 1	128, 1, 1	141,020	124.10	70.28	35.92	122
8	1	volta_sgemm_128x64_nn	...	...	...	128, 64, 8	128, 1, 1	88,670	79.62	75.09	40.37	122

Python version

ID	Iss	Function Name	De	Pr	De	Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread
0	**1**	**volta_sgemm_128x64_tn**	...	...	...	**384, 64, 1**	**128, 1, 1**	**137,208**	**121.18**	**72.07**	**36.46**	**122**
5	1	volta_sgemm_128x64_tn	...	...	...	128, 64, 8	128, 1, 1	54,533	54.02	62.26	46.03	122
10	1	volta_sgemm_128x64_tn	...	...	...	384, 64, 1	128, 1, 1	136,357	119.49	72.51	36.69	122
8	1	volta_sgemm_128x64_nn	...	...	...	128, 64, 8	128, 1, 1	87,765	80.42	75.90	40.82	122

Cpp version

Optimizations and Results

- Most people like to organize such topics as a sequence of optimizations with results for the baseline and after each step, but anything is fine so long as you have numbers for the baseline and the final version.
  Let's see the result first.

ID	Iss	Function Name	Pr	De	Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread
1	2	mysgemm_linear_kernel	...	...	4096, 20, 1	1024, 1, 1	1,020,454	851.68	34.28	68.42	40
3	2	mysgemm_linear_kernel	...	...	1024, 20, 1	1024, 1, 1	967,939	803.97	21.65	23.30	40
5	2	mysgemm_linear_kernel	...	...	1024, 20, 1	1024, 1, 1	968,523	803.87	21.63	23.28	40
12	2	mysgemm_linear_kernel	...	...	4096, 20, 1	1024, 1, 1	1,031,118	857.18	33.97	67.76	40

  We can see that our implementation is much worse than the original design.

- Explain any techniques used
  In the original architecture, the matrix multiplications are done by using nn.Linear, while batch matrix multiplications (bmm for short) are done using einsum. We implemented linear with ease and guaranteed the whole model will still give us correct output. However, we got stuck on bmm and softmax. They seemed easy to implement as well, but we just couldn't get them right. Also, while one teammate is doing implementation, other teammates found out that our

implementation of normal matrix multiplication did terrible after profiling. Thus we gave up after hundreds of tries.

- Illustrate code if possible (pseudo-code is fine, if that's easier).
  The code for our cpp implementation can be found in python/crossattn/cpp/gten_cuda.cu. It is basically a variation of the SGEMM lab.

  1. 3 matrix multiplications to get query, key, value.

```
// Matrix Multiplication of q, k, v from input and context and their corresponding weights
Tensor q = sgemm_linear(x, q_weight, torch::empty({ 0 }), 1, 0);
Tensor k = sgemm_linear(context, k_weight, torch::empty({ 0 }), 1, 0);
Tensor v = sgemm_linear(context, v_weight, torch::empty({ 0 }), 1, 0);
cudaDeviceSynchronize();
```

  2. Some rearrangement for later calculation.

```
// Rearrangement of q, k, v
int b = q.size(0);
int n = q.size(1);
int d = q.size(2) / h;
q = rearrange(q, h);
k = rearrange(k, h);
v = rearrange(v, h);
v = v.permute({ 0, 2, 1 });
```

     Rearrange is a function that reshapes tensors from 3D to 4D using h, permute, and then reshape back into 3D.

```
Tensor rearrange(Tensor& tensor, int h) {
  int b = tensor.size(0);
  int n = tensor.size(1);
  int d = tensor.size(2) / h;
  tensor = tensor.reshape({ b, n, h, d });
  tensor = tensor.permute({ 0, 2, 1, 3 });
  tensor = tensor.reshape({ b * h, n, d });
  return tensor;
}
```

  3. Batch matrix multiplication on query and key. (Failed to implement)

```
// Batch Matrix Multiplication
Tensor sim = torch::einsum("b i d, b j d -> b i j", { q, k }) * scale;
```

  4. Softmax on the previous result. (Failed to implement)

```
sim = sim.softmax(-1);
```

5. Another batch matrix multiplication on previous result and value. (Failed to implement)

```
// Batch Matrix Multiplication
Tensor out = torch::einsum("b i d, b j d -> b i j", { sim, v });
```

6. Rearrangement into desired shape.

```
// Rearrangement of output
out = out.reshape({ b, h, n, d });
out = out.permute({ 0, 2, 1, 3 });
out = out.reshape({ b, n, h * d });
```

7. Matrix multiplication to get output.

```
// Matrix Multiplication
out = sgemm_linear(out, out_weight, out_bias, 1, 0);
cudaDeviceSynchronize();
```

Conclusions

Our implementation fell short of beating the original volta SGEMM, and there could be multiple reasons for this, including poor parameter choices and a a less efficient algorithm. However, we did see improvements from parallelizing the computation of q, k, v and synchronizing only after all three. Additionally, we didn't implement streaming since there was no need for memory copy in our case. Looking towards the future, there is still room for further optimization and exploration of different approaches to achieve even better performance.

References

[1] https://github.com/lllyasviel/ControlNet#controlnet-with-user-scribbles
[2] Real-time Neural Radiance Caching for Path Tracing | Research (nvidia.com)
[3] https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a