Pix2Pix Optimizing

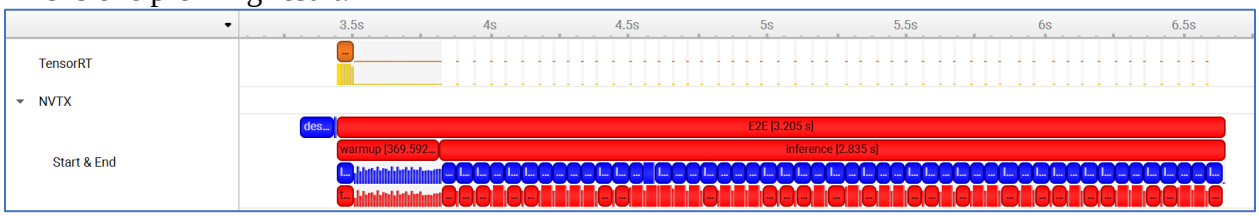
Contents

1 Pix2Pix Optimizing

1.1 Performance Optimization

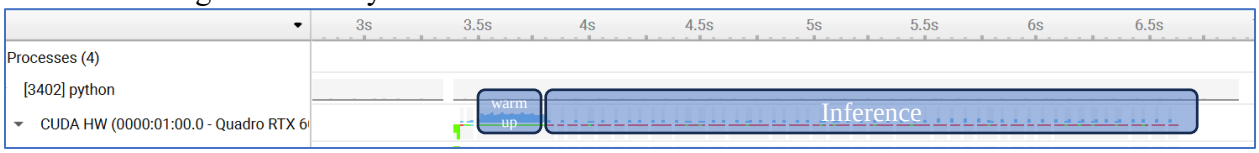
1.1.1 Nsight system profiling & nvtx annotations

Generally using Nsight system is the simplest way to get high-level performance information. Nsight system shows the details of each CUDA/TensorRT/cuBLAS/cuDNN/etc kernel calls, and nvtx can help to group them into meaningful stages, like “init”, “data loading”, “pre-processing”, “inference 0”, “frame encoding”, “warmup” …
This is one profiling result:



We have stages “warmup”, “E2E”, “inference” and any other user arbitrary defined stages names which can help he/her positioning the bottleneck asap.
Here shows “warmup” is much faster than “inference” stage, as I remove “save image” from “warmup”. “save image” occupies too much time than pix2pix model inference as we zoom in the timeline:

Each “save” stage occupies about 55ms and “trt” inference stage only with about 3ms. This is not the right way and we need to move “save” from main pipeline, to create another CUDA stream to handle it, later.
This “save” stage GPU utility much:

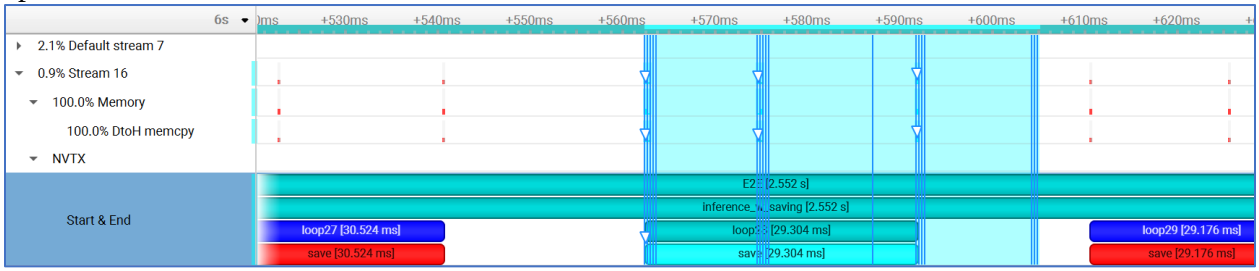


We can see from this figure that for my RTX6000 server, GPU utility of “warmup” is much higher than “inference” stage, as “save” stage has many CPU workload and GPU is just waiting for it to finish.
Each pix2pix inference has a “image load” and “image store” stage, which involves H2D/D2H memory copy. These operations can be parallel with AI inference, while running AI inference for image_n, we can parallel to load image_n+1, and storing image_n-1. GPU support H2D/D2H to executing parallel with CUDA computing.

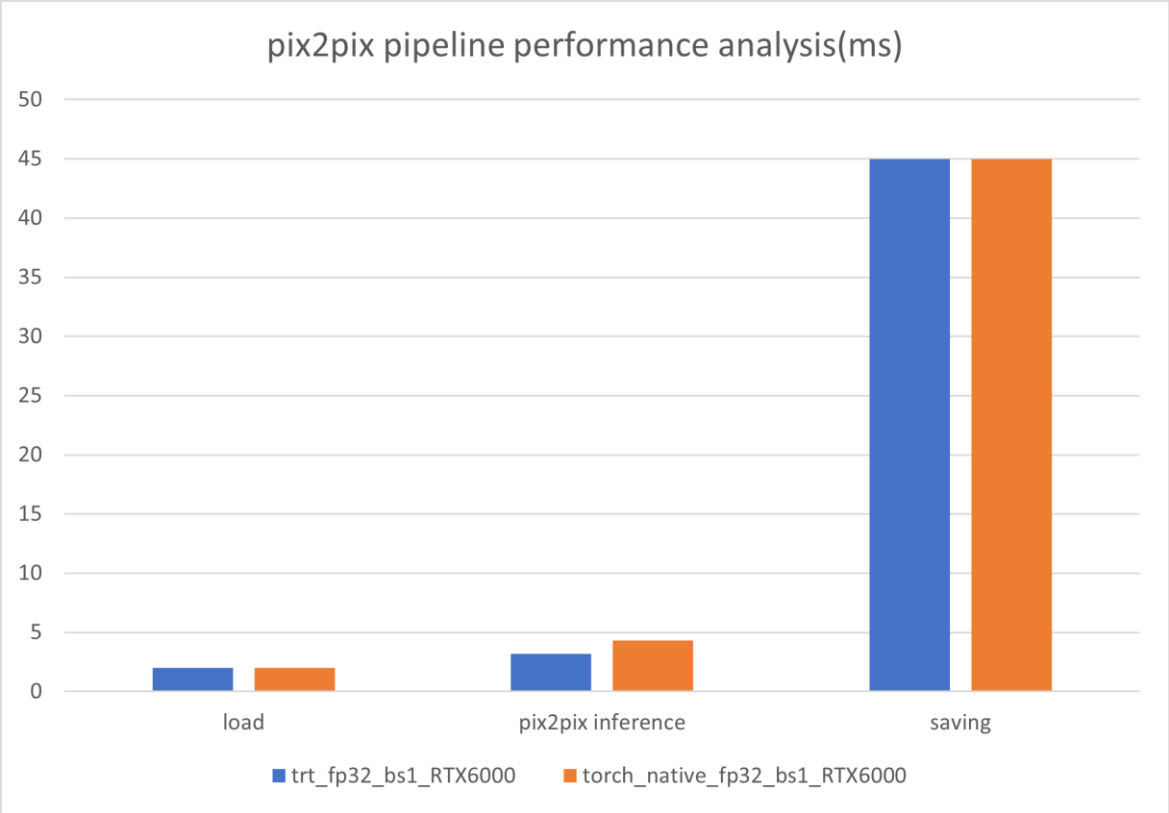
1.1.2 Inference parallel with saving images

We changed source code a little to check how much performance we can get from hiding “save” stage in “inference” stage:

If just remove “save” stage, the “inference_wo_saving” just consumes 260ms for 50 images, this is our performance target. “inference_w_saving” consumes 2889ms as saving to CPU is of low performance.
To hide this low performance “saving” stage, we can use two CUDA streams, one for inference, and one for saving, a ping-pang buffer is used between them to copy output image of “inference” stage to “saving” stage.
After we use two streams (saving is executed at stream 16), we find save is blocked by CPU operators:



As each save only has 3 small D2H memcpy and all others are CPU operators. CUDA streams parallel has no benefit to this situation. To move saving stage at the end of pipeline is a better solution.
To here, we get some basic profiling information:



"saving" stage consumes too much time in one inference pipeline, "load" stage consumes ~2ms, pix2pix inference stage consumes ~4ms, TRT inference is a little faster than torch native implementation.

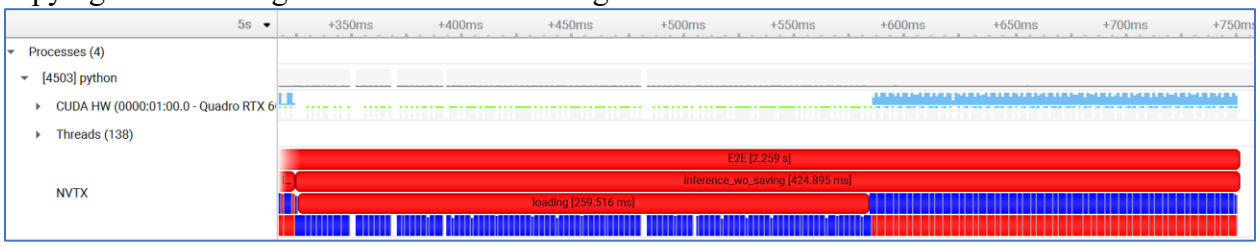
1.1.3 Increase batchsize

TRT pix2pix inference not fully occupies GPU, increasing batch size is straightforward.

Bs=1, with no saving stage, the gap in timeline is at "loading" stage, and GPU utility at TRT inference stage is not as high as expected.

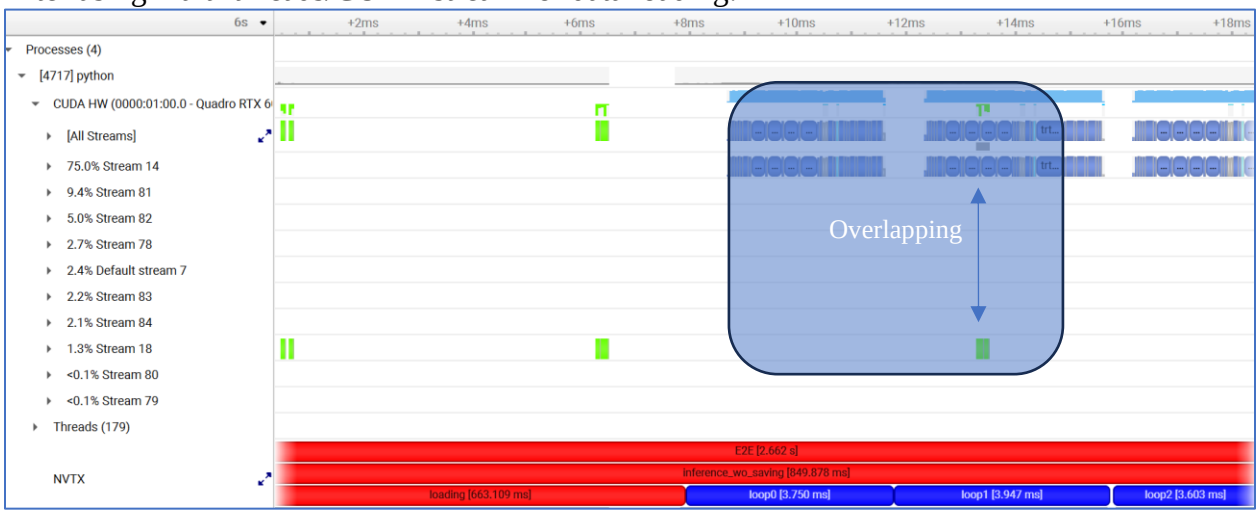
1.2 Multi-thread for data loading but performance is getting bad

We tried threading.Thread package to move data loading into CPU threads, and copy data to GPU before inference begins, this figure show “loading” stage consumes more than half of the time, and only H2D operators is executed on GPU. We use another CUDA stream for data copying task. The target is to hide this “loading” time.



Before using multi-threads/CUDA stream for data loading:

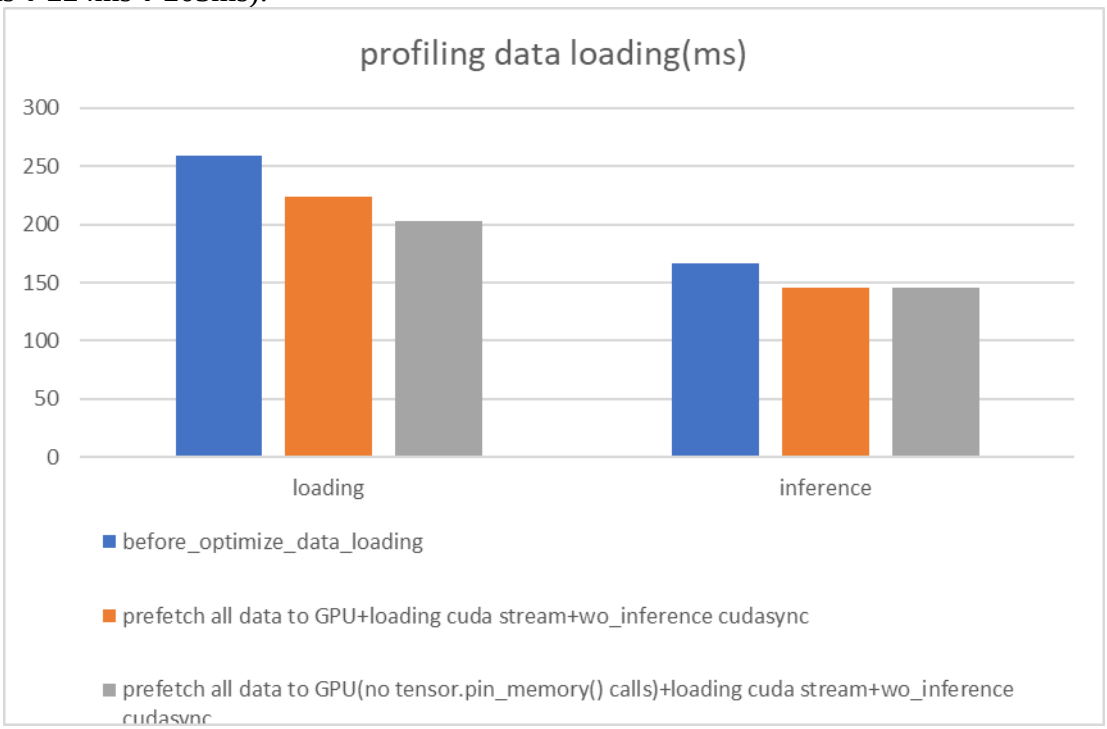
After using multi-threads/CUDA stream for data loading:



We can see there is an overlapping in timeline between data loading and inferencing. But E2E time is all getting longer(CPU multi-thread make performance worse), it is too bad. So we finally removed multi-thread data loading feature.

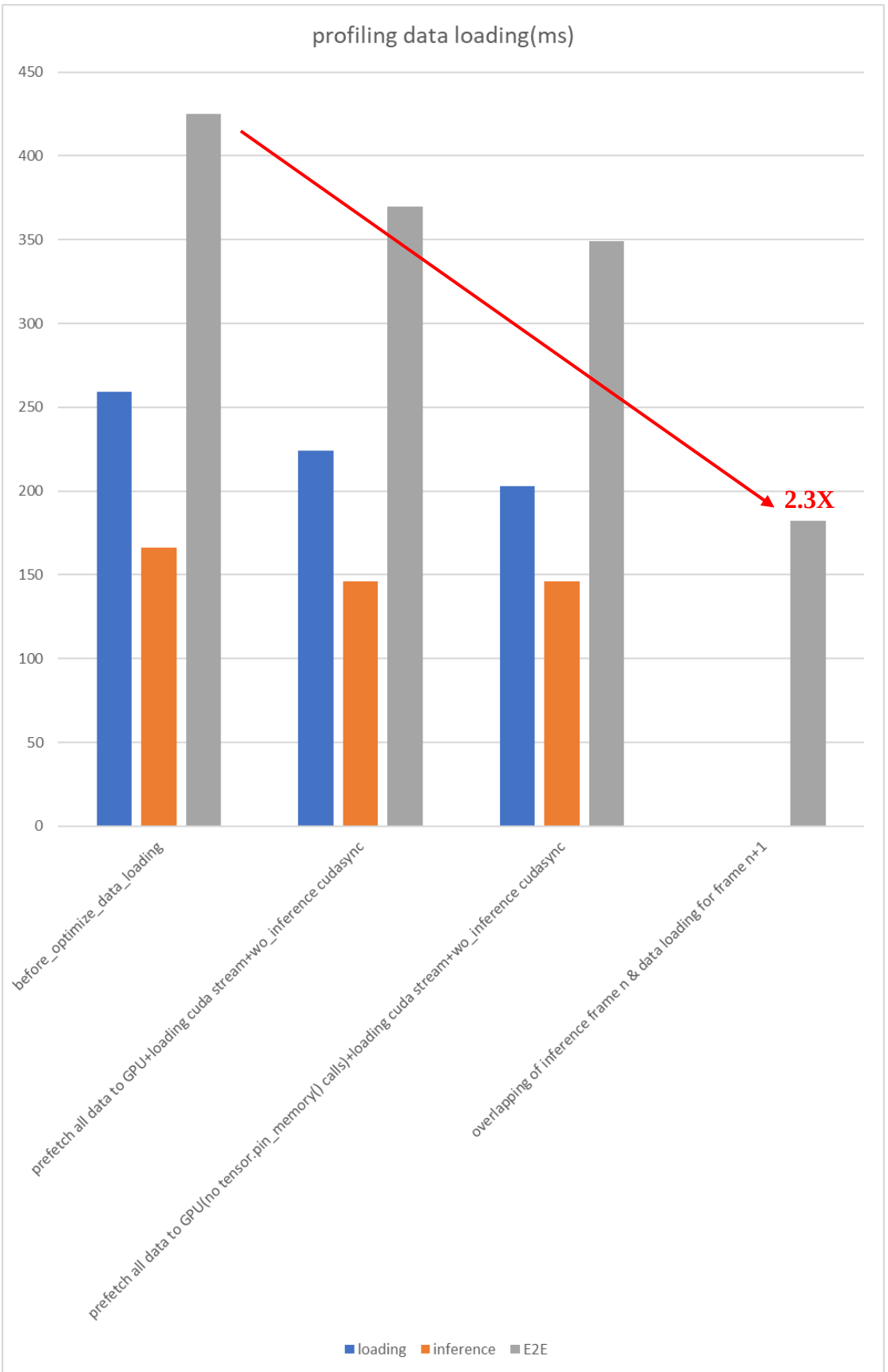
As input images are now already in GPU, inference can go for all inputs without cudaSynchronize at each end of pix2pix model run, the inference performance is getting better from 166ms→146ms:

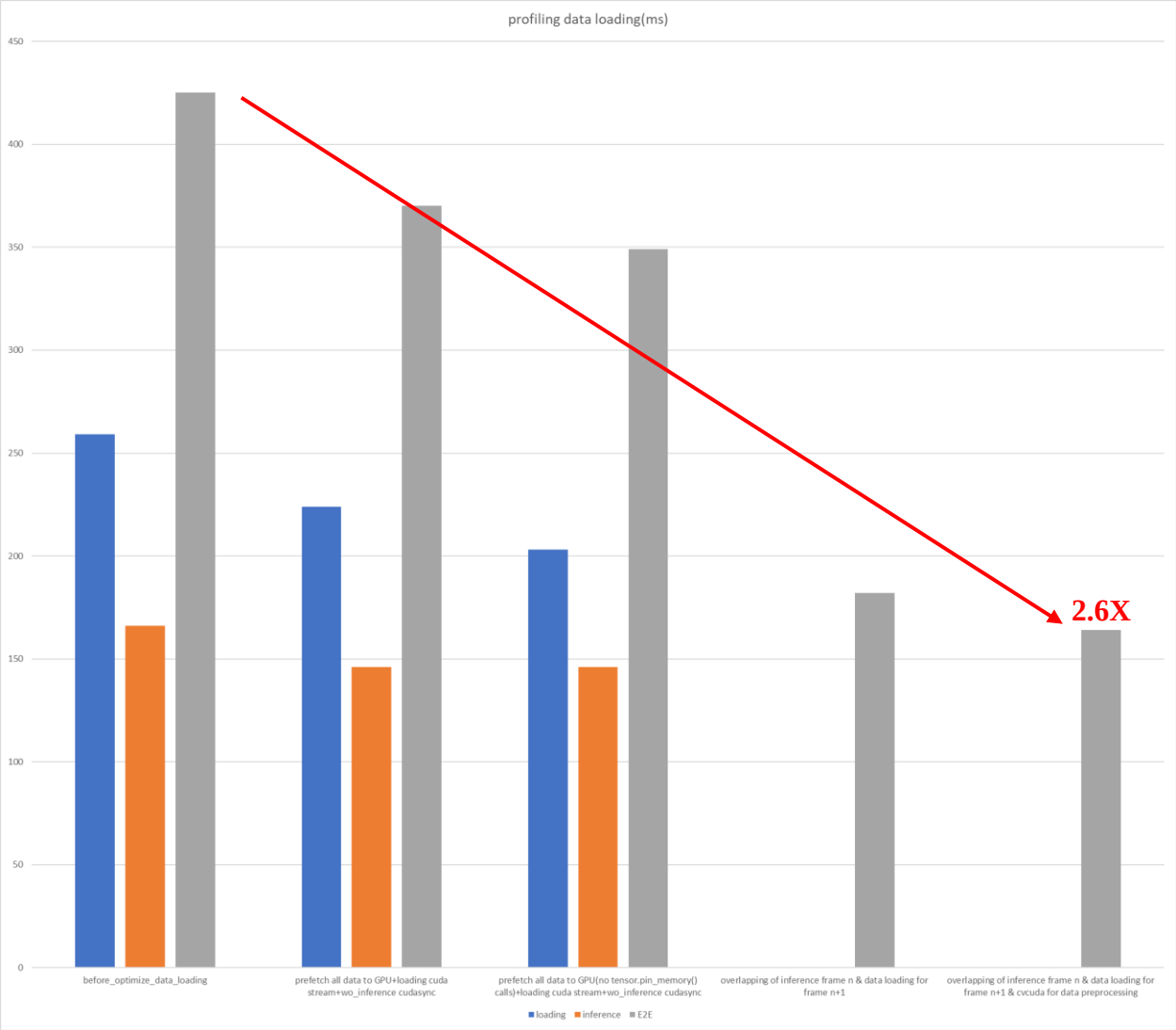
Not using pinned_memory, the performance is getting better for data loading stage(from 259ms→224ms→203ms):



Which means explicity pin_memory calling is not as good as implicit pin_memory calling, for this case.

We find multi-thread data loading has no performance gain, but overlapping of inference for image n and data loading for image n+1makes E2E pipeline performance from 425ms->182ms, makes 2.3X speed up.


profiling data loading(ms)
450
400
350
300
250
200
150
100
50
0
2.3X
before_optimize_data_loading
prefetch all data to GPU+loading cuda stream+wo_inference cudasync
prefetch all data to GPU(no tensor.pin_memory() calls)+loading cuda stream+wo_inference cudasync
overlapping of inference frame n & data loading for frame n+1
loading
inference
E2E



And using cv-cuda for image pre-processing and using another cuda stream to make data loading stage overlapping computing onto GPU, makes the performance speed up from 425ms->182ms→164ms with 2.6X speed up for 50 images inferencing.

The source code looks like this (but not used in final version):

```
with NVTXUtil("inference_wo_saving", "red", mm), SynchronizeUtil(torchutil.torch_stream):
            import queue
            q = queue.Queue()# loaded data are put in this queue
            import threading


            def loading_one_data(dataset, q, idx, device, load_stream): # each cpu thread loads one data and put in queue, and move data to GPU
                with NVTXUtil(f"load{idx}", "blue"), torch.cuda.stream(load_stream):
                    with NVTXUtil(f"CPU", "blue"):
                        data = dataset.__getitem__(idx)
                    data["A"] = torch.unsqueeze(data["A"], dim=0).to(device, non_blocking=True) # non_blocking is necessary
```

```
                data["B"] = torch.unsqueeze(data["B"], dim=0).to(device, non_blocking=True)
                q.put(data)


        threads = []
        with NVTXUtil(f"loading", "red"):
            for idx in range(dataset.__len__()): # start threads to load data
                t = threading.Thread( target=loading_one_data, args=(dataset, q, idx, model.device, load_stream) )
                threads.append(t)
                t.start()


        for idx in range(dataset.__len__()): # get data from queue and perform AI inference
            i = idx
            threads[i].join()
            data = q.get()
            with NVTXUtil(f"loop{i}", "blue", mm), SynchronizeUtil(torchutil.torch_stream):
                model.set_input(data)  # unpack data from data loader
```

1.3 CV-CUDA for data loading stage image preprocessing

CV-CUDA can use its own CUDA stream, and this CUDA stream can be the same for torch computing later. Set like this:

```
    cvcuda_stream = cvcuda.Stream()
    load_stream = torch.cuda.ExternalStream(cvcuda_stream.handle)
```

Use like this:

```
with torch.cuda.stream(torchutil.torch_stream), cvcuda_stream:
    torch ops
    cvcuda ops
```

You can also set CUDA stream will calling cvcuda APIs like this:

```
tensor1 = cvcuda.reformat(tensor0, "NHWC", stream=cvcuda_stream)
```

There is one API in cvcuda can't be set CUDA stream to be used, cvcuda.as_tensor, which convert torch tensor to cvcuda for later image processing. And solving by using this script:

```
with torch.cuda.stream(torchutil.torch_stream), cvcuda_stream:
    tensor1 = cvcuda.as_tensor(tensor0, "NCHW")
```

I modified image preprocessing to cvcuda as shown here:

```
        w, h = AB.size
        w2 = int(w / 2)
        AB = transform(AB).unsqueeze(dim=0).to(device="cuda:0")
        ccAB = cvcuda.as_tensor(AB, "NCHW")
        ccAB = cvcuda.reformat(ccAB, "NHWC", stream=cvcuda_stream)
        rectA = cvcuda.RectI(x=0 , y=0, width=w2,    height=h)
        rectB = cvcuda.RectI(x=w2, y=0, width=w-w2 , height=h)
        cropA = cvcuda.customcrop( ccAB, rectA, stream=cvcuda_stream )
        cropB = cvcuda.customcrop( ccAB, rectB, stream=cvcuda_stream )
        normalizedA = cropA
```

```
            normalizedB = cropB
            # Normalize with mean and std-dev.
            normalizedA = cvcuda.normalize(
                normalizedA,
                base= mean_tensor,
                scale= stddev_tensor,
                flags=cvcuda.NormalizeFlags.SCALE_IS_STDDEV,
                stream=cvcuda_stream
            )
            normalizedA = cvcuda.reformat(normalizedA, "NCHW", stream=cvcuda_stream)

            normalizedB = cvcuda.normalize(
                normalizedB,
                base= mean_tensor,
                scale= stddev_tensor,
                flags=cvcuda.NormalizeFlags.SCALE_IS_STDDEV,
                stream=cvcuda_stream
            )
            normalizedB = cvcuda.reformat(normalizedB, "NCHW", stream=cvcuda_stream)

            if isinstance(normalizedA, torch.Tensor):
                if not normalizedA.is_cuda:
                    normalizedA = normalizedA.to("cuda:0")
            else:
                # Convert CVCUDA tensor to Torch tensor.
                normalizedA = torch.as_tensor( normalizedA.cuda(), device="cuda:0" )

            if isinstance(normalizedB, torch.Tensor):
                if not normalizedB.is_cuda:
                    normalizedB = normalizedB.to("cuda:0")
            else:
                # Convert CVCUDA tensor to Torch tensor.
                normalizedB = torch.as_tensor( normalizedB.cuda(), device="cuda:0" )

            return {'A': normalizedA, 'B': normalizedB, 'A_paths': AB_path, 'B_paths': AB_path}
```

For further optimization, you can try to fuse cvcuda operators into one, cat image A/B into one cvcuda tensor with batchsize=2.

Why I prefer to use cvcuda is that it is of high performance to handle these simple image processing operations. Another reason is it can hide itself in inferencing by using another CUDA stream, so data loading stage occupies less space in Nsight system timeline.