CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Project Final Assignment — Solution

Total number of points: 100.

Most of the code has been written during previous homework assignments. In this assignment, you will focus on analyzing and discussing the performance of the code. This assignment is more open-ended than the previous ones, and you will receive less guidance. You are responsible for submitting a PDF to Gradescope. Although you may want to develop some new code to test new ideas, code submission is not required for this assignment.

When including figures, tables, etc., to make a point, ensure that each figure/table is useful. They should be referenced in the text and have a caption. The caption should be informative and self-contained. Figures should be easy to understand without reading the text. The same applies to tables and code snippets. You should comment on each figure and table in the text. The code snippets should be short and relevant. They should be referenced in the text and commented on. Avoid including material that is not commented on and is simply included for archival purposes.

Warning: Because this homework requires using the cluster for longer periods of time, please be respectful of other students and cluster usage. Make sure to use `squeue` to check the status of your jobs. Use `scancel [jobid]` to kill jobs that run for too long. Do not submit more than 1 script or job at a time.

Starter code information

In the starter code we provide, we have added several `MPI_Barrier`, `cudaDeviceSynchronize`, and `nvtxRangePushA` and `nvtxRangePop` calls to help profile the code using Nsight Systems. Please refer to the online documentation for information about NVTX. For this homework, you will need to install Nsight Systems on your local machine by following these instructions.

In summary, `nvtxRangePushA` will start a new range in the timeline, and `nvtxRangePop` ends the range. You can use these calls to profile the code and understand where time is spent. They will create ranges in the profiler, as shown in Figure 1. This is very useful for facilitating the analysis of the code and better understanding the profiler output.

`MPI_Barrier` is not required for correctness and will affect performance to some extent. But it helps simplify and clarify the profiling. For example, if you call `MPI_Allreduce` it may look like some processes take a lot of time to complete that command. In reality, because `MPI_Allreduce` is a synchronous collective operation, the process is mostly waiting for the other processes to complete. By adding a barrier, you can see the time spent in the `MPI_Allreduce` operation more clearly.

`cudaDeviceSynchronize` is used to synchronize the CPU with the GPU. Without this, the NVTX information may be inaccurate.

The starter code contains the tests that were previously used in Homework 6. They are contained in the `gtestTrain` test suite. At the end, you will find a different test: `gtestProfile`. This test is a little different. The code is similar to the one in Homework 6 except that we additionally load a test set for the MNIST data set. This test set is used to evaluate the precision of your DNN model.

Recall that precision is defined as:
precision = total number of labels correctly predicted / total number of labels

You can adjust the size of the hidden layer (`hidden_size`) and the number of epochs in order to train your model.

You will see the precision only if `test_size` is non-zero. See `main.cpp` for more details.

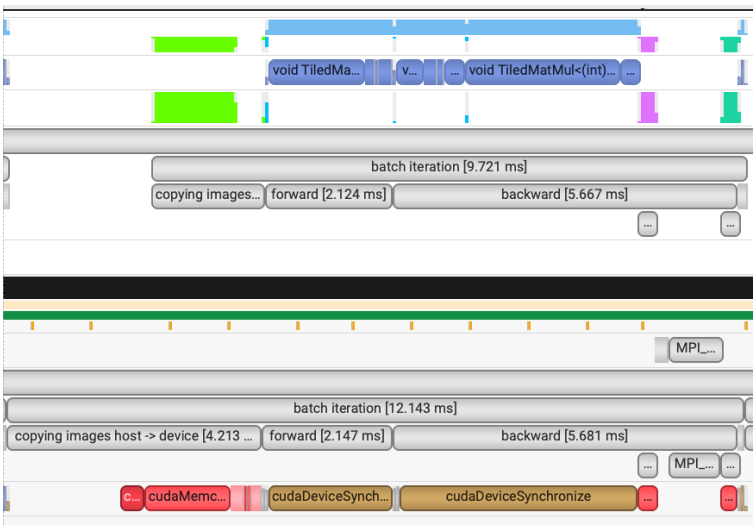


Figure 1: Sample Nsight Systems screenshot

Problem 1 Convergence of your model

Question 1.1

(5 points) First test the code to make sure that it is correct. Run:

```
make clean && srun make -j && salloc -n 4 -p gpu-turing mpirun ./main
```

Run your code again in double precision:

```
make clean && CXXFLAGS=-DUSE_DOUBLE srun make -j && salloc -n 4 -p gpu-turing mpirun ./main
```

Please report on the output. Does your code pass all the tests?

Focus now on the test

```
TEST(gtestProfile, nsys)
```

You may run this test only using the command:

```
salloc -n 4 -p gpu-turing mpirun ./main --gtest_filter=gtestProfile.*
```

You can use a similar-looking command in a batch script as well. Note that you can also get rid of the color in the google unit test output using the command:

```
salloc -n 4 -p gpu-turing mpirun ./main --gtest_filter=gtestProfile.* --gtest_color=no
```

This may clean up your output in some cases.

Solution

The code passes all tests in both precisions. The relative error in the weights is around 10^{-7} in single precision and around 10^{-16} in double precision.

Question 1.2

(10 points) Make changes to this test. In particular, vary:

```
hidden_size, num_epochs, learning_rate
```

Does your code report errors? This means that the weights produced by the GPU code are different from the weights produced by the CPU code. Can you explain this discrepancy? Does it matter? What precision do you get using the CPU and GPU codes?

Solution

For a large enough learning rate, the code reports errors. Specifically, the relative errors in weights between the CPU and GPU implementations are around 10^{-2}, and this causes the corresponding tests to fail. This discrepancy is due to the fact that the neural network training procedure is highly chaotic—small perturbations in weights and gradients grow exponentially with iterations. One source of the perturbations is the fact that the GPU implementation accumulates gradients in a different summation order than the CPU implementation. Usage of the fused multiply-add instructions on the GPU also leads to small errors that can accumulate over time.

However, this discrepancy is not reflected in the precision of the model, so it does not really matter. The precision of the model is around 0.97 for both the CPU and GPU implementations. This is because the gradient computed at each iteration is still accurate enough to guide the optimization procedure to a good local minimum in both implementations.

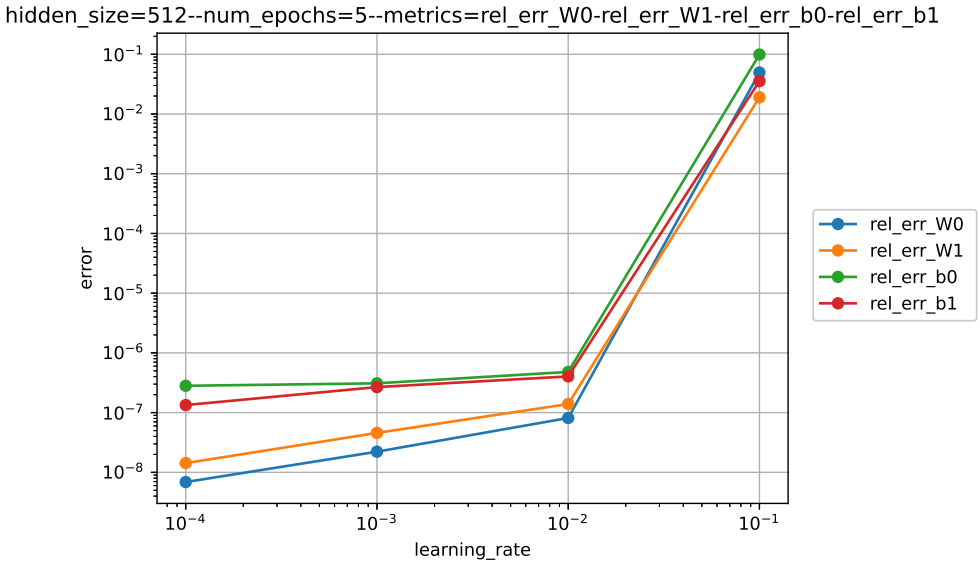


Figure 2: Discrepancy in weights between CPU and GPU implementations.

Question 1.3

(10 points) What learning rate seems to lead to the fastest convergence? What happens when you increase or decrease this learning rate?

Solution

The learning rate that seems to lead to the fastest convergence is around 10^{-1}. When the learning rate is too small, the model converges very slowly. When the learning rate is too large, the model does not converge to a good local minimum, and the final precision is much worse.

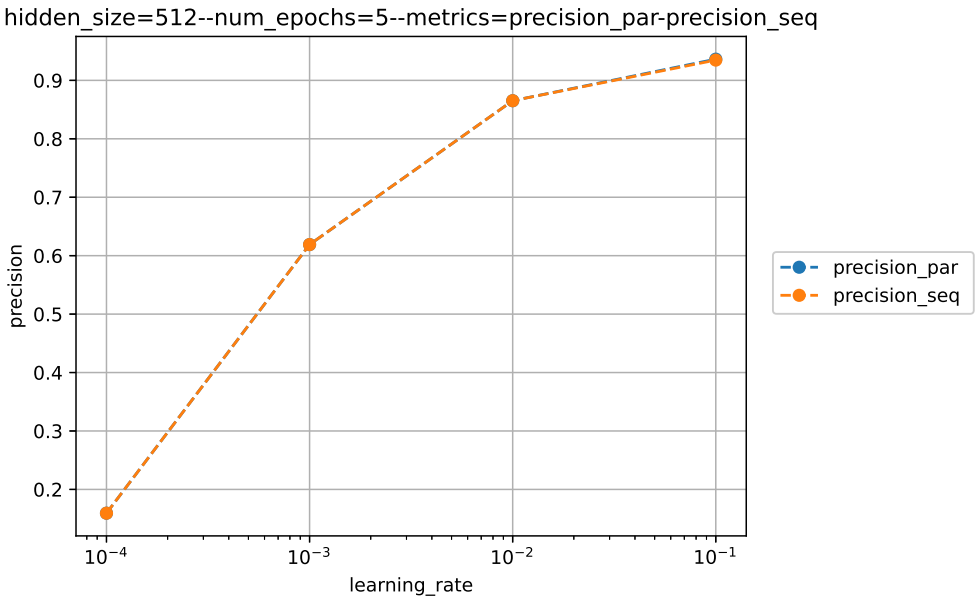


Figure 3: Both CPU and GPU implementations achieve a precision of around 0.97.

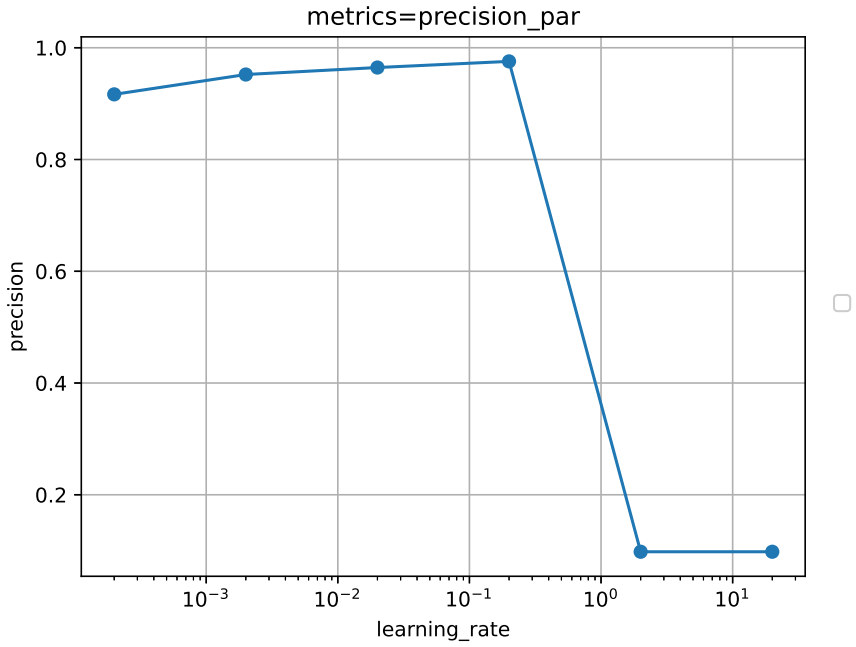


Figure 4: Effect of varying learning rate.

Question 1.4

> (10 points) Train your model for many epochs. What loss are you able to achieve? Has your model converged? What is the precision of your model? Does the precision depend on `hidden_size`? What value of `hidden_size` would you recommend? What happens when you reduce or increase it?

For this question, you may want to set debug=1 in the code to print the loss at each epoch and monitor the convergence. debug is the last parameter in

```
hyperparameters{60000, 10000, 784, 64, 3200, 1, 10, 1e-4, 0.2, 1};
```

You can change the frequency of the loss printouts by changing `printout_frequency` in this line:

```
const int printout_frequency = 4;
const int printout_period = std::max(1, (hparams.num_epochs * num_batches) / printout_frequency);
print_flag = (hparams.debug == 1) && ((iter % printout_period) == 0);
```

Solution

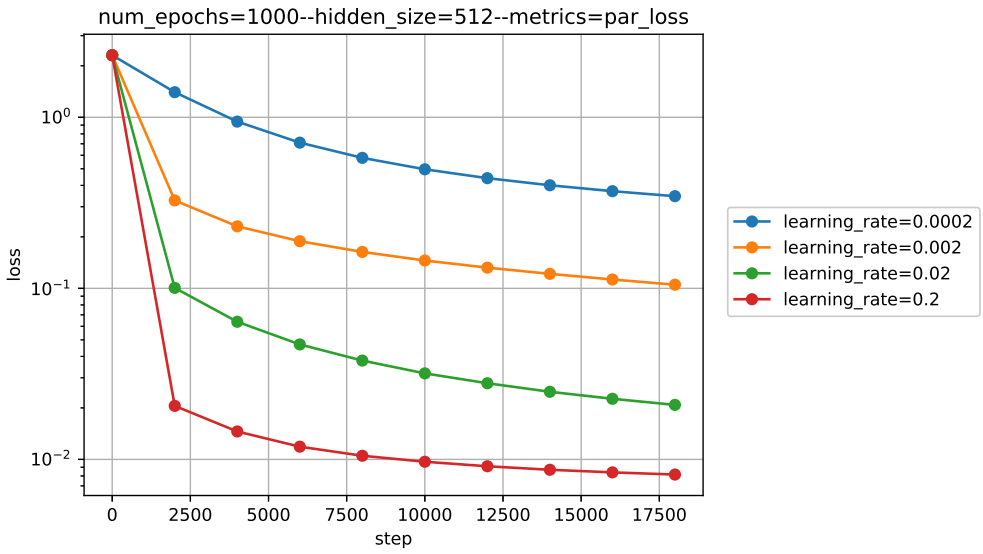


Figure 5: Convergence of the loss.

The best loss achieved is around 10^{-2}. The model converges after around 1000 epochs with a learning rate of 0.2. The precision of the model is around 0.97. The precision does vary with `hidden_size`, but saturates at around 0.97 as it is increased. When you reduce `hidden_size`, the model converges more slowly. When you increase `hidden_size`, the model converges faster, but the gains in precision are marginal, and even reduce if it gets too large due to overfitting.

Problem 2 Performance Analysis

To analyze the performance, we recommend using Nsight Systems. You can use the following command to profile the code:

```
sbatch nsys.sh
```

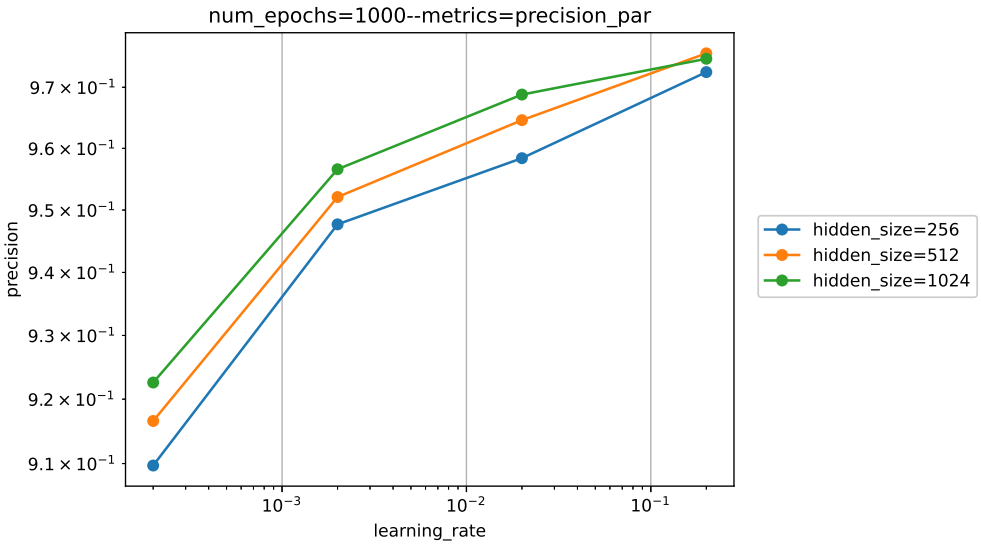
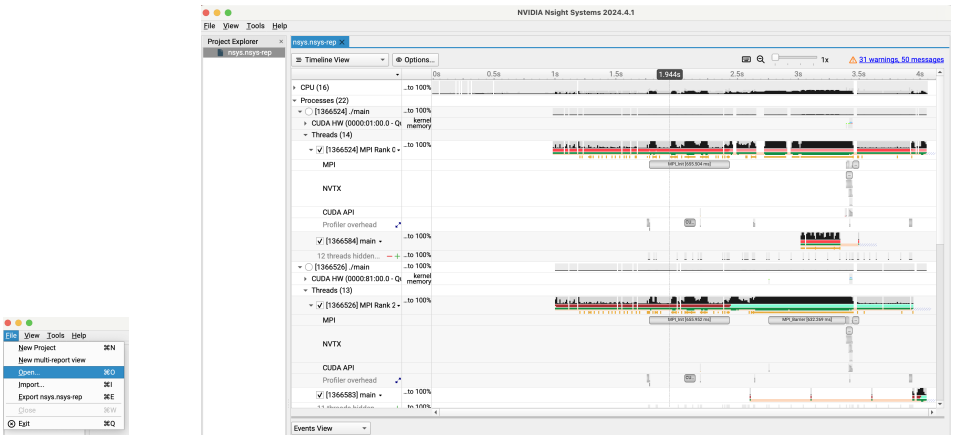


Figure 6: Varying hidden size.

Note that `nsys` and `mpirun` do not work properly from the command line. You have to use a batch submission. This command will produce the usual `slurm*.out` file along with a file `report*.nsys-rep`. Copy the `report*.nsys-rep` file to your local machine.

Load this file into Nsight Systems on your local machine. These screenshots may be useful to illustrate what you should see:



You will see the labels from NVTX that will guide your analysis.

Question 2.1

> (10 points) In your profile, what are the steps that take the most time? Is this consistent with the analysis from Homework 6? Does this match your expectations or are there surprises?

Solution

The profile was generated by running `sbatch nsys.sh` with `--gtest_filter=gtestProfile.nsys1`. I.e., the profile was generated by running the function `profileTrain` in `main.cpp` using 2 MPI processes and the following hyperparameters:

```
// hyperparameters:
//   int N;
//   int test_size = 0;
//   int input_size;
//   int hidden_size;
//   int batch_size;
//   int num_epochs;
//   int num_classes;
//   nn_real reg;
//   nn_real learning_rate;
//   int debug;

hyperparameters{60000, 10000, 784, 512, 3000, 1, 10, 1e-4, 1e-2,
0};
```

The analysis below is specific to these hyperparameters.

The vast majority of the overall runtime is used to read the MNIST images into armadillo matrices and train on the CPU. These two steps roughly correspond to the 2.721s `MPI_Barrier` event in Figure 7. Other things which take a significant proportion of the overall runtime are `MPI_Init`, other initial setup steps, verifying correctness of the GPU implementation, and `MPI_Finalize`. It is perhaps suprising that training on the GPU takes such a small fraction of the overall runtime.

In parallel training on the GPU, the setup (which mostly consists of the rank 0 MPI process broadcasting MNIST images to the rank 1 MPI process) takes around 22% of the total time. See Figure 8.

In a typical batch iteration, the majority of the time (around 51%) is spent copying the current batch of images from host to device. The forward pass takes only around 12% of the time, and the backward pass takes the remaining 37% of the time. See Figure 9. This matches our expectation from Homework 6 that `cudaMemcpy` operations are a bottleneck in our application.

Zooming in further, in Figure 10 we can see that most of the time in the forward and backward passes is spent doing a single large matrix multiplication. This is consistent with the analysis from Homework 6. With the hyperparameters above, however, we can see that the second matrix multiplcation in the forward pass and the first matrix multiplication in the backward pass also take a non-negligible amount of time; it was perhaps an oversimplification to ignore these in the analysis from Homework 6. Finally, we can see that just under half the time in the backward pass is spent reducing gradients across the two MPI processes. Only around 20% of this time is spent in `cudaMemcpy` operations and the other 80% is spent in the `MPI_Allreduce` operation; it was an oversimplification to ignore the time spent in `MPI_Allreduce` operations in the analysis from Homework 6.

Question 2.2

> (15 points) Can you identify areas of the code that could be improved based on the profiling? What changes would you suggest?

Examples of possible changes include:

- Changing the way memory is allocated and managed.
- Changing the pattern of MPI communications and using non-blocking MPI calls.
- Using CPU pinned memory. This can be done using `cudaMallocHost`. See this blog post for more information.

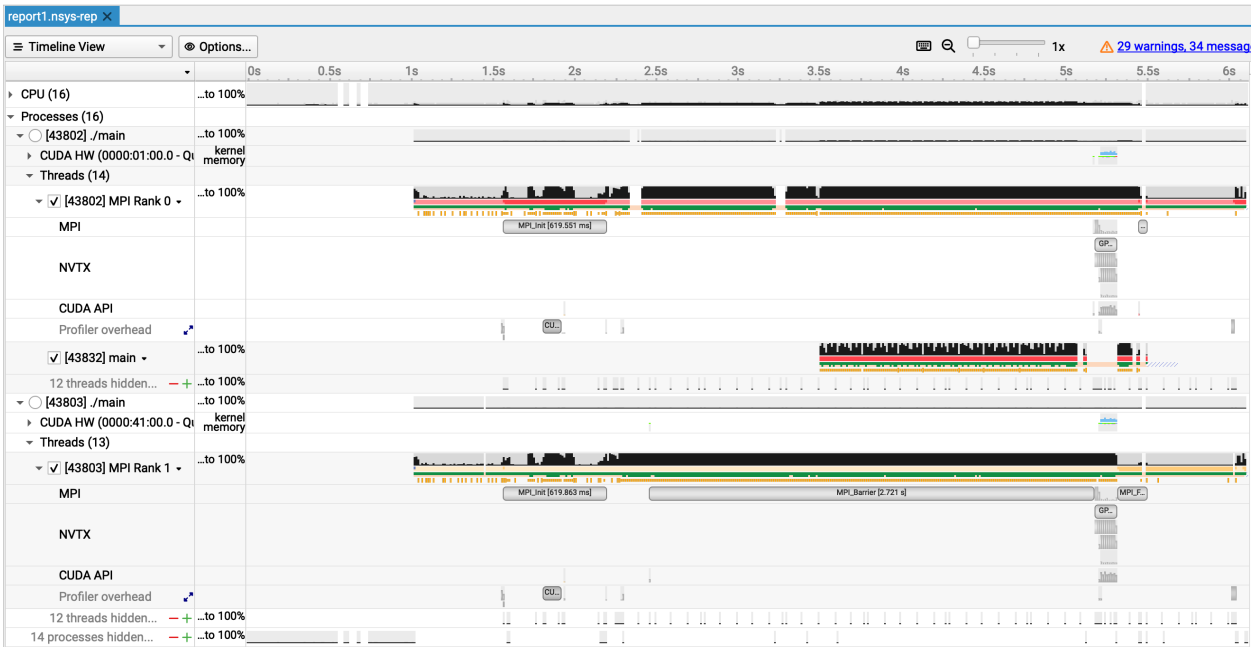


Figure 7: `report1-nsys.rep`, zoomed all the way out.

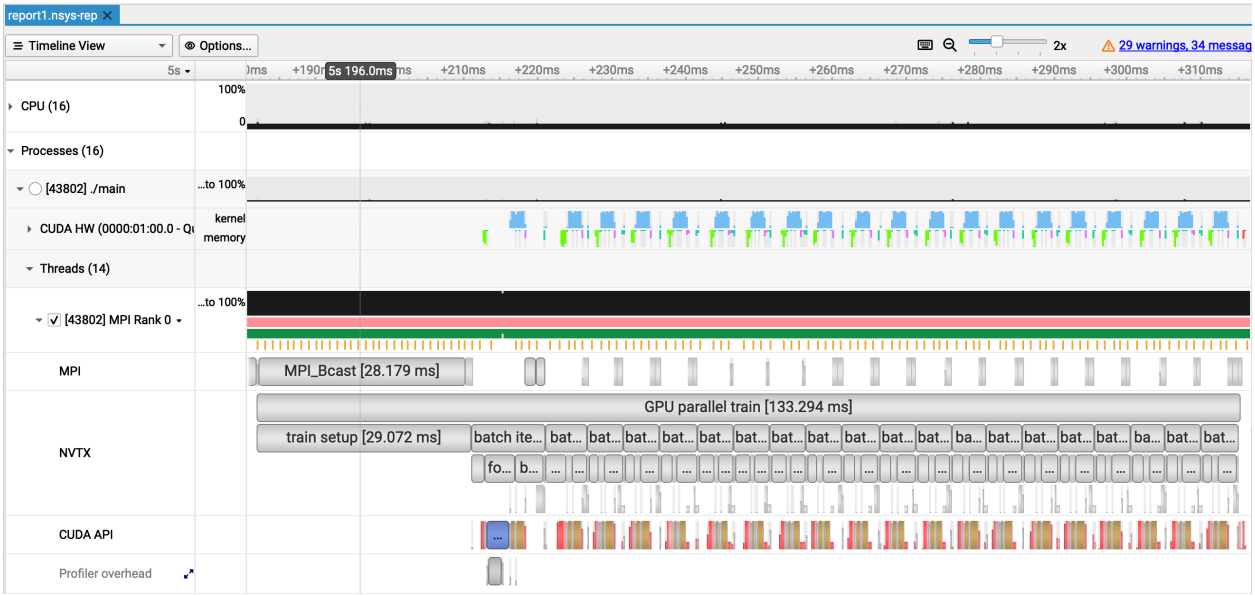


Figure 8: `report1-nsys.rep`, zoomed in to GPU parallel train.

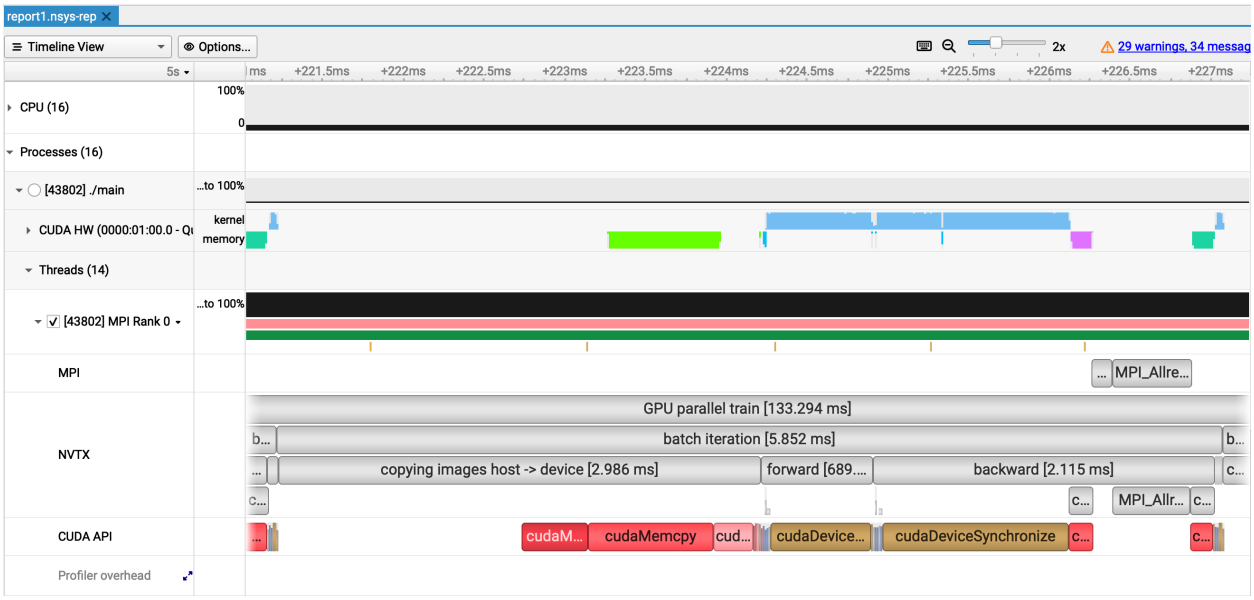


Figure 9: `report1-nsys.rep`, zoomed in to the second batch iteration.

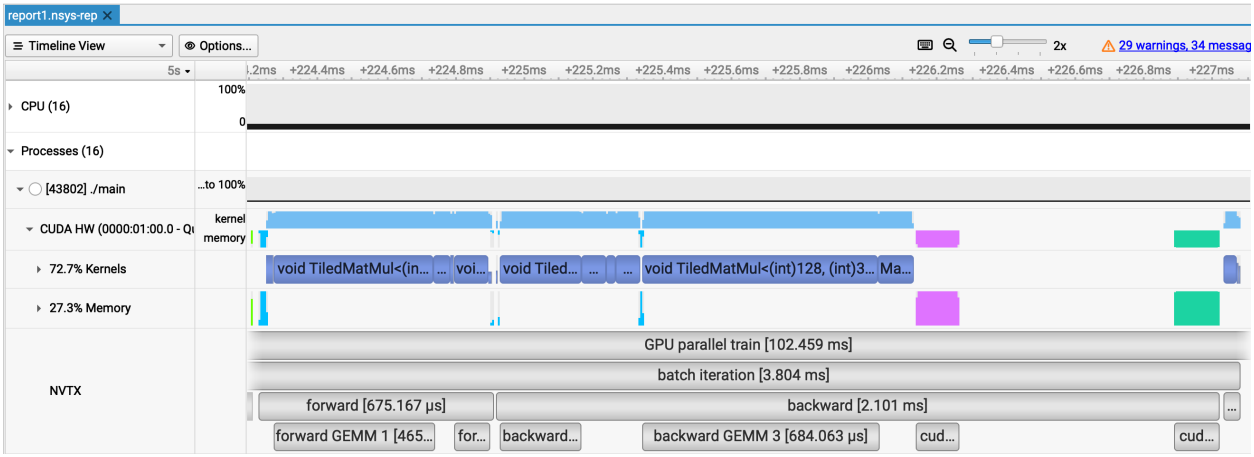


Figure 10: `report1-nsys.rep`, zoomed in to the forward and backward passes. Note that we have zoomed into the same batch as in Figure 9, but the timings for the CUDA row group here are slightly different from the Threads row group in Figure 9. See `https://forums.developer.nvidia.com/t/nvtx-ranges-in-threads-and-in-cuda-do-not-align-with-each-other/123325`.

- Using GPU memory pointers in MPI calls. See this blog post for more information.
- Pipelining various MPI and CUDA memcpy operations.
- Using CUDA streams. See this NVIDIA presentation for more information.
- Optimizing the CUDA kernels.

These are examples. You do not need to discuss all of them. You can focus on the ones that you think are most important.

We would like you to come up with an approximate model of the performance and arithmetic intensity of the code, similar to Homework 6, with a roofline model.

In the discussion below, all the numbers are normalized per GPU and per batch. We assume single precision. For simplicity, we only consider the case of two GPUs. This makes it easier to model the running time of MPI collective operations.

Question 2.3

(10 points) Using your profiling information, list the CUDA kernels and calculations that take a substantial amount of runtime during training. Then for these kernels, estimate the number of flops.

For memory transfers, focus on the cudaMemcpy and MPI operations. These are the slowest memory transfer operations. Estimate the number of bytes transferred.

For `MPI_Allreduce`, you can make the following assumptions. Assume we want to reduce b bytes. The reduction on 2 GPUs requires communicating b bytes. Then the result needs to be sent to the other process, requiring another b bytes. This gives a total of $2b$ bytes communicated.

You may ignore operations that are independent of the epoch, including: copying image data from one process to other processes, and copying image data from host to device memory.

Please write your estimates in terms of these variables (following Homework 6 notations):

```
D = input_size,
H = hidden_size,
C = num_classes,
B = batch_size.
```

Solution

- With the hyperparameters in the test `gtestProfile.nsys1`, the CUDA kernels that take a substantial amount of runtime when training are both `TiledMatMuls` in `forward` and the first and last `TiledMatMuls` in `backward`.

  Let p be the number of MPI processes, and let $B' = B/p$. Approximately, both `TiledMatMuls` in `forward` require $2DHB' + 2HCB' = 2B'H(D + C)$ flops, and the two largest `TiledMatMuls` in `backward` require $2B'HD + 2B'CH = 2B'H(D + C)$ FLOPs.

- Let $b = 4$ be the number of bytes required to store an `nn_real` (`report1-nsys.rep` is a profile of training in single-precision), and let N be the number of images in the training set. We ignore calls of `DeviceMatrix::to_gpu` since they are device to device `cudaMemcpys` and are much faster than host to device or device to host `cudaMemcpys`.

  Per batch, the number of bytes transferred by `MPI_Allreduce` is $2(HD + H + CH + C)b$.

  Per batch, the number of bytes transferred by `cudaMemcpy` when copying gradients between device and host in `backward` is $2(HD + H + CH + C)b$.

Question 2.4

> (10 points) Based on your profiling measurements, provide an estimate for the effective compute throughput P_{flops}, and the `cudaMemcpy` (P_{pci}) and `MPI_Allreduce` (P_{mpi}) memory bandwidth throughputs.

For example, let's say you want to multiply two square matrices of size n. The total number of flops is $2n^3$. If you measure a running time of t seconds, the effective compute throughput is $2n^3/t$ flops/second. Similarly, the memory transfer for a square matrix is $4n^2$ (4 bytes per float) and the effective memory throughput is $4n^2/t$ bytes/second.

This estimate will be more useful than the peak performance listed by the manufacturer.

Similar to the previous question, **you may ignore** `copying images host -> device` and `train setup` since these are independent of the epoch.

Solution

We have

$$P_{\text{flops}} = \frac{\text{FLOPs}}{t_{\text{FLOPs}}} = \frac{2B'H(D+C) + 2B'H(D+C)}{t_{\text{forward GEMM 1}} + t_{\text{forward GEMM 2}} + t_{\text{backward GEMM 1}} + t_{\text{backward GEMM 3}}};$$

$$P_{\text{pci}} = \frac{\text{bytes\_PCIe}}{t_{\text{PCIe}}} = \frac{2(HD + H + CH + C)b}{t_{\text{cudaMemcpyDeviceToHost}} + t_{\text{cudaMemcpyHostToDevice}}};$$

$$P_{\text{mpi}} = \frac{\text{bytes\_MPI}}{t_{\text{MPI}}} = \frac{2(HD + H + CH + C)b}{t_{\text{MPI\_Allreduce}}}.$$

Using median times for the relevant kernels and `cudaMemcpys` from the NVTX GPU Projection Summary and using the median time for `MPI_Allreduce` from the NVTX Range Summary in `report1-nsys.rep` (see Figure 11), we get

```
P_flops = 1.632730e+12 FLOPS/sec
P_pci = 1.2720e+10 bytes/sec
P_mpi = 7.9502e+09 bytes/sec
```

See `fp_p2_solution.ipynb` for calculations.

Question 2.5

> (10 points) As in Homework 6, we want to plot FLOPS/sec vs arithmetic intensity. Assume that FLOPS is the total number of flops that you estimated and that Bytes is the total number of bytes transferred. The arithmetic intensity is defined as FLOPS/Bytes. Following the example in Homework 6, provide a theoretical estimate of the roofline model. The y-axis is FLOPS/sec and the x-axis is arithmetic intensity. The answer will involve P_{flops}, P_{pci}, and P_{mpi}. Plot this function in a way similar to Homework 6.

Solution

Instead of plotting

$$\frac{\text{FLOPs}}{\text{sec}} = \frac{\text{AI}}{\frac{\text{AI}}{\text{Peak GPU FLOPs/sec}} + \frac{1}{\text{Peak PCIe bandwidth}}}$$

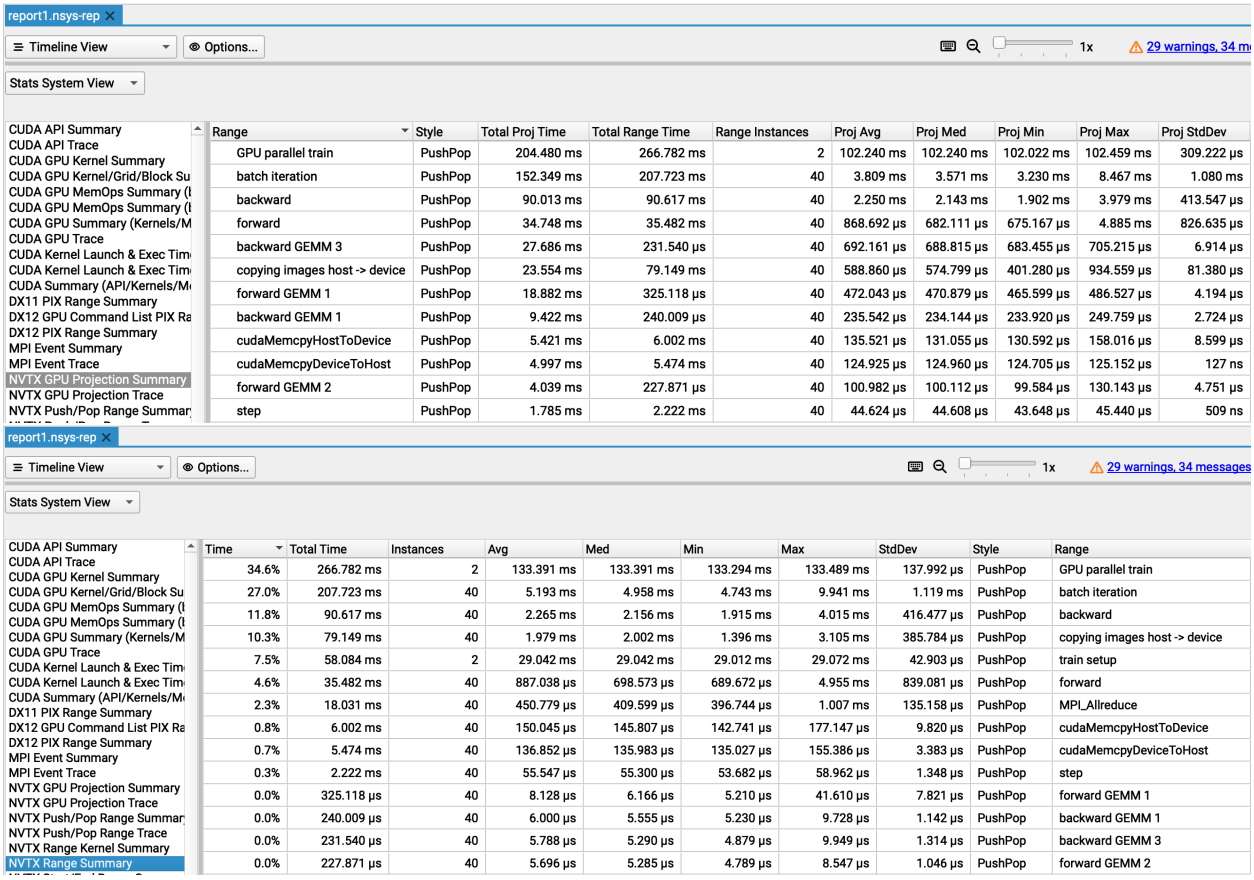


Figure 11: NVTX GPU Projection Summary and NVTX Range Summary for `report1-nsys.rep`, seen in Stats System View in Nsight Systems.

as in Homework 6, in Figure 12 we plot

$$\begin{aligned}
\frac{\text{FLOPs}}{\text{sec}} &= \frac{\text{AI}}{\frac{\text{AI}}{\text{Effective GPU FLOPs/sec}} + \frac{1}{\text{Effective memory bandwidth}}} \\
&= \frac{\text{AI}}{\frac{\text{AI}}{P_{\text{flops}}} + \frac{1}{\frac{\text{bytes_PCIe}+\text{bytes_MPI}}{t_{\text{PCIe}}+t_{\text{MPI}}}}} \\
&= \frac{\text{AI}}{\frac{\text{AI}}{P_{\text{flops}}} + \frac{1}{2}\left(\frac{t_{\text{PCIe}}}{\text{bytes_PCIe}} + \frac{t_{\text{MPI}}}{\text{bytes_MPI}}\right)} \\
&= \frac{\text{AI}}{\frac{\text{AI}}{P_{\text{flops}}} + \frac{1}{2}\left(\frac{1}{P_{\text{pci}}} + \frac{1}{P_{\text{mpi}}}\right)}
\end{aligned}$$

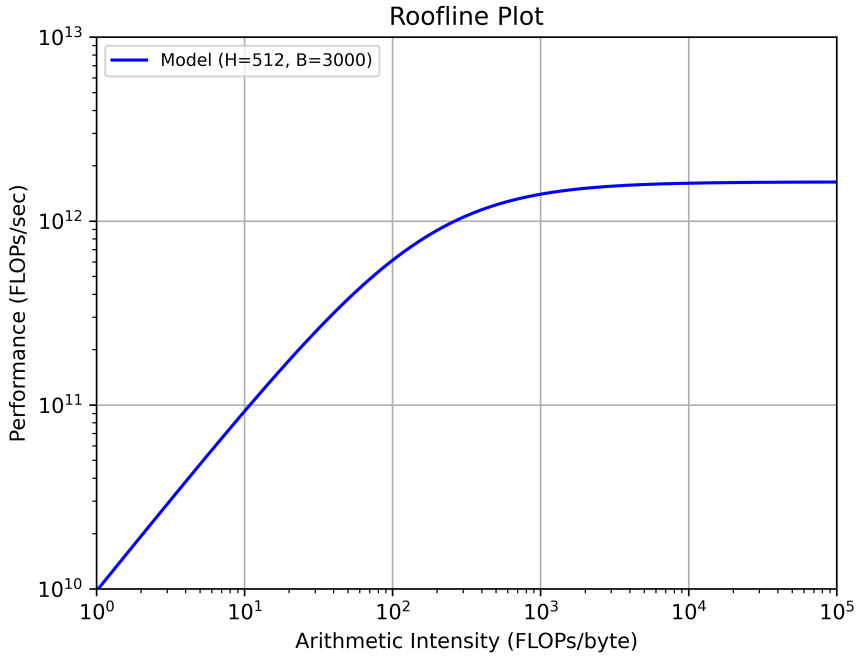


Figure 12: Roofline plot.

Question 2.6

(10 points) Similar to Homework 6, we now want to compare the theoretical estimate with experimental values. For this purpose, calculate the FLOPS/sec. The FLOPS is based on your estimate of the flops performed by the CUDA kernels. The running time should be based on the profiling information.

For the arithmetic intensity, you may use the formula from Question 2.5.

The main difference with Question 2.5 is that you replace your theoretical estimate of the running time with the actual running time.

Run a few experiments with different values of H and B. Plot your experimental values along with the roofline model. Discuss the results. What trends do you observe? What is required to increase the arithmetic intensity and the FLOPS/sec performance? Give your answer in terms of D, H, C, and B. Do

the theoretical and experimental estimates agree? Why/why not? Comment on the difference. Explain how you would improve your theoretical roofline model to better represent the experimental results.

Solution

Increasing H increases performance, but the increase in performance is smaller when H is larger. Increasing H does not increase AI since `FLOPs` is linear in H and bytes_PCIe + bytes_MPI is very close to being linear in H.

Increasing B increases both AI and performance. Increasing B increases AI since in our simplified model, `FLOPs` is linear in B but bytes_PCIe + bytes_MPI is independent of B.

Some experimental estimates fall below the theoretical roofline model but others fall above.

- The blue plus falls below the roofline because we estimated P_{flops}, P_{pci}, and P_{mpi} without considering the time spent in `train setup` or `copying images host -> device`, but we used $t_{\text{batch iteration}}$ to plot this point in Figure 13, and $t_{\text{batch iteration}}$ includes the time spent in `copying images host -> device`. If we used $t_{\text{forward}} + t_{\text{backward}} + t_{\text{step}}$ instead of $t_{\text{batch iteration}}$ to plot each point, we instead get Figure 14.
- Some other points like the green circle fall above the roofline because the roofline is a function of P_{flops}, P_{pci}, and P_{mpi} estimated using the hyperparameters $H = 512, B = 3000$. If we had used the hyperparameters $H = 2048, B = 12000$ to estimate P_{flops}, P_{pci}, and P_{mpi}, the corresponding roofline would have been higher and the green circle would have fallen below the roofline.

See `fp_p2_solution.ipynb` for calculations.

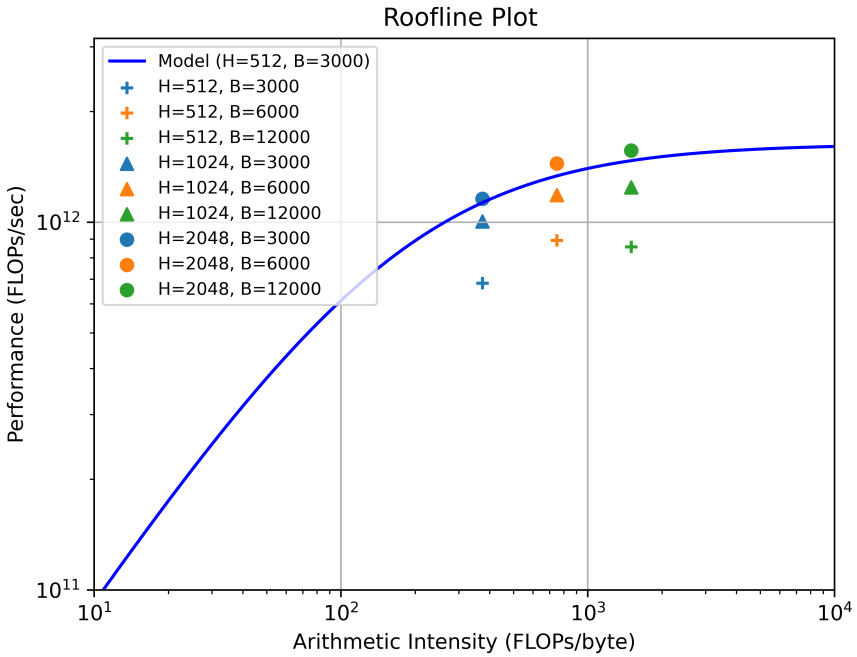


Figure 13: Roofline plot.

3 Optional questions

Depending on your time and interest, you may want to consider the following questions. These will not be graded but they may help you better understand the material and improve your programming skills.

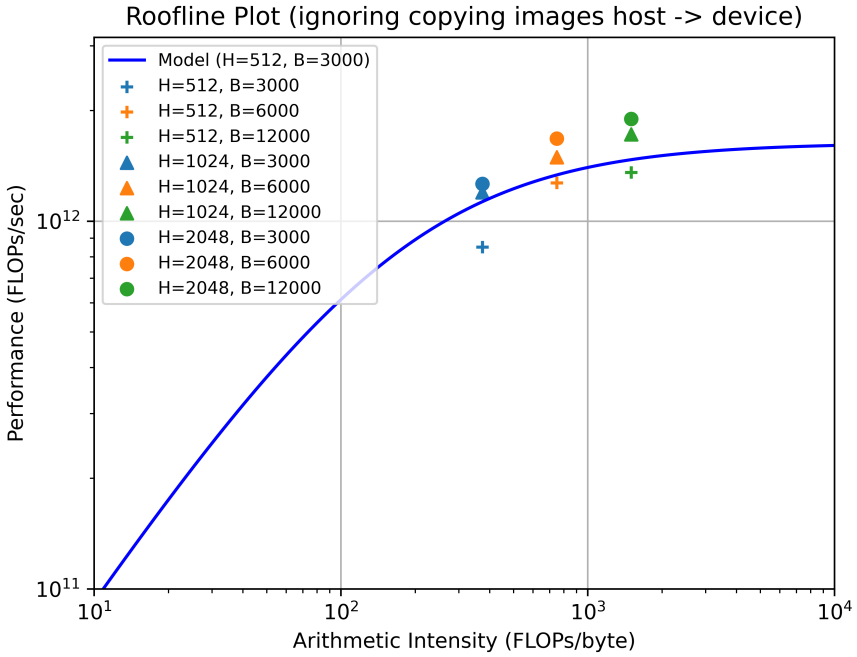


Figure 14: Roofline plot ignoring `copying images host -> device`.

1. You could implement one or two of the optimizations suggested in Question 2.2.
2. Analyze and improve the performance of your CUDA kernels using the Nsight Compute profiler.
3. You could implement the model parallel approach instead of the data parallel approach we have used so far. This will reduce considerably the amount of bytes transferred.
4. Alternatively, you could simply do a theoretical estimate of the running time of the model parallel approach compared to the data parallel approach using the data from Question 2.5.

More information on the model parallel approach is provided below.

Model parallel approach

Let's simplify the problem by considering the following simplified situation. We can think of the feedforward phase as a series of matrix-matrix multiplications. In our case, since we have two layers, we can simplify our analysis and define the feedforward pass as $y = W^{(2)}W^{(1)}x$ where x is the input data and y is the output. The number of columns of x is the batch size. The number of rows of y is 10.

Similarly, the backpropagation phase corresponds to a series of matrix-matrix multiplications. In our case, it can be simplified to $[W^{(2)}]^T y$. The NN weights are then updated using a sum of rank-1 matrices of the form $\Delta W = \sum_i d_i a_i^T$, where the sum is over all the input images. The vector d_i (from image sample i) is the "gradient" vector from the backpropagation (e.g., $[W^{(2)}]^T y$) at layer l while a_i is the activation for layer $l-1$.

The goal is then to subdivide or block the data and operations in such a way that communication is minimized. In the current approach, we split the input sample x into column blocks and assign one block to each processor. Then we do a reduction of all the $d_i a_i^T$ contributions from all processes. Since the size of the matrix ΔW is quite significant, this approach leads to a large MPI overhead.

To reduce the communication time, a common optimization is to partition the NN coefficient matrices W across different processes. This is called model parallelism as opposed to data parallelism. Data parallelism corresponds to splitting the input "data" across different processes, while model parallelism corresponds to splitting the NN coefficients W because the NN is our model in this problem. Model parallelism typically splits the weights along a certain dimension, and synchronization occurs when outputs from other parts of the model are required. Splitting weights across different dimensions results in different operations when synchronization is required.

In an abstract manner, let's focus on a general matrix-matrix product AB and write it out as

$$\sum_k A_{ik} B_{kj}$$

For each triplet (i, k, j) there is one multiplication and addition to perform. We can map all the triplets onto a parallelepiped Ω whose dimensions are given by the dimensions of A and B (namely the number of rows in A, the number of columns in A, and the number of columns in B). The question is then, generally speaking, how to partition this set of triplets Ω across the different processes to minimize communication.

You can experiment with different ideas to reduce the amount of data that needs to be communicated using MPI.

For example, as we perform many iterations, the input data is fixed across iterations (the input images). Therefore for the first layer, you can experiment with splitting $W^{(1)}$ along the rows (index i) and sharing the input images across all processes (which needs to be done only once).

Similarly, $W^{(2)}$ is a very wide matrix with only 10 rows (the 10 labels). You can therefore experiment with splitting this matrix along its columns (index k).

See for example this documentation page from Hugging Face for your information on this topic.