Programming Assignment 4

ECE522: Emerging Memory and Storage Systems (Fall 2024)

This machine problem (MP) has four parts. In Part I, you need to analyze the characteristics of representative DNN models (§3). In Part II, you need to benchmark the GPU execution performance under different settings (§4). In Part III, you need to implement an optimized data migration policy (§5). Finally, you are expected to propose your own optimization design (§6). We list all the deliverables for this MP in §7. Before starting the MP, you need to investigate the related work on this topic (§1) and also set up the necessary experimental environment (§2).

You should work in a team of at most 3 for this MP. You should submit a final report following the given LaTeX template[1], as well as your source codes. In your submissions (related work summary and final project report), please clearly specify each member's contributions in the reports.

1 Summary of Related Work [5%]

While GPU has its onboard memory, the memory capacity is constrained by the memory scaling wall and the limited on-board space of memory packages. Therefore, its memory cannot host the entire working set of large-scale deep-learning workloads. Recent studies have been focusing on using the memory and storage devices to expand the GPU memory.

In the summary of related work, you should investigate the recent studies on GPU memory and storage. As a general guideline, you may want to answer the following questions when writing your related work summary:

Which work(s) solve the same or similar problems? What are the core idea(s) of the related works? Specifically, which part of the related work is "related"? What are the main differences/novelties of your proposed optimization (required in §6) compared to the related works?

In the report, you should include 1 page (2 columns) to show your summary of related work.

2 Setting up the Environment for Simulator

This section will guide you in setting up the experimental environment for this MP. You need to install an in-house GPU simulator. We recommend that you should reserve at least 40GB of free disk space and at least 32GB DRAM of your computer for this MP. The simulator has been tested on Linux (including Ubuntu 18.04/20.04, Manjaro 6.19, and WSL Ubuntu 20.04). Due to the long simulation time, high CPU demand, and high memory consumption, please reserve sufficient hardware resources. We would suggest you NOT to use VM. You can use Cloudlab machines, if your computer does not have enough storage space or computation power.

2.1 Installing the Simulator

You can retrieve the source code of the simulator for this MP from GitHub using the following command. It is a trace-driven discrete event simulator that models GPU page fault handling, data migration, and address translation.

```
git clone https://github.com/0-EricZhou-0/ece522-mp4.git
```

[1] Report template:https://drive.google.com/file/d/1Z7gaIOdOJWFUAmJ1Hmc_seGZbHJvLSyx/view?usp=sharing

First of all, you should install the required package from your package manager:

```
sudo apt install flex bison tmux python3-pip
```

Then build the simulator executable:

```
cd ./src
make clean && make
```

Now you can test the simulator by running a single experiment. Directly execute it with the corresponding config file:

```
./sim configs/example.config
```

3 Understanding DNN Model Characteristics [4%]

In this MP, we provide a set of traces of five real-world large DNN models. You need to write a script to analyze their GPU memory usage patterns. Specifically, you need to work on the following aspects in the study: (1) the minimal GPU memory demand, (2) the distribution of tensor active time and lifetime for each model, and (3) the potential opportunities for swapping tensor off GPU memory when not used by computation.

3.1 Trace Format

Before you start to work on the trace analysis, you should understand the basic trace format we provided to you. Take the model Inceptionv3 as an example, all the profiling results regarding Inceptionv3 are stored in `results/Inceptionv3/` folder, under the directory of your mp4 folder. For each of the batch sizes, a `KernelInfo` file and a `TensorInfo` file are used to describe the characteristics of the kernels. The hierarchy is shown below.

```
results
|--> Inceptionv3/
|  |--> sim_input/
|     |--> 1024Kernel.info  # Kernel Information for batch size 1024
|     |--> 1024Tensor.info  # Tensor Information for batch size 1024
|     |--> 1536Kernel.info
|     |--> 1536Tensor.info
|     |--> ...
|--> ResNet152/
|--> ...
```

We will describe the files below using the batch size of 1024 as an example.

3.2 Tensor Information

For DNN training, all the data used and produced by the model are organized and represented as multidimensional arrays, which are generally referred to as tensors. Tensors are separated into two categories: global tensors and intermediate tensors. A global tensor, such as model weight, is used across multiple training iterations. An intermediate tensor, such as the activation, gradient, and workspace, is only created and used within one iteration.

To ease the tracking and management of these tensors, we assign each tensor used by a model a unique ID. We present the description of each field in Table 1.

```
39 5664669696 false   # example line for tensor #39
76 320 true           # example line for tensor #76
```

Tensor ID	Tensor Size (Bytes)	Is Global Tensor (true/false)
39	5664669696	false
76	320	true

Table 1: Tensor Info Trace Example

The above example shows that the tensor with the ID 39 has the size of 5664669696 bytes and is not a global tensor. While tensor 76 has a size of 320 bytes and is a global tensor.

3.2.1 Kernel Information

A DNN model is made up of a sequence of basic computation elements (e.g., vector add, convolution, ReLU, SoftMax, batch normalization, and so on), and they are usually called kernels. Each kernel has a set of input tensors, a set of output tensors, and potentially a workspace. The workspace shows the amount of additional memory the kernel requires apart from the input and output tensors. Note that not all the kernels require a workspace. The workspace tensor for each kernel is different, and they are only valid at that given kernel.

Each of the lines in the file is formatted as space-separated fields as follows. We showed each field's meaning in Table 2.

```
21 Conv2d_Forward 27.568128 [120,98] [119] 1896   # example line for kernel #21
23 ReLU_Forward 0.435200 [123] [123]              # example line for kernel #23
```

Kernel ID	Kernel Type	Kernel Execute Time (ms)	Input Tensor ID List	Output Tensor ID List	Workspace Tensor ID
21	Conv2d_Forward	27.568128	[120,98]	[119]	1896
23	ReLU_Forward	0.435200	[123]	[123]	

Table 2: Kernel Info Trace Example

The first example shows that, for the kernel with ID 21, there are two input tensors and one output tensor. The two input tensors are the tensors with ID 120 and 98, respectively. The output tensor has the ID 119. Besides the input and output data, the kernel requires a workspace tensor with ID 1896. When all the tensors required by the kernel are present in the GPU memory, the kernel's execution time will be 27.568128 milliseconds.

The field for kernel workspace will be empty for kernels that do not have a workspace. As the second example shows, kernel 23 will take tensor 123 as input and use the same tensor 123 as the output. The kernel will take 0.435200 ms to execute ideally, and the last field is not populated because no workspace is required for ReLU.

3.2.2 Extract Lifetime Information of a Tensor from Given Traces

Let us take tensor 105 as an example. By inspecting the tensor and kernel information files, we can find the tensor's properties and list all the kernels that use it.

```
# tensor info (from Inceptionv3/1024Tensor.info)
105 321126400 false
# kernel info (from Inceptionv3/1024Kernel.info)
891 BatchNorm2d_Backward 72.459267 [106,111,110,118] [115,114,113,112,107,105]
893 Conv2d_Backward_Weight 117.957633 [98,105] [102] 2160
894 Conv2d_Backward_Input 610.128906 [101,105] [103] 2161
```

We can see that the tensor is not global. The tensor is **created** (or *born*) at kernel 891, because it is first used in the iteration by that kernel. The tensor is used in kernels 891 as an output, and in kernels 893 and 894 as an input. It is **destroyed** (or *dead*) at kernel 895, because it is not used in any later kernels in the same iteration. In this manner, we define the tensor 105 is **active** from kernel 891 to kernel 894, and is **inactive** otherwise. Global tensors are always **active**.

When a tensor is created, the corresponding GPU memory is allocated and assigned to the tensor. When a tensor is destroyed, the GPU memory used by the tensor is freed. The memory used by global tensors always exists across iterations.

3.3 Analysing the Characteristics of Models

In this part of the MP, you should write a Python/C++ program named `analyze_model` to analyze the raw traces we provide to you. The traces include different batch sizes from five models: Bert, Inceptionv3, ResNet152, SENet154, and ViT[2]. You should answer the following questions.

(1) Can you determine the mininum memory demand to execute each model for each batch size, when no data is allowed to be evicted from the GPU before a tensor is destroyed?

(2) What is the distribution of the tensor size and lifetime? Plot the total size of the tensors against lifetime in milliseconds.

(3) Plot a figure to show the distribution of the active/inactive time of tensors in each model. Explain the potential for data offloading with the figure.

(4) If we can swap tensors off GPU memory when not used, what is the minimal memory requirement for executing each model (assuming data transfer takes no time)?

For each question, you should plot a figure for each trace to show the results and you should explain your answer based on the figures.

To begin with your program, an input parser has already been done for you. It is used to populate data structures used in the simulator from the input files[3]. Part of your task is to implement two compiler-pass functions for collecting liveness information and tensor inactive period information. The first one is `tensor_first_pass_liveness_analysis` (at `analysis.cc:153`). In this function, you need to analyze the liveness information of every tensor. The second one is `tensor_second_pass_interval_formation` (at `analysis.cc:183`). The function is used to get the inactive period information of every tensor.

These functions are essential for the rest of the MP, so please be careful when implementing them. We have provided more comments in the code for details to help you understand your tasks.

4 Benchmark Performance with Simulator Framework [5%]

In this part, you need to benchmark the model execution performance with different memory and storage settings. For the traces we provided, all benchmarks are running with enough GPU memory to cache the entire working set (ideal case).

4.1 Introduction to the Simulator

4.1.1 Unified Virtual Memory

Compared to the GPU memory, the host machine usually equips a larger memory with limited memory bandwidth, making it a natural option for expanding GPU memory. Modern GPUs make this procedure transparent with unified virtual memory (UVM). UVM enables a unified and coherent virtual memory space between the host and GPU, and the application data can be allocated to the space and accessed by the host and GPU with shared virtual addresses. With the cooperation of GPU hardware and runtime, UVM maintains data consistency transparently and enables on-demand data migrations between the host and GPU at the page granularity.

[2]Specifically, all the traces we provided is listed as follows: Bert with batch sizes 256, 384, 512, 640, 768, 1024; Inceptionv3 with batch sizes 512, 768, 1024, 1152, 1280, 1536, 1792; ResNet152 with batch sizes 256, 512, 768, 1024, 1280, 1536; SENet154 with batch sizes 256, 512, 768, 1024; and ViT with batch sizes 256, 512, 768, 1024, 1280.

[3]See function `void loadKernelInfo()` in `src/main.cc` for more details.

Upon accessing a UVM page absent in the GPU memory, a GPU page fault will be triggered to request a data migration from the host. When the GPU memory is fully occupied, the least recently used pages are evicted from the GPU memory to the host memory. However, GPU memory still cannot scale purely relying on the host memory to meet the increasing demands of large deep learning workloads.

An alternative approach is to expand GPU memory with flash-based SSDs. The rapidly shrinking process technology has allowed SSDs to boost their bandwidth and capacity by increasing the number of chips. The simulator models a system in which the SSD's logical address space is also part of the unified and coherent virtual memory space.

4.1.2 Test the Ideal Case Performance with Enough GPU memory

To begin with, you should first use the simulator to test the DNN training performance under the ideal case, which means the GPU has enough memory to perform the training.

For each simulation setting, you need a config file. We still take the `Inceptionv3` with the batch size 1024 as an example. The configurations in the file are listed below[4].

```
# Simulation output specifications
    output_folder           ../results/Inceptionv3/1024-ideal
    stat_output_file        sim_result
# Simulation input specifications
    tensor_info_file        ../results/Inceptionv3/sim_input/1024Tensor.info
    kernel_info_file        ../results/Inceptionv3/sim_input/1024Kernel.info
    kernel_aux_time_file    ../results/Inceptionv3/sim_input/1024AuxTime.info
# System specifications
    system_latency_us       45
# CPU specifications
    CPU_PCIe_bandwidth_GBps 15.754
    CPU_memory_line_GB      128
# GPU specifications
    GPU_memory_size_GB      40
    GPU_frequency_GHz       1.2
    GPU_PCIe_bandwidth_GBps 15.754
    GPU_malloc_uspB         0.000000814
# SSD specifications
    SSD_PCIe_bandwidth_GBps 3.2
    SSD_read_latency_us     12
    SSD_write_latency_us    16
    SSD_latency_us          20
# PCIe specifications
    PCIe_batch_size_page    50
# Simulation parameters
    is_simulation           1
    is_ideal                1
    num_iteration           2
    eviction_policy         LRU
    use_movement_hints      0
```

Suppose the config file is `src/configs/Inceptionv3/sim-1024-ideal.config`. To run the simulation, execute the following commands in the `src/` folder, and the simulation will start shortly.

[4]The config file is parsed using the function `void parse_config_args()` in `main.cc`. You can see how simulator-wide variables are set in this function, and you can modify it if more parameters are needed for your simulation setting in the future.

```
make
./sim ./configs/Inceptionv3/sim-1024-ideal.config
```

4.1.3 Output Statistics

After the execution with the simulator, the output statistics should be generated using the designated path indicated in the configuration. In this case, `results/Inceptionv3/1024-ideal/sim_result.*`. The exact hierarchy is shown below[5].

```
results
|--> Inceptionv3/
   |--> 1024-ideal/
      |--> statistics/
      |  |--> ...
      |--> sim_result.evc
      |--> sim_result.final
      |--> sim_result.kernel
      |--> sim_result.pcie
      |--> sim_result.pf_tensor
```

The overall statistics can be found in `sim_result.final`. The timing information in the output file is measured in the number of GPU cycles. Now, let us verify the output statistics. If you convert all the kernel times into GPU cycles and sum them up (the value is also printed at the beginning of the simulation, you can find the line `Ideal Execution Time: 71175129375 cycles`), in the ideal case, the total number of execution cycles should match the sum of all kernel time.

In the file `sim_results.final`, we can find the execution time of the two iterations we simulated. The execution time is logged as follows.

```
kernel_stat.iter0.exe_time = 71175129375
kernel_stat.iter1.exe_time = 71175129375
kernel_stat.total.exe_time = 142350258750
```

We can see that the results match what we expect. You can read the function `Stat::analyzeKernelStat()` (at `simulationComponents.cc:886`) for the meaning of different output statistics. These stats can help you analyze the performance of the system and potentially help you with the debugging.

Moreover, in the non-ideal case, it is advised to run the simulation for multiple iterations (at least 4) and get an average from later iterations. Because some actions will not take effect until the next iteration, the performance will not be stable before the system reaches a steady state.

4.2 Expand GPU Memory with Host Memory

Compared to the GPU memory, the host machine usually has a larger memory. Now consider the case if the GPU memory is limited and cannot host the entire model, we can expand the GPU memory with host DRAM using techniques like unified virtual memory (UVM). We treat the GPU memory as a cache and all access to pages within the host memory requires a migration to the GPU memory first. A simple LRU policy is used when we need to evict pages to the Host DRAM.

To enable this in the simulator, you should modify the following parameters in the example config file located at `src/configs`.

```
CPU_memory_line_GB      128   // the upper limit of CPU memory that will be used
GPU_memory_size_GB      40    // the size of GPU memory that will be used
```

[5] Refer to the class `Stat` in `SimulationComponents.h` and `SimulationComponents.cc` to see the description of all output files. You can always add your own output stats using this class.

You should set the GPU size under the working set size and set the upper limit of CPU memory to be able to contain entire working set. Run the model with all data residing in the host DRAM before the training starts, and evaluate the performance compared to the ideal case.

4.3 Expand GPU memory with Flash Memory

An alternative approach is to use the SSD device to expand the GPU memory. We follow the same setup but replace the host DRAM with SSD. To only use SSD to expand GPU memory, you should set the upper limit of CPU memory in the config file to 0 so it will not use the host DRAM.

You should test the provided models, compare them to the ideal setup, and show what is the slowdown of the execution? In what percentage of the execution time contributes to the page migration process?

4.4 Leverage Both Host DRAM and SSD to Expand GPU Memory

Now, you should test the setup by using both DRAM and SSD to expand the GPU memory. You can try a different combination of memory and SSD sizes. You should try at least 5 setups and report the experimental results within a figure.

4.5 Cost-effectiveness Analysis

We use SSD as a memory expander of GPU since it offers a much larger capacity at a lower cost than DRAM. We define cost-effectiveness improvement as the performance benefits over the setup cost. For example, if we can achieve 90% of the ideal case throughput with 70% of the unit cost, we improve cost-effectiveness by $90\%/70\% = 1.29\times$. To obtain the cost, you can investigate the current market price and calculate the unit cost ($/GB) of each setup based on their price and capacity. State your assumptions clearly when answering the question.

Based on the performance results, calculate the cost-effectiveness of all the above hardware setups. You should report the numbers with a table/figure.

5 Optimizing Data Migrations [5%]

In this part, you need to implement a tensor migration policy to improve the overall GPU performance. The default migration only triggers when a required page is needed, which misses the opportunity to hide the migration latency with background transfer. Here, we expect you to implement a naive prefetching algorithm that fetches the tensor data in advance that will be executed in the near future.

You should also implement a new page eviction policy. By default, when the GPU memory is full, we need to leverage LRU to select a victim page to be evicted to the host memory or SSD. Consider that you have both host memory and SSD as a memory expander, and propose a new eviction policy that evicts tensor based on the data hotness.

5.1 Hints for Tensor Prefetching and Eviction

In the simulator, we use a movement hint to guide the migration of the tensor data, as shown below:

```
enum TensorLocation { NOT_PRESENT, IN_SSD, IN_CPU, IN_GPU, NOT_KNOWN };
class TensorMovementHint {
    int issued_kernel_id; TensorLocation from; TensorLocation to; Tensor* tensor;
};
```

The `TensorLocation` indicates the location of a tensor, it will be used to indicate the current location and migration target. A movement hint consists of four major parts: before which kernel should the data movement be issued, which tensor needs to be moved, where the tensor is moved from, and where the tensor should be moved to. With this representation[6],

1. A tensor pre-allocation can be encoded as `TensorMovementHint(from=NOT_PRESENT, to=IN_GPU)`.
2. A tensor prefetch can be encoded as `TensorMovementHint(from=NOT_KNOWN, to=IN_GPU)`.
3. A tensor pre-eviction can be encoded as `TensorMovementHint(from=IN_GPU, to=<destination>)`, where the destination can be `IN_CPU` or `IN_SSD`.

When implementing the migration scheme, you should implement the function `scheduling_movement_hints` (at `analysis.cc:422`) and populate the data structure `std::vector<TensorMovementHint> movement_hints`, so that the simulator can make use of your data movement hints. The migration will be triggered before each kernel is executed[7].

To implement your prefetching, generate the data movement hint to allow GPU to always prefetch the tensor that will be used in the next kernel execution.

5.2 Changing GPU On-Demand Data Eviction Scheme

You should also change the default data eviction policy in `selectEvictPTE` in `simulationComponents.cc:258`. This function is called when the GPU memory is full, but a page needs to be migrated, due to either an on-demand migration or a prefetching. The function selects a candidate page to be evicted and returns it back to the caller. In default, we use a basic LRU policy to select the candidate page, we also provide you with some examples of policies. You can have a better understanding about how the function works after reading the comments and documentation in the code.

You should decide which candidate tensor to be selected based on its hotness, and aslo decide the target location (Host DRAM or SSD) of the eviction. To decide the tensor hotness, you may need to develop your own data structure (e.g., a table or counter) to record the tensor accesses during execution.

5.3 Result Report

Test your migration/eviction schemes with different settings. Compare the result with results in §4. Does the new migration policy work better? Why or why not? (Conduct the analysis based on the experiment results: number of page faults, miss/hit ratio on prefetch data, and so on).

6 Propose Your Own Optimization [6%]

After §3, §4, and §5, you should be able to understand the GPU memory/storage and get familiar with the simulator. It is time for you to propose your own optimization. Here are some hints on design choices for your consideration, you definitely have the choices to develop your own optimization schemes.

- Enable background eviction to reduce computation stalling.
- Use heuristic/ML predictor for smarter tensor prefetching/eviction.
- Explore fetching/eviction policy with the data transfer between the host and SSD.
- Explore fetching/eviction policy with a tiered memory hierarchy with NVM or CXL memory.

[6] Read the function `KernelBeginEvent::guidedTransfer(TensorMovementHint *hint)` (at `simulationEvents.cc:164`) to understand how each of the movement hints is interpreted exactly.

[7] The exact triggering logic can be found in function `KernelBeginEvent::execute` at `simulationEvents.cc:322`

- ...

In the final report, you should describe your design choices in detail (e.g., the motivation for choosing the design). Describe the changes you made in the simulator and how it works. You should also include performance numbers to show its efficiency compared to the previous schemes and describe the reason why it can work. Please follow §7 on how to present your design and performance results in the final report.

7 Submission Guideline

You are expected to turn in the following deliverables:

1. The MP report in PDF format, `report.pdf`. You should follow the provided LaTeX template to draft the report. Please include the following:
   (a) Abstract section.
   (b) The study of the tensor characteristic with the provided 5 models. (§3)
   (c) Reporting the training performance results under different experiment settings. (§4)
   (d) Evaluate the performance of the optimized data migration design in §5, and explain why it works.
   (e) A detailed description and evaluation of your proposed optimization in §6.
   (f) Summary of related works.
2. The raw output of the benchmark experiments in Part II, III, and IV.
3. Your scripts for analyzing the model characteristic in Part I. Your modifications to the simulator in Part III, and IV. You should submit the modified code in a zip.
4. If you change any other simulator settings that this MP document does not require or use a different environment, please document your setup at the beginning of your report.

You are expected to submit a zip file arranged in the following manner:

```
<your_netid>_mp4.zip
|---> report.pdf
|---> part1/
|   |---> results/
|---> part2/
|   |---> hostdram_results/
|   |---> ssd_results/
|---> part3/
|   |---> results/
|---> part4/
|   |---> results/
|---> src/
    |---> <source code of entire project>
```