Compiling Neural Networks with Dynamic Resource Adaptation:

Final Report

Patrick Coppock
pcoppock@andrew.cmu.edu

Yuttapichai (Guide) Kerdcharoen
ykerdcha@andrew.cmu.edu

https://pnxguide.github.io/15745_project/

1 Introduction

1.1 Problem and Opportunity

Our project will examine neural network (NN) compilation, specifically large language models (LLMs). While NN compilers may optimize within GPU kernels, many traditional NN frameworks only optimize across kernels. Inter-kernel optimization simply schedules kernel launches and memory copies to improve the run time (or throughput) of the NN. Furthermore, such compilation doesn't resemble that done in the front and back ends of traditional CPU compilers—where high-level, human-readable source code is transformed into low-level machine code—but is nevertheless analogous to the scheduling performed in the middle end of modern, optimizing CPU compilers.

NNs are compiled either for training to fit parameters or for inference to compute with a trained NN. Training is a batch job, while inference is usually more interactive and may serve a bursty request pattern. In our project, we will focus on inference. Sharing a device between multiple bursty jobs with different priorities requires the low-priority jobs to be able to respond to varying resource availability. For example, one high-priority job might suddenly receive many more requests, resulting in a need for more memory. Lower-priority jobs will need to be able to give up memory while continuing to utilize the GPU.

The largest inference jobs are so-called "large language models" (LLM), such as OpenAI's GPT [4] and Meta's OPT [6]. These models require billions of weights and are trained on hundreds of cutting-edge GPUs. The state of the art in LLM inference compilation depends heavily on offloading of weights, activations, or the transformer key-value (KV) cache to CPU DRAM memory (or even disk storage). How many batches to process through a layer at a time and how much of what to offload where are both important questions that affect how memory is copied and kernels launched. These questions are traditionally answered before inferencing starts, resulting in inelastic GPU memory use throughout inferencing.

Can we compile inference applications to elastically use GPU resources? Our project will examine the feasibility of placing offloading-based optimization in the application runtime, allowing for elastic GPU resource usage.

1.2 Our Approach

While FlexGen searches policies offline, we suspect it could be extended to search policies online, throughout inferencing. This could lead to a straightforward solution for memory elasticity: drop batches from the current block as memory becomes scarce and research policies at the start of each block to determine whether its size may be increased.

A more extensive solution could tune other model inputs throughout block inference. Offloading can be tweaked throughout block inference; the only limit is that batches cannot be added to the block, in the middle of an inference pipeline (as their KV cache and activations depend on earlier layers).

1.3 Related Work

As we already mentioned, state-of-the-art LLM inference compilation offloads memory objects from GPU memory to CPU memory and storage. [5] proposes FlexGen, which does just this. The authors notice that for LLMs, it makes more sense to offload the batch transformer key-value (KV) cache and activations than to offload the layer weights. Therefore, they propose combining batches into blocks. All batches in a block are run through a layer at a time.

FlexGen contributes a cost model for the above algorithm as well as a PyTorch interposer that manages data movement according to a statically-given policy. The model relates inference throughput to the bandwidths of each memory, the sizes of each memory, and the block size (number of batches). FlexGen's policy search minimizes the modeled cost by tuning both the degree of offloading of each operand type (weights, activations, and KV cache) onto each memory and the block size. The optimized inputs are then used to guide inference. FlexGen is published on GitHub.

1.4 Contributions

Our project makes contributions to FlexGen as follows:

- Python implementation of cost model and policy search
- Extension of FlexGen to allow it to respond to changing device memory capacity
- GPU memory simulator

These contributions are published in our fork of FlexGen on GitHub at coppock/FlexGen.

2 Details Regarding Our Approach

2.1 Cost Model and Policy Search

Key components of FlexGen as it was released are the cost model and policy search. FlexGen seeks to optimize the placement of model weights, activations, and key-value cache in order to maximize token throughput. The cost model relates inference latency to data placement, memory bandwidth, and compute bandwidth, while the policy search uses this model to optimize throughput varying batch size, number of batches per block, and data placement.

The FlexGen code release at this time (April 26, 2023) does not implement the cost model or policy search, though it appears that the authors hope to include these components soon. Our project relies on the components, and we implemented them from descriptions in the paper [5].

2.1.1 Cost Model

The cost model relates inference latency to data placement and machine bandwidths. Large language model inference has two stages, **prefill** and **generate**. In the first stage, the KV cache and activations are initialized and the first token is generated. In the second, **generate**, stage, the remaining output tokens are generated. In both stages, generation is composed of the five following steps:

1. Move data from disk to host memory.
2. Move data from host memory to device memory.
3. Compute token.
4. Store data from device memory to host memory.
5. Store data from host memory to disk.

Note that the amount of data moved by the data movement steps depends on the layout. Furthermore, the latency of each step depends on its respective bandwidth, e.g., the first step depends on the disk read bandwidth. These steps may be pipelined, if memory capacities allow. Pipelining the steps causes the amortized latency to be the maximum latency of the five steps. Because the prefill stage data movement may look different from that of the generation stage, the stage latencies are calculated separately.

The latency of each step depends on three things: (1) the model architecture, (2) the machine profile (bandwidths), and (3) the execution policy (batch size, batches per block, and data layout). The machine profile is composed of the constants in table reftable:prof

An execution policy is a set of configurations for scheduling neural network batches (including weights, activations, and key-value caches). The definition of the policy is similar to [5], where policy p =

Table 1: Machine Profile

Variable	Definition
$ctog\_bdw$	device write (float/s)
$gtoc\_bdw$	device read (float/s)
$dtoc\_bdw$	disk read (float/s)
$ctod\_bdw$	disk write (float/s)
mm_flops	device matrix multiplication (FLOPS)
bmm_flops	device batched matrix multiplication (FLOPS)
cpu_flops	CPU computation (FLOPS)

$(bls, gbs, wg, wc, wd, cg, cc, cd, hg, hc, hd)$. The bls and gbs are the numbers of batches per block and batch sizes, respectively. The wg, wc, and wd are placement distributions (out of 100%) among GPU (wg), CPU (wc), and disks (wd) for weights. The cg, cc, and cd are placement distributions (out of 100%) among GPU (cg), CPU (cc), and disks (cd) for key-value caches. The hg, hc, and hd are placement distributions (out of 100%) among GPU (hg), CPU (hc), and disks (hd) for activations.

With *weights* bytes of weights and *activations* bytes of activations, as well as the machine profile variable definitions in table 2.1.1, we can relate inference latency, T. See [5] for calculations of weights, activations, and key-value cache sizes.

$$T = T_{pre} \cdot l + T_{gen} \cdot (n-1) \cdot l \tag{1}$$

$$T_{pre} = \max(ctog_{pre}, gtoc_{pre}, dtoc_{pre}, ctod_{pre}, comp_{pre}) \tag{2}$$

$$ctog_{pre} = \frac{(wc + wd) \cdot weights + (hc + hd) \cdot activations}{ctog\_bdw} \tag{3}$$

T_{gen} is calculated similar to equation 2, while $gtoc_{pre}$, $dtoc_{pre}$, $ctod_{pre}$, $comp_{pre}$ are calculated similar to equation 3; i.e., the total amount of data processed is divided by the bandwidth.

2.1.2 Policy Search

[5] suggests that an optimal policy can be searched by solving the following linear programming problem.

$$
\begin{aligned}
&\min_{p} \ T/p.bls \\
&\text{s.t. GPU peak memory} < \text{GPU memory capacity} \\
&\quad \text{CPU peak memory} < \text{CPU memory capacity} \\
&\quad \text{disk peak memory} < \text{disk memory capacity} \\
&\quad wg + wc + wd = 1 \\
&\quad cg + cc + cd = 1 \\
&\quad hg + hc + hd = 1
\end{aligned} \tag{4}
$$

From the equation 8, the optimization problem minimizes a function of the latency T predicted by the cost model in Section 2.1.1. The function of the latency T is the latency predicted from the model divided by the block size bls. The reason is the latency T is computed per block (a collection of batches). Therefore, the cost per layer must be calibrated by the number of batches per block to make fair games for other bls values.

GPU and CPU peak memory are both functions of the policy p and constants from neural network architectures. GPU peak memory can be roughly computed based on [5] as follows.

$$
\text{GPU peak memory} = \text{GPU fixed memory} + \text{GPU working memory} \tag{5}
$$

The GPU fixed memory is a function of bls, wg, hg, and cg, which are important parameters for GPU memory placement distributions. The GPU working memory is a function of gbs, wg, hg, and cg, where it considers the largest memory consumption among layers including qkv, att_1, att_2, $embed$, mlp_1, and mlp_2.

Similarly, CPU peak memory can be roughly computed based on [5] as follows.

$$
\text{CPU peak memory} = \text{CPU fixed memory} + \text{CPU working memory} \tag{6}
$$

The CPU fixed memory is a function of bls, wc, hc, and cc, which are also important parameters for CPU memory placement distributions. The CPU working memory is a function of gbs, wg, hg, hd, and cd.

2.1.3 Implementing policy search

The policy search is implemented as a linear programming problem. We chose PuLP [1], a linear programming solver written as a Python library, to implement the policy search since FlexGen, the system we extended, is

written in Python. However, the policy search in [5] is not an exact linear programming problem since the function computing GPU peak memory contains a max function. Thus, we needed to reformulate the policy search using the Minimax linear programming technique [2]. Given the following GPU working memory function in [5].

$$\begin{aligned}\text{GPU working memory} = {} & 2(1-wg)(8h_1^2 + 4h_1 \cdot h_2) + \\ & (1-hg) \cdot 2s \cdot h_1 \cdot gbs + \\ & \max(qkv, att_1, att_2, embed, mlp_1, mlp_2)\end{aligned} \tag{7}$$

Simply, $qkv, att_1, att_2, embed, mlp_1, mlp_2$ are functions of gbs, wg, hg, and cg. The full equations to compute them can be found in [5]. Hence, the linear programming problem reformulated using the Minimax linear programming technique is

$$\begin{aligned}\min_{p, GPU_{max}} \quad & T/p.bls \\ \text{s.t. GPU peak memory} < {} & \text{GPU memory capacity} \\ & GPU_{max} > qkv \\ & GPU_{max} > att_1 \\ & GPU_{max} > att_2 \\ & GPU_{max} > embed \\ & GPU_{max} > mlp_1 \\ & GPU_{max} > mlp_2 \\ & ... \\ \text{where GPU working memory} = {} & 2(1-wg)(8h_1^2 + 4h_1 \cdot h_2) + \\ & (1-hg) \cdot 2s \cdot h_1 \cdot gbs + \\ & GPU_{max}\end{aligned} \tag{8}$$

in which we introduced GPU_{max} as another parameter to eliminate the max function in the GPU peak memory. We did the Minimax technique for every max function in the equation.

Rather, with the bls and gbs, the problem is still not a linear programming problem. However, the bls and gbs are integers can be ranged from $(0, 20)$ and $(0, 64)$ (only the multiples of 4), respectively. Since the permutations of bls and gbs are limited, [5] suggested enumerating those permutations and solving multiple linear programming problems for each permutation. Thus, we can solve it as a linear programming problem.

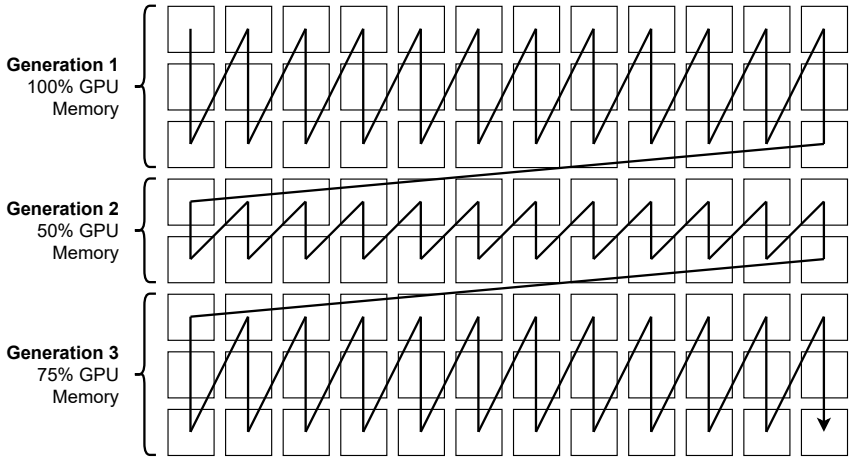

Figure 1: An example of dynamic scheduling based on our approach.

2.2 Dynamic Scheduling

While the cost model and policy search needed to be implemented, the primary contribution of our project is to make FlexGen even more flexible. Specifically, our goal was to build in support for changing resource availability.

There is a trade-off between elasticity and implementation complexity. FlexGen could be extended to allow for changing resource availability between layers, or between tokens; however, because of the extensive use of pipelining in FlexGen, this would have a high implementation complexity. We chose instead to allow FlexGen to modify its resource usage between block generations.

FlesGen is designed such that a model object is created before the first generation and then used for each generation. Initially, we hoped to keep the same model object around while changing resource usage. However, changing resource availability results in a changing execution policy and the model object is built with this policy in mind. Therefore, we modified FlexGen to recreate a model object whenever the execution policy. The model doesn't keep state across generations, so creating a new one is simple.

An example of dynamic scheduling based on our approach can be visualized as in Figure 1. The figure shows that each generation when rerunning the policy search can adapt their computation and resource placement to the new GPU memory constraint while trying to minimize the latency based on the cost model.

2.3 GPU Memory Simulator

To simplify our evaluation, we built a simple GPU memory simulator to show that our approach allows FlexGen to run along with other processes elastically. The memory simulator simply takes program traces and simulates how GPU memory gets utilized.

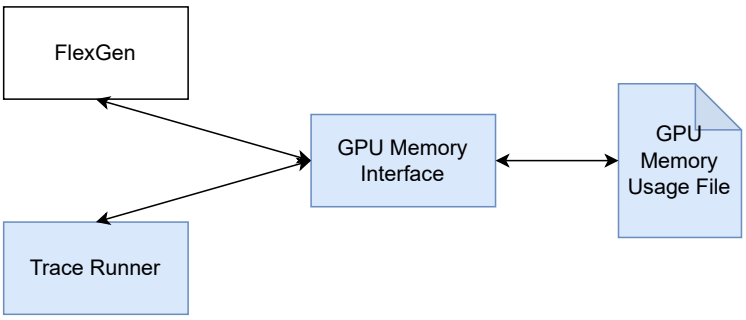

Figure 2: GPU memory simulator design (blue components represented our implementations).

2.3.1 Overall design

The GPU memory simulator consists of three logical components. The first component is the GPU memory utilization file. This file contains a single integer representing bytes used by the GPU. This file will be read by both the GPU simulator and the FlexGen. The second component is the trace runner. The trace runner is simply a loop taking a trace file, whose format is described in Section 2.3.2. More details are provided in Section 2.3.3. The third component is the Python interface that allows implementors to read and update the memory utilization file via Python functions. This is used by both the trace runner and the FlexGen. All of the designs are captured as a diagram shown in Figure 2.

2.3.2 Input trace format

The format for input traces is a list of triples. Each triple consists of (1) ideal start time, (2) ideal end time, and (3) peak GPU memory utilization (in bytes) for each process. All the fields for each triple are delimited by a space. An example of an input trace file can be viewed at coppock/FlexGen.

2.3.3 Running traces

The trace runner will read all triples for each process. It will start to interpret from the earliest process. If the GPU memory is still available for the process, the trace runner will increment the GPU memory usage file by the peak GPU memory utilization of the process. Otherwise, the process will be queued (but firstly prioritized) until GPU memory is sufficient. After processes spend all the time specified in the triple (i.e., $end - start$), the GPU memory usage file will decrement by the peak GPU memory utilization of the process.

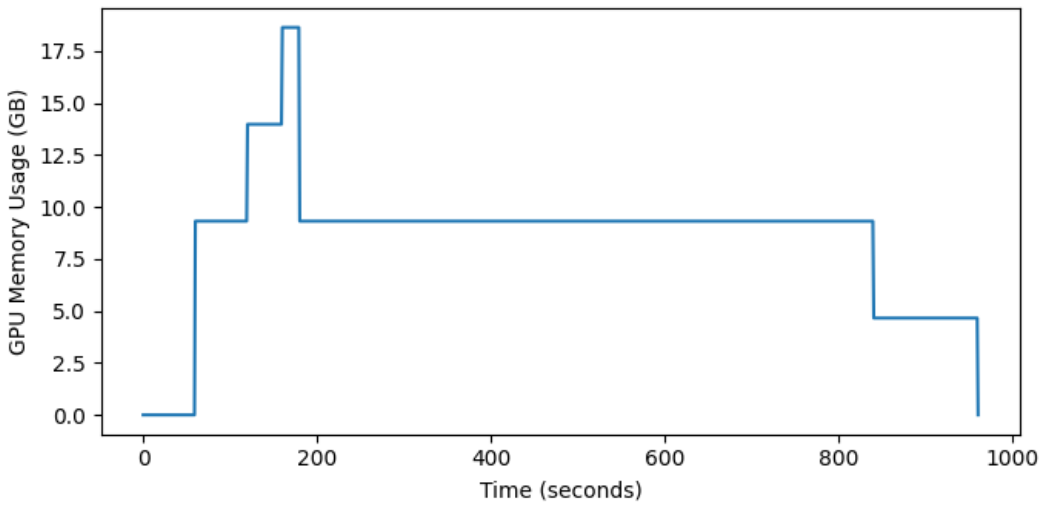

Figure 3: GPU memory simulator with our program traces only.

3 Experimental Setup

3.1 Dynamic scheduling evaluation

In order to test the functionality of our changes, we used FlexGen (both before our modifications and after them) to run a scenario from the HELM benchmark [3]. We ran OPT-6.7B for a Massive, Multitask Language Understanding scenario on an NVIDIA A30 GPU with 24GB of device memory and 256GB of host memory.

We monitored memory usage through the `torch.cuda.memory_allocated()` PyTorch interface. When running the policy search, we decided to input a device memory capacity of only 20GB after running into overallocation problems. (FlexGen's memory size model is simple and may not be capturing PyTorch's actual memory usage.)

3.2 Evaluation with synthetic environment

We simulated the program traces that are expected to use GPU memory inconsistently to see how our approach can be used in a more realistic environment. The program traces were simulated under our GPU memory simulator mentioned in Section 2.3 Figure 3 shows how GPU memory is used by our synthetic program traces.
Without any interference, our program traces should be ended in 16 minutes (i.e., 960 seconds).

We used the same setting as in Section 3.1 and ran the GPU memory simulator along with the same FlexGen task (i.e., OPT-6.7B).

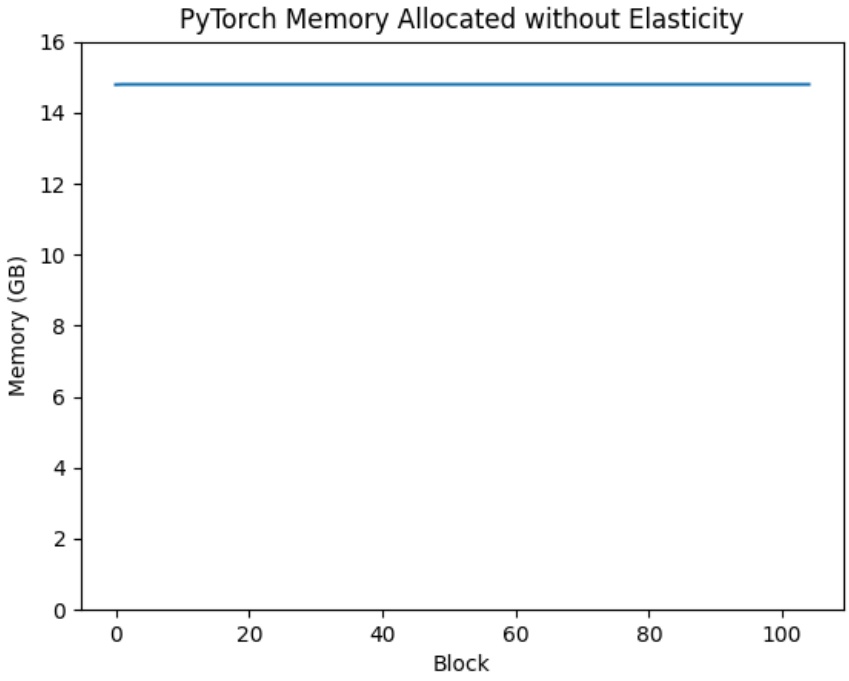

Figure 4: Baseline FlexGen device memory usage is nearly constant.

4 Experimental Evaluation

4.1 Dynamic scheduling evaluation

4.1.1 Baseline

The memory usage is shown to be almost constant across blocks in figure 4. There is a small jump of 8.5MB after generation of the first block. Interestingly, although the device memory capacity is set to 20GB, FlexGen is only using 15GB. While we suspect this is because PyTorch is managing memory differently than the cost model calculations, we were unable to determine why this is and, therefore, leave it as future work.

4.1.2 Elastic Experiment

In our experiment, we induced a resource shrinkage of 4GB after the third block generation, dropping the effective device memory capacity to 16GB. We were able to demonstrate FlexGen's ability to use less memory on subsequent block generations. See figure 5. However, we again see that FlexGen's actual use of device memory doesn't seem to accord with the cost model. Specifically, its usage dropped by under 1GB, while we reduced capacity by 4GB. There are two possible reasons for this. First, Python may not be reclaiming all objects in the original model. We have noticed leaks in PyTorch before, and this could be one of those cases. The second possibility relates directly to the issue we noticed in the baseline evaluation: the cost model doesn't seem to be correctly relating memory usage to execution policy and model architecture.

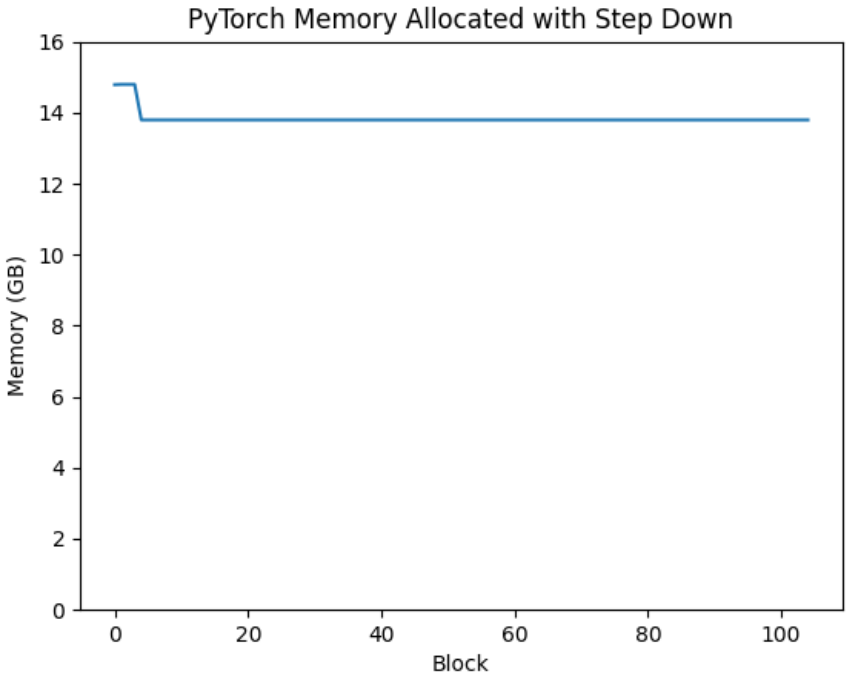

Figure 5: Elastic FlexGen can respond to resource shrinkage.

4.2 Evaluation with synthetic environment

4.2.1 Baseline

We firstly ran the GPU memory simulator with the input traces along with the original FlexGen. The result shows that until our program traces could start running, they needed to wait for the whole FlexGen task to be completed first. This shows how inelastic the original FlexGen is for multi-tenancy environments.

4.2.2 Elastic Experiment

We then ran the GPU memory simulator with the modified FlexGen using our approach. Figure 7 shows the GPU memory usage extracted from our GPU memory simulator. The number should be higher than the actual GPU memory utilization since we update the memory file with the computed memory usage from the cost model. The result shows that other programs do not need to wait for the whole FlexGen task to be completed; it just needs to wait for each FlexGen generation loop to end. As a result, all program traces can be completed within 980 seconds (20 seconds later) due to the FlexGen interference. However, this is better than the original FlexGen, in which we need to wait for all inference tasks to be done. The spike patterns shown in the plot represent the gap between generation loops that allow other processes to run.

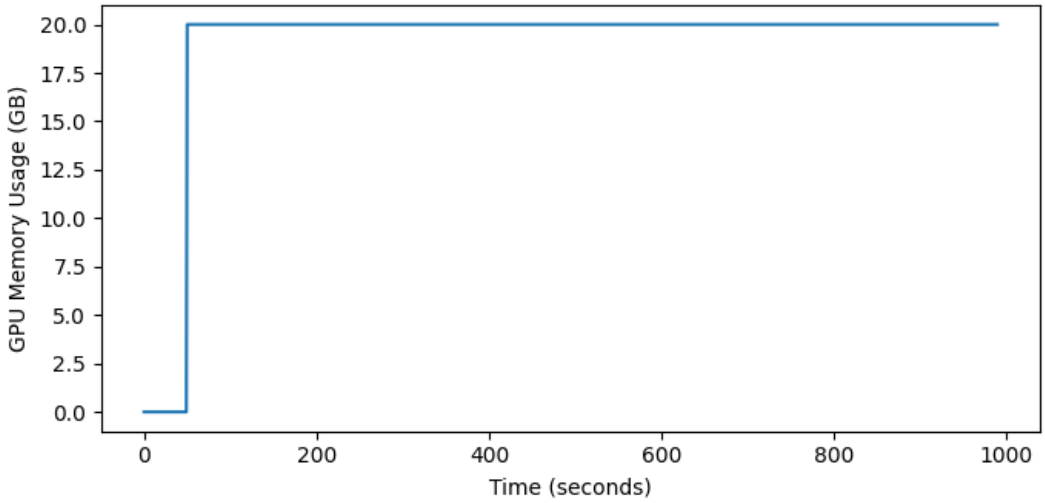

Figure 6: GPU memory simulator with the original FlexGen.

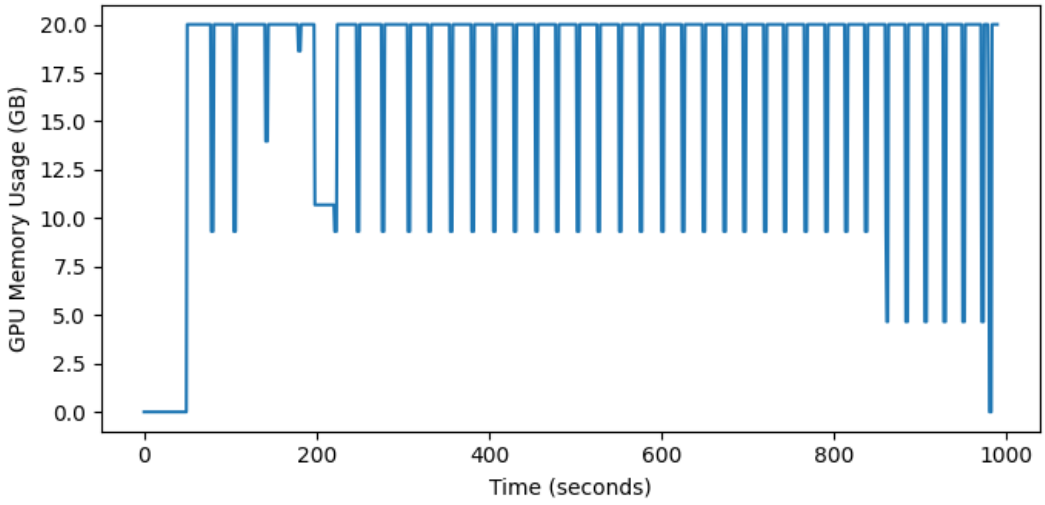

Figure 7: GPU memory simulator when FlexGen using dynamic resource adaptation.

5 Surprises and Lessons Learned

One surprise was that the cost model wasn't accurate. This may be why the authors of FlexGen didn't publish code, and instead suggest users manually tune parameters.

Another surprise was that the policy search took longer to implement than we expected. This was probably mostly due to our limited experience with LP problems.

Originally, we planned to reduce our device memory usage by dropping batches from the block during generation. It wasn't until late in the project that we realized the device memory usage didn't depend on block size (as the policy search was putting only weights on the GPU; cache and activations were left on the host) Therefore, this approach didn't work. We had to shift gears and change data layout, which we expected would be significantly more challenging. Thankfully, by recreating the model object between generation loops, we could change the layout in a straightforward manner. One drawback with this, is that our project can't react to resource availability changes within a generation. We leave such intra-generation adaptation to future work.

6 Conclusions and Future Work

There is a lot of room for improvement in integrating our solution with existing FlexGen code. We made changes in the most straightforward way possible which was not the most elegant way. Examples of this include redefining a data structure for the execution policy and recreating the model object. Also, it is possible to fix data layout up as necessary after resource availability has changed, rather than rebuilding all model data structures. Furthermore, there is no need to rerun the policy search if resource availability hasn't changed which is how our changes are currently implemented.

According to our proposal, we left two stretch goals (finer-grained dynamic scheduling and uses of other sub-optimal policies) untouched. These both are good directions for future work.

Another important question to answer is, "What overheads are introduced by rerunning the full policy search between token generations?" FlexGen does not seem to suffer much of a performance loss, but this should be quantified. The cost model runs in under a second, while block generation takes tens of seconds, so rerunning the policy search may be worth it.

FlexGen implements some compiler functionality (data movement scheduling), and we have extended it in a way analogous to JIT: FlexGen can now respond dynamically to changes in resource availability.

7 Distribution of Total Credit

Each author contributed a similar effort to the project. We suggest the credit be divided fifty-fifty to Patrick Coppock and Yuttapichai Kerdcharoen.

References

[1] Optimization with PuLP; PuLP 2.7.0 documentation — coin-or.github.io. `https://coin-or.github.io/pulp/index.html`. [Accessed 26-Apr-2023].

[2] R.K Ahuja. Minimax linear programming problem. *Operations Research Letters*, 4(3):131–134, October 1985.

[3] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models, 2022.

[4] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training.

[5] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput Generative Inference of Large Language Models with a Single GPU, March 2023. arXiv:2303.06865 [cs].

[6] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, June 2022. arXiv:2205.01068 [cs].