OPTIMAL MODEL SCHEDULING USING OPERATING SYSTEMS TECHNIQUES

Cole Smith [1]

ABSTRACT

In the past few years, we've seen an enormous spike not in ML theory, but in application. However, although many great papers have come out of this, few generally applicable systems have shown their face as a result. Self-driving cars and intelligent security systems have become huge and immensely popular, but also bringing their own challenges. Hardware has become increasingly limited in recent years, and as we move to modular programming paradigms we have begun to rely on job formats for machine learning applications. Scheduling these jobs on limited hardware has been looked at a lot, but this problem has actually been solved for over 50 years in Operating Systems. In this paper, we explore multi-tiered shortest job first implementations with intelligent bin packing to allow for efficient execution of multiple jobs. Github link for code here: https://github.com/Th3OnlyN00b/MLScheduling

1 PROBLEM STATEMENT

Machine learning code has spread everywhere, and the applications are nearly endless. The two most relevant applications to this paper are self-driving cars and intelligent security systems. Self driving cars observe their surroundings using a large variety of sensors, and this data is collected and processed through a series of ML models. Frequently, these are not part of one computer, and are instead sent asynchronously. This leads to the information being used for inference from each sensor set to be formatted like a job. These jobs often have radically different priorities, for example the job coming from the front of the car where the distance sensor has been tripped may need to be run immediately, before the camera from the back where no other sensors were tripped. Conventional solutions include two priority lanes, Real-Time (RT) and Best Effort (BE). These jobs are classified as either RT or BE, and the RT jobs have the potential and ability to interrupt BE jobs. This definitely helps, but does not solve the issue of multiple priorities, as the priorities can easily exceed the 2 priority levels defined here. In addition, we may be able to run multiple models at the same time by more efficiently utilizing the GPU available to us. This could be done through intelligent allocation of GPU memory, and this would allow us to run more jobs at the same time, without worrying about general virtual paging taking over and throwing off the priorities.

As such, the problem this paper attempts to address is three-fold.

1.1 Scheduling Models

The first issue arises in model scheduling. This problem has been approached before in ML including using ML to schedule ML (Aytug et al., 1994), but I personally believe that it's been approached very naively, as ultimately what we're trying to do is make a series of processes with unknown durations run on a limited amount of hardware. While many modern papers have taken a stab at this, I believe many are ignoring the fact that this problem has been solved for over half a century, in Operating Systems algorithms used to schedule processes for execution on the CPU (or whatever integrated chip is available). These algorithms include memory allocation algorithms, which allow for multiple programs to request shared memory from the hardware, as well as scheduling algorithms for process execution, some of which have provably optimal run times (Gomez et al., 2020). These algorithms can be adapted into the solution well, with the memory allocation algorithms used in operating systems (Irabashetti & Patil, 2014) being adapted for use with bin packing jobs into the amount of GPU memory available to the system. Likewise, Shortest Job First (SJF) algorithms can be adapted using their modification Weighted Shortest Job First to work with multi-rank models to get an accurate order for which to run jobs as they come into to system to minimize weighted turnaround time.

1.2 Estimations

This project requires a series of estimated parameters for most of the algorithms to work. Fortunately, most of these estimations don't have to be very accurate, just in the ballpark.

[1]University of Massachusetts, Amherst, USA. Correspondence to: Cole Smith <chsmith@umass.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020.

1.2.1 *Size Estimation*

One important thing that should be noted about the previous memory allocation solution is that for the algorithm to work, it needs to know the amount of memory usage that the model with consume during a given job. This problem has been explored previously such as Sohoni et al. (2019) and even Microsoft (Gao et al., 2020), but while many of their solutions are very accurate, most of them are fairly slow.

For this project, we need something a little faster, but it can be done with less accuracy. For this purpose, I use float estimation as inspired by the float reduction optimization shown in Ortiz et al. (2018). The purpose is essentially to calculate the maximum memory usage throughout the model by estimating the number of floats that will be loaded into memory at any given time. How this is done will be talked about later, but it tends to slightly overestimate but is significantly faster than the published solutions. This is an acceptable loss in this paper.

1.2.2 *Weight Estimation*

For weighted shortest job first to work, it requires a weight. Regular shortest job first simply takes the shortest job available to be run and runs it (see Figure 1). However, we want the jobs to be run in order of priority, not just duration. As such, we much come up with a method to weight the jobs accordingly.

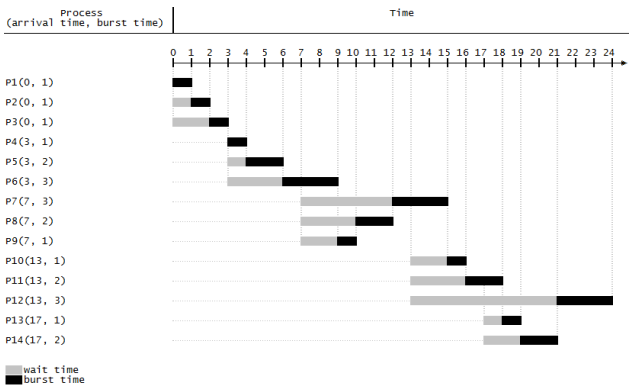

Figure 1. Shortest job first example run

This can actually be done at the same time as and in a very similar way to the memory usage estimation. By using the estimated number of flops required for a model to complete, along with the number of data points needing to be evaluated, we can determine relative lengths (because absolute time lengths don't matter, so we don't need to know the speed of the hardware). This lets us efficiently compute an approximate weight and send the completed models along to the job scheduler.

1.2.3 *Evaluation Metric*

This is more complicated than normal, because unlike the standard job scheduling algorithms we can't just look at turnaround time or even things like average response time or total time to completion. This is because we have priorities associated with the jobs, so if a high priority job is delayed, that means far worse performance than if a low priority job is delayed by the same amount. This evaluation metric is important, as we want to be able to accurately compare the different types of scheduling algorithms presented.

1.3 Framework Unification

Much of the research coming out around model scheduling recently has been extremely interesting and promising, but their custom code bases have made the usages so limited that they're nearly useless in industry as they can't hook into common libraries like Tensorflow or PyTorch. In this project, I aim to change that and make this scheduling framework work with any model and dataset written in PyTorch. This will allow this exact solution to be cloned from the Github repository and configured directly to work on a model I as the author don't even know exists. This versatility is something I feel has been lacking in many modern papers, and I'm not about to let my own become a victim of it as well.

2 Proposal vs Accomplishments

At the beginning of this project, I was planning to pour most of the time and effort into finer granularity interruptions of jobs (so that when a higher priority job came in, it could interrupt other jobs at certain intervals and run instead) but this turned out to be a waste of time (more on that later) as it was inefficient. As such, the goals of this project changed slightly. The items achieved:

- ✓Develop a unified framework to allow all PyTorch models and their datasets to be run through this system
- ✓Create a fair evaluation metric to score different scheduling methods on jobs with an unknown number of priorities
- ✓Design and implement a memory usage estimator
- ✓Design and implement a job length estimator
- ✓Invent a multithreaded job scheduler to handle an unknown number of priorities and jobs, allowing for online usage (dynamic job creation and allocation, allowing jobs to come in asynchronously) and implement

multiple scheduling algorithms for comparison and optimization purposes

- ✗ Implement fine-grain interruption for job scheduling

The choice to cut finer grain scheduling came from the discovery of runtimes during testing of smaller and smaller granularity values early on, as the runtimes actually bottomed out much earlier than layer granularity. This will be talked about more in implementation.

3 IMPLEMENTATION

3.1 Intent and Limitations

This project is designed to be one of potentially many steps in a machine learning pipeline. As such, it is modular and accepts widely variable input. As constructed, the system is designed to accept json-formatted jobs, which was done so that jobs could be sent using common HTTP requests, which allows the system to adapt to implementation scale with very little bottleneck. As it is currently constructed, the system can be configured to run on machine clusters with one scheduler per cluster. Interestingly, this system can be used upstream of itself, allowing it to schedule which clusters are assigned which jobs by using the default fifo queuing system. This allows the system to scale reasonably well, but with slightly more work it could definitely achieve much lower overhead and work on larger machines.

Each scheduler can only handle a single GPU in it's current configuration, but this is good enough for testing and for most industry application (self driving cars and intelligent security systems), as those applications are typically running on smaller systems with a single dedicated GPU. It is worth noting that I did not bother having the system scan the GPU to determine maximum available memory, as I felt it should be user-specifiable instead. This means that this system can be assigned a lose subset of the available resources and still optimize well.

3.2 The Algorithms

The system has five different scheduling algorithms:

1. **FIFO:** Poorly named, this algorithm actually runs the jobs in order of highest priority to lowest priority, only one job at a time, and with no care for length. This implementation is incredibly naive and slow, as it is effectively singlethreaded (there are actually two threads, the scheduler and the execution thread, but only one ever runs at one time). This implementation is designed as a naive baseline, as it's what is typically implemented by default for priority queues.
2. **GPU:** This algorithm simply starts each new job on its own thread simultaneously, with no regard for the memory or resources available. While this might sound awful at first, this allows the GPU's kernel-level scheduler to schedule each job's execution without outside interference (as basically everything else I do in this project just interferes with the driver's native schedule). My initial estimate is that this would result in an optimal total turnaround time as there is no overhead other than the kernel-level scheduling from the driver (which will be very small). This design though does not make use of the priorities, as all the jobs are effectively scheduled at the instruction level by the driver, but the driver does not know about the priorities associated with each thread (Kato et al., 2011). This solution is heavily impacted by the speed of the hard drive in a given system, as the scheduler will load all the models into memory. For a standard application, this will greatly exceed the amount of available memory in the system, and thus the number of virtual memory pages will begin to outnumber the true number of pages, meaning much of it will be automatically written to disk. All that included though, this acts as a more realistic baseline, as it is multithreaded.
3. **wsjfJ:** Confusing abbreviation aside, this algorithm (Weight Shortest Job First - Job) is the job-level granularity version of the shortest job first. This implementation does not have job interruption, as it would "interrupt" after a job has already been completed and thus not really be an interruption. This system also uses the bin packing memory usage algorithms, which allow for us to generally avoid the virtual paging issue, but this could still happen if the user sets the available memory to greater than the total memory in the GPU, or if a single job has a memory usage exceeding that limit. That's why the limit specified earlier is still a "soft limit" as if a single job's requirements exceed the memory available, the algorithm will intentionally go over and allow the CPU to handle the virtual pages as there's really no other way to do it other than implementing virtual paging manually, which would do the same thing but be slower because it's running on an emulation layer inside python's virtual machine interpreter.
4. **wsjfI:** As the algorithm that showed there was no point in pursuing finer grain granularity levels, this one is definitely the most customizable. This algorithm uses the same scheduling as the above weighted shortest job first - job, but as the final letter being swapped to 'I' indicates, this granularity level is now the Individual data point inference is being run on. This allows for the tightest level of interruption available in this system, and obviously allows for interruption as well. This

means that if a higher priority job comes along while lower priority jobs are running, the system will stop as many as necessary for the high priority job to run immediately. As it turns out, allowing this level of granularity checking causes the system to slow down drastically, which will be talked about in the evaluation section. Something to note with this algorithm and the one below is that due to their interruption, they can cause way more models to be loaded in memory than would typically be scheduled and thus we get the virtual paging issue again where hard drive speed matters.

5. **wsjfB:** Like the last algorithm, this is another weighted shortest job first. The granularity here is the batch size. This was chosen as an option because pausing in between batches means that nothing except the model itself actually has to be loaded into memory, as the next batch will be retrieved from the data loader once the thread has resumed. This means that the virtual paging issue described above is not nearly as bad, as there will never be data points loaded into memory when halting the thread's execution.

3.3 Memory Estimation

The memory estimation code is actually quite short, and fairly simple. This is good though, as something fast is beneficial to the system, and it doesn't have to be amazingly accurate. This part of the project is modular, meaning it can be removed from the pipeline and replaced with another estimation algorithm if different requirements are needed. In the same way, this part can be used standalone for other project implementations.

The program takes the incoming job json, and extracts the model and the dataset file path from it. Using the dimensions of the input, it simulates a run through the layers of the PyTorch model only by looking at the layer type, number of inputs, and number of outputs. This allows it to avoid actually computing the values, which could be an expensive operation (and could, in the worst case of an enormous model or extremely limited hardware, crash the machine). It simulates python's garbage collector and uses that information to estimate how many floats would be in memory at each stage of the model's forward pass. Once it's run through, it takes the maximum number of floats in memory, multiplies by the size of a float on the given machine (32 or 64 bits) and uses that product to estimate the memory used by the job in megabytes. The other part done here is the weighting. This stage modifies the priority of the job to instead use the estimated run time of the job (determined by the total number of flops). This allows the priority to include information about both length and priority. This information is then added back to the json, which is then sent to the next stage in the pipeline.

Algorithm 1 Weighted Turnaround Time

```
Input: data jobs
Initialize max = 0.
for job in jobs do
  if job["priority"] > max then
    Set max = job["priority"].
  end if
end for
for job in jobs do
  job["weight"] = (1 − job["priority"])
  job["weight"]* = job["turnaround"]
end for
```

It's worth noting that the pipeline stages would have to be adjusted manually, as it currently uses files to pass json information, but could easily be adjusted to http requests or any other form of json sending and receiving. It's just that I had to choose something when first implementing it, and http has a high overhead for a local machine pipeline. The ability to change to http would allow it to be used in cloud pipelines however, which is especially important given that this stage in the pipeline is not resource intensive, and can thus be run on systems with very low processing power, including serverless systems like AWS Lambda or Azure Functions (Wang et al., 2018).

3.4 Evaluation Metric

To evaluate the performance of these models, the weight must be included as part of the calculation. If regular turnaround time were used, we would consider the loss of a delayed high priority job and a delayed low priority job as exactly the same. As such, the evaluation metrics here are two-fold: the total time taken to run all jobs in the provided simulation, as well as this new metric. The new metric is obtained by finding the highest priority in the simulation, then for each job subtracting the job's actual priority from that, then multiplying the result by the job's turnaround time (see Algorithm 1).

These results are then summed to find the total weighted turnaround time for all the jobs combined.

4 RESULTS

4.1 Experimental Setup

This experiment was performed by generating random jobs using combinations of PyTorch's included pretrained models as well as PyTorch's included datasets. This required swapping the input layer of many of the models just to make it work, which in turn screwed up the pre-trained weights

Algorithm	Total Time (s)	Weighted Total Turnaround Time (s)
FIFO	101.7062490	903.7751508
GPU	36.0974123	677.6040406
wsjfJ	35.8711956	574.9272339
wsjfB	34.9923828	618.3685517
wsjfI	95.8428006	1372.3827059

Table 1. Results from each algorithm

(as many of these models were being used on datasets they were not designed for) thus tanking the accuracy of the evaluations. This was acceptable though as we are not interested in the actual performance of the models as this is just a bunch of randomly generated jobs while real jobs would be trained correctly for their input data. All algorithms were given the same set of asynchronous jobs, which (for testing purposes) were set to arrive in two asynchronous batches, to test the algorithm's ability to handle online, live jobs.

4.2 Experimental Results and Analysis

After running the test on all the models fifty times then averaging, the results were collected and put in Table 1.

Figure 2. FIFO gpu usage

These results were interesting, and had some unforeseen trends. Unsurprisingly, FIFO was in dead last in total time due to it's single-threaded nature. As it was unable to fully use the GPU at any point, this is very much expected. Figure 2 shows the GPU utilization of FIFO. We can see not very tall peaks, when it even starts to peak at all. It also has a low score on Weighted Total Turnaround Time (WTTT) which is not surprising for a very similar reason.

Total time tanks once we begin using multithreading, as seen with the GPU tests. Total time actually remains largely the same, only minor variations due to overhead of the scheduling. Interestingly, the drivers had a larger overhead for shuffling jobs, resulting in a slightly longer total run time. The WTTT is still quite large, which again makes sense as it's not prioritizing high priority jobs over low priority.

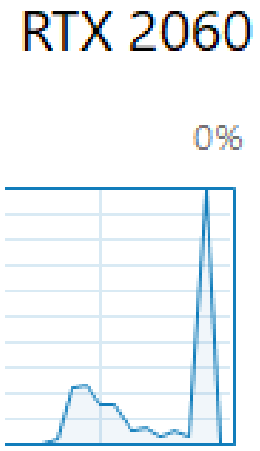

Figure 3. GPU gpu usage

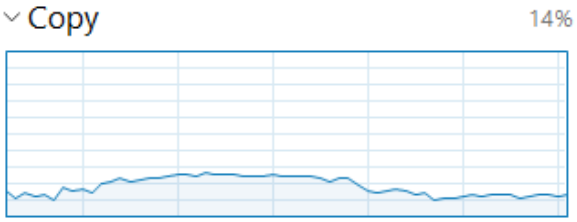

Figure 4. wsjfJ gpu memory usage

These results show that, as it is taking longer for high priority jobs to finish. It's GPU utilization can be seen in figure 3.

wsjfJ was surprisingly the best algorithm I ran. The total time is slightly smaller, which makes sense as the GPU scheduler has to do nearly no work because all the jobs are scheduled once and never have to be stopped like they did in the GPU implementation. This also stopped the virtual paging issues discussed earlier about the GPU implementation. The WTTT is lower than the GPU implementation, which isn't surprising either. This is because wsjfJ takes priority into account, and thus would run higher priority jobs way before lower priority jobs as best as it can. The utilization here meets the cap and stays at it, showing steadier GPU memory usage (which is good). It can be seen in figure 4.

What was surprising is that wsjfJ lost to wsjfB in total time, but beat it in WTTT. I was expecting the opposite to happen, as there is more overhead with wsjfB so I expected the total time to be slightly higher. I truly have no idea why wsjfB took less time, I imagine it has something to do with the drivers for the GPU thinking it is a higher priority process (*process*, not thread. All threads are in

the same process and have the same process id) due to the activity and thus running it slightly more. As I was doing the testing on my local laptop, I couldn't entirely isolate the GPU from other tasks, thus the GPU's internal process priority level may have had a slight effect (but not much). Incredibly, wsjfB lost to WTTT, which was surprising given that wsjfB can interrupt running processes. However, as I mentioned before the virtual paging can start to kick in due to jobs being interrupted and effectively cached in memory. I imagine this hurt the WTTT because it would take some time for jobs to get re-started once stopped.

This prior theory really comes into play with wsjfI, showing that the effect of having many jobs cached in virtual memory can cause large performance hits. One of the other things I think happened here is that this model's threads have to check if they have been locked far more frequently than the batched or job versions. As this is an atomic operation, I imagine the threads began to get very bottle necked, as they were spending most of their time polling these atomic signals (which were unique for each thread). As such, I believe this design did not accurately test this setup, and recommend that future work revisit this with a better implementation of the multi-threading that doesn't require checking atomic signals.

5 Conclusion

Based on the results, I can conclude two major things: The first one, interestingly, is that the native GPU thread scheduler actually does a really good job when priority is not an issue. I think that a lot of comparisons in newer papers are done to single-threaded implementations, but I now encourage those authors to check their baselines against a naive multithreaded solution. The GPU drivers definitely have improved a lot in the past few years, and seeing a nearly 66% reduction in execution time really shows that the native scheduler is getting up there in performance.

More relevant to this actual project, the results clearly show that more research needs to be done while looking at what solutions have already been built. Ignoring my moral high horse for making a usable project instead of a proof of concept, the results of simply implementing 50 year-old algorithms in a modern problem speak for themselves that there is a lot of potential in older works that may well go ignored because many ML researchers believe themselves to be too "front line" to look at older solutions. After all, an improvement capable of cutting the WTTT nearly in half is certainly nothing to scoff at. I also think it's worth noting that the algorithmic complexity of having a variable number of priority levels is only marginally higher than having only two, and as such this can clearly become a standard, allowing for greater customization in use.

Lastly, this project shows there is clearly still room to improve on ML systems pipelines. This project is only one piece of a pipeline, and is compatible with *any* pytorch model, including one that has already gone through various other optimization steps. Likewise, this scheduler can be adapted for many other purposes too. The point here is that this project is a fairly critical piece of the pipeline which I believe has been missing from many infrastructure projects. Sometimes, we have to look to the past to grasp the future.

6 Future Work

Speaking of: In the future, work should be looked at for implementing finer granularity levels without instruction-heavy atomic actions. This could be done at a lower level of abstraction, or just with a better architecture for thread execution. In addition, adding the ability to schedule to multiple GPUs should be done. This wouldn't be very difficult, as the bin packing algorithm would just attempt to front load every GPU (fill the first GPU up to the limit, then the second, etc). This would allow this system to scale nearly infinitely, with the strategy I described earlier of using the scheduling module apart from the model running module and sending the jobs via HTTP request to each cluster. This could allow datacenter-level jobs to be scheduled for nearly optimal execution.

References

Aytug, H., Bhattacharyya, S., Koehler, G., and Snowdon, J. A review of machine learning in scheduling. *Engineering Management, IEEE Transactions on*, 41:165 – 171, 06 1994. doi: 10.1109/17.293383.

Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating gpu memory consumption of deep learning models. Technical Report MSR-TR-2020-20, Microsoft, May 2020. URL https://www.microsoft.com/en-us/research/publication/estimating-gpu-memory-consumption-of-deep-learnin

Gomez, R. R., Bermudez, C., Cachero, V., Rabang, E., Baquirin, R., Fronda, R. J., and Bayani, E. End to end dynamic round robin (e-edrr) scheduling algorithm utilizing shortest job first analysis. pp. 218–221, 01 2020. doi: 10.1145/3383845.3383869.

Irabashetti, P. and Patil, N. Dynamic memory allocation: Role in memory management. *International Journal of Current Engineering and Technology*, Vol.4:531–535, 03 2014.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y.

Timegraph: Gpu scheduling for real-time multi-tasking environments. pp. 17, 01 2011.

Ortiz, M., Cristal, A., Ayguadé, E., and Casas, M. Low-precision floating-point schemes for neural network training. 04 2018.

Sohoni, N. S., Aberger, C. R., Leszczynski, M., Zhang, J., and Ré, C. Low-memory neural network training: A technical report. *CoRR*, abs/1904.10631, 2019. URL http://arxiv.org/abs/1904.10631.

Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 133–146, Boston, MA, July 2018. USENIX Association. ISBN ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.