Multi-Architecture Parallelism in Deep Learning Compilers

CS 243 Project Report

Weifan Jiang
Harvard University

Joel Runevic
Harvard University

Eric Zhang
Harvard University

Abstract

Neural networks have different layers with varied compute, memory, and throughput landscapes. We investigate the performance of running deep learning training and inference workloads on *multi-architecture* compute resources, specifically layer parallelism that schedules optimized compute tasks between CPU and GPU. This differs from past work that focused on optimizing computation graphs on a single accelerator, through graph rewriting and distributed data techniques. Through experiments, we find that augmenting traditional GPU-based training with some operations running on CPUs can be more cost-effective while maintaining comparable runtime, and we describe a general framework for comparing the effectiveness of layer parallelism between different hardware resources. Our initial results demonstrate that *multi-architecture parallelism* may be more resource-efficient for some real-world machine learning problems.

1 Introduction

The recent trend toward larger models like massively scaled transformers [6, 7] (in terms of number of FLOPS and parameters) comes a necessity for systems that support training and evaluating such models on multi-GPU and multi-node settings. Machine learning frameworks and compilers are flexible enough to allow for a huge variety of large-scale parallelization approaches. However, the trend towards massive models also makes training and inference expensive.

The most basic type of parallelism in distributed training is data parallelism, where a copy of the model resides on each device and operates on difference batches in parallel. However, this does not scale to large networks, and more fine-tuned parallelism strategies within operators (SPMD) and between operators (model parallelism) in a computation graph are necessary to make large language models tractable. These types of parallelism make evaluation possible when the parameters do not fit in GPU memory, and they also reduce GPU memory bandwidth requirements for some models.

However, parallelism is not straightforward. Oftentimes the best parallelism method depends on a huge number of application and model-specific quantities, including model architecture, hardware memory and compute throughput, network topology, parameter counts, quantization, floating-point precision, and architecture. For example, it has been known since before 2014 that convolutional neural networks are best parallelized using multiple different data and model-parallel methods depending on the layer [14]. Also, state-of-the-art language models such as Megatron [25] and PaLM [3] have handcrafted model parallelism solutions that allow them to train multi-billion parameter systems at scale, which would ordinarily be impossible because the parameters do not fit in memory on a single node.

This paper aims to address a new topic in parallelism that fills a gap in the large space of possible strategies: *multi-architecture parallelism*. We believe that it is crucial to understand whether parts of models can run more effectively between heterogeneous devices of different types, including CPUs, GPUs, or a mixture of both. This is important because some deep learning layers are more amenable to running on different hardware architectures. For example, GPUs are standard for inference, but a substantial amount of deep learning inference in production is done on CPUs for cost efficiency reasons [10]. Also, matrix multiplication is far more optimized than any other compute kernel in most specialized GPUs due to targeted support, and it's possible that more exotic kernels would have less speedup on GPUs relative to the cost. It stands to reason that GPU clusters, which tend to have significant co-located CPU resources, may be more effectively utilized if certain computations (such as embedding) could be run on CPUs under a unified parallelism system.

Therefore, the focus of this paper is on developing a framework for describing and measuring the performance of multi-architecture parallelism on training clusters with both GPU and CPU resources, producing benchmarks for different real-world use cases, and finally describing why parallelism may lead to efficiency gains in heterogeneous data centers.

2 Model Parallelism Background

Due to its inherent difficulty and importance, there has been a surge of interest in automated parallelism strategies. Recent work such as Pipedream[1] [20], FlexFlow [18], GShard [16], and Alpa [31] have taken this flexible parallelism approach to its logical extreme, exploring automated, profile-driven methods using numerical optimization to develop parallelization strategies that are much more intricate and complex than any manually-designed system.

There are two broad classes of ways to use model parallelism between multiple devices (GPUs and TPUs), whether they are shared by a single node or split between many nodes in a cluster. The first class is *intra-layer parallelism* that synchronizes computations within a single operation, but which partitions the operation along some tensor axis to split the computations. The second class is *inter-layer parallelism*, where the computation graph is actually partitioned between different devices, lending itself to model or pipeline parallelism algorithms. The JAX [5] framework based on XLA has basic support for SPMD-based (single program, multiple data) intra-layer parallelism through compiler passes and an intermediate representation that shards computation. However, this requires manual effort and is not automated. It also has no built-in support for inter-layer parallelism.

To remedy these issues, Alpa [31] introduced an automatic parallelization layer on top of JAX that inspects the computation graph and applies compiler optimization passes with numerical optimization to automatically produce intra-layer and inter-layer parallelism splits. This resolved the issue of making large models easier to train on clusters, making efficient use of devices, and without having to implement complex manual splitting and scheduling algorithms.

However, Alpa only works on a single device type and does not support training on heterogeneous clusters, with different devices. Other systems in this category also focus on optimization for a single device class, typically profiling each layer once and relying on assumptions like runtime being approximately the same across successive executions of the same computation [18].

3 Computation Graphs

For our initial experiments,[2] we use JAX due to its very flexible automatic differentiation approach (based on JVP/VJP transformations) and optimization passes supported by an advanced JIT compiler (also shared with TensorFlow XLA and TorchScript) [17]. Some prior work like FlexFlow [18] pioneered exploring automatic parallelization in a controlled environment, based on heuristics and optimization algorithms.

[1] In this paper, when we mention "model parallelism" we mean a broad family of approaches that includes pipelining, which be more network-efficient or compute-saturating than plain, blocking parallelism.

[2] Code at https://github.com/ekzhang/archax.

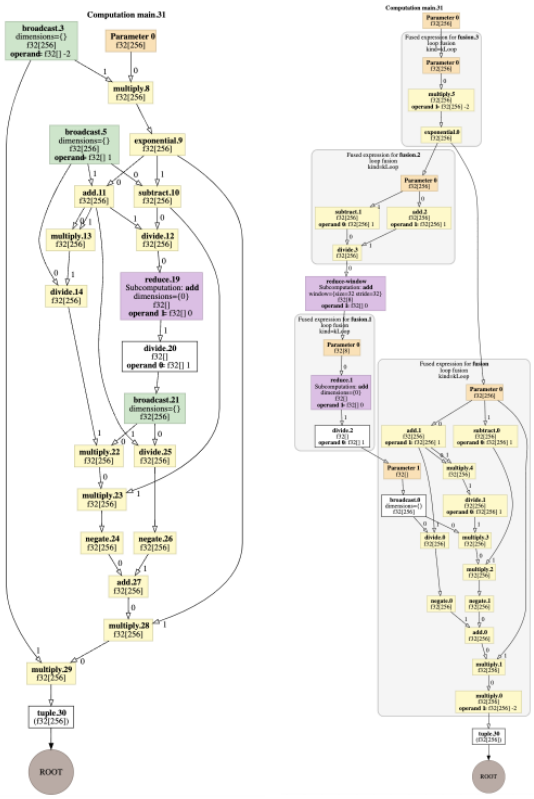

Figure 1: HLO computation for $\frac{\mathrm{d}}{\mathrm{d}\mathbf{x}} \log(\tanh(\mathbf{x}).\mathrm{sum}())$ where $\mathbf{x} \in \mathbb{R}^{256}$, before (left) and after (right) optimization passes.

However, these methods need a great deal of *control* over the model specification, which limits practical usability and integration with existing deep learning frameworks.

3.1 Inspecting HLO IR

To better understand the computational graph that JAX generates, consider the lifetime of a simple program that uses the JAX framework. We extracted intermediate representations for some numeric Python functions in JAX, before and after applying combinators like the `jax.grad` gradient operator. The compilation pipeline first traces executing Python code with a dummy input to produce a "Jaxpr", then translates the Jaxpr into HLO IR, feeds it into the HLO-MLIR pipeline for optimization, and finally generates low-level CPU or GPU instructions through LLVM.

These HLO IR programs can be inspected using their Protocol Buffers definition, which stores the syntax tree as blocks of operations, as well as built-in static analysis metrics like total memory reads. An example of this is shown in Fig. 1. The graph on the right, after optimization, has more instructions, but the instructions are in large chunks that form efficient subcomputations with *loop fusion*, reducing the total number of memory accesses.

Model	GB Access	GFLOPs
log(tanh([2^{16}]).sum())	0.012 / 0.002	0.001 / 0.001
MobileNetV1 (256x256)	1.6 / 1.3	4.5 / 15.0
ResNet50 (256x256)	4.0 / 2.7	31.0 / 31.5
ResNet50 (16x256x256)	57.3 / 33.8	496.9 / 504.4

Figure 2: Memory and compute profiles for neural networks in JAX with backpropagation, as HLO IR. The first number is before optimization, and the second is after optimization.

Information from these kinds of computation graphs can be programmatically analyzed and intercepted, which therefore makes it a fertile place to experiment with implementing different automatic parallelization or scheduling patterns of our own design. In particular, the optimization passes in HLO are done before any architecture-specific lowering (CPU, GPU, or TPU), so the graph can be split at different locations after initial optimization passes and before lowering HLO IR to machine code.

3.2 Memory Bandwidth and FLOPs

The profiles for memory bandwidth and floating-point operations for several real-world neural networks in JAX is shown in Fig. 2. These quantitative metrics were obtained from the compiler on a computation graph during training, with backward propagation for gradients. Two numbers are shown, the amount of memory accesses in MB, as well as the number of floating-point operations in millions.

The first model is a simple function that simply computes `log(sum(tanh(x)))` on a vector of length 65536, while the next two models are convolutional neural networks operating on 256x256 images. We can see from these numbers that the optimization step typically reduces the memory accesses substantially, though it sometimes increases the FLOPs as a result. These tradeoffs are made because of the small input size, and we can also directly observe how they affect the plan with a batch size of 16.

4 Heterogeneous-Hardware Benchmarking

To accelerate ML workflow (e.g. data embedding, model training, inference), operators leverage specialized hardware accelerators such as GPUs, TPUs [12], and FPGAs. Even though such accelerators are widely adopted in industry, we lack a systematic understanding of their performance and cost efficiency for different tasks/model/data. Though recent works are able to approximate the most optimal parallelization strategy, they do not factor in whether the *underlying* hardware itself is optimal for the ML job. We aim to rectify this gap through our implementation of TBS: TensorFlow Benchmarking System.[3] Specifically, we hope to be able to answer the following questions:

- How does hardware choice impact the training and embedding efficiency for different models and layers? In particular, are GPUs always more runtime-efficient than CPUs?
- Does the impact of other configurable settings (e.g. batch size) remain consistent across heterogeneous hardware choices?

The user of TBS provides a hardware-independent model structure definition and a dataset that is compatible with the model. TBS then measures each layer's runtime on CPU and GPU. The measurement is done repeatedly for statistical significance. Per user's request, TBS can also vary configurations such as batch size for fine-tuned measurement result. In addition, with user-provided cost-per-unit-time information, the runtime measurements can easily be converted to cost measurements if the user of the system is interested in such metrics.

Currently, TBS is only capable of measuring layer execution time, instead of training time. There exists certain non-trivial programming challenges that we have not yet solved at this time to measure per-layer training time, such as measuring forward/backward propagation. Refer to Section 7 for a more detailed discussion regarding this. However, we argue that measuring layer execution time still leads to insights on heterogeneous-hardware model training:

Measuring non-trainable layers. TensorFlow provides non-trainable pre-processing layers to convert various-formatted inputs to numerical feature vectors that can be directly processed by computational dense layers during training. For example, `TextVectorization` layer converts each word to a numerical index in the vocabulary. For image processing, non-trainable layers are still popular to perform input augmentation tasks: for example, `RandomFlip`, `RandomDrop`, and `RandomRotation` layers are used to randomly alternate the training image to regularize CNNs. TBS is capable of measuring the runtime/cost efficiency for such non-trainable layers on different hardwares and with different configurations. The insights derived from measurement results (i.e. whether `RandomFlip` is more cost-efficient on CPUs/GPUs) can facilitate the development of more optimal parallelized training strategies that are *hardware-aware*.

Measuring learned embeddings. Learned embedding is also widely used in fields such as transfer learning. For example, instead of using images directly as model input (where each feature value corresponds to a pixel's intensity), we can apply a separately learned dense layer to embed the image. Even if the learned embedding was initially trained for a different purpose, it can still convert the image to a meaningful

[3] We implemented this stage in TensorFlow rather than JAX due to time constraints, given the availability of many prebuilt models in TensorFlow.

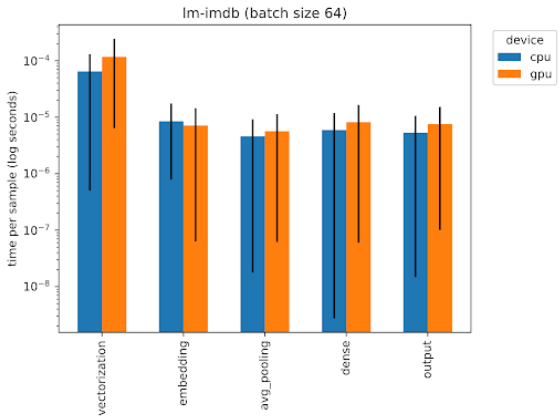

Figure 3: The average time to process a sample on a per-layer granularity for a language model (ML)

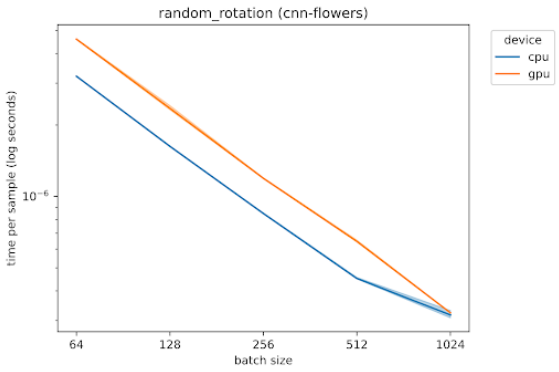

Figure 4: The average time to perform a random rotation within the preprocessing step of a CNN pipeline. Note that the CPU performance increase scales relatively linearly with batch size.

representation that can accelerate future training. The TBS system can measure the inference time for dense layers, which is equivalent to measuring learned embedding.

The code for TBS is available on GitHub[4]. We provide four examples of differently structured models (CNN, RNN, Language Modeling, LSTM) that TBS is able to evaluate, to demonstrate our system.

In Section 5 and Section 6, we walk through some example time and cost measurements obtained using TBS, and correspondingly derive insights that can facilitate efficient heterogeneous-hardware training in the future.

5 Analysis of Processing Time

We ran several experiments in order to motivate our benchmarking system. Such experiments involved running a Language Model (LM), [23], a Recurrent Neural Network (RNN), [8], and a Convolutional Neural Network (CNN) [22] - the latter of which is discussed in Section 6. As we are primarily interested in embedding and preprocessing layers, we analysed our TensorFlow Benchmarking System on ML inference applications with a preprocessing step. Refer to Section 7 for a discussion on ML training.

5.1 Time Analysis: Example with Discussion

As aforementioned, we developed an LM using the TensorFlow [27] framework. In particular, the LM contains a vectorization and an embedding layer. We utilized our benchmarking system to investigate the throughput of different layers within the LM and gained insightful results.

One key observation, as can be supported by Fig. 3, is the fact that, on a per-layer basis, GPU accelerators are *not* the most effective hardware architecture. In particular, for our LM-based example, our TBS system clearly shows that processing time is minimized when *all* layers are run on a CPU rather than a conventional accelerator, such as a GPU. We also note a similar observation for our CNN example (which is discussed in more detail in Section 6.1). In particular, we note that certain preprocessing layers are faster on CPU than GPU, with the relationship scaling relatively linearly even as batch sizes are increased. This is especially interesting considering the fact that recent works [11] have shown that low batch sizes are often the bottleneck in GPU inference. Refer to Section 6.2 for a more in-depth analysis of batch size variation within TBS.

The motivation for a user-friendly benchmarking tool, such as TBS, is clear after such examples. As models and use-cases become more and more specialized, simple "rules of thumb" for embedding and preprocessing tasks are no longer effective. The flexibility of TBS ensures that users are able to quickly profile their models, irrespective of the use-case, and verify in real-time what the best hardware architecture is for their respective use-case. Since TBS is hardware-agnostic, it is equally possible for users to benchmark their models using TBS on TPUs or any other specialized computer architecture, as long as the respective architecture can interoperate with the TensorFlow library [1].

6 Cost Analysis

We also investigated the various cost-performance tradeoffs associated with utilizing different hardware architectures for the respective preprocessing and embedding layers. Previous works focused on running embedding on different computer architectures are limited [2] [21]. Moreover, to the best of our knowledge, a comprehensive analysis of the costs associated with preprocessing and embedding layers on different computer architectures is even more limited (for example,

[4] https://github.com/weifanjiang/tbs

we could not find a relevant paper to cite, which shows the limited quantity of research on this particular topic). This is not surprising as research within the field of ML has mainly been focused on increasing the compute of models, as well as reducing any network bandwidth bottlenecks associated with the training of such ever-larger ML models.

However, inevitably, beyond the realm of academic research, there are many cost-performance tradeoffs to consider, particularly for startups that have limited funding, smaller research teams, and general programmers that are interested in developing ML models. Considering research has shown that the main reason why many technology startups fail is insufficient funding to continue [24], it is important that programmers are able to contextualize the throughput that they can achieve for their ML models for a given computer architecture with respect to how expensive it is to actually *run* the ML model on the respective architecture (i.e. a particular GPU instance on AWS); so that the user can ultimately make the best decision for their respective use-case(s).

To do so, we ensured that our command-line TensorFlow Benchmarking System can take an extra argument as input that expresses an approximation for the per-hour cost for the underlying architecture that the ML model is deployed on. We note that, considering how prevalent the use of various cloud computing services are becoming for the training and deployment of ML models [15] [4] [19], such an approximation is very easy for the user to obtain. Whilst experiments are insightful, and we comment on some of our findings subsequently, the flexibility of TBS is the most important aspect of our work as inevitably the cost-effectiveness of any model layer depends on various factors, such as the dataset used for the respective training/inference, the architectures being compared, and so on. The goal of our system, after all, is to empower users to make more cost-effective decisions for their respective use cases.

6.1 Cost-Throughput Tradeoffs: Example with Discussion

We now present a discussion of results for one of the many experiments that we conducted. This particular experiment was selected as it presents an interesting discussion of why viewing performance throughput alone does not convey enough information for a user. We consider the following use-case that involves running a Convolutional Neural Network (CNN) [22] on Google TensorFlow's flowers dataset [27]. We intentionally ensured that the respective CNN has various preprocessing layers, such as RandomRotation, RandomZoom, and Rescaling [26]. The CPU used was the AMD EPYC 7302 with 16 cores and the GPU used was NVIDIA A100 40GB. Cost estimates were derived by matching the respective hardware to various cloud providers that offer identical (or very similar) computing resources [28]. Since this analysis was focused on the preprocessing layers, we chose to focus on

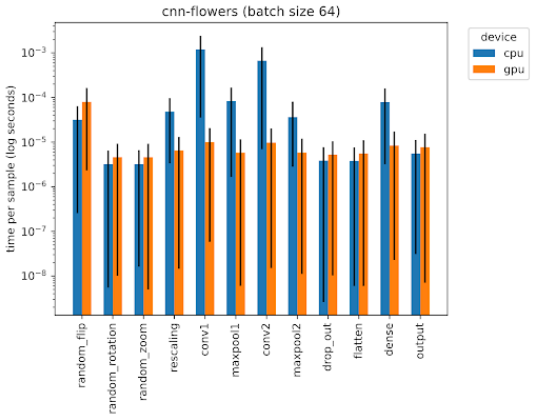

Figure 5: A graph demonstrating how quickly on average (in log seconds) a dataset sample is processed through the various layers of the CNN

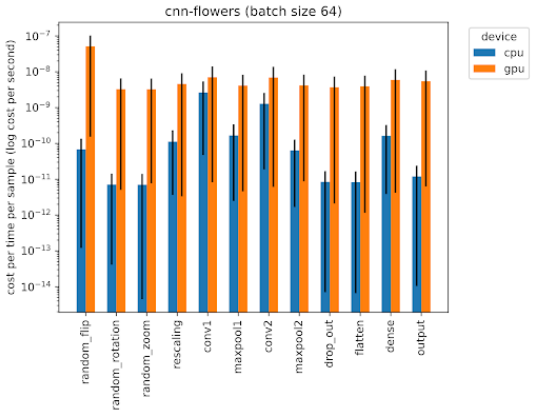

Figure 6: A graph demonstrating the average cost of processing one sample (in log scale) through the various layers of the CNN

ML inference for this particular example.

From Fig. 5, it is clear that the linear algebra intensive layers (conv1, conv2 etc) are faster on GPU accelerators relative to CPU, which is inline with general practice. However, it is worth noting that the preprocessing layers are slightly faster on CPU. Considering the average time per sample alone, a user may conclude that it is advantageous to run the respective use-case on GPU alone as the preprocessing speedups using CPU relative to GPU are relatively marginal and thus do not warrant significant code change (i.e. the removal of the preprocessing layers and creating a separate preprocessing pipelining instead and/or partitioning the preprocessing layers on a different architecture altogether). However, when factoring in the cost associated with processing one sample, as shown in Fig. 6, the user will likely come to a different conclusion altogether.

With this information, a user will more likely to conclude

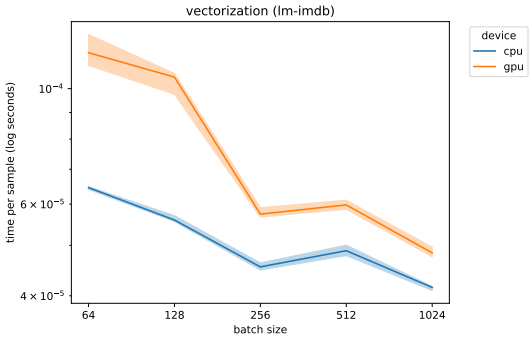

Figure 7: Median per-sample time for `TextVectorization` layer on CPU/GPU using different batch sizes. The shaded region is between 25 and 75 percentiles.

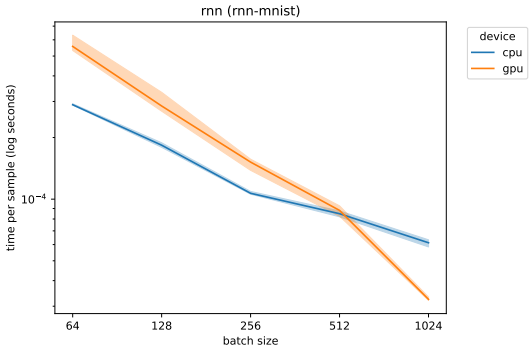

Figure 8: Median per-sample time for `RNN` layer on CPU/GPU using different batch sizes. The shaded region is between 25 and 75 percentiles.

that the creation of such a CPU-based preprocessing pipeline is *effective* in terms of cost-performance tradeoff as the preprocessing layers include non-trivial cost improvements with no sacrifice in throughput when run on CPU versus GPU. We hope this example provided further motivation for our TensorFlow benchmarking system, especially with respect to our decision to include cost analysis also.

6.2 Varying other configurations in TBS

TBS is able to perform measurement varying configurations other than hardware choices. Figure 7 shows an example of measuring the per-sample runtime of `TextVectorization` on CPUs/GPUs with different batch size configuration. In general, as batch size increases, the per-sample runtime overhead decreases. We notice that GPU benefits more from increased batch size.

Figure 8 shows the runtime for `RNN` layer in TensorFlow on CPU/GPU with varying batch sizes. We notice that for batch size larger than 512, it is more time efficient to run `RNN` layers on GPUs rather than CPUs. This observations shows that changes in configuration would lead to different optimal hardware choices. TBS can help users to benchmark model's runtime and cost efficiency on heterogeneous hardwares under different configurations, to find the configuration-specific optimal hardware choices.

7 Future Work

Inevitably, we were not able to investigate everything within multi-architecture parallelism and preprocessing. As can be seen in our research code, we had implemented an LSTM network [30] using time-series data as well as a graph neural network (GNN) [29]. Our TensorFlow benchmark system utilizes the popular Python object serialization library Pickle [9] in order to deserialize ML models for benchmarking. However, not all models are serializable this way, such as our LSTM and GNN implementations, and a better system should use more advanced formats like `cloudpickle` or ONNX.

In addition to this, our current benchmarking system has the assumption that the input model was developed using the TensorFlow library. Considering the prevalence of TensorFlow [1], this is a fairly weak assumption; however, future work could include the integration of our benchmarking system for other model frameworks.

The ideal framework for this would be JAX. If TBS were a longer-term project, the most obvious direction would be to integrate our research with the JAX [5] compiler to generate automatic multi-architecture parallelism. We faced two major challenges when attempting this that we could not overcome in the time allocated for the project.

Firstly, novel contributions to autonomous parallelism is inevitably very difficult, especially considering the fact that it is currently still a cutting-edge research area largely spearheaded by a single research group at UC Berkeley. Even then existing solutions, such as Alpa [31], are still rigid in the sense that they require a user to solely use JAX [5], which, though a great step forward in the right direction, has inevitably not received nearly the same usage as TensorFlow [1]. A more feasible addition to TBS could be a simple recommendation algorithm that outputs which layers or preprocessing tasks should be run on specific computer architectures. However, considering TBS already provides visualizations of such metrics, careful thought would have to be taken to avoid any redundant features.

Secondly, we primarily focused on embedding and preprocessing tasks considering we were motivated by the lack of research on multi-architecture embedding and the like. That being said, it would be inevitably useful if TBS could be expanded to support profiling of other training pipelines of ML models (beyond just trainable embedding/preprocessing layers) as this would serve an even greater purpose for the ML community as a whole. The major obstacle to this, however, is the development of an accurate approximation for processing times on a layer-by-layer granularity. Research within this

space is limited with some recent works [13] investigating the feasibility of using DNNs to approximate how long parts of ML models take to process samples.

Despite these technical challenges, we believe that our compiler explorations and related benchmark results show some that there is promise in multi-architecture parallelism in deep learning. We hope that the ideas in this report present a path forward for exploring this design paradigm. Time will tell if machine learning runtimes of the future will seamlessly parallelize devices with multiple architectures in tandem, and we eagerly await.

Acknowledgements

We would like to thank the staff of CS 243 at Harvard for a wonderful graduate seminar on computer networks applied to emerging problems in the field of machine learning. We are especially grateful to Prof. Minlan Yu for her guidance and advice throughout all stages of this project.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. Aljundi, T. Akyildiz, and K. Kaya. Boosting graph embedding on a single gpu. *IEEE Transactions on Parallel & Distributed Systems*, 33(11):3092–3105, nov 2022.

[3] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[4] Ekaba Bisong. Building machine learning and deep learning models on google cloud platform, 2019.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[8] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61–98, 2015.

[9] Python Documentation. Pickle python object serialization. https://docs.python.org/3/library/pickle.html.

[10] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[11] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference, 2019.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham,

Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[13] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models. *2018 IEEE International Conference on Big Data (Big Data)*, 2018.

[14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[15] Mandeep Kumar. An incorporation of artificial intelligence capabilities in cloud computing. *International Journal Of Engineering And Computer Science*, 2016.

[16] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[17] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[18] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[19] Sumit Mund. *Microsoft Azure Machine Learning*. Packt Publishing, 2015.

[20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[21] Vinh Nguyen, Ann Spencer, Joey Wang, and Jianbing Dong. Accelerating embedding with the hugectr tensorflow embedding plugin, 2021.

[22] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[23] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2021.

[24] Bednár R. and Tarišková N. Indicators of startup failure. https://stumejournals.com/journals/i4/2017/5/238, Jan 2017.

[25] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[26] TensorFlow. Module: Tf.keras.layers, tensorflow v2.11.0. https://www.tensorflow.org/api_docs/python/tf/keras/layers.

[27] TensorFlow. Tf_flowers - tensorflow datasets. https://www.tensorflow.org/datasets/catalog/tf_flowers.

[28] Vultr. Cloud compute. https://www.vultr.com/pricing/#cloud-gpu.

[29] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[30] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, 31(7):1235–1270, 2019.

[31] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.