OPPA: OPTIMIZING PARALLELISM FOR LANGUAGE MODEL TRAINING

Apivich Hemachandra, Yizhan Han, See-Kiong Ng, Bryan Kian Hsiang Low
Department of Computer Science, National University of Singapore, Singapore 117417

ABSTRACT

Training of modern large neural networks (NNs) is often done in parallel across multiple GPUs. While there are existing parallel training frameworks which easily allow NN training using multi-dimensional parallelism, the challenge remains in optimizing the balance between size of the parallelism dimensions, and in tuning the hyperparameters within each parallelism dimension. Due to a large number of possible parallelism configurations (PCs) for a given training process, it is infeasible to perform exhaustive search over all candidates. Even though there exists PC optimization methods, they either rely on an approximate cost model which may be inaccurate and hardware-specific, or on a large number of NN training trials on different PCs each which are expensive to evaluate. To overcome these issues, we present OPPA, which combines Bayesian optimization with prior knowledge in the form of a parallelism-informed prior belief, to obtain an optimal PC using minimal number of NN training trials. We demonstrate that OPPA is able to more efficiently find an optimal PC for training transformers when compared to methods used in existing parallel training frameworks.

1 INTRODUCTION

Modern advances in deep learning have arisen from the ability to scale neural networks (NNs) to larger sizes. In natural language processing, for example, transformer-based models (Vaswani et al., 2017; Devlin et al., 2019), large language models (LLMs) (Touvron et al., 2023; OpenAI et al., 2024) and multimodal models (Radford et al., 2021; Liu et al., 2023), composed of millions or even billions of parameters, have shown tremendous success in tasks such as text classification, text generation, and language understanding. Due to their sizes, these large NNs cannot be trained on a single GPU. To scale up the training process, it is necessary to distribute the NN training workload across many machines and parallelize the training process. Different parallelism dimensions for NN training have been proposed, including data parallelism (Rajbhandari et al., 2020; Zhao et al., 2023), pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019), and tensor parallelism (Shoeybi et al., 2020). Frameworks for *multi-dimensional parallelism* have also been proposed (Shoeybi et al., 2020; Rasley et al., 2020; Li et al., 2023), to combine different dimensions of parallelism together.

In NN training, to fully utilize the given hardware and reduce the computation time, we typically would like to maximize the *throughput* of training, or the number of training steps processed in a given time. The throughput will depend on the selected *parallelism configurations* (PCs), which in large-scale parallel training frameworks (Kuchaiev et al., 2019; Rasley et al., 2020; Shoeybi et al., 2020; Li et al., 2023), typically consist of discrete or continuous parameters which control the size of each parallelism dimension, and specifics on how each parallelism dimension is executed. In practice, it is *difficult to exactly quantify how the choice of PC affects the training throughput*, as it would depend on the model architecture, the training data, the compute hardware, or the exact implementations of the parallel training framework. In this sense, the throughput could be treated as a *black-box function* whose exact form cannot be recovered.

In an ideal scenario, to consider all possible training factors, the best PC would be chosen by conduct training trials using real hardware on all possible PCs. Unfortunately, due to the large number of possible PCs, *performing an exhaustive search would be extremely inefficient*. To circumvent this,

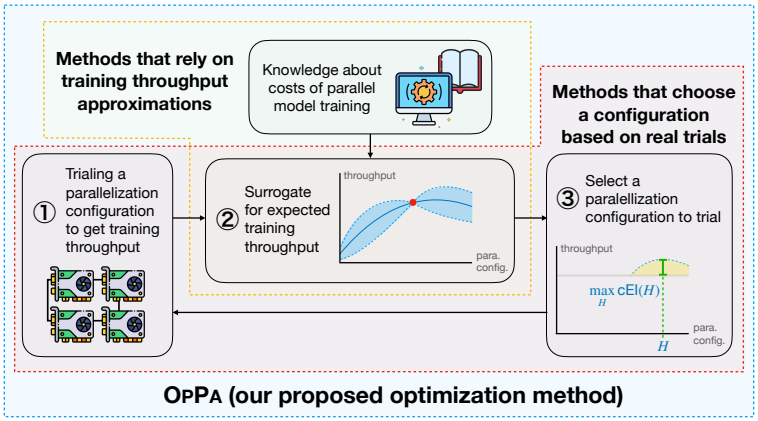

Figure 1: Main idea of OPPA algorithm. OPPA combines adaptive selection techniques with domain knowledge on parallel model training in order to find the parallelism configuration which achieves the highest training throughput.

existing parallel training frameworks such as DEEPSPEED[1] (Rasley et al., 2020) and NEMO[2] use methods to select a subset of candidate PCs to trial. However, this is still inefficient due to simplistic optimization algorithms which are *unable to adapt to known training throughputs*, therefore still need many trials to find a good candidate. While it may be possible approximate for the training throughput via an analytical expression (Zheng et al., 2022; Li et al., 2022; Zhang et al., 2024), *these approximations require strong assumptions* on the compute hardware and the specific parallelized NN training implementation, and as a result may not capture all nuances of a parallel training instance. Despite this, they may still provide useful information, even if not fully accurate, which may allow us to filter out some suboptimal PCs which should not be trialed. In order to efficiently select the PC that achieves the best throughput, we therefore need the ability to adaptively select potentially good PCs to trial, while also filtering out poor candidates based on the trialed PCs and on any existing domain knowledge. Due to the black-box nature of the throughput, it may be possible to use black-box optimization methods such as Bayesian optimization (Gelbart et al., 2014; Frazier, 2018), which would allow incorporation of information from training trials ran with existing domain knowledge, into performing adaptive search of good PCs.

In this paper, we introduce OPTIMIZER FOR PARALLELISM CONFIGURATIONS, abbreviated as OPPA, which optimizes the PC for training a NN through trials selected via Bayesian optimization and informed by prior knowledge on parallelized NN training. The main idea of OPPA is presented in Figure 1. In Section 2, we first formulate the problem of finding the optimal PC as a black-box function optimization with black-box constraints. In Section 3, we introduce a framework based on constrained Bayesian optimization, which performs minimal NN training trials in order to determine the most efficient PC for distributed NN training, and briefly discuss how we encode domain knowledge into OPPA via a parallelism-informed prior belief, to better model the effects of the PCs on the training speed and memory usage. We finally empirically demonstrate the effectiveness of OPPA in Section 4, showing that OPPA with a parallelism-informed prior belief can more quickly find a good PC for training transformers compared to existing methods, and compared to when no prior belief is used. Related works can be found in Appendix A.

2 PROBLEM SETUP

In this section, we describe the problem setup we wish to consider. For our problem setting, we consider a parallelism configuration (PC), visualized in Figure 2, which contains a list of tunable hyperparameters found in typical parallel training frameworks, and controls various aspects of parallelized NN training. A subset of hyperparameters in a PC determine the *size of each parallelism dimension*. In our paper, we consider 3D parallelism where we use `dp`, `tp`, and `pp`, to indicate

[1] https://www.deepspeed.ai/tutorials/autotuning/

[2] https://docs.nvidia.com/nemo-framework/user-guide/latest/usingautoconfigurator.html

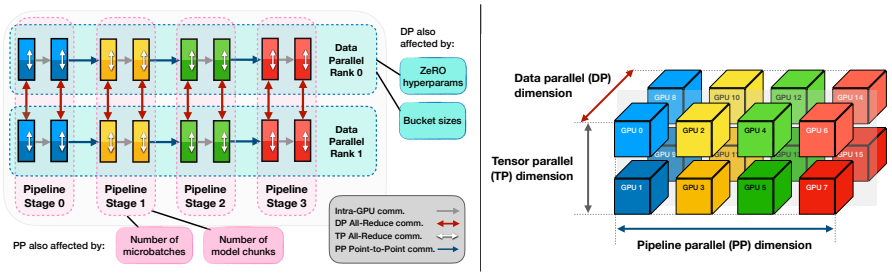

Figure 2: *Left:* Visualization of the parallelization configuration, including the parallelism dimensions and the hyperparameters affecting their specific implementations. *Right:* Visualization of GPU allocation for 3D parallelism according to the dimension sizes.

the size of the data, tensor, and pipeline parallelism dimensions respectively. We assume that their product $\texttt{dp} \cdot \texttt{tp} \cdot \texttt{pp}$ is equal to the number of GPUs. The remaining hyperparameters determine the *specific implementations of each parallelism dimensions*. These may include hyperparameters of the ZeRO optimizer which controls the DP execution, or the number of microbatches and model chunks which control the PP execution. These may also extend to other hyperparameters which are specific to different parallel training frameworks. We discuss these hyperparameters further in Appendix C.

We consider a space of feasible PCs denoted by $\mathcal{H}$, and would like to find the optimal PC $H \in \mathcal{H}$ which results in the highest training throughput (i.e., most training steps per time), while still not exceeding the available GPU memory M_0. Mathematically, we aim to solve the optimization problem

$$\underset{H \in \mathcal{H}}{\text{maximize}}\ \mathcal{R}(H) \quad \text{s.t.} \quad \mathcal{M}(H) \leq M_0 \tag{1}$$

where $\mathcal{R}(H)$ is the training throughput when using PC H and $\mathcal{M}(H)$ is the maximum memory usage on a GPU. In optimizing the PC, we also consider several design factors.

- A Since $\mathcal{R}$ and $\mathcal{M}$ are dependent on many factors which may be difficult to model or even known exactly, *they are assumed to be black-box functions*. Despite this, the values $\mathcal{R}(H)$ and $\mathcal{M}(H)$ for a PC H *can be queried* by trialing the PC on real training hardware. This motivates us to use a surrogate that is flexible enough to model black-box functions given the trials, and able to quantify the uncertainty in its prediction.
- B Even though a PC can be trialed on real hardware, *running a single trial incurs a high cost*. This is especially true with suboptimal PCs since the same number of training steps on a suboptimal PC would require more time to execute[3]. We therefore should ensure that the trials ran are carefully selected in order to avoid trialing suboptimal PCs.
- C Despite the black-box nature of $\mathcal{R}$ and $\mathcal{M}$, we assume that *they can be approximated based on some domain knowledge* which should not be treated as accurate and blindly trusted. This motivates us to construct a surrogate which incorporates some prior knowledge that can guide the optimization process and reduce the number of training trials required.

3 Method

In this section, we describe how we can apply BO to the optimization problem as described in the previous section. As shown in Figure 1, in the optimization process, we alternate between three steps; ① performing real NN training trials to obtain the training throughput and maximum memory usage for a selected PC, ② modeling the training throughput and maximum memory usage using the observed data via a GP, and ③ finding the best PC to query in the proceeding round using BO.

Step ①: Trialing a PC. We first perform actual model training on a PC for m training steps, and collect the time required to perform a training step given by $t_{H,1}, \ldots, t_{H,m}$. The measured throughput is given by $r_H = \sum_{i=1}^{m} (t_{H,i})^{-1}$. To measure the maximum allocated GPU memory

[3]An exception to this observation is when a PC results in an out-of-memory error where the time to obtain this result is short since no training needs to (or even can) be ran.

throughout the training steps, using CUDA-based PYTORCH, this can be easily done using the `torch.cuda.max_memory_allocated()` function, which records the maximum allocated GPU memory achieved m_H at any point during training.

Step ②: Construct a GP surrogate based on observed data. Next, we use a GP to model the training throughput data $\mathcal{Y}_\mathcal{R} = \{r_H\}_{H\in\mathcal{X}}$ and maximum memory $\mathcal{Y}_\mathcal{M} = \{m_H\}_{H\in\mathcal{X}}$ collected in ①. Since GPs require weaker assumptions of the true function, and can capture *uncertainty* of its predictions, they are a good choice as a surrogate for black-box functions such as $\mathcal{R}$ and $\mathcal{M}$ as required in A. We consider a GP family with a constant mean prior, i.e., $\mu_{\text{prior}}(H; c) = c$. For the kernel function, we first embed the PC H via an embedding $e : \mathcal{H} \to \mathbb{R}^p$ which first applies the logarithm operation only on continuous hyperparameters, then concatenates the values in H into a p-dimensional vector. For our GP, we use the Matern kernel with $\nu = 5/2$. Given the observed throughput $(\mathcal{X}, \mathcal{Y}_\mathcal{R})$ and maximum memory $(\mathcal{X}, \mathcal{Y}_\mathcal{M})$ values, we can separately find the corresponding optimal prior mean constants and lengthscales by maximizing the marginal log-likelihood of the observation as usually done in GP modeling (Rasmussen & Williams, 2006). The predicted throughput $\hat{r}_H$ and maximum memory $\hat{m}_H$ is then obtained by performing GP regression.

Step ③: Selecting the next PC to run. Finally, the next PC to trial is chosen based on the GPs modeled in ② using BO. The next PC $H_{\text{next}} \in \mathcal{H} \setminus \mathcal{X}$ to trial is chosen to be the PC which maximizes the constrained expected improvement (cEI) (Gelbart et al., 2014), which we describe in Appendix D.2. The cEI considers a balance between *exploration* of PCs which have not been trialed, and *exploitation* of PCs which are similar to those with already high throughputs. With this balance, the BO iteration is able to try enough PCs to construct a reasonable surrogate for the functions, while utilizing the remaining computational resources to trial good PC candidates for to achieve optimal training throughput. This allows the optimization to be more guided and more efficient, satisfying B.

Parallelism-Informed Prior Belief. While we can use a normal GP to directly model the throughput and maximum GPU memory in ②, it is inefficient since the surrogate has to be constructed solely using the observed data, meaning more trials are needed to be ran for a good surrogate. Meanwhile, if domain knowledge can be added into the modeling, it could allow for additional inductive bias which would filter out suboptimal PCs more easily, reducing the number of trials that would be needed. Instead of using a simple GP prior proposed in ②, we introduce a *parallelism-informed prior belief*, consisting of a new prior mean and kernel, which embeds domain knowledge on the *approximate* throughput and maximum memory usage, which we discuss in detail in Appendix D.3. We only require the prior mean to be reasonable estimates of the throughput and maximum memory, such that their values can eventually be more easily modeled by the GP. This design choice is in accordance with C where the incorporated domain knowledge is only used as an inductive bias for the surrogate model, and is still accurate enough for ③. We present the pseudocode for OPPA in Appendix D.6.

4 EXPERIMENTS

In this section, we present the results for OPPA when used to find the optimal PC for training transformer models on multi-GPU systems. The full experimental setup is detailed in Appendix E. In our results, we present the best obtained throughput (in training steps per second) that is obtained versus the time the optimization has been ran for, rather than versus the number of PC that has been trialed, to also account for time required to conduct the training trials.

Ablation on smaller models. We first consider training smaller transformer models on a single machine with 8 GPUs. We consider the BERT model (Devlin et al., 2019) for a classification task. In Figure 3a, we present the results for the ran methods. We see that the methods which use BO outperforms both non-adaptive random sampling, and also adaptive method which uses XGBOOST as its surrogate. Furthermore, OPPA, which uses both BO with a parallelism-informed prior, is able to achieve better performances than using standard BO, suggesting that the incorporation of domain knowledge can aid in finding the optimal PC. We also consider the scenario of training a GPT-2 model (Radford et al., 2019) on the same hardware configuration. In this scenario, we use a larger training batch size, which will result in out-of-memory errors in many poorly-selected PCs. Despite this, we again see that OPPA is able to outperform all other benchmarks, and is able to more quickly arrive at an optimal PC compared to using standard BO.

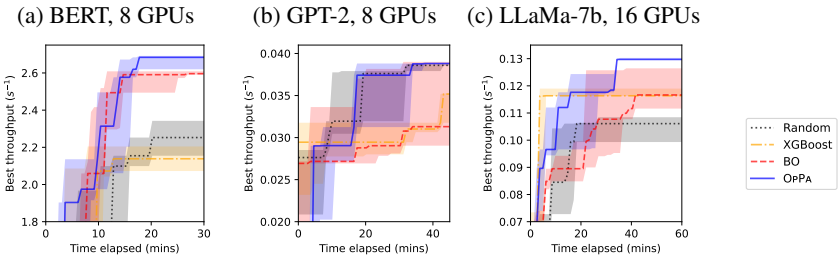

Figure 3: Results of the best obtained throughput (higher is better) plotted against the amount of time each algorithm has been run for. The lines represent the median value of the best obtained throughput across five trials, while the error bar represent the quantile values.

Accuracy of surrogate model. In Figure 4a we compare the throughput predicted by the GP with the actual values. We see that even after a few trials, the predictions made by our GP already correlate well with the actual values. As we progress, the prediction also becomes more accurate, especially among PCs with high throughput. This is because the BO process would prioritize selecting those PCs with higher throughput anyway. We demonstrate this further in Appendix F.2, showing that incorporating a parallelism-informed prior belief allows better modeling of the throughput compared to a standard prior.

We note that the accuracy of the surrogate is not the only reason OPPA performs well. In Figure 4b, we plot the results obtained when BO is not used, but rather only the surrogate function fitted on trials from randomly selected PCs. We see that even though the general throughput values are better modeled, the optimal PC is not reflected by the prediction, which would have led in selecting a suboptimal PC. This is likely because the surrogate focuses on a good general model rather than one which predicts the optimum well. This shows a necessity of BO in focusing trials and prioritize modeling the good PCs.

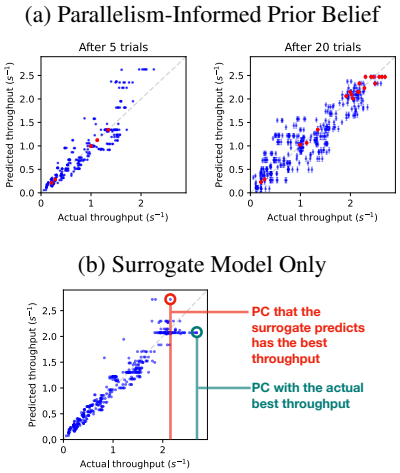

Figure 4: Predicted throughputs versus the measured throughputs for the BERT example. In Figure 4a, the red points are the trialed PCs.

LLaMa model on larger hardware. We also optimized the PC for training LLaMa-7b model (Touvron et al., 2023) on 16 GPUs distributed across 4 machines. Note that in the prior belief, we do not explicitly assume that the GPUs are on multiple hosts, therefore not accounting for communication speeds across hosts. From the results in Figure 3c, we see that OPPA still consistently outperform other benchmarks. This demonstrates that *despite using an imperfect model* to approximate the training throughput, it is still sufficient information that can be corrected for by the uncertainty as captured by the GP and result in a model good enough to learn an optimal PC. This also demonstrates that especially in the larger search space with more possible parallelism dimension size combinations, OPPA still shows impressive scalability not present in the other algorithms.

5 CONCLUSION

We have presented OPPA, which uses constrained Bayesian optimization techniques with a parallelism-informed prior distribution to efficiently optimize the parallelization strategy which can achieve the best training throughput. While OPPA is currently implemented specifically for one parallel training framework, OPPA can easily be adapted to other frameworks as well due to the minimal assumptions on the implementations of the training parallelism and the simplicity to extend to other hyperparameters. We also believe that the parallelism-informed prior belief could be embedded with more prior knowledge on specific implementation or training of specific NN architectures, which should boost OPPA even further.

REFERENCES

Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. Maximizing Parallelism in Distributed Training for Huge Neural Networks, May 2021. URL http://arxiv.org/abs/2105.14450. arXiv:2105.14450.

Lichang Chen, Jiuhai Chen, Tom Goldstein, Heng Huang, and Tianyi Zhou. InstructZero: Efficient Instruction Optimization for Black-Box Large Language Models, August 2023. URL http://arxiv.org/abs/2306.03082. arXiv:2306.03082 [cs].

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, August 2016. doi: 10.1145/2939672.2939785. URL http://arxiv.org/abs/1603.02754. arXiv:1603.02754 [cs].

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. URL http://arxiv.org/abs/1810.04805. arXiv:1810.04805 [cs].

David Duvenaud, Hannes Nickisch, and Carl Edward Rasmussen. Additive Gaussian Processes, December 2011. URL http://arxiv.org/abs/1112.4394. arXiv:1112.4394 [stat].

Peter I. Frazier. A Tutorial on Bayesian Optimization, July 2018. URL http://arxiv.org/abs/1807.02811. arXiv:1807.02811.

Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams. Bayesian Optimization with Unknown Constraints, March 2014. URL http://arxiv.org/abs/1403.5607. arXiv:1403.5607 [stat].

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 10, pp. 103–112. Curran Associates Inc., Red Hook, NY, USA, December 2019.

Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, 13(4):455–492, 1998. ISSN 09255001. doi: 10.1023/A:1008306431147. URL http://link.springer.com/10.1023/A:1008306431147.

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, and Jonathan M. Cohen. NeMo: a toolkit for building AI applications using Neural Modules, September 2019. URL http://arxiv.org/abs/1909.09577. arXiv:1909.09577 [cs].

Bowen Lei, Tanner Quinn Kirk, Anirban Bhattacharya, Debdeep Pati, Xiaoning Qian, Raymundo Arroyave, and Bani K. Mallick. Bayesian optimization with adaptive surrogate models for automated experimental design. *npj Computational Materials*, 7(1):1–12, December 2021. ISSN 2057-3960. doi: 10.1038/s41524-021-00662-x. URL https://www.nature.com/articles/s41524-021-00662-x. Publisher: Nature Publishing Group.

Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. AMP: Automatically Finding Model Parallel Strategies with Heterogeneity Awareness, October 2022. URL http://arxiv.org/abs/2210.07297. arXiv:2210.07297 [cs].

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, August 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415530. URL https://dl.acm.org/doi/10.14778/3415478.3415530.

Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training, October 2023. URL http://arxiv.org/abs/2110.14883. arXiv:2110.14883.

Xiaoqiang Lin, Zhaoxuan Wu, Zhongxiang Dai, Wenyang Hu, Yao Shu, See-Kiong Ng, Patrick Jaillet, and Bryan Kian Hsiang Low. Use Your INSTINCT: INSTruction optimization for LLMs usIng Neural bandits Coupled with Transformers, June 2024a. URL http://arxiv.org/abs/2310.02905. arXiv:2310.02905 [cs].

Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. {nnScaler}: {Constraint-Guided} Parallelization Plan Generation for Deep Learning Training. pp. 347–363, 2024b. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning, December 2023. URL http://arxiv.org/abs/2304.08485. arXiv:2304.08485 [cs].

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*, SOSP '19, pp. 1–15, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Lukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Lukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John

Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. GPT-4 Technical Report, March 2024. URL http://arxiv.org/abs/2303.08774. arXiv:2303.08774 [cs].

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision, February 2021. URL http://arxiv.org/abs/2103.00020. arXiv:2103.00020 [cs].

Tom Rainforth, Adam Foster, Desi R. Ivanova, and Freddie Bickford Smith. Modern Bayesian Experimental Design, February 2023. URL http://arxiv.org/abs/2302.14545. arXiv:2302.14545 [cs, stat].

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, May 2020. URL http://arxiv.org/abs/1910.02054. arXiv:1910.02054.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, Virtual Event CA USA, August 2020. ACM. ISBN 978-1-4503-7998-4. doi: 10.1145/3394486.3406703. URL https://dl.acm.org/doi/10.1145/3394486.3406703.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 2006. ISBN 978-0-262-18253-9. OCLC: ocm61285753.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020. URL http://arxiv.org/abs/1909.08053. arXiv:1909.08053.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms, August 2012. URL http://arxiv.org/abs/1206.2944. arXiv:1206.2944 [stat].

Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. *IEEE Transactions on Information Theory*, 58(5):3250–3265, May 2012. ISSN 0018-9448, 1557-9654. doi: 10.1109/TIT.2011.2182033. URL http://arxiv.org/abs/0912.3995. arXiv:0912.3995 [cs].

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005. ISSN 1094-3420, 1741-2846. doi: 10.1177/1094342005051521.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee,

Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023. URL http://arxiv.org/abs/2307.09288. arXiv:2307.09288 [cs].

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections, September 2024. URL http://arxiv.org/abs/2312.05181. arXiv:2312.05181.

Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis, January 2024. URL http://arxiv.org/abs/2401.05965. arXiv:2401.05965.

Yichi Zhang, Daniel W. Apley, and Wei Chen. Bayesian Optimization for Materials Design with Mixed Quantitative and Qualitative Variables. *Scientific Reports*, 10(1):4924, March 2020. ISSN 2045-2322. doi: 10.1038/s41598-020-60652-9. URL https://www.nature.com/articles/s41598-020-60652-9. Publisher: Nature Publishing Group.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, September 2023. URL http://arxiv.org/abs/2304.11277. arXiv:2304.11277.

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, June 2022. URL http://arxiv.org/abs/2201.12023. arXiv:2201.12023.

A Background and Related Works

In this section, we provide an overview of current techniques of parallelized model training on multiple GPUs, and how optimal parallelism configurations are currently found. We also provide a brief overview of Bayesian optimization, which is a technique we will use in our proposed method.

A.1 Distributed Deep Learning

In order to effectively train large neural networks (NNs), practitioners resort to different techniques to parallelize the training workload across multiple GPUs. Different parallelism dimensions divide the computation workload in different ways, which affect the amount of computation per GPU, amount of communication between each GPU, and the amount of memory required in each GPU. Here, we discuss the existing parallelism dimensions, the hyperparameters involved, and how they differ from each other.

Data parallelism. The most basic type of training parallelism is data parallelism (DP) (Li et al., 2020) where a batch of training data is split into shards and distributed among each devices. These shards are then fed into the local replicas of the models, before the parameter updates from each devices are synchronized. While simple and often the fastest, the naive DP approach requires replication of the model on each devices, which takes up additional storage on each machines. Several methods have since been proposed to perform DP with sharded models, including the Zero Redundancy Optimizer (ZeRO) introduced by Rajbhandari et al. (2020) in the DeepSpeed package, and Fully-Sharded Data Parallel (FSDP) introduced by Zhao et al. (2023). While these frameworks allow for efficient DP implementations, their effectiveness can still heavily depend on the choice of hyperparameters. For example, ZeRO involves three different stages of optimization which chooses whether the optimizer states, the model gradients or the model parameters are sharded between each GPUs. The choice of sharded items affect the amount of data that has to be stored in each GPUs and communicated across GPUs, which in turn affects the throughput of the training and the memory usage in each GPUs.

Tensor parallelism. Another method for scaling operation is tensor parallelism (TP) where individual tensors are sharded across multiple devices, so that the matrix multiplication operations are instead done in a distributed manner, allowing these operations to scale to larger sizes than otherwise that would fit on a single GPU. TP initially involved splitting a tensor along a single dimension (Shoeybi et al., 2020), however has since also incorporated sharding tensors across multiple dimensions as well (Bian et al., 2021).

Pipeline parallelism. In pipeline parallelism (PP) (Huang et al., 2019; Narayanan et al., 2019) we instead partition the model along its execution pipeline, with each model partitions running synchronously with microbatches of data. The gradients are accumulated for each microbatches and updated at the end of each training step. By sharding the model and training data into smaller chunks, the GPU memory required at any one time becomes lower, allowing for the training of larger models at the cost of more sequential operation rounds and higher cost of communication between each GPUs. The tradeoff between training speed and maximum memory usage can be further controlled based on the size of microbatches and the number of model chunks.

Multi-dimensional parallelism. Many frameworks (Shoeybi et al., 2020; Rasley et al., 2020; Li et al., 2023) have also since been developed to allow DP, TP, and PP to be used togethter in the same training process. These frameworks provide simple interfaces for the users to specify the desired *parallelism configuration* (PC), which include the size of each parallelism dimension along with other hyperparameters specific to each parallelism dimension. These frameworks would then automatically handle model and data sharding and execute the parallelized training pipeline as per the specifications. These frameworks may also manage training on multi-node setting, heterogeneous setting, and even varying resources during runtime.

While these frameworks allow practitioners to easily specify a PC for training, *selecting the best PC to achieve the most efficient training is often difficult*, since the optimal PC will often depend on factors such as the GPU specifications, communication bandwidth of the GPU devices, the specific architecture of the NN or the training data (Li et al., 2023; Wagenländer et al., 2024; Lin et al., 2024b). For example DP alone is ineffective for large models or when using large batch sizes, since the replicated model parameters may likely cause an out-of-memory error on the GPUs. Meanwhile,

PP is less effective on smaller models, as communication costs between each pipeline stages may dominate the actual computation of the fragmented pipeline.

A.2 PARALLELISM CONFIGURATION OPTIMIZATION

To find the optimal PC which achieves the highest efficiency, we can attempt to trial all possible PCs on the actual training hardware in order to determine which one results in the highest training throughput. However, this is prohibitively expensive since there can be a large number of possible PCs, and each trial would itself require computational resource and time which may be limited on real clusters.

To circumvent this, frameworks such as DEEPSPEED[4] (Rasley et al., 2020) and NEMO[5] (Kuchaiev et al., 2019) have implemented methods for automatic PC tuning based on running NN training trials for a few training steps on a number of PCs. The PCs trialed are often either selected non-adaptively (e.g., based on random selection), or adaptively based on a simple surrogate function. However, these methods are unable to *efficiently use the measured throughput of trialed PCs* to model the true throughput scores and perform informed optimization, meaning they still require a large number of training trials to obtain a good PC.

Since running training trials may be expensive, we may consider constructing a surrogate model to approximate the computation and communication costs for different parallelism strategies (Zheng et al., 2022; Li et al., 2022; Zhang et al., 2024), which would allow us to use domain knowledge to filter out suboptimal PCs while performing fewer trials, or even by not trialing any PCs at all. This methods, however, would *require an implicit assumption that the surrogate of the true training throughput is correct*, which may not always be possible because surrogates may be unable to fully capture the nuances of practical parallel training implementations. Furthermore, a fixed surrogate model would not be easily extendable to new hyperparameters or parallelism nuances which may arise in a PC, which is important especially with the ever-growing parallelism training literature.

A.3 BAYESIAN OPTIMIZATION

In order to more efficiently select a PC to trial and to optimize for, we will utilize Bayesian optimization (BO) (Frazier, 2018). BO aims to maximize some black-box function $f : \mathcal{X} \to \mathbb{R}$ which is often expensive to query and whose derivative is unknown. The black-box function is modeled a Gaussian process (GP) (Rasmussen & Williams, 2006), which is characterized by a prior mean $\mu_{\text{prior}}(\cdot)$ and a kernel function $k(\cdot, \cdot)$. Given a set of observations, we can perform Bayesian inference in order to obtain a posterior GP, which is made up of a posterior mean and posterior covariance, encoding the expected value and the uncertainty of the function respectively. With the posterior GP, the BO procedure selects an input that maximizes some acquisition function, such as the expected improvement (Jones et al., 1998) and the upper confidence bound (Srinivas et al., 2012). These acquisition functions balance between exploring unique inputs that have not been queried, and exploiting inputs likely to have high function values. We provide a more technical overview of GP modeling and BO in Appendix B.

BO is a widely used to optimize black-box functions which have no closed form and are expensive to evaluate. This include a wide range of problems in, e.g., experimental design (Lei et al., 2021; Rainforth et al., 2023), material design (Zhang et al., 2020), and prompt optimization (Chen et al., 2023; Lin et al., 2024a). More relevant to our work, BO is also commonly used for NN hyperparameter optimization to select the NN architecture that performs best in a given task (Snoek et al., 2012). Unlike in traditional NN hyperparameter optimization, however, the effects of the hyperparameters in a PC on the training throughput have better-defined mechanics (even if not completely known), which can at least be partially described based on domain knowledge. Modeling via a GP allows incorporation of such knowledge through a good choice of prior belief, which can reduce the number of trials required.

[4]https://www.deepspeed.ai/tutorials/autotuning/

[5]https://docs.nvidia.com/nemo-framework/user-guide/latest/usingautoconfigurator.html

B TECHNICAL PRIMER ON GAUSSIAN PROCESSES AND BAYESIAN OPTIMIZATION

In this section, we provide a technical overview of Gaussian process (GP) regression and on Bayesian optimization (BO). The contents are adapted from Rasmussen & Williams (2006); Frazier (2018).

A Gaussian process (GP) $\mathcal{GP}(\mu_{\text{prior}}, k)$ with prior mean μ_{prior} and kernel k is a random process where for any subset of input $\mathbf{X}$, its corresponding output is given by a normal distribution $f(\mathbf{X}) \sim \mathcal{N}\big(\mu_{\text{prior}}(\mathbf{X}), k(\mathbf{X}, \mathbf{X})\big)$. The prior mean $\mu_{\text{prior}}(x)$ describes the expected value of the random function $f(x)$ at a certain input, while the kernel function $k(x, x')$ roughly captures the covariance between $f(x)$ and $f(x')$.

Assume we have an unknown function f drawn from the GP. Given a set of observations $D = (\mathbf{X}, \mathbf{y}) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ where $y_i = f(x_i) + \epsilon_i$ are noisy observations of the true function with Gaussian noise $\epsilon_i \sim \mathcal{N}(0, s^2)$. Then, when performing Bayesian inference, we can express the posterior mean and covariance of the GP as

$$\mu(x) = k(x, \mathbf{X})\big(k(\mathbf{X}) + s^2 I\big)^{-1}(\mathbf{y} - \mu_{\text{prior}}(x)) \,, \tag{2}$$

$$\sigma^2(x) = k(x, x) - k(x, \mathbf{X})\big(k(\mathbf{X}) + s^2 I\big)^{-1} k(\mathbf{X}, x) \,. \tag{3}$$

In practice, the prior mean and kernel may have hyperparameters θ which specify what functions it is able to model. For example, many kernel functions include lengthscale values which govern how correlated the function output is when a certain input dimension changes. One method to find the optimal hyperparameters for the kernel is by finding the hyperparameter which maximizes the marginal log-likelihood, i.e.,

$$\theta^* = \arg\max_{\theta} \; \log p\big(\mathbf{y} | 0 \,, k(\mathbf{X}, \mathbf{X}; \theta) + s^2 I\big) \,. \tag{4}$$

In Bayesian optimization (BO), the goal is to find the maxima of the unknown function f. This function is black-box, and assumed to have no analytical form. To do so, we can learn more about f by querying it at different inputs, and perform Bayesian inference to update our belief on the unknown function.

Given the current observations $D_t = (\mathbf{X}_t, \mathbf{y}_t) = \{(x_1, y_1), \ldots, (x_t, y_t)\}$ in round t of data selection, GP regression can be performed to obtain a posterior mean μ_t and posterior variance σ_t^2. The next input to query x_{t+1} can be chosen as the input which maximizes some acquisition function. Examples of such acquisition function include the expected improvement

$$\text{EI}(x; D_t) = \mathbb{E}_{y' \sim \mathcal{N}(\mu_t(x), \sigma_t^2(x))}\big[\max(0, y' - \max_{y \in \mathbf{y}_t} y)\big] \tag{5}$$

or the upper confidence bound

$$\text{UCB}(x; D_t) = \mu_t(x) + \beta_t \sigma_t(x) \tag{6}$$

where $\beta_t > 0$ is a constant that may vary with t. In all of these acquisition functions, a tradeoff is performed between selecting inputs that the GP is uncertain about (i.e., with high $\sigma_t^2(x)$) to learn more about those unknown region, and selecting inputs in regions where the function value is known to be higher (i.e., with high $\mu_t(x)$).

C HYPERPARAMETERS CONSIDERED

Below, we list several hyperparameters which we include in our parallelism configuration.

- **Parallelism dimension sizes.** These include the degrees of DP, TP and PP, which are denoted as `dp`, `tp`, `pp` respectively.
- **Hyperparameters for DP.** These are hyperparameters which control the ZeRO optimizer and the bucket sizes for the amount of parameters to be synchronized in one round. The specific hyperparameters considered are ZeRO optimizer stage, the bucket size of the ZeRO optimizer and the bucket size for the DP process, which are denoted by `zs`, `zb`, and `dpb` respectively. Note that out of all the variables listed, `zs` is the only one which is a categorical variable.

- **Sharding degrees for PP.** These include the number of model chunks and the number of data microbatches, which are denoted by `mc` and `mb` respectively, and they control the amount of pipeline bubble which occurs, at the cost of more communication and sequential computation rounds.

D Detailed Discussion of OpPa

D.1 Training Time Results Collection

In practice, since we collect multiple time measurements for each step, we can also provide the variance of the throughput as well. Since we are mostly concerned with the mean throughput over all training steps, we therefore would be interested in the mean throughput fraom each training step, and the variance of the mean throughput. These two quantities are given by

$$r_H = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{t_{S,i}}, \quad \sigma^2_{r_H} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{t_{S,i}} - r_H\right)^2. \tag{7}$$

D.2 Criterion for Bayesian Optimization

In our work, we choose the next PC to query according to the constrained expected improvement (cEI) criterion (Gelbart et al., 2014), which extends the expected improvement (EI) criterion to the constrained setting and is given by

$$\text{cEI}(H;\mathcal{X}) = \mathbb{E}_{\hat{r}_H,\hat{m}_H}\left[\ \max(\hat{r}_H - r^*_{\mathcal{X}}, 0)\ \mathbb{1}[\hat{m}_H \le M_0]\ \right] \tag{8}$$

where distributions for $\hat{r}_H$ and $\hat{m}_H$ are from the GPs in ②, and $r^*_{\mathcal{X}} = \max_{H\in\mathcal{X}} r_H$ is the best throughput observed so far. Intuitively, cEI(H) is higher if H is likely to be feasible and its throughput likely to improve upon r^*.

D.3 Parallelism-Informed Prior Belief

As suggested in [C], instead of using a simple GP prior proposed in ②, we introduce a *parallelism-informed prior belief*, which embeds domain knowledge on the *approximate* throughput and maximum memory usage. To do so, we assume the true throughput $\mathcal{R}$ and maximum memory usage $\mathcal{M}$ to be decomposed to a known part that can be modeled, and an unknown part which cannot be modeled, i.e.,

$$\mathcal{R}(H) = \hat{\mathcal{R}}(H;\mathbf{r}) + f_{\mathcal{R}}(H)\ , \tag{9}$$

$$\mathcal{M}(H) = \hat{\mathcal{M}}(H;\mathbf{m}) + f_{\mathcal{M}}(H) \tag{10}$$

where $\hat{\mathcal{R}}$ and $\hat{\mathcal{M}}$ are based on (potentially imperfect) domain knowledge and (potentially unknown) hyperparameters $\mathbf{r}$ and $\mathbf{m}$, while $f_{\mathcal{R}}(H)$ and $f_{\mathcal{M}}(H)$ incorporate factors not captured by our surrogate. With this, we can model $\mathcal{R}$ and $\mathcal{M}$ with GPs whose prior means are $\hat{\mathcal{R}}$ and $\hat{\mathcal{M}}$ respectively, and some choice of kernel function which can model $f_{\mathcal{R}}(H)$ and $f_{\mathcal{M}}(H)$.

To justify this, consider Equation (9). We assume $\hat{\mathcal{R}}(H;\mathbf{r})$ is deterministic and can be constructed by our prior knowledge. Since $f_{\mathcal{R}}(H)$ is unknown, we can assume a prior belief on it given by the GP $\mathcal{GP}(0,k)$ with zero mean and some kernel function. This means that for any H, we are assuming $f_{\mathcal{R}}(H) \sim \mathcal{N}(0, k(H,H))$. Based on the properties of Gaussian random variables, this also means $\mathcal{R}(H) \sim \mathcal{N}(\hat{\mathcal{R}}(H), k(H,H))$. The argument can be extended to the multivariate setting (to consider the multivariate normal distribution) to show that assuming $f_{\mathcal{R}} \sim \mathcal{GP}(0,k)$ implies we are assuming $\mathcal{R}(H) \sim \mathcal{GP}(\hat{R}, k)$. This justifies the use of a GP to model the throughput, and why the domain knowledge should be incorporated into the prior mean function. The same argument can be used to justify the choice to model $\mathcal{M}$ as well.

As we introduce the prior mean functions used, we will see that we *do not require $\hat{\mathcal{R}}$ and $\hat{\mathcal{M}}$ to be completely accurate*. Instead, we only require them to be reasonable estimates, such that the unknown parts $f_{\mathcal{R}}(H)$ and $f_{\mathcal{M}}(H)$ can be more easily modeled by the GP covariance. This design choice is in accordance with [C] where the incorporated domain knowledge is only used as an inductive bias

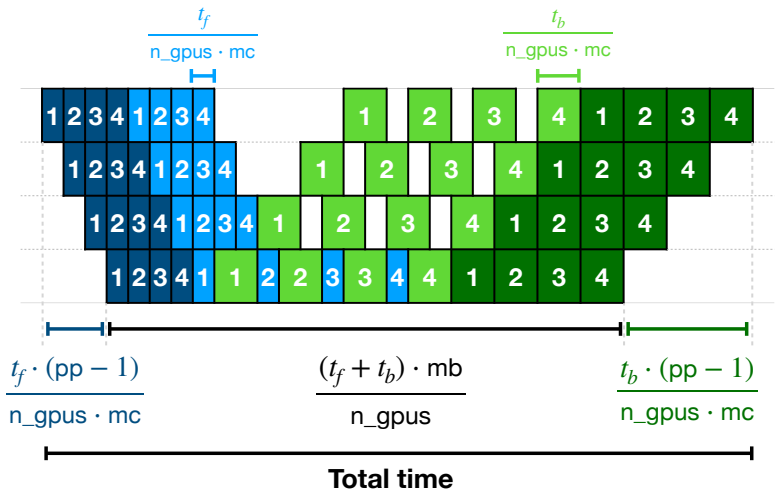

Figure 5: Estimate for computation time for PP where t_f and t_b are the time required for the forward and backward stages respectively, for when $\texttt{pp} = \texttt{n\_gpus} = 4$, $\texttt{mb} = 4$, and $\texttt{mc} = 2$

for the surrogate model whose uncertainty can be quantified, and is still accurate enough to be used in ③. Furthermore, since OPPA only requires the prior mean to perform forward evaluations, it can be replaced by a better surrogate function to fit under the same optimization framework, which could enhance the performance of OPPA. They can be made as complex as wanted, without needing additional intermediate functions beyond performing forward evaluations.

New prior mean for training throughput. We first introduce construct a prior mean which roughly captures the training throughput for a given PC. To do so, we provide an approximation for the time per training step, and find its reciprocal to get the throughput. For a training step, we can separately consider the time from some computation and time from communication across GPUs. To estimate the computation time, we assume an idealized machine that allows infinite parallelization, such that DP and TP are perfectly parallelized. Meanwhile, PP using an interleaved schedule incurs additional computation time from the microbatches being ran sequentially, and from pipeline bubble when the first microbatch is being fed through the pipeline (Narayanan et al., 2019). This additional computation time from PP, visualized in Figure 5, is roughly equal to

$$\hat{\mathcal{T}}_{\text{c}}(H; t_{\text{c}}) = \frac{t_{\text{c}}}{\texttt{n\_gpus}} \cdot \left(\texttt{mb} + \frac{\texttt{pp} - 1}{\texttt{mc}} \right) \tag{11}$$

where $\texttt{mb}$ is the number of microbatches used in PP (set to 1 when PP is not used), $\texttt{mc}$ is the number of model chunks for PP (also set to 1 when PP is not used), and $t_{\text{c}} = t_f + t_b$ is the total time to perform the forward and backward passes.

To estimate the communication costs, we consider the cost model from Thakur et al. (2005), which approximates the cost to perform a Point-to-point (P2P) communication for message of size n between two processes to be

$$T_{\text{p2p}}(n; \alpha, \beta) = \alpha + \beta n \,, \tag{12}$$

and an All-Reduce operation on a message of size n along p parallel processes to be

$$T_{\text{a-r}}(n, p; \alpha, \beta) = \alpha \log p + \frac{p - 1}{p} \cdot \beta n \tag{13}$$

where α is the communication initialization costs, and β captures costs that scale with message size such as the transmission and local reduction costs. As visualized in Figure 2, we assume that DP and TP involve All-Reduce communications, and PP P2P communications.

For DP, the only communication involved will be to synchronize the parameter updates after a training step pass. This cost can be written as

$$\hat{\mathcal{T}}_{\text{comm,dp}}(H; a, b_{\text{dp}}, c_{\text{dp}}) = T_{\text{a-r}}(c_{\text{dp}}, \texttt{dp}; a, b_{\text{dp}}) \tag{14}$$

where a, $b_{\texttt{dp}}$ and $c_{\texttt{dp}}$ are unknown variables. For TP, we assume that in a single pass of the NN there are $D_{\texttt{tp}}$ rounds of synchronization which would need to occur. When the training batch is divided into $\texttt{mb}$ microbatches, there would be $\texttt{mb}$ times more rounds of communication on message which are approximately $\texttt{mb}$ times smaller. In total, this would contribute to a total communication cost of

$$\hat{\mathcal{T}}_{\text{comm},\texttt{tp}}(H; a, b_{\texttt{tp}}, c_{\texttt{tp}}, D_{\texttt{tp}}) = D_{\texttt{tp}} \cdot \texttt{mb} \cdot T_{\text{a-r}}\left(\frac{c_{\texttt{tp}}}{\texttt{mb}}, \texttt{tp}; a, b_{\texttt{tp}}\right) \tag{15}$$

where we treat a, $b_{\texttt{tp}}$, $c_{\texttt{tp}}$ and $D_{\texttt{tp}}$ as unknown parameters. Finally, for PP, there would be $\texttt{mc} \cdot \texttt{mb}$ rounds of point-to-point communications each whose size would be inversely proportional to the number of microbatches, resulting in the overall PP communication cost of

$$\hat{\mathcal{T}}_{\text{comm},\texttt{pp}}(H; a, b_{\texttt{pp}}, c_{\texttt{pp}}) = \texttt{mc} \cdot \texttt{mb} \cdot T_{\text{p2p}}\left(\frac{c_{\texttt{pp}}}{\texttt{mb}}; a, b_{\texttt{pp}}\right) \tag{16}$$

where a, $b_{\texttt{pp}}$ and $c_{\texttt{pp}}$ are unknown. When combining the communication costs for all three types of parallelism, we obtain an overall cost of

$$\hat{\mathcal{T}}_{\text{comm}}(H; \mathbf{C}) = \hat{\mathcal{T}}_{\text{comm},\texttt{dp}}(H; a, b_{\texttt{dp}}, c_{\texttt{dp}}) + \hat{\mathcal{T}}_{\text{comm},\texttt{tp}}(H; a, b_{\texttt{tp}}, c_{\texttt{tp}}, D_{\texttt{tp}}) + \hat{\mathcal{T}}_{\text{comm},\texttt{pp}}(H; a, b_{\texttt{pp}}, c_{\texttt{pp}}) \tag{17}$$

where $\mathbf{C} = \{a, b_{\texttt{dp}}, b_{\texttt{tp}}, b_{\texttt{pp}}, c_{\texttt{dp}}, c_{\texttt{tp}}, c_{\texttt{pp}}, D_{\texttt{tp}}\}$ are constants which are to be inferred.

The estimate of the overall throughput is then written as

$$\hat{\mathcal{R}}(H; \mathbf{r}) = \left[\hat{\mathcal{T}}_{\text{comp}}(H; t_{\text{c}}) + \hat{\mathcal{T}}_{\text{comm}}(H; \mathbf{C})\right]^{-1} \tag{18}$$

where $\mathbf{r} = \{t_{\text{comp}}, \mathbf{C}\}$ are learned hyperparameters.

New prior mean for maximum memory usage. In addition to modeling the throughput of training, we also require a prior mean for the maximum memory usage when each PCs are used. In the simplest model, we only consider the memory required to store the NN parameters and those for backpropagation computation.

For NN parameters, its sharding can be done on the pipeline or on the layers, allowing us to approximate the GPU memory required for storing the NN parameters to be inversely proportional to $\texttt{pp} \cdot \texttt{tp}$. Note that assuming the simplest DP implementation, the NN parameters are duplicated and stored on each DP dimensions, and so the maximum memory usage is not affected by the DP.

Meanwhile, in the case of backpropagation computation, the maximum memory used will roughly be proportional to how many model parameters a certain GPU has to perform the forward and backward passes for, times how many training samples the GPU has to process at any one time. We expect this quantity to be inversely proportional to the number of total GPUs times the latter to depend on the number of microbatches used. We can therefore assume a prior mean for the maximum GPU memory usage to be in the form of

$$\hat{\mathcal{M}}(H; m_1, m_2, m_3) = \frac{m_1}{\texttt{pp} \cdot \texttt{tp}} + \frac{m_2}{\texttt{n\_gpus} \cdot \texttt{mb}} + m_3 \tag{19}$$

where m_1 captures the memory used for storing model parameters, and m_2 captures the memory used during backpropagation computations, and m_3 are any other additional memory overheads unaccounted for by our simple model.

In practice, we are unable to obtain an accurate measurement for the maximum memory usage when the actual allocated memory is above the threshold M_0. Therefore, we clip the prior mean such that they are bounded by the maximum memory possible, i.e., set the prior to be

$$\hat{\mathcal{M}}(H; \mathbf{m}) = \min\left\{\hat{\mathcal{M}}(H; m_1, m_2, m_3),\ M_0\right\} \tag{20}$$

where $\mathbf{m} = \{m_1, m_2, m_3\}$ are learned hyperparameters.

D.4 KERNEL

For our GP, we use the Matern kernel, which is given by (Rasmussen & Williams, 2006)

$$\begin{aligned} k(H, H'; \sigma_k, \lambda) &= k_{\text{Matern},\nu}\big(e(H), e(H'); \sigma_k, \lambda\big) \\ &= \sigma_k^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \big(\sqrt{2\nu}\, d_\lambda(H, H')\big)^\nu K_\nu\big(\sqrt{2\nu}\, d_\lambda(H, H')\big) \end{aligned} \tag{21}$$

where Γ is the Gamma function, K_ν is the modified Bessel function, σ_k is the kernel scaling constant,

$$d_\lambda(H, H') = \big(e(H) - e(H')\big)^\top \mathbf{\Lambda}^{-2} \big(e(H) - e(H')\big) \tag{22}$$

is the distance between two PCs embeddings, and $\mathbf{\Lambda} = \text{diag}(\lambda) = \text{diag}([\ \lambda_1 \cdots \lambda_p\ ])$ is a diagonal matrix of the lengthscales. In our method, we choose $\nu = 5/2$.

For OPPA specifically, to better capture the effect of the hyperparameters within a PC, we use an *additive kernel* (Duvenaud et al., 2011). Our motivation for this is the fact that different hyperparameters affect the training throughput both in different scales, and the choice of hyperparameter value correlates to the performance in different ways. In our case, for simplicity, we only consider the first order interactions, which are given by

$$k_{\text{additive}}(x, x') = \sum_{i=1}^{n} \sigma_i k(x_i, x_i'; \lambda_i) \tag{23}$$

where σ_i and λ_i are the scaling factor and lengthscale for dimension i of the input respectively, allowing the scaling and correlation effects from each hyperparameters to be learned more independently. In practice, we find that the performances are no worse compared to using a standard kernel, however results in interestingly interpretable results.

D.5 RANDOM SAMPLING FOR ADDITIONAL EXPLORATION

In OPPA, we sometimes select PCs at random for additional exploration. There are two scenarios which triggers a random selection of PC in OPPA.

1. In the first few chosen PCs. This is because in the beginning there are no PCs which can be used to infer the hyperparameters for the prior distribution of the GP, therefore a few PCs are chosen at random to kick-off the modeling process and provide a reasonably diverse set of samples to infer the hyperparameters well.
2. When too many out-of-memory errors have been encountered in a row. This is because any out-of-memory trials will not result in a usable training data for the throughput modeling and possibly minimal data for the maximum memory GP, which does not aid the GP model. When too many such cases are encountered, we attempt to do random exploration so that the model can receive some information that can be used to model better with and find new feasible PCs.

For the random selection process, we select a PC using a weighted random strategy, such that the probability of obtaining a PC with a certain parallelism dimension size configurations are equal.

D.6 PSEUDOCODE

We present the pseudocode for OPPA in Algorithm 1.

E DETAILED EXPERIMENTAL SETUP

E.1 BRIEF OVERVIEW OF EXPERIMENTAL SETUP

Our model training is implemented based on the COLOSSAL-AI framework (Li et al., 2023), which allows execution of NN training with 3D parallelism. We note, however, that OPPA is also general enough to be applied to any other training framework as well, whose implementation we leave to future works. We consider optimizing PC on different transformer-based language model training scenarios, and on different hardware configurations with varying number of GPUs – both details are as listed in Tables 1 and 2 respectively in Appendix E.

We perform PC optimization using standard BO which uses a constant prior mean and standard Matern-$5/2$ kernel, and using OPPA which uses parallelism-informed prior belief. We compare our proposed methods with RANDOM selection and XGBOOST (Chen & Guestrin, 2016), which are two methods that are used to optimize PCs in existing frameworks (Kuchaiev et al., 2019; Rasley et al., 2020). Note that in our experiments, the size of the search space $|\mathcal{H}|$ is in the order of 10^4, which renders exhaustive search infeasible.

Algorithm 1 OPTIMIZER FOR PARALLELISM CONFIGURATIONS (OPPA)

$\mathcal{X} \leftarrow \emptyset,\ \mathcal{Y}_{\mathcal{R}} \leftarrow \emptyset,\ \mathcal{Y}_{\mathcal{M}} \leftarrow \emptyset$
Generate all valid PCs $\mathcal{H}$
Select an initial PC H_1 at random
for $i = 1, 2, \ldots$ **do**
  // Step ① – Querying some PC
  Run training with PC H_i to obtain time per step $t_{H_i,1}, \ldots, t_{H_i,n}$ and maximum memory usage m_{H_i}
  Obtain r_{H_i} and $\sigma^2_{r_{H_i}}$ based on Equation (7)
  $\mathcal{X} \leftarrow \mathcal{X} \cup \{H_i\}$
  $\mathcal{Y}_{\mathcal{R}} \leftarrow \mathcal{Y}_{\mathcal{R}} \cup \{(r_{H_i}, \sigma^2_{r_{H_i}})\}$
  $\mathcal{Y}_{\mathcal{M}} \leftarrow \mathcal{Y}_{\mathcal{M}} \cup \{m_{H_i}\}$
  // Step ② – Modeling throughput and memory usage
  Model GP with data $(\mathcal{X}, \mathcal{Y}_{\mathcal{R}})$, with prior mean from Equation (18) and covariance from Equation (21)
  Model GP with data $(\mathcal{X}, \mathcal{Y}_{\mathcal{M}})$, with prior mean from Equation (20) and covariance from Equation (21)
  // Step ③ – Selecting the next PC to query
  if $i < N_{\text{random}}$ **then**
    Select H_{i+1} randomly
  else
    $H_{i+1} \leftarrow \underset{H \in \mathcal{H} \setminus \mathcal{X}}{\arg\max}\ \text{cEI}(H; \mathcal{X})$
  end if
  if time exceeded **then**
    terminate
  end if
end for
return $\underset{H \in \mathcal{X}\ :\ m_H < M_0}{\arg\max}\ r_H$

E.2 ALGORITHMS RAN

We list the algorithms we have ran along with their implementation details here.

- RANDOM. We perform random selection where the probability of selecting any valid PC $H \in \mathcal{H}$ is equally likely.
- XGBOOST. We use a gradient-boosted tree (Chen & Guestrin, 2016) to construct a surrogate with the observed throughput values, and then find the next PC to try based on the maximum value predicted by the surrogate. Note that this is implemented as done in NEMO framework (Kuchaiev et al., 2019), as discussed in the AutoConfigurator from `https://docs.nvidia.com/nemo-framework/user-guide/latest/usingautoconfigurator.html`.
- STANDARD-BO. We perform BO with a GP with constant mean and Matern-5/2 kernel, as described in Section 3.
- OPPA. We perform BO with a GP with parallelism-informed prior belief which is described in Appendix D.3.

E.3 MODELS TRAINED

We list the models used in our experiments in Table 1. All models used are based on the transformer architecture, and were retrieved from Huggingface.

Table 1: Details of models used in our experiments

Setup name	Model	Param. count	Batch size	Max. seq. length	PC Search Time (mins)
BERT	BERT Base, Uncased	110M	256	256	30
GPT-2	GPT-2	124M	4096	1024	45
LLaMa-7b	LLaMa 2	7B	256	1024	60

Table 2: Configurations of tested hardwares.

Config. Name	GPU Model (Memory)	GPUs per host	# Host
8 GPUs	NVIDIA RTX A5000 (24GB)	8	1
16 GPUs	NVIDIA A100 (40GB)	4	4

E.4 Additional Details on Hardware Configurations

In Table 2, we list out the hardware configuration used in our experiments. For the configurations with 16 and 32 GPUs, the training process involves multiple host machines, and therefore an assumption that all communication occurs at the same speed is no longer true. We demonstrate that OPPA does not require this to be modeled explicitly, however can still be efficiently learned by the surrogate model.

F Additional Results

F.1 Plots of Trials Ran versus Best Throughput

In Figure 6, we plot the best achieved throughput versus the number of training trials that have been ran. We see that in this view, OPPA still outperforms other benchmarks, however at a seemingly smaller margin than when plotted in Figure 3. This is due to the fact that when plotted against the number of trials, bad PCs are not penalized as heavily as in practice where the trials actually take a longer time to be ran. Despite this, we see that OPPA is both more time efficient and query efficient, which can be useful when the overhead to perform one trial may become higher, for example when the framework is adapted to run on a cluster with a job scheduler.

F.2 Predicted Throughput and Memory Use by Parallelism-Informed Prior Belief

In Figure 7, we compare the modeled throughput with the true throughput value, for when a parallelism-informed prior belief is used and when it is not used. We see that in both cases, the predicted throughput correlates well with the true throughput, even when the prior belief is not used. This suggests that GPs themselves are already good surrogate models to model the throughput.

In Figures 8 to 10, we compare the modeled throughput and memory usage with the true values, however for a training scenario where there are more possible PCs. We see that in this case, using the prior belief allows for the values to be modeled adequately well, but more importantly, allow for the PC which achieves the best throughput to also have the highest values, and therefore be identified correctly. We find that for the BO process, a surrogate only needs to model the good PCs well in order to select a good PC in the end. Meanwhile, the GP without prior belief learns the patterns much less efficiently or do not learn them at all. This correlates well with the results in Figures 3b and 3c where standard BO selects a worse PC compared to OPPA which uses a better prior belief.

F.3 Kernel Interpretability

In this section, we attempt to interpret some hyperparameters that result from our surrogate modeled in OPPA. In Table 3, we show the additive kernel hyperparameters. We notice that out of all of the parameters, `tp`and ZERO stage hyperparameters seem to have the shortest lengthscale and largest output scales. This may mean these hyperparameters are influential in determining whether the PC is optimal or not, but may not be properly captured by the prior belief.

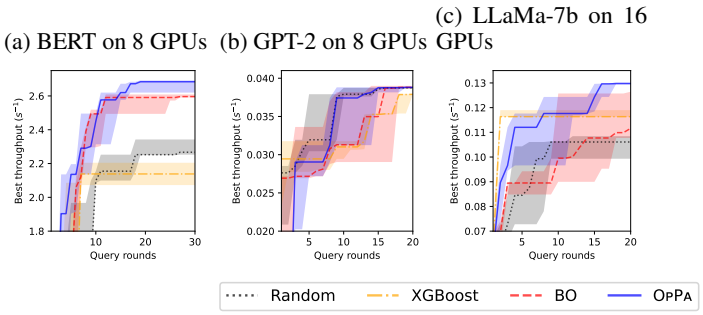

Figure 6: Results of the best obtained throughput (higher is better) plotted against the amount of training trials ran. The lines represent the median value of the best obtained throughput across five trials, while the error bar represent the quantile values.

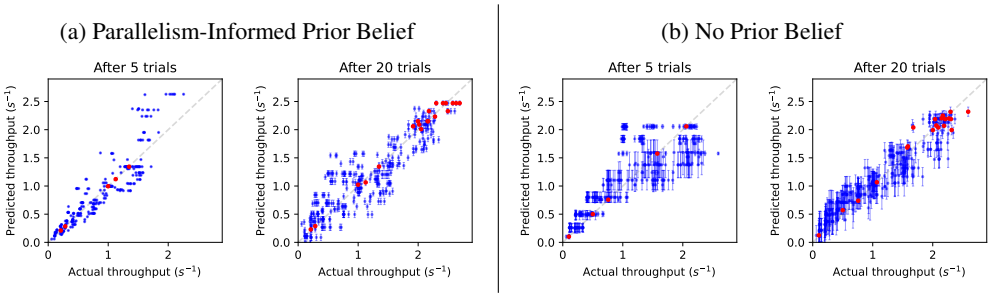

Figure 7: Comparison of modeled throughput values versus the true throughput for training of BERT model.

This matches our intuition, where since the optimal PCs for the BERT training typically rely heavily on DP, whose hyperparameters have not been considered in the approximate throughput. This may be an indication that a more sophisticated surrogate may be possible that more explicitly considers the effects of these hyperparameters.

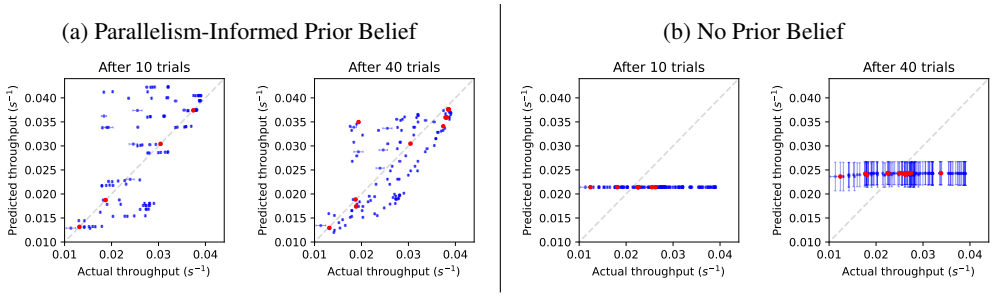

Figure 8: Comparison of modeled throughput values versus the true throughput for training of GPT-2 model.

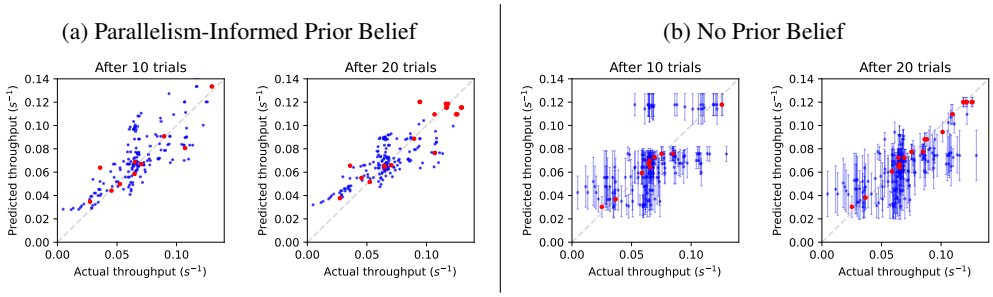

Figure 9: Comparison of modeled throughput values versus the true throughput for training of LLaMa-7b model.

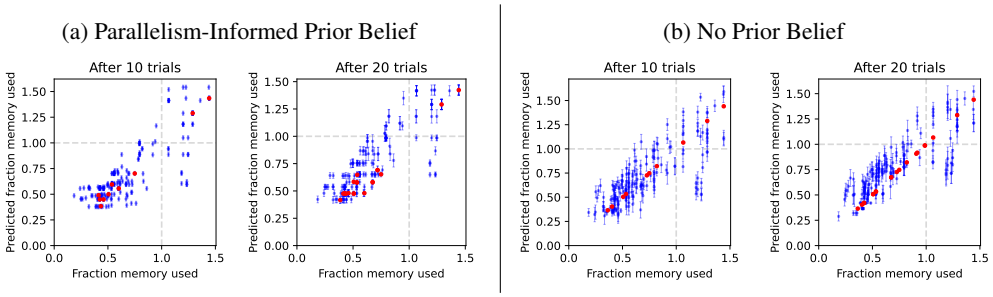

Figure 10: Comparison of modeled maximum memory use versus the true memory use for training of LLaMa-7b model.

Table 3: Kernel hyperparameters from GP fitted in OPPA on BERT training

Quantity	$\log \sigma_i$	$\log \lambda_i$
`dp`	-9.815	5.490
`tp`	**-1.334**	**-1.047**
`pp`	-9.764	5.731
DP bucket size	-9.896	5.543
ZERO stage	**-3.231**	**-1.021**
ZERO bucket size	-8.892	5.142
Number of microbatches	-8.569	6.272
Number of model chunks	-7.668	5.233