ProxyVM: A Scalable and Retargetable Compiler Framework for Privacy-Aware Proxy Workload Generation

Xida Ren Alif Ahmed Yizhou Wei Kevin Skadron Ashish Venkat
University of Virginia

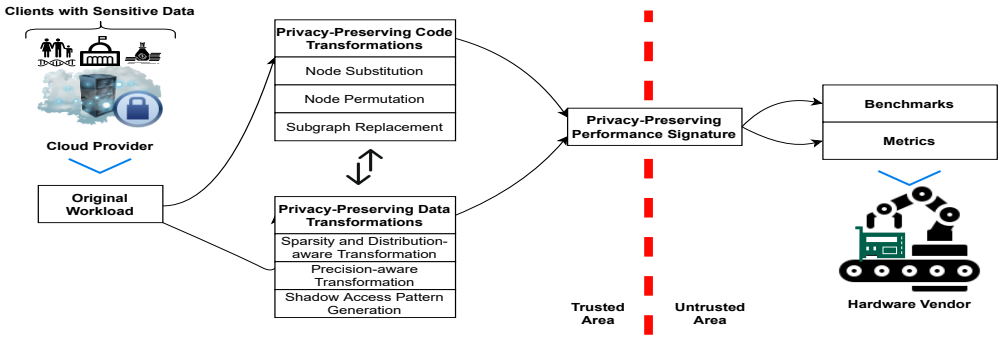

Figure 1: Overview of ProxyVM

I. Introduction

Cloud providers invest in specific hardware architectures to optimize emerging workloads. Due to the private nature of such workloads' algorithms and privacy concerns over potentially sensitive user data, the cloud hardware supply chain has increasingly relied on synthetic benchmarks to guide application-specific hardware design.

We propose ProxyVM, a scalable, retargetable compiler system that generates synthetic workloads with great performance predictability. The key to ProxyVM is a *privacy-preserving performance outline*, a rich intermediate representation implemented with MLIR [1], generated by a performance characterizing front-end deployed at the customer site. This allows the performance characteristics of a sensitive workload to be seamlessly shared with hardware vendors, with the guarantee that proprietary algorithms and sensitive user data remain invulnerable to disclosure. This privacy-preserving performance blueprint allows workloads to be classified as computational chunks or algorithms of variable granularity, parameterizable by deidentified/obfuscated input data properties. ProxyVM's backends can use this outline to generate synthetic workloads and guide application-specific hardware design.

This research is expected to benefit key stakeholders in the machine learning (ML) supply chain by streamlining the hardware design process and minimizing vendor clearing expenses. This research should also boost ML supply chain resilience as privacy-based regulations change over time.

II. Technical Approach

A. Need for Unified Performance Characterization

The literature offers several characterization methodologies that allow us to make compromises based on workloads and hardware platforms to be supported. Top-down, motif-based techniques [2], [3] break down domain-specific algorithms into motifs such as sparse matrix operations, sorting, and fourier transformations, allowing workloads to be characterized as mixes of motifs parameterized by input data properties. This tends to work well when detailed information on the workload is available, but the specific hardware architecture is unknown. In contrast, when limited workload information is available but the hardware platform is known bottom-up, performance trace-based proxy workload generation [4] may be used. Here, the proxy workload is often generated by tracing trace-based proxy workloads based on traces that characterize performance aspects such as instruction mixes, memory access, and branching patterns.

However, existing techniques have limitations. Top-down approaches don't scale with new, emergent, and rapidly evolving workloads, and bottom-up approaches don't scale with exotic and accelerator-rich hardware platforms. Also, neither technique allows fine-grained privacy and performance tradeoffs, underscoring the need for a unified approach that supports diverse workloads and hardware architectures.

B. Overview of ProxyVM

One of the major goals of our approach is to provide scalability, catering to diverse workloads and hardware architectures. To this end, we design our *privacy-preserving performance outline* as a rich, MLIR/TVM-based [1], [5] intermediate representation that allows us to securely encapsulate the performance characteristics of a wide range of applications without divulging proprietary information and support seamless and efficient code generation for a variety of target platforms from multicore processors to specialized accelerators.

We define our IR to be similar to TensorFlow's computation graph [6], allowing workloads to be represented as graphs where nodes represent algorithms (e.g., sorting, pattern matching, matrix multiplication, etc.) along with their associated states and parameters and directed edges represent data flow. Our compilation workflow based on MLIR [1] and TVM [5] then progressively lowers the intermediate representation to the

appropriate hardware target, applying both machine-independent and machine-dependent optimizations as needed. Further, the compilation workflow is also equipped with privacy-preserving data and code transformations to remove proprietary and sensitive information, while retaining intimate performance characteristics of the workload. Figure 1 provides an overview.

C. Privacy-Preserving Data Transformations

To capture data-induced performance effects, attributes of the real workload's dataset need to be encoded into the IR as metadata. For example, the same ML algorithm may have vastly different cache effects depending upon the network configuration, size and composition of the weight matrices, and other hyperparameters. However, it is important to ensure that such metadata and other input-dependent properties are properly anonymized and obfuscated before inclusion to protect proprietary and sensitive data from getting accidentally leaked. This work explores several privacy-preserving data transformations to generate synthetic data that accurately captures data-induced performance effects, while also simultaneously ensuring that the original data may not be recreated using the synthetic data. To exemplify the need for such transformations, we highlight key properties of modern ML workloads that impact their performance.

Sparsity-induced effects. Tensor sparsity could drastically affect the performance characteristics of a workload, as many accelerators tend to skip zero values in MAC operations to save power and time [7]–[9]. To accurately capture the performance of workloads on such hardware, it is crucial that the generated data has similar sparsity at nodes of the data-flow graph where sparsity matters, and further ensure that the input data distribution is preserved since the product of two dense matrices could have arbitrary sparsity, impacting the composition of future compute nodes. In addition, our data transformation passes take into account the distribution of input tensors and their impact on sparsity as those values propagate through the compute graph, where tensors are multiplied, aggregated, run through activation functions, and subjected to sparsity-inducing regularization schemes. While it is possible to generate synthetic data that preserve data-induced performance effects by simply retaining broadly derived statistical measures such as data distribution and sparsity, we may still need to apply appropriate differential privacy techniques when computing these measures.

Precision-induced effects. Recent work on low-precision accelerators, including IBM's RaPiD [10] and Google's BFloat16 tensors [11], [12], has not only shown that it is viable to train neural networks using low-precision integer and floating-point values for weights and activations, but that it could result in substantial savings in energy and hardware cost, while sustaining a minor loss in inference accuracy and training efficiency. However, synthetic workloads that solely rely on replicating input data characteristics without taking into account the increased number of epochs to train models and the decreased accuracy of the trained model on such low-precision accelerators may not completely capture the characteristics of the original workload. Thus, it is important that our data transformation passes not only capture the data-induced performance effects, but also mimic the training efficiency and the inference accuracy of the original workload when running on reduced-precision hardware.

Data-dependent memory access patterns. Several modern machine learning workloads are increasingly characterized by data-dependent memory access patterns that heavily influence their execution behavior and accelerator design requirements. For example, consider GNNs that are characterized by a mix of regular and data-dependent irregular access patterns. The data-dependent irregular accesses occurs in the *aggregation* stage at the beginning of each layer in the GNN, where every vertex receives an aggregation of features from vertices in its neighborhood. The regular accesses occur right after aggregation in the *transformation* stage, as the aggregated features of each neighborhood are run through various neural networks to produce input features for the next layer [13], [14]. The composition of such a regular/irregular mix is highly dependent upon the input graph, and running the same algorithm on different datasets can produce vastly different performance characteristics [13]. To this end, our privacy-preserving passes allow generating shadow loads and stores that mimic the access patterns of the original workload, while ensuring that the access patterns do not reveal the structure of the graph itself if it is considered sensitive/proprietary.

D. Privacy-Preserving Code Transformations

To protect confidential algorithmic details, certain segments of the computation graph may need to be obfuscated. This involves performance-sensitive node substitutions (e.g., substituting hash table lookups with sorting) and/or edge permutations, since they entail irregular memory accesses. This involves marking sensitive nodes on the graph and then running novel privacy-preserving passes that randomly substitutes nodes and permutes edges while attempting to preserve performance characteristics such as memory access patterns (for node substitution) and inherent parallelism (for edge permutations).

To support non-traditional workloads that may not be represented as computation graph nodes, our workflow grafts the performance profile-based bottom-up approach into our generally motif-based top-down computation graph. In particular, the nodes in our computation graph typically correspond to a standard algorithm and are parameterized with algorithm-specific parameters. If the behavior of certain phases cannot be correlated to any algorithm/motif, it is still possible to extend our library to include *shadow motifs* using existing proxy workload generation techniques, such as our work on [15], albeit enhanced with data/control dependency information, and fine-grained differential privacy techniques. A key challenge here is that such shadow motifs may not easily be ported across radically different architectures, but we want to be able to design accelerators or functional units to target emerging code behaviors. To this end, this work features a design space exploration of workloads to fine-tune our proxy workload generation process such that the

performance of the generated motif when ported and optimized for a new architecture is predictive of the performance of the original code ported/optimized for that architecture.

III. Preliminary Results

A. Top-Down Exploration

As a preliminary exploration of top-down modeling of performance and privacy, we chose to examine the relationship between convolunional neural network hyperparameters and a set of standard performance counters that focused on three aspects of execution: branching, caching, and throughput. We chose covolunional neural network workloads as our "model workload class" for two reasons. First, they're ubiquitous in computer vision and many other ML settings. Second, convolutional neural networkloads have model-structure hyperparameters such as filter stride and filter size that directly influence their branch and cache behavior.

For this preliminary exploration, we compiled our workloads into vectorized and optimized x86 binary with ONNX-MLIR. While we ran them on a Intel(R) Core(TM) i7-8700T CPU @ 2.40GHz CPU under the coffee lake architecture, we are ready to switch to GPUs and even FPGAs and ASICs as the ONNX and MLIR ecosystems have support for direct translation into CUDA and other domain specific codes.

We consider the neural network hyperparameters as secrets, and explore the information gain a potential adversary can achieve by querying the performance counter data for an unknown workload. In Section III-A1, we demonstrate that performance counter measurements can have nonlinear dependencies on workload hyperparameters. In Section III-B, we show some factors that play into the difficulty at which hyperparameters can be inferred based on performance counters.

1) Relationship Between ML Workload Hyperparamters and Hardware Performance

Figure 2 showcases possible relationships between performance counter measurements and hyperparameter values. The relationship could be a simple linear one, as row 1 in Figure 2 indicates. Here, linear classifier (logistic regression) based on cache miss rate and branch miss rate can produce a good separation between workloads with different hyperparameter values (in this case, number of convolutional filters).

Nonlinear Relationships. Contrast that with row 2 in Figure 2, where a nonlinear model produces significantly higher classification accuracy. Here, the hyperparameter convolutional kernal size can be predicted with high accuracy from the performance counters given, but a more sophisticated model has to be applied.

Some hyperparamters are difficult to predict from performance counters. For example, in colume 3 of Figure 2, the activation function does not significantly affect cache or branch behavior, so neither linear classifiers nor SVM produces good separation between the possible values. When only these hyperparameters are considered secret, it would be safe to share performance counter data.

param_activation	0.57
param_kernel_size	0.81
param_layer_count	0.53
param_filters	0.84
param_strides	0.98
param_batch_size	0.54
param_pool_strides	0.66
param_pool_size	0.55

Table I: SVM prediction accuracy by Hyperparameter using branch miss rate, cache miss rate, and instruction count per batch; Average accuracy over 4-fold cross validation

However, this comes caveats. For example, in colume 3 of Figure 2, there is a cluster of ReLU activation workloads to the lower far right part. If an interested party knows this as a general trend with real world workloads, they could infer the hyperparameter values for specific workloads with high confidence even if they remain unable to do so for the average workload.

Adding Performance Counters can Dramatically Increase Predictability. Also, we must consider the possibility that adding a performance counter may yield signals that cause massive leakage. For example, in colume 4 of Figure 2, even the nonlinear classifier has only an accuracy of 72% over the performance counters given, adding the per-batch instruction count gives an accuracy above 90% for just the linear model, as increasing the convolutional strides dramatically reduces the number of instructions needed for each batch.

2) Inferring Hyperparameters from Performance Counters

As shown in Table I, directly sharing performance counters will allow potential attackers to infer workload secrets via different methods with varying successes, with incredible accuracy for hyperparameters with high performance impact (e.g. convolutional strides). Surprisingly, some performance-impactful hyperparameters (e.g. layer count) do not have high prediction accuracies, perhaps due to how other hyperparameters can have a confounding effect (in this case, while layer count increases instruction count per batch, so does batch size and pool strides, and the SVM has trouble distinguishing between the effects of the three).

B. Preliminary Results: Obfuscating ML Hyperparameters via Collisions

In this preliminary study, we explore potential for hiding sensitive hyperparameters. Unless otherwise stated, this study uses the same setup as the one on hyperparameter inference/stealing.

First, we want to see whether there is any "collision": models with different hyperparameters but similar performance. Collision models provides protection via "deniability". That is, given a set of performance counters, the attacker does not know which model it is. Theoretically, we should be able to find collision models if we limit the number of performance counters the attacker could observe.

Towards this, our preliminary experiment explores the effect of varying the dimensionality of hyperparameter space and performance-counter space on the difficulty

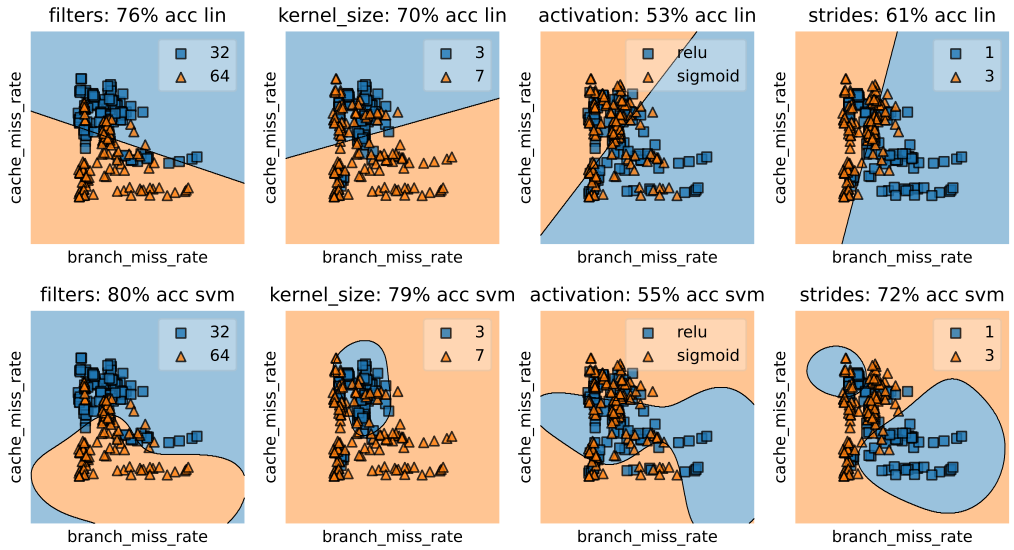

Figure 2: Linear classifiers works well for the number of convolutional filters, a nonlinear classifier works better for the kernel size (SVM with RBF kernel), and neither classifiers can separate workloads with different activation functions based on the performance counters given (branch and cache miss rates).

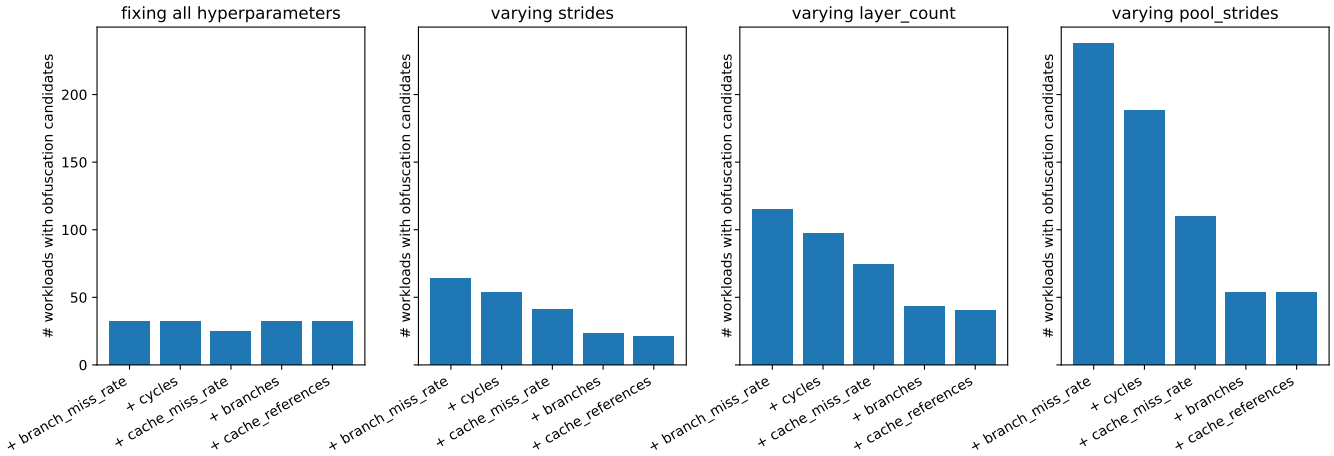

Figure 3: Demonstration of hyperparameter obfuscation by measuring the number of models that share similar performance counters but with different hyperparameters. From left to right, we first fix three covolutional hyperparameters (stride, layer-count, and pool-strides) while varying others (e.g. filter count, pool size), and (within each panel, from left to right) consider more and more performance counters. We then relax the 3 hyperparameters one by one. The number of workloads with obfuscation candidates increase as more hyperparameters are allowed, and decreases as more performance counters are measured.

of hyperparameter inference, and the possibility of anti-inference obfuscation. For each workload w, we computed its eight closest neighbors in performance-counter space. If a closest neighbor n has a different value for hyperparameter h, n is counted as an obfuscation candidate for w under h. The distribution of the obfuscation candidate counters can also be interpreted as a cluster-mixing metric: when the workloads with the same hyperparameter values fall into disjoint clusters in the performance-counter space, most workloads should have no obfuscation candidates at all; in contrast, when the clusters overlap more, each workload is expected to be closely adjacent to more workloads with different hyperparameter values.

For each hyperparameter h, under a variety of scenarios, we computed the number of obfuscation candidates for every workload. In figure 4, note that the number of obfuscation candidates increases as more hyperparameters are added and performance-counter space clusters get mixed up, while the number of obfuscation candidates decrease as more performance counters become available to separate the previously mixed clusters.

Our preliminary results shows that larger sets of potential workloads makes it hard to distinguish different workloads based on performance counters. However, as more performance counters are added, more information is revealed. Accurate performance estimates require performance counter disclosure, and so tradeoffs have to be made.

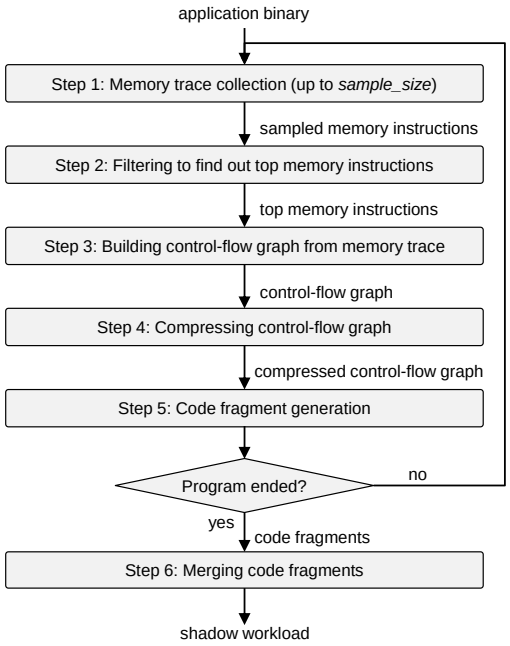

Figure 4: Overview of the proposed bottom-up proxy generation framework.

We explore methods for optimizing these tradeoffs in the upcoming research.

C. Bottom-Up Exploration

Currently our bottom-up exploration framework focuses on accurately capturing the memory access pattern. The memory access pattern of an application is composed of thousands of inter-dependent load/store instructions. To capture the pattern properly, we need to make sure that: (a) The instructions are executed in the correct order, and (b) When executed, the instructions are accessing the appropriate memory address. The bottom-up framework we are proposing strives to enforce these properties. The framework consists of six steps, as shown in Figure 4. The first step profiles the instrumented binary to collect up to a certain amount (configurable) of memory trace. The second step filters the trace to reduce the effective number of memory instructions and select only the instructions that are responsible for the majority of the accesses. This step helps minimizing cloning noise while retaining high level of accuracy. The third step processes the memory access trace for these top instructions and builds a statistical control-flow graph. The fourth step performs several transformations on the control-flow graph to reduce the number of basic blocks. This step also infers loop structures from the flow graph. The fifth step takes in the simplified control-flow graph and generates a code fragment representing that particular execution interval. The control-flow graph helps to enforce the instruction ordering. This step also determines the access pattern of each individual instruction from their execution trace (i.e., preserves the memory access address ordering of instructions). Steps one through five are repeated to generate additional code fragments for each execution interval until the program terminates or a user-specified number of instructions are executed. Afterward, the final step merges the code fragments to generate the shadow workload.

IV. Acknowledgement

This work was supported in part by Semiconductor Research Corporation (SRC).

References

[1] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.

[2] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, F. Tang, B. Xie, C. Zheng, X. Wen, X. He, *et al.*, "Data motifs: A lens towards fully understanding big data and ai workloads," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[3] W. Gao, J. Zhan, L. Wang, C. Luo, Z. Jia, D. Zheng, C. Zheng, X. He, H. Ye, H. Wang, *et al.*, "Data motif-based proxy benchmarks for big data and ai workloads," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[4] R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[6] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 1–1, 2016.

[7] M. Mahmoud, I. Edo, A. H. Zadeh, O. Mohamed Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 781–795, 2020.

[8] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bittactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[9] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2018.

[10] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran, Y. Li, E. Ogawa, K. Ishizaki, H. Inoue, M. Schaal, M. Serrano, J. Choi, X. Sun, N. Wang, C. Chen, A. Allain, J. Bonano, N. Cao, R. Casatuta, M. Cohen, B. Fleischer, M. Guillorn, H. Haynie, J. Jung, M. Kang, K. Kim, S. Koswatta, S. Lee, M. Lutz, S. Mueller, J. Oh, A. Ranjan, Z. Ren, S. Rider, K. Schelm, M. Scheuermann, J. Silberman, J. Yang, V. Zalani, X. Zhang, C. Zhou, M. Ziegler, V. Shah, M. Ohara, P. Lu, B. Curran, S. Shukla, L. Chang, and K. Gopalakrishnan, "Rapid: Ai accelerator for ultra-low precision training and inference," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[11] "BFloat16: The secret to high performance on Cloud TPUs,"

[12] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[13] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29, IEEE, 2020.

[14] F. Shi, A. Y. Jin, and S.-C. Zhu, "Versagnn: a versatile accelerator for graph neural networks," *arXiv preprint arXiv:2105.01280*, 2021.

[15] A. Ahmed, "Fine grained shadow workload generation preserving memory access patterns." https://alifahmed.github.io/res/qualifying_exam_report_033021.pdf, 2021. Ph.D. Qualification Exam.