LoopStack: ML-friendly ML Compiler Stack

Bram Wasti Meta AI bwasti@
Dejan Grubisic Meta AI dejang@
Benoit Steiner Meta AI benoitsteiner@
Aleksandar Zlateski Meta AI zlateski@

Abstract

We present LoopStack, a domain-specific compiler stack for tensor operations, composed of a front-end, LoopTool, and an efficient optimizing code generator, LoopNest. LoopStack is designed to produce highly efficient but also predictable code. Such a design allows both experts, and more importantly, ML–based approaches to find good schedules (algorithms). LoopStack is extensible and supports various processors and accelerators while incorporating HPC optimizations often missing from other machine learning compiler back-ends. To show the quality of the generated code we designed a rudimentary AI to search for schedules and compare the speed of generated code with the most optimized, hand-tuned libraries. Further, we show that for a large collection of schedules LoopNest's compilation is orders of magnitude faster than LLVM, while resulting in equal or improved run time performance.

1 Introduction and Motivation

The availability of massive amounts of computing power has fueled the explosive growth of machine learning techniques for almost a decade and has greatly affected the design of modern hardware. Companies, such as NVIDIA with their tensor cores, Intel/AMD with their specialized AVX, FMA, and VNNI instructions, an ARM with the Neon and SVE extensions to their instruction set, and Apple with their M1 CPU and its matrix co-processor, have tailored the computational capabilities of their respective hardware to better serve the needs of deep learning workloads.

However, leveraging the raw computational power of such hardware for the fast execution of machine learning models remains a challenge. Several approaches have been proposed over time. The traditional approach is to rely on libraries of optimized primitive tensor operations, such as cuDNN [10], OneDNN [11], or XNNPACK [15]. This approach is unsustainable because the ever-growing set of use cases renders these libraries huge in code size, which requires a large amount of manpower for maintenance. And more importantly, such operators can only exchange data through global memory, which can be a significant bottleneck.

Another approach attempts to avoid these problems; projects such as Halide [35] and TVM [8] represent computations using a declarative domain-specific language. This high-level representation is then scheduled and lowered into LLVM intermediate representation, and then compiled into instructions that can be executed directly on hardware by the LLVM compiler. To make this approach more practical, researchers have resorted to ML-based auto-tuning techniques. The mathematical problem is defined by a human and the schedules are determined by the machine using various ML techniques [2, 4, 9, 26, 38, 40, 42, 48].

Due to the improved execution times of modern chips designed to handle a variety of machine learning workloads, benchmarking useful operators or fused blocks can be done efficiently. However, this approach suffers from very large compilation times, thus making certain techniques, such as auto-tuning impractical.

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on ML for Systems Workshop

To overcome these limitations, we introduce LoopStack, a lightweight tool-chain designed specifically for deep learning workloads. To express computations, LoopStack introduces a **domain specific representation** based on Einstein notation [12]. This representation can capture many common dense neural network operations in a concise form. LoopStack also provides a **frontend** capable of converting neural networks, expressed as data-flow graphs, into our representation. Finally, we present a **code generator** that can compile this representation into highly optimized code for X86 and ARM CPUs extremely quickly.

Unlike other approaches that rely on off-the-shelf components such as LLVM, LoopStack was designed from the ground up to both: 1) allow for use of ML techniques on top of our compiler, and 2) target machine learning workloads. Our system makes the following contributions, which are particularly important for ML-based compilation research:

- Rapid compile times, a fraction of time compared to traditional compilers.
- Predictable and quick feedback.
- Generated machine code with performance comparable to or exceeding that of hand-optimized libraries such as MKL-DNN or XNNPACK.

2 Machine Driven Scheduling and Optimization

A core challenge in high-performance tensor operations is identifying a good execution order — the schedule. There can be many valid schedules for a given computation: loops can be split and reordered, dimensions can be split and swapped, intermediate tensors can be packed, and so on. As a result, any scheduling tool is faced with the challenge of an exponentially large search space making simple enumeration infeasible. Recent work [48, 40, 2] has explored using a combination of machine learning and structured search strategies to automatically explore the search space.

A key goal of LoopStack is to facilitate ML-based exploration of schedules. Effective exploration depends on rapid feedback, which is crucial to ML approaches, particularly when using reinforcement learning.

3 LoopStack's ML-centric Design

LoopStack is composed of a front-end (LoopTool) and a back-end (LoopNest). This design separates the representation of the program being described from the logic to emit target-specific code.

The front-end (LoopTool) is designed to enable the expression of both the user's mathematical intent as well as the preferred execution of the program — the order in which basic mathematical operations should be executed (i.e. the schedule) to perform the desired computation. This is done by defining a structured intermediate representation composed exclusively of a data-flow graph (DFG) as well as a loop order annotation language that denotes how each node is executed. The omission of control flow in the representation of the program is consistent with a large majority of neural networks, which can be completely represented as purely functional operations on N-dimensional tensors.

The back-end (LoopNest) is designed to generate optimized machine-code efficiently that **strictly** adheres to the user's input. LoopNest exposes an API that allows the user to define a specific loop nest order in which the computation should be performed. LoopNest will **not make any attempt** to change the provided nest order or use any logic to recover user intent. In short, LoopNest will generate a machine code that performs the exact computation that the user requested, and in the same order. This is crucial for ML-based auto-tuners, as the hidden functionality in traditional compilers can render the learning problem extremely complicated.

However, LoopNest applies low-level optimizations that are specific to the target hardware. Good schedules yield performances comparable to or better than hand-tuned primitives, and/or traditional compilers, as shown below.

3.1 Simplicity and Expressiveness

The goal of simplicity extends to both the description of intended mathematical operations and the process of defining the execution order (the schedule) of the resultant loops.

We use an Einstein-like notation [12] as it allows the user to easily describe the intended mathematical operations even for complex machine learning models. We further provide a minimal, yet powerful, intermediate-representation (IR) and limit the API such that all relevant optimization operations can be decomposed into a series of operations on individual nodes in the DFG. Despite resembling Halide's pipeline scheduling states[2], it is not possible to represent illegal schedules in our IR. This is particularly important for ML-based approaches, such as reinforcement learning.

3.2 Performance Predictability & Rapid Feedback

Traditional compilers perform expensive analyses to understand user intent and optimize execution while performing equivalent computation. This can both significantly increase the compilation times, as well as obscure the impact the intended schedule had on performance.

LoopStack takes a different approach. In no case does LoopStack attempt to understand the user's intent and/or reorder the intended schedule provided by the user, except in very few, hardware-specific cases, such as reordering loads and stores. Such optimizations highly depend on the limitations of the target hardware, don't require tuning, and are not beneficial to expose to the user.

This approach allows LoopStack to perform extremely fast compilation (quasi-linear in the number of issued instructions), which allows the user and/or ML agent to directly and rapidly evaluate the performance of their intended schedules. This approach greatly benefits auto-tuners [2, 48] and reinforcement learning techniques which require both fast and accurate feedback on the quality of the schedule. Additionally, the vast amounts of data required for supervised-based methods have taken a long time and/or many resources to collect [40] with previous approaches. As shown below, LoopStack provides orders of magnitude improvements in this domain.

4 Evaluation

The most important enabler for ML-based ML compilation provided by the LoopStack is its extremely low compilation times. We perform a set of experiments to compare both compilation time and execution time of LoopStack with state-of-the-art compilers often used for machine learning based auto-tuning. Additionally, we analyze how quickly we can evaluate the performance of generated code by comparing LoopNest with a cost model-based approach.

4.1 Auto-tuner for workload generation

An efficient schedule of ML workloads uses various techniques that improve performance by increasing data locality [41, 47, 7, 46, 24, 6, 28, 22, 45, 31]. We wrote a three-step automatic tuner to emulate a relatively simple machine learning process. This script sweeps through possible tile sizes that are expected to perform well on modern hardware (as presented in [16, 39]) and runs on the order of seconds to minutes on a single core for most individual neural network operations or blocks.

First, we define a set of tile sizes (based on modern hardware's cache and register file sizes). In the second step, we permute the loop order. Finally, we explore various memory layouts, including creating copies of packed intermediate layouts, to increase cache use.

Note that the above optimizations vastly reduce the search space of schedules by relying on linearizing assumptions. We could produce a more general optimization space by relaxing these assumptions, but this would require an intelligent agent to learn a search policy as brute force would become intractable.

4.2 Compilation time experiments

We compare LoopNest's compilation time to the LLVM compiler, a popular backend choice for prominent tensor autotuners such as Halide [35] and TVM [8]. For benchmarks, we used 12 common operators found in machine learning workloads over varying input sizes. For LLVM code generation, we use Halide to emit schedules identical to the ones used with LoopStack.

Table 1 presents summary of compile time across benchmarks. Our experiments show that LoopNest's code generation is faster than LLVM's compilation for all schedules, all workloads, and on all target hardware. In most cases, LoopNest can generate code orders of magnitude faster.

	x86 based CPUs						Aarch64 (ARM) based CPUs (NEON)											
	AMD (AVX2)			Intel (AVX512)			Cortex A57			NVIDIA Denver2			Cortex A73			Apple M1		
	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio
CONV-1	820.78	**2.5153**	**326.31**	9881.9	**6.2062**	**1592.3**	2948.4	**9.2109**	**320.1**	3972.4	**30.28**	**131.19**	4914.6	**22.077**	**222.61**	555.24	**24.763**	**22.422**
CONV-2	1590.1	**11.717**	**135.7**	9531.3	**5.8035**	**1642.3**	2481.1	**8.9475**	**277.29**	4798.5	**21.768**	**220.44**	4310.1	**31.053**	**138.8**	531.12	**15.261**	**34.802**
CONV-3	762.52	**5.9325**	**128.53**	11061	**11.411**	**969.29**	3113.2	**17.749**	**175.4**	4184.3	**24.184**	**173.02**	3919.1	**25.51**	**153.63**	444.29	**20.435**	**21.742**
CONV-4	885.74	**41.163**	**21.518**	10706	**14.264**	**750.56**	4084.3	**66.102**	**61.788**	5169.9	**85.97**	**60.136**	6408.8	**99.952**	**64.119**	827.22	**35.605**	**23.233**
DWCONV-1	838.46	**0.29416**	**2850.3**	10551	**0.28027**	**37647**	2775.9	**3.8011**	**730.28**	4108.2	**5.0464**	**814.09**	4844.8	**2.8047**	**1727.4**	571.46	**1.209**	**472.67**
DWCONV-2	1033.8	**0.47162**	**2192**	11812	**0.67874**	**17402**	2795.3	**2.5241**	**1107.4**	3814.5	**5.2036**	**733.06**	4160.9	**1.8825**	**2210.3**	470.53	**0.5241**	**897.79**
DWCONV-3	969.88	**0.30031**	**3229.6**	13759	**1.4601**	**9423.3**	2726.6	**2.158**	**1263.5**	3666.1	**4.2326**	**866.15**	3320	**3.8216**	**868.74**	506.63	**0.66237**	**764.87**
DWCONV-4	1000.7	**0.33429**	**2993.4**	10704	**0.81343**	**13159**	3474	**5.5112**	**630.34**	4577.5	**9.4771**	**483.01**	4197.4	**2.8767**	**1459.1**	449.89	**1.6**	**281.18**
MM-64	697.37	**0.38452**	**1813.6**	9099.5	**0.85309**	**10667**	3432.4	**7.3588**	**466.43**	4779.2	**13.103**	**364.75**	4417.6	**9.1708**	**481.7**	397.59	**1.3267**	**299.68**
MM-128	925.47	**1.578**	**586.47**	9044.8	**0.79484**	**11379**	4991.9	**11.153**	**447.57**	5754.2	**16.025**	**359.07**	6681.1	**13.956**	**478.72**	387.82	**1.1188**	**346.63**
MM-256	1118.5	**2.6925**	**415.41**	18119	**3.0202**	**5999.4**	5003.1	**18.511**	**270.28**	5770.9	**26.485**	**217.9**	6800.6	**27.446**	**247.78**	390.29	**5.1647**	**75.57**
MM-512	1262.3	**4.3405**	**290.83**	12336	**4.4847**	**2750.8**	4814.2	**7.4852**	**643.17**	5698.9	**12.25**	**465.22**	6471.7	**14.545**	**444.94**	1084.4	**9.5121**	**114**

Table 1: Compile times, in milliseconds, for LLVM and LoopNest. LoopNest performs the compilation orders of magnitude faster

	x86 based CPUs						Aarch64 (ARM) based CPUs (NEON)											
	AMD (AVX2)			Intel (AVX512)			Cortex A57			NVIDIA Denver2			Cortex A73			Apple M1		
	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio	LLVM	LN	Ratio
CONV-1	86.767	**87.638**	**1.01**	72.943	**162.48**	**2.2274**	11.021	**11.126**	**1.0095**	10.968	**14.782**	**1.3477**	**9.8466**	9.3994	0.95458	**98.121**	97.51	0.99377
CONV-2	45.765	**71.482**	**1.5619**	13.813	**156.78**	**11.35**	**11.276**	11.179	0.99134	13.407	**14.511**	**1.0824**	**9.8105**	9.2468	0.94254	**95.743**	92.027	0.96119
CONV-3	9.4946	**72.433**	**7.6289**	96.448	**184.4**	**1.9119**	8.1811	**10.183**	**1.2447**	6.4069	**13.27**	**2.0712**	6.4671	**7.9984**	**1.2368**	53.681	**83.252**	**1.5509**
CONV-4	3.307	**90.722**	**27.434**	123.73	**181.72**	**1.4687**	5.3246	**9.5687**	**1.7971**	4.3611	**12.532**	**2.8736**	2.9514	**7.686**	**2.6042**	46.806	**77.952**	**1.6654**
DWCONV-1	48.695	**62.541**	**1.2844**	48	**57.592**	**1.1998**	**6.3521**	6.2343	0.98146	10.243	**11.858**	**1.1577**	3.9988	**4.5019**	**1.1258**	**40.133**	39.01	0.97201
DWCONV-2	39.203	**53.465**	**1.3638**	28.302	**37.671**	**1.331**	5.1196	**5.7031**	**1.114**	7.1075	**7.9688**	**1.1212**	2.5178	**2.7729**	**1.1013**	42.865	**43.438**	**1.0134**
DWCONV-3	62.873	**84.848**	**1.3495**	57.544	**88.094**	**1.5309**	6.9338	**7.7209**	**1.1135**	10.65	**12.551**	**1.1785**	5.7644	**6.4571**	**1.1202**	70.796	**76.261**	**1.0772**
DWCONV-4	77.071	**84.21**	**1.0926**	125.04	**159.38**	**1.2747**	9.6875	**10.174**	**1.0502**	12.665	**14.717**	**1.1621**	7.6075	**8.0169**	**1.0538**	**81.588**	80.901	0.99157
MM-64	85.808	**102.2**	**1.191**	144.67	**187.65**	**1.2971**	12.751	**13.441**	**1.0541**	14.039	**15.754**	**1.1221**	9.523	**10.166**	**1.0675**	91.18	**199.81**	**2.1913**
MM-128	92.692	**102.5**	**1.1058**	168.84	**185.19**	**1.0969**	12.417	**12.496**	**1.0064**	14.253	**15.291**	**1.0728**	9.2462	**9.5062**	**1.0281**	91.763	**98.4**	**1.0723**
MM-256	92.862	**100.21**	**1.0791**	170.18	**182.46**	**1.0722**	**11.428**	11.401	0.99761	14.241	**14.645**	**1.0284**	**8.6863**	8.6103	0.99125	95.597	**99.902**	**1.045**
MM-512	90.189	**98.199**	**1.0888**	**160.42**	159.59	0.9948	6.9971	**8.3267**	**1.19**	**14.395**	13.894	0.9652	6.3364	**6.8905**	**1.0874**	89.618	**97.65**	**1.0896**

Table 2: Measured execution performances, in GFLOPS, for code generated by LLVM and LoopNest. LoopNest achieves comparable (within measurement error) or superior performances while taking a fraction of the time to generate the code.

4.3 Runtime experiments

In Table 2 we show the average run–times of the top 5 fastest schedules, found by our tuner. Our results suggest that LoopNest is generating code that has comparable or better performances than the one generated by LLVM. We additionally perform the comparison of the fastest schedules found by our tuner to the extremely efficient, hand-optimized ones. In nearly all cases LoopNest matched or exceeded the performances of the hand-optimized ones (data not presented).

4.4 Evaluating cost

Another key component of a machine learning-based compiler is the fast evaluation of generated code. In recent research, a cost model (some form of a neural network) is used to estimate the performance of a given schedule. Often this cost model is computationally more expensive than the program being optimized, as in [40]. This is because the very long compilation times of LLVM make direct benchmarking impractical. LoopStack lifts this limit. Using the LoopStack for direct benchmarking could take less time than running the cost model. To confirm this we ran a set of ML-based tuning algorithms using both LoopStack and a neural network cost model. In all of the searches — brute force, greedy with lookahead of 1 and 2, and beam search — the measured search times were comparable.

5 Conclusion

We presented LoopStack, an extremely efficient code generator tailored to machine learning based approaches for optimizing machine learning workloads. We showed that LoopStack has orders of magnitude faster compilation, even achieves better performance than hand-tuned libraries. Besides this, in some ML-based auto-tuners LoopStack can replace the cost models providing exact values of performances. This however doesn't mean that LoopStack is a replacement for ML-based approaches but rather its an enabler for more sophisticated ones.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[3] R ARM. Cortex-a57 software optimization guide. *ARM*, 2016.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, 3:181–193, 2021.

[5] Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[6] Henricus M Bouwmeester. *Tiled algorithms for matrix computations on multicore architectures*. University of Colorado at Denver, 2012.

[7] Gerald G Brown. Numerical performance of matrix inversion. 1975.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[11] Intel Corporation. Onednn. `https://github.com/oneapi-src/oneDNN`, 2020.

[12] Albert Einstein. Die grundlage der allgemeinen relativitätstheorie. In *Das Relativitätsprinzip*, pages 81–124. Springer, 1923.

[13] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14629–14638, 2020.

[14] Roman Gareev, Tobias Grosser, and Michael Kruse. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–27, 2018.

[15] Google. Xnnpack. `https://github.com/google/XNNPACK`, 2020.

[16] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[17] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[18] John A Gunnels, Greg M Henry, and Robert A Van De Geijn. A family of high-performance matrix multiplication algorithms. In *International Conference on Computational Science*, pages 51–60. Springer, 2001.

[19] Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. High order seismic simulations on the intel xeon phi processor (knights landing). In *International Conference on High Performance Computing*, pages 343–362. Springer, 2016.

[20] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991. IEEE, 2016.

[21] Alexander Heinecke, Hans Pabst, and Greg Henry. Libxsmm: A high performance library for small matrix multiplications. *Poster and Extended Abstract Presented at SC*, 2015.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[23] R Intel. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept*, 2014.

[24] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, 1988.

[25] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. Optimizing n-dimensional, winograd-based convolution for manycore cpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 109–123, 2018.

[26] Daya Khudia, Protonu Basu, and Summer Deng. Open-sourcing fbgemm for state-of-the-art server-side inference, 2018.

[27] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[28] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and Ponnuswamy Sadayyapan. Efficient tiled sparse matrix multiplication through matrix signatures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[29] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[30] Jakob Leben and George Tzanetakis. Polyhedral compilation for multi-dimensional stream processing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–26, 2019.

[31] Michael Metcalf. Fortran program optimization. Technical report, European Organization for Nuclear Research, 1980.

[32] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[33] Neungsoo Park, Wenheng Liu, Viktor K Prasanna, and Cauligi Raghavendra. Efficient matrix multiplication using cache conscious data layouts. In *Prof. of HPCMO User Group Conference*, 2000.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[36] MMG Ricci and Tullio Levi-Civita. Méthodes de calcul différentiel absolu et leurs applications. *Mathematische Annalen*, 54(1-2):125–201, 1900.

[37] Alexander Rush. *Tensor Considered Harmful*, 2018 (accessed August 26, 2020).

[38] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 89–103, 2022.

[39] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1049–1059. IEEE, 2014.

[40] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value learning for throughput optimization of deep learning workloads. *Proceedings of Machine Learning and Systems*, 3, 2021.

[41] Harold S Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, 100(1):73–78, 1970.

[42] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

[43] Field G Van Zee and Robert A Van De Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):1–33, 2015.

[44] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[45] Wikipedia contributors. Plagiarism — Wikipedia, the free encyclopedia. [Online; accessed 15-November-2021].

[46] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, 1991.

[47] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.

[48] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. *arXiv preprint arXiv:2006.06762*, 2020.

[49] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. A deeper look at fft and winograd convolutions, 2018.

[50] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. The anatomy of efficient fft and winograd convolutions on modern cpus. In *Proceedings of the ACM International Conference on Supercomputing*, pages 414–424, 2019.

[51] Aleksandar Zlateski and H Sebastian Seung. Compile-time optimized and statically scheduled nd convnet primitives for multi-core and many-core (xeon phi) cpus. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

A Tensor Contractions

Tensor operations, such as inner, outer, and cross product, as well as matrix multiplication, trace, transpose, and many other tensor operations can be expressed as special cases of contraction. In fact, with a few extensions, contractions can encode all the operations that the most popular deep neural networks rely on. We demonstrate how to express computations over tensors as generalized contractions. We also discuss the challenges inherent in optimizing (aka scheduling) these contractions.

A.1 Generalized Tensor Contractions

Before introducing generalized, d-dimensional tensor contractions, we first present a simple example in 2-dimensions – the matrix multiplication operation. Figure 1 demonstrates two different, though equivalent, ways that describe the matrix multiplication operation. The imperative pseudocode in Figure 1a shows the typical set of 3 nested for-loops iterating over dimensions i and j of the output matrix (C) and the *reduction dimension* k. Note the reduction over k is explicit in this notation.

In the tensor index (Einstein) notation [36, 12, 27] (Figure 1b), the i and j index variables appear on both the left– and right–hand side of the formula indicating that in order to compute the corresponding entry in C we index A and B with the same values of i and j. The index variable k, represents a reduction variable, as it appears solely on the right–hand side the formula. In particular, the for–loop over varying values of k for a specific assignment to i and j, and the corresponding aggregation (i.e. contraction), is left implicit. Similarly, the for–loops over varying values of i and j, needed to populate C, are left implicit.

We use generic $\oplus$ and $\otimes$ notation to emphasize that different operators can be used to instantiate a generalized matrix *multiplication* operation.

```
for i:
    for j:
        C[i, j] = alpha * C[i, j];
        for k:
            C[i, j] += beta * A[i, k] *
                B[k, j]
```

(a) Pseudocode for basic imperative matrix-multiplication.

$$C_{i,j} = \alpha C^{in}_{i,j} \oplus \beta(A_{i,k} \otimes B_{k,j})$$

(b) Equivalent declarative Einstein notation for matrix multiplication, with $\oplus = +$ and $\otimes = *$.

Figure 1: Matrix multiplication

We can generalize the matrix-multiplication example to d–dimensions, where our inputs are now d–dimensional tensors, to result in *tensor contractions*. Following notation presented in [14], let A, B, and C now be d-dimensional tensors (for potentially different d). Let the set of dimensions for A, B, and C be denoted by I_A, I_B, and I_C, respectively. Let I_{AC} be the dimensions of C present in A, while I_{AR} are reduction dimensions[1] in A. Similarly I_{BC} are dimensions of C present in B, while I_{BR} are reduction dimensions in B. We now write:

$$C_{I_C} = \alpha C^{in}_{I_C} \oplus \beta(A_{I_{AC} \cup I_{AR}} \otimes B_{I_{BC} \cup I_{BR}})$$

We further extend this notation to allow indexing into a tensor to be an affine transformation of the indices.

Finally, we extend our notation to allow for an element-wise operation that initializes the output tensor (*pre-operation*) and an element-wise operation that can transform a value before final storage into the output tensor (*post-operation*). We can now write

$$C_{(I_C)} = \text{post}(\text{pre}(C^{in}_{f_C(I_C)}) \oplus \beta(A_{f_A(I_{AC} \cup I_{AR})} \otimes B_{f_B(I_{BC} \cup I_{BR})}))$$

where f_C, f_A and f_B are affine transformations and pre/post are the element-wise pre- and post-operations. These concepts are useful for expressing biases and activation functions in machine learning workloads.

[1] Dimensions contracted by a user-defined operation, such as the shared dimension of the input matrices in a matrix multiplication.

```
import loop_tool as lt
M, N, K = lt.Var("m"), lt.Var("n"), lt.Var("k")

A = lt.Tensor([M, K])
B = lt.Tensor([K])
C = lt.Tensor()

C[m, n] += A[m, k] * B[k]
```

Figure 2: LoopTool's Python embedded declarative DSL

With these extensions, we can not only express GEMM, GEMV, and GEVM operations, but many more operations as well. Such as:

- Convolutions – $O_{r,c} = I_{r+k,c+j} \cdot \omega_{k,j}$
- pooling – $O_{r,c} = \max(I_{2r+k,2c+k})$
- reductions – $O_r = I_{r,c}$
- transpositions – $O_{r,c} = I_{c,r}$
- concatenations – $O_{r,c'} = I^1_{r,c'}; O_{r,|c'|+c''} = I^2_{r,c''}$
- broadcast – $O_{r,c} = I_r$

A.2 Optimizing Generalized Contractions

The way we carry out these generalized contraction computations has a significant impact on performance. For example, a naive 3-nested for-loop implementation of matrix multiplication of figure 1a will result in long running times for non-trivially sized matrices. In contrast, efficient implementations, as proposed by many researchers over the past few decades [16, 33, 39, 29, 18, 25, 50], might modify the imperative loops to tile (i.e. split into sub-matrices of appropriate sizes) the input matrices to align with architecture-dependent resources such as cache sizes, re-order loop dimensions to re-use data in the innermost loop, exploit architecture features such as vector instructions to operate on multiple elements at a time, and emit extended instructions such as fused-multiply-add, which combine multiplication and accumulation at the instruction level. Further, the data might be kept in *exotic* memory layouts, such as the channel–interleaved format often used on Intel machines [11, 51], or the matrix tile interleaved format proposed by Jia et al. [25, 50].

Typical scheduling operations include: re-ordering, splitting, fusing, parallelizing, vectorizing, and unrolling loops [35, 17]. The scheduling options for a single task such as matrix-multiplication highlight the combinatorial challenge underlying scheduling. Different computations and platforms require varying schedules to deliver performance gains. This results in a challenging optimization problem. Historically, varying techniques have been put forward to tackle this optimization, ranging from expert-based manual optimization, analysis-driven heuristic optimization [17, 30], to recent advances in data-driven automated schedule exploration [8, 2]

B Frontend (LoopTool) Design

In order to allow users to effortlessly describe the desired computation of neural network workloads using LoopStack, we built a domain specific language (DSL) embedded in Python and a compiler front-end that we termed LoopTool. LoopTool exposes a declarative API for user definition of computation and operates on an IR composed of an annotated dataflow graph (DFG) consisting of N-dimensional tensor operations. The DFG describes the underlying computation, memory layouts and execution. LoopTool then lowers the DFG into a series of loops, which are then compiled with LoopNest.

B.1 Declarative API

Figure 2 shows an example of matrix multiplication in LoopTool's declarative Python DSL. The expression `lt.Var("m")` defines an indexing variable with the corresponding name. Next, the expression `lt.Tensor([M, K])` defines a 2-dimensional tensor. Note that LoopTool uses symbolic (i.e.

named) dimensions [37], which simplify indexing semantics and encourages a simple interaction model for manipulating traversal and memory layouts of higher dimensional tensors. The expression `C[m, n] += A[m, k] * B[k]` defines computation using the aforementioned tensor (Einstein) notation; here `k` is a reduction dimension. LoopTool's computation language supports element-wise computations (with broadcast semantics), associative reductions across arbitrary dimensions, and a restricted set of indexing semantics (see Section B.2.3).

B.2 Intermediate Representation (IR)

LoopTool's DFG is based on an intermediate representation with annotations on each node. Each node is associated with its output – a virtual buffer that is materialized according to the user provided schedule. Figure 3 (left) demonstrates a matrix multiplication without annotations. A node's output size is not materialized until the IR is lowered to loops. The materialization logic always attempts to minimize the total memory used. For example, two nodes operating on virtual buffer of size N in a shared loop over N do not need to allocate the full N elements of memory for their intermediate. Instead, the intermediate will be of size 1 and reused N times. This can be seen in Figure 2.

LoopTool's DFG has three fundamental types of nodes.

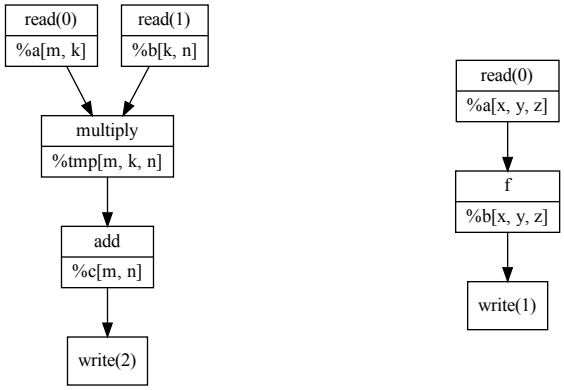

Figure 3: Matrix multiplication in LoopTool (left). A point-wise application of the function f across all three dimensions of `%a` (right).

B.2.1 Read/write nodes

Read and write nodes denote reads or writes from and into user provided memory, as well as associated layouts and sizes. These nodes contain an ordered list of symbolic dimensions. The ordered list of dimensions represents a row-major order (lexographical order) of either input or output memory. Because LoopStack is embedded in either Python or C++ applications, these nodes are used to interface with other operations that are not handled by LoopStack. An example would be the specification of NCHW[34] or NHWC[1] layouts for the inputs to convolution operations, both of which are well handled and trivially manipulated in the LoopTool IR. In the IR, read nodes take no inputs whereas write nodes take a single input and have no outputs. These are effectively source and sink operations in the graph.

B.2.2 Arithmetic nodes

Arithmetic nodes operate on one or multiple input virtual buffers and output a single virtual buffer. The ordered list of dimensions associated with these nodes denotes how output memory should be laid out (given the corresponding scope of their execution).

Arithmetic nodes, unlike read and write nodes, can take multiple inputs. This represents the typical operation applied to the inputs. For example, addition of two inputs works as expected to yield a single output.

Extending this concept to higher dimensions forces us to consider when two inputs do not have the same dimensions. Due to the symbolic nature of the dimensions in LoopTool, we can distinguish two dimensions of the same size as being mathematically distinct. To handle application of arithmetic in higher dimensions we employ implicit broadcasting akin to Numpy[32] semantics. Dimensions not present in the output are implicitly reduced over according to the arithmetic of the associated node.

B.2.3 View nodes

The final type of node deals with the representation of symbolic indexing constraints over tensor dimensions, allowing powerful view semantics. Any affine combination of iteration over the output dimensions can be used to index into an input dimension. This enables us to represent windowed operations as well as concatenations. By using index constraints rather than index equations, we preserve the meaning of the underlying computation and can freely split and permute variables and layouts. Further, by keeping indexing math symbolic, we can still apply the scope based memory minimization logic.

B.3 Lowering to loops

LoopTool lowers its IR to an internal loop tree structure before invoking LoopNest. This is done by traversing the DFG topologically and eagerly emitting loops that are required for each node. A reference to the current innermost loop is maintained throughout this entire process, with each subsequent loop nesting inside the reference. If any node requires a loop that is an ancestor to the current reference, there is no need to emit the loop again (it would be incorrect to do so) and it is skipped. This naturally induces loop coalescing (typically referred to as loop fusion).

Consider nodes shown in Figure 3 (right). We can assume an annotation of `%a` and `%b` both with loop order `[x, y, z]`. When visiting `%a`'s node, we emit the loop tree shown in Listing 1.

Later, while visiting `%b`'s node, we note that the reference (which is at location `%a`) contains a loop for `x`, `y`, and `z` in order. We thus reuse all loops and implicitly "fuse" node `%b`. This is shown in Listing 2

We find the nodes can be executed in the same innermost loop. The resultant allocation size of `%a` would be 1 in this case.

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
      ...
```

Listing 1: Lowering `%a` emits three new loops.

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
      %b[x, y, z] = f(%a[x, y, z])
```

Listing 2: Lowering `%b` reuses all loops.

However, if the nodes had different loop annotations, such as `%b` annotated with `[x, z, y]`, we would not be able to share every loop. The memory allocated for `%a` in that case would be of size $|y| \cdot |z|$ This is shown in Listing 3.

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
  iter z:
    iter y:
      %b[x, y, z] = f(%a[x, y, z])
```

Listing 3: Only loop x is shared across the two nodes.

In the case of reductions, we cannot always share loops. Consider a reduction node %R over variable z with loop order [x, y, z] depended on by by %a with the same loop order; see Listing 4. The loop for z is required to run twice for correctness. This type of behavior, while necessary for reductions, may be preferable in other contexts as well.

```
iter x:
  iter y:
    iter z:
      %R[x, y] = reduction(...)
    iter z:
      %a[x, y, z] = %R[x, y] + ...
```

Listing 4: Sharing loop z between %R and %a would be mathematically incorrect.

Manually preventing loop sharing can induce larger intermediate memory allocations. This is often beneficial when computation benefits from packing memory into cache–friendly layout before computation. To express this, LoopTool has a second form of annotation for nodes called *staging* that prevents reuse of specific loops. In listing 2, %a and %b share an entire loop nest. If we were to *stage* the loop for %a over z, the resultant lowering would increase the allocation size of %a to $|z|$ (Listing 5).

```
iter x:
  iter y:
    iter z:
      %a[x, y, z] = read(x, y, z)
    iter z:
      %b[x, y, z] = f(%a[x, y, z])
```

Listing 5: z is staged, so %a is materialized with an allocation of size $|z|$.

C Backend (LoopNest) Design

LoopNest is a domain specific compiler for a series of nested loops, with user specified innermost operation. LoopNest is designed to have extremely short compilation times and use well known and studied high-performance computing (HPC) optimizations to generate high performance code.

C.1 LoopNest's HPC Philosophy

While traditional compilers typically perform multiple optimization passes, some of which might be repeated, LoopNest's is designed to only do a limited number of optimizations, but do them very well. These include HPC optimizations that are commonly used in expert designed, custom assembly, or code generator primitives for specific problems, such as matrix multiplications [43, 5, 44, 20, 21, 26] and other primitives used in machine learning [10, 49, 50, 25, 11, 13, 19]. Additional HPC optimization might include instruction reordering, r–sum, and other optimizations suggested by optimization manuals for the target hardware [23, 3].

LoopNest's generated code directly reflects the computation requested by the user: in particular, operations are performed in user-specified order. Generating code directly from the user-defined execution order simplifies code generation logic, which results in a more intuitive mapping between

the quality of the provided execution order and the observed performance of the code. In addition, the provided (high level) information about the loop order and sizes are used for LoopNest's simple register allocation approach, described below.

While having default choices based on the target hardware manuals, LoopNest optionally delegates the choice of which loops to be unrolled. The user may override the hardware tuned default maximum number of unrolled instructions, and LoopNest will determine the outermost loop such that the number of unrolled instructions doesn't exceed the user specified value.

Vectorization

LoopNest assumes that the user intent is to vectorize the innermost loop in the user provided schedule. This design simplifies the logic, while not introducing any limitations to the user – the elements of the vector operation are executed at the same time, thus naturally belong to the innermost loop. LoopNest will, thus, attempt to vectorize the innermost loop. However, in certain scenarios, when the data accessed inside the innermost loop is not contigious, and the target hardware doesn't support efficient gather operations, LoopNest will fallback to scalar operations.

No–Spilling of the Resulting Tensor

The concept of tiling (or "blocking") for the multiple levels of a cache hierarchy and the register file is a well known optimization technique [16, 18, 29, 39]; referred to as *Cache Blocking Techniques* in the Intel's optimization manual [23]. LoopNest exposes this common HPC optimization, where the subset of the output tensor is kept in register file [11, 21, 25, 49, 50, 20, 19]. LoopNest never produces code that spills the content of the register file to the stack, an approach that is commonly used in traditional compilers, such as LLVM or GCC. Spilling the content of the register file to stack puts pressure on the hardware's load/store units, preventing full hardware utilization.

LoopNest does not decide on blocking or tiling sizes, nor on the size used for the data kept in registers (register blocking). LoopNest instead identifies the outermost user provided loop for which the all compute can be performed with a subset of the output tensor kept in registers. Thus, it's up to the user to provide a well chosen loop order and sizes – one where the values kept in the register file can be reused many times. This approach gives the user a greater control, with more predictable performance as compared to the case when spilling is allowed.

C.2 Single Operand LoopNest

LoopNest also provides functionality for generating efficient code for a simplified tensor contraction, where there are no reduction dimensions, and only one input is provided. This functionality is typically used for *reshaping* a tensor (such as numpy's **reshape** function), but can also be used for extracting a subset of a tensor into a smaller tensor, or broadcasting elements along a tensor dimension. This functionality is required by LoopStack in order to allow the user's schedules that prefer to reorganize the memory for faster access [16, 18, 29, 39]

C.3 Generalizing to computations with multiple loop nests

Some computations or their schedules can result in a sequence of multiple nested loops that may share a set of outer loops, effectively forming a tree of loops. To generalize our approach to these workloads, we developed a loop tree interface. The loop tree interface provides a simple API to build up a tree, where inner nodes correspond to for-loops, and leaves correspond to an innermost computation over tensors or a transposition of tensors. LoopNest then compiles all independent nests and executes the tree. The final result is a function that can be called with the appropriate input, intermediate, and output tensors to realize the computation defined by the tree.