Tensor Compilers for AI and HPC

Paul Mifsud
Jülich Supercomputing Centre (JSC)
Forschungzentrum Jülich
Juelich, Germany
p.mifsud@fz-juelich.de

Stefan Kesselheim
Jülich Supercomputing Centre (JSC)
Forschungzentrum Jülich
Juelich, Germany
s.kesselheim@fz-juelich.de

***Abstract*—Interest in tensor compilers has grown substantially in the last seven years motivated by applications in machine learning, where they are better known as machine learning compilers. However, adoption and use of machine learning compilers is not universal, and in instances where they are used, they are often coupled with hand written kernels for increased optimization. In this chiefly pedagogical work, we dive deep into tensor compilers and adjacent systems and technologies to provide a holistic systems understanding of tensor program optimization. To facilitate reasoning about tensor program optimization we define our own tensor operator taxonomy. By focusing on modern GPU architectures, and taking the NVIDIA H100 SXM GPU as an example, we condense GPU kernel optimization into two principles, from which many optimization techniques emerge. We then study tensor compilers in detail, focusing primarily on the intermediate representations and the optimization techniques they employ. The PyTorch 2.0 compiler backend, TorchInductor, and the Triton compiler, are used as illustrative examples. We find that kernel fusion is the most consequential optimization. Having the full picture, we are then able to answer the pertinent questions: What are the benefits of using a tensor compiler? How do tensor compilers optimize tensor programs, and what optimizations are tensor compilers able to apply? What are the limitations of the current generation of tensor compilers? And how can tensor compilers be augmented for increased optimization?**

***Index Terms*—ml compilers, tensor compilers, ml systems, GPU kernels, parallel programming, high performance computing**

I. Introduction

A tensor compiler [1]–[10] takes as input a directed acyclical graph (DAG) of operations on multi-dimensional arrays, for which the terminology "tensors" is often used. The compilation reduces the DAG of tensor operations into a quotient graph of kernels, which are optimized codes targeting hardware accelerators. Figure 1 depicts this process at a high level.

Graph level optimizations are transformation of the DAG, the most consequential of which is operator fusion [11]–[13]. As a consequence the DAG of generated kernels is a quotient graph of the DAG representing the original tensor program.

In the past 6 to 8 years, interest in tensor compilers grew rapidly driven by the need to optimize machine learning (ML) workloads. As such tensor compilers are maybe better known as Machine Learning Compilers [1]. However, the essence of what ML compilers are is applicable outside the domain of ML.

A. Methodology

Driven by the goal of providing clear and deep understanding of tensor compilers targeting real workloads, we opt to specialize our study, by for example focusing on one GPU architecture (NVIDIA H100 SXM) [14] and one tensor compiler design (PyTorch 2.0 TorchInductor) [7], rather than to present the work in more general terms.

II. Tensor Operator Taxonomy

It is typical for tensor compiler system design to include a classification of tensor operators [5], [15]–[17], either implicitly or explicitly. In this work we opt to define, our own very general taxonomy where we group operators by memory access patterns. We provide examples of *pointwise, reduction, contraction, convolution, scatter/gather* and *broadcast* operators. These require *free, reduction, indirect* and *offset indexes*. Grouping operators by access indexes proves helpful when one wants to reason about which operators can be fused with other operators (i.e. one operator's output being another operator's input), which is not trivial in the context of the Single Instruction Multiple Thread (SIMT) paradigm [18].

III. The GPU: NVIDIA H100 SXM as an example

Using the NVIDIA H100 SXM GPU [14] we present an overview of modern GPU architectures. In particular we want to point out the massively parallel nature of the hardware, the memory hierarchy, the different pipelines, the scheduler, the notion of a warp, and most importantly the relationship between compute throughput and global memory bandwidth. We condense GPU programming into two principles, namely first and foremost to optimize global memory traffic [19]–[21] and secondly to optimize for latency hiding [22], [23]. Most known GPU kernel optimization techniques originate from one of these two principles.

IV. Tensor Compilers: TorchInductor and Triton as examples.

By studying TorchInductor [7] in depth and OpenXLA[1] [8] to a lesser extent, we identified the core inner workings and system design ideas in contemporary tensor compilers. We decided to use TorchInductor as a case study. In TorchInductor

[1]We did not include details about OpenXLA in the poster due to space constraints, however from our work we found out that the system design at a high level is quite similar.

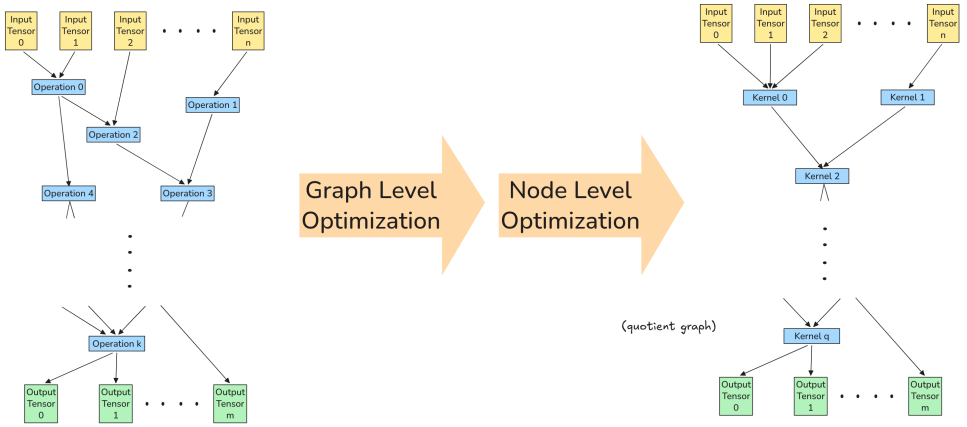


Fig. 1. Tensor Compiler optimization stages. A graph level optimization stage transforms the graph, while node level optimization optimize the (possibly) fused nodes into optimized accelerator kernels.

there are two intermediate representations during the lowering process. The LoopLevel IR [7] describes how each individual tensor operator can be implemented on an arbitrary parallel processor. The operation is described in the shape of a nested loop, with the loop body describing the per thread operation, and the indices providing the memory access patterns. This is further lowered into SchedulerNodes [7] at which point scheduling is performed. In particular, SchedulerNodes are fused together depending on eligibility. This operator fusion is the key optimization that the compiler performs and its motivation follows directly from the first principle of GPU programming, above. By fusing operators the same work is performed, but the total global memory traffic is reduced, because the output from the producer node is immediately used up by the consumer node, avoiding a roundtrip to global memory.

TorchInductor then uses Triton [24] to generate the kernels. It is at this stage that node level optimization is performed. Within the Triton compiler, more optimizations are applied to both optimize use of memory bandwidth (e.g. global memory coalescing, vectorization, tiling, use of shared memory) and also to optimize for latency hiding by for example using warp specialization and software pipelining [19], [25]–[27].

V. Evaluation and Conclusion

Our initial interest in tensor compilers originated from machine learning in the domain of molecular dynamics. Indeed, tensor compilers are even more powerful for these applications which tend to contain more opportunities for operator fusion than in more hot application areas such as the training or inference of Large Language Models or Diffusion Models.

In our experiments we obtain speedups of 2x to 3x for very simple models such as SchNet [28], while even larger speedups are possible for equivariant models [29]. By contrast, more modest speedups are typically obtainable when compiling transformer based architectures, [30], [31], which is simply due to the fact that there is less opportunity for operator fusion in those architectures and the fact that the sub-components of the architectures which are compute intensive, such as the attention mechanism are using optimized kernels [21], [27] in both eager and compiled modes.

To conclude we believe we have adequately answered the questions we set out to investigate. Namely:

1. Tensor compilers are useful and indeed necessary when the tensor program includes a substantial number of operators of low arithmetic intensity, which can be fused together, and that moreover this task is performed adequately by tensor compilers and it is therefore ill-advised to program kernels by hand whose sole task is to perform operator fusion.

2. The main graph level optimization is operator fusion, which turns out to work symbiotically with gradient checkpointing [11]. Node level optimization includes most of the known kernel optimization techniques, however work is still in progress at this level.

3. Limitations include chiefly opportunities for operator fusion that the compiler is not equipped to handle. Other limitations include cutting-edge node level optimization techniques that might not already be integrated in the kernel compiler of choice.

4. Compilers can be augmented to be able to detect and perform more complex operator fusions, or to use highly optimized hand written kernel templates.

References

[1] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.

[2] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron *et al.*, "Theano: Deep learning on gpus with python," 2011.

[3] R. Wei, L. Schwartz, and V. Adve, "Dlvm: A modern compiler infrastructure for deep learning systems," *arXiv preprint arXiv:1711.03016*, 2017.

[4] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[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, pp. 578–594.

[6] H.-I. C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik, and S. Laurenzo, "Tinyiree: An ml execution environment for embedded systems from compilation to deployment," *IEEE micro*, vol. 42, no. 5, pp. 9–16, 2022.

[7] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM international conference on architectural support for programming languages and operating systems, volume 2*, 2024, pp. 929–947.

[8] A. Sabne, "Xla: Compiling machine learning for peak performance," 2020.

[9] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan *et al.*, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Molecular Physics*, vol. 104, no. 2, pp. 211–228, 2006.

[10] 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)*. IEEE, 2021, pp. 2–14.

[11] H. He and S. Yu, "Transcending runtime-memory tradeoffs in checkpointing by being fusion aware," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 414–427, 2023.

[12] E. C. Lewis, C. Lin, and L. Snyder, "The implementation and evaluation of fusion and contraction in array languages," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, 1998, pp. 50–59.

[13] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, 2010, pp. 344–350.

[14] NVIDIA Corporation, "NVIDIA H100 Tensor Core GPU Architecture," NVIDIA Corporation, White Paper, 2022. [Online]. Available: https://www.advancedclustering.com/wp-content/uploads/2022/03/gtc22-whitepaper-hopper.pdf

[15] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[16] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[17] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 883–898.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[19] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[20] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[21] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 16 344–16 359.

[22] S.-Y. Lee and C.-J. Wu, "Characterizing the latency hiding ability of gpus," in *2014 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2014, pp. 145–146.

[23] V. Volkov, *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[24] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[25] H. Chen, B. Fan, A. Collins, B. Hagedorn, E. Gaburov, M. Masuda, M. Brookhart, C. Sullivan, J. Knight, Z. Zhang *et al.*, "Tawa: Automatic warp specialization for modern gpus with asynchronous references," *arXiv preprint arXiv:2510.14719*, 2025.

[26] R. Soi, R. Yadav, F. Kjolstad, A. Aiken, M. M. Dehnavi, M. Garland, and M. Bauer, "Optimal software pipelining and warp specialization for tensor core {GPUs}," in *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, 2026, pp. 1875–1890.

[27] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 658–68 685, 2024.

[28] K. Schütt, P.-J. Kindermans, H. E. Sauceda Felix, S. Chmiela, A. Tkatchenko, and K.-R. Müller, "Schnet: A continuous-filter convolutional neural network for modeling quantum interactions," *Advances in neural information processing systems*, vol. 30, 2017.

[29] C. W. Tan, M. L. Descoteaux, M. Kotak, G. d. M. Nascimento, S. R. Kavanagh, L. Zichi, M. Wang, A. Saluja, Y. R. Hu, T. Smidt *et al.*, "High-performance training and inference for deep equivariant interatomic potentials," *arXiv preprint arXiv:2504.16068*, 2025.

[30] P. Vellaisamy, T. Labonte, S. Chakraborty, M. Turner, S. Sury, and J. P. Shen, "Characterizing and optimizing llm inference workloads on cpu-gpu coupled architectures," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2025, pp. 49–61.

[31] S. Paul, A. Jain, and B. Bossan, "torch.compile and Diffusers: A hands-on guide to peak performance," PyTorch Blog, Jul. 2025, accessed: Jul. 28, 2026. [Online]. Available: https://pytorch.org/blog/torch-compile-and-diffusers-a-hands-on-guide-to-peak-performance/