Control Flow Operators in PyTorch

Yidi Wu [1] **Thomas Ortner** [2] **Richard Zou** [1] **Edward Yang** [1] **Adnan Akhundov** [1] **Horace He** [3] **Yanan Cao** [1]

Abstract

Representing control flows in machine learning (ML) compilers and intermediate representations (IRs) has been a long-standing problem, with many ML compilers opting to avoid supporting them altogether. A common practice is to trace the model and get a straight-line computational graph, for example, by specializing if predicates or by unrolling loops. Although this strategy may work well in some cases, it can result in performance issues and long compilation times. Furthermore, it is problematic in cases where the control flow of the program depends on data or a dynamic shape.

In this paper, we present the PyTorch control flow operator library, which addresses the challenge through the introduction of five control flow operators to PyTorch. We will explain their usage, present real world use cases and demonstrate their benefits such as the ability to capture dynamic control flows in the PyTorch 2 IR with `cond` and reduce compilation time and peak memory usage when rewriting loops with `map`, `scan`, `associative_scan` and `while_loop`.

1. Introduction

Python is the de facto authoring language for most machine learning (ML) models (Agrawal et al., 2019; He, 2019). PyTorch (Paszke et al., 2019b), one of the most widely adopted ML libraries, allows users to write native Python code for rapid model iteration. Users can compose models with complex control flows using native Python constructs like `if-else` statements, `for` loops, and `while` loops.

Many ML compilers only support optimizing a straight-line computational graph without any control flow due to the inherent complexity (Chen et al., 2018; Jia et al., 2019; Ma et al., 2020; Niu et al., 2021; Wang et al., 2021; Zheng et al., 2022). Consider the following simple control flow:

```
if mod.static_config == 0:
    return f(x)
return g(x)
```

The two branches above might have entirely different code paths and might execute different operations. If we were to compile both branches whenever we encounter such an if-statement, the number of code paths to compile will explode exponentially and very quickly become intractable for any practical model.

Most recently, PyTorch introduced a Just-in-Time (JIT) compiler (i.e. PT2) that accelerates native Python models by analyzing bytecode, extracting straight line computational graph and compiling the graph into dynamically generated kernels (Ansel et al., 2024). To address the control flow challenge, PT2 uses the **specialization and guard** strategy for static control flow. Specifically, the compiler traces the model and chooses a code path based on the value of current predicate, i.e. specialization. In the above code example, the compiler takes the branch of `f(x)` and introduces a guard check for `mod.static_config == 0`. At runtime, the compiler will verify the guard, e.g., if `mod.static_config == 0`, before executing the optimized code, and if the check fails it restarts compilation. The same strategy also applies to loops. The compiler unrolls the loops and runs each iteration sequentially and guards on the number of iterations. This design produces a straight-line computational graph. The compiler can focus on optimizing tensor operators.

However, this design doesn't resolve all control flow problems, especially in cases where the control flow is dynamic.

- **Data dependent control flow** is the case where the predicate depends on the value of tensor. In this case, the compiler cannot unroll the `while` because the number of loop iterations depends on the value which is only available at runtime. It cannot choose a branch if the predicate is a tensor, either. The PT2 compiler handles it by a graph break and falling back to Python see (Ansel et al., 2024). This also makes it impossible to run the compiled model in an environment without Python, e.g. on edge devices or servers for inference.

[1]Meta [2]IBM Research Europe - Zurich [3]Thinking Machines Lab; Work done while at Meta. Correspondence to: Yidi Wu <yidi@meta.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25).

- **Dynamic shape dependent control flow** is the case where the number of loops or predicate is based on a dynamic size of a tensor. Specialization will cause the compiled code to only work for a specific size and the compiler has to recompile the model whenever it changes.

- **Large computational graph**. Even if the loop has static number of iterations, unrolling a large loop can result in a computational graph whose size grows linearly with number of iterations despite the computational graph in each iteration may be the same. This leads to unnecessary long compilation time and potentially significant memory overheads.

Control flow operators have been implemented in other ML frameworks such as TensorFlow (Agrawal et al., 2019), Theano (Al-Rfou et al., 2016), and JAX (Bradbury et al., 2018). Compared to native Python control flow, these operators impose restrictions on valid data types and syntax, making them less flexible but easier to optimize (Moldovan et al., 2019). Recent works also studied efficient compilation (Zhang et al., 2023; Zheng et al., 2023) and automatic model rewriting with control flow operators (Moldovan et al., 2019; Jeong et al., 2019; Kim et al., 2021).

To address the above challenges, we introduce the **control flow operator library** into PyTorch in this work. In particular, we introduce five control flow operators: `cond`, `while_loop`, `map`, `scan`, and `associative_scan`, along with their restrictions and rationale. We then describe their autograd and codegen implementation. Finally, we demonstrate real use cases where control flow operators drastically reduce compilation time, save peak memory consumption, and reduce recompilation.

2. Control Flow Operators

The interface of control flow operators follows existing machine learning libraries such as TensorFlow (Yu et al., 2018; Agrawal et al., 2019) and JAX (Bradbury et al., 2018). For simplicity, we uses tensor inputs to illustrate the interface but nested tuple/dict/list of tensors are allowed.

- **cond(pred, true_fn, false_fn)** returns `true_fn()` if the predicate is True, otherwise `false_fn()`. By default, both functions take no arguments and access inputs via closure. Optionally, users can pass operands as `cond(pred, true_fn, false_fn, operands)`, in which case `cond` returns `true_fn(*operands)` or `false_fn(*operands)`.

- **while_loop(cond_fn, body_fn, operands)** first executes `cond_fn(*operands)`, which must return a scalar boolean tensor indicating whether to continue. If True, it executes `body_fn(*operands)`, which must return the operands for the next iteration.

```
def while_loop(cond_fn, body_fn, operands):
    while cond_fn(operands):
        operands = body_fn(operands)
    return operands

x = tensor(0)
# r = tensor(5)
r = while_loop(lambda x: x < 5,lambda x: x + 1,[x])
```

- **scan(combine_fn, init, xs)** applies a cumulative operation to tensor `xs` using initial value `init` and combination function `combine_fn`. The `combine_fn` takes the current carry and a slice of `xs`, returning the next carry and an output y. The final output is a tuple of stacked carries and outputs, enabling efficient computation of cumulative sums, products, or other accumulations.

```
def scan(combine_fn, init, xs):
  carries = []
  ys = []
  carry = init
  for x in xs:
    carry, y = combine_fn(carry, xs)
    carries.append(carry)
    ys.append(y)
  return torch.stack(carries), torch.stack(ys)

xs = torch.arange(1, 5)
init = tensor(2)
# r = (tensor(48), tensor([2, 4, 12, 48]))
r = scan(lambda x, y: (x * y, x * y), init, xs)
```

- **associative_scan(combine_fn, xs)** is similar to `scan` but differs in three ways: 1) it uses the first slice of `xs` as the initial value rather than taking an init input, 2) `combine_fn` must be associative (e.g., add, matmul), and 3) `combine_fn` only returns the carry. The associativity constraint enables more optimization opportunities.

```
def associative_scan(combine_fn, xs):
  carry = xs[0]
  carries = [carry]
  for x in xs[1:]:
     carry = combine_fn(carry, x)
     carries.append(carry)
  return torch.stack(carries)

xs = torch.arange(1, 5)
# r = tensor([1., 2., 6., 24.])
r = associative_scan(lambda x, y: x * y, xs)
```

- **map(fn, xs)** computes `fn` on each slice of `xs` and returns the stacked output.

```
def map(fn, xs):
  return torch.stack([fn(x) for x in xs])
```

2.1. Restrictions

The PT2 compiler uses a simple straight-line IR of tensor operators that is easy to maintain and optimize (Ansel et al., 2024). Since Python's flexibility includes syntax not representable in this IR, we impose the following restrictions for control flow operators:

- **Structure Match.** Control flow operators require inputs and outputs to match structure in two cases:
  - **Output match:** For `cond`, both `true_fn` and `false_fn` must return the same container structure (e.g., both return a list of 2 tensors). This prevents downstream divergence that could create exponential branches.
  - **Carry match:** For loops (`while_loop`, `scan`, `associative_scan`), carries must have matching structure across iterations. Since the IR assumes fixed input structure for `combine_fn`, `while_loop` outputs must match next iteration inputs, and `scan`'s init must match the output carry structure.
- **Tensor Match.** Corresponding tensors in the structure must have matching device, dtype, and dimensionality. The PT2 compiler specializes and guards these metadata similarly to control flow. Tensors with different sizes in the same dimension are automatically inferred as dynamic for that dimension.
- **No side effects.** Function arguments to control flow operators must not create side effects which cannot be represented in the IR, such as mutating external objects (appending to lists, deleting dictionary keys, or setting global attributes). While prior work attempted to support restricted side effects (e.g., TensorArray (Agrawal et al., 2019; Moldovan et al., 2019)), achieving full Python expressiveness would require supporting complete Python syntax, making the IR complex and difficult to maintain (DeVito & et al, 2019).

2.2. Pytorch Integration

The core design principle for integrating with PyTorch is to **reuse existing PyTorch infrastructure whenever possible**. This approach (1) leverages robust, well-tested modules to enhance reliability, and (2) minimizes additional instrumentation, improving long-term maintainability of PyTorch.

Autograd. We integrate control flow operators into PyTorch's autograd system using `autograd.Function` and customizing the `forward()` and `backward()` methods (Paszke et al., 2019a) for each of them.

A key observation that simplifies integration is that operators like `map`, `cond`, and `scan` exhibit **symmetry** between forward and backward, i.e. their backward can be formulated using the same operator with different inputs and functions. For example, `cond(pred, true_fn, false_fn)`'s backward is `cond(pred, true_fn_bw, false_fn_bw)`, `map`'s backward is another `map`, and `scan`'s backward is a reverse-time `scan`. We provide detailed descriptions of backward implementations in the appendix A-E. The `while_loop` operator's autograd implementation is ongoing at the time of writing.

The symmetry reduces the problem to computing gradients for user-defined functions, a core PyTorch functionality. It roughly looks like:

```
def bw_fn(*fw_args, *fw_out_gradient):
fw_out = fn(*fw_args)
return torch.autograd.grad(fw_out, fw_args,
    fw_out_gradient)
```

where `torch.autograd.grad` computes gradients of `fw_out` with respect to `fw_args` using `fw_out_gradient`.

However, always recomputing `fn` in backward can be expensive. We can checkpoint forward intermediate results to save re-computation overhead. Existing checkpointing strategies (He & Yu, 2023; Chen et al., 2016; Jain et al., 2020; Zheng et al., 2020; Zhang et al., 2022) that balance memory consumption and recomputation overhead can be easily integrated with our operators, except for `cond` and `while_loop` where data-dependent intermediate activations present challenges we leave for future work.

Codegen. Each control flow operator has a corresponding IR node, but during code generation, we lower map and scan into `while_loop`, supporting codegen only for `cond`, `while_loop`, and `associative_scan`. We generate `cond` and `while_loop` using the host language's native statements: `if-else` for `cond` and `while` for `while_loop` in Python. For `associative_scan`, we implement an optimized elementwise implementation leveraging its associativity rather than lowering to `while_loop`.

3. Use Cases

In this section, we present early results of applying control flow operators to real-world use cases. Our objective is to demonstrate the benefits of representing control flow in PT2 IR versus native implementations, rather than achieving state-of-the-art performance compared to manually optimized kernels. We believe techniques from hand-written kernels can be incorporated into the PT2 compiler to achieve comparable or better performance (Dong et al., 2024).

Invalid Value Clamping. In practice, tensors may have NaN or infinity due to reasons such as invalid mathematical operators (e.g. division by zero), invalid data. Invalid values can propagate through calculations and lead to wrong or

unstable results during training and inference. We could

```
x = torch.cond(
    (torch.isinf(x).any() or torch.isnan(x).any()),
    lambda:torch.clamp(
        x, min=-MAX_V, max=MAX_V),
    lambda:x.clone(),)
```

Chunked Loss Computation. Recent work uses "chunking" to reduce memory usage in loss computation layers (Hsu et al., 2024). This approach chunks inputs into smaller sizes, fuses the final linear layer with loss computation, and calculates backward gradients during the forward pass, significantly reducing intermediate activation memory.

Although customized Triton kernels can implement these approaches (Hsu et al., 2024), writing models in native PyTorch and letting PT2 compile them is simpler. A naive PyTorch implementation causes PT2 to unroll loops along the chunked dimension, creating long sequences of repeated operations. Rewriting loops as `scan` can significantly reduce computational graph size and compilation time. Figure 1 compares compilation time and peak memory usage as the chunk count increases for `scan` vs a naive loop. See more details in appendix F.3.

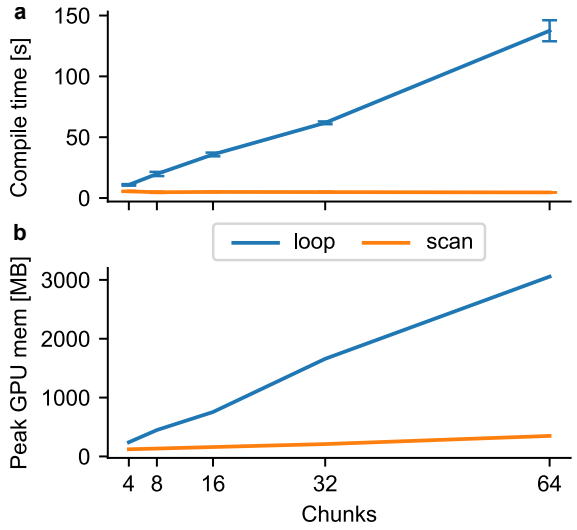

Figure 1. **a** Comparison of compile time between an approach using a loop and `scan` for the chunked loss. **b** Comparison of memory consumption between an approach using a loop and `scan`.

State-Space Models (SSMs) have shown promising results as Transformer alternatives for language modeling (Gu & Dao, 2023; Dao & Gu, 2024; Vaswani et al., 2017; Gu et al., 2022; 2023). This class of models scale linearly or near-linearly in sequence length. The `associative_scan` operator is well-suited for such architectures. For example, the S5 architecture (Smith et al., 2023) can be implemented using `associative_scan` with the following `combine_fn`, see[1]):

[1] https://github.com/i404788/s5-pytorch/blob/c74be7270fe2ec9dc13efcffcfd7f5355d884030/s5/s5_model.py#L10-L22

```
def s5_operator(x, y):
  A_i, Bu_i = x
  A_j, Bu_j = y
  return A_j * A_i, A_j * Bu_i + Bu_j
```

Figure 2 compares an implementation of an S5 layer, with a state dimension of 20, using eager PyTorch and using the `scan` operator. We investigate various sequence lengths T and show the compile and the run times. The eager PyTorch implementation utilizes a tree-like reduction structure, which enable parallelization with O(log(T)), while the `associative_scan` operator lowers to a single fused kernel.

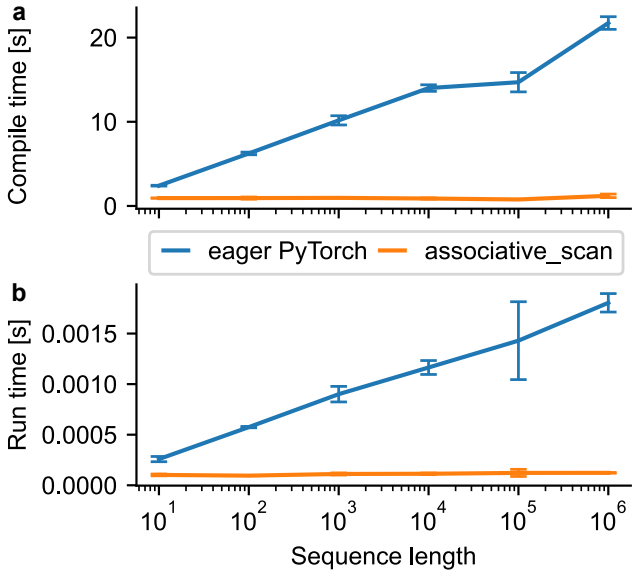

Figure 2. **a** Comparison of compile time for an S5 layer using an eager PyTorch implementation and the `associative_scan`. **b** Comparison of run time for an S5 layer using an eager PyTorch implementation and the `associative_scan`.

Experimental setup. All our experimental verifications are done with PyTorch version 2.8.0a0+gita565834 on a desktop PC with an AMD Ryzen Threadripper 3960X 24-Core processor with 3.9GHz, an NVIDIA GeForce RTX 3090 with 24GB of VRAM, 128GB of main memory.

4. Conclusion

This paper introduces the PyTorch control flow operator library, which enables the PT2 compiler to holistically represent user programs that consists of dynamic control flows. Benefits include avoiding recompilation for dynamic control flow, reducing compilation time from linear to constant, saving memory consumption, and supporting optimizations with native PyTorch operators without requiring low-level kernel development.

References

Agrawal, A., Modi, A., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *Proceedings of Machine Learning and Systems*, 1:178–189, 2019.

Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Snyder, J. B., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P., Mastropietro, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016. URL http://arxiv.org/abs/1605.02688.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. 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*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: an automated end-to-end optimizing compiler for deep learning. In Arpaci-Dusseau, A. C. and Voelker, G. (eds.), *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pp. 578–594. USENIX Association, 2018. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In Moschitti, A., Pang, B., and Daelemans, W. (eds.), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179/.

Dao, T. and Gu, A. Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality. *arXiv*, May 2024. doi: 10.48550/arXiv.2405.21060.

DeVito, Z. and et al. Jit language reference, 2019. URL https://docs.pytorch.org/docs/1.9.0/jit_language_reference_v2.html. Accessed: 2025-05-24.

Dong, J., Feng, B., Guessous, D., Liang, Y., and He, H. Flex attention: A programming model for generating optimized attention kernels. *CoRR*, abs/2412.05496, 2024. doi: 10.48550/ARXIV.2412.05496. URL https://doi.org/10.48550/arXiv.2412.05496.

Gu, A. and Dao, T. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. *arXiv*, December 2023. doi: 10.48550/arXiv.2312.00752.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=uYLFoz1vlAC.

Gu, A., Johnson, I., Timalsina, A., Rudra, A., and Ré, C. How to train your HIPPO: state space models with generalized orthogonal basis projections. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=klK17OQ3KB.

He, H. The state of machine learning frameworks in 2019. *The Gradient*, 2019.

He, H. and Yu, S. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*. mlsys.org, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/hash/8a27bb69950c0b46cdb36d10e5514cc8-Abstract-mlsys2023.html.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

Hsu, P.-L., Dai, Y., Kothapalli, V., Song, Q., Tang, S., Zhu, S., Shimizu, S., Sahni, S., Ning, H., and Chen, Y. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*, 2024. URL https://arxiv.org/abs/2410.10989.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., and Gonzalez, J. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/hash/0b816ae8f06f8dd3543dc3d9ef196cab-Abstract.html.

Jeong, E., Cho, S., Yu, G., Jeong, J. S., Shin, D., and Chun, B. JANUS: fast and flexible deep learning via symbolic graph execution of imperative programs. In Lorch, J. R. and Yu, M. (eds.), *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pp. 453–468. USENIX Association, 2019. URL https://www.usenix.org/conference/nsdi19/presentation/jeong.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Brecht, T. and Williamson, C. (eds.), *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pp. 47–62. ACM, 2019. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Kim, T., Jeong, E., Kim, G., Koo, Y., Kim, S., Yu, G., and Chun, B. Terra: Imperative-symbolic co-execution of imperative deep learning programs. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 1468–1480, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/0b32f1a9efe5edf3dd2f38b0c0052bfe-Abstract.html.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Jurafsky, D., Chai, J., Schluter, N., and Tetreault, J. R. (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 7871–7880. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.ACL-MAIN.703. URL https://doi.org/10.18653/v1/2020.acl-main.703.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pp. 881–897. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., Sculley, D., Rompf, T., and Wiltschko, A. B. Autograph: Imperative-style coding with graph-based performance. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of the Second Conference on Machine Learning and Systems, SysML 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/hash/

e31cef6b735cf838db79202dbee7b093-Abstract.html.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Freund, S. N. and Yahav, E. (eds.), *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pp. 883–898. ACM, 2021. doi: 10.1145/3453483.3454083. URL https://doi.org/10.1145/3453483.3454083.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019a.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019b. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Smith, J. T., Warrington, A., and Linderman, S. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Ai8Hw3AXqks.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Wang, H., Zhai, J., Gao, M., Ma, Z., Tang, S., Zheng, L., Li, Y., Rong, K., Chen, Y., and Jia, Z. PET: optimizing tensor programs with partially equivalent transformations and automated corrections. In Brown, A. D. and Lorch, J. R. (eds.), *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pp. 37–54. USENIX Association, 2021. URL https://www.usenix.org/conference/osdi21/presentation/wang.

Yu, Y., Abadi, M., Barham, P., Brevdo, E., Burrows, M., Davis, A., Dean, J., Ghemawat, S., Harley, T., Hawkins, P., Isard, M., Kudlur, M., Monga, R., Murray, D. G., and Zheng, X. Dynamic control flow in large-scale machine learning. In Oliveira, R., Felber, P., and Hu, Y. C. (eds.), *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pp. 18:1–18:15. ACM, 2018. doi: 10.1145/3190508.3190551. URL https://doi.org/10.1145/3190508.3190551.

Zhang, C., Ma, L., Xue, J., Shi, Y., Miao, Z., Yang, F., Zhai, J., Yang, Z., and Yang, M. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In Geambasu, R. and Nightingale, E. (eds.), *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pp. 681–699. USENIX Association, 2023. URL https://www.usenix.org/conference/osdi23/presentation/zhang-chen.

Zhang, H., Yu, Z., Dai, G., Huang, G., Ding, Y., Xie, Y., and Wang, Y. Understanding GNN computational graph: A coordinated computation, io, and memory perspective. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of the Fifth Conference on Machine Learning and Systems, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/hash/b559156047e50cf316207249d0b5a6c5-Abstract.html.

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*, pp. 1089–1102. IEEE, 2020. doi: 10.1109/ISCA45697.2020.00092. URL https://doi.org/10.1109/ISCA45697.2020.00092.

Zheng, B., Yu, C. H., Wang, J., Ding, Y., Liu, Y., Wang, Y., and Pekhimenko, G. Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus. In *Proceedings of the 56th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023, pp. 1364–1380. ACM, 2023. doi: 10.1145/3613424.3614248. URL https://doi.org/10.1145/3613424.3614248.

Zheng, Z., Yang, X., Zhao, P., Long, G., Zhu, K., Zhu, F., Zhao, W., Liu, X., Yang, J., Zhai, J., Song, S. L., and Lin, W. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In Falsafi, B., Ferdman, M., Lu, S., and Wenisch, T. F. (eds.), *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pp. 359–373. ACM, 2022. doi: 10.1145/3503222.3507723. URL https://doi.org/10.1145/3503222.3507723.

A. Autograd details for `cond`

As outlined in the main manuscript, the autograd for `cond` can be computed utilizing another `cond` operator employing a different `true_fn` and `false_fn`. In particular, for the example below, the individual functions are:

```
def true_fn(x):
  return x**2

def false_fn(x):
  return torch.sin(x)

def joint_true_fn(x, tangents):
  o = x**2
  return 2 * x * tangents

def joint_false_fn(x, tangents):
  o = torch.sin(x)
  return torch.cos(x) * tangents
```

Therefore, the `forward()` and `backward()` conceptually look like

```
def pred(x):
  return x > 0

def forward(x):
  save_tensors_and_symints_for_backward(x)
  return cond(pred(x), true_fn, false_fn, x)

def backward(tangents):
  operands = saved_tensors_and_symints()
  joint_true_fn = create_bw_fn(true_fn, operands)
  joint_false_fn = create_bw_fn(false_fn, operands)

  return cond(pred(x), joint_true_fn, joint_false_fn, (x, tangents))
```

B. Autograd details for `map`

As described in the main manuscript, the map function computes a function `fn` on each slice of its inputs `xs` and then return the stacked outputs. Therefore, the backward of `map` is another `map` operator, but instead of the function `fn`, the joint function `joint_fn` is used. Similar as in A above, it is computed as

```
def fn(x):
  return x**2

def joint_fn(x, tangents):
  o = x**2
  return 2 * x * tangents
```

Given this joint function, the `forward()` and `backward()` conceptually look like

```
def forward(x):
  save_tensors_and_symints_for_backward(x)
  return map(fn, x)

def backward(tangents):
  operands = saved_tensors_and_symints()
  joint_fn = create_bw_fn(fn, operands)
  return map(joint_fn, (x, tangents))
```

C. Autograd details for `scan`

As described in the main manuscript, `scan` applies a cumulative operator, `combine_fn`, sequentially over a particular dimension of the input `xs`, given an initial value `init`. It produces all the intermediate outputs as well as the last carry. However, in order to implement the backward path of `scan`, all the intermediate carries are also required. Therefore, we create a wrapper of the `combine_fn` which would allow to return all individual carries. For example, given the `combine_fn` of the RNN

```
def combine_fn(prev_h, xs):
  h = torch.sigmoid(torch.matmul(xs, W) + torch.matmul(xs, H))
  return h, h

def combine_fn_with_carry_checkpoint(prev_h, xs):
    carry, out = combine_fn(prev_h, xs)
    return carry, (carry, out)
```

We can then later use these carries to compute the backward function, which is illustrated in the sketch below

```
The forward output of scan is computed as:
carry, ys = scan(combine_fn, init, xs),
where init and xs are the primals.

This computation can be unpacked as
c_0, ys_0 = combine_fn(init, xs_0)
c_1, ys_1 = combine_fn(carry_0, xs_1)
c_2, ys_2 = combine_fn(carry_1, xs_2)
...
c_T, ys_T = combine_fn(carry_(T-1), xs_T)

With the wrapper function combine_fn_with_carry_checkpoint we collect c_0, c_1, ..., c_T into a vector of carries
     that we save for the backward. The output of the forward of scan will be c_T, ys, where ys is the vector of all
      intermediate outputs [y_0, y_1, ..., y_T].

Given the carries and the ys, the gradients for xs and for init can be computed as follows:
We receive the upstream gradients (tangents) in torch.autograd.Function, i.e., we get g_c_T and g_ys,
where g_ys is the vector of all intermediate gradients of the outputs [g_ys_0, g_ys_1, ..., g_ys_T]

We can then proceed to compute the gradients for the init (g_init) and the xs (g_xs) by running a
scan operation reverse over time. In particular,

g_c_(T-1), g_xs_T = joint_combine_fn(c_(T-1), xs_T, g_c_T, g_ys_T)
g_c_(T-2), g_xs_(T-1) = joint_combine_fn(c_(T-2), xs_(T-1), g_c_(T-1), g_ys_(T-1))
g_c_(T-3), g_xs_(T-2) = joint_combine_fn(c_(T-3), xs_(T-2), g_c_(T-2), g_ys_(T-2))
...
g_init, g_xs_1 = joint_combine_fn(c_0, xs_1, g_c_0, g_ys_1)
0     , g_xs_0 = joint_combine_fn(init, xs_0, g_init, g_ys_0),

where joint_combine_fn takes the primals at step t (i.e. c_(t-1), xs_t),
as well as the tangents at step t (i.e. the gradient for the carries g_c_t) and
the upstream gradient of the output of step t (i.e. g_ys_T).
It returns the gradient of xs_t -> g_xs_t, as well as the gradient for the carry of step t-1 -> g_c_(t-1).
```

Therefore, the `forward()` and `backward()` of scan conceptually look like

```
def forward(init, xs):
  c_T, (carries, ys) = scan(combine_fn_with_carry_checkpoint, init, xs)
  save_tensors_and_symints_for_backward(init + xs + carries + ys)
  return c_T, ys

def backward(tangents):
  init, xs, carries, ys = saved_tensors_and_symints()
  joint_fn = create_bw_fn(fn, init, xs)
  return scan(joint_fn, (carries, xs, tangents))
```

D. Autograd details for `associative_scan`

The autograd of `associative_scan` can be implemented in multiple ways, which may be more or less efficient. Therefore, below we provide a sketch of how one such version can be realized. The interested reader is referred to either our online code documentation or to this blog post[2], for more details.

```
def combine_fn(a: torch.Tensor, b: torch.Tensor):
    return a * b

The forward output of associative_scan can be computed as
ys = associative_scan(comine_fn, xs).
This can be unpacked as:
ys_0 = xs_0
ys_1 = combine_fn(ys_0, xs_1) = combine_fn(1, 2)
...
ys_T = combine_fn(ys_(T-1), xs_T) = combine_fn(6, 4)

For the backward, we can then create the joint function again:
def joint_combine_fn(a: torch.Tensor, b: torch.Tensor, g_ys_t: torch.Tensor):
    o = a * b
    return g_ys_t * b, g_ys_t * a,

where g_ys are the upstream gradients (tangents) in torch.autograd.Function.
In particular, g_ys is the vector of all intermediate upstream gradients
of the outputs [g_ys_0, g_ys_1, ..., g_ys_T].

The first output of `joint_combine_fn is the gradient g_y_t,
which is the gradient for the forward output at step t-1, i.e., a.
The second output of joint_combine_fn is the gradient g_x_t,
which is the gradient for the previous forward input at step t, i.e., b.

Given the outputs ys, the gradients for xs can be computed as follows:
We fist utilize the joint_combine_fn to compute the instantaneous gradients g_x_t and g_y_t
at every step as:
g_y_t, g_x_t = joint_combine_fn(ys_(t-1), xs_t, 1.),
where instead of using the elements of g_ys_t, we use 1s. This is required to get the instantaneous
gradients at every step t and we incorporate the upstream gradients g_ys at a later time.

For example, this results in:
g_y_0, g_x_0 = [1, 1] (Initial gradients are 1 by definition)
g_y_1, g_x_1 = joint_combine_fn(ys_0, xs_1, 1)
g_y_2, g_x_2 = joint_combine_fn(ys_1, xs_2, 1)
...
g_y_T, g_x_T = joint_combine_fn(ys_(T-1), xs_T, 1).

With these instantaneous gradients, we can compute the gradients of the inputs xs (g_xs) naively as:
g_xs_t = (\sum_{i=T}^t g_ys_i . (\prod_{k=i}^{k>t} g_y_k)) . g_x_t                                    (1)

In particular,
g_xs_T = g_ys_T . 1 . g_x_T
g_xs_(T-1) = g_ys_T . g_y_T . g_x_(T-1) + g_ys_(T-1) . g_x_(T-1)
g_xs_(T-2) = g_ys_T . g_y_T . g_y_(T-1) . g_x_(T-2) + g_ys_(T-1) . g_y_(T-1) . g_x_(T-2) + g_ys_(T-2) . g_x_(T-2)
...
```

E. Autograd details for `while_loop`

The autograd of `while_loop` is work in progress and not yet part of the control flow operator library. This is because there are challenges that currently prevent an efficient implementation. In particular, to compute the gradients of `while_loop`, all intermediate outputs from the `forward()` need to be stored for the `backward()`. This is problematic, because the number of loop iterations is undetermined during compile time. One naive solution would be to specialize the number of loop iterations during compile time, but this would to some degree defeat the purpose of using the `while_loop` operator in the first place. The alternative approach that we are actively pursuing at the moment, is to figure out a way, how to store a "dynamic" / undetermined number of intermediates in the IR.

[2] https://justintchiu.com/blog/pscan_diff/

F. Additional use cases

F.1. Shape-dependent Branching.

In LLMs, there are cases where we want to create different attention masks based on dynamic input shapes. `cond` would also help in this case:

```
combined_attention_mask = cond(
    seq_len > 1,
    lambda: _make_causal_mask(...),
    lambda: _expand_mask(...),
)
```

F.2. Early Stopping.

During inference, inputs are usually grouped into batches before feeding to the model. GPT-like language models continue to generate next token until a special $endoftext$ token is generated. There's also a static max number of tokens that the models can generate. Early stopping refers to the case when the next tokens for all inputs in a batch are $endoftext$, we can stop early to save computation cycles before hitting the max output size. The model needs to look at the value of all newly generated tokens to decide if the model continues. This is a data-dependent loop that can be captured by `while_loop`:

```
def cond_fn(idx, all_tokens):
  return _all_sequnce_done(all_tokens[idx])

def body_fn(idx, all_tokens):
  next_tokens = model(all_tokens)
  return idx + 1, next_tokens

idx = torch.tensor(0, dtype=torch.int64)
# Omitting pre-allocating logic
all_tokens = ...
n_iter, all_tokens = while_loop(cond_fn, body_fn, (idx, all_tokens)
```

F.3. Chunked loss computation

```
class ChunkedCE(torch.nn.Module):
    def __init__(self, chunk_size):
        super().__init__()
        self.chunk_size = chunk_size
        self.ce = torch.nn.CrossEntropyLoss()

    def forward(self, _input, weight, target, bias):
        CHUNK_SIZE = self.chunk_size

        def compute_loss(input_chunk, weight, bias, target):
            logits = torch.addmm(bias, input_chunk, weight.t()).float()
            loss = self.ce(logits, target)
            return loss

        grad_weight = torch.zeros_like(weight)
        grad_bias = torch.zeros_like(bias)
        loss_acc = torch.zeros((), device=_input.device)

        chunks = _input.shape[0] // CHUNK_SIZE
        _input_chunks = _input.view(chunks, CHUNK_SIZE, *_input.shape[1:])
        target_chunks = target.view(chunks, CHUNK_SIZE, *target.shape[1:])

        def combine_fn(carry, xs):
            grad_weight, grad_bias, loss_acc = carry
            input_chunk, target_chunk = xs
            (
                chunk_grad_input,
                chunk_grad_weight,
                chunk_grad_bias,
            ), chunk_loss = torch.func.grad_and_value(
                compute_loss, argnums=(0, 1, 2)
            )(
                input_chunk, weight, bias, target_chunk
            )
            return (
                (
                    grad_weight + chunk_grad_weight,
                    grad_bias + chunk_grad_bias,
                    loss_acc + chunk_loss,
                ),
                chunk_grad_input,
            )

        (grad_weight, grad_bias, loss_acc), grad_inputs = scan(
            combine_fn,
            (grad_weight, grad_bias, loss_acc),
            (_input_chunks, target_chunks),
        )
        return (
            grad_weight / chunks,
            grad_bias / chunks,
            loss_acc / chunks,
            grad_inputs.view(-1, *_input.shape[1:]) / chunks,
        )

# chunk size 1024
mod = ChunkedCE(1024)
B, T, D, V = 32, 1024, 768, 128256
torch.set_default_device('cuda')
model = torch.nn.Linear(D, V).to(torch.bfloat16)
x = torch.randn(B, T, D, requires_grad=True, dtype=torch.bfloat16)
label = torch.randint(0, V, (B, T)).to(torch.int64)

inp, weight, target, bias = (
        x.view(-1, D), model.weight, label.view(-1), model.bias)
    )
mod(inp, weight, target, bias)
```

Figure 3 and Figure 4 shows the memory usage pattern for a naive PyTorch loop implementation and and a scan implementation, where Scan accumulates the intermediate results in a buffer instead of materializing intermediate results.

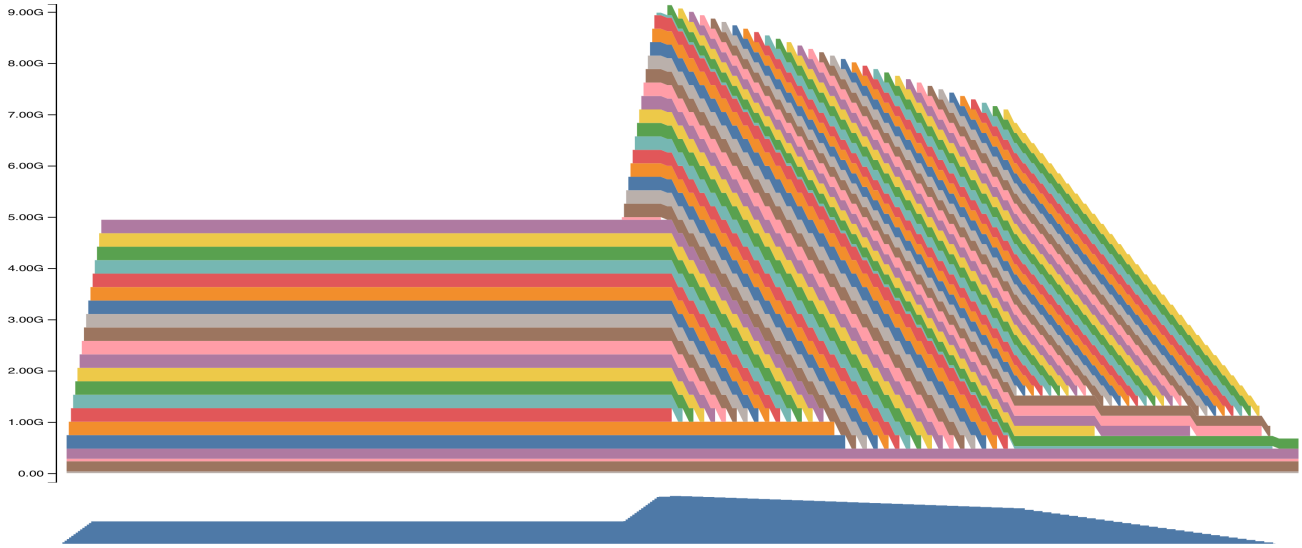

Figure 3. Memory profile for the Chunked Loss using a naive implementation with a loop.

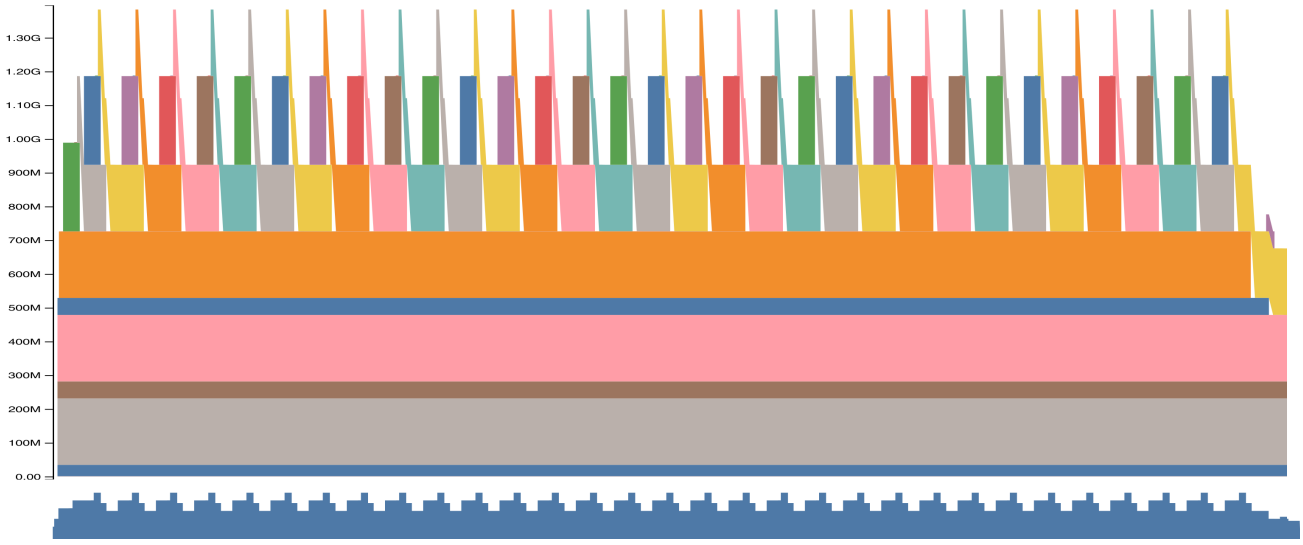

Figure 4. Memory profile for the Chunked Loss using an enhanced implementation with `scan`.

F.4. RNN

Another usecase for the `scan` operator are stateful models, such as long short-term memory units (LSTM (Hochreiter & Schmidhuber, 1997)) or gated recurrent units (GRUs (Cho et al., 2014)). Because their inherent dynamics requires sequential processing of the input, which traditionally has been implemented via loops. As discussed, these loops cause significant issues during compilation if proper control flow operators are absent. With the `scan` operator, RNNs can be efficiently realized without explicit loops. To illustrate this, the listing below showcases the implementation of a classic RNN with loops and with the `scan` operator.

```
W = torch.rand(2, 4)
H = torch.rand(4, 4)

def classicRNN(init_h, x):
  h = init_h
  outs = []
  for xs in x:
    h, o = combine_fn(h, xs)
    outs.append(o)
  return h, torch.stack(outs)

def combine_fn(prev_h, xs):
  h = torch.sigmoid(torch.matmul(xs, W) + torch.matmul(xs, H))
  return h, h

x = torch.rand(4, 2)
init_h = torch.zeros(4)
res = classicRNN(init_h, x)
res_scan = scan(combine_fn, init_h, x)
```

Moreover, using the `scan` operator unlocks significant savings in compile and memory consumption, which is illustrated in Figure 5.

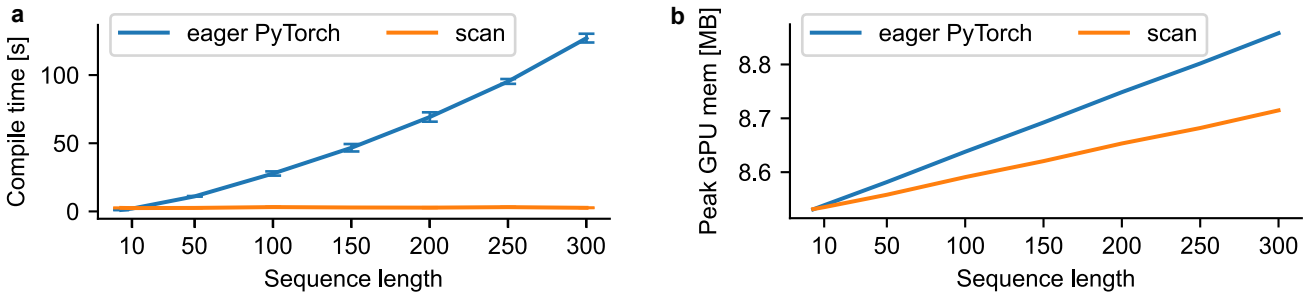

Figure 5. **a** Comparison of compile time for an RNN using an eager PyTorch implementation and the `scan` operator. **b** Comparison of memory consumption for an RNN using an eager PyTorch implementation and the `scan` operator.