Decoupled Triton: Exploring Coupled and Decoupled Machine-Learning Kernel Languages

Quinn Pham (qpham@ualberta.ca)[1], Danila Seliayeu[1], Prasanth Chatarasi[2] and José Nelson Amaral[1]
University of Alberta[1] and IBM[2]

Decoupled Triton (DT)

Machine-learning (ML) applications frequently utilize ML tensor kernels to execute linear-algebra computations. A tensor kernel includes two components: the **computation**, which defines the operation to be performed, and the **schedule**, which defines how the operation is executed. An efficient schedule is necessary for a high-performance tensor kernel. Halide [2] and Triton [1] are two domain-specific languages (DSLs) designed to define tensor kernels. Halide's approach decouples computation and schedule, relying on expert programmer knowledge to create efficient kernels. In contrast, Triton tightly couples computation and schedule, leveraging automatic compiler optimizations.

We propose **Decoupled Triton (DT)**, a DSL for writing GPU tensor kernels. DT is a decoupled language, like Halide. DT acts as an abstraction layer on top of Triton that adopts the benefits of both a decoupled kernel language like Halide, and a coupled kernel language like Triton.

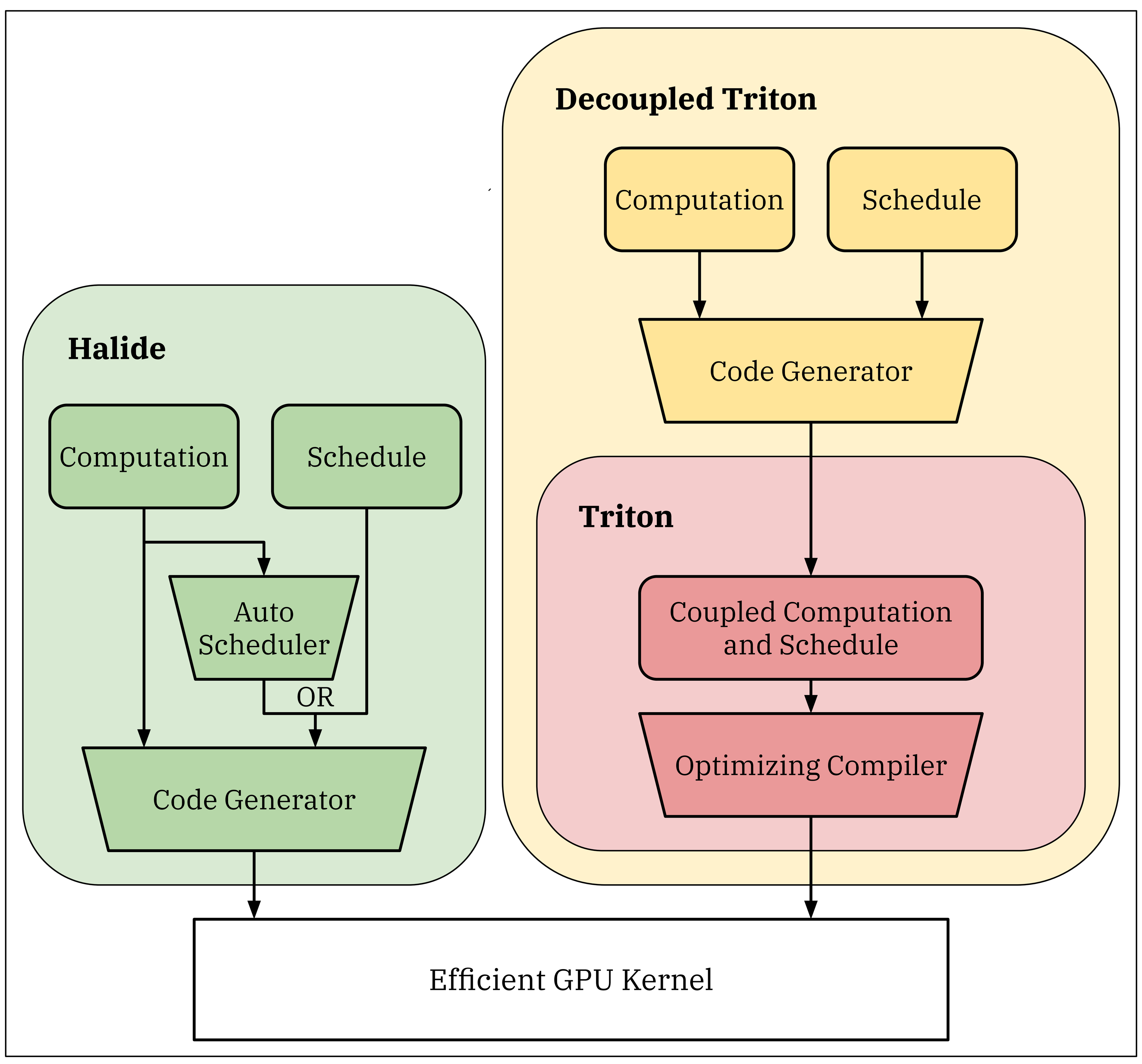


Figure 1. Overview of Halide, Triton, and Decoupled Triton.

Halide Approach		Triton Approach		DT Approach
• **Modularizes** computation and schedule • Leverages **expert programmer knowledge** • **Abstracts low-level details** e.g., pointer arithmetic	+	• **Block-level** programming • Leverages **automatic compiler optimizations**	=	• **Modularizes** computation and schedule • Leverages **expert programmer knowledge** • **Abstracts low-level details** e.g., pointer arithmetic • **Block-level** programming • Leverages **automatic compiler optimizations**

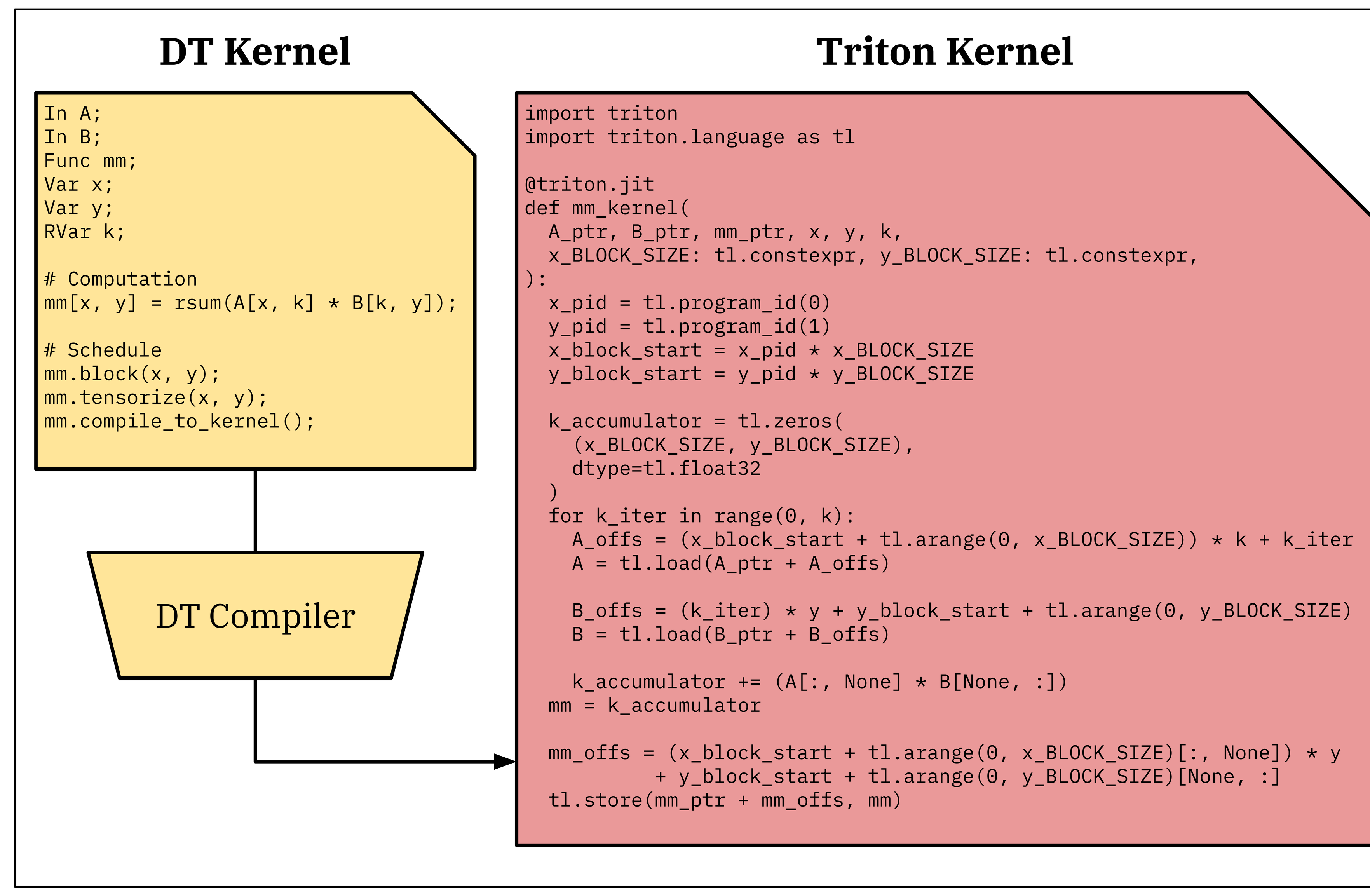
Figure 2. Decoupled Triton combines the advantages of Halide and Triton.

Implementation

- An ANTLR4 [4] grammar defines the Decoupled Triton DSL.
- The ANTLR4 runtime generates the lexer, parser, and parse tree code of the compiler.
- ~2K lines of C++

`block()`	Each Triton program computes a block of the function. Without this directive, a single Triton program is launched to compute the entire function.
`tensorize()`	Triton programs compute tensors of the function using SIMD operations on tensors from the inputs.
`stride()`	Requires `block()`, each Triton program computes multiple blocks in a strided pattern sequentially.
`group()*`	Requires `block()`, the mapping of Triton programs on the launch grid follows a grouped order.

Table 1. Scheduling directives in DT. *`group` is not fully implemented in the DT compiler yet.




```
In A;
In B;
Func mm;
Var x;
Var y;
RVar k;

# Computation
mm[x, y] = rsum(A[x, k] * B[k, y]);

# Schedule
mm.block(x, y);
mm.tensorize(x, y);
mm.compile_to_kernel();
```

```
import triton
import triton.language as tl

@triton.jit
def mm_kernel(
  A_ptr, B_ptr, mm_ptr, x, y, k,
  x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr,
):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE

  k_accumulator = tl.zeros(
    (x_BLOCK_SIZE, y_BLOCK_SIZE),
    dtype=tl.float32
  )
  for k_iter in range(0, k):
    A_offs = (x_block_start + tl.arange(0, x_BLOCK_SIZE)) * k + k_iter
    A = tl.load(A_ptr + A_offs)

    B_offs = (k_iter) * y + y_block_start + tl.arange(0, y_BLOCK_SIZE)
    B = tl.load(B_ptr + B_offs)

    k_accumulator += (A[:, None] * B[None, :])
  mm = k_accumulator

  mm_offs = (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y
          + y_block_start + tl.arange(0, y_BLOCK_SIZE)[None, :]
  tl.store(mm_ptr + mm_offs, mm)
```

Figure 3. Example DT kernel and the Triton kernel generated by the DT compiler.

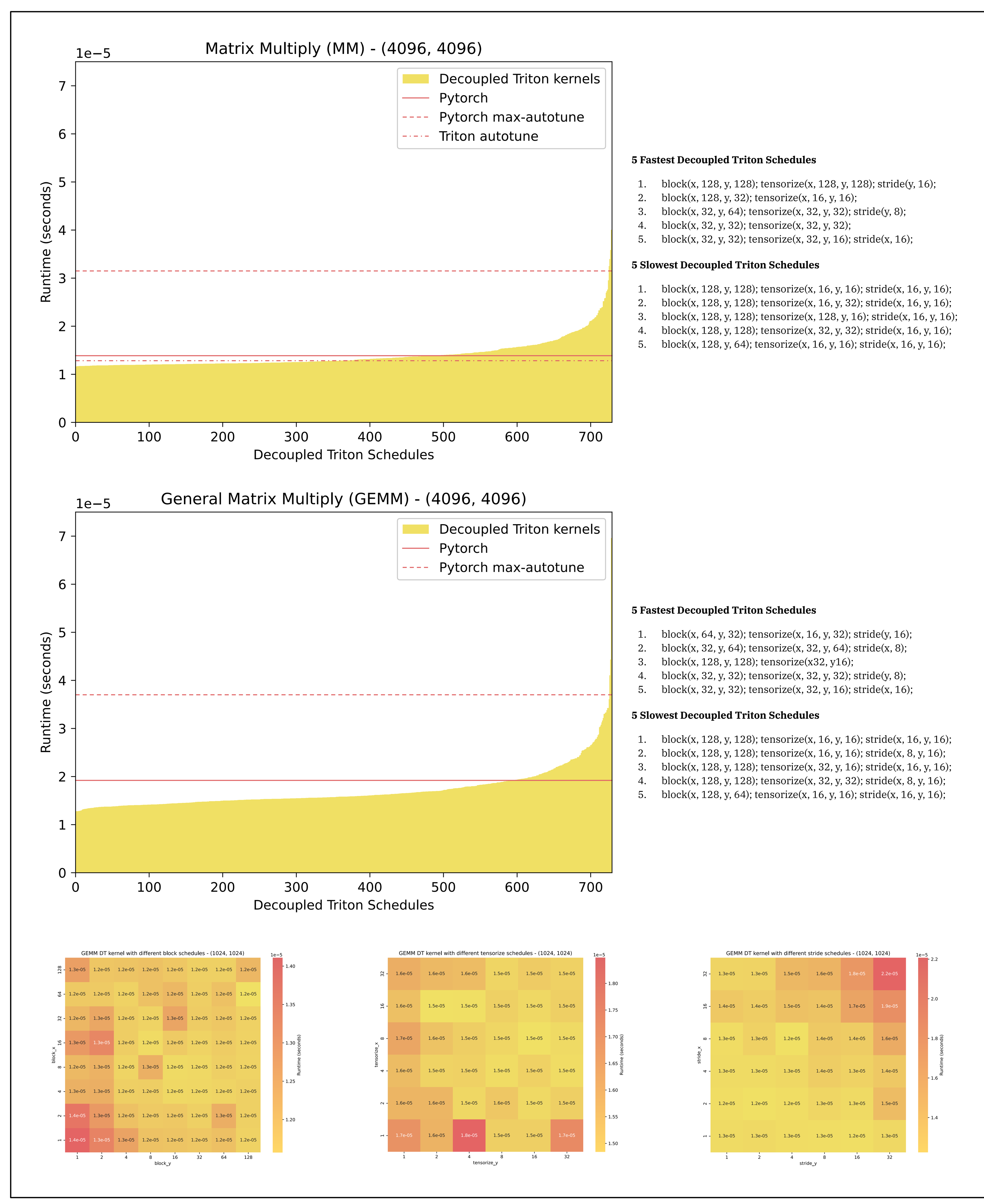


Figure 4. Runtime of DT kernels with different schedules against Pytorch and Triton.

Future Work

- A DT auto-scheduler (similar to Halide auto-schedulers [5])
- Schedule directives for multi-GPU programming:
  - Generate wrapper code for multiple Triton kernel invocations

References

1. Philippe Tillet, H. T. Kung, and David Cox. 2019. 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 (MAPL 2019). Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973
2. Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176
3. Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. 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), Vol. 2. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366
4. Terence Parr. 2013. The Definitive ANTLR 4 Reference (2nd. ed.). Pragmatic Bookshelf.
5. Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient automatic scheduling of imaging and vision pipelines for the GPU. Proc. ACM Program. Lang. 5, OOPSLA, Article 109 (October 2021), 28 pages. https://doi.org/10.1145/3485486

Acknowledgement

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).
Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.