Decoupled Triton: A Block-Level Decoupled Language for Writing and Exploring Efficient Machine-Learning Kernels

by

Quinn Leo Pham

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Machine-learning (ML) applications frequently utilize high-performance ML kernels to execute tensor operations like matrix product and softmax. An ML kernel can be decomposed into two components: the implicit algorithm, which defines the tensor operation that computes the output tensor, and the schedule, which defines how the operation is implemented. The schedule of an ML kernel determines performance factors such as memory access patterns and vectorization. Therefore, an efficient schedule is necessary for an efficient ML kernel. Unfortunately, finding an efficient schedule for a given ML kernel is difficult and may require intimate knowledge of the hardware that executes the ML kernel. A decoupled language represents programs as the combination of a modular algorithm and a modular schedule. Triton is a high-abstraction language and compiler for parallel programming that is commonly used to write high-performance ML kernels. Triton follows a block-level programming paradigm that requires programmers to define ML kernels at the thread-block level. This programming paradigm allows developers to ignore low-level details such as shared memory and thread coalescence, which are automatically handled by the Triton compiler. Despite these abstractions, writing efficient Triton kernels by hand remains a tedious and error-prone experience for two reasons: Triton kernels still include some low-level details—such as pointer arithmetic—and iterating on a kernel schedule in Triton is difficult due to a tight coupling between the algorithm and the schedule. Furthermore, higher-level ML frameworks that generate Triton kernels, like

PyTorch, may not generate efficient schedules and do not allow users to provide their own scheduling.

We propose Decoupled Triton (DT), a block-level decoupled Domain Specific Language (DSL) and compiler for writing parallel tensor kernels. The DT compiler takes a modular algorithm and schedule defined in the DT DSL and generates a Triton kernel. Thus, DT acts as an abstraction layer on top of Triton, decoupling the algorithm from the schedule. The block-level programming paradigm, adopted from Triton, allows for simple and intuitive scheduling, while the decoupling of the algorithm from the schedule makes ML kernel schedule exploration easy and fast. We demonstrate that DT enables developers to rapidly explore schedule spaces and find efficient ML kernels with performance matching, and even exceeding, that of both ML kernels hand-written by expert Triton developers and ML kernels generated by PyTorch, a mature ML programming framework.

Preface

This thesis is an original work by Quinn Leo Pham. No part of this thesis has been previously published.

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.

To Kana,

月が綺麗ですね。

To Mom, Dad, Jace, Gavin, Jaxon, Memere, and Grandad,

I love you.

Acknowledgements

I thank my supervisor, Dr. José Nelson Amaral, for the guidance, teachings, and opportunities he has provided me throughout the more than five years that we have worked together. I truly appreciate the understanding, flexibility, and compassion he showed me during the difficult times in my degree. His influence has helped me grow not only as a researcher but also as a person. Above all, I will not forget his insightful lessons in communication nor the many cooking and board game parties at his home.

I thank Dr. Michael Bowling and Dr. Zhou Yang for serving on my defense committee and evaluating my thesis.

I thank Dr. Prasanth Chatarasi for his feedback and guidance in my research. He posed the initial research questions that formed the motivation of my work and provided valuable direction and feedback throughout the project.

I thank Kevin and Arsh for providing feedback on my project as users.

I thank Dan for the many discussions about my research.

I thank everyone whom I've had the pleasure of working with in the lab, including Caio, Tyler, Dan, Dhanraj, Nathan, Reza, Ayrton, Rajan, Patrick, Mariana, Sarah, João, Justin, Madison, Rouzbeh, and Deric. Their kindness and humour made the lab a welcoming and fun space. I valued all of our conversations, both research-related and otherwise.

I thank everyone who played games with me; they served as refreshing breaks from the work.

I thank Caio for his friendship and mentorship throughout my degree. He was always willing to break out the whiteboard and get into the details whenever I got stuck on a problem. Our runs, bouldering sessions, and board games kept me motivated.

I thank Tyler and Enlik for their friendship and support.

I thank my dad for the desk and chair I use every day. My apartment would be half as full without his resourcefulness.

I thank my mom for making sure I keep in touch with my family. I always laugh a lot during our calls.

I thank Jace for all of the fun we've had playing games together. Despite the distance, he is an exceptional brother and friend.

Finally, I thank Kana for supporting me through all the long nights and frustrations in my research. She always helped me stay motivated and confident. I could not have made it this far without her by my side.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI Artificial Intelligence.

API Application Programming Interface.

DL Deep-Learning.

DSL Domain Specific Language.

DT Decoupled Triton.

GPU Graphics Processing Unit.

IR Intermediate Representation.

JIT Just-In-Time.

LLM Large Language Model.

ML Machine-Learning.

SIMD Single-Instruction-Multiple-Data.

TMA Tensor Memory Accelerator.

Chapter 1

Introduction

Modern Deep-Learning (DL) applications such as natural language processing [27] and computer vision [32] often require training large models on massive datasets for weeks to compete with state-of-the-art results. This extensive computation leads to a significant financial and environmental impact due to substantial energy consumption [37]. Furthermore, modern performance-sensitive DL models such as game-playing Artificial Intelligence (AI) systems [26] and Generative AI-powered chatbots [27] have significant pressure to execute their computational graphs quickly during inference. Consequently, the efficient execution of DL models has become a crucial research focus.

An important factor in efficiently executing a DL model is the utilization of high-performance Machine-Learning (ML) kernels that efficiently perform tensor operations such as matrix product and softmax. Such ML kernels are generally implemented using parallel programming languages and executed on parallel hardware, such as Graphics Processing Units (GPUs) to leverage the parallel nature of tensor operations. High-performance vendor libraries such as cuDNN [8] and cuBLAS [9] supply expert-written low-level kernels — library kernels — for common tensor operations. However, when working with an uncommon or novel tensor operation that is not implemented by existing high-performance libraries, developers must implement

a high-performance ML kernel that performs the tensor operations themselves.

Halide [30, 31] introduced the concept of decoupling the algorithm from the schedule in programming languages for image-processing pipelines. An ML kernel can be decomposed into two components: the implicit algorithm, which defines the tensor operation that computes the output tensor, and the schedule, which defines how the operation is implemented [7]. The schedule of an ML kernel determines important performance factors such as memory access patterns and vectorization. Therefore, finding an efficient schedule is necessary for writing an efficient ML kernel. Note that finding an efficient schedule for a given ML kernel is difficult and may require intimate knowledge of both the hardware that executes the ML kernel and the dimension sizes of the tensors passed into the ML kernel at runtime. A decoupled language is a programming language that represents programs as the combination of a modular algorithm and schedule [30]. This decoupling of the algorithm from the schedule provides major advantages such as allowing programmers to define the schedule of their programs using scheduling primitives and to quickly explore different schedules to find an efficient program.

When writing high-performance ML kernels, developers have two main options: a low-abstraction parallel programming language such as CUDA [24], OpenCL [36], and HIP [14], or a high-abstraction parallel programming language such as Triton [38]. Many developers opt for the high-abstraction alternatives like Triton [38], because low-abstraction parallel languages have steep learning curves and may not be portable across different hardware. Triton's main advantages compared to low-abstraction languages are the result of Triton's block-level programming paradigm that requires programmers to define ML kernels at the thread-block level. This programming paradigm allows developers to ignore low-level details — such as shared-memory and thread coalescence — that are handled by the Triton compiler automatically. Despite

these abstractions, writing efficient Triton kernels by hand remains undesirable because Triton kernels still include some low-level details, such as pointer arithmetic, which are bug-prone [33] and tedious to write. Furthermore, the tight coupling of the algorithm and the schedule in the Triton language makes it difficult to explore different schedules and iterate upon a schedule to find the most efficient implementation. Also, ML frameworks that generate Triton kernels, like PyTorch [4], may not generate efficient schedules and do not allow programmers to provide their own scheduling for an operation.

This thesis presents Decoupled Triton (DT), a block-level decoupled Domain Specific Language (DSL) and compiler for writing and exploring efficient ML kernels. The Decoupled Triton compiler takes a modular algorithm and schedule defined in the Decoupled Triton DSL and generates a Triton kernel. Thus, Decoupled Triton acts as an abstraction layer on top of Triton. This layer decouples the algorithm from the schedule. Adopting the block-level programming paradigm from Triton allows for intuitive and simple scheduling primitives, while decoupling the algorithm from the schedule allows for user-defined scheduling and rapid schedule exploration. Furthermore, the higher-level representation of the algorithm in Decoupled Triton simplifies the definition of ML kernels. This thesis demonstrates that: (1) the Decoupled Triton DSL is highly expressive, nearing the expressivity of Triton kernels themselves, limited mainly by an inability to represent highly specialized schedules like FlashAttention [10]; (2) Decoupled Triton simplifies programming Triton kernels by abstracting intricate details; (3) Decoupled Triton enables developers to rapidly explore schedule spaces and find efficient ML kernels with performance meeting, and even exceeding, that of both ML kernels written by expert Triton developers and ML kernels generated by PyTorch, a mature ML programming framework.

The statement of this thesis is as follows:

Decoupling the algorithm from the schedule in a high-abstraction language focused on Machine-Learning kernels increases the programmability of the language and facilitates the exploration of schedules for a given computation.

In summary, this thesis makes the following contributions:

1. Decoupled Triton, a block-level decoupled DSL and compiler for defining and exploring efficient ML kernels (Chapter 3). This contribution includes an algorithm representation based on tensor expressions and a set of block-level scheduling primitives.

2. An implementation of the Decoupled Triton DSL and compiler (Section 3.5).

3. An experimental workflow for rapid schedule exploration to find efficient ML kernels in Decoupled Triton (Section 4.2).

4. A detailed experimental evaluation of Decoupled Triton comparing against both expert-written Triton kernels and kernels generated by PyTorch (Chapter 4).

The remainder of this thesis is organized as follows. Chapter 2 discusses the relevant background for Decoupled Triton. Chapter 3 presents the Decoupled Triton DSL and compiler. Chapter 4 describes an experimental evaluation that compares Decoupled Triton against expert-written kernels and kernels generated by PyTorch. Chapter 5 discusses the related work of Decoupled Triton. Chapter 6 discusses future research directions related to Decoupled Triton. Chapter 7 concludes the thesis.

Chapter 2

Background

This chapter presents the background for decoupling the algorithm from the schedule in a programming language, illustrating it with an example, and listing its advantages. Next, the chapter discusses parallel ML kernel languages, including Triton and its block-level programming paradigm.

2.1 Decoupling the algorithm from the schedule

Halide [30, 31] is a DSL for describing image-processing pipelines that introduced the concept of algorithm and schedule in a programming language. An image processing pipeline represented in the Halide DSL is decomposed into two components: the algorithm and the schedule. The algorithm is a functional representation of the pipeline, while the schedule is a description of how the pipeline is implemented as a program. In particular, the schedule determines the details of the program implementation, such as the order in which the output is computed, vectorization, and parallelism. Listing 2.1 shows an example image-processing pipeline described in Halide. The algorithm of the pipeline (lines 6–7) is decoupled from the schedule (lines 10–12). In this text, Halide and other programming languages that decouple the algorithm from the schedule are referred to as decoupled languages; a programming language that is not decoupled, like Triton, is referred to as a coupled language.

Listing 2.1: A 3x3 blur operation defined in Halide's C++ Application Programming Interface (API) [30].

```
1  Func blur_3x3(Func input) {
2    Func blur_x, blur_y;
3    Var x, y, xi, yi;
4
5    // The algorithm - no storage or order
6    blur_x(x, y) = (input(x - 1, y) + input(x, y) + input(x + 1, y)) / 3;
7    blur_y(x, y) = (blur_x(x, y - 1) + blur_x(x, y) + blur_x(x, y + 1)) / 3;
8
9    // The schedule - defines order, locality; implies storage
10   blur_y.tile(x, y, xi, yi, 256, 32)
11         .vectorize(xi, 8).parallel(y);
12   blur_x.compute_at(blur_y, x).vectorize(x, 8);
13
14   return blur_y;
15 }
```

In decoupled languages, the schedule is defined by a series of scheduling primitives (lines 10–12 of Listing 2.1). Scheduling primitives are operations that transform the schedule. An empty schedule, without any scheduling primitives, results in a default schedule that is sequential, and typically naive, for any given operation. Similarly, any scheduling decisions not explicitly specified by the scheduling primitives in the schedule default to some predefined naive scheduling or sometimes the result of a simple heuristic.

The schedule defines the implementation details of a program. Thus, the schedule, together with the hardware that executes the program, determines performance factors of the program, such as memory access patterns, hardware utilization, cache utilization, and feature utilization. Therefore, an efficient schedule leads to an efficient program. Coupled languages typically use an optimizing compiler with analyses to set the low-level scheduling details, such as vectorization, that are not represented in the high-level coupled representation [34]. Decoupled languages, such as Halide, do not use an optimizing compiler. Instead, a code generator converts the provided

algorithm and schedule into an executable program. Therefore, decoupled languages rely on the programmer defining an efficient schedule to produce an efficient program.

To facilitate finding an efficient schedule, researchers have developed autoschedulers that automatically find a schedule to implement a given algorithm. The schedules found by an autoscheduler can be used directly or modified by the developer. Multiple autoschedulers have been written for Halide [1, 3, 21, 23].

Compared to a coupled language, a decoupled language offers the following main advantages:

1. Expert developers can input their knowledge directly into the schedule of the program. A compiler will not necessarily generate an efficient schedule for any given program. An expert developer, armed with knowledge about the hardware that will execute their program, is best positioned to define an efficient schedule.

2. The algorithm and schedule are modular. Extracting the schedule of a program written in a coupled language like Triton is often difficult, which in turn makes it harder to modify the algorithm and the schedule independently. Decoupled languages force a separation of the algorithm from the schedule, making both components modular. This modularity allows developers to modify the algorithm or schedule of a program without impacting the other component, thereby simplifying development and improving maintainability.

3. Developers can easily explore different schedules for an algorithm. Due to the modular representation of the schedule and the exposed scheduling primitives in decoupled languages, developers can quickly and easily explore many different schedules for an algorithm to find efficient schedules. This ease of exploration speeds up development and can teach developers how to write efficient schedules in general or even for a particular hardware.

The concept of decoupling the algorithm from the schedule is not exclusive to image-processing pipelines. For ML kernels, the algorithm is the tensor expression that computes the output tensor, and the schedule is the description of how the kernel is implemented. Related work that decouples the algorithm from the schedule for ML kernels are discussed in Subsection 5.3. This thesis proposes Decoupled Triton, a decoupled DSL for defining ML kernels that benefits from the advantages of decoupled representations.

2.2 Parallel Machine-Learning kernel languages

ML applications frequently utilize ML kernels to execute tensor computations such as matrix product and softmax. Due to the parallel nature of these tensor computations, parallel architectures like GPUs are natural targets for executing ML kernels efficiently. Thus, developers typically write ML kernels in parallel programming languages. Prior to the introduction of Triton [38], the available parallel programming languages were low-abstraction programming languages such as CUDA [24], OpenCL [36], and HIP [14]. Programming an efficient ML kernel in these low-abstraction parallel programming languages is error-prone and requires a strong understanding of parallel hardware — GPU architectures — and parallel programming patterns. Consequently, writing high-performance ML kernels was inaccessible to many developers and strictly the domain of performance engineers.

2.2.1 Triton and block-level programming

Triton is a high-abstraction parallel programming language and compiler that is primarily used to define ML kernels. The main innovation that Triton introduced is the block-level programming paradigm [38]. Instead of defining parallel programs at the thread-level, like in low-abstraction parallel programming languages such as

CUDA [24], OpenCL [36], and HIP [14], Triton programs are defined at the thread-block level. To execute a Triton kernel, it is launched as a grid of parallel program instances. The number of programs in the grid depends on the dimension sizes that are blocked. Each program instance is responsible for computing a block of the output. This higher-level programming paradigm abstracts many low-level details such as thread coalescence and shared memory management, which are handled by the Triton compiler. Triton is a Pythonic language and has direct integration with PyTorch types. Experiments have shown that Triton kernels can match the performance of ML kernels from high-performance vendor libraries such as cuBLAS [9]. The Triton compiler supports NVIDIA GPUs (Compute Capability 8.0+) and AMD GPUs (ROCm 6.2+).[1]

Listing 2.2 shows an example ML kernel written in Triton, taken from the Triton tutorial.[2]. In the example kernel, `tl.program_id(axis=0)` (line 20) retrieves the program instance index in the launch grid. This index is used to compute the offset (lines 26–27) into the input and output tensors that the kernel should be loading from (lines 33–34) and storing to (line 37), respectively. The value of `BLOCK_SIZE` (line 15) is a static constant that specifies how many output elements are computed by each program instance.

Despite the advantages listed above, writing high-performance ML kernels in Triton has the following disadvantages:

1. Triton kernels still include some low-level details — such as pointer arithmetic (lines 26–27 and 33–34 of Listing 2.2) — which are bug-prone [33] and tedious to write.

2. Due to the tight coupling of the algorithm and schedule in the Triton language,

[1] https://github.com/triton-lang/triton?tab=readme-ov-file#compatibility
[2] https://triton-lang.org/main/getting-started/tutorials/01-vector-add.html

Listing 2.2: A vector addition kernel written in Triton.

```
import triton
import triton.language as tl

@triton.jit
def add_kernel(
    # *Pointer* to first input vector.
    x_ptr,
    # *Pointer* to second input vector.
    y_ptr,
    # *Pointer* to output vector.
    output_ptr,
    # Size of the vector.
    n_elements,
    # Number of elements each program should process.
    BLOCK_SIZE: tl.constexpr,
    # NOTE: `constexpr` so it can be used as a shape value.
    ):
  # There are multiple 'programs' processing different data.
  # We identify which program we are here:
  pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
  # This program will process inputs that are offset from the initial data.
  # For instance, if you had a vector of length 256 and block_size of 64,
  # the programs would each access the elements
  # [0:64, 64:128, 128:192, 192:256].
  # Note that offsets is a list of pointers:
  block_start = pid * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  # Create a mask to guard memory operations against out-of-bounds
  # accesses.
  mask = offsets < n_elements
  # Load x and y from DRAM, masking out any extra elements in case the
  # input is not a multiple of the block size.
  x = tl.load(x_ptr + offsets, mask=mask)
  y = tl.load(y_ptr + offsets, mask=mask)
  output = x + y
  # Write x + y back to DRAM.
  tl.store(output_ptr + offsets, output, mask=mask)
```

it is difficult to explore different schedules for an ML kernel and iterate upon a schedule to find the most efficient implementation.

Triton has also been used as an intermediate representation (IR) for ML frameworks. In fact, PyTorch [4] targets Triton kernels as its main code-generation target for its TorchInductor backend. However, TorchInductor may not generate an efficient schedule for a given ML kernel. Also, PyTorch does not allow programmers to provide their own scheduling for compiling an ML kernel.

We propose Decoupled Triton, a block-level decoupled DSL and compiler for writing high-performance GPU tensor kernels. Decoupled Triton acts as an abstraction layer on top of Triton that decouples the algorithm from the schedule. Decoupling the algorithm from the schedule allows users to rapidly iterate upon and explore schedules to find efficient kernels quickly. Adopting the block-level programming paradigm from Triton allows for simple and intuitive scheduling in the Decoupled Triton DSL.

In Triton, only static parameters — such as block sizes — can be auto-tuned.[3] Consequently, it is not possible to express the same block-level schedule control that Decoupled Triton exposes using only built-in Triton auto-tuning. That is, the schedule space defined using Triton's built-in auto-tuning is a small subset of the schedule space defined by the scheduling primitives of the Decoupled Triton DSL.

[3]https://triton-lang.org/main/python-api/generated/triton.autotune.html

Chapter 3

Decoupled Triton

The Decoupled Triton DSL uses a representation based on tensor expressions to describe the algorithm of an ML kernel and exposes scheduling primitives that transform the schedule of an ML kernel, directly influencing the generated Triton kernel. The Decoupled Triton compiler is an ahead-of-time compiler that converts the Decoupled Triton DSL into code that the Triton compiler can compile.

3.1 Overview

Decoupled Triton is a high-level DSL and compiler workflow for writing ML kernels that function as an abstraction layer on top of Triton to decouple the algorithm from the schedule. This role allows Decoupled Triton to benefit from both the advantages of decoupled languages (Section 2.1) and of Triton's block-level programming model (Subsection 2.2.1). Decoupled Triton empowers developers to rapidly explore block-level schedule spaces for a particular ML kernel and find efficient Triton kernel schedules.

Figure 3.1 shows the compilation flow of Decoupled Triton. The input to the Decoupled Triton compiler is a DT file that contains a decoupled kernel definition — algorithm and schedule — written in the Decoupled Triton DSL. Listing 3.1 shows an example ML kernel definition written in Decoupled Triton DSL. Given an input

DT file, the Decoupled Triton compiler compiles the algorithm into IR, applies the scheduling to the IR, and generates a Python file as output. Listing 3.2 shows the Python file generated by the Decoupled Triton compiler when the kernel definition in Listing 3.1 is compiled. The generated Python file contains a Triton kernel (lines 5–30), and a wrapper function (lines 32–47). The generated Triton kernel implements the provided algorithm with the provided schedule. The generated wrapper function takes `torch.tensor` and scalar values as arguments (line 32), allocates space for the output tensor (line 35), invokes the generated Triton kernel (lines 38–45), and returns the result of the computation (line 47).

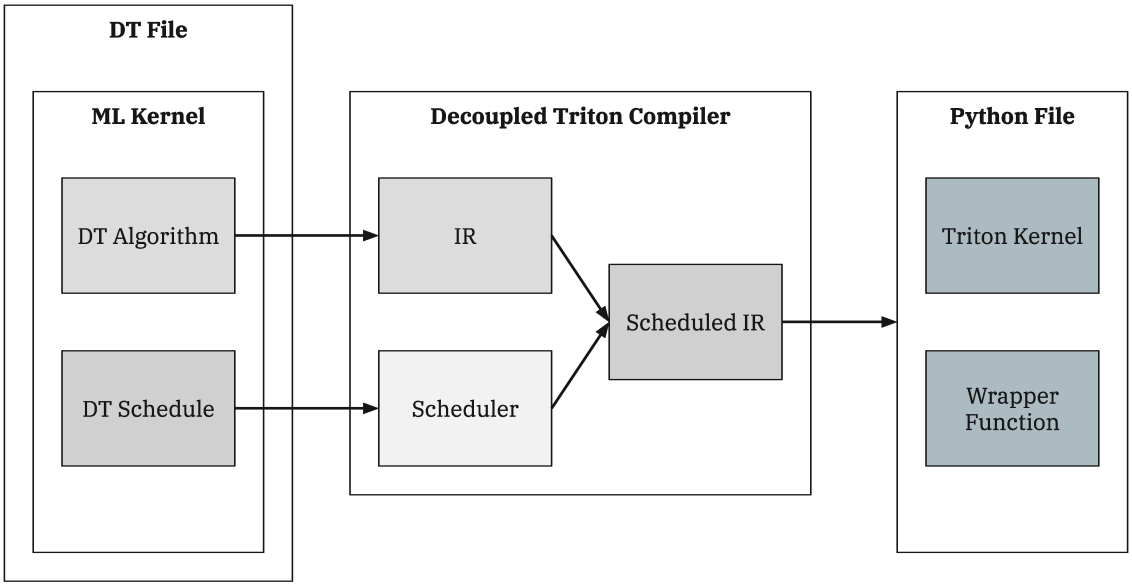


Figure 3.1: Overview of the Decoupled Triton compilation flow.

In Listing 3.1, lines 2–5 are declarations for symbols that are used in the algorithm and schedule definitions of the kernel. The symbols in the Decoupled Triton DSL are summarized in Table 3.1. Line 2 declares a kernel output named `add_out`. Line 3 declares two input tensors named `A` and `B`. Line 4 declares an input scalar named `alpha`. Line 5 declares variable labels for the two dimensions of the kernel compu-

Listing 3.1: A simple kernel definition written in the Decoupled Triton DSL.

```
1  # Declarations
2  Func add_out; # Tensor function to be defined.
3  In   A, B;    # Input tensors.
4  SIn  alpha;   # Input scalar.
5  Var  x, y;    # Dimensions labels.
6
7  # Algorithm
8  add_out[x, y] = alpha * (A[x, y] + B[x, y]);
9
10 # Schedule
11 add_out.block(x:1, y:256);
12 add_out.tensorize(y:64);
13 add_out.compile();
```

tation, `x` and `y`. Line 8 defines the algorithm of the `add_out` function (Section 3.2). Lines 11–12 define the schedule of `add_out` (Section 3.3).

A kernel definition can have multiple `Func` symbols, like in Listing 3.3, to define multiple independent kernels or to define a chain of kernels that may be fused together(Section 3.3).

3.2 Algorithm

All `Func` symbols in a Decoupled Triton kernel definition must have a corresponding algorithm definition. In the Decoupled Triton DSL, algorithms are defined by tensor expressions. A tensor expression can have an arbitrary rank — number of dimensions. Line 8 in Listing 3.1 defines the algorithm of `add_out`. The left-hand side of line 8, `add_out[x, y]`, specifies that we are defining the algorithm for a `Func` named `add_out` with a 2-dimensional output tensor. The dimensions of the output shape are labelled with the `Var` symbols `x` and `y`. The right-hand-side of line 8, `alpha * (A[x, y] + B[x, y])`, is the tensor expression that defines the algorithm of `add_out`. A tensor expression is composed of parentheses, element-wise operations, and

Listing 3.2: The Python file generated by the Decoupled Triton compiler after compiling the kernel definition in Listing 3.1.

```
import torch
import triton
import triton.language as tl

@triton.jit
def add_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    B_ptr,
    B_x_stride: tl.constexpr, B_y_stride: tl.constexpr,
    alpha,
    add_out_ptr,
    add_out_x_stride: tl.constexpr, add_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  y_BLOCK_COUNT = y_SIZE // 256
  x_BLOCK_COUNT = x_SIZE // 1
  y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
  x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
  y_block_start = y_pid * 256
  x_block_start = x_pid * 1
  y_arange = tl.arange(0, 64)
  for y_iter in range(0, 256, 64):
    A = tl.load(A_ptr + x_block_start * A_x_stride +
                (y_block_start + y_iter + y_arange) * A_y_stride)
    B = tl.load(B_ptr + x_block_start * B_x_stride +
                (y_block_start + y_iter + y_arange) * B_y_stride)
    add_out = alpha * (A + B)
    tl.store(add_out_ptr + x_block_start * add_out_x_stride +
             (y_block_start + y_iter + y_arange) * add_out_y_stride,
             add_out)

def add_out(A, B, alpha, x, y):
  A_x_stride, A_y_stride, = A.stride()
  B_x_stride, B_y_stride, = B.stride()
  add_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  add_out_x_stride, add_out_y_stride, = add_out.stride()
  add_out_grid = (triton.cdiv(x, 1) * triton.cdiv(y, 256)),
  add_out_kernel[add_out_grid](
    A, A_x_stride, A_y_stride,
    B, B_x_stride, B_y_stride,
    alpha,
    add_out, add_out_x_stride, add_out_y_stride,
    y, x,
    num_stages=3, num_warps=4
    )

  return add_out
```

Table 3.1: Description of symbols in the Decoupled Triton DSL.

`Func`	A tensor function that is the output of a kernel, must be defined by an algorithm, may be an input to a tensor expression (Section 3.2), and has a schedule defined by scheduling primitives (Section 3.3).
`In`	A tensor input that may be an input to a tensor expression (Section 3.2).
`SIn`	A scalar input that may be a scalar input to a tensor expression (Section 3.2).
`Var`	A dimension label that may be used in scheduling primitives to describe scheduling (Section 3.3) and may be used to define the shapes of both tensors and tensor expressions (Section 3.2).
`RVar`	Same as a `Var`, but, additionally, may be a reduction dimension in a reduction operation (Subsection 3.2.2).

Listing 3.3: A kernel definition for softmax written in the Decoupled Triton DSL.

```
Func softmax_out, sum_exp_A, exp_A;
In   A;
Var  x;
RVar y;

exp_A[x, y]       = exp(A[x, y]);
sum_exp_A[x]      = rsum(exp_A[x, y], y);
softmax_out[x, y] = exp_A[x, y] / reshape(sum_exp_A[x], x, 1);

softmax_out.tensorize(x:4, y:128);
softmax_out.compile();
```

reduction operations on tensors, scalars, and dimension labels. `A[x, y]` and `B[x, y]` are tensor expressions defined as 2-dimensional input tensors with shape `[x, y]`. The `+` between the two input tensors denotes element-wise binary addition between the two tensors. Finally, `alpha` is a scalar input and `*` performs element-wise multiplication between the implicitly broadcast scalar and the result of the parenthesized addition. To summarize, the `add_out` kernel algorithm computes the element-wise addition between two input tensors and multiplies the resulting tensor by a scalar input.

To allow operations between tensor expressions with different shapes, Decoupled Triton uses the following implicit broadcasting semantics: after prepending outer dimensions of size 1 to the shape with the smaller rank, two shapes are compatible if and only if, for each dimension in the operation, the dimension sizes of both shapes are equal or one of the dimension sizes is 1. For example, `A[x, y] + B[y]` is a valid tensor expression, but `A[x, y] + B[x]` and `A[x, y] + B[x, z]` are not. These broadcasting semantics are the same as those of NumPy [13] arrays.[1]

3.2.1 Element-wise operations and parentheses

Parentheses are used in tensor expressions to manipulate the precedence between binary operations. Table 3.2 shows all of the element-wise operations sorted by decreasing precedence. To simplify programming and Triton kernel generation, the Decoupled Triton DSL exposes the built-in math operations that are provided by the `triton.language` Application Programming Interface (API) as element-wise primitives.[2] Listing 3.4 shows a kernel definition that uses the `pow` primitive on line 8. The result of an element-wise operation has the same shape as its inputs. If the inputs of an element-wise operation have incompatible shapes, the tensor expression, and thus the algorithm, is malformed.

[1] https://numpy.org/doc/stable/user/basics.broadcasting.html
[2] https://triton-lang.org/main/python-api/triton.language.html#math-ops

Table 3.2: Element-wise operations supported in the Decoupled Triton DSL sorted by decreasing precedence.

`+`, `-`	Unary addition and subtraction.
`*`, `/`, `%`	Multiplication, division, and remainder.
`+`, `-`	Binary addition and subtraction.
`<`, `<=`, `>`, `>=`, `==`, `!=`	Comparisons.

Listing 3.4: A kernel definition for root mean square layer normalization written in the Decoupled Triton DSL.

```
Func rms_norm_out;
In   X, W;
SIn  Eps, Offset;
Var  x, y;
RVar k;

rms_norm_out[x, y] = X[x, y] *
                     rsqrt(rsum(pow(X[x, k], 2), k) / len(y) + Eps) *
                     (Offset + W[y]);

rms_norm_out.compile();
```

3.2.2 Reduction operations

Figure 3.2 illustrates the difference between a reduction operation and an element-wise operation. A reduction operation reduces the rank of a tensor by aggregating all of the elements along a dimension — the reduction dimension — using an associative binary function. To describe a reduction operation in the Decoupled Triton DSL, reduction primitives are used. When using a reduction primitive, you must pass an `RVar` that represents the reduction dimension as the last argument. Listing 3.4 shows an example kernel definition that describes a summation reduction along the `k` dimension in the Decoupled Triton DSL using the `rsum` reduction primitive on line 8. The reduction primitives exposed in the Decoupled Triton DSL are listed in Table 3.3.

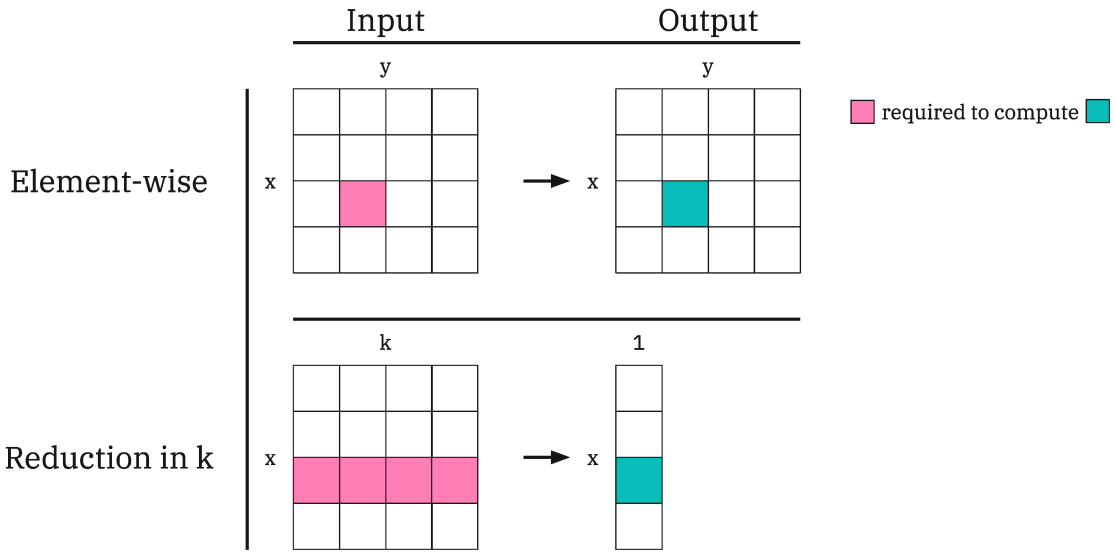


Figure 3.2: Comparison of element-wise and reduction operations. Element-wise maps each element in the input to an output element, while reduction collapses across one dimension (here k) to produce a smaller output.

Table 3.3: Reduction primitives supported in the Decoupled Triton DSL.

`rmax(input, rvar)`	Returns the maximum of all elements in the tensor, `input`, along the reduction dimension `rvar`.
`rmin(input, rvar)`	Returns the minimum of all elements in the tensor, `input`, along the reduction dimension `rvar`.
`rsum(input, rvar)`	Returns the sum of all elements in the tensor, `input`, along the reduction dimension `rvar`.
`rdot(left, right, rvar)`	Returns the matrix product of the tensors `left` and `right`. The inner dimensions of the tensors must match the reduction dimension `rvar`.

3.2.3 Other primitives

Many ML kernels use the size of a tensor dimension in their computation. To describe dimension sizes in a tensor expression, Decoupled Triton exposes the `len` primitive that takes a `Var` as input and returns a scalar representing the size of the dimension matching the `Var`. Listing 3.4 shows an example kernel definition that uses the `len` primitive on line 8.

The Decoupled Triton DSL also exposes the `reshape` primitive that maps directly to the `triton.language` primitive with the same name.[3] `reshape` takes an input tensor expression followed by any number of `Var` or literal 1 arguments to define the output shape. Listing 3.3 shows an example kernel definition that uses the `reshape` primitive on line 8.

The Decoupled Triton DSL also exposes the `program_id` primitive, which maps to the `triton.language` primitive with the same name.[4] The `program_id` primitive returns a scalar representing the program instance index in the Triton program launch grid. Section 3.3 discusses Triton program instances.

[3]https://triton-lang.org/main/python-api/generated/triton.language.reshape.html#triton.language.reshape

[4]https://triton-lang.org/main/python-api/generated/triton.language.program_id.html#triton.language.program_id

3.3 Schedule

The decoupling of the algorithm from the schedule simplifies the process of writing ML kernels, allowing experts to define a scheduling based on their knowledge of the memory hierarchy or specialized hardware computing infrastructure. This decoupling also enables developers to find an efficient schedule by rapidly exploring a schedule space. This section describes the schedule representation in Decoupled Triton.

The Decoupled Triton compiler generates Triton kernels. It adopts the block-level programming paradigm from Triton — any low-level details handled by the Triton compiler such as shared memory and thread coalescence are not exposed in the Decoupled Triton scheduling API. Consequently, schedules defined in Decoupled Triton are block-level schedules. The block-level schedule of an ML kernel defines how the output is computed at the thread-block level.

3.3.1 Scheduling primitives

When writing an ML kernel in the Decoupled Triton DSL, developers use scheduling primitives to define the block-level schedule. The signature of every scheduling primitive is as follows: `FUNC.PRIMITIVE_NAME(ARGS)`. A scheduling primitive must be applied to a `Func` and may take arguments in some form depending on the particular primitive. Lines 11–13 of Listing 3.1 shows an example block-level schedule defined by three scheduling primitives in the Decoupled Triton DSL. Table 3.4 provides a summary of the schedule primitives available in the Decoupled Triton DSL. The scheduling primitives presented in this section form the complete initial set of scheduling primitives available in Decoupled Triton. To form this initial set, we were primarily focused on the scheduling primitives with the highest expected performance impact based upon our intuition and preliminary experimentation. The Decoupled Triton DSL and compiler are extensible, especially with new scheduling primitives.

In particular, Decoupled Triton can be extended with any scheduling primitive that expresses a code transformation, which can be applied to Triton code. For example, many of the loop-level scheduling primitives supported by languages like Halide [31], such as `split`, `reorder`, and `unroll`, could be added to the Decoupled Triton DSL and compiler.

Table 3.4: Summary of scheduling primitives available in the Decoupled Triton DSL.

`compile`	Compiles a `Func` to generate a wrapper function and all of the Triton kernels required to implement the `Func`.
`block`	Block a `Func`'s implementation in a dimension according to a block size.
`tensorize`	Tensorize a `Func`'s implementation in a dimension according to a tensor size.
`map`	Defines a `Func`'s mapping from program instance index to output block.
`fuse_at`	Fuse a `Func` into another `Func` at a particular loop dimension.
`num_warps`	Sets the `num_warps` argument for a `Func`'s Triton kernel.[5]
`num_stages`	Sets the `num_stages` argument for a `Func`'s Triton kernel.[6]

Listing 3.5 will be referenced throughout this section as an example ML kernel definition in the Decoupled Triton DSL.

3.3.1.1 Compile

When applied to a `Func f`, the scheduling primitive `compile` directs the Decoupled Triton compiler to generate a Triton kernel and wrapper function for `f` that computes the algorithm of `f` with the schedule applied by other scheduling directives in the kernel definition. Listing 3.6 shows an example of a generated Triton kernel (lines 5–17) and wrapper function (lines 20–33). The only scheduling primitive in Listing 3.5

[5]https://triton-lang.org/main/python-api/generated/triton.Config.html#triton.Config
[6]https://triton-lang.org/main/python-api/generated/triton.Config.html#triton.Config

Listing 3.5: A kernel definition for the rectified linear unit function written in the Decoupled Triton DSL.

```
1 Func relu_out;
2 In   A;
3 Var  x, y;
4
5 relu_out[x, y] = maximum(0, A[x, y]);
6
7 relu_out.compile();
```

is `compile`. Therefore, the ML kernel is generated with a default schedule — a single program instance computes the entire output sequentially without using any tensor operations.

If the `compile` primitive is applied to a `Func f` and the algorithm of `f` depends on the result of another `Func g` as input, directly or indirectly, and `g` is not fused into `f` with the `fuse_at` scheduling primitive, then the Decoupled Triton compiler also generates the Triton kernel for `g` and calls it in the wrapper function before calling `f`. Listing 3.3 and Listing 3.7 show such an ML kernel definition written in the Decoupled Triton DSL and the corresponding Python file generated by the Decoupled Triton compiler, respectively. The schedule in Listing 3.3 contains a `compile` primitive applied to `softmax_out` (line 11) and no `fuse_at` primitives. Listing 3.7 contains a generated Triton kernel for `softmax_out` (lines 13–15), the `Func` that `compile` is applied to, and, because the algorithm of `softmax_out` depends on them, both `exp_A_kernel` (lines 5–6) and `sum_exp_A_kernel` (lines 9–11). The wrapper function (lines 17–44) first calls the kernel for `exp_A_kernel` (lines 22–24), then the kernel for `sum_exp_A_kernel` (lines 30–32), and finally the kernel for `softmax_out` (lines 40–42).

The `compile` primitive is the only scheduling primitive that takes no arguments. The Decoupled Triton compiler will generate no code if the input kernel definition

Listing 3.6: The Python file generated by the Decoupled Triton compiler after compiling the kernel definition in Listing 3.5.

```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    relu_out_ptr,
    relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    for y_iter in range(0, y_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
      relu_out = tl.maximum(0, A)
      tl.store(relu_out_ptr + x_iter * relu_out_x_stride +
               y_iter * relu_out_y_stride, relu_out)


def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (1,)
  relu_out_kernel[relu_out_grid](
    A,
    A_x_stride, A_y_stride,
    relu_out,
    relu_out_x_stride, relu_out_y_stride,
    y, x,
    num_stages=3, num_warps=4)

  return relu_out
```

Listing 3.7: The Python file generated by the Decoupled Triton compiler after compiling the kernel definition in Listing 3.3. Triton kernel arguments and bodies have been omitted for brevity.

```
import torch
import triton
import triton.language as tl

@triton.jit
def exp_A_kernel(...):
  ...

@triton.jit
def sum_exp_A_kernel(...):
  ...

@triton.jit
def softmax_out_kernel(...):
  ...

def softmax_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  exp_A = torch.empty(x, y, dtype=torch.float32, device='cuda')
  exp_A_x_stride, exp_A_y_stride, = exp_A.stride()
  exp_A_grid = (1,)
  exp_A_kernel[exp_A_grid](
    A, A_x_stride, A_y_stride, exp_A, exp_A_x_stride, exp_A_y_stride,
    y, x, num_stages=3, num_warps=4)

  exp_A_x_stride, exp_A_y_stride, = exp_A.stride()
  sum_exp_A = torch.empty(x, dtype=torch.float32, device='cuda')
  sum_exp_A_x_stride, = sum_exp_A.stride()
  sum_exp_A_grid = (1,)
  sum_exp_A_kernel[sum_exp_A_grid](
    exp_A, exp_A_x_stride, exp_A_y_stride, sum_exp_A, sum_exp_A_x_stride,
    y, x, num_stages=3, num_warps=4)

  exp_A_x_stride, exp_A_y_stride, = exp_A.stride()
  sum_exp_A_x_stride, = sum_exp_A.stride()
  softmax_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  softmax_out_x_stride, softmax_out_y_stride, = softmax_out.stride()
  softmax_out_grid = (1,)
  softmax_out_kernel[softmax_out_grid](
    exp_A, exp_A_x_stride, exp_A_y_stride, sum_exp_A, sum_exp_A_x_stride,
    softmax_out, softmax_out_x_stride, softmax_out_y_stride,
    y, x, num_stages=3, num_warps=4)

  return softmax_out
```

does not include any `compile` primitive because the compiler has no instruction about which `Func` to compile.

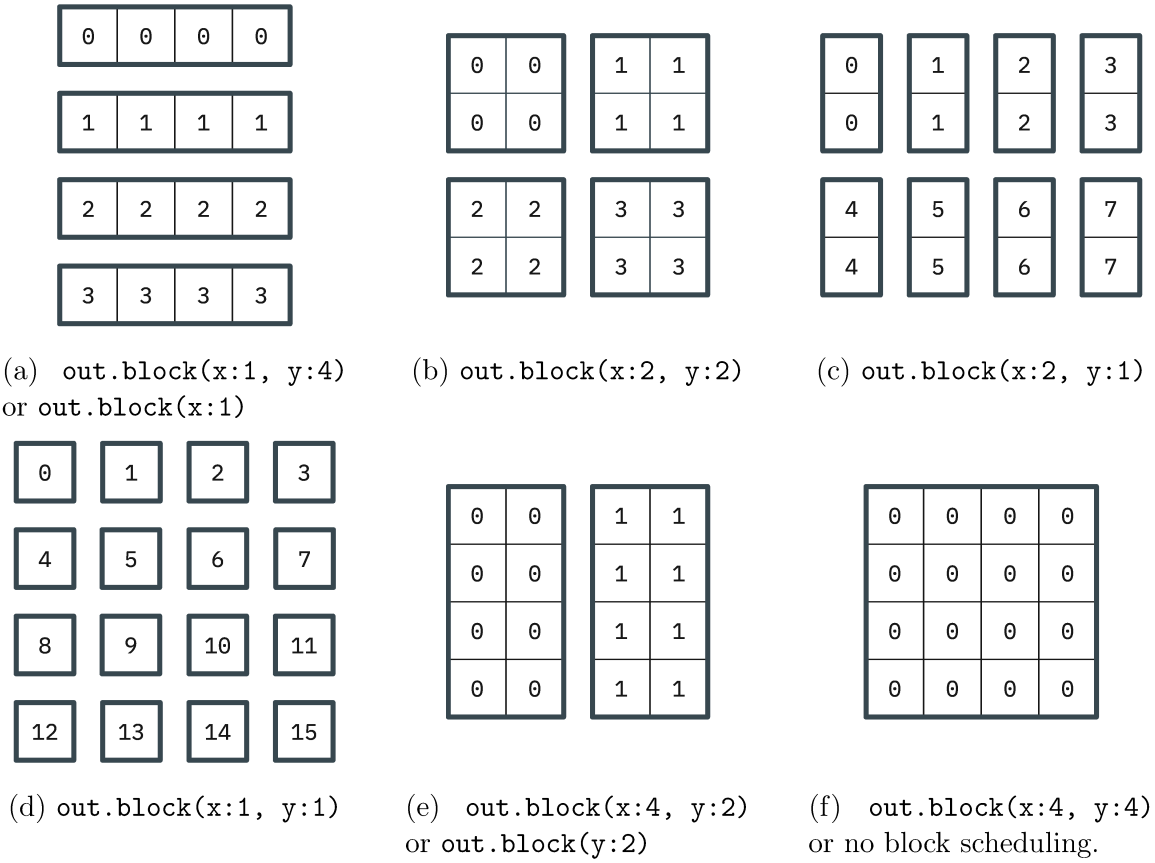
3.3.1.2 Block

The `block` primitive in the Decoupled Triton DSL partitions the output tensor of a `Func` into blocks such that each block is computed by a separate program instance launched from the kernel launch grid. In Decoupled Triton, given a `Func f`, a dimension is either blocked and it has a specified block size in `f`'s schedule, or not blocked in `f`'s schedule. The block size of a dimension `x` determines the number of output elements along `x` that a single program instance is responsible for computing. That is, the output block dimensions are defined by the block sizes of the blocked dimensions in `f`. Therefore, the number of program instances in the launch grid can be computed using the following equation:

$$\#\text{ of program instances} = \#\text{ of blocks} = \prod_{x:x \text{ is blocked}} \frac{x_{\text{size}}}{x_{\text{block size}}}$$

To use the `block` scheduling primitive, pass a list of argument pairs specifying a dimension (`Var`) and block size (`unsigned integer`) using the following syntax: `f.block(VAR:SIZE, VAR:SIZE, ...)`. The blocking strategy described in this section does not apply to reduction dimensions (`RVar`). Blocking in a reduction dimension is possible, and the strategy to do so is described in Subsection 3.3.2. To minimize control-flow divergence and to simplify code generation, Decoupled Triton requires the block size of a dimension to be a factor of its size. Figure 3.3 shows several examples of how the `block` primitive partitions the computation of the output tensor into blocks. Each subfigure shows the partitioning of the output tensor computation into blocks based on the corresponding `block` scheduling with the `x` and `y` vars in the vertical and horizontal dimensions, respectively. The smallest granularity box represents an element of the output tensor. The output tensor is separated

into thick-outlined boxes that denote the blocks. An output element labeled with the number n is part of the n-th block and thus computed by the n-th program instance.



(a) `out.block(x:1, y:4)` or `out.block(x:1)`

(b) `out.block(x:2, y:2)`

(c) `out.block(x:2, y:1)`

(d) `out.block(x:1, y:1)`

(e) `out.block(x:4, y:2)` or `out.block(y:2)`

(f) `out.block(x:4, y:4)` or no block scheduling.

Figure 3.3: Visual examples of the `block` scheduling primitive applied to a `Func` that computes a 4x4 output tensor.

Listing 3.8 shows an example Triton kernel and wrapper function generated by the Decoupled Triton compiler for a kernel definition whose schedule includes the `block` primitive. In general, the `block` primitive code transformation:

1. Modify the kernel launch in the wrapper function to use a launch grid with the appropriate number of program instances (lines 30–31).

2. For each blocked dimension, compute the dimensional program instance index from the 1-dimensional program instance index into the launch grid (lines 12–

15).

3. For each blocked dimension, compute the block start offset. This offset points to the output block that the program instance must compute relative to the memory address of a tensor (lines 16–17).

4. Set the upper-bound of each blocked dimension's loop to its block size (lines 18–19).

5. Modify tensor memory accesses in the Triton kernel by computing the addresses according to the block start offsets (lines 20–21 and lines 23–24).

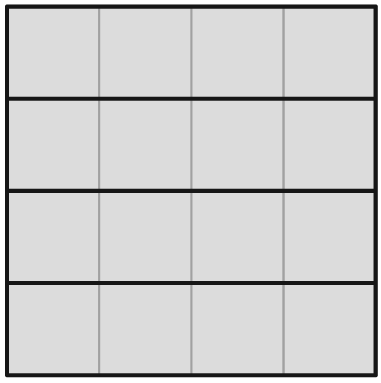
3.3.1.3 Tensorize

The `tensorize` primitive in the Decoupled Triton DSL enables single-instruction-multiple-data (SIMD) tensor operations within a program instance. It partitions the output block of a program instance into tensors, such that each tensor is computed by a sequential loop iteration in the program instance. A dimension is fully tensorized in `f`'s schedule if and only if its tensor size is equal to its block size in `f`'s schedule or the tensor size is explicitly `0`. Otherwise, the tensorized dimension is partially tensorized. The tensor size specifies the step size of the loop in the kernel. Consequently, if a dimension is fully tensorized in `f`'s schedule, then the kernel generated for `f` does not have a loop iterating over the dimension. If a dimension is blocked, its tensor size cannot be greater than its block size. Figure 3.4 shows several examples of how the `tensorize` primitive partitions the output block of a program instance into tensors. Each subfigure shows how the computation of the output tensor is partitioned across sequential loop iterations, each computing a tensor of the output using tensor operations, based on the corresponding `tensorize` scheduling. The `x` and `y` vars are not blocked and visualized in the vertical and horizontal dimensions, respectively.

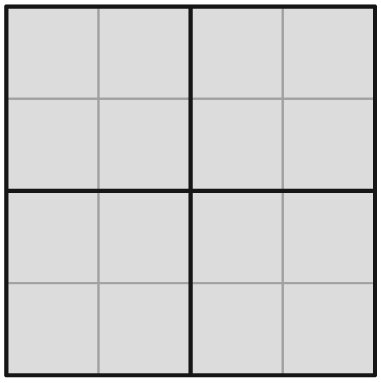
Listing 3.8: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitive `relu_out.block(x:16, y:64)` to the kernel definition in Listing 3.5.

```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    relu_out_ptr,
    relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  y_BLOCK_COUNT = y_SIZE // 64
  x_BLOCK_COUNT = x_SIZE // 16
  y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
  x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
  y_block_start = y_pid * 64
  x_block_start = x_pid * 16
  for x_iter in range(0, 16, 1):
    for y_iter in range(0, 64, 1):
      A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +
                  (y_block_start + y_iter) * A_y_stride)
      relu_out = tl.maximum(0, A)
      tl.store(relu_out_ptr + (x_block_start + x_iter) * relu_out_x_stride +
               (y_block_start + y_iter) * relu_out_y_stride, relu_out)

def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (triton.cdiv(x, 16) * triton.cdiv(y, 64)),
  relu_out_kernel[relu_out_grid](
    A,
    A_x_stride, A_y_stride,
    relu_out,
    relu_out_x_stride, relu_out_y_stride,
    y, x,
    num_stages=3, num_warps=4)

  return relu_out
```

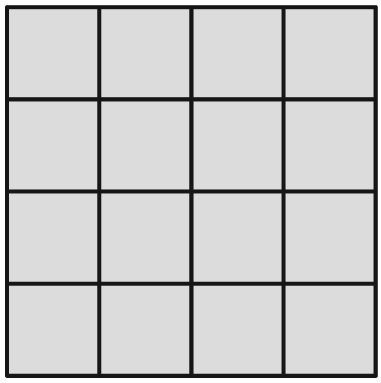
The smallest granularity box represents an element of the output tensor. The output tensor is divided into dark-outlined boxes that denote the tensors.



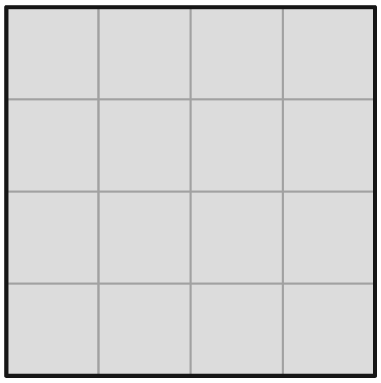
(a) `out.tensorize(x:1, y:4)` or `out.tensorize(x:1, y:0)`



(b) `out.tensorize(x:2, y:2)`



(c) `out.tensorize(x:1, y:1)` or no tensorize scheduling



(d) `out.tensorize(x:4, y:4)` or `out.tensorize(x:0, y:0)`

Figure 3.4: Visual examples of the `tensorize` scheduling primitive applied to a `Func` that computes a 4x4 output tensor.

Listing 3.9 shows an example Triton kernel and wrapper function generated by the Decoupled Triton compiler for a kernel definition whose schedule includes the `tensorize` primitive. The step of the `x` loop is modified to equal the tensor size of `x` (line 15) and the `y` loop is removed because `y` is fully tensorized. The kernel uses range

tensors (lines 13–14) to compute address tensors that tensorize the load (lines 16–17) and store (lines 19–21) instructions. The Triton kernel must include an argument, because `y` is fully tensorized but not blocked, which is equal to the smallest power of 2 greater than or equal to the size of `y` (lines 12 and 34), to create the address tensors. Listing 3.10 shows an example Triton kernel and wrapper function generated by the Decoupled Triton compiler for a kernel definition whose schedule includes both `block` and `tensorize` primitives. The block size is used as the tensor size for the blocked and fully tensorized dimension `x` (line 19).

3.3.1.4 Map

The number of program instances that can execute in parallel at a given time is limited by the available hardware resources. Also, depending on the algorithm, program instances may reuse some of the data loaded into the cache by other program instances. Consequently, the order in which program instances are launched may impact the efficiency of a kernel. The `map` scheduling primitive allows users to specify a mapping from program instance index to output tensor block using a notation for describing loop nests. Map takes any number of arguments, each argument represents a loop level in the loop nest with the argument order describing the nesting order. Figure 3.5 shows several examples of how the `map` primitive changes the program instance to output block mapping of a kernel. Each box represents an output block of the output tensor. The `x` and `y` vars are visualized in the vertical and horizontal dimensions, respectively. Each subfigure describes how a program instance maps to the output block that it computes. The numbers and arrows specify the program instance ordering. A block labeled with the number n is computed by the n-th program instance in the launch grid.

A `map` primitive argument can either be of the form `VAR` or `VAR:ID/FACTOR`. In the first case, the dimension loop for `VAR` is placed in the loop nest. In the second

Listing 3.9: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitive `relu_out.tensorize(x:64, y:0)` to the kernel definition in Listing 3.5.

```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
  A_ptr,
  A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
  relu_out_ptr,
  relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
  y_SIZE: tl.constexpr, x_SIZE: tl.constexpr,
  y_NEXT_POWER_OF_TWO: tl.constexpr):
  y_arange = tl.arange(0, y_NEXT_POWER_OF_TWO)
  x_arange = tl.arange(0, 64)
  for x_iter in range(0, x_SIZE, 64):
    A = tl.load(A_ptr + (x_iter + x_arange[:, None]) * A_x_stride +
                y_arange[None, :] * A_y_stride)
    relu_out = tl.maximum(0, A)
    tl.store(relu_out_ptr + (x_iter + x_arange[:, None]) *
             relu_out_x_stride + y_arange[None, :] * relu_out_y_stride,
             relu_out)

def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (1,)
  relu_out_kernel[relu_out_grid](
    A,
    A_x_stride, A_y_stride,
    relu_out,
    relu_out_x_stride, relu_out_y_stride,
    y, x,
    triton.next_power_of_2(y),
    num_stages=3, num_warps=4)

  return relu_out
```

Listing 3.10: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitives `relu_out.block(x:4, y:128)` and `relu_out.tensorize(x:0, y:16)` to the kernel definition in Listing 3.5.

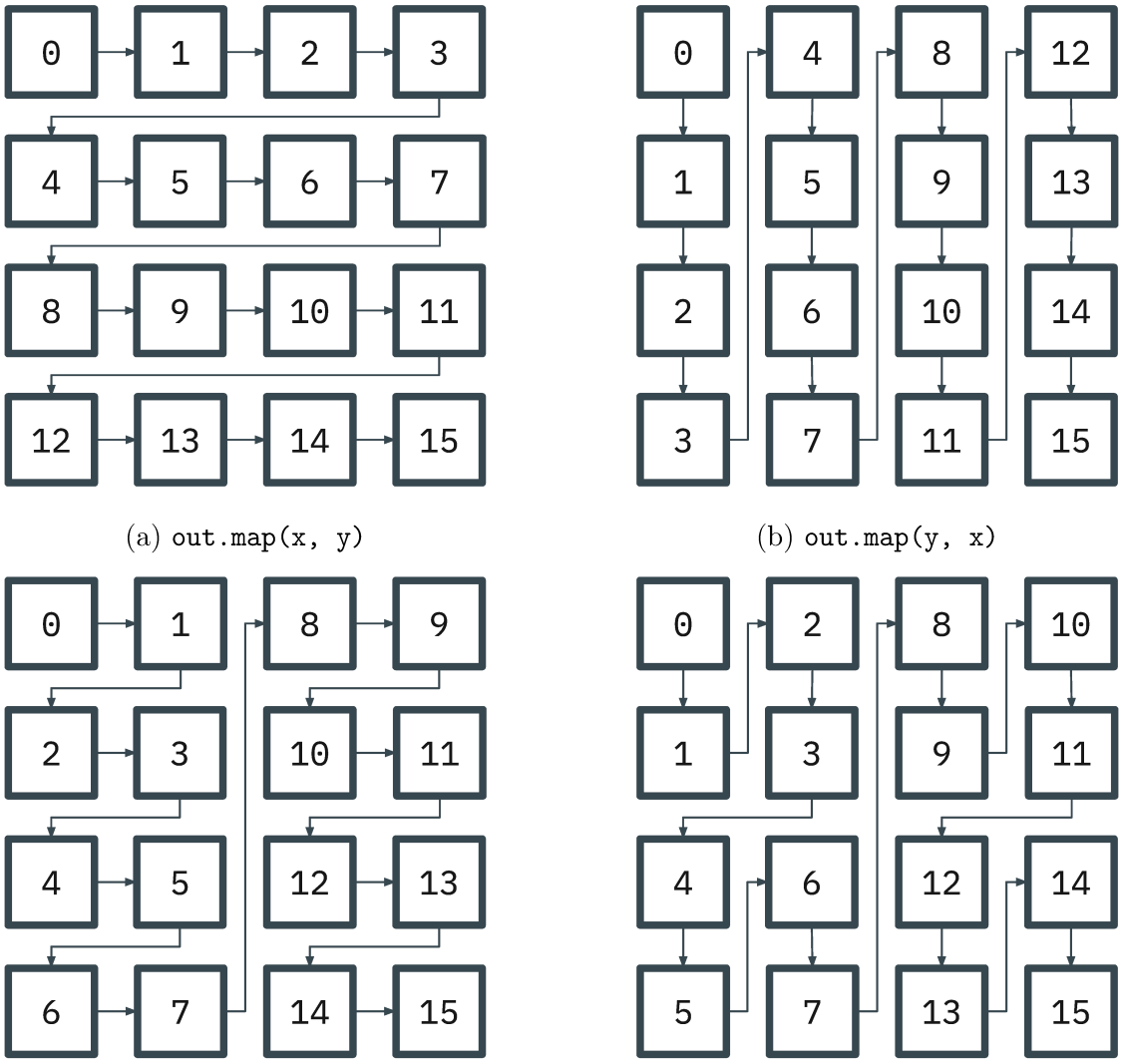
```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    relu_out_ptr,
    relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  y_BLOCK_COUNT = y_SIZE // 128
  x_BLOCK_COUNT = x_SIZE // 4
  y_pid = tl.program_id(0).to(tl.int64) // x_BLOCK_COUNT % y_BLOCK_COUNT
  x_pid = tl.program_id(0).to(tl.int64) % x_BLOCK_COUNT
  y_block_start = y_pid * 128
  x_block_start = x_pid * 4
  y_arange = tl.arange(0, 16)
  x_arange = tl.arange(0, 4)
  for y_iter in range(0, 128, 16):
    A = tl.load(A_ptr + (x_block_start + x_arange[:, None]) * A_x_stride +
                (y_block_start + y_iter + y_arange[None, :]) * A_y_stride)
    relu_out = tl.maximum(0, A)
    tl.store(relu_out_ptr + (x_block_start + x_arange[:, None]) *
             relu_out_x_stride + (y_block_start + y_iter +
             y_arange[None, :]) * relu_out_y_stride, relu_out)

def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (triton.cdiv(x, 4) * triton.cdiv(y, 128)),
  relu_out_kernel[relu_out_grid](A, A_x_stride, A_y_stride, relu_out,
    relu_out_x_stride, relu_out_y_stride, y, x, num_stages=3, num_warps=4)

  return relu_out
```

case, the dimension loop for VAR is first split by the FACTOR, then it is placed in the loop nest. The new dimension loop bounded by FACTOR is given the name ID and must be placed in the loop nest by a later map argument. The traversal pattern of the loop nest defined by the map scheduling primitive defines the program mapping. Subsection 4.4.3 includes an experimental evaluation on the performance impact of the map scheduling directive on a matrix product ML kernel.

Listing 3.11 shows an example Triton kernel and wrapper function generated by the Decoupled Triton compiler for a kernel definition whose schedule includes the map and block primitives. The program instance mapping is applied from lines 14–18.

3.3.1.5 Fuse at

The fuse_at primitive directs the compiler to fuse the computation of a Func into another Func at a dimension loop level (Var). For instance, the scheduling primitive f.fuse_at(g, x) instructs the compiler to fuse the computation of f into the kernel of g at the loop level of x. If g does not depend on the output of f in its algorithm, then the scheduling primitive is malformed. Furthermore, for legal fusion, the Decoupled Triton compiler requires that x is a dimension of both f and g, and that the outer dimensions, relative to x, in the shape of both f and g are equal. These requirements are the same requirements that must be satisfied for a compiler to perform successive loop-fusion transformations [42] between two nested loops, because the shape of a Func defines the loop nest that computes its output. For example, f[u, v, k, w] can be fused into g[u, v, l, w] at the dimension u or v but not at the dimension k, l, or w. The fuse_at primitive also alters the dependencies of g. The dependency on f is removed, and any of f's dependencies that are not already dependencies of g are added. These dependency updates are important to ensure that the wrapper function can execute the entire algorithm of g and no extraneous kernels are generated. If f is not fully fused into g — x is not the innermost dimension of f — and at least one



(a) `out.map(x, y)`

(b) `out.map(y, x)`

(c) `out.map(y:yi/2, x, yi)`

(d) `out.map(y:yi/2, x:xi/2, yi, xi)`

Figure 3.5: Examples of the `map` scheduling primitive applied to a `Func` with a 4x4 launch grid containing 16 program instances.

Listing 3.11: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitives `relu_out.block(x:8, y:16)` and `relu_out.map(x:xi/4, y, xi)` to the kernel definition in Listing 3.5.

```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    relu_out_ptr,
    relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  y_BLOCK_COUNT = y_SIZE // 16
  x_BLOCK_COUNT = x_SIZE // 8
  x_pid = tl.program_id(0).to(tl.int64) // (y_BLOCK_COUNT * 4) %
          (x_BLOCK_COUNT // 4)
  y_pid = tl.program_id(0).to(tl.int64) // 4 % y_BLOCK_COUNT
  xi_pid = tl.program_id(0).to(tl.int64) % 4
  x_pid_ = x_pid * 4 + xi_pid
  y_block_start = y_pid * 16
  x_block_start = x_pid_ * 8
  for x_iter in range(0, 8, 1):
    for y_iter in range(0, 16, 1):
      A = tl.load(A_ptr + (x_block_start + x_iter) * A_x_stride +
                  (y_block_start + y_iter) * A_y_stride)
      relu_out = tl.maximum(0, A)
      tl.store(relu_out_ptr + (x_block_start + x_iter) * relu_out_x_stride +
               (y_block_start + y_iter) * relu_out_y_stride, relu_out)


def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (triton.cdiv(x, 8) * triton.cdiv(y, 16)),
  relu_out_kernel[relu_out_grid](
    A,
    A_x_stride, A_y_stride,
    relu_out,
    relu_out_x_stride, relu_out_y_stride,
    y, x,
    num_stages=3, num_warps=4)

  return relu_out
```

of the inner dimensions is not fully tensorized, then the Decoupled Triton compiler must allocate and manage an additional global memory tensor that holds the result tensor of `f`. We cannot use shared memory for this tensor because Triton does not expose shared memory details, and there is no way of storing to or loading from a slice of a local tensor.

Listing 3.12 shows a kernel definition that can be scheduled using the `fuse_at` primitive. The algorithm of `swish_out` takes the `Func tmp` as an input. Listing 3.13 shows the generated Triton kernel and wrapper function when `tmp` is fully fused into `swish_out`. The algorithm of `tmp` is fused in line 16. Listing 3.14 shows the generated Triton kernel and wrapper function when `tmp` is partially fused into `swish_out`. The `y` loop of `tmp` is fused in lines 16–20. The wrapper function must allocate an additional global memory tensor on line 33 that stores the result of `tmp` (lines 19–20). Listing 3.15 shows the generated Triton kernel and wrapper function when `tmp` is partially fused into `swish_out` and `y` is fully tensorized. The wrapper function does not need an additional global memory tensor and the algorithm of `tmp` is fused in line 17.

Listing 3.12: A kernel definition for the swish function written in the Decoupled Triton DSL.

```
1 Func swish_out, tmp;
2 In   A;
3 SIn  beta;
4 Var  x, y;
5
6 tmp[x, y]       = sigmoid(beta * A[x, y]);
7 swish_out[x, y] = A[x, y] * tmp[x, y];
8
9 swish_out.compile();
```

Listing 3.13: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitive `tmp.fuse_at(swish_out, y)` to the kernel definition in Listing 3.12.

```
import torch
import triton
import triton.language as tl

@triton.jit
def swish_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    beta,
    swish_out_ptr,
    swish_out_x_stride: tl.constexpr, swish_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    for y_iter in range(0, y_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
      tmp = tl.sigmoid(beta * A)
      swish_out = A * tmp
      tl.store(swish_out_ptr + x_iter * swish_out_x_stride +
               y_iter * swish_out_y_stride, swish_out)

def swish_out(A, beta, y, x):
  A_x_stride, A_y_stride, = A.stride()
  swish_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  swish_out_x_stride, swish_out_y_stride, = swish_out.stride()
  swish_out_grid = (1,)
  swish_out_kernel[swish_out_grid](
    A,
    A_x_stride, A_y_stride,
    beta,
    swish_out,
    swish_out_x_stride, swish_out_y_stride,
    y, x,
    num_stages=3, num_warps=4)

  return swish_out
```

Listing 3.14: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitive `tmp.fuse_at(swish_out, x)` to the kernel definition in Listing 3.12.

```
import torch
import triton
import triton.language as tl

@triton.jit
def swish_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    beta,
    tmp_fuse_ptr,
    tmp_fuse_x_stride: tl.constexpr, tmp_fuse_y_stride: tl.constexpr,
    swish_out_ptr,
    swish_out_x_stride: tl.constexpr, swish_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    for y_iter in range(0, y_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
      tmp = tl.sigmoid(beta * A)
      tl.store(tmp_fuse_ptr + x_iter * tmp_fuse_x_stride +
               y_iter * tmp_fuse_y_stride, tmp)
    for y_iter in range(0, y_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
      tmp = tl.load(tmp_fuse_ptr + x_iter * tmp_fuse_x_stride +
                    y_iter * tmp_fuse_y_stride)
      swish_out = A * tmp
      tl.store(swish_out_ptr + x_iter * swish_out_x_stride +
               y_iter * swish_out_y_stride, swish_out)

def swish_out(A, beta, y, x):
  A_x_stride, A_y_stride, = A.stride()
  swish_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  swish_out_x_stride, swish_out_y_stride, = swish_out.stride()
  tmp_fuse = torch.empty(x, y, dtype=torch.float32, device='cuda')
  tmp_fuse_x_stride, tmp_fuse_y_stride, = tmp_fuse.stride()
  swish_out_grid = (1,)
  swish_out_kernel[swish_out_grid](
    A,
    A_x_stride, A_y_stride,
    beta,
    tmp_fuse,
    tmp_fuse_x_stride, tmp_fuse_y_stride,
    swish_out,
    swish_out_x_stride, swish_out_y_stride,
    y, x,
    num_stages=3, num_warps=4)

  return swish_out
```

Listing 3.15: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitives `swish_out.tensorize(y:0)` and `tmp.fuse_at(swish_out, x)` to the kernel definition in Listing 3.12.

```
import torch
import triton
import triton.language as tl

@triton.jit
def swish_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    beta,
    swish_out_ptr,
    swish_out_x_stride: tl.constexpr, swish_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr,
    y_NEXT_POWER_OF_TWO: tl.constexpr):
  y_arange = tl.arange(0, y_NEXT_POWER_OF_TWO)
  for x_iter in range(0, x_SIZE, 1):
    A = tl.load(A_ptr + x_iter * A_x_stride + y_arange * A_y_stride)
    tmp = tl.sigmoid(beta * A)
    swish_out = A * tmp
    tl.store(swish_out_ptr + x_iter * swish_out_x_stride +
             y_arange * swish_out_y_stride, swish_out)

def swish_out(A, beta, y, x):
  A_x_stride, A_y_stride, = A.stride()
  swish_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  swish_out_x_stride, swish_out_y_stride, = swish_out.stride()
  swish_out_grid = (1,)
  swish_out_kernel[swish_out_grid](
    A,
    A_x_stride, A_y_stride,
    beta,
    swish_out,
    swish_out_x_stride, swish_out_y_stride,
    y, x,
    triton.next_power_of_2(y),
    num_stages=3, num_warps=4)

  return swish_out
```

3.3.1.6 Num warps and num stages

The `num_warps` and `num_stages` scheduling primitives each take a single unsigned integer argument and set the `num_warps` and `num_stages`, respectively, configuration parameter of the Triton just-in-time (JIT) compiler.[7] Listing 3.16 shows an example Triton kernel and wrapper function generated by the Decoupled Triton compiler for a kernel definition whose schedule includes both the `num_warps` and `num_stages` primitive. The `num_warps` and `num_stages` Triton configuration parameters are set on line 30.

3.3.2 Scheduling reductions

Listing 3.17 shows a simple reduction ML kernel defined in the Decoupled Triton DSL. Listing 3.18 shows the Triton kernel and wrapper function generated by the Decoupled Triton compiler for a reduction operation with no scheduling. A local accumulator variable (line 13) is used to accumulate the reduction across the reduction dimension as the program instance iterates through the reduction dimension loop (lines 14–16).

Listing 3.19 shows how tensorizing a reduction dimension influences the Triton kernel implementation. A tensor accumulator is used instead of a scalar (line 14), and the built-in Triton reduction operation is used to compute the result of the reduction after iterating through the entire reduction dimension (line 19).

Blocking a reduction dimension is not as simple. As illustrated in Figure 3.2, computing a single output element of a reduction operation requires all of the elements along the reduction dimension. Consequently, a program instance computing a reduction is required to use the entire reduction dimension of the reduction's input. Therefore, we cannot apply `block` to a reduction dimension using the same strategy as with a non-reduction dimension. For small reduction dimensions, this limitation

[7] https://triton-lang.org/main/python-api/generated/triton.Config.html#triton.Config

Listing 3.16: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitives `relu_out.num_warps(8)` and `relu_out.num_stages(4)` to the kernel definition in Listing 3.5.

```
import torch
import triton
import triton.language as tl

@triton.jit
def relu_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_y_stride: tl.constexpr,
    relu_out_ptr,
    relu_out_x_stride: tl.constexpr, relu_out_y_stride: tl.constexpr,
    y_SIZE: tl.constexpr, x_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    for y_iter in range(0, y_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + y_iter * A_y_stride)
      relu_out = tl.maximum(0, A)
      tl.store(relu_out_ptr + x_iter * relu_out_x_stride +
               y_iter * relu_out_y_stride, relu_out)

def relu_out(A, y, x):
  A_x_stride, A_y_stride, = A.stride()
  relu_out = torch.empty(x, y, dtype=torch.float32, device='cuda')
  relu_out_x_stride, relu_out_y_stride, = relu_out.stride()
  relu_out_grid = (1,)
  relu_out_kernel[relu_out_grid](
    A,
    A_x_stride, A_y_stride,
    relu_out,
    relu_out_x_stride, relu_out_y_stride,
    y, x,
    num_stages=4, num_warps=8)

  return relu_out
```

Listing 3.17: A kernel definition for the sum reduction function written in the Decoupled Triton DSL.

```
1 Func sum_out;
2 In   A;
3 Var  x;
4 RVar k;
5
6 sum_out[x] = rsum(A[x, k], k);
7
8 sum_out.compile();
```

may have no performance impact. However, as the size of a reduction dimension grows, it becomes increasingly important to exploit block-level parallelism in that dimension.

One method we experimented with was to use a global lock and atomic operations to create synchronization barriers between program instances. Using the synchronization barriers to avoid any data races, we could block the reduction dimension and implement a parallel tree-reduction across program instances. Unfortunately, Triton is not designed for this kind of programming pattern, and the overhead of the synchronization barriers was massive. Instead, we use kernel launches to implement the synchronization. In particular, to block a reduction dimension, we split the reduction into two separate kernels. The first kernel is blocked in the reduction dimension, and each program instance computes a partial result of the reduction. Each program instance writes the partial result into a global tensor at the index equal to its program instance index. The second kernel is not blocked in the reduction dimension and a single program instance computes the result of the reduction using the partial results computed by the first kernel. Due to the overhead of the additional kernel launch, this strategy is only beneficial for large reduction dimensions. Listing 3.20 and Listing 3.21 show an example using this strategy to block a reduction dimension. Lines

Listing 3.18: The Python file generated by the Decoupled Triton compiler after compiling the kernel definition in Listing 3.17.

```
import torch
import triton
import triton.language as tl

@triton.jit
def sum_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_k_stride: tl.constexpr,
    sum_out_ptr,
    sum_out_x_stride: tl.constexpr,
    x_SIZE: tl.constexpr, k_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    k_accumulator = 0.000000
    for k_iter in range(0, k_SIZE, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride + k_iter * A_k_stride)
      k_accumulator = k_accumulator + A
    k_reduction_result = k_accumulator
    sum_out = k_reduction_result
    tl.store(sum_out_ptr + x_iter * sum_out_x_stride, sum_out)

def sum_out(A, x, k):
  A_x_stride, A_k_stride, = A.stride()
  sum_out = torch.empty(x, dtype=torch.float32, device='cuda')
  sum_out_x_stride, = sum_out.stride()
  sum_out_grid = (1,)
  sum_out_kernel[sum_out_grid](
    A,
    A_x_stride, A_k_stride,
    sum_out,
    sum_out_x_stride,
    x, k,
    num_stages=3, num_warps=4)

  return sum_out
```

Listing 3.19: The Python file generated by the Decoupled Triton compiler after adding the scheduling primitive `sum_out.tensorize(k:16)` to the kernel definition in Listing 3.17.

```
import torch
import triton
import triton.language as tl

@triton.jit
def sum_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_k_stride: tl.constexpr,
    sum_out_ptr,
    sum_out_x_stride: tl.constexpr,
    x_SIZE: tl.constexpr, k_SIZE: tl.constexpr):
  k_arange = tl.arange(0, 16)
  for x_iter in range(0, x_SIZE, 1):
    k_accumulator = tl.zeros((16,), tl.float32)
    for k_iter in range(0, k_SIZE, 16):
      A = tl.load(A_ptr + x_iter * A_x_stride +
                  (k_iter + k_arange) * A_k_stride)
      k_accumulator = k_accumulator + A
    k_reduction_result = tl.sum(k_accumulator, 0)
    sum_out = k_reduction_result
    tl.store(sum_out_ptr + x_iter * sum_out_x_stride, sum_out)

def sum_out(A, x, k):
  A_x_stride, A_k_stride, = A.stride()
  sum_out = torch.empty(x, dtype=torch.float32, device='cuda')
  sum_out_x_stride, = sum_out.stride()
  sum_out_grid = (1,)
  sum_out_kernel[sum_out_grid](
    A,
    A_x_stride, A_k_stride,
    sum_out,
    sum_out_x_stride,
    x, k,
    num_stages=3, num_warps=4)

  return sum_out
```

1–19 and 21–39 in Listing 3.20 define the first and second kernels, respectively, of the reduction. The first kernel stores its partial result to the global tensor on lines 18–19. The second kernel loads the partial results on lines 33–35 and computes the result of the reduction in the reduction loop from lines 32–36. Lines 7–14 in Listing 3.21 launch the first kernel using a grid with multiple program instances. Lines 20–26 launch the second kernel with a single program instance.

Another strategy to block a reduction dimension is possible if the reduction we are blocking is the last operation in the kernel. In this case, each program instance can use atomic instructions to update the global output tensor with its partial result. Data races in the update are not an issue because the updates are atomic. This strategy is not implemented in the current Decoupled Triton compiler.

3.4 Limitations

A limitation of the current Decoupled Triton design is that its scheduling primitives transform iteration spaces through tiling, blocking, and mapping the computation; however, some high-performance ML operator schedules rely on stateful scheduling strategies that coordinate computation across iterations using auxiliary data structures. These strategies are not limited to transforming the iteration space; they also require mechanisms to express multi-pass dependencies, maintain running statistics, and perform reductions that are updated online while partial outputs are generated. An example schedule that relies on these scheduling strategies is FlashAttention [10, 11, 35]. FlashAttention is a fast and memory-efficient exact implementation of attention that utilizes a fused online softmax [22] implementation to compute the softmax operation. Online softmax maintains the maximum and normalization factors as it iterates over input blocks, incrementally producing partial outputs. After processing the entire sequence, the outputs are updated using the accumulated state to obtain

Listing 3.20: The Triton kernels generated by the Decoupled Triton compiler after adding the scheduling primitive `sum_out.block(k:32)` to the kernel definition in Listing 3.17. Listing 3.21 contains the corresponding wrapper function.

```
@triton.jit
def split_k_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_k_stride: tl.constexpr,
    split_k_ptr,
    split_k_x_stride: tl.constexpr, split_k_k_stride: tl.constexpr,
    x_SIZE: tl.constexpr, k_SIZE: tl.constexpr):
  k_BLOCK_COUNT = k_SIZE // 32
  k_pid = tl.program_id(0).to(tl.int64) % k_BLOCK_COUNT
  k_block_start = k_pid * 32
  for x_iter in range(0, x_SIZE, 1):
    k_accumulator = 0.000000
    for k_iter in range(0, 32, 1):
      A = tl.load(A_ptr + x_iter * A_x_stride +
                  (k_block_start + k_iter) * A_k_stride)
      k_accumulator = k_accumulator + A
    k_reduction_result = k_accumulator
    tl.store(split_k_ptr + x_iter * split_k_x_stride +
             k_pid * split_k_k_stride, k_reduction_result)

@triton.jit
def sum_out_kernel(
    A_ptr,
    A_x_stride: tl.constexpr, A_k_stride: tl.constexpr,
    split_k_ptr,
    split_k_x_stride: tl.constexpr, split_k_k_stride: tl.constexpr,
    sum_out_ptr,
    sum_out_x_stride: tl.constexpr,
    x_SIZE: tl.constexpr, k_SIZE: tl.constexpr):
  for x_iter in range(0, x_SIZE, 1):
    k_accumulator = 0.000000
    for k_iter in range(0, k_SIZE // 32, 1):
      k_reduction_intermediate = tl.load(split_k_ptr + x_iter *
                                         split_k_x_stride + k_iter *
                                         split_k_k_stride)
      k_accumulator = k_accumulator + k_reduction_intermediate
    k_reduction_result = k_accumulator
    sum_out = k_reduction_result
    tl.store(sum_out_ptr + x_iter * sum_out_x_stride, sum_out)
```

Listing 3.21: The wrapper function generated by the Decoupled Triton compiler after adding the scheduling primitive `sum_out.block(k:32)` to the kernel definition in Listing 3.17. Listing 3.20 contains the corresponding Triton kernels.

```
def sum_out(A, x, k):
  A_x_stride, A_k_stride, = A.stride()
  sum_out_split_k = torch.empty(
    x, k // 32,
    dtype=torch.float32, device='cuda')
  split_k_x_stride, split_k_k_stride, = split_k.stride()
  split_k_grid = (triton.cdiv(k, 32)),
  split_k_kernel[split_k_grid](
    A,
    A_x_stride, A_k_stride,
    split_k,
    split_k_x_stride, split_k_k_stride,
    x, k,
    num_stages=3, num_warps=4)

  A_x_stride, A_k_stride, = A.stride()
  split_k_x_stride, split_k_k_stride, = split_k.stride()
  sum_out = torch.empty(x, dtype=torch.float32, device='cuda')
  sum_out_x_stride, = sum_out.stride()
  sum_out_grid = (1,)
  sum_out_kernel[sum_out_grid](
    A,
    A_x_stride, A_k_stride,
    split_k,
    split_k_x_stride, split_k_k_stride,
    sum_out, sum_out_x_stride, x, k, num_stages=3, num_warps=4)

  return sum_out
```

the correct result. The scheduling primitives that are currently available in the Decoupled Triton DSL cannot transform the schedule in a way that generates the online softmax schedule. However, Decoupled Triton can represent the attention algorithm with different, non-FlashAttention, schedules. Subsection 4.4.4 describes an experimental evaluation on attention kernels defined in Decoupled Triton.

3.5 Implementation

We created a prototype implementation of the Decoupled Triton DSL and compiler, `dtc`, for the experimental evaluation presented in Chapter 4. To define the grammar and generate both the lexer and parser code for the DSL, we used ANTLR4 [28]. The rest of `dtc` is defined in just over 10,000 lines of C++20 code.

For each kernel in the input file, `dtc` initially compiles the algorithm to a high-level IR, then applies the relevant scheduling primitives in a series of compiler passes to transform the IR. The order and number of passes in the compiler is fixed. Some passes may not transform the IR depending on the provided schedule.

For rapid testing, `dtc` can optionally generate a *test* function in addition to the kernel and wrapper function. The *test* function takes no arguments. When called, the *test* function generates random inputs, with dimension sizes determined by the user, and calls the wrapper function.

To perform the experimental evaluation, we also developed an experimental workflow and set of tools. The experimental workflow is discussed in Section 4.2. To simplify the creation of a schedule space, we developed a tool in Python that takes a schedule template as an input and produces all of the schedules in the space defined by the template as output. To simplify this tool and the schedule template representation, we created a shorthand for Decoupled Triton schedules. We also created a Python script for running the experiments described in Section 4.2.

Chapter 4

Evaluation

Can Decoupled Triton be used to define efficient ML kernels? How easy is it to find efficient schedules in Decoupled Triton? This chapter aims to provide answers to these questions and to present an experimental methodology for evaluating Decoupled Triton on ML kernels. The results of the evaluation indicate that Decoupled Triton can be used to define efficient ML kernels, which can easily be found through the rapid schedule exploration enabled by Decoupled Triton.

4.1 Research questions

This chapter describes an experimental evaluation that aims to answer the following research questions:

1. Can Decoupled Triton be used to represent and generate Triton kernels that achieve performance equivalent to, or better than, expert-written Triton kernels and Triton kernels generated by a mature ML programming framework?

2. If the answer to research question 1 is yes, can developers reasonably find the efficient schedules?

3. For which ML kernels do expert-written Triton kernels and Triton kernels generated by a mature ML programming framework fail to achieve good performance?

4.2 Experimental setup and methodology

Table 4.1 and Table 4.2 provide the system specifications and the software versions, respectively, used in the experimental evaluation.

Table 4.1: Hardware specifications.

CPU	Intel® Core™ i7-14700K
GPU	NVIDIA RTX 5000 Ada Generation
Memory	64 GiB DDR5

Table 4.2: Software versions.

CUDA	12.4
Liger Kernel	0.5.8
PyTorch	2.6.0
Triton	3.2.0

Our focus is on schedule exploration and finding efficient schedules. Figure 4.1 shows the experimental setup for evaluating Decoupled Triton on an ML Kernel. In an experiment, both the algorithm and dimension sizes are fixed while we evaluate the performance of each schedule in a pre-defined schedule space.

Determining a schedule space for a given ML kernel and dimension size configuration starts with a small manual exploration of schedules. After observing the performance of a few different schedules, we define a schedule space that we expect to contain good performance schedules. This process simulates the pattern of a developer using Decoupled Triton to optimize an ML kernel.

We measure kernel run time to compare the performance of each schedule against the baseline functions and the other schedules. To measure run time, we use the Triton benchmarking function, `triton.testing.do_bench`, which measures run time

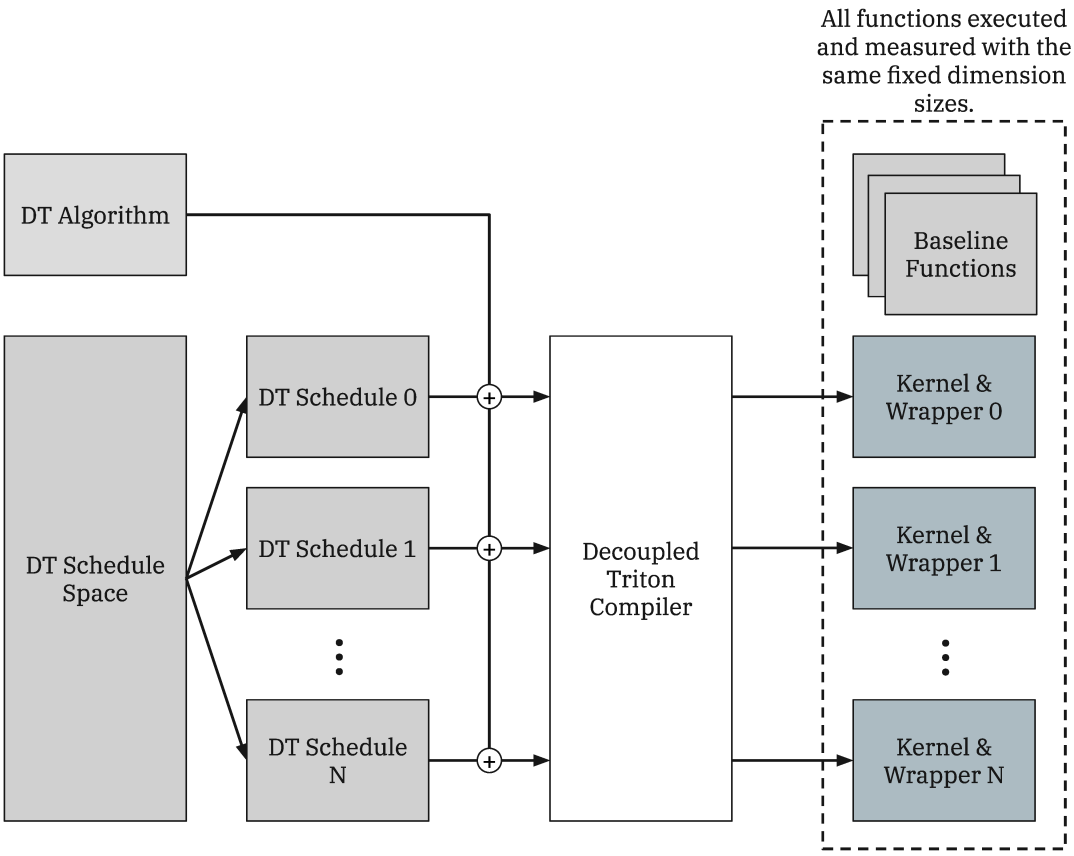


Figure 4.1: Overview of the experimental methodology for a single ML Kernel.

by first calling the function repeatedly for 25 milliseconds to warm up, and then calling the function repeatedly for 100 milliseconds, while recording run times, and finally returning the median recorded run time.[1] All reported run times are the arithmetic average of 5 runs of the experiment. Within a single run of an experiment, the order of functions evaluated — includes both Triton kernels generated by Decoupled Triton and baseline functions — is randomized. All kernels in an evaluation are tested for correctness against the baseline functions using `torch.allclose`.[2]

The comparison is between the kernels generated by Decoupled Triton against expert-written kernels and Triton kernels generated by a mature ML programming framework. Liger Kernels is a collection of efficient Triton kernels, which implement ML operations [16]. The Liger team selected kernels that are performance-critical transformer operations with high memory or latency costs and high reusability across Large Language Models (LLMs). Table 4.3 lists the different baselines used in the experiments in this evaluation. Not all experiments contain each baseline. In particular, for a given ML kernel, the `liger-kernel` or `triton-kernel` baselines are only included if there is an implementation of the ML kernel in the Liger Kernels [16] collection or we found another Triton implementation of the ML kernel.

This comparative study includes any ML kernel that produces a tensor as the output — no other criteria is used to select ML kernels to evaluate.

4.3 How to read the graphs

All of the graphs presented are run-time graphs. Therefore, a lower value on the y-axis signifies better performance. Baseline function run times are displayed as horizontal lines in the graphs. The generated DT kernels from the schedule space are sorted by

[1] https://triton-lang.org/main/python-api/generated/triton.testing.do_bench.html
[2] https://docs.pytorch.org/docs/stable/generated/torch.allclose.html
[3] https://docs.pytorch.org/docs/stable/generated/torch.compile.html

Table 4.3: Baselines.

`torch.compile`	A kernel implemented in PyTorch and annotated with the decorator `@torch.compile()` to instruct PyTorch to JIT compile the function using its default backend TorchInductor.[3]
`torch.compile(max-autotune)`	Same as torch.compile, but sets the keyword argument, `mode="max-autotune"`.
`liger kernel`	An expert-written Triton kernel taken from Liger Kernels [16], an open-source set of efficient Triton kernels developed specifically for LLM training that utilizes kernel optimization techniques such as operation fusing.
`triton kernel`	An expert-written Triton kernel taken from another repository — specified in the text.

increasing run time, and their run times are displayed as vertical bars. The x-axis denotes the number of schedules in the schedule space explored by the experiment.

4.4 Results

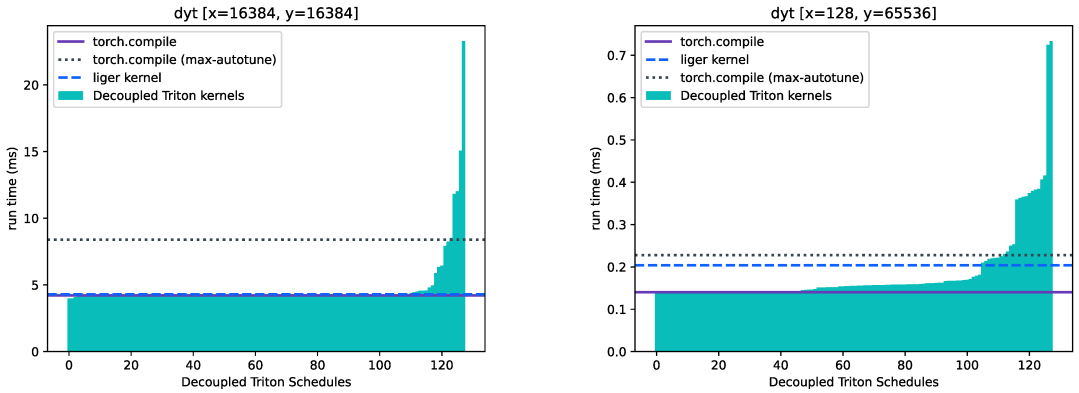
This section presents the results of the experimental evaluation of the Decoupled Triton system.

4.4.1 Element-wise kernels

Element-wise kernels such as DyT, GeGLU, and SwiGLU are fundamental operations in ML. Commonly used as activation functions, element-wise operations are generally easy to schedule due to their simplicity: a given schedule, generally, has similar performance across different element-wise functions.

Figure 4.2, Figure 4.3, and Figure 4.4 show the run time graphs from the perfor-

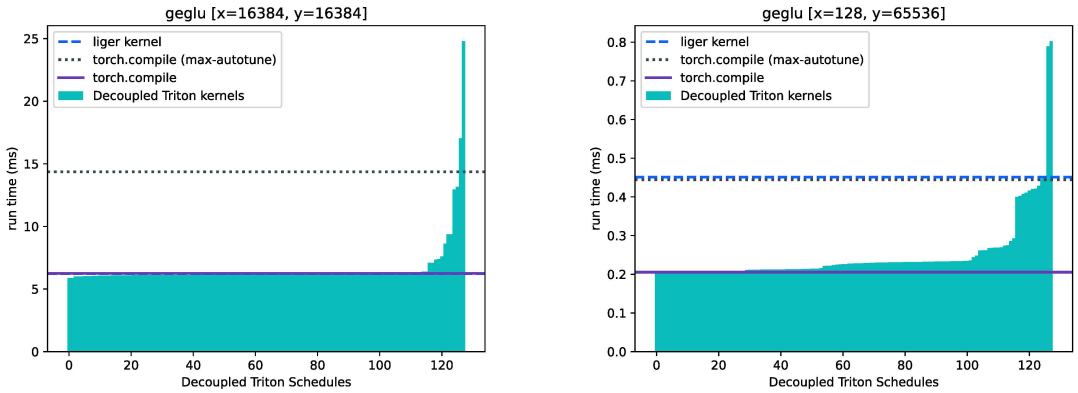
mance evaluations for DyT, GeGLU, and SwiGLU, respectively. Listing 4.1 shows the fastest kernel definition for Figure 4.3a. For these ML kernels, Decoupled Triton can compete with both the expert-written Liger kernels [16] and the kernels generated by PyTorch [4]. PyTorch with `max-autotune` enabled performs poorly for nearly all of the ML kernels we tested. Our experience with PyTorch during this evaluation indicates that the kernel templates used by PyTorch with `max-autotune` enabled are poorly optimized for most operations.



(a) `x=16384, y=16384` (b) `x=128, y=65536`

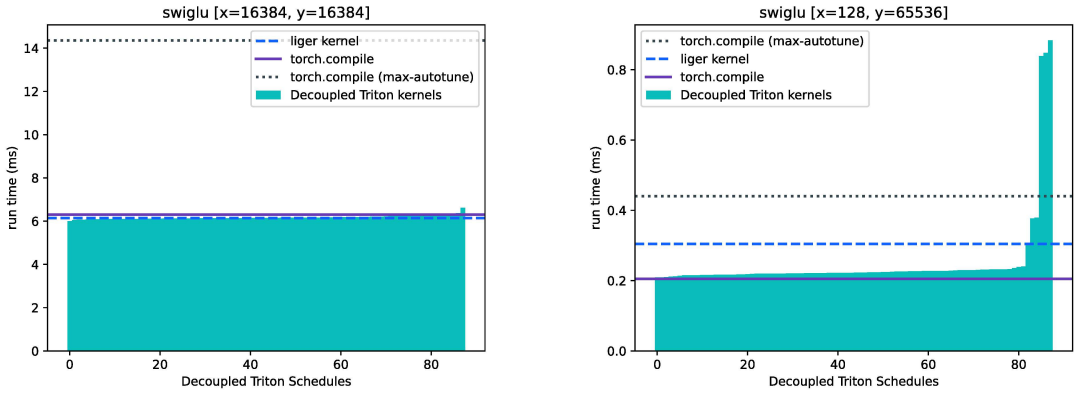
Figure 4.2: Experimental evaluation of the DyT operation in Decoupled Triton. Lower is better.

The Liger-Kernel baseline Triton kernels for these operations have a significant limitation — they never block in the innermost dimension. Thus, for Figure 4.2b, Figure 4.3b, and Figure 4.4b, when the innermost dimension is large, the Liger kernels experience a significant slowdown. Furthermore, these Liger kernels report an error if the input has an innermost dimension size greater than 65536. Although it may be uncommon to perform these operations on tensors with very large innermost dimensions, these results show that expert-written kernels may not have the most efficient schedule for each use case.



(a) x=16384, y=16384 (b) x=128, y=65536

Figure 4.3: Experimental evaluation of the GeGLU operation in Decoupled Triton. Lower is better.



(a) x=16384, y=16384 (b) x=128, y=65536

Figure 4.4: Experimental evaluation of the SwiGLU operation in Decoupled Triton. Lower is better.

Listing 4.1: The most efficient Decoupled Triton kernel definition evaluated in the experiment shown in Figure 4.3a.

```
Func geglu;
In   A, B;
Var  x, y;

geglu[x, y] = 0.5 * A[x, y] * (1 + tanh(0.7978845608028654 *
              (A[x, y] + 0.044715 * pow(A[x, y], 3)))) * B[x, y];

geglu.block(x:1);
geglu.tensorize(x:0);
geglu.block(y:512);
geglu.tensorize(y:0);
geglu.map(x, y);
geglu.num_warps(32);
geglu.compile();
```

4.4.2 Reduction kernels

Compared to element-wise ML kernels, ML kernels that contain reduction tensor operations such as TVD, KL Divergence, and Root Mean Squared Normalization can be more difficult to schedule (Subsection 3.3.2).

Figure 4.5, Figure 4.6, and Figure 4.7 show the run time graphs from the performance evaluations for TVD, KL Divergence, and Root Mean Squared Normalization, respectively. The exploration of the schedule space with Decoupled Triton finds Decoupled Triton kernels that can compete with both the expert-written Liger kernels [16] and the kernels generated by PyTorch [4].

4.4.2.1 Softmax

Softmax is an ML operation that performs a form of normalization. In particular, softmax converts values to probabilities. Figure 4.8 and Figure 4.9 show the run time graphs from the performance evaluations for the softmax operation in Decoupled Triton. Listing 4.2 shows the fastest kernel definition for Figure 4.9c. The baseline

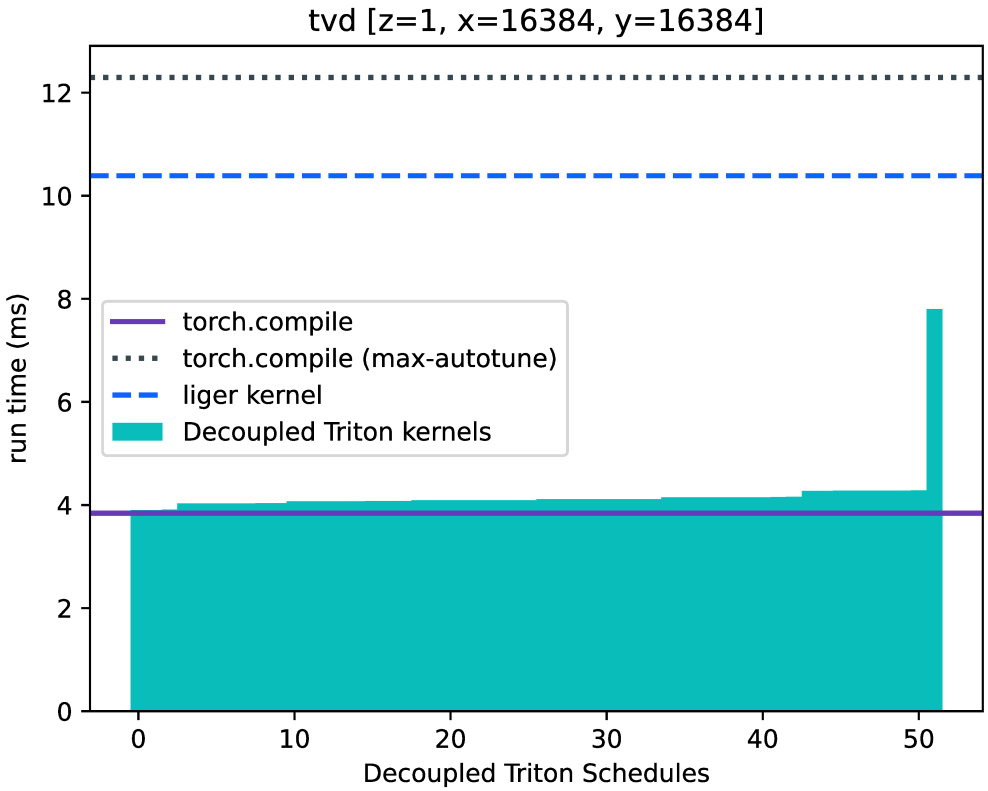


Figure 4.5: Experimental evaluation of the TVD operation in Decoupled Triton. Lower is better.

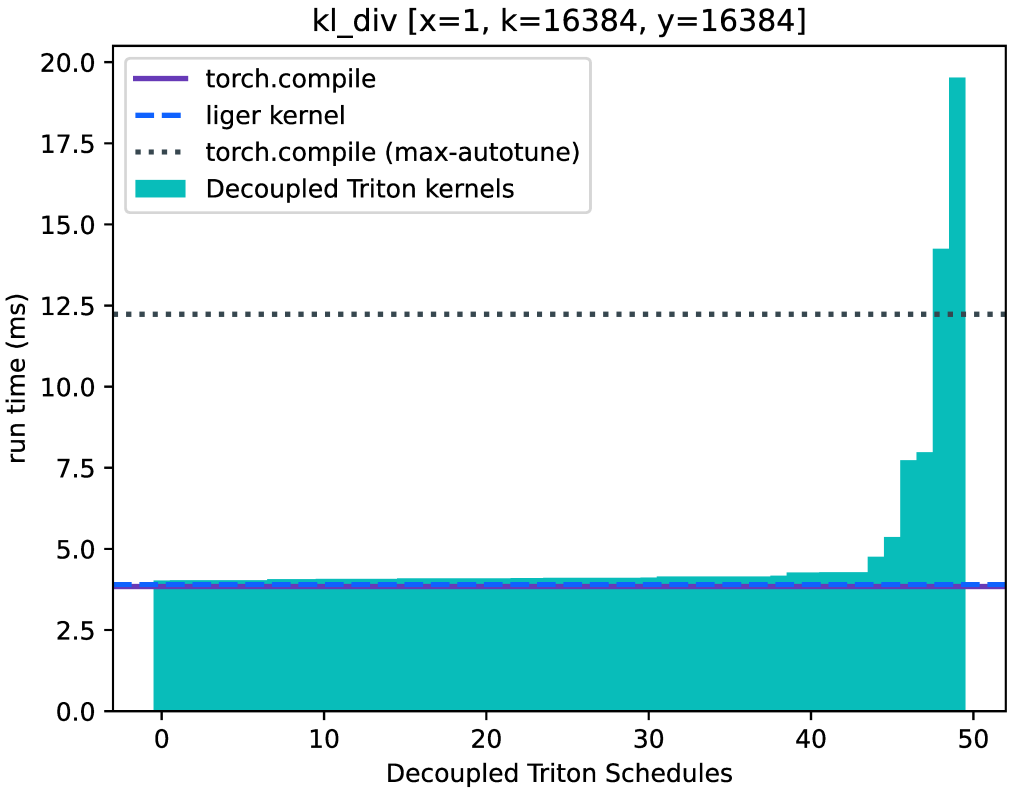


Figure 4.6: Experimental evaluation of the KL Divergence operation in Decoupled Triton. Lower is better.

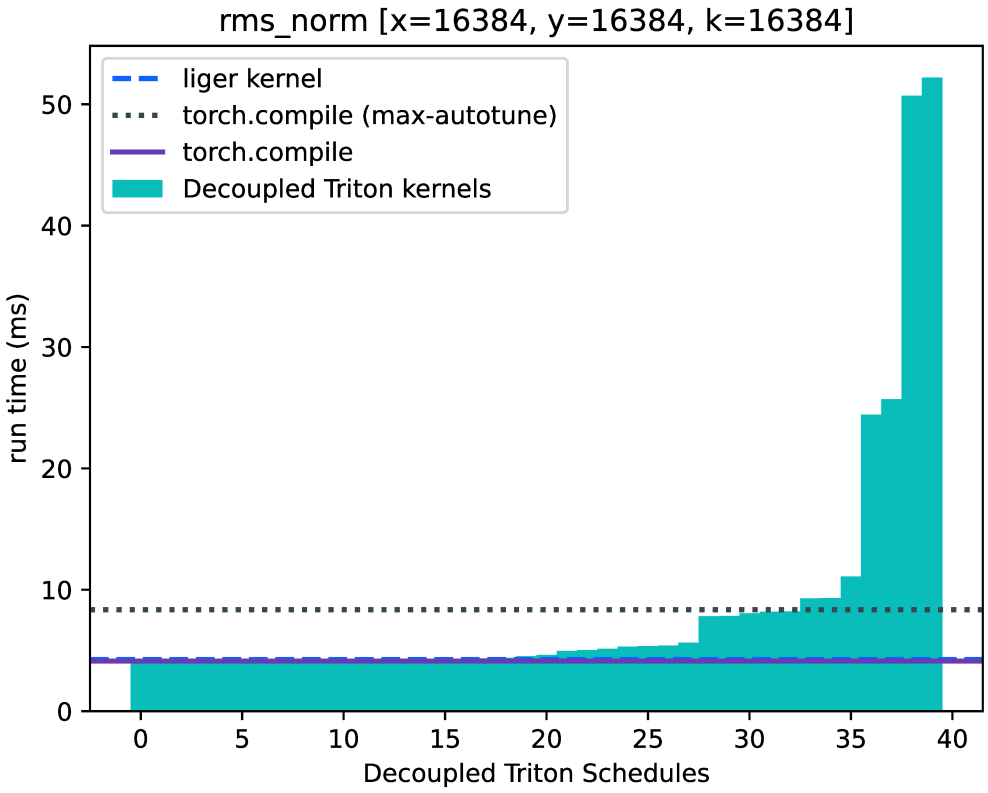
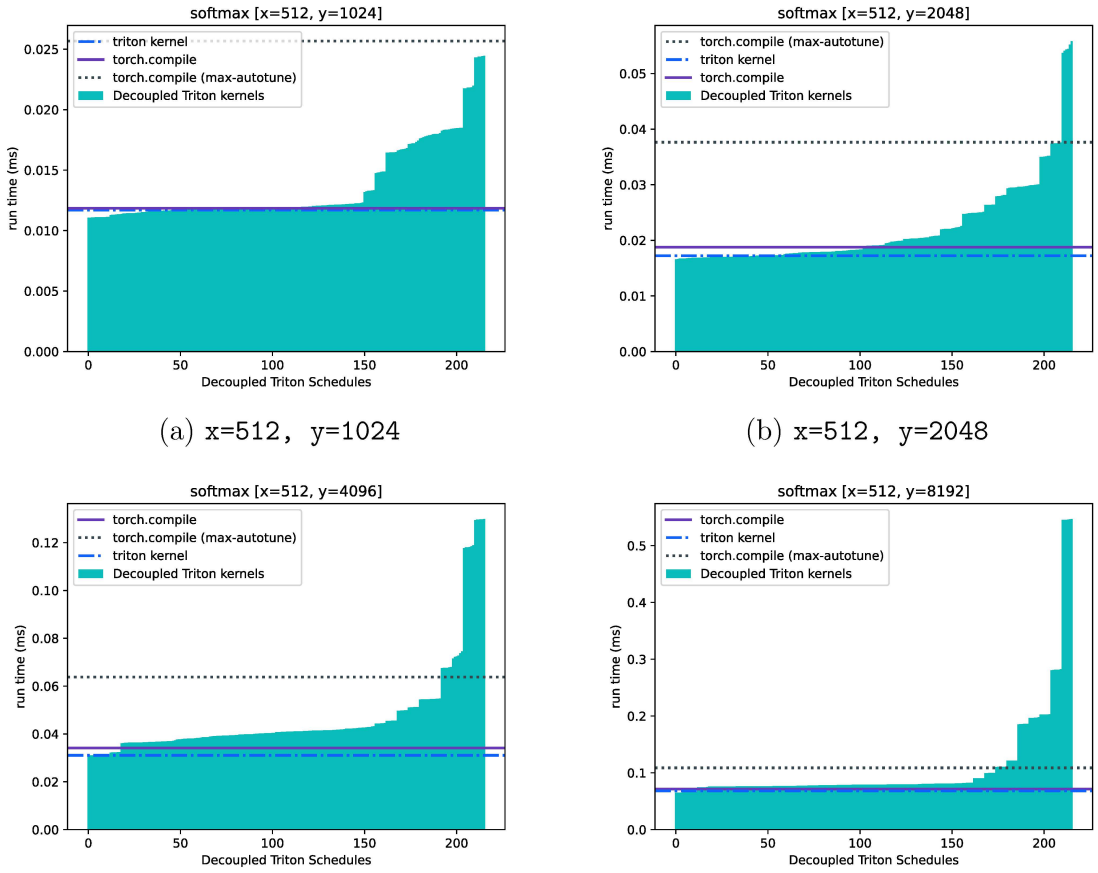


Figure 4.7: Experimental evaluation of the Root Mean Squared Normalization operation in Decoupled Triton. Lower is better.

expert-written Triton kernel is from the Triton tutorial.[4] The schedule space exploration finds Decoupled Triton kernels that can compete with both the expert-written Triton kernel and PyTorch. Furthermore, in many of the experiments depicted in Figure 4.9, this exploration finds Decoupled Triton kernels that are significantly faster than all baselines.



(a) x=512, y=1024

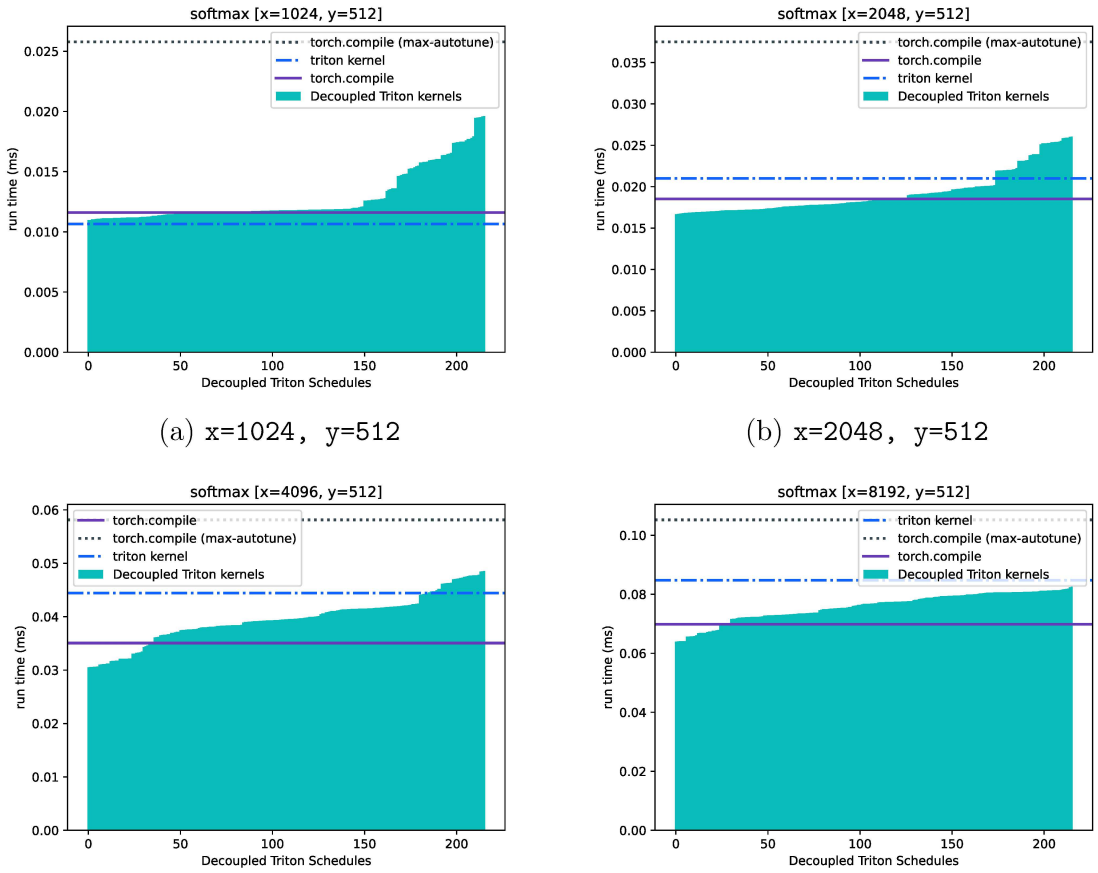
(b) x=512, y=2048

(c) x=512, y=4096

(d) x=512, y=8192

Figure 4.8: Experimental evaluation of softmax in Decoupled Triton. The size of the x dimension is constant. Lower is better.

[4]https://triton-lang.org/main/getting-started/tutorials/02-fused-softmax.html



(a) x=1024, y=512

(b) x=2048, y=512

(c) x=4096, y=512

(d) x=8192, y=512

Figure 4.9: Experimental evaluation of softmax in Decoupled Triton. The size of the y dimension is constant. Lower is better.

Listing 4.2: The most efficient Decoupled Triton kernel definition evaluated in the experiment shown in Figure 4.9c.

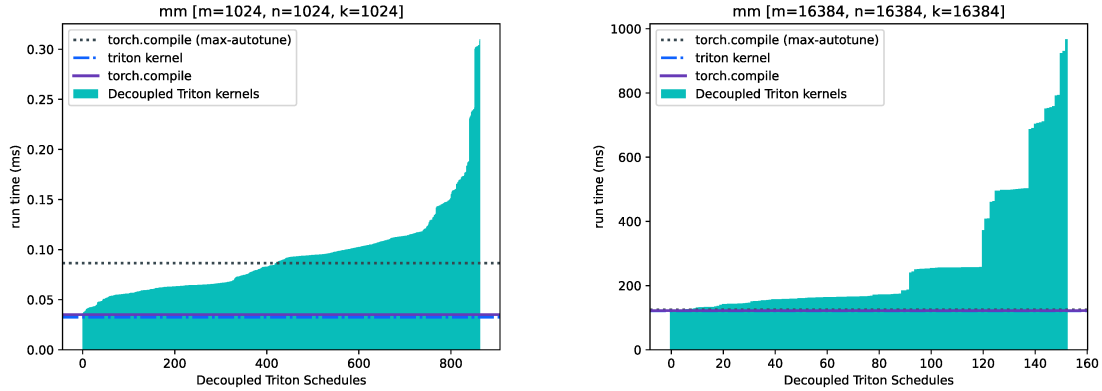
```
Func _softmax, _sum;
In   A;
Var  x;
RVar y;

_sum[x]        = rsum(exp(A[x, y]), y);
_softmax[x, y] = exp(A[x, y]) / reshape(_sum[x], x, 1);

_softmax.block(x:4);
_softmax.tensorize(y:512);
_softmax.tensorize(x:0);
_softmax.num_warps(32);
_sum.fuse_at(_softmax, x);
_softmax.compile_to_kernel();
```

4.4.3 Matrix product

Matrix product, or matrix multiplication, is an extremely common operation in ML models. Matrix product schedule optimization has received a lot of research attention in the past, and vendor libraries offer high-performance kernels to efficiently compute matrix products using these schedule optimizations. By default, PyTorch uses high-performance vendor libraries to compute matrix product instead of generating a Triton kernel. The schedule exploration with Decoupled Triton finds schedules that compete with these highly optimized baselines, demonstrating the strength of Decoupled Triton's scheduling API. The expert-written Triton kernel baseline is taken from the Triton tutorial.[5]

Figure 4.10 and Figure 4.11 show the run time graphs from the performance evaluations for the matrix product operation in Decoupled Triton. We find a Decoupled Triton kernel that competes with all baselines in each experiment.

[5]https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html



(a) `m=1024, n=1024, k=1024` (b) `m=16384, n=16384, k=16384`

Figure 4.10: Experimental evaluation of matrix product with uniform dimension sizes in Decoupled Triton. Lower is better.

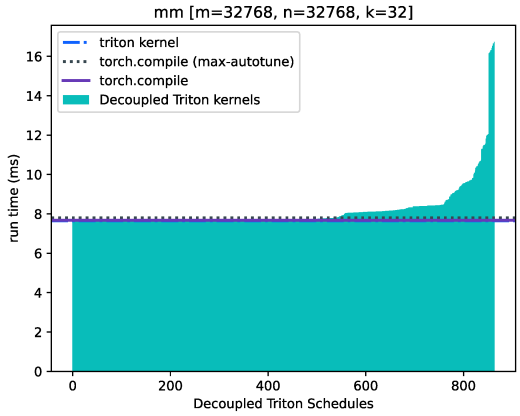
4.4.3.1 Program instance mapping

The program mapping scheduling directive described in Subsection 3.3.1 can have a significant impact on the efficiency of a matrix product schedule because of the data-reuse qualities inherent to the matrix product computation.[6] Table 4.4 shows the run-time results for an experiment that demonstrates the importance of program-instance mapping. Despite only changing the mapping from program instance to output block, the second schedule is more than twice as fast as the first schedule.

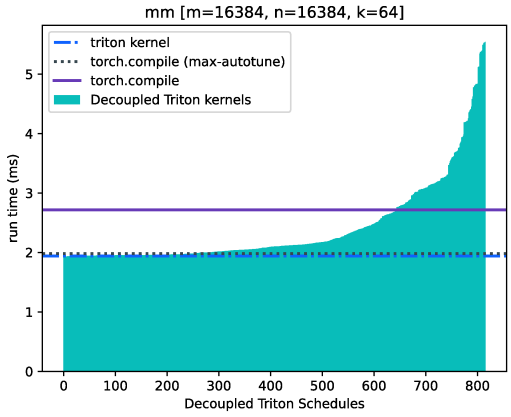
Table 4.4: Run times (ms) for matrix product kernels with different `map` scheduling but otherwise identical schedules. The dimension sizes are `x=16384 y=16384 k=16384`.

`map(x, y)`	254
`map(y:yi/8, x, yi)`	114

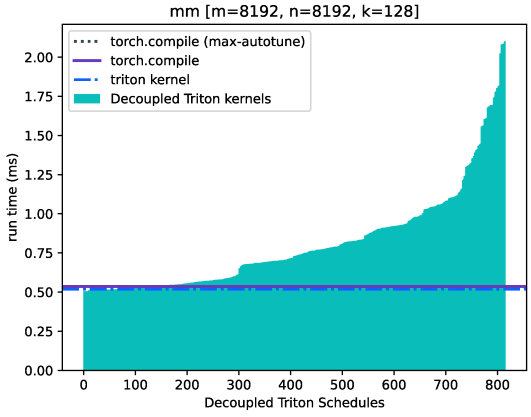
[6]https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html#l2-cache-optimizations



(a) `m=32768, n=32768, k=32`



(b) `m=16384, n=16384, k=64`



(c) `m=8192, n=8192, k=128`

Figure 4.11: Experimental evaluation of matrix product with a small inner dimension and large outer dimensions in Decoupled Triton. Lower is better.

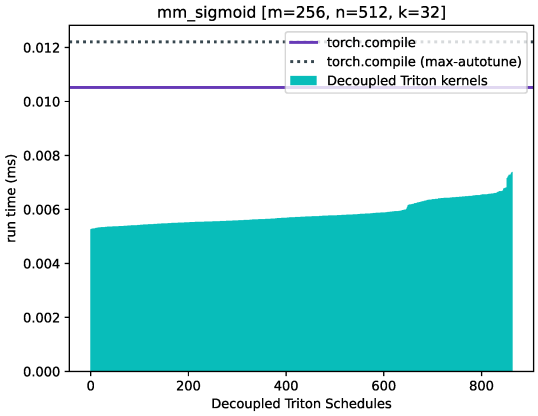
4.4.3.2 Element-wise fusion

A common pattern in ML models is to perform a matrix product, then apply an element-wise operation, such as an activation function, to the result of the product. This experimental evaluation uses the `mmsigmoid` operation, defined as the sigmoid function applied to the result of a matrix product between two matrices. Figure 4.12 shows the run-time graphs from the performance evaluations for the `mmsigmoid` operation in Decoupled Triton.

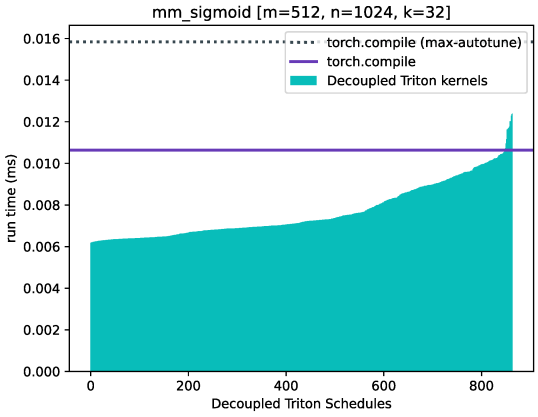
The Decoupled Triton kernels significantly outperform PyTorch because PyTorch is unable to fuse the sigmoid operation into the high-performance library kernel that it calls to compute the matrix product. Therefore, PyTorch launches two kernels, the first computes the matrix product using a high-performance library kernel, and the second applies element-wise sigmoid to the result. Decoupled Triton can generate efficient schedules by using scheduling primitives to instruct the compiler to fuse the matrix product and the sigmoid operation into a single kernel. The `max-autotune` PyTorch mode should excel for this type of computation. However, while it does generate a single fused Triton kernel for the `mmsigmoid` operation, the generated kernel is less efficient than the default PyTorch behavior.

4.4.3.3 2mm

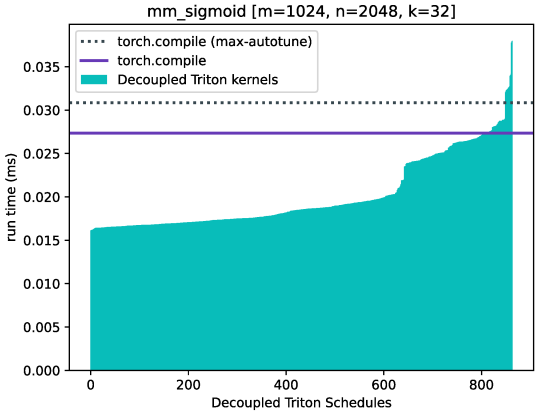
2mm is another common ML operation that consists of two matrix products. The first matrix product is computed between two inputs, then the second matrix product is computed between the result of the first product and a third input. There are two strategies to fuse the two matrix products into a single kernel: an inner-loop fusion and an outer-loop fusion. Figure 4.13 and Figure 4.14 show the run time graphs for the performance evaluation of 2mm in Decoupled Triton using an inner- and outer-loop fusion, respectively. Listing 4.3 shows the fastest kernel definition for Figure 4.14a.



(a) `m=256, n=512, k=32`

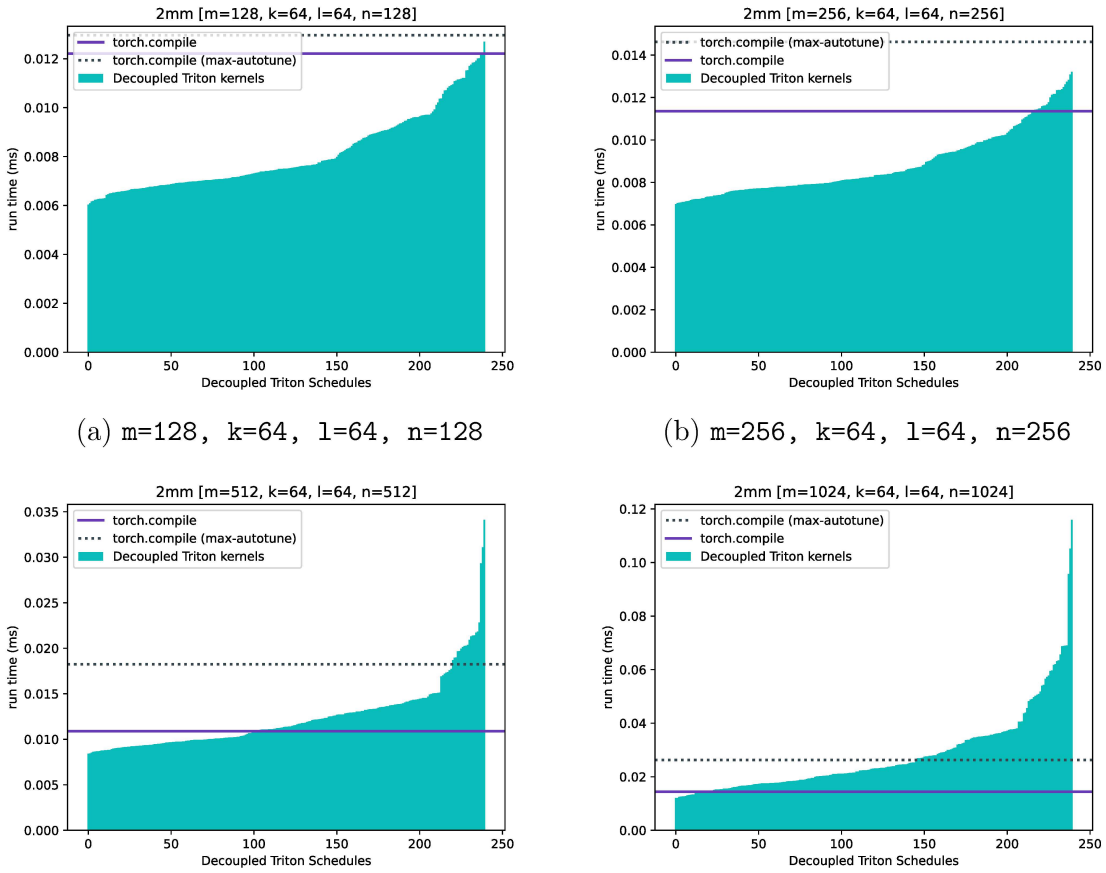


(b) `m=512, n=1024, k=32`



(c) `m=1024, n=2048, k=32`

Figure 4.12: Experimental evaluation of `mmsigmoid` in Decoupled Triton. Lower is better.



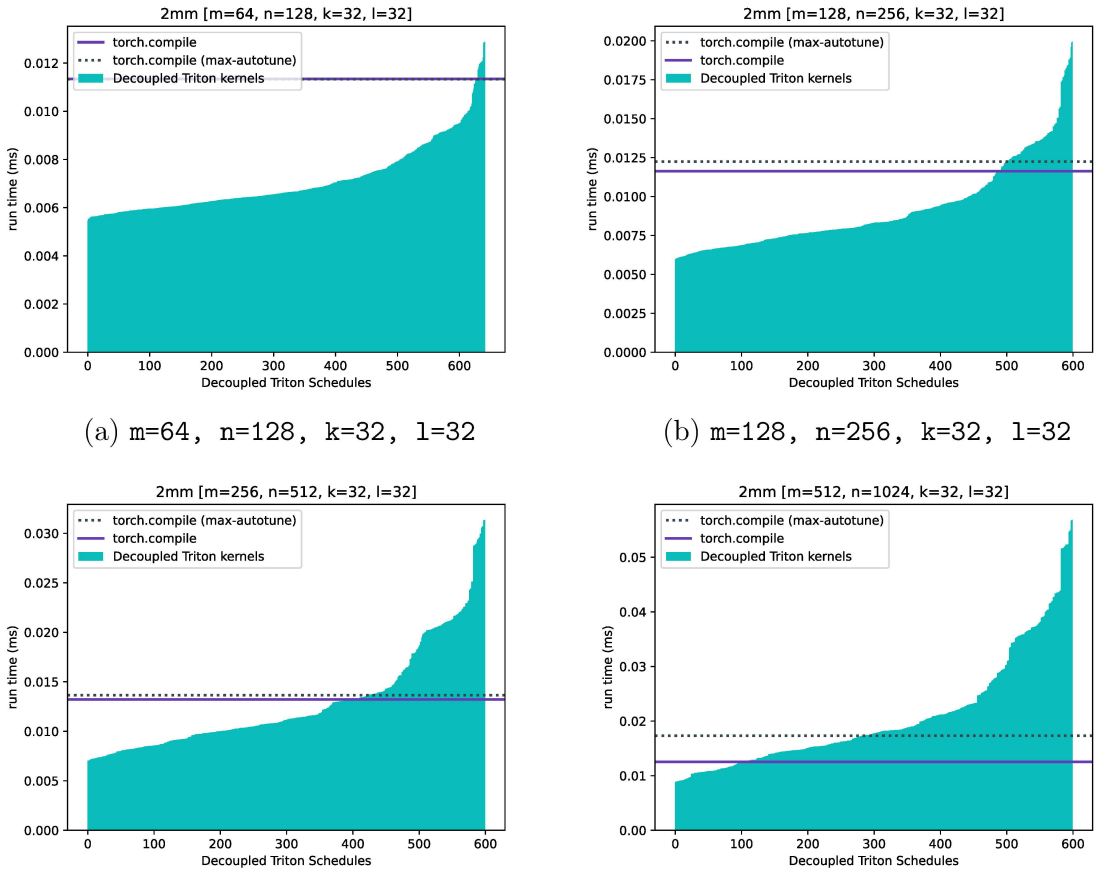
(a) `m=128, k=64, l=64, n=128`

(b) `m=256, k=64, l=64, n=256`

(c) `m=512, k=64, l=64, n=512`

(d) `m=1024, k=64, l=64, n=1024`

Figure 4.13: Experimental evaluation of 2mm in Decoupled Triton. These schedule spaces explore inner-loop fusions. Lower is better.



(a) `m=64, n=128, k=32, l=32`

(b) `m=128, n=256, k=32, l=32`

(c) `m=256, n=512, k=32, l=32`

(d) `m=512, n=1024, k=32, l=32`

Figure 4.14: Experimental evaluation of 2mm in Decoupled Triton. These schedule spaces explore outer-loop fusions. Lower is better.

Listing 4.3: The most efficient Decoupled Triton kernel definition evaluated in the experiment shown in Figure 4.14a.

```
Func _2mm, mm;
In   A, B, C;
Var  m, n;
RVar k, l;

mm[m, l]   = rdot(A[m, k], B[k, l], k);
_2mm[m, n] = rdot(mm[m, l], C[l, n], l);

_2mm.block(m:16);
_2mm.tensorize(m:16);
_2mm.tensorize(n:64);
_2mm.tensorize(k:32);
_2mm.tensorize(l:0);
_2mm.num_stages(4);
_2mm.num_warps(4);
mm.fuse_at(_2mm, m);
_2mm.compile_to_kernel();
```

The Decoupled Triton kernels significantly outperform PyTorch because, even with `max-autotune` enabled, PyTorch does not fuse the two matrix product operations. Thus, both PyTorch baselines must pay the overhead of an additional kernel launch. Combined with the results for `mmsigmoid`, these results demonstrate that PyTorch struggles with matrix product kernel fusions.

4.4.4 Attention

Attention is an ML operation used in transformer models [39]. The operation consists of a matrix product, followed by a softmax operation, and finally another matrix product. Figure 4.15 shows the run-time graphs for the performance evaluation of attention in Decoupled Triton. Listing 4.4 shows the fastest kernel definition for Figure 4.15a. The baseline function is PyTorch's `scaled_dot_product_attention`.[7]

[7]https://docs.pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html

This evaluation only examines small attention kernels because Decoupled Triton cannot represent FlashAttention (Section 3.4) and thus would perform poorly for larger attention kernels. Despite this limitation, the results are impressive. Schedules discovered with Decoupled Triton outperform PyTorch for small attention kernels.

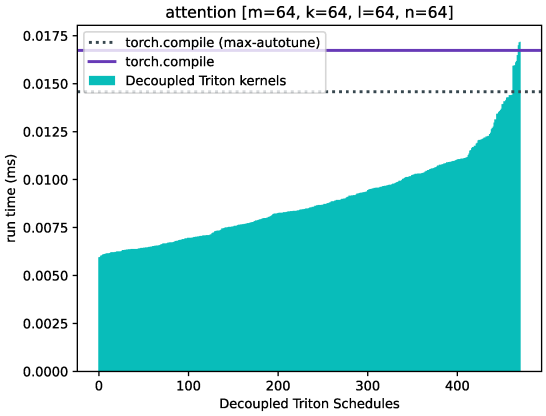
Listing 4.4: The most efficient Decoupled Triton kernel definition evaluated in the experiment shown in Figure 4.15a.

```
Func attention, mm, e, sm, dvsr;
In   A, B, C;
Var  m, n;
RVar k, l;

mm[m, l]        = rdot(A[m, k], B[k, l], k) / sqrt(len(l));
e[m, l]         = exp(mm[m, l]);
dvsr[m]         = rsum(e[m, l], l);
sm[m, l]        = e[m, l] / reshape(dvsr[m], m, 1);
attention[m, n] = rdot(sm[m, l], C[l, n], l);

attention.tensorize(m:16);
attention.block(m:16);
attention.tensorize(n:64);
attention.tensorize(k:16);
attention.tensorize(l:0);
attention.num_stages(8);
attention.num_warps(8);
mm.fuse_at(e, l);
dvsr.fuse_at(sm, m);
sm.fuse_at(attention, m);
e.fuse_at(attention, m);
attention.compile();
```

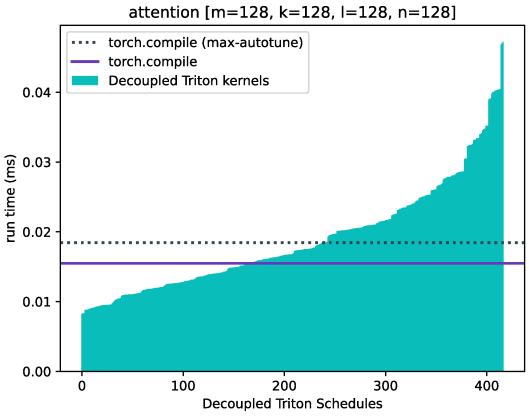
4.5 Discussion

The results of the experimental evaluation lead to the following answers to the research questions posed in Section 4.1.

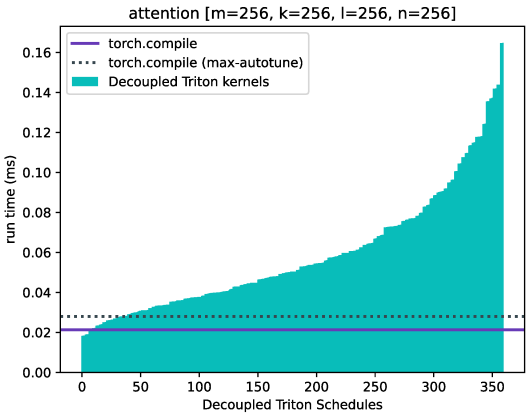
1. Yes, for all the ML kernels tested in the experimental evaluation, there are



(a) `m=64, k=64, l=64, n=64`



(b) `m=128, k=128, l=128, n=128`



(c) `m=256, k=256, l=256, n=256`

Figure 4.15: Experimental evaluation of attention in Decoupled Triton. Lower is better.

Decoupled Triton kernels that exhibit competitive performance compared to the baselines. These results indicate that the scheduling power exposed in the Decoupled Triton DSL is comparable to that of Triton itself. The only schedules that Decoupled Triton is unable to describe and generate are highly specialized ML kernel schedules such as FlashAttention [10] (Section 3.4).

2. Yes, for all of the ML kernels tested in the experimental evaluation, finding an efficient schedule in Decoupled Triton is a very reasonable task. Two numbers indicate how difficult it is to find an efficient schedule for an ML kernel: (1) the number of schedules in the explored schedule space and (2) the number of explored schedules with performance that is competitive with the most efficient baselines. The ratio of competitive schedules over the total number of explored schedules estimates how easy it is to find an efficient schedule. In general, about a quarter of the schedules explored are competitive with the baselines. Therefore, finding efficient schedules in Decoupled Triton is quite easy. Moreover, schedule exploration in Decoupled Triton is fast thanks to its modular schedule, and the tools described in Section 3.5. The experimental methodology presented in Section 4.2 describes the process for rapid schedule exploration.

3. Expert-written Triton kernels may fail to achieve good performance when unusual dimension sizes are used. In particular, when the innermost dimension is large, the following Liger kernels perform poorly or result in an error: DyT, GeGLU, and SwiGLU (Subsection 4.4.1). Also, PyTorch fails to generate efficient Triton kernels when it is presented with a kernel that benefits from fusing a matrix product with another operation. This limitation is evident in the experimental evaluations of `mmsigmoid` and 2mm (Subsection 4.4.3). Furthermore, PyTorch with `max-autotune` enabled generates Triton kernels that do not have

good performance. The only kernels for which PyTorch with `max-autotune` enabled generated an efficient kernel were simple matrix product kernels.

Chapter 5

Related Work

This chapter compares Decoupled Triton against other related research efforts. The chapter discusses works that extend the Triton language or compiler and notes that the existing efforts are orthogonal to Decoupled Triton. Next, the chapter describes systems that generate Triton kernels as output. There is no existing work that both generates Triton kernels and allows user-defined scheduling. Finally, the chapter compares Decoupled Triton against other decoupled ML kernel languages. Compared to these languages, Decoupled Triton sacrifices low-level scheduling control for intuitive and simple scheduling.

5.1 Extensions to Triton

Inspired by the industry-wide adoption of Triton for writing high-performance ML kernels, researchers have extended the Triton language and compiler to achieve different goals.

Wang *et al.* [40] proposed ML-Triton, an extension to Triton that introduces a multi-level compilation flow and a warp-level IR. ML-Triton introduces a Triton language extension, which exposes a warp-level API and a compiler hint, to give developers fine-grained control over the warp-level scheduling of their kernels. While ML-Triton empowers developers to define warp-level scheduling, its language exten-

sion maintains a tight coupling between the algorithm and the scheduling of the kernel. Therefore, ML-Triton's contributions are orthogonal to those of Decoupled Triton. In fact, Decoupled Triton could be expanded to allow warp-level scheduling by introducing warp-level scheduling primitives, generating Triton kernels with ML-Triton's language extensions, and utilizing the ML-Triton compilation flow when compiling generated Triton kernels.

Zheng *et al.* [43] proposed Triton-Distributed, a Triton extension that enables distributed programming in Triton. Triton-Distributed introduces communication primitives to Triton and overlapping optimizations to the Triton compiler to hide communication latency. The contributions of Triton-Distributed and Decoupled Triton are independent and complementary. Decoupled Triton could be extended to support distributed programming by adding a communication representation to the kernel definition, generating Triton kernels that employ Triton-Distributed's communication primitives, and leveraging the Triton-Distributed compiler to compile generated Triton kernels.

5.2 Generating Triton kernels

Triton has been selected by some tools and higher-level programming languages as an intermediate code generation target because of its intuitive block-level programming paradigm and optimizing compiler that can compete with expert-written kernels from high-performance vendor libraries.

Li *et al.* [20] proposed AutoTriton, an AI model specialized for generating Triton kernels. AutoTriton aims to simplify GPU programming and automatically generate efficient ML kernels written in Triton. While the goals of AutoTriton are similar to those of Decoupled Triton, their method and contributions differ significantly. Like the Decoupled Triton compiler, the AutoTriton model takes an ML kernel description

as input and generates a Triton kernel and wrapper function. However, compared to AutoTriton, Decoupled Triton offers several advantages. While AutoTriton may generate syntactically or semantically incorrect code, the Decoupled Triton compiler is robust and will always generate correct code. Also, users are not able to explicitly specify kernel scheduling to AutoTriton and therefore it is impossible to iterate upon a schedule or explore different schedules to find an efficient implementation.

Ansel *et al.* [4] proposed PyTorch 2.0, a version of the popular ML framework that introduced JIT compiler extensions to run ML kernels efficiently. In particular, PyTorch 2.0 introduced TorchDynamo, the JIT compiler for PyTorch programs, and TorchInductor, the default compiler backend for TorchDynamo that generates Triton kernels. TorchInductor uses a combination of heuristics, auto-tuning, and predefined templates to determine the schedule of an ML kernel. TorchInductor is a baseline compared against Decoupled Triton in the experimental evaluation of this thesis (Chapter 4). As shown in Section 4.4, TorchInductor generally generates efficient Triton kernels. However, for ML kernels involving matrix product, TorchInductor is unable to fuse operations with the matrix product because it uses a high-performance library kernel unless `max-autotune` is enabled. If the mode is set to `max-autotune`, then TorchInductor uses Triton kernel templates for matrix product and is able to fuse other operations into the kernel template. Unfortunately, TorchInductor with `max-autotune` enabled is generally much worse than the default TorchInductor at generating efficient Triton kernels. We believe that these results are caused by kernel templates with schedules that are poorly optimized for many algorithms and dimension sizes. Using Decoupled Triton, developers can define schedules that compete with TorchInductor in most cases and outperform TorchInductor for ML kernels that benefit from fusing operations with matrix product. The other advantage that Decoupled Triton has over TorchInductor is that its decoupling of the algorithm from

the schedule enables developers to specify their own scheduling using their expert knowledge, explore schedules quickly, and find an efficient implementation.

Dong *et al.* [12] proposed Flex Attention, a compiler-driven programming model to implement different attention variants. Flex Attention uses a template-based lowering to generate high-performance attention Triton kernels. The system uses a hand-written high-performance attention Triton kernel template that is filled by the computation graphs of two special PyTorch functions defined by the user. Flex Attention is only flexible in terms of the attention algorithm — using the two special functions — it is not flexible in terms of scheduling. Unlike Decoupled Triton, Flex Attention does not allow users to define the scheduling of their attention kernels. Furthermore, Flex Attention is limited to generating high-performance Triton kernels for attention variants, while Decoupled Triton is a general DSL that can be used to define any ML kernel.

5.3 Decoupled languages for Machine Learning

Halide [30, 31] introduced the concept of decoupling the algorithm from the schedule in programming languages. Section 2.1 discusses Halide in greater detail. Since the introduction of Halide, many researchers have applied this concept of decoupling to the domain of ML kernels.

Apache TVM [7] and TACO [19] are decoupled languages for defining high-performance ML kernels. Like Decoupled Triton, both Apache TVM and TACO use tensor expressions to define the algorithm of a kernel and expose a scheduling API to allow users to specify the schedule of the kernel using scheduling primitives. TileLang [41] is a tiled decoupled programming language built on top of Apache TVM for writing efficient ML kernels. Inspired by Triton, TileLang describes computation using a tile-level programming model, but exposes low-level scheduling details, such as the control

over the memory hierarchy, to the programmer. The primary difference between Decoupled Triton and these works is that Decoupled Triton is an abstraction layer on top of Triton. Consequently, the scheduling primitives available in Decoupled Triton are designed for defining schedules in the Triton language. Thus, Decoupled Triton has a comparatively smaller set of simple scheduling primitives that are intuitive to users familiar with the block-level programming model presented by Triton. Furthermore, because it generates Triton kernels, Decoupled Triton is well positioned to act as an educational tool for developers to learn how to write efficient block-level schedules in Triton. Unfortunately, as an abstraction layer on top of Triton, Decoupled Triton is unable to schedule low-level details that are not representable in a Triton kernel. By leaving these low-level scheduling details to the compiler, we sacrifice some fine-grained user control in exchange for intuitive and simple scheduling.

Slapo [6] is a schedule language for scheduling at the model level. Slapo takes a PyTorch [4] model definition and uses a set of scheduling primitives to transform the model-level schedule. Decoupled Triton is focused on the block-level schedule of ML kernels rather than the entire model.

Exo [5, 17, 18] is a user-schedulable programming language based on the novel exocompilation compilation strategy that externalizes both target-specific code generation and optimization from the compiler and exposes it in user-level code. Most decoupled languages, similar to Decoupled Triton, have a code generator or compiler that takes an algorithm and a schedule as inputs and generates scheduled code. Exo uses a rewriter strategy where the user iteratively applies scheduling primitives to an IR to rewrite it successively until they have sufficiently transformed the IR, which is then compiled to a code generation target. Exo2 enables users to define new schedule rewrite rules external to the compiler. Unlike Decoupled Triton, Exo does not support GPUs as code generation targets at the moment.

Chapter 6

Future Work

The design of Decoupled Triton is extensible. There are many possible directions for future work in the area of high-performance ML kernels. Here are some of the possible future directions:

1. As described in Section 3.4, the main limitation of Decoupled Triton is its inability to represent highly specialized kernel schedules like FlashAttention [10]. Creating new scheduling primitives that enable the Decoupled Triton DSL to represent schedules like FlashAttention is an interesting challenge.

2. Gluon [25] is an experimental language emerging from the Triton project. It is a lower-level language than Triton and is intended to replace the middle-end of the Triton compiler. Gluon exposes more control over low-level details like shared memory and tensor layouts to the programmer. Gluon maintains the same block-level programming paradigm as Triton. Targeting Gluon kernels with Decoupled Triton, would enable the creation of new scheduling primitives that interact with the low-level details exposed in Gluon. However, Gluon is currently highly experimental. Thus, it may be difficult to make progress in this direction at the moment.

3. Build an autoscheduler for Decoupled Triton. A system that automatically

generates efficient schedules for a given DT algorithm would make schedule exploration even faster [23]. Decoupled Triton's block-level scheduling may present a unique challenge when creating an autoscheduler.

4. Integrate the Decoupled Triton system into PyTorch [4]. Integrating a decoupled language like Decoupled Triton into a coupled language like PyTorch provides an interesting challenge. Such an integration would make exploring schedules in Decoupled Triton even faster.

5. Add communication and distributed programming primitives to Decoupled Triton. Inspired by Triton-Distributed [43] (Section 5.1), there is an opportunity to incorporate distributed programming and communication primitives into the Decoupled Triton DSL to create a decoupled distributed programming DSL.

6. NVIDIA's Hopper architecture introduced a Tensor Memory Accelerator (TMA) unit as a new asynchronous execution feature [2]. The TMA unit can efficiently transfer large blocks of memory between global memory and shared memory. Triton has support for TMA, however existing Triton kernels cannot use TMA without being rewritten [15]. Therefore, old Triton kernels are no longer portable. Enhancing Decoupled Triton to generate Triton kernels that use TMA is an important research direction towards more efficient and more portable kernels.

7. To enable backpropagation for training, a backward ML kernel is needed in addition to the original ML kernel — the forward kernel. Some systems can generate backward kernels automatically using automatic differentiation [29]. In Decoupled Triton however, developers must define both the forward and backward kernels individually. Implementing automatic differentiation to generate the backward kernel in addition to the forward kernel from a single kernel

definition would simplify writing algorithms for training. This direction would present the new research problem of how to schedule the automatically generated backward kernel.

Chapter 7

Conclusion

This thesis presents Decoupled Triton, a block-level decoupled DSL for writing and exploring efficient parallel ML kernels. Decoupled Triton acts as an abstraction layer on top of Triton that decouples the algorithm from the schedule. Adopting the block-level programming paradigm from Triton allows for intuitive and simple scheduling primitives, while decoupling the algorithm from the schedule allows for user-defined scheduling and rapid schedule exploration. Furthermore, the higher-level representation of algorithm in Decoupled Triton further simplifies the definition of ML kernels.

An ML kernel definition in Decoupled Triton is represented as a modular algorithm, which is defined by a tensor expression, and schedule, which is defined using block-level scheduling primitives. The Decoupled Triton compiler takes an ML kernel definition and generates a Triton kernel and a wrapper function. By targeting Triton kernels, Decoupled Triton sacrifices low-level scheduling control for simple and intuitive block-level scheduling. Moreover, Decoupled Triton may be used as an educational tool for learning how to write efficient block-level schedules in Triton.

We implement different ML kernels in Decoupled Triton, explore parts of their schedule spaces, and measure the performance of the generated kernels. The experimental evaluation of Decoupled Triton demonstrates how Decoupled Triton enables developers to rapidly explore schedule spaces and find efficient ML kernels with per-

formance competitive with, and even exceeding, that of both ML kernels written by expert Triton developers and ML kernels generated by PyTorch, a mature ML programming framework.

Despite the expressive algorithm definition and scheduling primitives, Decoupled Triton cannot be used to define highly-specialized schedules like FlashAttention. Future research directions include extending Decoupled Triton with scheduling directives that enable the definition of FlashAttention, building an autoscheduler to automatically generate efficient schedules, and generating Triton kernels that utilize TMA.

Bibliography

[1] A. Adams *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019, ISSN: 0730-0301. DOI: 10.1145/3306346.3322967. [Online]. Available: https://doi.org/10.1145/3306346.3322967. 7

[2] M. Andersch *et al.* "NVIDIA Hopper Architecture In-Depth." (Mar. 2022), [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/ (visited on 08/22/2025). 81

[3] L. Anderson, A. Adams, K. Ma, T.-M. Li, T. Jin, and J. Ragan-Kelley, "Efficient automatic scheduling of imaging and vision pipelines for the GPU," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485486. [Online]. Available: https://doi.org/10.1145/3485486. 7

[4] J. Ansel *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*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 929–947, ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. [Online]. Available: https://doi.org/10.1145/3620665.3640366. 3, 11, 55, 57, 77, 79, 81

[5] A. Castelló, J. Bellavita, G. Dinh, Y. Ikarashi, and H. Martínez, "Tackling the matrix multiplication micro-kernel generation with exo," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 182–193. DOI: 10.1109/CGO57630.2024.10444883. 79

[6] H. Chen, C. H. Yu, S. Zheng, Z. Zhang, Z. Zhang, and Y. Wang, "Slapo: A Schedule Language for Progressive Optimization of Large Deep Learning Model Training," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 1095–1111, ISBN: 9798400703850. DOI: 10.1145/3620665.3640399. [Online]. Available: https://doi.org/10.1145/3620665.3640399. 79

[7] T. Chen *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)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594, ISBN: 978-1-939133-08-3. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen. 2, 78

[8] S. Chetlur *et al.*, *cuDNN: Efficient Primitives for Deep Learning*, 2014. arXiv: 1410.0759 [cs.NE]. [Online]. Available: https://arxiv.org/abs/1410.0759. 1

[9] *cuBLAS documentation*, 12.9.1, https://docs.nvidia.com/cuda/archive/12.9.1/cublas/index.html, NVIDIA Corporation, Santa Clara, CA, USA, Jun. 2025. 1, 9

[10] T. Dao, *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*, 2023. arXiv: 2307.08691 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2307.08691. 3, 46, 73, 80

[11] T. Dao, D. 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*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 16 344–16 359. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf. 46

[12] J. Dong, B. Feng, D. Guessous, Y. Liang, and H. He, *Flex Attention: A Programming Model for Generating Optimized Attention Kernels*, 2024. arXiv: 2412.05496 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2412.05496. 78

[13] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2. 17

[14] *HIP documentation*, 6.4.3, https://rocm.docs.amd.com/projects/HIP/en/docs-6.4.3/index.html, Advanced Micro Devices, Inc, Santa Clara, CA, USA, Jun. 2025. 2, 8, 9

[15] A. Hoque, L. Wright, and C.-C. Yang. "Deep Dive on the Hopper TMA Unit for FP8 GEMMs." (Jul. 2024), [Online]. Available: https://pytorch.org/blog/hopper-tma-unit/ (visited on 08/22/2025). 81

[16] P.-L. Hsu *et al.*, *Liger Kernel: Efficient Triton Kernels for LLM Training*, 2025. arXiv: 2410.10989 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2410.10989. 53–55, 57

[17] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718, ISBN: 9781450392655. DOI: 10.1145/3519939.3523446. [Online]. Available: https://doi.org/10.1145/3519939.3523446. 79

[18] Y. Ikarashi, K. Qian, S. Droubi, A. Reinking, G. L. Bernstein, and J. Ragan-Kelley, "Exo 2: Growing a Scheduling Language," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 426–444, ISBN: 9798400706981. DOI: 10.1145/3669940.3707218. [Online]. Available: https://doi.org/10.1145/3669940.3707218. 79

[19] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. DOI: 10.1145/3133901. [Online]. Available: https://doi.org/10.1145/3133901. 78

[20] S. Li *et al.*, *AutoTriton: Automatic Triton Programming with Reinforcement Learning in LLMs*, 2025. arXiv: 2507.05687 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2507.05687. 76

[21] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in halide," *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018, ISSN: 0730-0301. DOI: 10.1145/3197517.3201383. [Online]. Available: https://doi.org/10.1145/3197517.3201383. 7

[22] M. Milakov and N. Gimelshein, *Online normalizer calculation for softmax*, 2018. arXiv: 1805.02867 [cs.PF]. [Online]. Available: https://arxiv.org/abs/1805.02867. 46

[23] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016, ISSN: 0730-0301. DOI: 10.1145/2897824.2925952. [Online]. Available: https://doi.org/10.1145/2897824.2925952. 7, 81

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. [Online]. Available: https://doi.org/10.1145/1365490.1365500. 2, 8, 9

[25] J. Niu and B. Yoshimi, *Triton Community Meetup 20250709 130219 Meeting Recording: Gluon Update*, Jul. 2025. [Online]. Available: https://youtu.be/5e1YKqsP8i8?feature=shared&t=1039. 80

[26] OpenAI *et al.*, *Dota 2 with Large Scale Deep Reinforcement Learning*, 2019. arXiv: 1912.06680 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1912.06680. 1

[27] OpenAI *et al.*, *GPT-4 Technical Report*, 2024. arXiv: 2303.08774 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2303.08774. 1

[28] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd. Pragmatic Bookshelf, 2013, ISBN: 1934356999. 49

[29] A. Paszke *et al.*, "Automatic differentiation in PyTorch," NIPS Workshop Autodiff, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ. 81

[30] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012, ISSN: 0730-0301. DOI: 10.1145/2185520.2185528. [Online]. Available: https://doi.org/10.1145/2185520.2185528. 2, 5, 6, 78

[31] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, Jun. 2013, ISSN: 0362-1340. DOI: 10.1145/2499370.2462176. [Online]. Available: https://doi.org/10.1145/2499370.2462176. 2, 5, 22, 78

[32] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016. 1

[33] A. Saeed, A. Ahmadinia, and M. Just, "Tag-Protector: An Effective and Dynamic Detection of Illegal Memory Accesses through Compile Time Code Instrumentation," *Advances in Software Engineering*, vol. 2016, no. 1, p. 9 842 936, 2016. DOI: https://doi.org/10.1155/2016/9842936. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2016/9842936. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2016/9842936. 3, 9

[34] C. Salvador Rohwedder, N. Henderson, J. P. L. De Carvalho, Y. Chen, and J. N. Amaral, "To Pack or Not to Pack: A Generalized Packing Analysis and Transformation," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '23, Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 14–27, ISBN: 9798400701016. DOI: 10.1145/3579990.3580024. [Online]. Available: https://doi.org/10.1145/3579990.3580024. 6

[35] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision," in *Advances in Neural Information Processing Systems*, A. Globerson *et al.*, Eds., vol. 37, Curran Associates, Inc., 2024, pp. 68 658–68 685. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/7ede97c3e082c6df10a8d6103a2eebd2-Paper-Conference.pdf. 46

[36] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. DOI: 10.1109/MCSE.2010.69. 2, 8, 9

[37] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Modern Deep Learning Research," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 09, pp. 13 693–13 696, Apr. 2020. DOI: 10.1609/aaai.v34i09.7123. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7123. 1

[38] 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*, ser. MAPL 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19, ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. [Online]. Available: https://doi.org/10.1145/3315508.3329973. 2, 8

[39] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 70

[40] D. Wang *et al.*, *ML-Triton, A Multi-Level Compilation and Language Extension to Triton GPU Programming*, 2025. arXiv: 2503.14985 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2503.14985. 75

[41] L. Wang *et al.*, *TileLang: A Composable Tiled Programming Model for AI Systems*, 2025. arXiv: 2504.17577 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2504.17577. 78

[42] M. J. Wolfe, C. Shanklin, and L. Ortega, *High Performance Compilers for Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0805327304. 34

[43] S. Zheng *et al.*, *Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler*, 2025. arXiv: 2504.19442 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2504.19442. 76, 81