Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs

Yaoyao Ding†
University of Toronto
Toronto, Canada
yaoyao@cs.toronto.edu

Cody Hao Yu
Amazon Web Services
Santa Clara, USA
hyuz@amazon.com

Bojian Zheng†
University of Toronto
Toronto, Canada
bojian@cs.toronto.edu

Yizhi Liu
Amazon Web Services
Santa Clara, USA
yizhiliu@amazon.com

Yida Wang
Amazon Web Services
Santa Clara, USA
wangyida@amazon.com

Gennady Pekhimenko†
University of Toronto
Toronto, Canada
pekhimenko@cs.toronto.edu

ABSTRACT

As deep learning models nowadays are widely adopted by both cloud services and edge devices, reducing the latency of deep learning model inferences becomes crucial to provide efficient model serving. However, it is challenging to develop efficient tensor programs for deep learning operators due to the high complexity of modern accelerators (e.g., NVIDIA GPUs and Google TPUs) and the rapidly growing number of operators.

Deep learning compilers, such as Apache TVM, adopt declarative scheduling primitives to lower the bar of developing tensor programs. However, we show that this approach is insufficient to cover state-of-the-art tensor program optimizations (e.g., double buffering). In this paper, we propose to embed the scheduling process into tensor programs and use dedicated mappings, called task mappings, to define the computation assignment and ordering directly in the tensor programs. This new approach greatly enriches the expressible optimizations by allowing developers to manipulate tensor programs at a much finer granularity (e.g., allowing program statement-level optimizations). We call the proposed method the task-mapping programming paradigm. In addition, we propose a new post-scheduling fusion optimization that allows developers to focus on scheduling every single operator and automates the fusion after scheduling. It greatly reduces the engineering efforts for operator fusion. Our proposed paradigm also constructs an efficient hardware-centric schedule space, which is agnostic to the expressible optimizations by allowing developers to manipulate tensor programs directly in the tensor programs. This new approach greatly enriches the expressible optimizations by allowing developers to manipulate tensor programs at a much finer granularity (e.g., allowing program statement-level optimizations).

With the proposed paradigm, we implement a deep learning compiler – Hidet. Extensive experiments on modern convolution and transformer models show that Hidet outperforms state-of-the-art DNN inference framework, ONNX Runtime, and compiler, TVM equipped with scheduler AutoTVM and Ansor, by up to 1.48× (1.22× on average). It also reduces the tuning time by 20× and 11× compared with AutoTVM and Ansor, respectively. We open-sourced hidet at https://www.github.com/hidet-org/hidet.

CCS CONCEPTS

• Computing methodologies → Parallel programming languages; Machine learning; Artificial intelligence.

KEYWORDS

deep learning systems, systems for machine learning, programming models, compilation, tensor computation

ACM Reference Format:
Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gen-
tional Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS ’23), March 25–29, 2023, Vancouver,

1 INTRODUCTION

Deep neural networks (DNNs) [35] have achieved state-of-the-art (SOTA) results in various tasks such as image recognition [25, 34, 49, 50], natural language translation [16, 36, 48], and autonomous driving [14]. In deployment environments, these models are repeatedly executed to serve continuous user requests, named model serving. Thus, it is crucial to reduce the latency and maximize the throughput of model execution to ensure safety, save energy, and improve user experience.

There are two major ways to execute a DNN model. (1) Deep learning (DL) frameworks such as Tensorflow [1], PyTorch [41] and ONNX Runtime [15] dispatch operators to kernel libraries such as cuDNN [12], cuBLAS [26], and CUTLASS [32] during execution. (2) On the other hand, DL compilers such as TensorFlow-XLA [44] and TVM [9] automatically generate kernels through a compilation process for the given operators. Various schedulers such as Ansor [65] and AutoTVM [11] are used to schedule the kernels during compilation to achieve high performance.

Kernel libraries (e.g., cuDNN [12] and cuBLAS [26]) provide a collection of highly optimized hand-crafted kernels (e.g., convolutions and matrix multiplications). These libraries typically achieve
near-peak performance on widely used input sizes, as they are able to implement a large spectrum of optimizations in low-level languages (e.g., CUDA C/C++ and assembly code). However, manually tweaking a kernel to optimize for performance is laborious, error-prone, and requires expertise in writing low-level language codes. Thus, it is difficult to generalize to other input shapes, new operators, and kernel fusion patterns. In addition, template-based libraries such as CUTLASS [32] employ C++ templates to generate tensor programs for different input shapes on the fly. Although template-based libraries can achieve competitive performance on many input shapes by dynamically tuning the optimization hyper-parameters, they do not reduce the complexity of writing tensor programs for new operators and only provide limited fusion capability (e.g., only a small number of predefined operators can be fused with matrix multiplication).

Alternatively, DL compilers [4, 9, 43, 44, 67] are proposed to compile deep learning networks into tensor programs automatically. Existing state-of-the-art DL compilers adopt the idea of decoupling computation definition and scheduling, originally proposed by Halide [43] and TVM [9]. The computation definition of an operator only defines how each element of the output tensor is computed mathematically, and the schedule defines the way the execution is performed, such as the loop order and thread binding [9, 43]. Compilers leverage schedulers like AutoTVM [11] and Ansor [65] to tune the hyper-parameters of the schedule to optimize operator performance for each input shape. Unlike kernel libraries and templates that target a fixed set of operators and limited fusion patterns, compilers are capable of supporting more operators and more flexible fusion patterns automatically.

However, existing state-of-the-art compilers are mostly based on the loop-oriented scheduling primitives, which manipulate the loop structure of a tensor program in a declarative manner (e.g., loop split and reorder). Although loop-oriented scheduling primitives have achieved great success in simplifying tensor program writing [9, 11, 65], certain key optimizations (e.g., double buffering [32]) are hard to implement. Specifically, loop-oriented scheduling primitives cannot express the fine-grained tensor program transformations required by the key optimizations discussed in Section 3.1. Besides, loop-oriented scheduling also suffers from the long kernel tuning time due to the rarity of efficient schedules in the tremendous tuning spaces. For instance, AutoTVM [11] takes 15 hours to tune a single CNN model Inception V3 [50] on a modern GPU.

In this work, we propose a new paradigm for writing efficient tensor programs: task-mapping-oriented programming paradigm. In this paradigm, we define the parallelizable computations in an operator as tasks, and the process of assigning and ordering the tasks to parallel processing units (e.g., threads) as scheduling. The developers can directly define the scheduling in the tensor program through task mappings. This paradigm simplifies the development of tensor programs without sacrificing the ability to express optimizations requiring fine-grained program manipulation. With the in-program style of scheduling, this paradigm also allows us to search the tensor program in an efficient hardware-centric schedule space that is agnostic to input size to dramatically reduce the tuning time. We also propose post-scheduling fusion to fuse the scheduled operator with surrounding operators automatically, so developers don’t need to worry about fusion when writing schedule templates.

We implement a new DL compiler called Hidet based on the proposed ideas. In this work, we mainly focus on optimizing DNN inference on GPUs, as it is the most commonly used DNN accelerator. The proposed ideas also apply to other accelerators such as CPUs and TPUs [31]. Extensive experiments on modern convolutional and transformer models show that Hidet outperforms state-of-the-art DL inference frameworks and schedulers, AutoTVM [11] and Ansor [65], by up to 1.48× (1.22× on average) while reducing the tuning time of the two schedulers by 20× and 11×, respectively.

We summarize our contributions as follows:

- We identify and present the limited expressiveness of loop-oriented scheduling adopted by state-of-the-art DL compilers to be their fundamental limitation in efficiently compiling complex tensor programs (e.g., matrix multiplication).
- We introduce the task-mapping-oriented programming paradigm to simplify tensor program development without sacrificing the expressiveness of optimizations compared with hand-crafted implementations. Based on this paradigm, we propose post-scheduling fusion to fuse the scheduled program with surrounding operators. The paradigm also allows us to search in the hardware-centric schedule space to reduce the tuning time significantly.
- We implement a new DL compiler, named Hidet, based on the proposed ideas. Extensive experiments show that Hidet outperforms state-of-the-art DL frameworks and compilers by up to 1.48× and reduces tuning time by 11×. We have open-sourced Hidet here.

2 BACKGROUND

2.1 CUDA Programming Model

The CUDA programming platform [40] is widely used by deep learning systems on NVIDIA GPUs. In this section, we briefly introduce the CUDA programming model on modern GPUs.

Figure 1: An overview of CUDA programming model.

Kernel, Thread Block, and Thread. When running a workload on the GPU, thousands of threads will be executed. Each thread executes the same piece of code, called kernel code. When launching a kernel, a grid of thread blocks will be dispatched onto the GPU as shown in Figure 1. Each grid usually comprises tens to thousands of thread blocks, while each thread block comprises tens to hundreds of threads. In the kernel code, pre-defined variables threadIdx and blockIdx, and suffix x, y, and z are used to access the 3-dimensional...
index of thread in a thread block and the thread block in the grid of blocks, respectively.

**Hardware Implementation.** Each modern GPU has tens to hundreds of streaming multiprocessors (SMs). Each SM supports scheduling up to thousands of concurrent threads [40]. Threads in a thread block are partitioned into warps, and each warp contains 32 consecutive threads executing the same instructions. There are two kinds of programmable on-chip memory: shared memory and registers. Registers are privately allocated to each thread, while shared memory is allocated to each thread block and only threads in the thread block can access it. When launching a kernel, the thread blocks are dispatched to the SMs wave by wave [23]. Each thread block will only be dispatched to a single SM while each SM may contain multiple thread blocks. The number of maximum resident thread blocks per SM is limited by the size of shared memory, register file, and warp scheduling units.

Operators in the deep neural network are implemented as GPU kernels. When running a neural network, we launch these kernels following an order satisfying the operator dependency. Among these operators, matrix multiplication (also known as a linear or dense layer) is one of the most important operators. We next present an efficient implementation of matrix multiplication using CUDA and take it as an example throughout the paper.

### 2.2 Efficient Matrix Multiplication

This section illustrates an efficient implementation of matrix multiplication C = AB (all matrices are M×K×N) on modern NVIDIA GPUs via Tensor Cores [13]. Figure 2 shows the desired workflow. In step 1, we decompose the matrix multiplication into independent subtasks by tiling the M and N dimensions. After tiling, there will be \( \frac{M}{M_{\text{tile size}}} \times \frac{N}{N_{\text{tile size}}} \) independent subtasks while each sub-task is a matrix multiplication with size: M tile size × N tile size × K. Each subtask will be assigned to a thread block. Inside each thread block, the K dimension will be further tiled into \( K_{\text{tile size}} \) tiles, and the thread block will apply steps 2-3 to each K tile. In step 3, threads in the thread block load fragments of matrix A and B from global memory to shared memory collectively (i.e., different threads load different parts of the fragments). All threads in a thread block will be synchronized to make sure the data loading is finished before proceeding to the next step. In step 4, 4 warps in the thread block work on \( 4 \times 4 = 16 \) matrix multiply accumulate (MMAs), each of which is an operation \( C_{16\times16} = A_{16\times8}B_{8\times16} + C_{16\times16} \). Each warp conducts 4 MMAs using Tensor Core [13] with 4 sequential iterations. Once we accumulate the results of matrix multiplication for each K tile, we store the results from the accumulating register to global memory. Figure 3 gives the pseudo-code of the 4 steps.

There are two ways to implement the kernel: (1) directly write the CUDA C code as in kernel libraries [12, 26, 32], or (2) use declarative loop-oriented scheduling primitives. In the next subsection, we would give a brief introduction to the second method.

### 2.3 Declarative Loop-Oriented Scheduling

To simplify tensor program optimization, Halide [43] proposes a programming paradigm of tensor programs, in which the computation definition and scheduling of the computation are decoupled. This programming paradigm is adopted by state-of-the-art DNN compiler TVM [9] and schedulers (e.g., AutoTVM [11] and An- sor [65]). Since this paradigm offers a set of declarative scheduling primitives to manipulate the loop structure of tensor programs, we name it declarative loop-oriented scheduling.

Figure 4 shows the workflow of loop-oriented scheduling. Developers first provide a mathematical computation of the operator that defines how each element in the tensor is computed. The example gives the definition of matrix multiplication, where the (i, j)-th element of the output is a sum reduction. Given the computation definition, the schedulers first generate a default tensor program...
from the computation definition automatically by translating the compute and reduce primitives to nested loops. Then, a series of declarative scheduling primitives are applied to transform the loop structure of the default tensor program for better performance on the specific hardware. Table 1 shows the scheduling primitives in TVM [9]. In the example of step 3, we only list the first few scheduling primitives to implement the matrix multiplication, as TVM has used over 80 primitives to schedule matrix multiplication. Starting from the default program, we first split the i and j loops with factor 64 into for (oi, ii) in range(64) and for (oj, ij) in range(64), respectively, then reorder loops into for (oi, oj) in range(64) and for (ij, i) in range(64), and finally bind oi and oj to blockIdx.x and blockIdx.y, respectively. With these primitives, we can get the scheduled program in Figure 4.

There are several ways to make use of a programming paradigm in a deep learning compiler. Intuitively, we can manually write a schedule for each workload (i.e., an operator with a concrete input on certain hardware) [9, 43]. However, this approach requires significant engineering efforts to achieve optimal performance for all widely used operators and their typical input sizes. Consequently, tunable parameters (e.g., tile size and loop orders) are introduced for developers to specify in the schedules. In this way, a manual schedule becomes a schedule template and can be optimized by auto-tuning frameworks [11] for various input shapes and hardware. To further save the time of writing a schedule template, auto-scheduling approaches that automatically generate a schedule by applying predefined rules to the computation definition have been proposed [2, 65].

However, as we illustrate in the next section, the schedule space from the loop-oriented scheduling paradigm is still inefficient. As a result, 1) it is challenging to achieve competitive performance on operators that are highly optimized by kernel libraries since loop-oriented scheduling can neither express some key optimizations, 2) schedulers need hours to days to find the best schedule configuration in the schedule space.

### 3 MOTIVATION

In this section, we summarize the challenges faced by state-of-the-art loop-oriented scheduling.

#### 3.1 Limited Optimization Support

The declarative loop-oriented scheduling primitives suffer from limited support for key optimizations. We use an important optimization, double buffering [6, 32], that has been adopted in several vendor libraries (e.g., cuBLAS [26] and CUTLASS [32]) but not supported by TVM [9], to illustrate this fundamental limitation.

The implementation of matrix multiplication in Figure 3 is suboptimal since all threads in the same thread block are likely to be blocked by one type of hardware resource (i.e., memory bandwidth in Step 2 or computation units in Step 3) while leaving the other idle. This is because, in Figure 3, the data loading (L7) and computation (L10) use the same buffer, and synchronization (L8) needs to be used to satisfy data dependency.

The double buffering optimization shown in Figure 5 alleviates the aforementioned problem by using two buffers: one is used for pre-loading the fragments for the next iteration (L8 and L10), while the other is used for computation in the current iteration (L9). We first pre-load the next tile of matrix A and B into registers (L8), and store them to shared memory after the computation of the current tile (L10). This is more efficient because computation in L9 can be executed while the global memory loading in L8 is on the fly with thread-level parallelism. With double buffering, the

---

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 
13. 
14. 

---

![Figure 5: Double Buffering Optimization.](image_url)
threads in a thread block can utilize both memory accessing units and computation units at the same time.

However, this optimization cannot be implemented using existing declarative loop-oriented scheduling primitives in Table 1. This is because none of the schedule primitives can manipulate the loop body at a fine granularity. As a result, although loop-oriented scheduling simplifies tensor program writing, its declarative style of scheduling prevents developers from implementing optimizations requiring fine-grained manipulation of tensor programs. We want to highlight that double buffering optimization is only an example of the limited expressiveness of existing loop-oriented scheduling. Besides double buffering, thread block swizzle [7, 53] and efficient usage of Tensor Core MMA PTX instruction [28], and multi-stage asynchronous prefetching [32] are widely used optimizations in kernel libraries [26, 32], but are difficult to implement with declarative loop-oriented scheduling. To implement these optimizations, we need a more expressive method to write tensor programs and schedule their computations.

### 3.2 Dedicated Schedule Template for Fusion

![Workflow of TVM sub-graph fusion](image)

One important advantage of compilers over kernel libraries is the ability to optimize arbitrary workloads, especially workloads with multiple fused operators (e.g., Conv2d–BN–ReLU in convolutional neural networks [25], and Reshape–Matmul–Transpose in transformer models [16]). For example, Figure 6 illustrates how TVM [9] fuses Conv2d–BN–ReLU into a single kernel. Specifically, TVM groups operators to form sub-graphs. Each sub-graph can contain only one anchor operator, which is usually the most compute-intensive one (e.g., convolution or matrix multiplication) with a carefully designed schedule template. Then, the schedule template of the anchor operator will be used to schedule the entire sub-graph, meaning that the schedule template has to support all possible fusion scenarios, which greatly increases the complexity of writing schedule templates. Although auto-schedulers (e.g., Ansor [65]) are proposed to generate schedule templates automatically from the compilation definition with pre-defined auto-scheduling rules, it is challenging to extend the auto-schedulers with new rules. This is because the new rule has to be compatible with all existing rules and needs to be general enough to support all operators. Thus, it is still challenging to support fusion, while not increasing the complexity of writing specialized schedule templates.

3 Even though TVM tried to use a new primitive called double_buffer to implement double buffering optimization, it does not separate the global memory loading and shared memory storing, thus can only achieve sub-optimal performance.

4 Directly use MMA PTX instruction instead of WMMA instruction [28].

### 3.3 Long Tuning Time

In addition to expressiveness and extensibility, the tuning time of existing state-of-the-art schedulers [2, 11, 65] typically ranges from hours to days due to inefficient schedule spaces. The majority of their schedule spaces are composed of loop tiling factors. To constrain the schedule size and avoid conditional if-else branches, existing schedulers only cover perfect tile sizes (i.e., only tile n-length loop with proper factors of n). For example, potential tile factors of a loop with length 10 only include 1, 2, 5, and 10. As a result, the space constructed by these schedulers with loop-oriented scheduling depends on the input shapes of the target workload. We name this category of schedule space as input-centric schedule space. We observe two challenges with input-centric schedule space. (1) The schedule space size grows exponentially along with the number of input size factors. Figure 7 shows the number of schedules for each convolution in ResNet-50 [25]. There are up to 10^8 schedules to search for a single convolutional layer. (2) The schedule space might not include the schedule with optimal performance as non-perfect tile sizes are not considered. An extreme example is that both Ansor and AutoTVM fail to find a valid schedule for matrix multiplication with M=N=K=2039 because 2039 is a prime number.

To address the first challenge, the state-of-the-art schedulers [11, 65] employ a cost model to predict the performance of schedules and use genetic evolution search to increase the search efficiency. However, the search process still requires about half an hour to tune a single operator, resulting in 8 to 15 hours to tune an Inception V3 model [50]. Long tuning time prevents existing schedulers from co-optimizing DNNs with graph-level optimizations [30, 37] and upper-level applications such as neural architecture search [71]. Both of them need the latency of a kernel to guide their optimization and network searching within a short amount of tuning time.

### 4 KEY IDEAS

To address the challenges mentioned above, we propose a new programming paradigm for tensor programs—task-mapping programming paradigm (Section 4.1). This paradigm defines descriptive objects, called task mapping, to specify the task assignment and ordering. Task mappings replace the original loop-oriented scheduling primitives and are directly defined and used in the tensor program, which allows more optimizations compared with the existing declarative style of scheduling. We also propose post-scheduling fusion (Section 4.2) to simplify sub-graph scheduling by automatically fusing surrounding operators to the operator with scheduled tensor program. The proposed paradigm also enables efficient partial tiling.
Figure 8: Scheduling the cooperative loading with declarative loop-oriented scheduling and task-mapping programming paradigm. In declarative loop-oriented scheduling, developers apply a series of declarative scheduling primitives to automatically generate a program to transform the tensor program into a more efficient one. Instead of employing declarative primitives, the task-mapping programming paradigm allows developers to directly embed the scheduling in the tensor program and enables a larger spectrum of optimizations compared with loop-oriented scheduling.

4.1 Task-Mapping Programming Paradigm

Loop-oriented scheduling manipulates a tensor program through declarative loop-oriented scheduling primitives to simplify the tensor programming, but at the same time prevents fine-grained manipulations and optimizations.

We observe that the goal of loop-oriented scheduling primitives is either to (1) assign the computations to parallel processing units (e.g., threads or warps), or (2) specify the execution order of the computations assigned to each processing unit. Figure 8 shows the cooperative loading of the matrix A in the matrix multiplication as an example (we omitted the block offset and only show the loading of the matrix A for simplicity). In this example, loop-oriented scheduling applies three primitives (i.e., loop split, fuse, and bind) to assign the loading of 512 (64x8) elements to 128 threads, and each thread loads 4 elements in order.

Instead of scheduling through applying declarative primitives, we propose to embed the scheduling into tensor programs and use dedicated mappings, called task mappings, to define the computations assignment and ordering directly in the program. We use the example in Figure 8 to demonstrate how to use task mapping to fulfill the desired scheduling. In step (1), a task mapping is first defined, which assigns 64x8 tasks to 128 threads. Then, in step (2), each task (i, k) assigned to a thread is iterated by calling the task mapping with thread index threadIdx.x. Finally, in step (3), the task is implemented using its index (i, k). The three steps decouple the task assignment and the implementation of every single task, greatly simplifying tensor program developments. Compared with declarative loop-oriented scheduling, it schedules directly in the tensor program and allows more fine-grained optimizations.

Besides this, it also allows developers to fall back on some dimensions to traditional loops to implement optimizations such as double buffering [32]. Since task mapping is the key component used in the three steps, we name our new approach to construct tensor programs – a task-mapping programming paradigm.

The task mapping defined in step (1) is derived from task mapping composition of two basic task mappings (i.e., repeat(4, 1) and spatial(16, 8)). The table in Figure 8 gives the details of all appeared task mappings. The formal definition of task mapping and its composition are given in Section 5.1.

The proposed paradigm simplifies tensor program development without sacrificing optimization expressiveness. Beyond the scheduling of a single operator, it is also important to schedule a fused sub-graph as operator fusion could greatly reduce the memory traffic to accelerate the end-to-end DNN execution [9, 18, 30].

4.2 Post-Scheduling Fusion

We propose to decompose the scheduling of a fused sub-graph into two steps, as shown in Figure 9. In step 1, we select the anchor operator as TVM [9] does, but only schedule the anchor operator alone. In step 2, we fuse the surrounding operators to the scheduled tensor program of the anchor operator automatically. With this decoupling, the scheduling of the anchor operator does not need to consider the whole sub-graph but only the implementation of itself, which greatly reduces the engineering efforts required to a design schedule template for sub-graph compared with AutoTVM [11].
Because the fusion is done after we schedule the operator, we call this approach post-scheduling fusion.

In post-scheduling fusion, the anchor operator can be fused with operators before (as prologues) and after (as epilogues) it. We decide if an operator is fusible based on its characteristics. If an operator has no reduction computation, it is defined as injective and qualified as a prologue operator. If an operator is injective and each element in the input tensor contributes to a single element in the output tensor, it is defined as bijective and qualified as an epilogue operator. For example, all elementwise operators (e.g., addition, ReLU) and transform operators (e.g., reshape, transpose) are bijective operators and are qualified as both prologue and epilogue operators. With post-scheduling fusion, we can concentrate on the scheduling of a single operator while supporting flexible and effective fusion.

4.3 Hardware-Centric Scheduling Space

Existing state-of-the-art schedulers [11, 65] adopt the input-centric schedule space discussed in Section 3.3, in which the schedule chooses the proper factors of loop extent as the split or tile factors, which makes the schedule space unscalable and fails to cover the optimal performance derived from tile sizes that are not proper factors of loop extents. In addition to constructing a schedule space based on input sizes, another approach is to design the schedule space based on hardware, named hardware-centric schedule space. Hardware-centric schedule space decouples the schedule space from the input size by employing predicated loading (i.e., protecting the data loading by checking if the accessing indices are in bounds), and is widely used by kernel libraries [12, 26, 32].

With the proposed paradigm, we can provide a small but efficient hardware-centric schedule space. Since the tile factors are based on hardware resources (e.g., 64x64, 128x64, 16x32, etc), hardware-centric schedule spaces are orders of magnitude smaller than input-centric schedule spaces. For example, the schedule space we adopted for matrix multiplication contains less than 200 schedules, which is on average $10^2 \times$ smaller than a typical schedule space in AutoTVM [11]. Simply enumerating all schedules would be enough and can be done within one minute of time.

5 HIDET: SYSTEM DESIGN

With the above key ideas, we design and implement a DNN compiler, named HIDET. Figure 10 shows the overall design. HIDET firstly imports a deep neural network from a widely used framework like PyTorch [41] or a model file in ONNX [5] format, and then schedules graph-level optimizations, such as constant folding and partition of fusible sub-graphs. After graph-level optimizations, each anchor operator in the fusible sub-graphs is lowered for scheduling. In HIDET, we schedule the operator with task-mapping programming paradigm (Section 5.1) into a tensor program and tune the schedule in hardware-centric schedule space. Then, in step 0, the post-scheduling fusion (Section 5.2) is applied to fuse the scheduled tensor program of the anchor operator with its surrounding operators automatically. Finally, the fused tensor programs in HIPET’s intermediate representation (IR) will be optimized and lowered. A code generator will convert the lowered IR to CUDA kernels.

5.1 Task-Mapping Programming Paradigm

One key challenge when optimizing tensor programs for certain hardware with parallel processing units (e.g., modern CPUs, GPUs, and TPUs) is how to assign the independent (sub) tasks to the parallel processing units. Using cooperative loading in Figure 8 as an example, when loading the fragment of matrix A with shape 64x8 from global memory to shared memory, the 512 tasks are assigned to the 128 threads in a thread block, and each thread is assigned with 4 loading tasks. In this example, tasks are assigned to parallel processing units, called workers, and the tasks assigned to each worker will be executed in a specific order. In this section, we will first formalize the task assignment and ordering as task mapping, then introduce a binary operator on task mappings to compose task mappings, and finally discuss the scheduling based on task mappings.

5.1.1 Task Mapping. Formally, we define a worker set $W_n$ to be a set containing $n$ workers with id from 0 to $n − 1$:

$$W_n = \{0, 1, \ldots, n−1\}.$$  

We also define a task domain $T$ as

$$T = \{(t_0, t_1, \ldots, t_m−1) \mid 0 \leq t_i < d_i, t_i \in \mathbb{Z}\},$$

to represent all tasks we are interested in, where $m$ is the task dimension and $d = (d_0, d_1, \ldots, d_m−1)$ is the task shape.
A task mapping \( f \) is defined as a function that maps each worker in the worker set to a list of tasks in the task domain, that is
\[
f(w) = \{t^{(0)}, t^{(1)}, \ldots, t^{(q-1)}\}.
\]
where \( w \in W \) and \( t^{(i)} \in T \).

We find two basic task mappings that are very useful. The repeat\((d_1, \ldots, d_m)\) task mapping maps a grid of tasks \((d_1, \ldots, d_m)\) to a single worker while the spatial\((d_1, \ldots, d_m)\) task mapping maps a grid of tasks \((d_1, \ldots, d_m)\) to the same number of workers and each worker only works on a single task.

Figure 11 shows examples of two basic kinds of task mappings: repeat\((2, 2)\) and spatial\((2, 2)\). The number indicates the execution order of the tasks assigned to the same task, while the color indicates the worker to which the task was assigned.

5.1.2 Task Mapping Composition. In the example of cooperative loading, we can observe a hierarchical structure. The 64x8=512 tasks can be partitioned into 4 groups of tasks and each group contains 16x8=128 tasks. The 128 tasks in each group are executed by 128 threads. If we take each task group as a macro-task and the 128 threads as a macro-worker, then task-mapping of the macro-tasks to macro-workers is a task mapping that maps 4 tasks to a single worker, denoted by repeat\((4, 1)\). This example demonstrates that all the tasks in a task mapping can be treated entirely as a single task and all the workers can be treated entirely as a single worker in another task mapping to create a composed task mapping.

We formalize this idea as follows. Let \( f_1, f_2 \) be two task mappings with the same task dimension. Let \( n_1, n_2 \) be the number of workers and \( d_1, d_2 \) be the task shapes of the two task mappings. We define \( f_3 \) be the composed task mapping of \( f_1 \) and \( f_2 \) that has \( n_1n_2 \) workers and task shape \( d_3 = d_1 \odot d_2 \).

The task mapping composition is denoted as \( f_3 = f_1 \circ f_2 \). Task composition is associative, that is
\[
(f_1 \circ f_2) \circ f_3 = f_1 \circ (f_2 \circ f_3),
\]
holds for arbitrary task mappings \( f_1, f_2, f_3 \).

Task mapping composition is a powerful tool to construct new task mappings. Figure 12 gives some examples of task mapping composition. Besides these examples, task mapping spatial\((4, 2) \times \text{repeat}(2, 2) \times \text{spatial}(4, 8) \times \text{repeat}(4, 4)\) is used in matrix multiplication with CUDA Core [40]. They correspond to the warps in a block (4x8), the number of repeats for each warp (2x2), the layout of threads in a warp (4x8), and the number of C elements each thread works on (4x4), respectively.

Figure 13: Use task mapping to schedule warp-level tasks.

The task and worker in a task mapping are abstract concepts and can be used to describe tasks and workers on different hierarchical levels. For example, besides a single thread, a worker can also represent a warp, a thread block, or a processing unit in other accelerators. Figure 13 shows an example with warps as workers.
in a task mapping. It implements the `block_mma` function used in the aforementioned matrix multiplication (see Figure 3 and 5). In the example, we use a task mapping to assign a grid of 4 \times 4 tasks to 4 warps, and each warp takes 4 warp-level matrix-multiply-accumulate (MMA) task, whose corresponding assignment is shown in step 3 of Figure 2.

Task mappings and their composition could greatly simplify the tensor program writing as it employs dedicated mappings to define the task assignment and ordering, and free developers from writing complex loops and index calculations to achieve the same goal. We call the tensor program writing paradigm based on task mappings as task-mapping programming paradigm for tensor programs.

5.1.3 Scheduling Mechanisms. Based on the paradigm, we further implement two scheduling mechanisms in Hidet: template-based scheduling and rule-based scheduling. Inspired by Ansor [65] and Halide-AutoScheduler [2], rule-based scheduling directly generates the tensor program from one operator’s computation definition, without any extra engineering efforts and is used for the majority of operators in Hidet. On the other hand, rule-based scheduling might not be able to generate an efficient enough tensor program for key operators such as matrix multiplication. Inspired by AutoTVM [11], we also allow developers to provide a tensor program template to support the efficient scheduling of these operators. Figure 14 illustrates the two scheduling mechanisms.

Figure 14: Two Scheduling Mechanisms in Hidet.

Rule-based Scheduling generates the tensor program given the computation definition automatically. It traverses the computation definition in the form of a directed acyclic graph (DAG) and applies pre-defined rules to translate each node in the DAG into a part of the final tensor program. Because this mechanism does not require developers to write a dedicated schedule template, it is widely used in Hidet for the operators that do not include reduction, such as reshape, transpose, slice, and all element-wise arithmetic operators. On the other hand, for operators demanding extreme optimizations like matrix multiplication, we use another scheduling mechanism, named template-based scheduling.

Template-based Scheduling schedules the operator with the given template. A schedule template is a tensor program written with parameterized task mappings. Each schedule template is equipped with a schedule space containing a collection of available parameters for the parameterized task mappings, and the template can be instantiated with an arbitrary choice from the schedule space. Taking the matrix multiplication in Figure 5 and 13 as an example, we could use different numbers of warps and repeat different numbers of times for each warp to implement the matrix multiplication. These different choices form the schedule space for matrix multiplication. During scheduling, Hidet first enumerates the schedule choice from the schedule space. Then the schedule choice is used to create the task mappings for the given program template. Finally, Hidet instantiates the template into a tensor program and measures its performance. The schedule with the best performance is used. We refer to the process as tuning.

Adding new operators to Hidet does not require high engineering effort. Most operators in Hidet are scheduled automatically through rule-based scheduling and are easy to add. The computation-intensive operators like convolution and matrix multiplication usually require template-based scheduling for high performance. The complexity of adding a new Hidet template is similar to that of the AutoTVM [11] template.

5.2 Post-Scheduling Fusion

To alleviate the complexity of scheduling a sub-graph as in AutoTVM, we propose to decouple the sub-graph scheduling into two stages: (1) scheduling the anchor operator and (2) fusing the scheduled tensor program with surrounding operators. The decoupling allows developers to focus on the scheduling of the anchor operator instead of the whole sub-graph, and automates the fusion of the scheduled tensor program with other operators in the sub-graph. During tuning, the performance of fused tensor programs will be used as the target to maximize, thus the decoupling does not hurt the final performance.

Figure 15 shows an example of post-scheduling fusion. In step 1, during the graph-level optimization stage, an optimization pass partitions the computation graph into sub-graphs. Given the sub-graph, in step 2, a selected anchor operator will be scheduled into a tensor program with one of the scheduling mechanisms in Section 5.1.3. Finally, in step 3, the remaining operators will be fused into the scheduled program. These operators are classified into two categories: prologue operators for each input tensor and...
epilogue operators for each output tensor. Each prologue operator defines how each element access for the input tensor is computed, and the epilogue operator defines how the output tensor elements are furthermore computed and stored in the output of the fused operator. In this example, the access of \( A[99 - i] \) will be replaced by \( C[99 - i] \times 2.0 \), and the \( i \)-th element of output tensor is furthermore computed (i.e., multiply by \( 3.0 \)) and stored to the fused output tensor \( D \) with indices \((i / 50, i \ % 50)\).

The post-scheduling fusion simplifies the operator fusion. It also allows us to reuse existing highly optimized operators (e.g., matrix multiplication) to support new operators (e.g., convolution). In Hidet, we can implement the convolution operators as four operators with img2col algorithm [8], one of which is matrix multiplication and the other three are simple transform operators. With post-scheduling fusion, we fuse the other three operators into a matrix multiplication and reuse all optimizations (e.g., parallel reduction on \( k \) dimension [32]) for matrix multiplications to convolutions.

6 EVALUATION

6.1 Experimental Setup

Implementation. We implement Hidet from scratch with \( \sim 20K \) lines of code in Python and C++. Two levels of IR are used in Hidet: graph-level IR to represent the computation graph of DNN models and tensor-level IR to represent tensor programs with schedules. Hidet lowers the tensor program written with task mappings to CUDA C code and compiles it with the CUDA compiler. Notably, we only implement two efficient schedule templates for matrix multiplication and reduction operators (e.g, sum reduction) to cover all operators in evaluated models. Most operators are either scheduled by the rule-based scheduling mechanism or converted to matrix multiplication to reuse existing templates (e.g., convolutions).

Platform. We conduct experiments on a server equipped with a 16-core 24-thread Intel i9-12900K CPU (with hyper-threading enabled), 64 GiB DRAM, and one NVIDIA RTX 3090 GPU. The server has installed the Linux distribution Ubuntu LTS 20.04 with NVIDIA driver 510.73.08 and CUDA 11.6.

Workloads. We benchmark on a wide range of representative networks to demonstrate the optimization generality of Hidet. ResNet-50 [25] is one of the most commonly used CNNs for image classification. Inception-V3 [30] is a CNN that employs multiple paths of convolutions with different kernel sizes. MobileNet-V2 [45] is a lightweight CNN based on separable convolutions. Bert [16] is a widely-used transformer-based natural language processing (NLP) model. GPT-2 [42] is an NLP model targeting sequence-to-sequence tasks such as natural language translation and question answering. We use 128 as the sequence length for the two language models throughout the experiments. We adopt the model implementations in torchvision and transformers packages and export them to ONNX [5] format for evaluation.

6.2 End-to-End Evaluation

We evaluate all workloads on Hidet against PyTorch [41] 1.11, Onnx Runtime [15] 1.11.1, AutoTVM [11] and Ansor [65] in TVM [9] 0.9.dev with commit c07a46327. PyTorch is a widely used DNN framework. Onnx Runtime is a high-performance inference engine. Both of them leverage high performance kernel libraries cuDNN [12] and cuBLAS [26]. AutoTVM and Ansor are two state-of-the-art schedulers based on loop-oriented scheduling and input-centric tuning spaces. We set the number of tuning trials in AutoTVM and Ansor to 1000 and 800, respectively, as suggested in their paper and official documentation.

Performance. Figure 16 shows the results of end-to-end inference latency with a single batch. Hidet outperforms all baselines on most models by up to 1.48×, and on average by 1.22×. This is because Hidet is able to automatically fuse sub-graph, tune the schedule for given input size (vs. PyTorch and Onnx Runtime), and export more optimizations such as double buffering [32] (vs. AutoTVM and Ansor). One exception is Ansor on MobileNetV2, as Ansor could find a better schedule for depthwise convolutions. We can implement similar schedules in Hidet, and we leave such implementations to future work. In addition, we note that AutoTVM performs worse on both Bert and GPT-2 models with 27ms and 41ms, respectively. This is because AutoTVM’s schedule templates for workloads in these two models lack optimizations.

Tuning Cost. We compare the tuning cost (i.e., elapsed time in the tuning process) of AutoTVM, Ansor, and Hidet in Figure 17. Hidet reduces the tuning cost by 11× and 20× compared with Ansor and AutoTVM, respectively. This is because Hidet adopts a small (e.g., 180 schedules in matrix multiplication) but efficient schedule space with the proposed paradigm. As a result, Hidet only needs minutes to exhaustively enumerate all candidates. On the other hand, AutoTVM [11] and Ansor [65] adopt schedule spaces with \( 10^5 \) to \( 10^6 \) candidates, which prevents them from finding the optimal schedule in their space in a short time, even equipped with a cost model. Note that although AutoTVM only spends 2 minutes for Bert and GPT-2 due to their small schedule spaces with less than 20 schedules, the schedule spaces are ineffective and can not achieve competitive performance (Figure 16).

6.3 Case Studies

In this subsection, we conduct several case studies to further de-mystify the effectiveness of Hidet. 6.3.1 Schedule Space Comparison. To compare the efficiency of three schedule spaces adopted by AutoTVM, Ansor, and Hidet, we depict the latency distribution of schedules in the three schedule spaces in Figure 18. The benchmark workload is a convolution in ResNet50 with batch size 1, input image size 28x28, input channels 256, kernel size 3, padding 1, and stride 2. Because the schedule spaces of AutoTVM and Ansor are too large, we take the 1000 and 800 schedules from the tuning process of AutoTVM and Ansor, respectively, as the samples in their schedule spaces. We compare them with the entire space with only 180 schedules in Hidet schedule space. The figure shows that most schedules covered by Hidet schedule space have superior performance (latency < 75\( \mu \)s) than those in spaces adopted by AutoTVM and Ansor thanks to the better expressiveness of the proposed paradigm.

6.3.2 Performance Sensitivity over Input Sizes. The quality of the final schedule derived from AutoTVM and Ansor is sensitive to the input size due to their input-centric schedule spaces. Even a small change in the input size would result in a large performance difference. To compare the performance sensitivity over input sizes,
we benchmark matrix multiplications with consecutive input sizes. Figure 19 shows that the performance of AutoTVM and Ansor fluctuates significantly. Even worse, for a prime number input size (e.g., 2039), both schedulers failed to find a valid schedule. On the other hand, with the hardware-centric schedule space, Hidet achieves consistent performance on these input sizes.

6.3.3 Evaluation on Different Batch Sizes. Figure 20 depicts the latency of ResNet50 with different batch sizes. When batch size is small (1 and 4), AutoTVM and Ansor outperform Onnx Runtime as they can find schedules that utilize the GPU computation resources well (e.g., enough thread blocks to saturate all SMs), while kernel libraries do not. At larger batch sizes (e.g., 8), we observe that although AutoTVM and Ansor can still find schedules that saturate all SMs, they cannot outperform Onnx Runtime, because the latency of each thread block is longer than Onnx Runtime’s, due to the lack of important optimizations such as double buffering [32]. On the other hand, Hidet outperforms all of them as Hidet could perform well on both aspects (i.e., enough and efficient thread blocks).

6.3.4 Post-Scheduling Fusion Evaluation. With post-scheduling fusion, we can implement an operator with a highly optimized schedule template, and composite new operators with pre-implemented,

Figure 16: End-to-end comparison between state-of-the-art DNN inference frameworks and compilers with Hidet.

Figure 17: Tuning cost of AutoTVM, Ansor, and Hidet.

Figure 18: Schedule latency distribution of schedule spaces from AutoTVM, Ansor, and Hidet. X-axis is in log scale.

Figure 19: Comparison of AutoTVM, Ansor, and Hidet on matrix multiplication with consecutive input sizes.
highly optimized operators to save engineering efforts. For example, in Hidet, we implement convolution through matrix multiplication, namely implicit general matrix multiplication (GEMM) convolution, which is also known as img2col [8] algorithm. With post-scheduling fusion, we are able to fuse the additional required operators in img2col into the matrix multiplication automatically and reuse the optimizations we implemented for it (e.g., parallel reduction on k dimension [32]). The implicit GEMM convolution with parallel k reduction allows Hidet’s generated kernels to saturate the GPU computation resources and outperforms the existing kernel libraries and DNN compilers. Figure 21 shows the performance of the Conv–Bn–ReLU sub-graphs in ResNet50 among Onnx Runtime, Anso, and Hidet. Hidet outperforms Onnx Runtime and Anso on most convolutions as the convolution can also parallelize on the reduction dimensions (e.g., input channels, and kernel sizes).

6.3.5 Comparison with TensorRT. We also compare Hidet with TensorRT [27] 8.4.1.5, a high-performance deep learning inference engine provided by NVIDIA. TensorRT applied both graph-level and operator-level optimizations. Figure 22 shows the comparison of TensorRT and Hidet. Hidet outperforms TensorRT on the three CNNs because Hidet is able to tune for the given input sizes and fuse operators automatically according to their mathematical definition. On the other hand, TensorRT outperforms Hidet on the transformer [55] networks such as Bert and GPT-2. Since TensorRT is close-sourced, we speculate, by interpreting its optimization log, that TensorRT recognizes self-attention layers in transformer models and applies dedicated optimizations due to the popularity of these models. On the other hand, Hidet only has two schedule templates to cover all operators in benchmarked networks.

7 RELATED WORK

Many existing DL compilers adopt loop-oriented scheduling primitives [9, 43] and establish auto-tuning frameworks on top of them [2, 11, 46, 54, 57, 58, 63, 65–67] with input-centric schedule spaces. In contrast, Hidet leverages task-mapping programming paradigm with hardware-centric schedule spaces, so that it is able to achieve better performance with a much shorter tuning time. In addition to loop-oriented scheduling, there are more approaches to optimize a tensor program. Deep learning frameworks such as PyTorch [41] and TensorFlow [1] leverage off-the-shelf kernel libraries (e.g., cuDNN [12] and cuBLAS [26]) as well as hand-crafted kernels to cover widely used operators. CUTLASS [32] is an open C++ template library with efficient matrix multiplication kernels on CUDA. Tiramisu [4] and AKG [62] employ the polyhedral model to schedule the tensor programs. Roller [69] constructs the tensor program with a bottom-up approach and aligns the tile sizes with hardware specifications. AI-Template [59] employs source-code level templates to construct tensor programs, which supports more fine-grained optimizations but sacrifices the flexibility of program transform. TVM community also noticed the limited expressiveness problem of the existing declarative loop-oriented scheduling mechanism. TensorIR [22], a concurrent work with Hidet, is recently proposed to allow developers to directly write tensor programs instead of applying a series of declarative primitives to the auto-generated tensor program. Moreover, XLA [44] is a domain-specific compiler for linear algebra. FreeTensor [51] and CoRa [20] study the compilation for irregular or ragged tensor programs. ASTitch [68] and Apollo [61] study the fusion of memory-intensive kernels to reduce memory consumption. Fireiron [24] proposes a data-movement-aware scheduling language for GPUs. Triton [52] proposes to write tensor programs by taking tile as the basic data type and thread block as the main parallel processing unit. Nimble [47], DISC [70], Cortex [21], and DietCode [63] study the compilation of dynamic models, which is also orthogonal with Hidet. Besides optimizing every single operator for DNN inference, Rammer [38] and ICS [18] propose to parallelize independent operators in a network. TASO [30], Fang et al. [19], TENSAT [50], and PET [56] apply auto-generated rewriting rules to optimize DNN at the graph level. Checkmate [29], Chen et al. [10], Echo [64], and DTR [33] are proposed to reduce memory footprint. These works are orthogonal to Hidet, and can be used to enhance different aspects of Hidet (e.g., the graph-level optimizations, memory consumption, and dynamic-shape support).

8 DISCUSSION

Optimization Expressiveness. The accelerators (e.g., GPUs and TPUs) usually have a hierarchical memory system and vector- or tensor-based computation engines. Both demand dedicated optimizations to achieve peak performance, and these optimizations are usually hard to be expressed through a series of loop transformations. The double buffering example we discussed in this paper is a good example of such a challenge. Instead of relying on a declarative style scheduling mechanism, Hidet proposes to directly express the task assignment and ordering with task mapping in a tensor program, which greatly increases the expressiveness of Hidet while reducing the complexity of tensor program writing.

Support More Hardware. Although we only focus on GPUs in this work, the concept of task mapping is general and can be used to describe the task assignment and ordering for other processors. The worker in a task mapping can be (1) iterations in a loop for a single-core CPU, (2) CPU threads for a multi-core CPU, (3) threads, warps, or thread blocks for a GPU, and (4) parallel processing units in other accelerators. And the tasks of a task mapping could be arbitrary indexed, homogeneous, and parallelizable operations.

Future Work. We plan to support CPU and other accelerators (e.g., Amazon Inferentia and Trainium) in the future. Besides this, we also plan to support training. Due to the long tuning time of TVM, it is hard to be directly used for accelerating training. Thanks to the hardware-centric schedule space adopted by Hidet, the tuning time has greatly been reduced for Hidet, which makes it possible to build a training system based on Hidet.
9 CONCLUSION
We observe that the state-of-the-art DNN compilers based on loop-oriented scheduling cannot express important optimizations that require fine-grained manipulation of the tensor program. To address this limitation, we propose task-mapping programming paradigm, a new paradigm to write and schedule tensor programs that simplifies tensor program writing and scheduling without sacrificing the ability to express optimizations as in kernel libraries. Based on this paradigm, we implemented a new DNN inference framework called Hidet. Experiments show that Hidet achieves up to 1.48x speedup (1.22x on average), compared with state-of-the-art DNN inference frameworks (e.g., Onnx Runtime) and compilers (e.g., TVM equipped with AutoTVM and Ansor). Hidet also reduces 11x tuning cost compared with Ansor.

ACKNOWLEDGEMENT
We would like to thank the members of EcoSystem research laboratory in University of Toronto for their feedback on the early manuscript, and special thanks to Xingyang Song, Christina Giannoula, Anand Jayarajan, and Jiacheng Yang. We also want to thank the anonymous ASPLOS reviewers for the valuable feedback and suggestions, and the artifact evaluation reviewers for reproducing our experiments. The authors with University of Toronto were supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

A ARTIFACT APPENDIX
A.1 Abstract
This appendix helps readers to reproduce all experiments in the evaluation section via the Hidet artifact [17]. In Section 6, there are 6 experiments (one end to end experiment and 5 case studies). These experiments compare Hidet with other DNN frameworks and compilers on representative DNN models from the perspective of execution latency, optimization time, schedule space, input sensitivity, and different batch sizes. In the public artifact, we provide scripts to launch the 6 experiments automatically. With the hardware and software described in Section A.3.2 and A.3.3, the artifact should reproduce all experimental results in the evaluation section.

A.2 Artifact Checklist
- **Compilation**: NVIDIA CUDA compiler (nvcc).
- **Model**: ResNet50, InceptionV3, MobileNetV2, Bert, and GPT-2.
- **Run-time environment**: Linux Ubuntu 20.04+.
- **Hardware**: A workstation equipped with Intel Core i9-12900K, NVIDIA RTX 3090, and 64 GiB RAM.
- **Metrics**: End-to-end inference latency and auto-tuning time.
- **How much disk space required (approximately)**?: 2 GiB.
- **How much time is needed to prepare workflow (approximately)**?: 2 hours.
- **How much time is needed to complete experiments (approximately)**?: 60 hours. Most of the time (about 50 hours) will be used for model tuning by baselines AutoTVM and Ansor.
- **Publicly available**: Yes.

A.3 Description

A.3.1 How to access. The source code can be downloaded from either the Zenodo archive (https://doi.org/10.5281/zenodo.7429879) or GitHub repository (https://github.com/yaoyaoding/hidet-artifacts).

A.3.2 Hardware dependencies. To get the exact numbers in the evaluation, the exact CPU and GPU is required: Intel Core i9-12900K CPU and NVIDIA RTX 3090 GPU. To functionally run the experiment, the only requirement is a modern NVIDIA GPU that supports CUDA 11.6+.

A.3.3 Software dependencies. The artifact requires:
- NVIDIA Driver 510.73.08
- NVIDIA CUDA Toolkit 11.6 [39]
- NVIDIA kernel library cuDNN 8.4 [12]
- Package torch 1.11 (PyTorch [41])
- Package torchvision 0.12 (CNN models [25, 45, 50])
- Package transformers 4.19.2 (NLP models [16, 42])
- Package onnxruntime-gpu 1.11.1 (ONNX Runtime [15])
- Package nvidia-tensorrt 8.2.5.1 (Tensor RT [27])
- Apache TVM 0.9.dev with commit c87a46327 [9]

A.3.4 Models. We conduct the experiments with five DNN models: ResNet50 [25], InceptionV3 [50], MobileNetV2 [45], Bert [16], and GPT-3 [42]. The three convolution networks are from torchvision model zoo, and the two transformer models are from transformers package. All of them will be automatically downloaded.

A.4 Installation
Download the source code or clone the git repository in section A.3.1. Follow the commands of the installation section in README.md file under the root of source code directory to build and install hidet and baselines.

A.5 Experiment Workflow
There are 6 sub-directories under hidet/artifacts directory starting with 0, 1, 2, 3, 4, and 5, corresponding the 6 experiments in Section 6.2, 6.3.1, 6.3.2, 6.3.3, 6.3.4, and 6.3.5. Each sub-directory contains a python script main.py that can be directly launched to conduct corresponding experiment.

A.6 Evaluation and Expected Results
Each experiment script would have multiple outputs like

<table>
<thead>
<tr>
<th>BatchSize</th>
<th>Model</th>
<th>Executor</th>
<th>Latency</th>
<th>Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>resnet50</td>
<td>hidet</td>
<td>1.329</td>
<td>0.000</td>
</tr>
</tbody>
</table>

that represents the average latency of one executor on a model with a specific batch size in multiple runs. This example shows that it takes Hidet 1.329 ms on average (with standard deviation 0.000 ms) to run a single batch of ResNet50 [25] model. Some column are omitted here for simplicity. When conducting the experiments with the hardware and software described in Section A.3.2 and Section A.3.3, the artifact should reproduce all experimental results in each evaluation section.