Effectively Scheduling Computational Graphs of Deep Neural Networks toward Their Domain-Specific Accelerators

Jie Zhao
Information Engineering University

Siyuan Feng
Shanghai Jiao Tong Univerisity

Xiaoqiang Dan, Fei Liu, Chengke Wang, Sheng Yuan, Wenyuan Lv, Qikai Xie
Stream Computing Inc.

Abstract

Fully exploiting the computing power of an accelerator specialized for deep neural networks (DNNs) calls for the synergy between network and hardware architectures, but existing approaches partition a computational graph of DNN into multiple sub-graphs by abstracting away hardware architecture and assign resources to each sub-graph, not only producing redundant off-core data movements but also under-utilizing the hardware resources of a domain-specific architecture (DSA).

This paper introduces a systematic approach for effectively scheduling DNN computational graphs on DSA platforms. By fully taking into account hardware architecture when partitioning a computational graph into coarse-grained sub-graphs, our work enables the synergy between network and hardware architectures, addressing several challenges of prior work: (1) it produces larger but fewer kernels, converting a large number of off-core data movements into on-core data exchanges; (2) it exploits the imbalanced memory usage distribution across DNN network architecture, better saturating the DSA memory hierarchy; (3) it enables across-layer instruction scheduling not studied before, further exploiting the parallelism across different specialized compute units.

Results of seven DNN inference models on a DSA platform show that our work outperforms TVM and AStitch by 11.15× and 6.16×, respectively, and obtains throughput competitive to the vendor-crafted implementation. A case study on GPU also demonstrates that generating kernels for our sub-graphs can surpass CUTLASS with and without convolution fusion by 1.06× and 1.23×, respectively.

1 Introduction and Background

Due to the slowing down of Moore's Law, moving to DSAs is acknowledged as promising to meet the keen desire of DNNs for computing power [24]. After several years of investigation on accelerators specialized for DNNs [7,8,15,22,25,29,32,55,58], a commonly used DSA abstraction depicted on the left of Fig.1 has been formed for this application domain, based on which most existing DNN accelerators are manufactured.

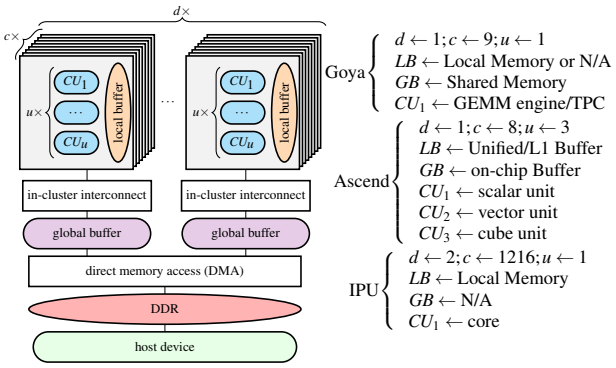


Figure 1: DNN DSA and its vendor customization.

We take the Habana Goya accelerator [32], the customized configuration of which is shown on the right of Fig.1, as an example to explain this abstraction. It is composed of $d = 1$ cluster, each including $c = 9$ cores that contains $u = 1$ compute unit (CU) for different DNN tasks. CUs are either a tensor-processing core (TPC) with a scratchpad local buffer (LB) or a general matrix multiplication (GEMM) engine with no LB. Cores are connected using an in-cluster interconnect mechanism, equipped with a scratchpad global buffer (GB). Clusters, if $d > 1$, are stacked, communicating data with DDR via DMA. We also show the customized configurations of a Huawei Ascend 910 platform [29] and a Graphcore IPU device [22] in Fig.1. The Graphcore IPU uses the term "tile" to denote a core and its unique LB. Hardware architecture of others [7,8,15] can also be deduced according to the abstraction in Fig.1.

Hence, effectively scheduling DNNs toward this abstraction is essential to exploit the computing power of DNN accelerators for DSA compilers. Specialized for machining learning (ML) applications, these accelerators exhibit a scratchpad-based memory hierarchy and parallelism across both multiple cores and several CUs, but prior work [5,13,31,54] devised to schedule DNNs on these DSA accelerators did not consider hardware architecture when partitioning a computational graph of DNN, introducing redundant off-core data move-

ments (*i.e.*, between LB and GB/DDR) and under-utilizing both faster local memory and parallelism across CUs.

1.1 Concepts and Notations

To explain the issues of prior work, we first introduce computational graphs, which are used by existing ML frameworks [1, 38] to represent DNN models. Fig.2 is an example. As it can contain a large number of nodes, each of which performs a computation task on several tensors, a computational graph usually references memory footprints that exceed the local memory capacity of its target platform and thus cannot be scheduled as a whole. Existing schedulers first partition it into sub-graphs and next assign resources to each sub-graph. A sub-graph, which is also known as a fused operator (*op*), is first initialized by a graph node and next grouped according to its producer-consumer relations with others, implemented as a kernel function or kernel executable on target platform.

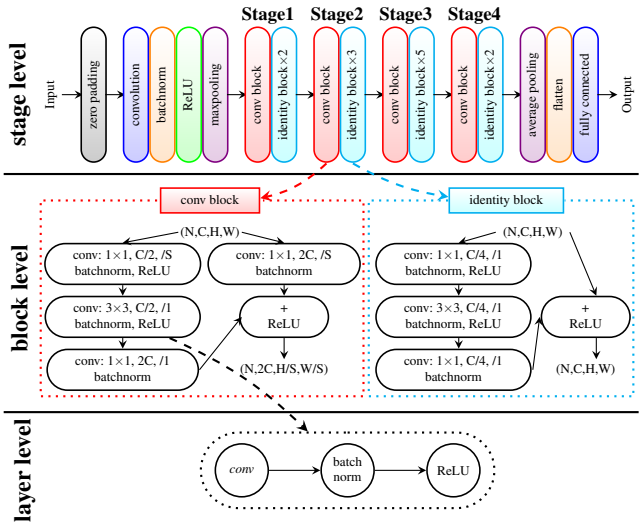


Figure 2: Computational graph of ResNet-50 [16]. A node (a circle) represents an *op*, and an edge (a solid arrow) denotes a producer-consumer relation of two *op*s. An *op* is a function that takes as inputs one to several tensors and outputs another. At the bottom is a 3×3 convolution (*conv*) layer composed of three nodes. A dashed arrow connects a stage/block with its internal structure; a dotted box denotes a block. Other layers can be expressed in a similar way.

We use the term "layer" to denote a set of nodes connected in a straight-line manner, at most one out of which contains parameters that should be learned using the gradients of the loss. A graph node represents an *op*, which is traditionally referred to as a neural layer in neural networks. Some neural layers, however, do not require parameters to be learned (*e.g.*, ReLU) or learned without using the gradients of the loss (*e.g.*, batch normalization) during the training process, and they can thus be considered as the auxiliary *op*s of those that indeed require parameters, *e.g.*, the convolution. We define layers as such because this definition summarizes the *op* fusion rules widely used by existing compilers [5, 53].

We also use the term "block" to represent an individual layer or a component composed of multiple layers that is recursively used in a DNN computational graph. For instance, the *conv* block composed of five layers is used once in each stage of Fig.2, while the identity block is used multiple times within each stage.

The term "stage" is a logical, high-level abstraction used in the architecture of the ResNet-50 model, taking the results of its preceding stage as inputs and generating output tensors. It is used to simplify the design of the network architecture but usually not considered by optimizing compilers.

1.2 Challenges of Prior Work

By obscuring hardware architecture, prior work [5,19,54] constraints sub-graph grouping within a layer [39] (the bottom level of Fig.2) and produces fine-grained sub-graphs. As each sub-graph is implemented by one kernel, prior work produces more kernels and **requires more off-core data movements between kernels**. Going one level upper in Fig.2 can observe five and four layers in the *conv* and identity blocks, so the entire network may require hundreds to thousands of kernels and off-core data movements of the same order of magnitude [21], resulting in high pressure on the limited memory bandwidth of a DSA platform. In addition, managing such a large number of off-core data movements for DSA is non-trivial because, unlike a general-purpose architecture reinforced by its mature hardware mechanisms, the hierarchical scratchpad memory of the later is still controlled by hand or software [37].

Even though managing the data movements across a DSA's memory hierarchy is possible, generating fine-grained sub-graphs still **misses the across-layer instruction scheduling opportunities**. Once formed, each sub-graph is lowered to a loop nest pipeline, to which memory optimizations and loop transformations like tiling and fusion are applied to better utilize hardware resources. While their different compositions constitute the search space that existing autotuners [2,6,26,57] navigate to select the optimal scheduling, across-layer instruction scheduling opportunities, *e.g.*, overlapping the memory promotion statement of weights and the computation task of a 3×3 *conv* layer with those of its preceding 1×1 *conv* layer in Fig.2, are not covered by such spaces.

Finally, since **the imbalanced memory usage distribution**, which refers to a phenomenon where memory usages vary across network architecture [30], **is not exposed/exploited**, the above scheduling paradigm also **under-utilizes the faster local memory of DSA**. On the top of Fig. 2, ResNet-50 is partitioned into four stages, each composed of one *conv* block and two or more identity blocks. A *conv* block converts its input with shape $[N,C,H,W]$ into $[N,2C,H/S,W/S]$, performing a down-sampling operation when $S > 1$, but an

identity block does not change its tensor shapes. The memory usage of stage1 is $\frac{S^2}{2}\times$ larger than that of stage2, and this property also exists in stage3 and stage4. If the faster memory of the target DSA is saturated when executing stage1, it will be under-utilized when executing the remaining stages.

1.3 Our Solution and the Organization of the Paper

To address the aforementioned issues on DSA platforms, we introduce a novel scheduling approach in this paper. First, as redundant off-core data movements are caused by fine-grained sub-graphs produced by existing tools [5, 13, 19, 44], our approach has to construct coarser-grained sub-graphs that can generate larger kernels, which can change massive output tensors originally exchanged through GB/DDR of Fig.1 into intermediate tensors that can stay in LB of the later, thereby converting many off-core data movements between kernels into on-core data exchanges within kernels. Second, to enable across-layer instruction scheduling outside the search spaces of prior work [31,57,59], a sub-graph constructed by our work must be able to group layers or even blocks like those of Fig.2, thus better hiding memory latency and exploiting the parallelism across CUs. Finally, to saturate the faster local memory of a DSA platform in the presence of imbalanced memory usage distribution [30], our method should consider the internal relations between coarser-grained sub-graphs such that a better scheduling order can be obtained.

With these considerations in mind, we design and implement a novel scheduling approach–GraphTurbo. §2 exemplifies the core idea and presents the overview of GraphTurbo. §3 explains how GraphTurbo constructs, splits and orders coarse-grained sub-graphs, to the results of which §4 generates larger kernels. §5 reports the experimental results of seven DNN inference models on a DSA platform, which demonstrate that, while achieving performance close to the vendor-crafted implementation, GraphTurbo outperforms TVM [5] and AStitch [60] by 11.15× and 6.16×, respectively. A case study on GPU also shows that GraphTurbo can surpass CUTLASS [27] with and without *conv* fusion by 1.06× and 1.23×, respectively. Finally, §6 discusses the related work, and §7 concludes the work.

1.4 Contributions

In summary, this paper makes the following contributions.

- We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.
- This synergy reduces off-core data movements, better saturates the valuable local memory, and empowers across-layer instruction scheduling.
- We design and implement a novel scheduling approach GraphTurbo, addressing the deployment of DNNs on DSA chips and offering insight to other platforms.
- The experimental results demonstrate that GraphTurbo can outperform two state-of-the-art tools and achieve performance comparable to the vendor-crafted code.

2 Core Idea and Overview

This section first explains the core idea of GraphTurbo and next presents its overview.

2.1 Exemplifying the Core Idea

We use Fig.2 that classifies a batch of input images into different categories as an example to explain our core idea. Data parallelism is exploited across the d clusters of Fig.1, which is always possible due to the multi-dimensional parallelism of tensors. Decomposing the input tensors of a DNN model into d clusters can be achieved by splitting one or multiple parallelizable dimensions. We assume the batch dimension of size $N = 32$ is split across these clusters, so each cluster processes $n = 8$ images that have been offloaded to GB of Fig.1.

Our work studies how a DNN model is scheduled within one cluster. **The core idea is to maximally preserve the input tensors in LB in order to convert as many off-core data movements as possible into on-core data exchanges**. For the sake of clarity, we reproduce the stages of Fig.2 in Fig.3a and assume that each stage performs a down-sampling operation with $S = 2$.

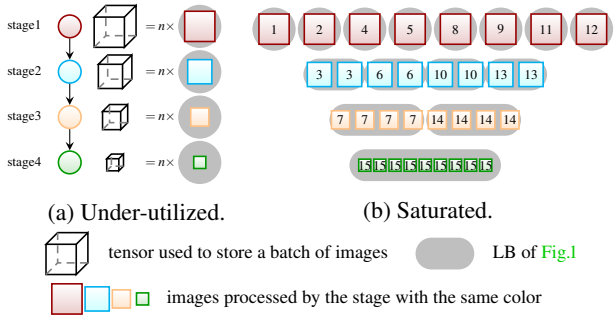


Figure 3: Utilization of LB under different scheduling methods. Timestamps in (b) define the scheduling of GraphTurbo.

Existing approaches [5, 13] can construct a sub-graph larger than a layer by grouping smaller ones, but they do not know when the grouping should terminate without hardware architectural information. Even though larger sub-graphs could be constructed for each stage of Fig.3a, these methods only schedule these sub-graphs according to their coarser-grained dependencies as the arrows show in Fig.3a, which produces a scheduling strategy that distributes each sub-graph of a stage onto the c cores of Fig.1. Suppose that the batch dimension is

split, then each core processes one image. If an image saturates LB when executing stage1, it will under-utilize this local memory when its core executes the later three stages, since their preceding stages reduce the tensor size by 2×, 4× and 8×, respectively, by performing down-sampling operations.

GraphTurbo can easily construct large sub-graphs for each stage by synthesizing network and hardware architectures. It splits these large sub-graphs into eight, four, two, and one instance, respectively, converting the coarse-grained dependencies between large sub-graphs into fine-grained ones between sub-graph instances. By eliminating redundant fine-grained dependencies, GraphTurbo executes two instances of stage1's sub-graph at timestamp 1 and 2 in Fig.3b, saturating LB while exploiting the parallelism across cores by distributing other parallelizable dimensions across them.

Next, GraphTurbo executes one instance of stage2's sub-graph at timestamp 3, which processes two images, both in LB, as shown in Fig.3b, because the image size is decreased by 2×. LB is thus not under-utilized. The parallelism across c cores is exploited by distributing both the batch and other parallelizable dimensions. Readers can follow the timestamps to infer the scheduling order and find that LB is never under-utilized while fully exploiting the parallelism between cores. In particular, this scheduling approach achieves a 7.97× speedup over TVM on our experimental platform.

2.2 Overview of GraphTurbo

To obtain scheduling strategies similar to Fig. 3b, GraphTurbo takes in a computational graph simplified by some standard graph optimizations [13,23] and schedules it at graph level (§3). To construct coarser-grained sub-graphs, *e.g.*, for stages of Fig.3a, GraphTurbo first collects hardware information (§3.1) to guide its grouping process (§3.2) by synthesizing network and hardware architectures. These sub-graphs are then split into instances, which are sorted (§3.3) to achieve the scheduling order like in Fig.3b. How parallelism is exploited and load balance is guaranteed across multiple cores are then explained (§3.4), with core binding and memory scopes automatically inferred. A concatenation step is then used to collect the tensors of producer sub-graph instances (§3.5), followed by some generalization discussions (§3.6).

The graph scheduler produces ordered sub-graph instances, which are delivered to the kernel generator (§4), producing kernels by combining loop fusion (§4.1) and buffer stitching (§4.2), with memory allocation and reuse automatically managed (§4.3). For the example in §2.1, the graph scheduler concentrates on input images. The convolution weights of this model are getting larger but only used within layers and do not result in communications between stages. GraphTurbo only allocates a small, fixed size of buffers in LB to allow for the promotion of such tensors to local memory when handling each layer, and the overhead of such memory promotion statements is hidden behind the computation (§4.4).

3 Scheduling Sub-graph Instances

This section constructs coarser-grained sub-graphs and schedules their instances. To achieve this goal, we need to address five issues and thus organize this section into five steps, with the difficulties explained at the beginning of each.

3.1 Collecting Splitting Information

GraphTurbo relies on producer-consumer relations between sub-graphs to group them into larger ones. This strategy, however, does not know when to stop without knowing hardware architectural information. Hence, **this section first collects hardware knowledge for sub-graphs**. As it will also be used to split sub-graphs (§ 3.3), we refer to it as splitting information `SplitInfo`, which is a set of 4D tuples $(split_d, n_d, f_d, d)$. Algo.1 summarizes how to compute it.

Algorithm 1: Compute `SplitInfo`

```
1  SplitInfo ← ∅;
2  foreach d in [1,··· ,depth ←dimof (output of SG)] do
3      n_d ← 0; split_d ← 0; f_d ← ∞;
4      foreach v in [1, 2, 4, 8, 9, ··· , size^(d)_output] do
5          if ⌈peak/v⌉ ≤sizeof (LB) then
6              n_d ← n_d + 1; split_d ← 1; f_d ← v; break;
7      foreach t in intermediates do
8          if split_d = 1 then
9              n_d ← n_d + num_of_op(t);
10             if match_dim(t, d) and size_t^(d) % f_d = 0 then
11                 SplitInfo ←SplitInfo ∪{(split_d, n_d, f_d, d)};
```

Before grouped, a sub-graph SG is a node that represents an *op*. When lowered, it may produce several loop nests since the *op* it represents can be complex such that multiple intermediate tensors are used to realize its function [53]. Except the last one that defines the output tensor, all remaining loop nests write to intermediate tensors. Algo.1 computes `SplitInfo` by first splitting the output tensor (lines 3-6) and next propagating its splitting information to each of intermediate tensors (lines 8-11) due to the following reasons.

First, a sub-graph has one output but its input tensors can be many. Considering only the output tensor simplifies the algorithmic design. Second, it is the output that determines how the loop nests of this sub-graph should be split or tiled [41,52]. Once the information of the output and intermediate tensors is determined, how input tensors should be split is also known.

Indeed, computing `SplitInfo` this way may introduce recomputation of intermediate or input tensors, which would be expensive when fusing multiple *conv* or matrix multiplication *op*s. We thus use a simple cost model to prevent excessive recomputations that offset the benefits brought by fusion.

$depth$ represents the loop nest depth of the output tensor. By iterating a loop dimension d from the outermost to inner (line 2), Algo.1 makes use of the parallelism across cores as early as possible. Next, Algo.1 defines three metrics (line 3),

$split_d$, f_d, n_d, that represent whether the current dimension d can be split, the splitting factor of this dimension, and the number of *op*s split by it, respectively.

v iterates the values of line 4 to instantiate f_d. We consider $size^{(d)}_{output}$ that denotes the loop extent of the current dimension d as the upper bound because $v > size^{(d)}_{output}$ does not split the current dimension d. The first three values decompose the dimension d into eight, four, and two cores, while guaranteeing load balance across them. the first three stages in Fig.3b are split this way. Values between $8 \leq v \leq size^{(d)}_{output}$ do not exploit the parallelism of the current dimension d across cores but parallelize other dimensions, with load balance across cores fully considered. The splitting of stage1 in Fig.3b is an example of this case. A value is used to instantiate f_d (line 6) if the size of memory footprints, $\lceil \frac{peak}{v} \rceil$, required by a sub-graph instance does not exceed the memory capacity of LB (line 5). *peak* is the size of memory footprints required by *SG*. n_d and $split_d$ are also updated accordingly. As a smaller v partitions *peak* into larger pieces, the v loop here is a greedy strategy.

t iterates each intermediate tensor (line 7). It takes in the dimension d and first inspects whether the dimension can split the output tensor (line 8). n_d is increased by the number of *op*s in t (line 9) if this condition is satisfied. In addition, the 4D tuple is added to `SplitInfo` (line 11) if the dimension d matches one loop dimension of t and the loop extent of the matched dimension $size^{(d)}_t$ is dividable by f_d (line 10).

By exactly computing `SplitInfo` for sub-graphs, GraphTurbo determines appropriate sizes for its generated kernels. As `SplitInfo` usually has several elements and each one encodes a loop dimension that can be split, GraphTurbo needs to select the best dimension for a sub-graph. We consider the following criteria for this issue. First, a loop dimension is better if it splits more *op*s. Second, a loop dimension with a smaller splitting factor is preferred since it tends to better saturate LB. Finally, a dimension with a smaller loop depth is considered superior since it exhibits outer parallelism and fewer communications. These criteria are modeled as computing the lexicographical maximum of an optimization problem

$$\text{lexmax}_{\forall d \in \texttt{SplitInfo}} (n_d, -f_d, -d) \quad (1)$$

where the order of the three metrics defines the priorities.

3.2 Grouping Sub-graphs

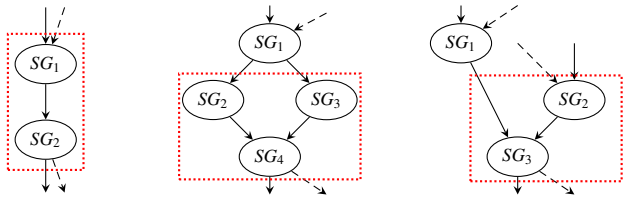
Now we can group sub-graphs. GraphTurbo still performs this step according to producer-consumer relations, but it **constructs larger sub-graphs by leveraging `SplitInfo` to determine the termination of grouping**, which is not restricted within layers [5, 19, 54]. Algo.2 outlines the process.

Algo.2 first sorts a computational graph G by topologically ordering all of its g nodes (line 1), each of which is treated as one sub-graph *SG* and delivered to Algo.1 to compute its

Algorithm 2: Group sub-graphs

```
1  SG[1,···,g] ←topological_order (G); b ← g;
2  foreach i in [1,···,g] do
3  |   SplitInfo[i] ← Algo.1 (SG[i]);
4  |   BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
5  repeat
6  |   {G,s} ←straight_merge(SG[1,···,b], SplitInfo[1,···,b]);
7  |   foreach i in [1,···,s] do
8  |   |   BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
9  |   {G,d} ←diamond_merge(SG[1,···,s], SplitInfo[1,···,s]);
10 |   foreach i in [1,···,d] do
11 |   |   BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
12 |   {G,b} ←branch_merge(SG[1,···,d], SplitInfo[1,···,d]);
13 |   foreach i in [1,···,b] do
14 |   |   BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
15 until s, d and b all do not decrease;
```

`SplitInfo` (lines 2-3). Next, Algo.2 considers three different merging patterns (lines 5-15) to group these sub-graphs, which reduces the number of sub-graphs from g to s, d, and b, respectively. The sub-graph index and `BestSplit` are updated each time a merging pattern is grouped.



(a) Straight. (b) Diamond. (c) Branch.

Figure 4: Merging patterns considered by GraphTurbo. A solid arrow is the producer-consumer relation, and a dashed one denotes a possible connection with another sub-graph.

The merging patterns considered by Algo.2 are defined as follows. First, two sub-graphs SG_1 and SG_2 form a straight pattern (Fig.4a) if and only if (iff) SG_1 is the unique producer of SG_2 and SG_2 is the unique consumer of SG_1. Second, four sub-graphs make a diamond pattern (Fig.4b) iff there are only one entry, one exit and at least two paths from the entry to the exit. The entry (SG_1)/exit (SG_4) can be a consumer/producer of multiple outside sub-graphs. Third, three sub-graphs constitute a branch pattern (Fig.4c) iff there exists only one exit and multiple paths to it, and SG_1 could be but not necessarily a producer of SG_2.

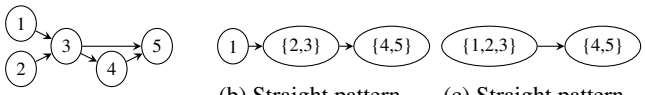
Instead of grouping all components of a merging pattern, Algo.2 fuses a subset of them into a larger sub-graph *SG*, as shown by a red dotted box in Fig.4. Algo.2 uses two heuristic rules to determine whether the grouping is allowed.

(i) SG_1 and SG_2 of Fig.4a (or SG_2 and SG_3 of Fig.4c) can be merged if two preconditions are satisfied. First, the splitting factor of SG_2 of Fig.4a (or SG_3 of Fig.4c) is no less than that of SG_1 (or SG_2). As a larger splitting factor partitions *peak* into smaller pieces, this precondition

prohibits the propagation of a sub-graph's large splitting factor to its followers in the case of down-sampling operations. Second, the splitting factor of SG is not larger than that of SG_2, ensuring that the grouping result does not deteriorate the utilization of LB exploited by SG_2. `SplitInfo` of SG can be computed using Algo.1.

(ii) SG_2, SG_3 and SG_4 of Fig.4b can be merged into SG if the splitting factor of SG is not larger than the maximum among the splitting factors of SG_2, SG_3 and SG_4, which is also used to guarantee good LB utilization.

We now explain how the *conv* and identity blocks of Fig.2 are grouped. First, Algo.2 identifies the straight patterns in each layer and obtains Fig.5a. Specifically, the three layers on the left of the *conv* block in Fig.2 is identified as a straight pattern, fused into one sub-graph denoted using label 1. Neither of the *conv* and the ReLU layers of this *conv* block is identified as a straight pattern; instead, they both form individual sub-graphs, represented using labels 2 and 3, respectively. Similarly, the three *conv* layers of the identity block constitute a straight pattern, depicted using the sub-graph with label 4; and its ReLU layer is denoted using label 5.



(a) Branch pattern. (b) Straight pattern. (c) Straight pattern.

Figure 5: Merging the *conv* and identity blocks of Fig.2.

Next, Algo.2 finds two branch patterns (*i.e.*, sub-graphs 3 and 5 in Fig.5a) and produces Fig.5b, the straight patterns of which are first merged into Fig.5c and finally form a single sub-graph {1, 2, 3, 4, 5}. The preconditions of the above two rules are all satisfied when merging these patterns. In practice, Algo.2 can also group the multiple identity blocks and a *conv* block into a single sub-graph, thus producing four large sub-graphs for stages in Fig. 3a. These large sub-graph are no longer grouped because the second precondition of Rule **(i)** is not satisfied.

3.3 Ordering Sub-graph Instances

The synergy between network and hardware architectures enabled by §3.1 and §3.2 partitions a computational graph into larger sub-graphs. In addition, GraphTurbo also uses `SplitInfo` to split each sub-graph into instances, **the order of which is determined in this section**.

For instance, each stage in Fig. 3a is converted into one to multiple labeled sub-graph instances with the same color, forming the new graph in Fig.6. All instances at the same horizontal level are homogeneous and thus can be executed in any other. The edges between these instances are inherited from sub-graphs, but the gray ones are redundant and easily eliminated. Specifically, we determine whether an edge is redundant or not by checking whether the output of a producer instance is used by one consumer instance. This is achievable because combining `SplitInfo` and the shape of an output tensor can perform this checking. The considered edge is redundant and eliminated if the checking result is true.

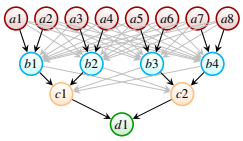


Figure 6: Sub-graph instances of the four stages in Fig.3a.

GraphTurbo currently uses two approaches to schedule a computational group G of sub-graph instances. First, it visits sub-graph instances in a breadth-first search (BFS), producing a schedule order shown in Fig.7a. Second, GraphTurbo visits them in a depth-first search (DFS), as described in Algo.3. We currently only use the BFS and DFS searches because they simplify the algorithmic design of GraphTurbo. As will be reported in §5, this choice achieves promising results. We are currently working on an integer linear programming approach to find a better solution than both of these search heuristics, which will be released soon.

Algo.3 first instantiates a list *visit* by visiting G in any orders (line 1) and next moves all of its sub-graph instances with no in-degrees from *visit* to another list *ready* in order (lines 3-4). The in- and out-degrees of *SGI* are only determined by the black edges in Fig.6. The last *SGI* with no in-degrees of *visit* (lines 6-8) is extracted and added into the ordered list *order* (line 9), with its consumers updated (lines 12-13) and moved from *visit* to *ready* (line 14).

Algorithm 3: Schedule Sub-graph Instances

```
1  visit ← get_subgraph_instances(G);
2  while visit ≠ ∅ do
3      foreach indegree(SGI) = 0 in visit do
4          ready ← ready.push(SGI); visit ← visit \ SGI;
5      while ready ≠ ∅ do
6          p ← sizeof(ready);
7          while indegree(SGI_p) ≠ 0 do
8              p ← p − 1;
9          order ← order.push(SGI_p); ready ← ready \ SGI_p;
10         foreach SGI in visit and ready do
11             if SGI_p is a producer of SGI then
12                 remove_producer(SGI, SGI_p);
13                 indegree(SGI)← indegree(SGI) −1;
14                 ready ← ready.push(SGI); visit ← visit \ SGI;
```

As an example, Fig.7 illustrates how the three lists change when Algo.3 is applied. In particular, as G can be visited in any order, we assume that it is visited in a BFS order for illustrative purpose. *order* is inspected from left to right. GraphTurbo finally selects the better one (Fig.3b) between the two results with respect to memory utilization. The labels

of each circle in Fig.7f would change if G is visited in other orders but the colors not, which does not matter because all instances of the same sub-graph are homogeneous.

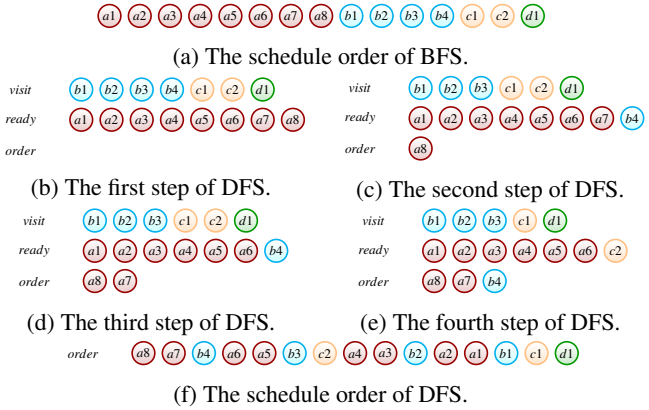


Figure 7: Different schedule orders of Fig.6. We only show the first four steps of Algo.3 for the sake of clarity.

3.4 Inferring Core Binding and Buffer Scopes

The next step is to **bind each ordered sub-graph instance to multiple cores**. A sub-graph instance's binding strategy can be determined by inspecting the loop dimensions of its output tensor. However, determining binding strategies individually may introduce more communications due to the mismatching between them. We use Algo.4 to infer binding strategies.

As Algo.4 can detect the mismatching between a pair of producer-consumer sub-graph instances, it also uses such information to **infer at which buffer scopes the output tensor of a sub-graph instance should be declared**. It first visits a schedule order O like Fig.7f in a reverse order and records all corresponding sub-graph instances in a list *visit* (line 1). Defined by default as an empty tuple [] and LB, $bind[i]$ and $scope[i]$ are used to record the binding strategy of a sub-graph instance and at which buffer level its output tensor is declared, respectively (line 2). As an output tensor is taken in by another sub-graph instance, we do not care about where the input tensors of a sub-graph instance should be declared. A sub-graph instance also produces intermediate tensors not considered by Algo.4, which GraphTurbo manages using its kernel generator (see §4.1).

Algo.4 infers binding strategies and buffer scopes for each sub-graph instance denoted by $visit[i]$ (lines 3-16). If $bind[i]$ is empty (*i.e.*, no information can be used for inference) or $scope[i]$ is not LB (*i.e.*, the known information cannot be used for inference) (line 4), Algo.4 instantiates $bind[i]$ using a plain strategy (line 5) and infers the binding strategy of the input tensors of the current sub-graph instance. A plain binding strategy is obtained by greedily allocating more cores from the outermost to inner loop dimensions of a tensor. The binding factors along multiple dimensions form a multi-dimensional

Algorithm 4: Infer Core Binding and Buffer Scopes

```
1  visit ← DFS_visit_reverse_order(O); size ← sizeof(visit);
2  bind[1,···,size] ← {[]}; scope[1,···,size] ← {LB};
3  foreach i in [1,···,size] do
4      if bind[i] = [] or scope[i] ≠ LB then
5          bind[i] ← plain_binding (output of visit[i]);
6          if infer_binding (bind[i]) = [] or is invalid then
7              continue;
8          foreach producer[j] in visit do
9              if bind[j] = [] then
10                 bind[j] ← infer_binding (bind[i]);
11             else if bind[j] ≠ infer_binding (bind[i]) then
12                 scope[j] ← GB;
13             else
14                 continue;
15     else
16         bind[i] ← update_binding (bind[i]) uses more cores than
             plain_binding (output of visit[i]) ? update_binding
             (bind[i]) : plain_binding (output of visit[i]);
```

tuple. Algo.4 tries to instantiate a binding factor by iterating integers 8, 4, 2, and 1, which guarantees load balance across cores. The iteration turns into its next value if a loop extent $size^d$ is not dividable by current one. Note that some tensors can have some specific requirements annotated to their loop dimensions, which cannot be bound to cores.

Next, Algo.4 uses **infer_binding** to infer the binding strategies of $visit[i]$'s input tensors, which are the output tensors of $visit[i]$'s producers (line 8). The inference is achieved by first matching loop dimensions of an input tensor as line 10 of Algo.1 does and next propagating the binding factors of the matched dimension from the output tensor. A binding strategy can be invalid if it does not satisfy the annotated requirements. Hence, Algo.4 falls into one of the following three cases when the inferred binding strategy is neither empty nor invalid: (1) $bind[j]$ is instantiated by the inferred binding strategy if it has not yet been defined (lines 9-10); (2) $scope[j]$ is overwritten by GB if the already defined $bind[j]$ does not match the inferred binding strategy (lines 11-12), since a communication is required here (see §3.5); (3) no mismatching between the already defined $bind[j]$ and the inference succeeds, and Algo.4 steps into next iteration (lines 13-14).

$bind[i]$ is inferred and not empty in the **else** case (line 15), for which Algo.4 tries to update $bind[i]$ by reconsidering the possibly annotated requirements of $visit[i]$'s output tensor and computes a plain binding strategy for it. The one using more cores between these two binding strategies is finally used to rewrite $bind[i]$ (line 16).

3.5 Concatenating Instance Outputs

As GraphTurbo splits a sub-graph into multiple instances, the output tensor of a sub-graph is also partitioned into multiple pieces. Hence, **this section introduces the step to concatenate the outputs of these sub-graph instances**.

To implement this step, GraphTurbo detects fine-grained

dependencies between sub-graph instances and introduces concatenation *op*s before each consumer of multiple producers, obtaining Fig.8 for the running example. A concatenation *op* is lightweight since its inputs and output stay in either LB or GB. GraphTurbo needs to insert additional *op*s for moving data across the memory hierarchy if the binding strategies and memory scopes of the tensors taken in by a concatenation *op* are different from each other and/or those of its output.

Fig.9 depicts two scenarios where such (gray) *op*s should be inserted. A sub-graph instance or an auxiliary *op* is denoted using an ellipse that displays the shape, scope and binding strategy of its output tensor. On the left, a copying *op* is introduced once mismatching between the memory scopes is captured, promoting data from its input (GB) to its output (LB). On the right, a redistribution *op* is inserted due to the difference between binding strategies, triggering a communication between cores. This can be achieved by resetting the tuple using the smaller binding factors along each dimension.

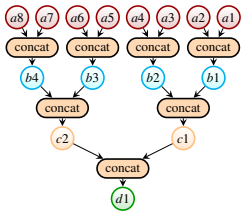


Figure 8: Concatenate instance outputs.

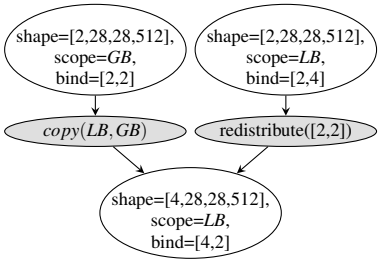


Figure 9: Insert data movement *op*s.

3.6 Generalizing the Approach

Now GraphTurbo can effectively schedule a DNN computational graph, but we made some assumptions in its design. This section discusses how they can be generalized. First, §3.1 assumes that a sub-graph has one output tensor, which is often the case in practice. One can repeat Algo.1 for each output tensor of a sub-graph that with multiple output tensors.

Second, the rules defined in §3.2 take into account down-sampling operations in a DNN model. However, our scheduler can also be generalized to target up-sampling operations by simply modifying Rule (**i**). A further split *op*, which can be considered as the opposite of the concatenation *op* introduced in §3.5 may be required to distribute the output of a sub-graph instance to its multiple consumers. We did not experiment with up-sampling operations in this work.

Third, §3.3 uses two methods to order sub-graph instances, which are simple but effective, as will be demonstrated in §5. We are now investigating another heuristic by allocating higher priorities to sub-graph instances with heavier memory footprints and plan to release it in the future.

Finally, GraphTurbo greedily uses LB, but, as a suggestion, it could be replaced by GB to schedule a computational graph across the d clusters of Fig. 1. It could also be substituted by faster memory of other platforms, *e.g.*, shared memory of GPU. Interestingly, much larger LB sizes would simplify the algorithmic flow of GraphTurbo. Even when splitting a sub-graph instance is unnecessary, GraphTurbo could obtain a similar scheduling to TVM but with across-layer memory optimizations (§4.4) fully considered. Moreover, making use of the higher-level buffer, *e.g.*, those residing in CUs of Fig.1 if any, is profitable, since data exchanges between such buffers and LB may dominate the communications in such cases.

4 Kernel Generation for Sub-graph Instances

Once scheduled sub-graph instances are obtained, the kernel generator can take each of them as input and lower them into loop nest pipelines. **The task of our kernel generator is to generate larger kernels by implementing loop transformations and stitching the intermediate tensor in the faster local memory of a DSA platform**. To minimize the engineering cost, we prototype our kernel generator in TVM [5], but it may fail to produce a single kernel for one sub-graph instance. Fig.10 exemplifies this issue by gradually expanding one sub-graph instance, *b*3, of Fig.8.

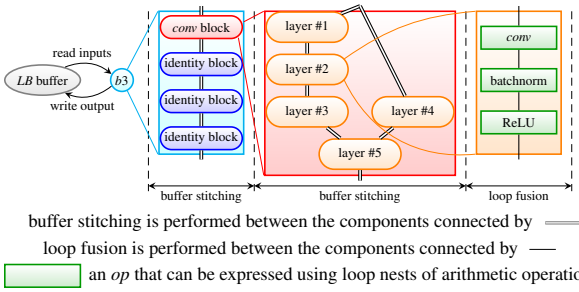


Figure 10: Expand *b*3 to generate a single kernel for it. *b*3 is sub-graph instance of stage2 of Fig.2. It is expanded to one *conv* block and three identity blocks, each of which is then expanded to multiple layers. How the *conv* block is expanded is shown in the middle, which obtains five layers shown on the left of the block level of Fig.2. These layers are labeled, and how layer #2 is expanded is shown on the rightmost, which produces the three *op*s shown at the layer level of Fig.2.

The rightmost part is what TVM's kernel generator takes in, but Fig.10 shows that *b*3 is composed of 38 *op*s (11 for the *conv* block and 9 for each identity block), out of which 13 are *conv op*s (4 for the *conv* block and 3 for each identity block). Performing loop fusion across multiple *conv op*s is outside the power of TVM's kernel generator. Although CUTLASS [27] was recently integrated into TVM to alleviate this problem for GPU [50], the number of acceptable *conv op*s is still limited. Furthermore, even if a similar vendor-crafted library can be offered on a DSA platform, the kernel generator would still put the output of a sub-graph instance in DDR, which in turn

would regress the benefit created by GraphTurbo.

4.1 Loop Fusion within Layers

The kernel generator can easily fuse *op*s within a layer. We use layer #2 in Fig.10 as an example and lower it to the loop nest pipeline shown in the middle of Fig.11. Since TVM requires users to write schedule templates for these loop nests, simply adopting its workflow cannot fully automate GraphTurbo.

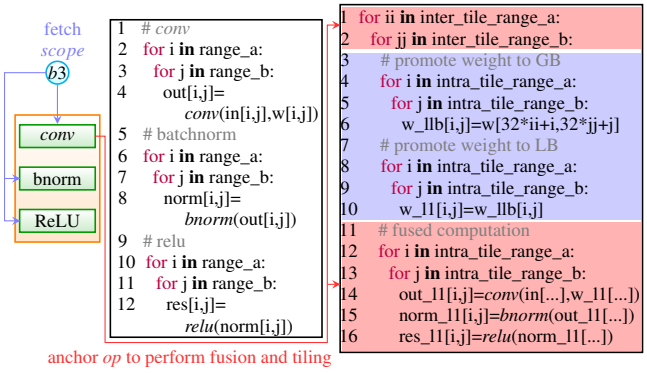


Figure 11: Loop fusion within a layer. Left: fetch the *scope* information of the sub-graph instance *b*3 in Fig.8. Middle: the loop nest pipeline of layer #2 in Fig.10. Right: the tiled and fused loop nest, with memory promotion statements of weights automatically inserted. Red arrows connect the anchor *op* with its tiled loop dimensions. Blue arrows represent that the *op*s fetch the *scope* information from Relay IR.

To resolve this issue, GraphTurbo selects an anchor *op* out of each layer and automatically performs loop tiling on this *op*. An anchor *op* should be set using either *conv*/matrix multiplication if the later appears in the current layer, or the last *op* of the layer otherwise. In the later case, the anchor *op* should be an elementwise *op*. The outermost two loop dimensions are selected for tiling because they are parallelizable in both cases. The tile sizes along the two dimensions of an anchor *op* are then selected in a similar way to that used to determine a plain binding strategy in §3.4, which greedily maximizes the memory utilization of LB. In other words, a tile size is instantiated using a largest integer that not only divides the current loop extent but also allows the resulted tiled tensors to stay in LB. Once the tile shape and sizes of the anchor *op* are determined, the loop bounds of other *op*s can be inferred and fused with the anchor *op*, just like what existing techniques [41, 52] did, producing the fused and tiled loop nest shown in the red regions of Fig.11.

By converting the tensors written by the *conv* and batch-norm *op*s into intermediate ones, this method automatically allocates them in faster memory, as mentioned in §3.4. Before doing so, the kernel generator fetches the *scope* information of the current sub-graph instance, *i.e.*, *b*3 in Fig.11, allocating intermediate tensors at the defined memory level. Memory promotion statements of a weight tensor are also automatically injected in the same way, as shown by the blue region of Fig.11. Note that some *op*s like batch normalization can be folded, but we keep it here for illustrative purpose.

4.2 Buffer Stitching across Layers/Blocks

After the internal of a layer is fused, we do not put its output back to DDR but still let it remain in LB, *e.g.*, *res_l1* in Fig.11. Hence, all layers of the *conv* block can exchange their data via LB, which we refer to as *buffer stitching* and the kernels used to implement these layers can be wrapped into one. The input tensors of the *conv* block are also put in LB, as declared by *scope* in §3.4. An identity block can be handled in a similar way. As the output tensors of each block's last layer also stay in LB, the four blocks can all be stitched together.

By targeting memory-intensive *op*s, AStitch [60] also implements a similar functionality. However, our work also considers compute-intensive *op*s. In addition, we also try to maximize faster local memory between sub-graph instances. By combining loop fusion and buffer stitching, our implementation generates a single kernel for the sub-graph instance *b*3. In contrast, TVM produces one kernel for each layer, increasing the number of generated kernels (65 in total) and thus requiring more off-core data movements.

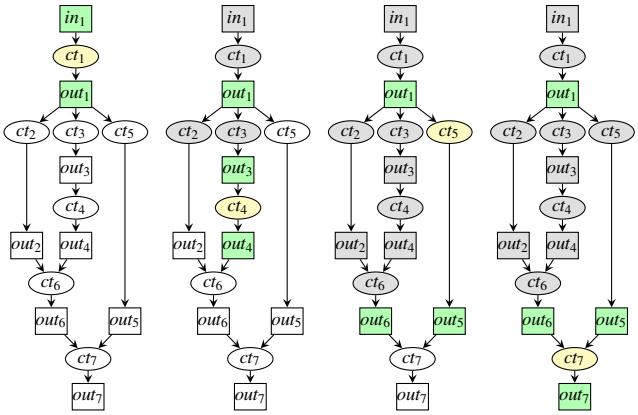
4.3 Memory Allocation and Reuse

The remaining task is to allocate space at the memory levels defined by *scope* for each tensor, which is trivial to implement. However, by putting many tensors in the faster local memory, GraphTurbo calls for a careful mechanism to reuse the limited LB's capacity. We always release the space consumed by an output tensor of a layer/block/sub-graph instance once it is no longer used. The space can thus be reused by other tensors. LB only needs to hold a limited number of tensors simultaneously. In case the total size of these tensors exceed LB's capacity, the one with the longest liveness across multiple computation tasks is spilled to first GB and next DDR. Fig.12 depicts the liveness of tensors across computation tasks.

Our heuristic is different from prior work [40, 46], which always spills the tensor with the largest memory size to lower memory hierarchy levels. Selecting the one with the longest liveness has a higher probability to spill a smaller tensor, which is likely to reduce the overhead of data movements.

4.4 Across-layer Instruction Scheduling

Combining loop fusion and buffer stitching not only produces a single kernel for a sub-graph instance, but also allows for overlaps of different layer computation tasks. On the right of Fig.11, promoting a weight is implemented by first copying its tensor from DDR to GB using DMA and next hoisting the tensor from GB to LB, which is possibly further dispatched



(a) Execute ct_1. (b) Execute ct_4. (c) Execute ct_5. (d) Execute ct_7.

Figure 12: Liveness of tensors across computation tasks. A (ellipse) computation task (*ct*) can be a sub-graph instance, a block or a layer. A (rectangle) tensor is live when colored in green or released if in gray. A *ct* is in execution if colored in yellow or finished when in gray. The space of tensor in_1 is released once it is not live, reused by out_3. out_1 is spilled to GB or DDR in case LB is insufficient to hold four tensors in (d), since it lives across seven *ct*s but others across fewer.

to individual CUs of Fig.1. The latency of these promotion statements can be hidden behind an earlier executed layer, and multiple CUs can execute computation tasks simultaneously.

Fig. 13 shows how this optimization is performed. A rounded rectangular represents a layer's tiled computation task. Across-layer memory latency hiding takes place between the two vertical lines, and CUs can execute different computation tasks of two tiles. Our approach significantly enhances the opportunities of such memory latency hiding and parallelism by increasing the optimization granularity to a degree beyond layers studied by prior work [5, 13, 31, 54].

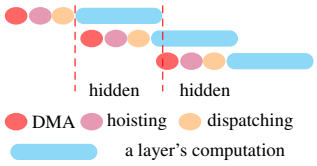


Figure 13: Across-layer instruction scheduling.

5 Experimental Results

We conduct experiments on STCP920 [51], an SoC DSA platform customizing the abstraction in Fig.1 using

$$\begin{cases} d \leftarrow 4; c \leftarrow 8; u \leftarrow 3 \\ LB \leftarrow 64 \text{ KB L1} \\ GB \leftarrow 8\text{MB last local buffer (LLB)} \\ CU_1 \leftarrow \text{vector core}; CU_2 \leftarrow \text{VME}; CU_3 \leftarrow \text{MME} \end{cases} \quad (2)$$

The eight cores connected bidirectionally using LLB. Each core has a 32-bit RISC-V CPU with vector extension, a vector MAC engine (VME) and a matrix MAC engine (MME) to handle different types of *op*s. As the target shares the common hardware abstraction of many existing DSA accelerators, one can expect similar results on other DSA platforms.

GraphTurbo resorts to LLVM v12.0.0 to compile its generated kernels on STCP920. The repository will be made publicly accessible soon. We experiment using ResNet-50 v1.5 [16], BERT [9], and DLRM [33], extracted from MLPerf v2.0 [43] and use their standard configurations. For BERT, we also consider three additional configurations. Other MLPerf models are not considered because they involve dynamically shaped tensors that GraphTurbo currently does not support. We also take into account MobileNet v2 [17], Vision_Transformer [11], DenseNet [18], and Conformer [14] that are not included in MLPerf. Except DLRM implemented using Pytorch v1.8.1 [38], all remaining models are implemented using TensorFlow v1.13 [1]. There are no fundamental reasons that limit the applicability of GraphTurbo to training models, which we intend to experiment in the future.

GraphTurbo is prototyped in TVM v0.8 [5], implemented using 19*k* Python, 44.2*k* C/C++ and 2*k* miscellaneous, among which the code used to implement the graph scheduling approach is about 7*k* lines. As our algorithms operate on computational graphs, it does not require much effort to target GraphTurbo to a new platform. What the engineers need to do is to feed these algorithms with necessary architectural information required, and a target platform should share the same properties as the DSA abstraction in Fig.1. The code to be changed should be lightweight in such cases.

5.1 Task Decomposition across Clusters

We first discuss how the optimal batch size is selected for a cluster of STCP920 using BERT-128, whose sequence length is 128. Table 1 collects the data for both throughput and latency. We report the results of TVM v0.8 for this model, and also consider the result of highly-crafted C++ implementations provided by the vendor of STCP920, which schedules a computational graph in a similar way to our idea and implements kernel generation by hand.

Table 1: Results of BERT-128 under different batch sizes.

approach	batch size	configuration		throughput (sentences/s)	latency (ms)
		iter.	batches/cluster		
TVM	8	1	2	138	6.79
	16	1	4	512	9.48
	32	2	4	512	18.96
	64	4	4	512	37.92
GraphTurbo	8	1	2	138	6.79
	16	1	4	512	9.48
	32	1	8	4052	7.67
	64	1	16	2716	23.58
	64	2	8	4052	15.34
crafted code	32	1	8	4048	7.62

For TVM, a cluster's LLB is sufficient to retain four data batches. As allocating four batches to each cluster makes two clusters idle, we allocate two batches to each cluster when the batch size is eight, which obtains a 138 sentences/s throughput and a 6.79ms latency. When the batch size increases to 16, TVM can allocate four batches to each cluster; the throughput and latency grow to 512 sentences/s and 9.48ms, respectively. As TVM cannot allocate more batches to a cluster, the throughput cannot further scale with the growth of the batch size. Instead, TVM introduces a loop execution within each cluster, which guarantees the throughput but the latency increases as proportional to the number of loop iterations.

GraphTurbo performs the same as TVM when the batch size is 8 and 16, since a cluster's LLB is sufficient to handle the allocated batches and we do not need to create larger sub-graphs or split them. This illustrates that **the scheduling of TVM can be considered as a special case of our work**. The difference is observed when the batch size increases to 32, for which GraphTurbo allocates eight batches to a cluster but TVM only allocates four. GraphTurbo creates larger sub-graphs and splits them into instances, achieving a higher throughput of 4052 sentences/s and a lower latency of 7.67ms.

When the batch size increases to 64, GraphTurbo allocates 16 batches to each cluster but suffers from both throughput degradation and latency increase, since such a batch allocation requires larger tensors than the implementation in §4.3 spills more of them to slower buffers. In this case, we also introduce a loop execution within a cluster by allocating eight batches to it. As a result, the throughput stays at 4052 sentences/s and the growth of latency is also alleviated when compared with the case of allocating 16 batches to each cluster.

Table 1 also indicates that GraphTurbo achieves very close throughput and latency to the vendor-crafted implementation. In the following context, we report the results of TVM and GraphTurbo by selecting their optimal numbers of allocated batches for a cluster. Both optimal batch allocation strategies are obtained after a round of beforehand autotuning executions. For the sake of simplicity, we discuss throughout numbers below but the results also apply to latency.

5.2 Performance Comparison

We now report the performance. BERT is configured using four sequence lengths, 256, 384 (the default MLPerf configuration), and 512. Table 2 summarizes the configurations of each model. TVM's throughput is listed in the rightmost column, which is preceded by the throughput units. We show the speedups of each approach over TVM's data in Fig.14, where we also report the results of AStitch [60].

TVM still fuses *op*s within a sub-graph and produces kernels that exchange data via DDR, and it also misses the instruction scheduling opportunities across layers. By (1) producing fewer kernels and reducing off-core data movements, (2) better saturating L1, and (3) further exploiting cross-layer instruction scheduling, GraphTurbo outperforms TVM by 11.15× on average.

Table 2: Summary of the models.

label	model	batch size	batches per cluster		throughput unit	TVM's result
			TVM	GraphTurbo		
Ⓐ	ResNet-50	64	2	16	images/s	1064
Ⓑ	BERT-128	32	4	8	sentences/s	512
Ⓒ	BERT-256	16	2	4	sentences/s	412
Ⓓ	BERT-384	8	1	2	sentences/s	36
Ⓔ	BERT-512	8	1	2	sentences/s	324
Ⓕ	DLRM	1024	64	256	queries/s	131000
Ⓖ	MobileNet-v2	128	2	32	images/s	1416
Ⓗ	Vision_Transformer	32	4	8	images/s	40
Ⓘ	DenseNet	32	4	8	images/s	456
Ⓙ	Conformer	12	1	3	sentences/s	184

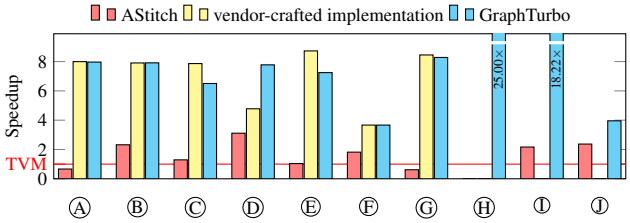


Figure 14: Speedups of throughput over TVM.

We reproduce the functionality of AStitch for STCP920 by maximally preserving tensors in L1. However, AStitch does not split a sub-graph into instances and thus fails to benefit from the imbalanced memory usage distribution enabled by better schedule orders of sub-graph instances. Hence, GraphTurbo obtains a mean speedup of 6.16× over it. Its performance falls behind that of TVM for ResNet-50 and MobileNet-v2 because AStitch prefers to produce a single kernel for compute-intensive *conv op*s in these models, which spills data to DDR and results in heavier data movements.

The vendor-crafted implementation considers all the optimization opportunities studied in this paper. Manually optimizing a computational graph can better exploit the trade-off between parallelism, load balance and locality, making crafted code sometimes obtain better performance than GraphTurbo, *e.g.*, for BERT-512 and MobileNet-v2. However, a manual scheduler is also non-trivial and thus sometimes misses the imbalanced memory usage distribution, *e.g.*, for BERT-384. Due to the complexity of such a scheduling strategy, the vendor implementation for the last three models is still under construction till now, and their data are thus missing. On average, GraphTurbo achieves a 1.04× speedup over the vendor-crafted implementation.

5.3 Performance Breakdown

This section studies how different factors contribute to the overall speedup of GraphTurbo over TVM. We consider four variants of GraphTurbo as follows. First, we only keep the outputs of each kernel generated by GraphTurbo in LLB as

much as possible. Second, we let these outputs stay in L1 to the greatest extent. Third, we split sub-graphs into instances and schedule them based on the second variant, but across-layer instruction scheduling is disabled. Finally, we turn on all optimizations. Fig.15 shows the comparison results.

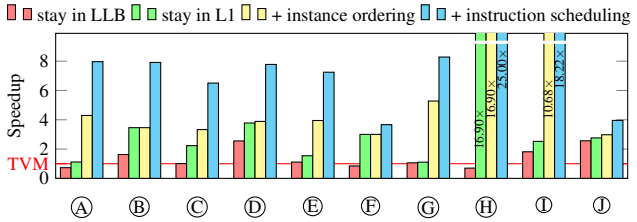


Figure 15: Individual contributions of each optimization.

The results of the second variant illustrate that converting off-core data movements into on-core data exchanges indeed makes sense. By achieving an average speedup of 3.67× over TVM, the green bars also outperform the red bars, which do not always obtain positive speedups over TVM. This demonstrates the importance of preserving tensors greedily in LB, *i.e.*, L1 of STCP920.

The third variant outperforms the green bars by 2.20× on average. ResNet-50, BERT-256, BERT-512, MobileNet-v2, and DesneNet, that exhibit imbalanced memory usage distribution caused by their down-sampling operations, benefit more from the ordering of sub-graph instances. Other models that either do not have such a network property (BERT-128 and Vision_Transformer) or introduce more concatenation *op*s than the remaining ones (DLRM) observe insignificant improvements. Finally, across-layer instruction scheduling (§4) obtains a mean speedup of 1.72× over the third variant.

5.4 Hardware Utilization

This section evaluates how effectively GraphTurbo can utilize DSA hardware resources. First, as the core idea is to convert off-core data movements to on-core data exchanges, we investigate how the memory hierarchy of STCP920 is utilized. To this end, we report in Table 3 the frequencies of each memory level that different approaches utilize.

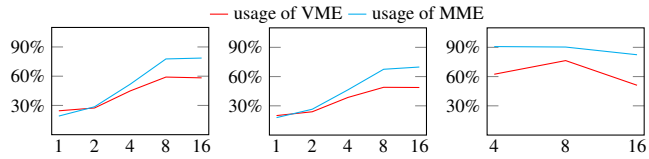
Table 3: Comparison of buffer scopes.

label	DDR			LLB			L1		
	TVM	crafted	GraphTurbo	TVM	crafted	GraphTurbo	TVM	crafted	GraphTurbo
Ⓐ	58	1	1	0	11	11	0	291	284
Ⓑ	242	2	1	0	0	0	0	304	305
Ⓒ	242	2	1	0	25	110	0	401	240
Ⓓ	515	2	1	0	49	75	0	968	967
Ⓔ	242	2	1	0	25	76	0	474	337
Ⓕ	76	1	0	0	0	0	0	75	76
Ⓖ	56	1	0	0	7	3	0	619	608
Ⓗ	214	-	24	0	-	60	0	-	340
Ⓘ	247	-	0	0	-	3	0	-	389
Ⓙ	1054	-	4	0	-	813	0	-	250

TVM always puts the output tensors of its sub-graphs in DDR, resulting in abundant off-core data movements. In contrast, GraphTurbo maximizes the utilization of faster local memory, converting many off-core data movements into on-core data exchanges. The vendor-crafted implementation also makes use of the faster local memory. Due to their familiarity with the hardware, the architects of STCP920 sometimes can better manage the memory hierarchy than our heuristics, but this manual scheduler is also tedious.

Second, we evaluate how VME and MME are utilized using ResNet-50 and BERT-128. We report in Fig.16 the data under different batch sizes, with the quantization version of ResNet-50 also considered, to validate the scalability of GraphTurbo. Other models observe similar results. The utilization of both VME and MME increases with the growth of batch size, which is exploited by our work. BERT-128 suffers from a degradation when the batch size changes from 8 to 16, as explained in §5.1.

Fig.17 shows the utilization of VME and MME when executing the four stages in Fig.3a. GraphTurbo performs similar to TVM for stage1, but it outperforms TVM for the other three stages, which demonstrates exploiting the imbalanced memory usage distribution can better utilize hardware resources.



(a) ResNet-50 (FP16) (b) ResNet-50 (INT8) (c) BERT-128

Figure 16: Usage of VME/MME. *x* axis denotes batch sizes.

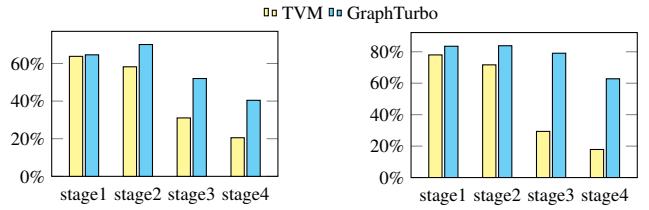


Figure 17: Utilization VME (left) and MME (right) when executing the stages in Fig.3a. *y* axis is the utilization percentage.

5.5 Comparison of Compilation Overhead

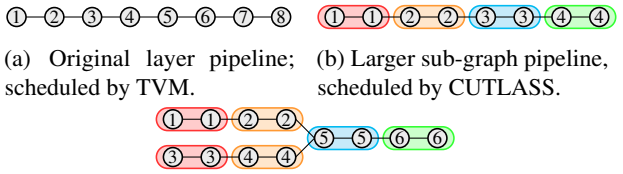
As compilation time is also a major concern for scheduling DNN models, this section reports the compilation overhead of GraphTurbo and compares it with those of the baseline methods. Table 4 reports the data in seconds, which demonstrates that GraphTurbo can achieve better performance than the state of the art without significantly aggravating the compilation overhead.

Table 4: Comparison of compilation overhead in seconds.

label	TVM	AStitch	GraphTurbo
Ⓐ	102	66	139
Ⓑ	159	128	199
Ⓒ	170	136	224
Ⓓ	312	290	699
Ⓔ	171	143	282
Ⓕ	25	22	23
Ⓖ	74	57	248
Ⓗ	189	146	340
Ⓘ	173	129	189
Ⓙ	382	238	296

5.6 Case Study on GPU

We now conduct a case study using the ResNet18-Tailor model to validate that scheduling sub-graph instances can be extended to NVIDIA A100 GPU. We use CUTLASS v2.9 [27] to implement kernel generation, which is compiled using CUDA toolkit v11.4 with *-O3* flag. We did not consider the *conv* and maxpooling *op*s at the start, and avgpooling and softmax *op*s at the end of this model since they do not contribute to imbalanced memory usage distribution, like those before and after the stages of Fig.2. The other layers, each composed of a *conv* and ReLU *op*s using a circle, are shown in Fig.18a.



(a) Original layer pipeline; scheduled by TVM.

(b) Larger sub-graph pipeline, scheduled by CUTLASS.

(c) Sub-graph instance pipeline, scheduled by GraphTurbo.

Figure 18: Different schedule orders of ResNet18-Tailor. The numbers define the schedule order of each method.

We let TVM wrap CUTLASS when generating kernels for layers. CUTLASS fuses two *conv* layers, partitioning the model into four larger sub-graphs (Fig.18b), the first two of which is split by GraphTurbo into two instances (Fig.18c). The output tensors of the (red) sub-graph instances are stitched via GPU registers and those of the (orange) ones through GPU shared memory. Table 5 summarizes the results.

Table 5: Execution time in milliseconds on A100 GPU.

batch size	TVM	CUTLASS fusion	graph scheduling	Speedup over	
				TVM	CUTLASS fusion
64	0.99	0.84	0.83	1.19×	1.01×
128	1.88	1.62	1.50	1.25×	1.08×
256	3.51	3.04	2.83	1.24×	1.07×
512	6.76	5.83	5.44	1.24×	1.07×
average				**1.23**×	**1.06**×

By scheduling sub-graph instances and exploiting GPU registers and shared memory between them, GraphTurbo outperforms TVM by 1.23× on average, which demonstrates that our idea is also useful on GPU. GraphTurbo also achieves a mean speedup of 1.06× over CUTLASS, because scheduling sub-graph instances also brings about benefits by exposing and exploiting imbalanced memory usage distribution.

The reasons why the performance improvements on GPU is not promising as on STCP920 are two-folded. On the software side, GraphTurbo can construct a larger sub-graph that contains more than two layers, but CUTLASS, which we use for kernel generation here, refused to accept three or more *conv* layers. Enhancing the kernel generator of TVM in the future can address this issue. We also did not apply instruction scheduling here. On the hardware side, the higher memory bandwidth of this GPU also makes the improvements caused by reducing off-core data movements not significant as on STCP920. While STCP920 only delivers a memory bandwidth of 136GB/s, this GPU can reach more than 1500GB/s.

Nonetheless, other NVIDIA software-defined platforms with limited memory bandwidth, *e.g.*, NVIDIA DRIVE AGX Orin [36], could benefit from GraphTurbo. We also believe that the software-controlled inter-cooperative-thread-array shared memory of the latest NVIDIA H100 GPU could be better exploited by the idea presented in this paper. Hence, our work also offers insights to the GPU micro-architectures.

6 Related Work

Scheduling its computational graph is the first step to deploy a DNN model on platforms. The difference between our scheduler and prior work is that we consider hardware architecture when grouping sub-graphs, which enables the synergy between network architecture and DSA, while existing methods [13, 19, 44, 49] not. By generating coarser-grained sub-graphs and splitting them into instances, GraphTurbo exposes the imbalanced memory usage distribution, a network property first studied by MCUNetV2 [30]. However, MCUNetV2 only discusses tiny DNN models on microcontroller units, while this paper considers large-scale DNN models and targets a cloud DSA chip. Some optimizations that can only be implemented when sub-graphs are lowered to loop nest pipelines are not considered by MCUNetV2 but studied in §4.

When a sub-graph is lowered to a loop nest pipeline, existing methods like TVM [5] fuse loop nests with the help of manually written schedule templates. As TVM does not scale well with the increase of *op* numbers within a sub-graph, we only use TVM's loop fusion to group loop nests within a layer. Our implementation also avoids the need to manually write schedule templates and inject memory promotion statements of weight tensors, the later of which is automated by interacting with the graph scheduler.

By expanding a high-level sub-graph into individual low-level *op*s, XLA [13] does not restrict fusion within layers. Nonetheless, retrieving the high-level information via low-level *op*s is critical to fuse low-level *op*s for XLA, and manually forming profitable high-level sub-graphs is considered as

more robust than through automatic pattern matching in this compiler [45]. GraphTurbo fully automates this process and achieves better performance than AStitch, which has already been demonstrated as superior to XLA [60].

IREE [20] is another work that makes use of its graph scheduling logic when communicating data between low-level parallel pipelined hardware/APIs. Our work differs from IREE by focusing on scheduling instances of larger sub-graphs, which tends to produce fewer kernels. By managing an internal map of *op* sequence on the fly, Zero-Infinity [42] exploits the fine-grained overlaps by prefetching the parameters required by future *op*s during the execution of the current *op*. A layer considered by our work usually includes multiple *op*s, the execution of which is more likely to fully hide the data transfer overhead.

Some autotuning frameworks [6, 57, 59] are also devised to enhance the power of TVM with fewer or no hand-written schedule templates. These autotuners use their search heuristics to tune memory optimizations to further improve the performance of their generated code. Unfortunately, their search spaces are all restricted within layers [39], while our work enables across-layer instruction scheduling (§4.4). In particular, Ansor [57] represents the state of the art of this kind, which is orthogonal to our work by neither considering the scheduling of GraphTurbo nor exploiting the fusion possibilities of multiple *conv*/matrix multiplication operators. Loop fusion is also investigated by polyhedral frameworks [3, 47, 54], but they did not consider buffer stitching that has been discussed in §4.2. Similar to GraphTurbo, DNNFusion [34] also studies across-layer fusion for mobile devices. We fail to obtain its repository to conduct an experimental comparison.

Another thread of works [10, 28, 35] investigate horizontal fusion between *op*s with no producer-consumer relations to better utilize the hardware resources of their targets. GraphTurbo tackles the same issue using a different idea. Our scheduler exploits parallelism within a sub-graph instance, which is always homogeneous to other instances of the same sub-graph. It always decomposes one or multiple dimensions of a tensor to exploit parallelism. On the contrary, *op*s grouped by horizontal fusion are heterogeneous, which calls for a more complicated parallelization mechanism.

Recently, schedulers and code generators for DSA platforms are widely studied. Rammer [31] and Roller [62] generate code for Graphcore IPU [22]. They maximize the utilization of faster memory by combining *op*s that cannot saturate hardware resources. AKG [54] targets code generation for Ascend 910 [29] using the polyhedral model [4, 48] to perform loop fusion. XLA [13] and NaaS [61] exploit the scheduling of sub-graphs for generating code on TPU [25].

The distinction between our work and these approaches is that GraphTurbo partitions a sub-graph along one output tensor's dimension while these methods partition tensors by tiles along multiple dimensions. The primary reason why GraphTurbo does this way is because cores in the DSA abstraction of Fig.1 are organized in 1D form. For instance, this level corresponds to 32 hardware cores sharing the shared memory of GPU. The partitioning approach of GraphTurbo is also extensible to deal with a multi-dimensional core grid organization by gradually partitioning and mapping multiple loop dimensions to these hardware dimensions. Moreover, as their targets share the DSA abstraction in Fig.1, the idea presented in this paper could also be used on their targets.

7 Conclusion

GraphTurbo is a scheduler for DNN models that enables the synergy between network and hardware architectures. This significant difference from prior work produces fewer kernels and thus reduces off-core data movements, better saturates faster local memory of DSA platforms by exploiting the imbalanced memory usage distribution, and opens opportunities for across-layer instruction scheduling. Results of seven DNN models demonstrate the effectiveness of our idea, whose applicability to GPU is also discussed.

GraphTurbo obtains sub-graph instances by selecting an appropriate size to split a DNN computation graph. Indeed, selecting the optimal size to perfectly model a DSA's memory hierarchy is challenging, and only making use of LB in Fig.1 is not the optimal solution. Instead, our method is just a greedy idea that has been demonstrated effective when compared with vendor-crafted implementation, which we believe can be considered as a good result. A more intelligent approach can be explored to catch up or even beat the performance obtained by hand for models like BERT-512 and MobileNet-v2.

GraphTurbo currently has two limitations. First, the optimal batch size for a cluster is still selected by a simple autotuning approach. We intend to develop an intellectual technique to better address this issue. Second, GraphTurbo cannot handle dynamically shaped tensors. Integrating with the recent methods [12, 56] along this direction may alleviate this issue.

Acknowledgments

We acknowledge Hyeontaek Lim for his shepherding and the OSDI'23 reviewers for their constructive comments that improve the quality of this work. We would also like to express our gratitude to Bojie Li and Jun Yang for their suggestions on the early versions of this paper, Zhongzhou Jiang, Yuqing Wang, Di Mei, and many other toolchain team members of the Streaming Computing Inc. for their help during the use of their vendor-crafted implementation. Jie Zhao and Xiaoqiang Dan are the corresponding authors of this paper, and the work of Jie Zhao is partly supported by the National Natural Science Foundation of China under Grant No. U20A20226. The views and conclusions presented in this paper belong to the authors. Interpreting them as the official policies of the Chinese Government in any way is not acceptable.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[8] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, oct 2016.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 1–14, 2021.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[12] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 721–747, 2022.

[13] Google. Xla: Optimizing compiler for machine learning. *https://www.tensorflow.org/xla*, 2017.

[14] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented Transformer for Speech Recognition. In *Proc. Interspeech 2020*, pages 5036–5040, 2020.

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[19] Intel. onednn graph specification 1.0-alpha. *https://spec.oneapi.io/onednn-graph/latest/index.html*, 2020.

[20] IREE. Iree. *https://iree-org.github.io/iree/*, 2021.

[21] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 711–732, 2021.

[22] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 47–62, New York, NY, USA, 2019. ACM.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, aug 2018.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA'17, pages 1–12, New York, NY, USA, 2017. ACM.

[26] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 387–400, 2021.

[27] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++. *https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/*, 2017.

[28] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8343–8354. Curran Associates, Inc., 2020.

[29] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801, 2021.

[30] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Memory-efficient patch-based inference for tiny deep learning. In M. Ranzato, A. Beygelzimer, K. Nguyen, P.S. Liang, J.W. Vaughan, and Y. Dauphin, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 1–13. Curran Associates, Inc., 2021.

[31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[32] Eitan Medina. Habana labs presentation. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–29, 2019.

[33] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[34] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 883–898, New York, NY, USA, 2021. ACM.

[35] NVIDIA. Nvidia tensorrt. *https://developer.nvidia.com/tensorrt*, 2016.

[36] NVIDIA. Nvidia introduces drive agx orin — advanced, software-defined platform for autonomous machines. *https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orin-advanced-software-defined-platform-for-autonomous-machines*, 2017.

[37] Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 584–597, 2021.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[39] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16, 2021.

[40] Yury Pisarchyk and Juhyun Lee. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288*, 2020.

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 519–530, New York, NY, USA, 2013. ACM.

[42] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 446–459. IEEE Press, 2020.

[44] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[45] Bjarke Roune. Compiling ml with xla (slides). *https://www.c4ml.org/c4ml2019*, pages 16–24, 2019.

[46] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

[47] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Trans. Archit. Code Optim.*, 16(4), October 2019.

[48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[49] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[50] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.

[51] Rongkai Zhan and Xiaobo Fan. Neuralscale: A risc-v based neural processor boosting ai inference in clouds. In *Fifth Workshop on Computer Architecture Research with RISC-V*, CARRV, 2021.

[52] Jie Zhao and Peng Di. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture*, MICRO-53, pages 427–441, Piscataway, NJ, USA, 2020. IEEE Press.

[53] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 1–19, 2022.

[54] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'21, pages 1233–1248, New York, NY, USA, 2021. ACM.

[55] Yongwei Zhao, Zidong Du, Qi Guo, Shaoli Liu, Ling Li, Zhiwei Xu, Tianshi Chen, and Yunji Chen. Cambricon-f: Machine learning computers with fractal von neumann architecture. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA'19, pages 788–801, New York, NY, USA, 2019. ACM.

[56] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 848–863, 2022.

[57] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[58] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 874–887, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 859–873, New York, NY, USA, 2020. ACM.

[60] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 359–373, New York, NY, USA, 2022. ACM.

[61] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Towards the co-design of neural networks and accelerators. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 141–152, 2022.

[62] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.