Welder: Scheduling Deep Learning Memory Access via Tile-graph

Yining Shi, *Peking University & Microsoft Research;* Zhi Yang, *Peking University;* Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/osdi23/presentation/shi

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

WELDER: Scheduling Deep Learning Memory Access via *Tile-graph*

Yining Shi[†◇*] Zhi Yang[†] Jilong Xue[◇] Lingxiao Ma[◇] Yuqing Xia[◇]
Ziming Miao[◇] Yuxiao Guo[◇] Fan Yang[◇] Lidong Zhou[◇]
[†]*Peking University* [◇]*Microsoft Research*

Abstract

With the growing demand for processing higher fidelity data and the use of faster computing cores in newer hardware accelerators, modern deep neural networks (DNNs) are becoming increasingly memory intensive. A disparity between underutilized computing cores and saturated memory bandwidth has been observed in various popular DNN models. This inefficiency is caused by both the conventional treatment of DNNs as compute-intensive workloads and the lack of holistic memory access optimization in DNN models.

In this paper, we introduce WELDER, a deep learning compiler that optimizes the execution efficiency from a holistic memory access perspective. The core of WELDER is *tile-graph*, an abstraction that facilitates fine-grained data management at tile level. By leveraging the observation of optimization independence across memory layers, WELDER is able to decompose the whole combinatorial DNN optimization space into several independent ones and effectively trade off between intra- and inter-operator data reuse using a tile traffic-based cost model. This allows WELDER to unify previous ad-hoc memory optimizations into a single space, generate efficient execution plans with 89 more optimization patterns, and outperform state-of-the-art solutions significantly. WELDER is also able to handle DNN models with arbitrarily large input by combining the existing accelerator memory and host memory as a whole system.

1 Introduction

Deep neural networks (DNNs) have been used in a wide range of tasks like vision and language analysis and synthesis. Conventional wisdom treats DNNs as compute-intensive workloads. A DNN model is often defined as a dataflow graph (DFG), where each node represents a compute-intensive operator (e.g., matrix multiplication). These operators are offloaded to modern accelerators with massive parallel computing cores, such as GPUs and TPUs [23], to speed up computation. To utilize accelerators efficiently, DNN frameworks and compilers explore various optimization techniques, such as code specialization [15,50,52] and operator fusion [15,31].

Although these computation centric optimizations are shown effective for classic DNN models, we observe that modern DNNs are becoming increasingly memory intensive. Our profiling on a range of state-of-the-art DNN models reveals that the bottleneck of the end-to-end DNN computation is mostly on GPU memory. The memory bandwidth utilization can be as high as 96.7% while the average utilization of computing cores is only 51.6% (§2). Moreover, we observe the disparity between the underutilized cores and the saturated memory bandwidth could become even larger with the evolution of both hardware and DNN models. Modern models are processing higher fidelity data, e.g., larger images, longer sentences, high-definition graphics, which consume more memory bandwidth in the computation. Furthermore, the faster computing cores (e.g., TensorCore [6]) impose an even greater pressure on memory.

Optimizing memory intensive DNN workloads is challenging as it requires improving the sophisticated data access and reuse patterns across multiple memory layers (e.g., GPU DRAM and shared memory). From the memory perspective, DNN computation comprises of a repetitive process for each operator to 1) load input tensors across memory hierarchy, 2) compute at the cores, and 3) store the resulting tensors across memory hierarchy. To derive a good data access pattern, it requires a careful calculation of the size of tile, a partition of a tensor, along each tile dimension. Such a tiling strategy is already difficult to obtain in existing practice [5, 50, 52]. As a further complication, due to the different algorithmic semantics, each operator may require a different data access patterns. Such diversity across operators makes *inter-operator* data reuse especially challenging, and often infeasible. If the derived tile shape of an operator at a certain memory layer does not match that of a downstream operator, it is difficult to reuse the tile at that layer. Consequently, existing approaches either focus on intra-operator optimization and leave all inter-operator intermediate tensors in the lowest memory layer

*Work is done during the internship at Microsoft Research.

(e.g., GPU memory), or rely on rule-based operator fusions to alleviate the inter-operator memory overhead. These rules are only applicable for specific operator combinations (e.g., register fusion for element-wise operators [10, 13, 15], shared memory fusion for a limited set of operator types [51]) and can be suboptimal when having different input sizes or running on different hardware configurations.

In this paper, we introduce WELDER, a deep learning compiler that holistically optimizes memory access for end-to-end DNN models consisting of general operators. The design of WELDER is based on three key observations. First, to resolve potential tile shape conflicts between two adjacent operators, we observe that their aligned tile shape can be automatically inferred by propagating an output tile shape from back to front, given that the computing logic in each operator can be accurately preserved (e.g., through the tensor expression). Second, to decide which tile shape will lead to better performance, by enforcing the computation pattern to be aligned with hardware feature (e.g., TensorCore), we can just minimize the data traffic across all memory layers. Given the operators with aligned tile configuration, we notice that their data traffic can be easily modeled based on their input/output tile sizes and the input/output tensor shapes. Finally, when considering the whole memory hierarchy, we observe that the optimization of memory traffic is inherently independent across memory layers, i.e., *inter-layer independence*. Particularly, the above traffic model is determined only by the tile configuration at the memory layer of interest. These observations allow us to optimize the whole space with an effective process: starting from aligning two adjacent operators at independent memory layers, deciding their optimal tiling size at the right memory layer guided by traffic costs, and expanding the optimization to include further operators.

WELDER incorporates these insights into a new DNN compiler design. First, to facilitate fine-grained data management, WELDER proposes *tile-graph*, a tile-level data-flow graph to model DNN computation. Each node in the graph processes one data tile of a tensor at a time. To map DNN computation to a multi-layered memory hierarchy, WELDER allows the control of each node's data tile size and the desired memory layer to reuse the data tile between two nodes. Specifically, WELDER provides a `SetConnect` interface to set the data reuse layer for each edge and a `Propagate` interface to infer the tile configurations within a group of connected nodes. Second, to efficiently optimize the tile level data-flow scheduling holistically, WELDER exploits the *inter-layer independence* properties in the data-flow computation to decouple the optimization space into multiple sub-spaces. Based on this, WELDER proposes a two-layered scheduling policy that enumerates different memory connection options for each edge and decides on an efficient tile configuration for each sub-space guided by the traffic cost model. Finally, the optimized execution plan is mapped to executable code for a specific hardware accelerator through four abstracted computing interfaces defined in the hardware layer: `Allocate`, `LoadTiles`, `ComputeTile`, and `StoreTiles`.

With the tile level holistic data-flow scheduling, WELDER is the first to unify all common operator fusions (e.g., register-based element-wise fusion, shared-memory fusion, etc.) into a single framework. This generality allows WELDER to find 89 uncommon operator fusion patterns automatically that are mostly unexplored by existing rule-based approaches (§5.2). Interestingly, our approach can easily support new requirements for handling DNN models with arbitrarily large input (e.g., high-resolution images), where even a single operator may be too large to fit in the GPU memory. Specifically, by extending the current memory hierarchy with additional layers (e.g., host memory), WELDER can generate an optimized execution plan across the combined hierarchy of host and device memory.

We have implemented WELDER on top of TVM [15], Rammer [31] and Roller [52]. Our evaluation is conducted on 10 state-of-the-art DNN models covering both classic and recent model structures for various tasks including vision, NLP, 3D-graphics, etc. The evaluation results show that WELDER significantly outperforms the state-of-the-art DNN framework and compilers like PyTorch, ONNXRuntime, and Ansor on both NVIDIA and AMD GPUs, with up to 21.4×, 8.7×, 2.8× speedups, respectively. WELDER's automatic optimization even outperforms TensorRT [7] and Faster Transformer [2], which are a highly optimized handcrafted DNN inference library and a model-specific implementation from NVIDIA, with up to 3.0× and 1.7× speedups. Furthermore, when running these models on hardware with faster computing cores such as TensorCore, we observe a larger improvement in performance, highlighting the importance of memory optimization for future AI accelerators.

2 Motivation

Modern DNNs are memory-bounded. Figure 1 presents the average GPU utilization, including both computational FLOPS and global memory throughput, for a representative DNN benchmark running with ONNXRuntime [8]. As shown, the average computation utilization is only 51.6% while memory utilization is 96.7%. When examining the model types, we find that ResNet and BERT, which are dominated by convolution and matrix multiplication operators and can achieve relatively high computation utilization (e.g., >80%), are two representative classical models. However, the remaining models, which are popular models proposed in recent years, exhibit low computation efficiency due to introducing more memory-intensive patterns beyond compute-intensive operators. Additionally, we observe that the new DNN models often have a higher ratio of memory store traffic to load traffic compared to classical models. The primary reason is these models tend to process high-fidelity data and generate large activations across layers. However, current systems such

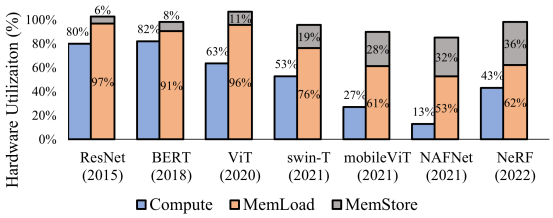

Figure 1: Computation FLOPS and memory bandwidth utilization for different models on NVIDIA V100 GPU.

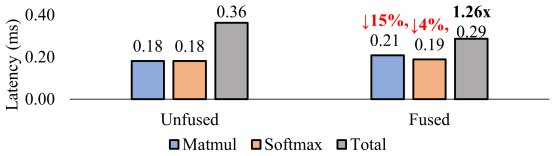

Figure 2: Latency numbers of unfused, fused, and each individual kernels of Matmul and Softmax.

as ONNXRuntime have limited optimizations for reducing inter-operator traffic. This indicates that these models will frequently exchange large intermediate data across operators through global memory. The results highlight the need for optimizing memory access efficiency across operators.

Conflicted intra- and inter-operator data reuse patterns. Optimizing intra-operator and inter-operator data reuse simultaneously is challenging. An operator is often implemented as nested multi-level loops over all tensor dimensions. Within the operator, the data reuse across multiple memory layers are often implicitly optimized using sophisticated loop tiling techniques [5, 50, 52]. We consider a typical pattern of two consecutive operators, i.e., Matmul and Softmax. When the two operators are optimized independently, their optimal tile sizes in shared memory are different, e.g., [32×64] for Matmul and [4×128] for Softmax. As a result, Softmax is unable to reuse the intermediate data from Matmul in shared memory, leading to a total latency of 0.36ms, as shown in Figure 2. However, if we force them to take into account both intra- and inter-operator data reuse, the fused operator latency can be reduced to 0.29ms, achieving a 1.26x speedup. Upon examining their aligned tile size (i.e., [16×128]), we observe that both operators sacrifice their own efficiency (e.g., with 15% and 4% performance degradation when running separately, due to suboptimal data tile for intra-operator data reuse) in favor of overall efficiency. This demonstrates the need for an efficient data reuse solution across intra-operator and inter-operator to optimize memory access holistically.

Key observations. Through a further analysis on the example in Figure 2, we have identified three key observations. First, an aligned tile configuration across operators can be deduced based on a chain of shape inference starting from an output tile shape. For example, if we want to compute a [4×128] output tile of Softmax, based on its computing logic (e.g., tensor expression), we can deduce that its dependent input tile shape is also [4×128]. Then, by using [4×128] as the output tile of Matmul, we can further deduce that input tile shapes of Matmul will be [4×k] and [k×128], where k is an reduction size that can be set as any number not exceeding the reduction dimension size of the Matmul. In this way, the two operators can be fused by reusing the intermediate data tile ([4×128]) in shared memory.

Second, given the aligned tile configuration and the original tensor shapes, the total memory traffic can be easily derived analytically. In this example, the Matmul takes input tensors A in shape [98304×64] and B in [64×128] respectively, and an output tensor C in [98304×128]. The Softmax then takes C as input and produces an output tensor D in the same shape. Input tensors A, B, and the output tensor D are in global memory. Given these shapes, we can first calculate the memory traffic when computing a single output tile (i.e., [4×128]) of tensor D. To do so, it will first load a tile of shape [4×k] from tensor A and a [k×128] tile from tensor B for Matmul, and then the intermediate tile [4×128] will be consumed by Softmax in shared memory, and write a tile of shape [4×128] to tensor D, where the k can be replaced as 64 given the input tensor shape of [98304×64]. Thus, the total traffic incurred in global memory for an individual output tile is 35KB ((4*64+64*128+4*128)*4Bytes(FP32)), where the traffic of the intermediate tile [4×128] is saved due to data reuse in shared memory. To compute the full output tensor D, a total of 24,576 such computations are required (i.e., (98304*128)/(4*128), resulting in a total global memory traffic of 840MB (i.e., 24,576*35KB). Interestingly, changing the output tile to [16×128] will reduce the total traffic to only 264MB, following the same calculation.

Finally, our traffic-cost calculation is only determined by the tile configuration at the memory layer of interest, e.g., the output tile shapes of [4×128] or [16×128] in shared memory, once the tensor shapes are specified. This allows us to choose the tile size for each layer independently in order to optimize the traffic cost from the lower memory layers.

These observations together provide us an effective way to optimize memory access holistically, i.e., aligning a group of adjacent operators through an output tile shape, deciding on the best tile shape based on memory traffic, and optimizing for each memory layer independently. In this way, WELDER is able to change the original coarse-grained inter-operator dependency into a more fine-grained tile-level dependency, which essentially removes some false barriers between operators and enables more concurrency.

3 WELDER Design

The observations in §2 motivate WELDER, a deep learning compiler that aims to improve the performance of modern DNNs in a holistic memory access scheduling space. Figure 3 shows the system overview. WELDER takes a full DNN model as input and converts it into a data-flow graph of tile-based

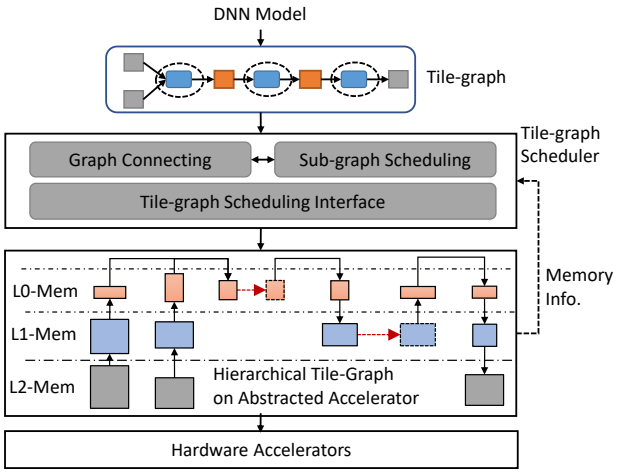

Figure 3: System overview of WELDER.

computing tasks (i.e., *operator-tiles*), which is called *tile-graph* (§3.1). A tile-graph provides fine-grained control over data tile configurations and memory placement. Given a tile-graph, WELDER resolves the intra-operator and inter-operator data-reuse conflicts through a "first-connect-then-schedule" approach: it first assumes two adjacent operators can reuse data tile at a certain memory layer (i.e., connect), and then derives the best common tile shape to see if the total memory traffic can be reduced. To facilitate this goal, WELDER provides two tile-graph scheduling interfaces: `SetConnect` and `Propagate` (for the chain of shape inference). Based on this, we propose a two-step scheduling algorithm, i.e., *graph connecting* and *sub-graph scheduling*, to recursively decide an efficient tile-graph execution plan for multiple memory layers, known as a *hierarchical tile-graph* (§3.2). Finally, this plan is then mapped to an executable code for a specific hardware accelerator using four abstracted computing interfaces defined in the hardware layer, i.e., `Allocate`, `LoadTiles`, `ComputeTile`, and `StoreTiles` (§3.3). The memory specification of the abstracted accelerator is used by the tile-graph scheduling layer to guide the optimization process.

3.1 Operator-tile and Tile-graph

WELDER defines DNN computation in a fined-grained task granularity named *operator-tile*. A DNN operator, such as convolution, can be implemented as multiple homogeneous operator-tiles, which are executed either in a streaming or parallel manner to compute all the data tiles in the output tensors [31]. Each operator-tile takes as input a data tile sliced from the input tensors and computes a data tile in the output tensors, with the computing logic described by an index-based tensor expression [15]. Figure 4(a) and (b) shows examples of operator-tiles for `Conv` and `MaxPool`, where the `Conv` operator computes a $[1 \times 1 \times C]$ data tile by taking a $[3 \times 3 \times C]$ data tile as input, and the `MaxPool` operator takes an input tile of $[2 \times 2 \times F]$ and computes an output tile of $[1 \times 1 \times F]$.

To improve the utilization of hierarchical memory resources, such as the shared memory, WELDER allows two adjacent operator-tiles to be "connected" through a common intermediate data tile, also known as a *reuse-tile*. This allows the second operator-tile to consume the data produced by the first operator-tile directly, without the need to materialize it into a full intermediate tensor. Figure 4(c) illustrates an example of this connection between two operator-tiles for `Conv` and `MaxPool`, using a $[2 \times 2 \times F]$ reuse-tile. Multiple operator-tiles can be connected along each adjacent edge to form a data flow graph of operator-tiles, known as a *tile-graph*.

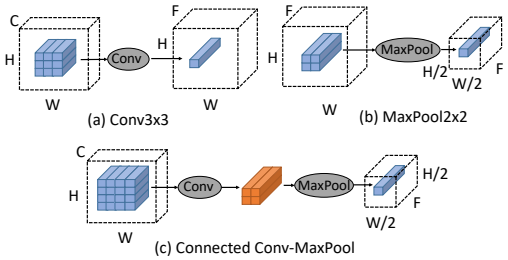

Figure 4: Illustration of two operator-tiles: (a) `Conv` and (b) `MaxPool`; and (c) connecting them into a tile-graph (the weight tensor of `Conv` is omitted for simplicity).

Tile propagation. Once connected, most tiles in a tile-graph are correlated, which can be automatically inferred by propagating an output tile shape to the entire graph. This is achieved by using a chain of shape inferences from the output nodes to the inputs. For each operator-tile, the dependent region of the input tensor can be accurately determined by analyzing its tensor expression and output tile size. In cases where the input region may contain irregular patterns such as sparse or noncontinuous access (e.g., `Gather` or `Convolution` with strides), our expression analysis provides a conservative upper bound as the input tile shape. If the tile-graph has multiple output nodes, their output shapes may also be correlated, as they may share a common ancestor node in the graph. In this case, after propagating the first output tile, we propagate separate shapes for the remaining output nodes, aligning them with the first one. If there is an inconsistent tile shape between the two propagations, we do not connect the latter output node to the current graph.

Memory traffic and footprint. After the tile propagation, the memory traffic and footprint of a tile-graph can be determined. First, the memory traffic for an individual tile-graph can be calculated by summing its input and output tile sizes. The total traffic is obtained through further multiplying this value by the number of tile-graphs needed to compute the full output tensor (e.g., through dividing the tensor size by the output tile size). Second, the minimum memory footprint for the tile-graph can be calculated using a memory allocation algorithm (e.g., bestfit [19]) by allocating all data tiles in a topological order. As a footprint optimization, input tiles that contain reduction axes can be further partitioned into smaller

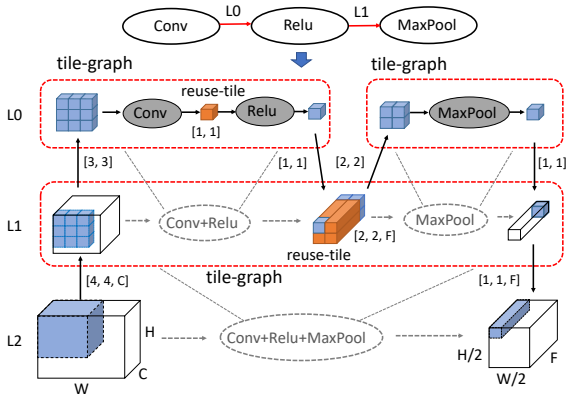

Figure 5: Map three consecutive operators to a three-layer memory hierarchy (the weight of `Conv` is omitted).

ones, which can be loaded and consumed sequentially by accumulating their results to the output tile. Specifically, a particular policy can automatically try different tiling sizes along the reduction axes during the tile propagation.

3.2 Tile-graph Scheduling

To map a DNN model represented by an inital data flow graph to an accelerator, we can recursively partition each operator into multiple operator-tiles to fit within each memory layer, and connect operator-tiles at higher memory layers to exploit inter-operator data reuse. As a result, an entire DNN computation can be modeled as a data streaming pipeline over a two-dimensional space, with data tiles moving up and down the memory hierarchy vertically and being passed to successor operators at different layers horizontally.

Figure 5 illustrates an example of mapping three consecutive operators (`Conv`, `ReLU`, and `MaxPool`) to a three-layered memory hierarchy (e.g., from L2 to L0). The input tile of the `Conv` operator is repeatedly loaded from L2 to L1 and then L0 for computation. By connecting the `Conv` and `ReLU` operators at L0, the output of the `Conv` operator can be reused as the input for the `ReLU` operator, and the two operators form a tile-graph at L0. At the same time, they are consolidated into a virtual node (i.e., `Conv+ReLU`) in L1. The output of the `ReLU` is then continuously spilled into the data tile at L1 and reused as the input for the `MaxPool`, through further connection at L1. This allows all three operators to form a single tile-graph at the L1 layer, resulting in the virtual node `Conv+ReLU+MaxPool` in L2. After this recursive process, all operators are connected at the lowest layer as a single tile-graph.

Decoupling optimization space. Given the observation that DNN computation is mostly memory-bounded, our major optimization goal of the data streaming pipeline can be transformed to minimizing the memory traffic. This allows us to decompose the whole optimization space into several sub-spaces by leveraging the inherent independence of optimizing

```
void SetConnect(Edge *edge, MemLevel level);
TileConfig Propagate(TileGraph g,
                     Map<Axis, Dim> config);
size_t MemFootprint(TileGraph g);
size_t MemTraffic(TileGraph g);
```

Figure 6: The scheduling interface in WELDER

traffic across memory layers. Specifically, the total data traffic loaded from and stored to a lower memory layer for a given tile-graph can be estimated by just its output tile shape, i.e., used to deduce all the input and output tile shapes. Based on this property, different tile-graphs from the same or different memory layers can independently optimize their memory traffic by searching for the optimal tile shapes. For example, in Figure 5, the tile-graph of `Conv` and `ReLU` at L0 can be optimized independently of the L1 tile-graph (e.g., formed by the `Conv+ReLU` and `MaxPool` operators), which is referred to as *inter-layer independence*. This further implies that the optimal tile configurations of the sub-graphs `Conv-Relu` and `MaxPool` at L0 are also independent, due to their independence with the tile-graphs at L1, to which we refer as *intra-layer independence*. In practice, the only constraint is that the tile size at the lower memory level must be larger than the tile size at the upper memory level. This is often the case, as the lower memory level typically has greater capacity than the upper memory level. With these properties, we can independently schedule each tile-graph given a graph connection plan.

Scheduling interface. WELDER provides two scheduling interfaces to control graph connecting and sub-graph tiling, as shown in Figure 6. First, the graph connecting is implemented using the `SetConnect` interface, which assigns a memory level for an edge in the tile-graph (the lowest level by default). After connecting, the tile shapes in the graph is inferred through the `Propagate` interface, by specifying the dimensional sizes of the output tiles and the optional reduction axes in input tiles. For example, in Figure 5, we can use the `SetConnect` interface to connect `Conv` and `Relu` at L0 and connect `Relu` and `MaxPool` at L1. After the connection, for the sub-graph `Conv+Relu`, we can use the `Propagate` to infer the intermediate reuse-tile shape (i.e., [1, 1]) by specifying the output tile shape of [1, 1]. Similarly, we can also infer the intermediate reuse-tile shape of sub-graph `Conv+Relu+MaxPool` (i.e., $[2,2,F]$) by specifying the output tile shape of $[1,1,F]$. The two scheduling primitives are essentially two interfaces to update the edges and vertices of the tile-graph. Particularly, `SetConnect` is used to add a connection between two nodes and `Propagate` is used to set tile configuration for a node. They together form a complete interface for updating the tile-graph. Note that these primitives are only used by WELDER's scheduling policy and transparent to the end users. WELDER also provides two cost interfaces, `MemFootprint` and `MemTraffic`, to calculate the memory

```
1  Func GraphConnecting(g:Graph, d:Device):
2      for node : TopologySort(g.nodes()) do
3          for edge : node.out_edges() do
4              for level : d.MemLevels() do
5                  SetConnect(edge, level);
6                  s = ExtractSubgraph(node, 0);
7                  configs = SubGraphTiling(s, 0, tensor_shapes);
8                  if t = Min(d.Profile(configs)) < best_latency
9                      best_latency = t;
10                     best_level = level;
11             SetConnect(edge, best_level);
12 Func SubGraphTiling(g:Graph, level:Memory, c: Config)
13     configs = PriorityQueue();
14     for subtile : EnumerateSubtiles(g, c) do
15         config = Propogate(g, subtile);
16         if MemFootprint(g) > level.capacity
17             continue;
18         configs.push(config, priority=MemTraffic(g));
19     results = Dict();
20     for config : TopK(configs, k) do
21         // return empty sub-graph at top level to exit recursion
           subgraphs = unique([ExtractSubgraph(node, level+1)
           for node in g.nodes()]);
22         for subgraph : subgraphs do
23             subgraph_configs = SubGraphTiling(subgraph,
               level+1, config);
24             results[config].append(subgraph_configs);
25     Return results;
26 Func ExtractSubgraph(node:Node, level:Memory)
27     nodes = Set();
28     for edge : node.InOutEdges() do
29         if edge.connect_level > level
30             nodes.insert(ExtractSubgraph(edge.node, level));
31     return SubGraph(nodes);
```

Figure 7: Two-step tile-graph scheduling algorithm.

footprint and the total traffic of a tile-graph, which serve as our cost models to guide the scheduling.

Scheduling policy. WELDER adopts a two-step scheduling algorithm to optimize data flow computation effectively. Specifically, a *graph-connecting scheduler* first enumerates different graph connecting plans by setting different memory reuse levels for each edge, and then a *sub-graph scheduler* quickly searches for efficient tile configurations for each sub-graph decoupled by the graph-connecting scheduler. Figure 7 shows the two-step scheduling algorithm in WELDER. First, given a DNN data flow graph g and an accelerator device d, the graph-connecting scheduler enumerates all graph nodes and their output edges in a topological order (line 1-3). For each edge, WELDER tries different connection levels (e.g., using the `SetConnect` interface) (line 5). It then extracts the connected sub-graphs where all edges have connection

`Allocate`	Allocate workspace in a memory layer
`LoadTiles`	Load input tiles from lower memory layer
`ComputeTile`	Compute an operator-tile at the top layer
`StoreTiles`	Store result tiles back to lower memory layer
`MemLevels`	Query memory hierarchy configurations

Table 1: Device interfaces in abstracted hardware accelerator.

levels higher than 0. Here, we use the number 0 to represent the lowest memory level, and larger numbers for higher levels. The `ExtractSubgraph` function is implemented in line 26-31. For the extracted sub-graph, WELDER calls the `SubGraphTiling` function to get several efficient tile configurations and chooses the optimal one by profiling on the hardware (line 7-10). After comparing with all other connection levels, WELDER sets the best connection level for the current edge.

Then, the sub-graph scheduler (i.e., the `SubGraphTiling` function) takes as input a sub-graph and the last level tile configuration and searches for efficient tile configurations for the current level. First, WELDER enumerates the tile sizes (i.e., `EnumerateSubtiles` in line 14) for output dimensions using a tile shape expanding approach similar to Roller [52], which enlarges initial tile shape (e.g., size of 1) towards the shapes that can reduce total traffic and align with hardware features. After getting the output tile shapes, we can infer the complete tile configuration using the `Propagate` interface and check if it exceeds the memory capacity using the `MemFootprint` interface, or appends it to a sorted result list with the memory traffic as the key (e.g., using the `MemTraffic` interface) (line 15-18). Finally, we choose the top K configurations with the least memory traffic for the current level, and then extract the upper-level sub-graphs and decide their best tile configurations recursively by calling `ExtractSubgraph` and `SubGraphTiling` (line 20-24).

Note that WELDER has no assumption on the memory size on different memory hierarchies, as our scheduling policy can always try its best to determine the optimal layer and tile size to place intermediate data, so as to minimize the overall latency. While WELDER always favors hardware with large higher-level fast memory (e.g., shared memory) that can hold a sufficiently large intermediate data tile, because too small tile sizes could lead to worse intra-operator data reuse. The scheduling result of a data flow graph in WELDER is a *hierarchical tile-graph*, which starts as a full graph at the lowest memory level and is recursively split into several sub-graphs in the upper layers, all the way to the top level.

3.3 Mapping to Hardware Accelerator

The hierarchical tile-graph generated by WELDER is an abstracted execution plan that can be mapped to an executable code for a specific hardware accelerator. To facilitate this mapping, WELDER provides an abstracted accelerator device with hierarchical memory layers. The memory configura-

```
void ExecuteGraph(TileGraph g, MemLevel level,
                  void *in, void *out) {
  void *mem = Allocate(g.MemFootprint(), level);
  LoadTiles(in, mem);
  for (auto n : g.nodes()
    if (level == MemLevel.top)
      ComputeTile(n, mem.in[n], mem.out[n]);
    else
      ExecuteGraph(n.TileGraph(), level+1,
                   mem.in[n], mem.out[n]);
  StoreTiles(mem, out);
}
// execute a full DNN graph at memory level 0
ExecuteGraph(graph, 0, inputs, outputs);
```

Figure 8: Compilation routine of hierarchical tile-graph.

tions, such as the number of layers, memory capacity, and transaction width of each layer, can be obtained through a `MemLevels` interface (e.g., used in Figure 7). With this abstracted memory layer, it is easy to extend an existing accelerator with additional memory layers (e.g., host memory or SSD) as a new device, allowing it to handle very large tensors that may not fit in the single device memory (§5.4 for more details). WELDER's performance gain mainly comes from the bandwidth gap between memory layers. Thus, as long as a lower-level memory becomes the bottleneck and a high-level memory can hold the intermediate data tile, WELDER can automatically pipeline the inter-operator data transfer on the faster, high-level memory.

In order to execute a hierarchical tile-graph on a hardware accelerator, WELDER provides four computing interfaces: `Allocate`, `LoadTiles`, `ComputeTile`, and `StoreTiles` (listed in Table 1). The routine for executing a hierarchical tile-graph using these interfaces is shown in Figure 8. The process starts by executing the bottom-layer tile-graph (i.e., the full DNN graph). For each tile-graph, it first allocates the necessary workspace in the corresponding memory layer (using the `Allocate` interface) and loads the input tiles into this space (`LoadTiles`). Then, it executes all the nodes in the sub-graph in a topological order. If the current memory layer is the top level, the node is executed directly in the computing cores (`ComputeTile`). Otherwise, the execution of the upper-level tile-graph is called recursively. Finally, the result tiles in the current space are stored in the lower memory layer (`StoreTiles`). This execution routine can be used as both a code generation process or a runtime process, depending on whether a specific accelerator implements these computing interfaces as code emitters or executable function calls. In WELDER, they are currently implemented as code emitters to generate the accelerator-specific computing logic. By executing this recursive routine, the entire hierarchical tile-graph is unrolled and a full-model computation program with all the necessary optimizations is generated automatically.

4 Implementation

WELDER is implemented based on open-source DNN compilers, TVM [15], Roller [52] and Rammer [31]. It leverages TVM for writing kernel schedule, Roller for enumerating efficient tile configurations, and Rammer for the end-to-end graph optimization. WELDER's core mechanisms, including the tile-graph, tile propagation, scheduling algorithm, code generation, etc., are implemented in 5.2k lines of code. WELDER takes an ONNX graph as input and performs common graph optimizations such as constant folding and simple element-wise fusion. It then converts the optimized graph into a tile-graph for holistic memory scheduling optimization. WELDER is implemented on both CUDA and ROCm GPUs, and GraphCore IPU through the unified device interface (Table 1). For CUDA and ROCm GPUs, WELDER schedules data tiles on three memory layers: global memory (DRAM), shared memory, and register. To handle large images on CUDA GPUs and GraphCore IPU, we also extend their device memory by adding a host memory layer.

4.1 Hardware-aligned Tile Search

Enumerate efficient data tile size. WELDER takes into account several hardware-related factors that could impact the data access efficiency by introducing a penalty factor to the traffic cost model. First, if there is uncoalesced memory access, the total memory traffic will include the additional transactions required for these accesses. For instance, in CUDA GPUs, it is always preferable to use coalesced memory access for a contiguous 128 bytes of data (one transaction). Second, when there is insufficient parallelism due to a large tile size, the memory traffic is increased proportionally based on the utilization percentage of the computing cores. Finally, we add an infinite penalty if the total memory footprint of a given tile configuration exceeds the memory capacity. To avoid enumerating inefficient candidates, WELDER searches for output tiles by only enumerating the dimensions that can reduce traffic the most according to the cost model, and retrieves only top k candidates with the minimum traffic.

Decide aligned computation parallelism. In GPUs, the top-level operator-tiles that are executed in the same thread-blocks must agree on a unified block size (e.g., number of threads). To ensure this alignment, WELDER first enforces sufficient parallel tiles at the register level to align with the hardware parallelism (i.e., by enumerating hardware-aligned tiles). For example, in NVIDIA V100 GPUs, the tile number should be greater than 128, as each SM has 4 warp schedulers and each warp has 32 threads. We then determine the greatest common divisor among the tile numbers of all operators as the common thread-block size, if it is larger than the hardware parallelism (e.g., 128) and less than the maximum limitation (e.g., 1024). Otherwise, we set the block size to a number

that equals the hardware parallelism. Once the block size is decided, we bind all operator-tiles at the register level to these threads. If a single thread needs to run multiple tiles, we use TVM's virtual thread to bind them, thus allowing concurrent data access over all memory banks and avoiding bank conflicts.

Support TensorCore. WELDER leverages TensorCore to accelerate certain operators such as GEMM, BatchMatmul, and Convolution (using implicit GEMM [28]) on CUDA GPUs. We add annotations to these operators indicating which axes will be bound to CUDA's Warp-Level Matrix Operations. For top-level operator tiles, we bind them to warps (instead of threads) to perform MMA operations. Additionally, we introduce some extra constraints when enumerating tile sizes, such as ensuring that the number of threads is an integral multiple of the warp size and that the axes (M, N, and K) in each tile are an integral multiple of the fragment size of the MMA operations.

4.2 Code Generation and Compilation

WELDER's kernel generation is based on TVM. In particular, the register level tile connection is implemented using TVM's `compute_inline` schedule primitive. For shared memory level connection, we only use TVM to generate standalone kernels for each connected part above the shared memory, and then apply several additional passes to compose these standalone kernels into a single fused kernel.

Load/store rewriting. The standalone kernels generated by TVM load and store data from global memory. We rewrite these global memory accesses to shared memory accesses by adding an additional TIR [11] pass to TVM's lowering procedure. Additionally, we add memory fences to prevent race conditions and apply padding to handle bank conflicts in the buffers. As a result, the original global kernel can be transformed into a device function, which is included in the final fused kernel.

Block/thread index remapping. Some operators cannot be directly connected to others and require remapping of their `blockIdx` and `threadIdx` values. The `BlockIdx` remapping is used for operators such as `Transpose`. The remapping relationship is deduced from their tensor expressions. The `ThreadIdx` remapping is used to connect 2D thread blocks to 1D thread blocks. This is necessary when inter-thread reduction or TensorCore primitives require the use of a 2D thread block (both `threadIdx.x` and `threadIdx.y`), while others may use a 1D thread block (only `threadIdx.x`). A 2D thread block can be mapped to a 1D thread block as long as their total number of threads is equal.

Memory management. We manage all shared memory, including that allocated in each standalone kernel and the inter-operator reuse buffer, in a uniform manner. First, we analyze the liveness of each buffer based on the topology execution order and convert them into a sequence of allocation and free operations. We then use the bestfit algorithm to compute the offset for each shared memory allocation, taking into account any alignment requirements for data types and TensorCore operations (e.g., aligning to 32 bytes to avoid misaligned address access).

Compilation speedup. WELDER optimizes the compilation speed through parallel compilations and sub-graph caching. First, by taking advantage of the independence between configurations, WELDER can use multi-processes to build and evaluate each configuration in parallel. Second, in most DNN models, some sub-graph patterns often repeat for multiple times. To avoid the redundant optimization, WELDER leverages a sub-graph signature to cache each unique graph pattern. For example, in a 12-layer BERT model, we can cache the optimization result (kernel code and profiled latency) for the first layer and reuse it for all the remaining 11 layers.

5 Evaluation

5.1 Experimental Setup

We evaluate WELDER using three servers equipped with different accelerators: NVIDIA GPU, AMD GPU, and Graphcore IPU. Two servers are equipped with the NVIDIA GPUs. The first one is an Azure NC24s_v3 VM with Intel Xeon E5-2690v4 CPUs and NVIDIA Tesla V100 (16GB) GPUs, running on Ubuntu 16.04 with CUDA 11.0. The second one is a local workstation with Intel(R) Xeon(R) E5-2678 v3 CPUs and NVIDIA GeForce RTX 3090 GPUs, running on Ubuntu 18.04 with CUDA 11.3. The AMD GPU server is equipped with Intel Xeon CPU E5-2640 v4 CPU and AMD Radeon Instinct MI50 (16GB) GPUs, running on Ubuntu 18.04 with ROCm 5.2.3. The IPU server is an Azure ND40s_v3 VM with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 3.0.

DNN workloads. WELDER is evaluated on 10 DNN models with different model types, including CNNs, Transformer, CNN-Transformer and multilayer perceptrons (MLP), and most of which are the state-of-art in the corresponding tasks. Table 2 characterizes them with a comparison of their model types, tasks, and the years of publication. For all models in the table, we use their official PyTorch implementations without modification.

Baselines. We compare WELDER with several DNN frameworks, including PyTorch (v1.12) [10] and ONNXRuntime (v1.12) [8], as well as state-of-the-art DNN compilers such as Ansor (v0.9) [50] and Rammer [31]. We also compare WELDER with TensorRT (v8.4) [7], a vendor-specific inference library for NVIDIA GPUs. For transformer models,

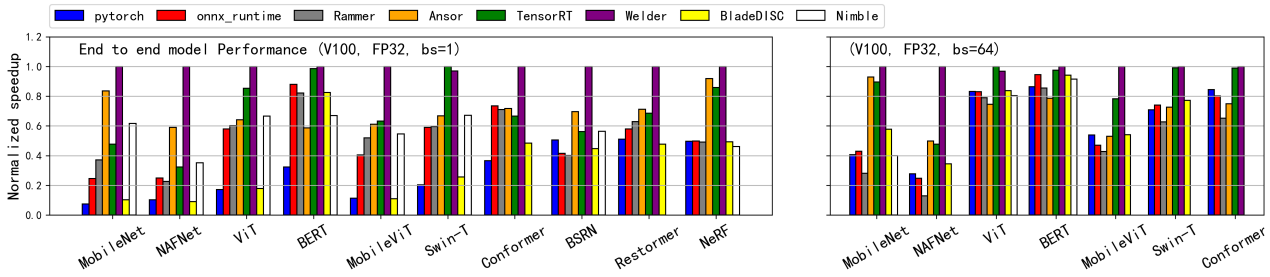

Figure 9: End-to-end model inference performance on NVIDIA V100 GPU (SIMT Core only). (left : batch size of 1, right : batch size of 64).

Model	Type	Task	Year
MobileNet [41]	CNN	Image Classification	2018
BERT [16]	Transformer	NLP	2018
ViT [17]	Transformer	Image Classification	2020
Conformer [20]	CNN+Transformer	Speech Recognition	2020
MobileViT [32]	CNN+Transformer	Image Classification	2021
Swin-Transformer [30]	Transformer	Image Classification	2021
NeRF [33]	MLP	3D-scene Generation	2021
NAFNet [14]	CNN	Image Restoration	2022
Restormer [48]	CNN+Transformer	Image Restoration	2022
BSRN [29]	CNN	Image Super-resolution	2022

Table 2: DNN models evaluated in WELDER.

we further compare WELDER with NVIDIA's FasterTransformer (v5.2) [2], a hand-crafted C++ library optimized for transformer models. We also include BladeDISC (v0.3.0) [1] that implements the latest AStitch [51] for the kernel fusion optimization. We also include Nimble [25] which implements multi-stream scheduling as a baseline on NVIDIA GPUs.

To evaluate a model on these baselines, we first trace the model in PyTorch and export it to the ONNX format. We then use this ONNX model as input to other frameworks, including WELDER, Ansor, ONNXRuntime, and TensorRT. For the ONNXRuntime, we use its CUDA execution provider and set its graph optimization level to "ALL" to achieve the best performance. For TensorRT, we use its Python API to build an engine for the input ONNX model. For Ansor, we set the total number of tuning trials to 800× the number of tasks in each model. For all frameworks, we place the input and output tensors in GPU device memory to avoid additional data movement costs. During evaluation, we first performe some warm-up iterations and then run each workload repeatedly for at least 5 seconds. We only report the average speed for each model, as we observe very little variation in all cases. The average performance (e.g., speedup) across models is calculated by geometric mean in all experiments.

5.2 Evaluation on NVIDIA GPUs

This section answers the following questions: 1) How does WELDER perform in comparison with state-of-the-art DNN frameworks or compilers? 2) To what extent can WELDER further boost performance with TensorCores? 3) Can WELDER automatically discover new optimization patterns beyond previous expert-designed fusion rules? 4) How well does WELDER improve both the memory and computational efficiency? 5) What is the search efficiency of WELDER's holistic optimization?

End-to-end performance. Figure 9 shows the performance of WELDER and other baselines for batch size of 1, expressed as the normalized speedup relative to the best result. The geometric mean speedup that WELDER achieves over DNN frameworks is 4.29× for PyTorch and 2.07× for ONNXRuntime. PyTorch does not perform well for models with batch size 1 due to high Python overhead in its computation graph. In contrast, ONNXRuntime is a more optimized framework that removes Python overheads and implements pattern-based graph optimizations. WELDER also outperforms Rammer by 1.96×, as Rammer can only fuse independent parallel kernels instead of dependent ones through shared memory. When evaluating BladeDISC (implementing AStitch), we notice that it encounters "unsupported operator" failures and falls back to PyTorch runtime for the majority of models. For models without encountering any failure (including BERT, MobileNet, BSRN and NeRF), WELDER is 2.70× faster than BladeDISC. Regarding the Nimble baseline, WELDER achieves an average speedup of 1.79×, excluding the models where Nimble fails to execute.

Ansor improves DNN performance by generating high-performance tensor programs and using rule-based fusion across operators at the register level (e.g., Matmul+BiasAdd, Conv2D+ReLU). However, it cannot exploit further memory reuse opportunities, leading to an average performance gap of 1.44× compared to WELDER. This is evident in CNN models such as NAFNet (1.70×) and BSRN (1.43×), which mainly consist of convolutions with relatively small channels that can be well optimized by WELDER. WELDER also outperforms Ansor by a significant margin on Transformer-based models such as BERT (1.71×), Swin-Transformer (1.45×), and ViT (1.56×), due to Ansor's inability to fuse patterns like LayerNorm or Softmax in the attention block. Furthermore, WELDER performs well for CNN+Transformer models, achieving speedups of 1.64×, 1.39×, and 1.29× on MobileViT, Conformer, and Restormer, respectively, as WELDER can cover fusion opportunities in both the CNN and Transformer parts of these models. We also observe that

WELDER only slightly outperforms Ansor on NeRF (1.09×), mainly due to that the compute-intensive MLP dominates the computation without further optimization opportunities.

Finally, TensorRT is a specialized DNN inference library provided by NVIDIA with highly optimized operators. WELDER is comparable to TensorRT on popular transformer models such as BERT (1.02×) and Swin-T (0.97×). This is because TensorRT has incorporated expert-designed fusion rules and in-house kernels for some popular models, including transformer-based models, thereby leaving limited room for further optimization. In contrast, WELDER identifies optimization patterns automatically and achieves performance that is on par with TensorRT, despite relying on less performant kernels for compute-intensive operators. It is worth noting that kernel optimization is complementary to WELDER, and further optimized kernels may offer even greater benefits for WELDER. Additionally, for newer and more diverse models such as NAFNet, WELDER has demonstrated superior performance to TensorRT, with speedups of up to 3.09× due to its generality. Overall, our system outperforms TensorRT with an average speedup of 1.47×.

Figure 9 also shows the normalized performance for a larger batch size of 64. The last three models in Table 2 are unable to be traced on PyTorch with large batch sizes due to their use of large input size. Under this setting, WELDER continues to outperform all other baselines, providing an average speedup of 1.83× over PyTorch, 1.90× over ONNXRuntime, 2.1× over Rammer, 1.57× over BladeDISC, 1.49× over Nimble, 1.47× over Ansor, and 1.21× over TensorRT, respectively. We observe that for large batch sizes, frameworks using CUDA libraries perform much better, compared to the results for a batch size of 1. This leads to smaller speedups over PyTorch, ONNXRuntime, and TensorRT for WELDER, while the speedup over Ansor remains similar to the results for a batch size of 1.

Performance with TensorCore. The faster computing throughput of TensorCore can put greater pressures on memory access. To understand the optimization behaviors when running on TensorCore, we convert our benchmark models (both weights and activations) to half-precision float type (FP16) with PyTorch, as TensorCore only supports FP16. This is done using the tools in the `onnxconverter_common` package [9], with the exception for TensorRT, which converts through its own converter as it often produces better results.

Figure 10 shows the performance comparisons of WELDER with other frameworks using TensorCore for batch sizes of 1 and 64. For the 10 cases that use a batch size of 1, WELDER outperforms PyTorch, ONNXRuntime, BladeDISC, Nimble, Rammer, and TensorRT. The averaged speedup is 7.18× (up to 21.4× on MobileNet) to PyTorch, 3.08× (up to 8.72× to on Conformer) to ONNXRuntime, 5.29× (up to 16.9× on MobileNet) to BladeDISC, 2.72× (up to 5.58× on NeRF) to Nimble, 2.76× (up to 5.42× on NAFNet) to Rammer, and 1.53× (up to 2.98× on NAFNet) to TensorRT, respectively.

Model	DT	BS	WELDER(ms)	FT-CPP(ms)
BERT	FP32	1	3.13	3.15
BERT	FP32	64	118.6	119.8
BERT	FP16	1	1.49	1.50
BERT	FP16	64	24.82	22.29
ViT	FP32	1	1.33	1.96
ViT	FP32	64	15.29	15.68
ViT	FP16	1	1.09	1.89
ViT	FP16	64	4.79	5.15
swin-T	FP32	1	2.59	2.38
swin-T	FP32	64	66.13	72.62
swin-T	FP16	1	1.43	1.60
swin-T	FP16	64	23.12	28.67
geometric mean			6.71	7.46

Table 3: Performance for WELDER and FasterTransformer

For the remaining 7 cases in Figure 10 that uses a batch size of 64, WELDER outperforms PyTorch by 1.98×, ONNXRuntime by 2.13×, BladeDISC by 1.97×, Nimble by 3.84×, Rammer by 3.45× and TensorRT by 1.16× respectively.

Some of the speedups are much larger than the ones achieved on SIMT cores. Especially for the NeRF model, WELDER outperforms TensorRT by 2.34× on TensorCore, while the speedup on SIMT cores is only 1.16×. This is mainly because TensorCore can greatly accelerate the compute-intensive part of the model, making the optimization of the remaining memory-intensive part more critical.

Note that Ansor is not included in this experiment as it does not support TensorCore. For a fair comparison, we disable WELDER's TensorCore feature and evaluate these FP16 models on SIMT cores by comparing with Ansor in Figure 11. It shows a slightly higher speedups (1.74× on average and up to 2.82×) compared with the ones in FP32.

Performance on another NVIDIA GPU We also conduct evaluations on RTX-3090, another widely-used GPU, which utilizes a distinct Ampere architecture. The RTX-3090 exhibits various new features compared to the V100, including advancements in memory load and TensorCore instructions, as well as a different number of streaming multiprocessors (SM). For the sake of conciseness, we solely compared WELDER with TensorRT on the RTX-3090, as TensorRT consistently delivers superior performance compared to other baselines on NVIDIA GPUs. The results, depicted in Figure 12, illustrate that WELDER outperforms TensorRT with an average speedup of 1.40×, calculated using the geometric mean of all 34 test cases. Notably, this speedup is similar to the one observed on the V100 GPU, which amounted to 1.36×, thereby highlighting WELDER's adaptability across diverse GPU architectures.

Patterns automatically discovered. WELDER automatically discovers around 300 different fused subgraphs, which is counted by unique operator types under all 34 compiled test cases of the 10 models. Among them, 89 patterns contain at least two reduction-based operators which cannot be covered

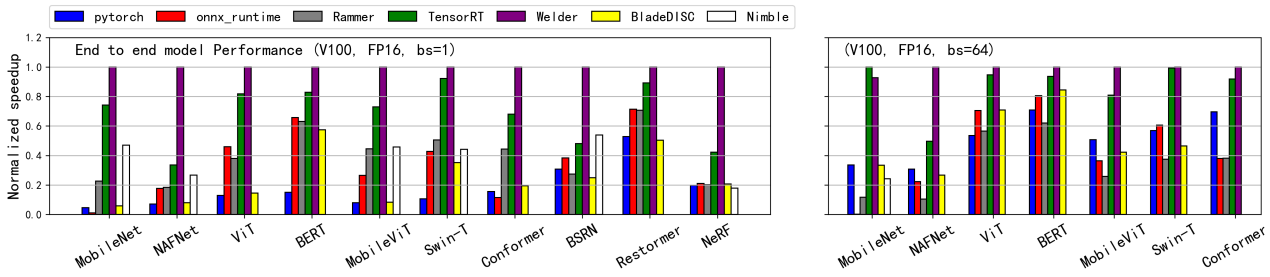

Figure 10: End-to-end model inference performance on NVIDIA V100 GPU (TensorCore enabled). (left : batch size of 1, right : batch size of 64).

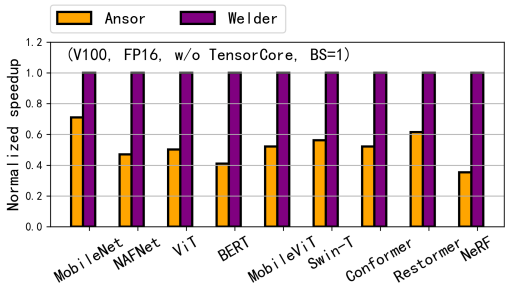

Figure 11: Comparing with Ansor under FP16 w/o TensorCore

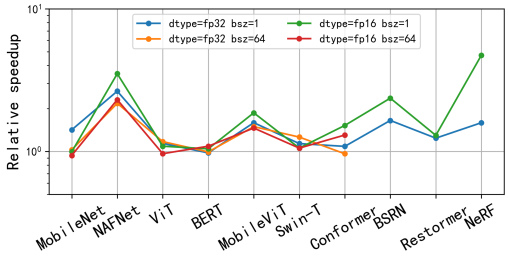

Figure 12: Comparing with TensorRT on NVIDIA RTX-3090

by simple element-wise fusion rule in Ansor. To the best of our knowledge, many of these patterns are uncommon fusion patterns that have not been explored by manually-designed rules or automatic fusion optimizations. Figure 4 illustrates two examples of such patterns, which fuse multiple Convolution or MatMul (i.e., Dot) operators with other memory-intensive operators into a single kernel. The number of operators fused in each pattern ranges from 2 to 48 and can achieve an average speedup of 1.87× (up to 5.4×) compared to basic fusion methods such as those used in Ansor. The most common pattern has been used 191 times in all models.

Such a general fusion capability often allows WELDER to outperform the model-specific implementations optimized by experts. For example, FasterTransformer [2] is a manually-optimized benchmark for transformer models from NVIDIA. It supports both element-wise fusion, such as BiasAdd+Transpose, and non-element-wise fusion, such as Layernorm+Softmax. In WELDER, all these patterns can be automatically fused. Even more, WELDER can further fuse Q*K with the following Softmax in the attention block when the sequence length is not long (e.g., they are fused

Fused operators	# Ops
DepthwiseConv2dNative Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply **Convolution** Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply **Convolution** Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply **Convolution** Broadcast Add Broadcast Divide Erf Broadcast Add Multiply Broadcast Multiply Concat **Convolution** Broadcast Add	48
Dot Relu **Dot** Relu **Dot** Relu **Dot** Relu **Dot** Relu **Dot** Relu **Dot**	13

Table 4: Examples of fusion patterns discovered by WELDER.

in BERT where the sequence length is 128, but are not fused in Conformer where the sequence length is 512, this is automatically decided by WELDER).

For the three models supported by FasterTransformer, we compare its performance with WELDER in Table 3. In general, WELDER achieves an average speedup of 1.11× (up to 1.73× on ViT) over FasterTransformer. Based on our profiled data, The notable speedup for ViT under batch size of 1 can be attributed to a convolution operator with a non-conventional shape, where both stride and kernel size are 32 (ViT's patch size). For this single operator, WELDER's generated kernel is 4.4x faster. This highlights WELDER's adaptability in managing new operator shapes or model patterns.

Another example is NeRF, a popular 3D scene generation model that is typically implemented as a 7-layer MLP. To take full advantage of GPUs, domain experts often need to implement such models from scratch to achieve better fusion result (e.g., fully-fused MLP in [35]). With WELDER, we can automatically fuse this 7-layer MLP into a single GPU kernel. The generated kernel uses TensorCore for the first 6 layers and uses SIMT Core for the output layer, with all intermediate results stored in shared memory. We observe that our automatic fusion result can achieve a similar speedup (over 5×) to the values reported in [35] (we are unable to evaluate their code [34] as it does not support V100 GPUs).

Finally, for CNN models such as NAFNet, BSRN, and MobileNet, WELDER is able to fuse different types of convolutions with other operators (e.g., Pooling, PixelShuffle, etc.). For example, in NAFNet, our system can fuse back-to-back pointwise convolutions together with the normalization

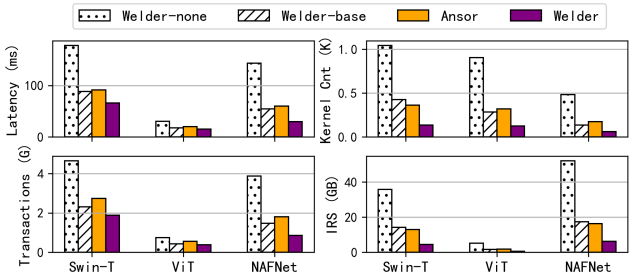

Figure 13: Latency, GPU kernel count, global memory transaction executed and intermediate result size (IRS) For 3 selected models (FP32, batch size 64).

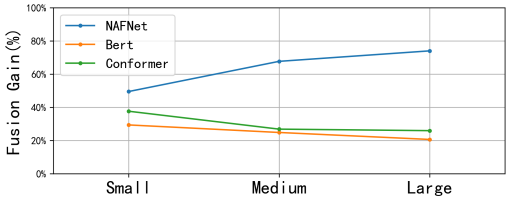

Figure 14: Varying input size, comparing with Welder-base.

operations between them. Another interesting pattern is in models with multiple separable convolution layers, where each layer consists of two operations: a depthwise convolution (DWConv) and a pointwise convolution (PWConv). WELDER is able to determine the optimal fusing order for these two types of operators based on their operator configurations. For example, on the top layers where the feature maps are large and the number of channels is small, WELDER constructs a DWConv+PWConv fusion group because it is better to cache a complete feature map in shared memory. In contrast, on the bottom layers, WELDER constructs a PWConv+DWConv fusion group which caches a complete channel for DWConv to reuse, as the feature map becomes smaller.

Ablation and sensitivity study. To demonstrate the benefits of the holistic memory optimization provided by WELDER, we create two variants of WELDER: "WELDER-none" disables all inter-operator tile connection and only searches for intra-operator schedules, and "WELDER-base" only enables inter-operator tile connection at the register layer. We also include Ansor in this experiment, as it is another codegen-based approach similar to ours. As shown in Figure 13, enabling register layer tile connection, WELDER-base reduces latency by an averaged 52% (i.e., 2.08× speedup), kernel launch count by 67% , global memory transactions by 52% and intermediate result size (IRS) by 66% compared with WELDER-none. Note that the metrics of WELDER-base is similar to that of Ansor, demonstrating the efficiency of our general tile-based memory scheduling compared with the rule-based fusion in Ansor. Moreover, by enabling tile connection at shared memory layer, WELDER is able to further reduce latency by an averaged 29% (with up to 1.82x speedup), kernel launches by 60%, transactions by 25% and IRS by 65% compared with WELDER-base. Note that the reduction of memory transactions is less than the reduction of IRS, because memory access on the model weights part cannot be optimized by fusion.

Model	Ansor time(s)	Ansor Trials	WELDER Time(s)	WELDER Trials
BERT	15285	8000	244	651
Mobilenet	45527	25600	561	927

Table 5: Compilation time of Ansor and WELDER

Model	Ansor	WELDER	TensorRT
Resnet50	2.403	2.327	2.351
Resnet18	1.071	1.094	1.158
UNet	8.670	9.251	4.429
VGG16	4.267	4.123	2.584

Table 6: Performance on compute intensive models

In addition, we conducted a sensitivity study by varying the input sizes of three selected models: BERT (128-512 text length), Conformer (128-512 audio frames), and NAFNet (256x256-1024x1024 image input). The results, as depicted in Figure 14, reveal that the fusion gain significantly increases for NAFNet when employing larger images. Conversely, the gain diminishes for the other two transformer-based models. This discrepancy can be attributed to the fact that transformer-based models exhibit quadratic computational growth with respect to the input sequence length, thereby reducing their memory-intensive nature.

Compilation time. Table 5 compares WELDER's compilation time against Ansor, which is a search-based compiler requiring many tuning and profiling trails. We chose not to include other baselines in the comparison since they directly invoke library kernels, thereby eliminating the need for extra time dedicated to tuning and code generation. It shows that the end-to-end compilation speed of WELDER is more than an orders of magnitude faster than Ansor. This is because Ansor generates a very large search space for all the operators, and implicitly optimizes data reuse through machine learning-based tuning. This often requires a large number of tuning trials (e.g., 800 per operator in our evaluation) and has additional overheads to train a cost model on the fly. In contrast, WELDER decomposes the optimization space using a layered scheduling policy and searches for efficient tiling configurations using an analytic cost model to estimate traffic costs. As a result, WELDER requires significantly fewer tuning trials (20 per subgraph in our evaluation) than Ansor.

Performance on compute intensive models. Traditional models like ResNet [21], VGG [43], and UNet [40] are typically dominated by some large operators such as convolution. For these compute intensive models, although WELDER mainly focuses on memory access optimization, WELDER can mostly achieve comparable performance to state-of-the-art baselines like TensorRT. This is because WELDER

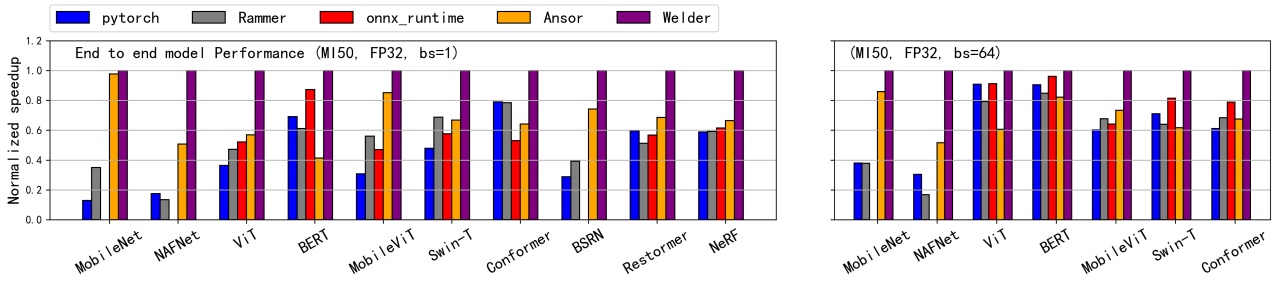

Figure 15: End-to-end model inference performance on AMD ROCm MI50 GPU (left : batch size of 1, right : batch size of 64).

can still generate high performance single operators (using the multi-level tiling abstraction, which is similar to Ansor [50] or Roller [52]) although there might be few chances to connect the tile at a higher memory level. However, for some convolution operators, existing libraries like cuDNN [4] implement them using an optimized numerical algorithm (e.g., winograd [26]), which are difficult to automatically derive from tensor expressions. This can result in WELDER performing worse than TensorRT if there is no additional memory optimization room to compensate for this gap. For example, Table 6 compares the performance of WELDER, Ansor, and TensorRT on four such models. For ResNet, both systems achieve comparable performance, as the majority of convolution operators in this model perform better when implemented with the DirectConv algorithm (which is supported by both Ansor and WELDER) instead of winograd. However, for UNet and VGG16, the dominant convolution operators are mostly implemented using winograd in TensorRT, and there are no further fusion opportunities for WELDER to exploit, resulting in better performance for TensorRT. Given that this is orthogonal to WELDER's optimization, we leave the support of the winograd algorithm (by rewriting tensor expressions) to our future work.

5.3 Evaluation on AMD ROCm GPUs

We evaluate the efficiency of WELDER on AMD ROCm GPUs by comparing its performance with PyTorch, ONNXRuntime and Ansor. TensorRT and AStitch are not included because they are exclusive to NVIDIA GPUs. Figure 15 shows the end-to-end performance of the 10 DNN models. Compared with PyTorch, ONNXRuntime and Rammer, WELDER can outperform them by an average of 2.62×, 1.71× and 2.14×, respectively. Compared to Ansor, WELDER achieves an average performance improvement of 1.53×. Figure 15 also shows the performance comparison with a larger batch size of 64, where WELDER outperforms PyTorch, ONNXRuntime, Rammer and Ansor by an average of 1.69×, 1.23×, 1.86× and 1.47×, respectively. Note that we have excluded some CNN models for ONNXRuntime as they fail to execute on it. We notice that WELDER's speedup on MI50 is slightly smaller than that of V100, this is because MI50's peak FLOPS is weaker than V100's, while its peak bandwidth is higher, according to the official data-sheet. Such difference makes the workload more compute-intensive on MI50, leaving less optimization chances for memory access optimization.

Model	Image Size	Device	WELDERBase(s)	WELDER(s)
UNet	8k*8k	GPU	38.2	14.5
VGG16	8k*8k	GPU	15.7	8.30
UNet	2k*2k	IPU	31.1	8.56
VGG16	2k*2k	IPU	4.98	1.61

Table 7: Scale-up large DNN models to host memory

5.4 Scale-up with Host Memory

WELDER's abstracted device layer allows us to extend the memory hierarchy to support large DNN tasks. For example, in cases where classical CNN models like UNet or VGG16 are used to process high-resolution medical images [42], a single tensor from some layers is often too large to fit in the GPU memory. In these scenarios, tensor-based memory swapping optimization techniques, such as SwapAdvisor [22] or Capuchin [37], may not be effective due to the large tensor granularity. WELDER addresses this issue by generating a tile-based execution plan on the extended memory hierarchy through holistic traffic optimization. This approach allows us to load a data tile from the host memory, compute several connected operator tiles by reusing the data in device memory, and store the result back, as if it was being processed on a single device. To evaluate the efficiency of this scheduling approach, we compared WELDER with a variant that only disables data reuse at the device memory layer.

Scale-up GPUs. As a preliminary experiment, Table 7 shows the performance of WELDER when scaling up UNet and VGG16 on large image data by augmenting the GPU memory with a host memory layer. As the results show, by enabling tile-connection at the device memory layer, WELDER is able to achieve average speedups of 2.63× and 1.89× for the two models, respectively. It also reduces host memory transfer by 3.11× and 2.90×. Note that the ratios of reduced memory traffic are higher than the actual speedup, as we have implemented double buffering (along with pinned memory and CUDA streams) to overlap some memory transfer with computation.

Scale-up GraphCore IPU. We also perform a preliminary evaluation of WELDER's ability to scale up on the Graphcore IPU [3], which is a DNN accelerator with a distinct architecture from NVIDIA and AMD GPUs. The IPU is equipped with massively parallel MIMD processors and a relatively small device memory (i.e., 300MB), which poses a challenge for it to handle even medium-sized tasks. We apply the same tile-based scheduling to the two models for the IPU and set the input image size to 2048*2048 to adapt to the IPU's memory capacity. The results in Table 7 show that WELDER's optimization is able to achieve average speedups of 3.63× and 3.09× for the two models, respectively. This improvement ratio is higher than that of the GPU, which is mainly due to that we disable the double-buffer optimization for the IPU due to its limited memory.

6 Discussion

WELDER's design and implementation mainly focuses on static models. For dynamic model execution, there are two practical ways to address this. First, the dynamic graph can be transformed into static sub-graphs through JIT compilation, such as PyTorch JIT compile, which has become a standard practice in PyTorch 2.0. Then, WELDER can concentrate on optimizing the static sub-graphs, which are typically the computationally dominant part. Second, even though tensor shapes may be dynamic, the internal tile in each operator can be statically determined. This presents an opportunity for WELDER to generate a static tile-level fusion plan but leave the number of parallel tasks determined by the input tensor shape.

7 Related Work

Compiler optimization like operator fusion is a widely-used technique in DNN computation to reduce kernel launch overhead and improve data locality in faster memory. Compilers such as TVM [15], Ansor [50], XLA [12], DNNfusion [36] all support operator fusion at register level. Other compilers try to further fuse operators at shared memory, relying on either fusion rules for a set of known operator types (e.g., AStitch [51], Apollo [49], DeepCuts [24]) or specific template for a few operator combinations (e.g., Bolt [47]). Specialized DNN runtimes such as TensorRT [7] and ONNXRuntime [8] have incorporated expert-designed fusion rules for some common patterns in popular models such as the transformer-based models. In contrast, WELDER works for general operators implemented in tensor expressions without the assumption on operator types and decides on the best fusion memory layer automatically. This is because an operator's resource usage behavior (memory- or compute-intensive) often depends on its shape, and therefore the fusion decision.

Systems like Rammer [31], HFuse [27], Nimble [25], etc., exploit better hardware parallelism utilization and reduce kernel launches by either horizontal fusion or scheduling parallel tasks through multi-stream and CUDA graph. WELDER builds upon Rammer by further exploring a complementary optimization to these systems, i.e., holistic memory optimization with a vertical fusion, resulting in a further speedup for memory-intensive models.

Ansor [50] and Roller [52] are representative tensor compilers that are focusing on intra-operator optimization through either loop optimization or tiling optimization. Especially, Roller [52] and Triton [44] also utilize the concept of tile to optimize kernel performance (e.g., intra-operator data reuse). In contrast, WELDER complements them by optimizing for intra- and inter-operator memory access holistically. WELDER generalizes the *tile* concept in Roller into a *tile-graph* abstraction, exposes a holistic tile-level scheduling space, and proposes an efficient scheduling mechanism over the holistic space and the explicit memory hierarchy.

Some works optimize for a specific pattern regarding to a type of models with more aggressive operator fusions, such as fully-fused MLP for the NeRF model [35], manually fused kernels for CNN models [46], and attention fusion for transformer models [2, 18]. Our evaluation shows that WELDER can achieve most of these fusions automatically and even produce new fusion patterns to help further optimization.

Moreover, kernel fusion techniques have been used in traditional image processing [38, 39] or HPC [45] areas. These efforts usually leverage domain-specific fusion rules for their workload. WELDER focuses on DNN workload, while it is applicable for general operators represented by tensor expressions. It is also potentially helpful for workload that can be implemented in tensor expressions in other domains.

8 Conclusion

By observing that modern DNN models are becoming increasingly memory intensive, we introduced WELDER, a DNN compiler that optimizes the execution efficiency based on a new tile-graph abstraction. WELDER is able to holistically optimize efficient intra- and inter-operator data reuse across multi-level memory hierarchy. WELDER is the first to unify all common operator fusions into a single framework, allowing for the discovery of 89 uncommon fusion patterns, with the largest one fusing 48 operators into a single kernel. This generality enables WELDER to significantly outperform state-of-the-art baselines. More importantly, WELDER provides a systematic approach to take advantage of emerging trends in the memory hierarchy, such as larger and more connected on-chip memory, in the future AI accelerators.

Acknowledgement

We thank anonymous reviewers and our shepherd, Prof. Byung-Gon Chun, for their extensive suggestions. This work was partially supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

A Artifact Appendix

Abstract

WELDER provides end-to-end DNN model compilation with its new tile-graph abstraction. This artifact reproduces the main results of the evaluation on NVIDIA V100 GPU.

Scope

This artifact will validate the following claims:

- End-to-end model performances. By reproducing the experiments of Figure 9, Figure 10, Figure 11, Table 3 and Table 6.
- Motivation experiments in Figure 1 and Figure 2.
- Ablation study in Figure 13.
- Compilation time in Table 5.
- GPU stale out experiments in Table 7.

Contents

This artifacts includes all the source code to implement WELDER. We provide a docker file to setup environments. For each figure and table mentioned above, we provide a script to reproduce its result. Since there are more than 50 model test cases to compile to fully reproduce the results, which will cost a long time (especially for the Ansor's baseline), we also provide pre-compiled logs and models for NVIDIA V100 GPU. Please refer to the README.md file in the repository for more details.

Hosting

The artifact is hosted at github repository[1]. Please use git to clone the repository and checkout to the `osdi2023welder` branch.

Requirements

This artifacts requires a NVIDIA V100 GPU with CUDA driver supporting CUDA runtime larger than 11.0.

[1] https://github.com/microsoft/nnfusion/tree/osdi2023welder

References

[1] BladeDISC. https://github.com/alibaba/BladeDISC.

[2] FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[3] IPU PROGRAMMER'S GUIDE. https://www.graphcore.ai/docs/ipu-programmers-guide.

[4] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[5] NVIDIA cutlass. https://github.com/NVIDIA/cutlass.

[6] NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/.

[7] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[8] ONNX Runtime. https://github.com/microsoft/onnxruntime.

[9] onnxconverter_common. https://github.com/microsoft/onnxconverter-common.

[10] PyTorch. https://pytorch.org/.

[11] TensorIR. https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872.

[12] XLA. https://www.tensorflow.org/xla.

[13] 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, GA, 2016. USENIX Association.

[14] Liangyu Chen, Xiaojie Chu, Xiangyu Zhang, and Jian Sun. Simple baselines for image restoration. *arXiv preprint arXiv:2204.04676*, 2022.

[15] 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, 2018. USENIX Association.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[18] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[19] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. STOC '72, page 143–150, New York, NY, USA, 1972. Association for Computing Machinery.

[20] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[23] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[24] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile GPU workloads. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, pages 190–205. ACM, 2021.

[25] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In *NeurIPS*, 2020.

[26] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[27] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27. IEEE, 2022.

[28] Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76, 2016.

[29] Zheyuan Li, Yingqi Liu, Xiangyu Chen, Haoming Cai, Jinjin Gu, Yu Qiao, and Chao Dong. Blueprint separable residual network for efficient image super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 833–843, June 2022.

[30] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *CoRR*, abs/2103.14030, 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] Sachin Mehta and Mohammad Rastegari. Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer. *arXiv preprint arXiv:2110.02178*, 2021.

[33] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[34] Thomas Müller. tiny-cuda-nn, 4 2021.

[35] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372*, 2021.

[36] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 883–898. ACM, 2021.

[37] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating (ASPLOS'20)*, 2020.

[38] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *21st International Workshop on Software and Compilers for Embedded Systems, (SCOPES'18)*, pages 76–85. ACM, 2018.

[39] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*, pages 242–253. IEEE, 2019.

[40] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[41] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[42] Nahian Siddique, Sidike Paheding, Colin P. Elkin, and Vijay Devabhaktuni. U-net and its variants for medical image segmentation: A review of theory and applications. *IEEE Access*, 9:82031–82057, 2021.

[43] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[44] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[45] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 191–202. IEEE Computer Society, 2014.

[46] Xueying Wang, Guangli Li, Xiao Dong, Jiansong Li, Lei Liu, and Xiaobing Feng. Accelerating deep learning inference with cross-layer data reuse on gpus. In *European Conference on Parallel Processing*, pages 219–233. Springer, 2020.

[47] 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 *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.

[48] Syed Waqas Zamir, Aditya Arora, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Ming-Hsuan Yang. Restormer: Efficient transformer for high-resolution image restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5728–5739, 2022.

[49] 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 *Proceedings of Machine Learning and Systems*, volume 4, pages 1–19, 2022.

[50] 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.

[51] 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, page 359–373, New York, NY, USA, 2022. Association for Computing Machinery.

[52] 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.