AKG: Automatic Kernel Generation for Neural Processing Units using Polyhedral Transformations

Jie Zhao
yaozhujiajie@gmail.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Bojie Li
Wang Nie
bojie.li@huawei.com
peter.nie@huawei.com
Huawei Technologies Co., Ltd.
Beijing, China

Zhen Geng
gengzhen1@huawei.com
Huawei Technologies Co., Ltd.
Hangzhou, China

Renwei Zhang
zhangrenwei1@huawei.com
Huawei Technologies Co., Ltd.
Beijing, China

Xiong Gao
Bin Cheng
xiong.gao@huawei.com
ckey.chengbin@huawei.com
Huawei Technologies Co., Ltd.
Hangzhou, China

Chen Wu
c.wu@huawei.com
Huawei TCS Lab
Shanghai, China

Yun Cheng
Zheng Li
chengyun9@huawei.com
lizheng53@huawei.com
Huawei Technologies Co., Ltd.
Hangzhou, China

Peng Di*
Kun Zhang†
dipeng1982@gmail.com
tegkunzhang@tencent.com
Huawei Technologies Co., Ltd.
Beijing, China

Xuefeng Jin
Huawei Technologies Co., Ltd.
Shenzhen, China
jinxuefeng@huawei.com

Abstract

Existing tensor compilers have proven their effectiveness in deploying deep neural networks on general-purpose hardware like CPU and GPU, but optimizing for neural processing units (NPUs) is still challenging due to the heterogeneous compute units and complicated memory hierarchy.

In this paper, we present AKG, a tensor compiler for NPUs. AKG first lowers the tensor expression language to a polyhedral representation, which is used to automate the memory management of NPUs. Unlike existing approaches that resort to manually written schedules, AKG leverages polyhedral schedulers to perform a much wider class of transformations, and extends the semantics of the polyhedral representation to combine complex tiling techniques and hierarchical fusion strategies. We also implement the domain-specific optimization of convolution in AKG. Moreover, to achieve the optimal performance, we introduce complementary optimizations in code generation, which is followed by an auto-tuner.

We conduct extensive experiments on benchmarks ranging from single operators to end-to-end networks. The experimental results show that AKG can obtain superior performance to both manual scheduling approaches and vendor provided libraries. We believe AKG will cast a light on the follow-up compiler works on NPUs.

CCS Concepts: • **Software and its engineering → Source code generation**.

Keywords: neural networks, neural processing units, polyhedral model, code generation, auto-tuning

ACM Reference Format:
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. 2021. 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), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454106

*Peng Di is now with Ant Group, Hangzhou, China
†Kun Zhang is now with Tencent Penglai Lab, Beijing, China.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454106

1 Introduction

Deep learning (DL) frameworks like TensorFlow [1], Pytorch [53], MXNet [11] and CNTK [59] express deep neural

networks as directed acyclic graphs of tensor operations. These frameworks provide promising performance for many applications by leveraging highly optimized vendor libraries and making transparent use of architectures, but fall short in supporting custom operators invented by users/high-level optimizing engines, exploiting transformations across operators, and tuning for divergent sizes and shapes.

A variety of optimizing compilers for tensor programs, *e.g.*, TVM [12], Tensor Comprehensions (TC) [62] and Tiramisu [4], have been devised to address these challenges. However, the ever increasing data and computation make DL a voracious appetite for computing power, resulting in the hardware race between tech giants to pursue emerging NPUs, like Google TPU [33], Graphcore IPU [32], Cerebras WSE [56] and Huawei Ascend [43]. Although some of these tensor compilers declare the portability to different architectures, most of them only target CPU/GPU/FPGA architectures.

Deploying DL models on modern NPUs [33, 43, 44] is quite challenging due to (1) the conflicting demands of parallelism and spatial/temporal locality between heterogeneous compute units, (2) the requirement of effective storage management between hierarchical memories, and (3) the difficulty of modeling non-trivial optimizations that are not present in general-purpose architectures. Although an expert can approach near-optimal performance through manual tuning, optimizing hand-written programs is error-prone, difficult to debug, and rarely scales with the increasing problem complexity. Developing an optimizing compiler has thus become a crucial obstacle of an NPU's ecosystem.

In this paper, we present an approach, called Automatic Kernel Generation (AKG), for NPUs to accelerate DL models. We inherit the graph engine and the domain-specific language (DSL) of TVM [12] for expressing tensor computations, and focus on the operator-level optimization and code generation. This isolates the hardware-specific transformations from high-level graph optimizations, allowing the reuse of existing functionalities like range inference, data layout transformations and abstraction level lowering and minimizing the engineering cost. AKG also provides DL frameworks with the ability to fuse any subgraphs into fewer operators.

The workflow of our tensor compiler branches from TVM by converting the HalideIR [55] lowered from the DSL to a polyhedral intermediate representation (polyhedral IR) [20], enabling the parametric range specification that is not accessible in TVM but essential in some scenarios, *e.g.*, dynamic shapes. Rather than resorting to manual schedule templates, AKG leverages versatile polyhedral scheduling algorithms that compute new schedules by solving integer linear programming (ILP) problems [9, 17] to exploit parallelism and locality of programs simultaneously, allowing the systematic formulation of ILP problems along with the flow of computation between heterogeneous compute units. This not only saves human efforts, but also captures a wider set of transformations than Halide [55] and TVM [12].

Another reason why writing manual schedules for NPUs is non-trivial is because the set of schedule primitives of TVM does not completely fit an NPU's architecture, making the management of dataflow across the memory hierarchy a difficult task. We leverage and extend the semantics of the polyhedral IR [20] to further widen the optimization space of AKG, facilitating an efficient storage management strategy. Our extension to the polyhedral IR models novel combinations of overlapped tiling [38, 69] and loop fusion [34, 48] which is impossible in existing polyhedral compilers [4, 40, 62], maximizing the use of our target architecture by triggering more aggressive storage optimizations.

Lowering to the polyhedral IR also solves the challenge to model domain-specific optimizations including the *img2col* transformation [21, 67] and *fractal* transformation [71] that are not expressible in either polyhedral optimizers [4, 40, 62] or semi-automatic compilers [12, 55]. We introduce a pattern-specific pass that converts the polyhedral IR into a new one, with the aforementioned transformations fully automated by relating the polyhedral IRs before and after this pass.

Finally, we postpone some low-level transformations that go beyond the power of polyhedral compilation to the code generator as complementary optimizations; a learning-based auto-tuner is also used to achieve the optimal performance. Unlike the enhancements of Halide [2, 42, 50] and TVM [13, 72, 73] that still rely on manual efforts, the AKG compiler implements a fully automatic workflow.

We implement AKG as the optimizer for the Huawei Ascend chips [43] and demonstrate its effectiveness by conducting extensive experiments. When experimenting on single operators, our approach can achieve comparable performance to the code manually optimized by experts and a mean speedup of 1.6× over TVM. AKG can outperform the two baselines by 1.3× and 5.6× on average, respectively, when optimizing subgraphs of DL models, and exceeds that of TVM by 20.2% in the case of end-to-end networks.

The contributions of our work are as follows.

- We present AKG, a fully automatic, end-to-end tensor compiler for NPUs, which significantly reduces the software development life cycle from months to hours.
- AKG achieves much better performance using a rich set of program transformations, which is further reinforced by complex tiling and fusion techniques and domain-specific transformations.
- Our approach enables the automation of managing complicated memory hierarchy, which effectively addresses the productivity gap between DL models and heterogeneous architectures.

2 Background

A DL optimizer is typically composed of a graph engine and a tensor compiler. The graph engine takes a model from DL frameworks, represents it using a computation graph, and outputs an optimized graph by applying high-level dataflow

rewriting transformations. The tensor compiler deploys each fused operator in the graph to the target machine by means of loop transformations, effective hardware binding and memory management, and code generation strategies. Existing approaches can be categorized into two groups.

Writing manual schedule templates. The idea of this approach is to separate *schedule* and *compute*, allowing users to describe their algorithm using *compute* without considering underlying architectures. The tensor compiler provides a set of *schedule primitives* to the developers to perform machine-dependent optimizations. Halide [55] and TVM [12] are two representatives of this category. While achieving promising performance on general-purpose platforms, this approach still relies heavily on manual efforts.

Many approaches [2, 13, 42, 50, 72, 73] work on optimizing the performance of manually written schedules. However, none of these compilers target an NPU. The schedule primitives available in these compilers constitute a limited set of transformations that are difficult to scale with the increasing scenarios due to the rapid advance in DL algorithms. Expanding the schedule primitives is a feasible solution, but it may result in the combinatorial explosion issue.

Leveraging polyhedral IR. This approach converts a tensor program into polyhedral IR and performs schedule transformations. The typical examples of this category include Tiramisu [4] and MLIR [40], which allow manual intervention but apply transformations on top of a polyhedral IR. While performing transformations systematically, polyhedral IR also simplifies the storage management.

TC [62] is another representative of this category that goes further by using the *polyhedral scheduling heuristics* [9, 17] to compute schedules in the absence of human efforts. Although the affine schedulers of the polyhedral model release the burden of developers by fully automating the scheduling process, the performance of TC is considered as inferior to those of the first category due to the ineffective modeling of target architecture. Besides, we did not find a polyhedral compiler specialized for NPU architectures.

Challenges from NPU architectures. Developing optimizing compilers for an NPU architecture is still an open issue, which faces (at least) the following challenges.

First, scheduling for NPUs is difficult. An NPU is usually designed as an accelerator that requires sophisticated fusion strategies in addition to complex tiling techniques. The computations should be grouped aggressively when offloaded onto the chip to maximize locality, while an architecture-specific fusion should be applied when data flow to different compute units. Besides, further rescheduling steps within the local buffers may also be required to benefit for vectorization.

Second, managing the dataflow within an NPU requires the explicit decoupled data orchestration [52] due to the multi-level, multi-directional memory hierarchy present in most domain-specific architectures, *e.g.*, Google TPU [33] and Huawei Ascend chips [43]. Existing approaches designed for traditional memory hierarchy pyramid are not suitable for these architectures.

Finally, an effective implementation of a convolution is calling for the *img2col* transformation [21, 67] that performs a convolution using a general matrix multiplication (GEMM) product. Some domain-specific hardware like the *fractal* architecture [71] wishes for the further decomposition of a GEMM product into *fractal* blocks. These domain-specific transformations are beyond the power of existing approaches.

Domain-specific architectures. We choose Huawei Ascend 910 chips, with its DaVinci architecture shown in Fig. 1, to address the above challenges. Implementing our approach for this target can provide a general solution that the tensor compilers for other NPUs can follow, since the DaVinci architecture adopts a similar memory hierarchy to TPU [33] and requires a *fractal* transformation introduced by the Cambricon architecture [71] when handling convolutions.

The DaVinci architecture uses a *Cube Unit* for processing matrix operations, a *Vector Unit* for executing vector computations and a *Scalar Unit* to handle scalar tasks. The arrows represent the possible dataflow paths between the multi-level hierarchy. *L0A* and *L0B* serve as the input buffers of the Cube Unit, whose output is stored in *L0C*. External data should be offloaded to *L1 Buffer* or *Unified Buffer (UB)* which constitute the second level buffers of the memory hierarchy. *img2col* is handled within the memory transfer engine (MTE), of which the result will be passed to the *fractal* transformation.

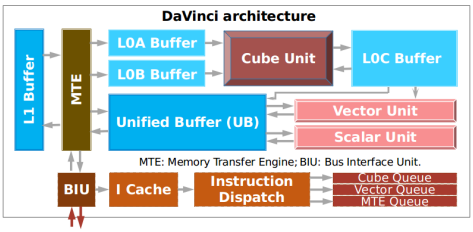

Figure 1. The DaVinci architecture of Ascend 910.

Our solution. None of existing approaches can fully meet the requirements of our target. We consider to complement TVM with a backend to support code generation for the DaVinci architecture. However, its limited schedule primitives are not sufficient to model the complete set of transformations nor the effective storage management. Fortunately, the decoupled modules of TVM make it easy to lower its DSL to HalideIR and polyhedral IR. We thus introduce another pass that converts the HalideIR generated from TVM's DSL to polyhedral IR where most transformations are performed. Integrating the polyhedral model into TVM not only allows the reuse of existing functionalities of the latter, but also balances the weaknesses of both approaches.

As revealed by MLIR [40] and TC [62], lowering to the polyhedral IR [20] can automate storage management. The polyhedral schedulers [9, 17] are integrated with different fusion heuristics; one can choose a suitable fusion strategy for each compute unit by switching between the heuristics.

However, the generic, black-box use of the ILP solver [63] makes the performance inferior to that of TVM [12, 13].

In spite of that, a recent study [7] demonstrates that a careful manipulation of polyhedral IR can obtain near-peak performance for GEMM. Inspired by this work, we isolate the fusion heuristics from the polyhedral schedulers and apply fusion in conjunction with tiling as post-scheduling transformations. This provides more choices for the combination of tiling and fusion by enabling complex tile shapes.

The last difficulty is the modeling of domain-specific transformations. To the best of our knowledge, there exists no previous work studying this issue in the context of polyhedral compilation. We introduce an external pattern-specific polyhedral IR of a *fractal* GEMM routine by formulating the relations between the original convolution and the final GEMM implementation, and use it to substitute its correspondence of the convolution in the original polyhedral IR. This results in the practical implementation of the domain-specific transformation in tensor compilers.

3 Overview of AKG

Fig. 2 depicts the architecture of AKG. AKG takes as input a fused operator that has been specified using the DSL of TVM, which rewrites the computation graph representation generated by the graph engine. The graph engine of TVM inherited by AKG provides the ability to transparently integrate with different DL frameworks; AKG targets the optimizations of individual layers/subgraphs of a DL model.

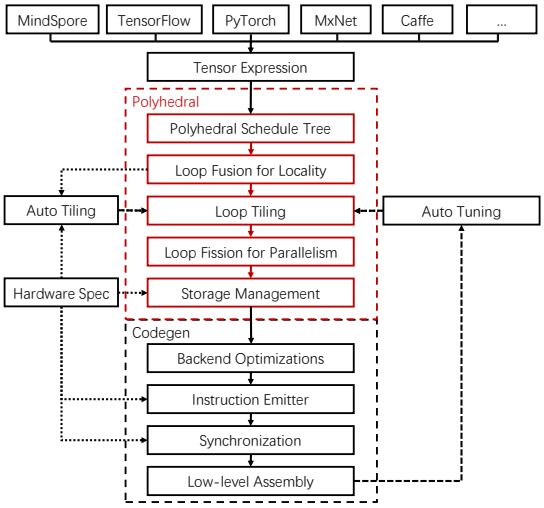

Figure 2. Architecture of AKG.

The DSL is then lowered to a parametric HalideIR expression, which is further converted into a polyhedral IR [20]. As the polyhedral model calls for the fulfillment of "static affine control" [18] of a program, we implement a class of automatic preparation steps including function inlining, common subexpression elimination, *etc.* before lowering to the polyhedral IR, which also moderates the compilation overhead.

Most loop transformations are applied on top of the polyhedral IR, with the scheduling heuristics of *isl* [63] to find a tiling-friendly composition of loop transformations, and a combination of tiling and fusion to model parallelism and spatial/temporal locality with respect to the hardware model. The tiling strategy is computed based on optimization patterns derived from the hardware model, and can be fine-tuned using a learning-based algorithm. Storage management is also fully automated in polyhedral compilation, with minimizing data movements taken into consideration.

Finally, the optimized polyhedral IR is delivered to the code generator, producing imperative code executable on the target architecture. Syntactic transformations that are difficult to model using polyhedral compilation are performed in the code generator. These post-polyhedral transformations further widen the optimization space of AKG.

4 Polyhedral Transformations

Performing computation on an NPU usually has to access off-chip data. The huge computing power brought by an NPU is usually hindered by the redundant data exchanges between the off-chip memory and on-chip buffers. A compiler should apply loop fusion to create more on-chip intermediate values such that the amount of data movement can be minimized. When data has been offloaded, one has to manage the memory hierarchy of an NPU, making loop tiling a natural candidate for exploiting this feature of the architecture.

However, a program is not always amenable to fusion or tiling due to data dependences. We thus first resort to a polyhedral scheduler to transform the original program into a form that can benefit for fusion and tiling. Given the composition of tiling and fusion has been modeled, one can transfer the data required by an NPU with the minimal amount of data. This works well for traditional memory hierarchy pyramid, but optimizing for the complicated memory hierarchy of NPUs requires a further exploration of on-chip loop fusion/distribution for the heterogeneous compute units. We introduce a post-tiling fusion strategy to address this issue.

The aforementioned analysis finally results in the design rationale of our compiler transformations, all of which are performed on top of the polyhedral IR–schedule trees [20]. We convert the DSL of TVM into the schedule tree representation, which is built using a rich set of node types that will be introduced along with the following steps.

4.1 Versatile Polyhedral Scheduling

Programs amenable to the polyhedral model are represented using integer sets and affine maps. Each program statement is instanced using its index variables, and the complete set of these runtime statement instances constitutes the *iteration domain*, expressed using integer sets, of a program. The iteration domain is represented using the identical *domain* node in the schedule tree of a program. The textual execution order defines the original *schedule* of the iteration domain, which is transformed automatically by polyhedral schedulers into a different one represented by affine maps.

Figure 3. A running example and its schedule tree representations.

Polyhedral schedulers expose a much wider set of affine transformations than TVM [12] while automatically guaranteeing the validity of the transformations by preserving each dependence. In particular, auxiliary loop transformations including skewing, shifting and scaling that are not considered by TVM are fully modeled. One is free to set different scheduling options that enable/disable certain types of loop transformations for tuning the scheduling process, which is much easier than writing schedule templates. We resort to the *isl* scheduler [65] that uses the Pluto algorithm [9] as a primary scheduling strategy and the Feautrier algorithm [17] as fallback for computing a new schedule, maximizing parallelism (executing independent statement instances in parallel) and temporal locality (executing dependent statement instances close in time) simultaneously.

The *isl* scheduler computes an affine function and decodes it using a *band* node in schedules trees by solving ILP problems. Considering the complexity of ILP problems, the *isl* scheduler introduces an *affine clustering* heuristic [62, 65] to decrease the size of ILP problems and implements loop fusion by iteratively grouping band nodes. Each fusion group is represented by a *filter* node in schedule trees, which is connected by a *sequence/set* node with its children. When the band nodes cannot be fused, a *sequence/set* node is introduced to express the particular/arbitrary order.

We show a typical fused pattern obtained from the graph engine in Fig. 3(a), with the pseudo code shown in Fig. 5(a). The statement IDs are also marked in Fig. 5(a) instead of Fig. 3(a) because the DSL represents the initialization and reduction statements as a compound operator. It performs a 2D convolution, followed by two vector operators, on the input feature maps `A` using kernels `B`, with the result written to the output feature maps `C`. A constant (bias) addition step is also used before the convolution. The initial schedule tree depicted in Fig. 3(b) is built using the textual order of the fused subgraph, with the domain node shown at the bottom.

Loop tiling can now be applied to each grouped band node produced by the affine clustering heuristic of *isl*. This isolated implementation manner, which is also adopted by other polyhedral compilers [9, 62, 64], results in two kernels that have to executed on an NPU, failing to meet the requirement of generating a single kernel for each fused sub-graph as expected by DL frameworks. In addition, such an implementation of loop tiling and fusion is also inferior at minimizing the data movement between the hierarchical memories of an NPU. This is because traditional polyhedral approaches only transform the iteration spaces, but the conflict is due to the lose of alignment between the tiled data spaces after loop fusion. The conflicting demands of tiling and fusion can be alleviated provided such alignments can be recovered.

To enforce the generation of a single kernel, we use the *reverse strategy* proposed in our earlier work [70] which completely eliminates the mismatches between tiled data spaces using elementary combinations of the operations on affine sets and maps. This reverse strategy allows for the construction of arbitrary tile shapes and enables post-tiling

fusion. However, we still have to perform additional on-chip loop optimizations and domain-specific transformations of convolution, leading to the following transformation order.

4.2 Tiling

Loop tiling [27] is essential to benefit from the locality and parallelism provided by faster local memories. The implementation of loop tiling boils down to two issues: one is to construct tile shapes and the other is to select tile sizes. We address the first issue with our previous reverse strategy and the second using a tile-size specification language which was not considered in the past [9, 64].

Constructing tile shapes. Following our prior work [70], we first adopt a conservative clustering strategy of *isl* to convert the initial schedule tree into the form shown in Fig. 3(c) by maximizing the tiling opportunities, with the corresponding pseudo code shown in Fig. 5(b). The two fusion groups produced by this clustering strategy, which are represented using the two filter nodes under the top sequence node, are considered as an *intermediate* and a *live-out* iteration space, respectively, as shown in Fig. 3(d). A live-out iteration space is composed of the statements writing to memory locations that will be referenced after the computation of the program, while the definitions of an intermediate iteration space to memory locations will be consumed within the computation.

Unlike the traditional manner used to perform tiling and fusion independently, the reverse strategy [70] only tiles the live-out iteration space using a quasi-affine function $\{S_2(h, w, kh, kw) \rightarrow (h/32, w/32, h, w, kh, kw)\}$, with 32×32 representing the tile sizes along h and w dimensions. This converts the 4D iteration space of S_2 into a 6D space, with $(h/32, w/32)$ expressing the tile loops (iterating between tiles) and (h, w, kh, kw) the point loops (iterating within tiles), producing a rectangular tile shape on the live-out space.

The memory footprint of each live-out iteration tile can be determined by applying the read access relations to the integer set representing this tile. In particular, we are interested in the data tiles of the input feature maps A that result in the producer-consumer relation between the two iteration spaces. As the convolution is performed over the input feature maps A, its data tile shape can be in arbitrary forms (overlapped, continuous or scattered) depending on the strides of the convolution kernels. The resulted arbitrary data tile shape can be used by the reverse strategy [70] to compute the relation between this tiled live-out iteration space (consumer) and the intermediate iteration tile (producer). As a result, the tile shape of the intermediate iteration space can be arbitrary. For the example shown in Fig. 3, the reverse strategy produces an affine function $\{(o_0, o_1) \rightarrow S_0(h, w) : 0 \leq o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \leq o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \leq h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \leq w < T_3 \cdot o_1 + KW + T_3 - 1\}$ which relates the tile loop dimensions $(o_0 = h/32, o_1 = w/32)$ of S_2 with those subregions (tiles) of S_0, allowing for the overlaps between the iteration tiles of S_0. This tile shape can also benefit for the aggressive storage optimizations [51, 55] and domain-specific transformations [21, 67] (Sec. 4.5).

Specifying tile sizes. Although parametric tiling has also been widely investigated [23, 35], many polyhedral optimizers [4, 62] still either use default values embedded in compilers or only allow users to specify tile sizes before applying transformations. TVM lets engineers specify tile sizes along with writing schedule templates and overcome this weakness. The complicated memory hierarchy of modern NPUs requires a more sophisticated strategy to select tile sizes.

We propose a tile-size specification language as shown in Fig. 4. A polyhedral statement can be specified using a user-defined tiling policy, which can be defined using the specifications on either single or multi-level buffers. A specification of tile sizes is composed of the tile sizes along each loop dimensions and the string indicating the location the data accessed by this statement should be placed.

```
stmt_id :: "S_" integer
tile_size :: integer
tile_spec :: tile_size @ buffer
tile_specs :: tile_spec | tile_specs, tile_spec
stmt_spec :: stmt_id : tile_specs
tiling_policy :: stmt_spec | tiling_policy stmt_spec
```

Figure 4. Tiling policy specification language.

While compatible with the strategy of constructing complex tile shapes, this specification language is more flexible than existing approaches in many aspects. First, the users do not need to worry about the tile shapes or check the validity of tiling, which is guaranteed by the polyhedral model. Second, this language avoids the scenario in which a compiler reorders an input program into a completely different form and the resulted loop dimensions do not match the user-specified tile sizes. This happens frequently when a compiler optimizes a neural network using the combination of tiling and fusion. Finally, like the manual scheduling approaches [72, 73], the language depicted in Fig. 4 also widens the optimization space of the auto-tuner.

Automating tile-size specification. We also introduce a mechanism, Auto Tiling, to automate the tile-size specification using the language shown in Fig. 4. Auto Tiling can determine the tile sizes by inspecting the structure of a schedule tree; it can also work better with the help of the hardware specification as will be introduced in Sec. 4.6.

Auto Tiling is designed to always find the tile sizes that can minimize the data movement, which is the sum of a fixed warm-up cost plus the amount of data movement along tile boundaries divided by the amount of computation. As for a non-contiguous data transfer requirement, the amount of data movement is defined as a weighted sum of the contiguous transfer count and the complete set of data movement. The amount of computation is simply specified as the total number of instances of each statement within the tile.

We express the buffer utilization using a multivariate polynomial of symbolic tile sizes according to the dependences within a tile. Next, we let the buffer utilization be smaller

or equal to the half size of the buffer capacity, enabling the double buffering optimization that can benefit the memory latency hiding (Sec. 5.2). A greedy searching algorithm is then used to find the best tile sizes. As a result, Auto Tiling can generate a much better tiling strategy for most simple computation patterns like tensor addition in DL models.

4.3 Fusion

Loop fusion [34, 48] is a transformation to minimize producer-consumer distances and therefore optimize the locality. It has been integrated into polyhedral compilation [8, 49] for exploiting more combinations with other loop transformations. Existing fusion heuristics did not consider the multi-directional memory hierarchy of NPUs. We refine the fusion strategies in accordance with the memory hierarchy.

Fusion when offloading data. The relation produced by our reverse strategy can be used to instance an *extension* node to implement fusion after tiling, which was impossible in classical polyhedral frameworks [4, 40, 62], as shown at the bottom of Fig. 3. The syntax of an extension node in schedule trees is to introduce foreign statement instances that are originally not scheduled by a (sub)tree.

This extension node is then introduced underneath a band node that represents the point loops of the pre-constructed tiles. The original subtree of a producer iteration space is marked as useless using a *mark* node, which will be skipped by the code generator for avoiding incorrect code replication. A mark node can attach any information expressed using strings. Note that facilitating post-tiling fusion exposes more combinations of tiling and fusion than prior work that either rely on standard schedule trees [20, 62, 69] or the simplified version [7]. We have no awareness of other polyhedral representations able to perform such post-tiling fusion.

As an illustrative example, the post-tiling strategy used by our compiler can fuse all statements into a single group, as shown in Fig. 3(e), while enabling overlapped tiling on the iteration space of S_0. The tensor A that is defined by S_0 and read by S_2 (Fig. 5(a)) can thus be allocated within the chip, mitigating the cost of transferring data. More importantly, modeling post-tiling fusion by manipulating schedule trees can find more aggressive fusion strategies without losing the parallelism/tilability than existing approaches [8, 28, 49] while ensuring the absence of redundant computation [70]. The fusion algorithm is performed greedily when there exist multiple intermediate iteration spaces.

Fusion when forking data. Once data is offloaded, one has to manage the multi-directional dataflow, which cannot be handled using the reverse strategy [70]. This requires a different fusion strategy from that is used when offloading data onto the chip. The tiled data accessed by different compute units should bifurcate, with some streaming to L1 and others to UB. An operator will be delivered to the Cube Unit when it is a convolution or a matrix product; others are handled by Vector or Scalar Unit. The hypothesis we made in this paper is that an operator involving dot-product reductions is viewed as a convolution. The bifurcation of data requires an architecture-specific intra-tile fusion strategy.

We mark the computations that do not involve dot-product reductions using a "local_UB" string, indicating that these statements should stream to UB. A subtree marked by a "local_UB" string is then isolated from its original position in the schedule tree, thereby automatically managing the branching of data within the chip. This isolation is always valid, since it is the reverse process of the fusion before tiling. The validity is guaranteed by the conservative clustering heuristic. Fig. 3(f) depicts the result of this operation.

Intra-tile rescheduling can be now introduced. First, loop distribution is enabled by default within the "local_UB" subtree, which separates the statements to be processed by Vector/Scalar Unit for the purpose of vectorization; second, an aggressive fusion strategy is used by the Cube Unit, grouping the initialization and reduction statements to cooperate with the optimization of convolution as will be discussed in Sec. 4.5. Note that loop distribution is the reverse process of fusion; one can implement it using ILP solvers like *isl* [63] by setting off the clustering strategies. The loop distribution transformation separates S_3 and S_4 in Fig. 5(b) into two filters, each of which can be vectorized, as shown in Fig. 3(f).

Optimizing intra-tile spatial locality has also been studied in the past for maximizing vectorization opportunities on CPUs by refining the scheduling heuristics of the polyhedral model [37]. Our work is inspired by this work but does not reformulate the ILP problem for a "local_UB" subtree. Instead, we sink the fast varying dimension to the innermost position of a *permutable* band node, each dimension of which can always be interchanged safely. The permutability of a band node is determined automatically by polyhedral schedulers, guaranteeing the correctness of the sinking operation without reformulating ILP problems [62]. One can thus expect a similar effect to that of the work [37] using our approach while alleviating complication overhead.

As an example, Fig. 5(c) shows the pseudo code corresponding to the schedule tree of Fig. 3(f). The post-tiling fusion groups all statements together, with the assistance of overlapped tiling of the statement S_0.

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias      // S0
for h in [0,H-KH), w in [0,W-KW):
  C[h,w] = 0                  // S1
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw] // S2
for h in [0,H-KH), w in [0,W-KW):
  C[h,w] = abs(C[h,w])        // S3
for h in [0,H-KH), w in [0,W-KW):
  C[h,w] = ReLU(C[h,w])       // S4
```

(a) Initial program

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias
for h in [0,H-KH), w in [0,W-KW):
  C[h,w] = 0
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw]
  C[h,w] = abs(C[h,w])
  C[h,w] = ReLU(C[h,w])
```

(b) Fused before tiling

```
for x0 in [0,(H-KH)/32), x1 in [0,(W-KW)/32):
  for x2 in [0,KH+31), x3 in [0,KW+31): //   S0, overlapped (exec. on UB)
    A[32x0+x2,32x1+x3] = A[32x0+x2,32x1+x3] + bias
  for x2 in [0,31), x3 in [0,31):       //   S1, (exec. on L1)
    C[32x0+x2,32x1+x3] = 0
    for kh in [0,KH), kw in [0,KW):     //   S2, (exec. on L1)
      C[32x0+x2,32x1+x3] += A[32x0+x2+kh,32x1+x3+kw]*B[kh,kw]
  for x2 in [0,31), x3 in [0,31):       //   S3, (exec. on UB)
    C[32x0+x2,32x1+x3] = abs(C[32x0+x2,32x1+x3])
  for x2 in [0,31), x3 in [0,31):       //   S4, (exec. on UB)
    C[32x0+x2,32x1+x3] = ReLU(C[32x0+x2,32x1+x3])
```

(c) Pre-tile fused, tiled, post-tile fused, and rescheduled

Figure 5. The pseudo codes of the running example.

4.4 Storage Management

The schedule tree representation also provides a convenient interface for automatic storage management. Like PPCG [64] and TC [62], we implement a software-controlled memory management by promoting data tiles to the multi-level buffers of our target. Given an iteration tile of a fusion group, a constant-size strided block can be inferred using the access relations or one can compute an over-approximated rectangular box [63] for those shapes that access non-rectangular blocks of data. The intermediate values accessed by an iteration tile can be promoted to the local buffers of our target and discarded after the computation of the tile.

The data tiles accessed by a subtree marked by "local_UB" are transferred to UB, while the data required to execute a convolution reside in L1. We transform each convolution using the optimization that will be introduced in Sec. 4.5 into a GEMM product, which will rewrite each convolution using the form of `Z+=X×Y`. The input matrix `X` and `Y` are always promoted to L0A and L0B respectively, while the output matrix `Z` is placed in L0C.

Note that hierarchical tiling may be required by those computations flowing to the Cube Unit, which can be implemented by rewriting the band nodes of the point loops obtained by the first level tiling using quasi-affine functions. The memory footprint required by the Cube Unit is always accurate thanks to our tiling strategy, regardless of overlapped, continuous or scattered access manners caused by the strides of convolution kernels. Also note that automatic insertion of synchronizations can be performed using the functionality of an extension node, which was also implemented by PPCG and TC.

4.5 Optimization of Convolution

Performing convolution in an efficient way is essential to achieve high performance for DL models. We resort to the *img2col* transformation that converts a convolution into GEMM product which is in general faster than a convolution. *img2col* has been integrated into Caffe [30] but no previous tensor compilers implement it as a first-class citizen due to the failure of modeling arbitrary tile shapes.

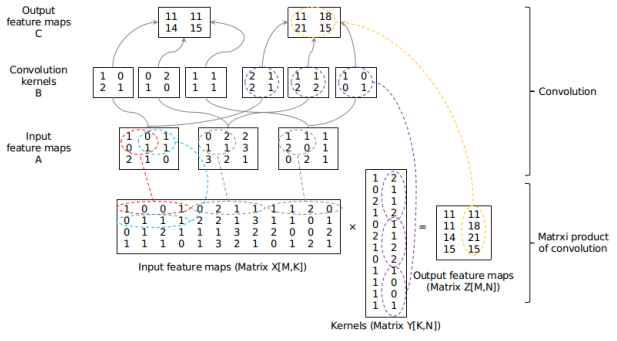

Figure 6. The *img2col* transformation.

Implemented by MTE in Fig. 1, the *img2col* transformation can be explained using Fig. 6. Each local patch of the input and the convolution kernel is expanded to a vector. This conversion requires the overlapped accesses between consecutive patches (the blue and red dashed ellipses). The output is converted to the matrix Z. The GEMM product is depicted at the bottom of Fig. 6, with its relation with the original convolution shown using dashed lines in the figure. Note that MTE is only responsible for transforming data layout. As a convolutional computation pattern should be converted into a matrix product form, the polyhedral model still has to perform transformations on iteration spaces.

img2col makes it possible to wrap highly optimized vendor libraries, but we take another way by decomposing the resulted matrix product into *fractal* GEMM kernels, since the target hardware uses a fractal architecture [71], of which the last-level block calls for a small GEMM kernel with aligned accesses for guaranteeing high performance.

We build an external polyhedral IR of a fractal matrix product, with the band nodes tiled and the innermost dimension aligned according to the fractal pattern shown in Fig. 7. The red broken lines within each matrix represent the execution order of tiles, while the green broken lines are the execution order of point loops within a tile. Each tile of a matrix is aligned (and padded if necessary) to fit in the last-level block of the fractal architecture.

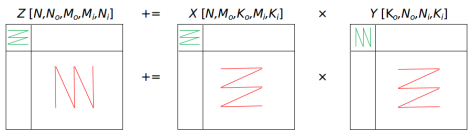

Figure 7. The fractal GEMM product.

Next, we introduce a custom pass that grafts the polyhedral IR of the fractal matrix product to replace the correspondent subtree, as the pink region shown in Fig. 3(f), of a convolution. The relation between the subtree and the external polyhedral IR is computed using the following affine functions, which are inferred from the *img2col* and *fractal* transformations. f is the size of the last-level block of the fractal architecture that will be instanced using an integer number in practice; s_h and s_w represent the convolution strides along h and w dimensions, of which the number of padding dimensions are expressed using pad_h and pad_w respectively. % is used to denote the modulo operation.

$$\begin{cases} i_0 = i_0';\ i_1 = \lfloor (i_2'/(KH \cdot KW)) \rfloor; i_4 = i_4' \\ i_2 = \lfloor (\frac{i_1' \cdot f + i_3'}{w_o}) \rfloor + s_h + \lfloor (\frac{i_2'}{KW}) \rfloor \% KH - pad_h \\ i_3 = ((i_1' \cdot f + i_3') \% w_o) \cdot s_w + i_2' \% KW - pad_w \end{cases} \quad (1)$$

The input feature maps of an *NCHW* convolution are required to be written in the 5D form of $A[N, C_1, H_i, W_i, C_0]$, which tiles the input channel dimension C_i and sinks the point loop dimension C_0 to the innermost position. The output feature maps C can be written as $C[N, C_o, H_o, W_o]$. The

subscript i/o is used to distinguish the input/output feature maps, *e.g.*, H_i represents the height of an input feature map.

One can conclude that $M{=}H_o \cdot W_o$, $N{=}C_o$ and $K{=}C_1 \cdot KH \cdot KW \cdot C_0$ hold, where M, N, K represent the extents of the target GEMM product in Fig. 6. i_k $(0 \le k \le 4)$ in (1) represent the index variables $(N, C_1, H_i, W_i, C_0$ from outermost to inner) of the input feature maps A; i'_k $(0 \le k \le 4)$ are the index variables $(N, M_o, K_o, M_i, K_i$ from outermost to inner) of the matrix X in Fig. 7. The indexes of matrices Y and Z can also be determined in a similar way. Note that embedding an external schedule tree does not only improve the performance of convolutions but also conceals the complicated architecture of NPUs from users.

One can remove the reduction dependences of the fractal GEMM product to implement parallel reductions. The subtree of a reduction after the removal of self-dependences is annotated using a mark node, which will be processed by the code generator using special hardware intrinsics.

4.6 Manual Scheduling and Debugging

We design a memory hierarchy specification language to provide the functionality of manual management of dataflow. Managing the complicated dataflow can be realized by specifying a (sequence of) "npu" statement(s), (each of) which is instanced using either a specific compute unit, a buffer specification indicating the destination buffers, or a detailed description of dataflow. A statement residing in a particular compute unit is specified by the computation type, followed by the input buffers, output buffers and two integer values representing the throughput and alignment of the compute unit respectively. A dataflow specification starts with a special string, followed by the same set of parameters as the compute unit statement. One can also introduce a memory allocation statement using the buffer specification statement, which is composed of a string representing a particular buffer location and its memory size.

```
buffer :: string
buffer_size :: integer
buffer_spec :: "buf" buffer ( buffer_size )
compute_type :: string in a predefined set
in_bufs :: buffer | in_bufs buffer
out_bufs :: buffer | out_bufs buffer
throughput :: integer
alignment :: integer
compute_unit :: compute_type ( in_bufs -> out_bufs,
throughput, alignment )
dataflow :: "dataflow" ( in_bufs -> out_bufs,
throughput, alignment )
npu_stmt :: compute_unit | buffer_spec | dataflow
npu_spec :: npu_stmt | npu_stmts npu_stmts
```

Figure 8. Memory hierarchy specification language.

The specification language enables fine-grained management of the memory hierarchy, going beyond the manual scheduling approaches by providing a much richer set of interfaces integrated with the domain-specific architectures. The design of the specification language is used to simplify debugging and manual management of dataflow on demand: the users can write such specifications or modify those generated by AKG to enforce the code generator to work as they expect. However, the versatility and effectiveness of our compiler still depend on the polyhedral IR. We never use this manual specification in the experiment.

5 Code Generation

Generating code from the polyhedral IR amounts to first generating Abstract Syntax Tree (AST) and then converting the AST to specific IR or imperative program. The schedule tree representation lowered from HalideIR is first converted back using the code generation algorithm of *isl* [63], which is then translated to an imperative program of our target.

An imperative program of our target is referred to as CCE code which is a C-like programming model designed by fully considering the architectural features of Ascend 910 [43] including the SIMD intrinsics. Scheduling for automatic vectorization is not considered by *isl*, which was addressed in our work by manipulating the polyhedral IR. Another weakness of the classical AST generator of *isl* is that it does not support automatic generation of vectorization code. We complement the code generator with a vectorization step.

5.1 Vectorization

Exploiting effective SIMD vectorization is vital due to the frequent presence of compute-bound operators in neural networks. Our fusion strategy introduced in Sec. 4.3 breaks down a subgraph into small pieces that can be translated into SIMD intrinsics of the target machine. The domain-specific transformations discussed in Sec. 4.5 maximize the alignment of vectorization; the code generator can thus improve the performance by making full use of the hardware intrinsics.

More specifically, the code generator takes as input the optimized HalideIR obtained from the polyhedral optimizer that captures the necessary amount of information, including alignments, strides, source and destination of a SIMD intrinsic, required to generate efficient SIMD code, with automatic alignment of unaligned loads taken into account. Data layout transformations may also be involved here to change the data organization into the expected form. Isolating full tiles from partial tiles [35] is also enabled by default to maximize the vectorization. Besides, the transformations that only change the loop structure but not the computations, *e.g.*, loop unrolling, can also be performed here as complementary optimizations. In summary, we maximize the utilization of SIMD hardware intrinsics.

5.2 Low-level Optimization of Synchronization

Much of the versatility and effectiveness of the high-level memory management resides in the manipulation of schedule trees. However, modern NPUs usually resort to the decoupled access-execute (DAE) architectures [60] where each

compute unit and data movement unit has an independent instruction pipeline, and synchronization is required between the pipelines to preserve data dependences. This optimization is out of the scope of polyhedral compilation.

Memory latency hiding, one of the optimizations considered by TVM, is implemented as a post-polyhedral transformation in AKG. However, TVM did not take into consideration the heterogeneous compute units, leading to the ineffective optimization of low-level synchronization. We first insert low-level synchronizations into the generated code by considering the data dependences across different compute units; a follow-up optimization is then used to group the synchronizations using a dynamic programming strategy, thereby reducing the number of synchronizations.

5.3 Auto Tuning Strategies

The polyhedral transformations optimize the schedule of a program by abstracting away the underlying architectures, making the performance far from the optimum in practice. We address this problem using an auto-tuner to measure the performance of divergent tiling strategies. The tuning space for tensor computations is usually huge, we thus use a machine learning guided sampling approach to reduce the tuning space, with the steps summarized as follow.

AKG first computes a tuning space composed of the valid tiling parameters constructed by the strategy introduced in Sec. 4.2. A first round of random samples is then extracted from the space, followed by the measurement of the performance on each sample. These random samples are used to train a machine learning model, which will generate a second round of random samples. A second-round sample is derived from one of the N best-performing samples of the first round by moving a random step towards higher performance in the learning model with probability p, or it is sampled randomly over the entire tiling space of the first round with probability $1 - p$. The auto-tuner measures the performance of each of the second round samples to update the learning model. This will be repeated until the number of sampling iterations reaches a pre-defined threshold or no performance gain can be obtained.

In practice, N is specified (as 64 in the experiment) by the developers, and the forwarding direction is determined by the derivatives of each tuning iteration. p is a varying value during the sampling iterations, which is computed using a formula with a pre-defined parameter (set as 0.5) and ranges from 0 to the exponential constant e. Our tuning strategy is not meant to guarantee the optimal performance, but it can usually find a better tiling strategy than the Auto Tiling in Sec. 4.2 that minimizes data movement.

6 Evaluation

We implement our approach as an optimizer within a full-stack, all-scenario AI computing framework, MindSpore [26], to generate executable code on the Huawei Ascend 910 chip, which can be used for both training and inference tasks of a deep neural network. The algorithmic implementation can be retrieved from https://gitee.com/mindspore/akg.

We compare the performance against manually optimized low-level code (CCE opt) written by experts of the Ascend chip. The software R&D team of the chip also adapted the schedule primitives of TVM to the DaVinci architecture, which we also take into account for comparison. The manual schedule templates are fully optimized by TVM's auto-tuner.

None of the enhancements of TVM [13, 72, 73] have been integrated into MindSpore's ecosystem; existing polyhedral DL compilers like TC [62] did not perform optimizations for an NPU. We thus do not compare with these techniques. We consider single operators, fused subgraphs and end-to-end networks as the benchmarks, with each version of the code compiled with the native compiler of the chip using the same set of compilation options for the fair comparison. The best tile sizes of AKG are selected by Auto Tiling (Sec. 4.2), while those of TVM are chosen by experienced experts and fully tuned by its auto-tuner.

6.1 Performance of Single Operators

The single operators used in the experiment are those commonly used in real-life deep neural networks, including convolution (op1), matrix multiplication (op2), ReLU (op3), batched matrix multiplication (op4), cast (op5), transpose (op6), one-hot (op7), tensor addition (op8), BatchNormed training reduction (op9) and BatchNormed training update (op10). We provide 10 shape configurations for each operator using a batch size of 16, leading to 100 test cases in total.

We show the relative performance speedup normalized to the execution cycles of our approach for each single operator in Fig. 9. To compute the normalized performance, we record the execution cycles of each shape configuration and compute their geometric mean. We also include the naïve implementation of the CCE code on the Ascend chips as a baseline, which is written by the experts without using vendor libraries or performing optimizations.

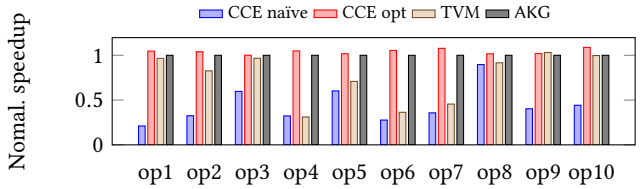

Figure 9. Performance of single operators (higher is better).

The optimized CCE code performs better than the naïve implementation, improving the performance of the latter by 2.8× on average. Manual optimizations allow a developer to wrap highly optimized libraries that can cooperate better with the native compiler of the chip. An expert can use hardware prefetching to overcome the ineffectiveness of double

buffering exploited by AKG when handling scalars. However, its high performance comes with the huge human efforts; it also fails to scale with different shape configurations. The code generated by our approach can achieve the performance comparable to the vendor libraries with a mean loss within 4%. It demonstrates that AKG can reach the performance of highly optimized libraries even for a single operator.

The developers of manual schedules also apply *img2col* and *fractal*. However, writing manual schedule templates suffers from the same problem of vendor libraries when experimenting with multiple shape configurations. TVM can outperform our approach under a small set of specific shape configurations. As a fully automatic workflow, AKG only allows manual padding before applying transformations. On the contrary, one can perform padding optimizations along with the manual scheduling process in TVM. Ineffective padding may result in partial tiles that lose the benefit of SIMD intrinsics. Modeling effective padding optimizations in AKG is our future work. However, the performance of our approach can provide a mean speedup of 1.6× over TVM.

AKG significantly reduces development efforts compared to the optimized CCE code and TVM. Fig. 10 compares the lines of code for three important single operators frequently used in DL models. As shown in the figure, developing high-performance vendor libraries always calls for a much higher engineering cost. Besides, manual optimizations also make it difficult to debug and it is not scalable to tensor shapes. Writing manual schedule templates alleviates this issue but it still requires much effort from the vendor developers. Conversely, AKG obtains performance comparable to or better than that of vendor libraries using much fewer lines of code.

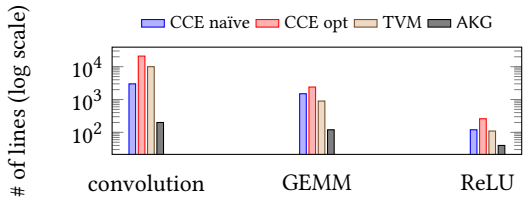

Figure 10. Comparison of lines of code (lower is better).

To demonstrate the scalability of our approach under different shape configurations, we study the performance of matrix multiplication since it is one of the most important routines in many application domains. Moreover, our approach first transforms a convolution into GEMM product, meaning that a convolution is also backed by the implementation of the latter. Evaluating GEMM product can thus also reflect the effectiveness of convolution optimizations.

We use 41 different shape configurations to evaluate the performance of the GEMM product. The shapes range from (64,64) to (4608,4608) for each matrix. The number of execution cycles under each shape configuration is collected, constituting the performance trend shown in Fig. 11. We compare the performance with that of TVM, which also exhibits a good scaling with the increasing number of shapes. Our approach follows a similar fluctuation to that of TVM, but the execution cycles of the code generated by AKG are generally fewer under 29 out of the 41 configurations.

The performance differences between AKG and TVM is due to the DAE synchronizations. The developers of the manual scheduling approach also enhance the memory latency hiding of TVM by considering dependences between heterogeneous compute units. However, their clustering strategy of synchronizations is performed empirically, which cannot always find an optimal grouping of synchronizations like the dynamic programming policy in AKG. The reason why AKG sometimes falls behind TVM is due to the constraints we put on the searching strategy of our auto-tuner; our approach can obtain better performance if the constraints are relaxed.

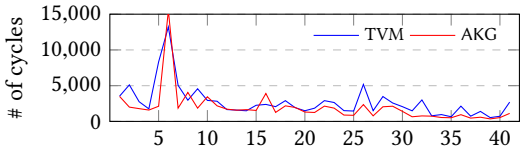

Figure 11. Performance of GEMM product under different shape configurations (1 μs = 10^3 cycles; lower is better).

6.2 Performance of Subgraphs

We next conduct experiment on five fused patterns covering a variety of scenarios that can be found in widely used models including ResNet-50 [24], Bert [15] and MobileNets [25]. These subgraphs include a large class of single operators like those introduced in Sec. 6.1 and some additional vector operators. We report the performance of a single shape configuration, which is listed in the rightmost two columns of Table 1, for each subgraph. We also evaluate these subgraphs using other shape configurations but did not see heavy performance variances. Table 1 also summarizes the number of operators for each subgraph.

Table 1. Summary of the subgraphs.

no.	# of ops	precision	batch size	input shape	output shape
1	6	FP16	16	(16,16,512,512)	(16,16,512,512)
2	21	FP16	16	(256,512,16,16)	(256,512,16,16)
3	15	FP32	16	(30522,1024)	(30522,1024)
4	11	FP32	16	(1024,1024)	(1024,1024)
5	9	FP16	16	(64,1,16,16)	(64,1,16,16)

We compare the performance of AKG with two baselines, one is the manually optimized CCE code and the other is the code generated by TVM. The vendor developers did not provide the naïve implementation for subgraphs, and its performance is thus missing. The result is shown in Fig. 12, with the speedup normalized to the execution cycles of AKG.

Our approach performs best among the three versions for each of the subgraphs; the code generated by AKG outperforms TVM and the manually optimized code by 1.3× and

5.6× on average, respectively. TVM also provides a mean speedup of 4.4× over the hand-written code by exploiting the optimizations across operators, further demonstrating the importance of developing tensor compilers.

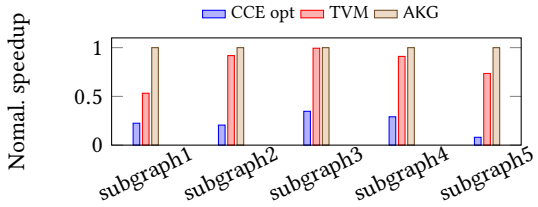

Figure 12. Performance of subgraphs (higher is better).

As explained in Sec. 6.1, writing manual schedules can also model the domain-specific transformations designed for convolutions. An expert can thus achieve similar optimization configurations for some of the benchmarks, resulting in the comparable performance between TVM and AKG for three subgraphs. However, the experience of an expert does not apply to all fused patterns, *e.g.*, subgraph1 and subgraph5, for which AKG provides significant improvement over TVM. These cases require the modeling of a complex tile shape that is difficult to reason about using manual optimization, which can create more opportunities for fusion and reduce the amount of offloaded data. On the contrary, AKG implements it by resorting to the techniques introduced in Sec. 4. The construction of complex tile shapes also involves loop skewing and shifting that are not expressible in TVM, each of which also contributes to the overall performance of subgraphs and end-to-end workloads as will be discussed in Sec. 6.3, validating the importance of using a general polyhedral framework.

6.3 Performance of End-to-End Networks

We finally evaluate five end-to-end workloads, which are composed of (1) three models for image classification including ResNet-50 [24], MobileNet-v2 [25], AlextNet [39], (2) the Bert [15] workload for language understanding, and (3) SSD [45] for object detection in images. We record the execution cycles of a single training epoch with batch size 16, and normalize the performance with respect to the result of our approach. Fig. 13 depicts the result on these benchmarks. Following the single operator and subgraph cases, we still use two baselines for the comparison. We consider two vocabulary sizes, 21,128 for the first version and 30,522 for the second, for the Bert model in the experiment.

Hundreds of engineers were involved in optimizing CCE code on the Ascend chips, which is a huge cost for the industry. Nonetheless, the optimized CCE code can only support one end-to-end workload, *i.e.*, ResNet-50, of those benchmarks used in the experiment. On the other hand, generating code using a tensor compiler provides a more flexible approach while achieving better performance even for the ResNet-50 model: AKG and TVM bring about 7.6% and 7.7% improvement over the optimized CCE code, respectively.

Our approach performs similarly to TVM for ResNet-50, MobileNet and AlexNet, which are mainly composed of 2D convolutions and similar fused patterns (like subgraph2 and subgraph3 in Sec. 6.2) that are optimized extremely by both AKG and TVM. However, AKG outperforms TVM when experimenting with (both versions of) Bert and SSD models.

The Bert workload is composed of multiple subgraphs which consume most of the execution time. It is difficult for an expert to find a good fusion configuration for each subgraph by writing manual schedules. AKG suggests better fusion configurations on more subgraphs using a systematic fusion heuristic. The schedule of SSD written by hand have not yet exploited the SIMD hardware intrinstics due to the large number of divergent vector operators in the model, which also validates that AKG can obtain a better performance within a much shorter period.

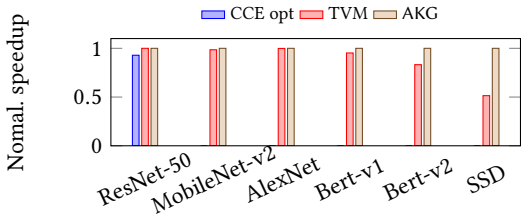

Figure 13. Performance of DL models (higher is better).

Our results demonstrate that using polyhedral compilation allows to approach or even exceed the performance achieved by writing manual schedules. In summary, the overall improvement of AKG over TVM on these workloads is 20.2%. Note that tens to hundreds of vendor developers are still striving to optimize the manual schedule templates of TVM, while the workflow of AKG is fully automated.

7 Related Work

Deep neural networks expressed using frameworks like TensorFlow [1], Pytorch [53] are typically represented as connected subgraphs by compilers like XLA [41], TASO [31], nGraph [14] and Glow [57] for the ease of graph-level optimizations. While high-level optimizations like kernel fusion [12] and layout transformations [46] are considered by these compilers, our work takes an orthogonal way by studying operator-level optimizations, providing an effective solution to code generation for custom NPUs.

Targeting the operator-level optimizations, a scheduling language was introduced in Halide [55] to optimize image processing pipelines and extended by TVM [12] to handle tensor programs. Halide is followed by a variety of enhancements [2, 42, 50] to optimize schedules automatically using learning-based auto-tuners; AutoTVM [13], FlexTensor [73] and Ansor [72] are used to reduce the human efforts required by TVM. On the contrary, AKG provides a large set of transformations thanks to the polyhedral schedulers. In addition,

our work also implements a fully automated management of memory promotion and synchronization for an NPU, which is non-trivial for manual scheduling approaches.

The polyhedral model was also used by DL compilers including TC [62], Tiramisu [4], Stripe [68] and Diesel [16]; they can perform loop transformations that are not accessible in XLA [41] and Latte [61]. The followers [6, 22] of the Diesel compiler support code generation for Tensor Cores of GPU, while the recent work Rammer [47] considers the code generation for Graphcore IPU [32]. AKG further widens the set of transformations by constructing arbitrary tile shapes, hierarchical fusion strategies and domain-specific transformations, which are essential to exploit the memory hierarchy of an NPU. There also exist polyhedral extensions [28, 29, 51, 69] to Halide that only target CPU and/or GPU.

Almost all existing DL compilers rely on auto-tuning techniques like OpenTuner [3] and Stock [58] to search the optimal implementation variants of tensor programs. High-performance libraries like BTO [5], FFTW [19] and ATLAS [66] also leverage auto-tuners. We use an auto-tuning strategy in our work as a complementary optimization to the polyhedral transformations. Our strategy to implement tiling and fusion moderates the tuning overhead.

8 Conclusion and Discussion

Developing tensor compilers is an effective solution to address the various weaknesses of existing DL frameworks. Unlike traditional approaches that either rely on manual schedules or leverage polyhedral compilation without extensions, we integrated the polyhedral model into the TVM compiler to benefit from both sides. The AKG compiler presented in this paper implements a versatile polyhedral scheduler, followed by the combination of sophisticated tiling and hierarchical fusion, and a sequence of domain-specific transformations for optimizing convolutions. We also introduced a learning-based auto-tuner, complementing the polyhedral transformations using a fine-tuned searching strategy. The experimental results show that the generated code of AKG, which has been further optimized by the vectorization and low-level synchronization, can achieve performance comparable or superior to those of manually optimized approaches while minimizing the programming effort of the users.

Our work uncovers the important components that have to be considered when developing tensor compilers. As the red parts shown in Fig. 2, effective scheduling is at the core of a tensor compiler. In particular, one should carefully handle the interplay between tiling and fusion which contributes most to improving performance, as shown in Sec. 6. We partially addressed this issue using the reverse strategy [70] but more efforts are still in need. The reverse strategy and the intra-tile rescheduling transformation are platform-neutral. The fusion algorithm introduced in Sec. 4.3 when forking data can also be adapted to other similer NPU architectures with slight human efforts, though it is designed for the Ascend 910 in this work.

Another essential factor is the automatic implementation of domain-specific transformations [21, 67, 71] for convolutions, which have been addressed by AKG using schedule trees. While the fractal tiling is specific for the Ascend 910 chips, the *img2col* transformation performed by grafting an external schedule tree can be used as a general method. In addition, designing a productive IR is also important: it should not only be used for representation, but also for delivering domain-specific knowledge and implementing program transformations easily. Our extension to schedule trees and MLIR [40] both target on these purposes.

The current state of AKG still shows some limitations, in particular with respect to compilation time, tile shapes/sizes and fusion strategies. AKG uses the *isl* scheduler [65] to compute a new schedule. While improving the performance of generated code, the ILP-based scheduling heuristics [9, 17] of *isl* are infamous for their long compilation time, especially in the case of multi-dimensional tensor computations.

We minimize the effect of solving ILP problems by tightening the range of feasible solutions using a fine-tuned combination of scheduling options, ensuring compilation time be less than one minute for 99% scenarios and always less than half an hour. On the contrary, an expert has to spend tens to hundreds of hours to write a good schedule templates, or several weeks to develop an optimized CCE code. Nonetheless, it is still far from the expectation for a just-in-time compiler, and we plan to optimize the scheduling process in the future. The model-driven [10, 36] or dimension-clustering [54] approaches may be interesting directions to follow.

The reverse policy proposed in [70], which is also adopted by AKG, to determine the tile shapes and the follow-up fusion strategies has shown its effectiveness, but it can only handle forward convolutions since the approach starts by tiling a live-out iteration space. Investigation on the composition of tiling and fusion for backward convolutions is still an ongoing research.

Finally, parametric tiling [23, 35] has not been integrated into our work, which is a difficult and open issue to be solved in the context of polyhedral compilation. We are now working on this topic to support more complex scenarios like dynamic shaped tensors; this functionality will be released soon.

Acknowledgments

The authors are grateful to the shepherd, Louis-Noël Pouchet, and the anonymous reviewers for their comments. We would also like to acknowledge Cédric Bastoul, Lei Chen and many people from the Huawei MindSpore team for their feedback and discussions on the early versions of this paper. This work was partly supported by the National Natural Science Foundation of China under Grant No. U20A20226 and 61702546.

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. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[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. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proc. of the 23rd Intl. Conf. on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT'14)*. ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 193–205. http://dl_acm.gg363.site/citation.cfm?id=3314872.3314896

[5] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. 2009. Automating the Generation of Composed Linear Algebra Kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC'09)*. ACM, New York, NY, USA, Article 59, 12 pages. https://doi.org/10.1145/1654059.1654119

[6] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. arXiv:2006.12645 [cs.PL]

[7] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. arXiv:2003.00532 [cs.PF]

[8] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) *(PACT'10)*. ACM, New York, NY, USA, 343–352. https://doi.org/10.1145/1854273.1854317

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI'08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[10] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic Generation of Multi-Objective Polyhedral Compiler Transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT'20)*. ACM, New York, NY, USA, 83–96. https://doi.org/10.1145/3410463.3414635

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

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

[14] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. arXiv:1801.08058 [cs.DC]

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[16] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/3211346.3211354

[17] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming* 21, 6 (1992), 389–420.

[18] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[19] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No.98CH36181)*, Vol. 3. 1381–1384 vol.3.

[20] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

[21] Junli Gu, Yibing Liu, Yuan Gao, and Maohua Zhu. 2016. OpenCL Caffe: Accelerating and Enabling a Cross Platform Machine Learning Framework. In *Proceedings of the 4th International Workshop on OpenCL* (Vienna, Austria) *(IWOCL'16)*. ACM, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/2909437.2909443

[22] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT'20)*. ACM, New York, NY, USA, 71–82. https://doi.org/10.1145/3410463.3414632

[23] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. 2009. Parametric Multi-Level Tiling of Imperfectly Nested Loops. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS'09)*. ACM, New York, NY, USA, 147–157. https://doi.org/10.1145/1542275.1542301

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference*

on Computer Vision and Pattern Recognition (CVPR). 770–778. https://doi.org/10.1109/CVPR.2016.90

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[26] Huawei. 2021. MindSpore. https://www.mindspore.cn/en

[27] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. ACM, New York, NY, USA, 319–329. https://doi.org/10.1145/73560.73588

[28] Abhinav Jangda and Uday Bondhugula. 2018. An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP'18)*. ACM, New York, NY, USA, 261–275. https://doi.org/10.1145/3178487.3178507

[29] Abhinav Jangda and Arjun Guha. 2020. Model-Based Warp Overlapped Tiling for Image Processing Programs on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT'20)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/3410463.3414649

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM'14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[31] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP'19)*. ACM, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[32] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. arXiv:1912.03413 [cs.DC]

[33] 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA'17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[34] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 301–320.

[35] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-level Tiling: M for the Price of One. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC'07)*. ACM, New York, NY, USA, Article 51, 12 pages. https://doi.org/10.1145/1362622.1362691

[36] Martin Kong and Louis-Noël Pouchet. 2019. Model-Driven Transformations for Multi- and Many-Core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 469–484. https://doi.org/10.1145/3314221.3314653

[37] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI'13)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2491956.2462187

[38] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI'07)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/1250734.1250761

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[40] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[41] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[42] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201383

[43] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE, 1–44. https://doi.org/10.1109/HOTCHIPS.2019.8875654

[44] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA'16)*. IEEE Press, 393–405. https://doi.org/10.1109/ISCA.2016.42

[45] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[46] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1025–1040. https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[47] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[48] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (July 1996), 424–453. https://doi.org/10.1145/233561.233564

[49] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. 2014. Revisiting Loop Fusion in the Polyhedral Framework. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP'14)*. ACM, New York, NY, USA, 233–246. https://doi.org/10.1145/2555243.2555250

[50] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[51] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS'15)*. ACM, New York, NY, USA, 429–443. https://doi.org/10.1145/2694344.2694364

[52] Angshuman Parashar, Prasanth Chatarasi, and Po-An Tsai. 2021. Hardware Abstractions for targeting EDDO Architectures with the Polyhedral Model. In *11th International Workshop on Polyhedral Compilation Techniques* (Vitural Event) *(IMPACT 2021)*. 12 pages.

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[54] Benoît Pradelle, Benoît Meister, Muthu Baskaran, Athanasios Konstantinidis, Thomas Henretty, and Richard Lethin. 2016. Scalable Hierarchical Polyhedral Compilation. In *2016 45th International Conference on Parallel Processing (ICPP)*. 432–441. https://doi.org/10.1109/ICPP.2016.56

[55] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI'13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[56] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean Francois Dietiker, Madhava Syamlal, and Michael James. 2020. Fast Stencil-Code Computation on a Wafer-Scale Processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC'20)*. IEEE Press, Article 58, 14 pages.

[57] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907 [cs.PL]

[58] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS'13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[59] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD'16)*. ACM, New York, NY, USA, 2135. https://doi.org/10.1145/2939672.2945397

[60] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, USA) *(ISCA'82)*. IEEE Computer Society Press, Washington, DC, USA, 112–119.

[61] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI'16)*. ACM, New York, NY, USA, 209–223. https://doi.org/10.1145/2908080.2908105

[62] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[63] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software* (Kobe, Japan) *(ICMS'10)*. Springer-Verlag, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[64] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[65] Sven Verdoolaege and Gerda Janssens. 2017. Scheduling for PPCG. *Report CW* 706 (2017).

[66] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (San Jose, CA) *(SC'98)*. IEEE Computer Society, USA, 1–27.

[67] Keiji Yanai, Ryosuke Tanno, and Koichi Okamoto. 2016. Efficient Mobile Implementation of A CNN-Based Object Recognition System. In *Proceedings of the 24th ACM International Conference on Multimedia* (Amsterdam, The Netherlands) *(MM'16)*. ACM, New York, NY, USA, 362–366. https://doi.org/10.1145/2964284.2967243

[68] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor Compilation via the Nested Polyhedral Model. arXiv:1903.06498 [cs.DC]

[69] Jie Zhao and Albert Cohen. 2019. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Trans. Archit. Code Optim.* 16, 4, Article 47 (Dec. 2019), 25 pages. https://doi.org/10.1145/3369382

[70] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Athens, Greece) *(MICRO-53)*. IEEE Press, Piscataway, NJ, USA, 427–441. https://doi.org/10.1109/MICRO50266.2020.00044

[71] Yongwei Zhao, Zidong Du, Qi Guo, Shaoli Liu, Ling Li, Zhiwei Xu, Tianshi Chen, and Yunji Chen. 2019. Cambricon-F: Machine Learning Computers with Fractal von Neumann Architecture. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA'19)*. ACM, New York, NY, USA, 788–801. https://doi.org/10.1145/3307650.3322226

[72] 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. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[73] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. 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* (Lausanne, Switzerland) *(ASPLOS'20)*. ACM, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508