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

Jie Zhao\textsuperscript{1} Bojie Li\textsuperscript{2} Wang Nie\textsuperscript{2} Zhen Geng\textsuperscript{2}
Renwei Zhang\textsuperscript{2} Xiong Gao\textsuperscript{2} Bin Cheng\textsuperscript{2} Chen Wu\textsuperscript{2}
Yun Cheng\textsuperscript{2} Zheng Li\textsuperscript{2} Peng Di\textsuperscript{2}\textdagger\hspace{1em} Kun Zhang\textsuperscript{2}\textdblame\hspace{1em} Xuefeng Jin\textsuperscript{2}

\textsuperscript{1}State Key Laboratory of Mathematical Engineering and Advanced Computing, China
\textsuperscript{2}Huawei Technologies Co. Ltd., China
\textdagger\hspace{1em} Now with Ant Group, China
\textdblame\hspace{1em} Now with Tencent Penglai Lab, China

2021.06.23, Virtual, Canada

42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI’21)
Why DL Compilers for NPUs

depth learning frameworks

Prior DL compilers [2, 10] do not support code generation for NPUs. We present AKG in this paper to implement Automatic Kernel Generation for NPUs using Polyhedral Transformations.
Why DL Compilers for NPUs

users  →  ease of use  →  deep learning frameworks

hidden  →  hardware targets

CPU  →  GPU  →  FPGA  →  NPU
Why DL Compilers for NPUs

users

ease of use

depth learning frameworks

hidden

depth learning compilers

hardware targets

Prior DL compilers \cite{2, 10} do not support code generation for NPUs. We present AKG in this paper to implement Automatic Kernel Generation for NPUs using Polyhedral Transformations.
Why DL Compilers for NPUs

users

ease of use

deep learning frameworks

hidden

hardware targets

flexibility

efficiency

deep learning compilers

Prior DL compilers \[2, 10\] do not support code generation for NPUs.
We present AKG in this paper to implement Automatic Kernel Generation for NPUs using Polyhedral Transformations.
Why DL Compilers for NPUs

Prior DL compilers [2, 10] do not support code generation for NPUs.
Prior DL compilers [2, 10] do not support code generation for NPUs.

We present AKG in this paper to implement Automatic Kernel Generation for NPUs using Polyhedral Transformations.
Challenges faced by DL Compilers for NPUs

Google TPU [6]  
Huawei Ascend [8]
Challenges faced by DL Compilers for NPUs

Google TPU [6]  
Huawei Ascend [8]

- Effective scheduling for the conflicting demands of parallelism and locality.
Challenges faced by DL Compilers for NPUs

- Effective scheduling for the conflicting demands of parallelism and locality.
- Software-controlled storage management between multi-level, multi-directional memory hierarchy.
Challenges faced by DL Compilers for NPUs

- Effective scheduling for the conflicting demands of parallelism and locality.
- Software-controlled storage management between multi-level, multi-directional memory hierarchy.
- Automatic implementation of domain-specific transformations for convolution.

Overview of Our Approach

- MindSpore
- TensorFlow
- PyTorch
- MxNet
- Caffe
- ...

Tensor Expression

- Polyhedral
  - Polyhedral Schedule Tree
  - Loop Fusion for Locality
  - Loop Tiling
  - Loop Fission for Parallelism
  - Storage Management

- Auto Tiling

- Hardware Spec

- Codegen
  - Backend Optimizations
  - Instruction Emitter
  - Synchronization
  - Low-level Assembly

- Auto Tuning
AKG inherits the graph engine and DSL of TVM [2] for expressing tensor computations.
AKG branches from TVM by lowering HalideIR [9] generated by the DSL to schedule trees.
AKG leverages versatile polyhedral scheduling algorithms, exploiting parallelism and locality of programs simultaneously.
AKG models the interplay between loop fusion and tiling, achieving automatic decoupled data orchestration between memory hierarchy.
AKG takes as input an external schedule tree to implement the \textit{img2col} transformations [5] for convolutions.
AKG also implements vectorization, low-level synchronization, auto-tuning, improving the performance of its generated code.
The polyhedral model \([1, 3, 12]\) is a mathematical abstraction used to analyze and optimize programs.
One can lower a tensor program written by TVM’s DSL to a so-called *schedule tree* representation [4] of the polyhedral model.
One can lower a tensor program written by TVM’s DSL to a so-called *schedule tree* representation [4] of the polyhedral model.
One can lower a tensor program written by TVM’s DSL to a so-called *schedule tree* representation [4] of the polyhedral model.
The schedule tree is functional due to its rich set of node types:

- a domain node, filter nodes
- band nodes, sequence nodes and set nodes
- extension nodes
- mark nodes
- and more ...
We leverage the ILP-based isl scheduler \cite{11, 13} to compute new schedules that exploit parallelism and temporal locality simultaneously. The polyhedral scheduler exposes a wider set of affine transformations than TVM, enabling auxiliary loop transformations like skewing, shifting, scaling.

The polyhedral model first computes a loop fusion configuration, based on which loop tiling is performed automatically.

```
for h in [0,H], w in [0,W):
    A[h,w] = A[h,w] + bias  // S_0
for h in [0,H-KH], w in [0,W-KW]:
    C[h,w] = 0  // S_1
for kh in [0,KH], kw in [0,KW]:
    C[h,w] += A[h+kh,w+kw]*B[kh,kw]  // S_2
for h in [0,H-KH], w in [0,W-KW]:
    C[h,w] = abs(C[h,w])  // S_3
for h in [0,H-KH], w in [0,W-KW]:
    C[h,w] = ReLU(C[h,w])  // S_4
```
We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.
We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.
We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.

The polyhedral scheduler exposes a wider set of affine transformations than TVM, enabling auxiliary loop transformations like skewing, shifting, scaling.
We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.

The polyhedral scheduler exposes a wider set of affine transformations than TVM, enabling auxiliary loop transformations like skewing, shifting, scaling.

The polyhedral model first computes a loop fusion configuration, based on which loop tiling is performed automatically.
### Constructing Tile Shapes

**Domain Sequence**

```
Filter\{S_3(h,w)\}
Band\{S_0 \rightarrow (h,w)\}
Filter\{S_1(h,w) ; S_3(h,w,kh,kw) ; S_0(h,w), S_4(h,w)\}
Band\{S_1 \rightarrow (h,w) ; S_2 \rightarrow (h,w) ; S_3 \rightarrow (h,w) ; S_4 \rightarrow (h,w)\}
Sequence
Filter\{S_1(h,w)\}
Filter\{S_2(h,w,kh,kw)\}
Band\{S_3 \rightarrow (kh,kw)\}
Filter\{S_3(h,w)\}
Filter\{S_4(h,w)\}
```
The classical polyhedral compilation workflow generates two kernels.
• The classical polyhedral compilation workflow generates two kernels.
• We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.
The classical polyhedral compilation workflow generates two kernels.

We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.

The reverse strategy first tiles a live-out iteration space,
The classical polyhedral compilation workflow generates two kernels. We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel. The reverse strategy first tiles a live-out iteration space.

\{(o_0, o_1) \rightarrow A(h', w') : 0 \leq o_0 < [(H - KH + 1)/T_2] \land 0 \leq o_1 < [(W - KW + 1)/T_3] \land T_2 \cdot o_0 \leq h' < T_2 \cdot o_0 + KH + T_2 - 1 \land T_3 \cdot o_1 \leq w' < T_3 \cdot o_1 + KW + T_3 - 1\}
The classical polyhedral compilation workflow generates two kernels.

We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.

The reverse strategy first tiles a live-out iteration space, and uses the data tiles to construct tile shapes for intermediate iteration spaces.
Prior tensor compilers use default tile sizes in compilers.

We propose a tile-size specification language.

\[
\begin{align*}
\text{stmt_id} & : \ "S_" \text{ integer} \\
\text{tile_size} & : \text{ integer} \\
\text{tileSpec} & : \text{ tile_size @ buffer} \\
\text{tile_specs} & : \text{ tile_spec | tile_specs , tile_spec} \\
\text{stmt_spec} & : \text{ stmt_id : tile_specs} \\
\text{tiling_policy} & : \text{ stmt_spec | tiling_policy stmt_spec}
\end{align*}
\]
Prior tensor compilers use default tile sizes in compilers.

We propose a tile-size specification language.

\[
\begin{align*}
\text{stmt_id} & : \text{"S" integer} \\
\text{tile_size} & : \text{integer} \\
\text{tile_spec} & : \text{tile_size} \oplus \text{buffer} \\
\text{tile_specs} & : \text{tile_spec} \mid \text{tile_specs}, \text{tile_spec} \\
\text{stmt_spec} & : \text{stmt_id} : \text{tile_specs} \\
\text{tiling_policy} & : \text{stmt_spec} \mid \text{tiling_policy} \text{stmt_spec}
\end{align*}
\]

This language simplifies the tile size selection issue, which has been automated by compiler.
This relation implies the overlapped tile shape \([14]\) of the intermediate iteration space, but it has to be used together with loop fusion. The post-tiling fusion strategy models a novel composition of loop transformations. The original subtree should be skipped.

\[
\{(o_0, o_1) \rightarrow S_0(h, w) : 0 \leq o_0 < \lceil(H - KH + 1)/T_2 \rceil \land 0 \leq o_1 < \lceil(W - KW + 1)/T_3 \rceil \land T_2 \cdot o_0 \leq h < T_2 \cdot o_0 + KH + T_2 - 1 \land T_3 \cdot o_1 \leq w < T_3 \cdot o_1 + KW + T_3 - 1\}
\]
This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.
Fusion When Offloading Data

\{(o_0, o_1) \rightarrow S_0(h, w) : 0 \leq o_0 < \lceil(H - KH + 1)/T_2\rceil \land 0 \leq o_1 < \lceil(W - KW + 1)/T_3\rceil \land T_2 \cdot o_0 \leq h < T_2 \cdot o_0 + KH + T_2 - 1 \land T_3 \cdot o_1 \leq w < T_3 \cdot o_1 + KW + T_3 - 1\}\n
- This relation implies the **overlapped** tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.
Fusion When Offloading Data

This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.

The post-tiling fusion strategy models a novel composition of loop transformations.
Fusion When Offloading Data

This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.

The post-tiling fusion strategy models a novel composition of loop transformations.

The original subtree should be skipped.
This schedule tree does not manage the multi-directional memory hierarchy of Ascend.

We use mark nodes to let some statements flow to different buffers, and each "local UB" filter node can be flowed to Vector/Scalar Unit. Intra-tile rescheduling is also performed, as a reverse process of loop fusion. A filter flowed to Cube Unit is not distributed.
Fusion When Forking Data and Intra-Tile Rescheduling

This schedule tree does not manage the multi-directional memory hierarchy of Ascend.
This schedule tree does not manage the multi-directional memory hierarchy of Ascend.

We use mark nodes to let some statements flow to different buffers, and each “local_UB” filter node can be flowed to Vector/Scalar Unit.
This schedule tree does not manage the multi-directional memory hierarchy of Ascend.

We use mark nodes to let some statements flow to different buffers, and each "local_UB" filter node can be flowed to Vector/Scalar Unit.

Intra-tile rescheduling is also performed, as a reverse process of loop fusion. A filter flowed to Cube Unit is not distributed.
Optimization of Convolution

The power of the Cube Unit can be fully exploited when executing matrix multiplication. We automate the `img2col` transformation [5] by grafting an external schedule and relating it using a formula (§4.5). We also implement a fractal tiling [16] within the Cube Unit.
The power of the Cube Unit can be fully exploited when executing matrix multiplication.
The power of the Cube Unit can be fully exploited when executing matrix multiplication.

We automate the *img2col* transformation [5] by grafting an external schedule and relating it using a formula (§4.5).
The power of the Cube Unit can be fully exploited when executing matrix multiplication.

- We automate the *img2col* transformation [5] by grafting an external schedule and relating it using a formula (§4.5).
- We also implement a fractal tiling [16] within the Cube Unit.
Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
- We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.
Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
- We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.
- We design a memory hierarchy specification language that can be generated automatically, allowing for the manual scheduling to make debugging easier (§4.6).
Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).

We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.

We design a memory hierarchy specification language that can be generated automatically, allowing for the manual scheduling to make debugging easier (§4.6).

We exploit effective SIMD vectorization as a post-polyhedral step, maximizing the utilization of the hardware intrinsics (§5.1).

We implement a DP-based low-level synchronization between emitted instructions, enabling efficient instruction-level pipelining (§5.2).

We develop an auto tuning strategy to achieve better performance in practice (§5.3).
Experimental Setup

- Code is executed on the Huawei Ascend 910 chip.
- Performance is compared against (1) manually optimized CCE code written by experts, and (2) the adapted TVM schedule templates developed by the software R&D team of the chip.
- Experiment is conducted on single operators, subgraphs and end-to-end workloads.
- Each code is compiled with the same set of compilation options.
op1: conv; op2: matmul; op3: ReLU; op4: batch matmul; op5: cast; op6: transpose; op7: one-hot; op8: add; op9: bnorm reduction; op10: bnorm update

CCE opt is 2.8× faster than CCE naïve.
AKG achieves the performance comparable to CCE opt, with a mean loss within 4%.
AKG outperforms adapted TVM by 1.6× on average.
Results of Single Operators

CCE opt is 2.8× faster than CCE naïve.

AKG achieves the performance comparable to CCE opt, with a mean loss within 4%.

AKG outperforms adapted TVM by 1.6× on average.
Results of Single Operators

Comparison of lines of code (lower is better).

- AKG significantly reduces development efforts compared to the optimized CCE code and adapted TVM schedule templates.
Performance of GEMM product under different shape configurations (1 \( \mu s = 10^3 \) cycles; lower is better).
Performance of GEMM product under different shape configurations (1 $\mu s = 10^3$ cycles; lower is better).

- 41 different shape configurations ranging from (64,64) to (4608,4608) are used to evaluate the performance of matrix multiplication.
- AKG outperforms the adapted TVM under 29 out of the 41 shape configurations.
## Summary of the subgraphs.

<table>
<thead>
<tr>
<th>no.</th>
<th># of ops</th>
<th>precision</th>
<th>batch size</th>
<th>input shape</th>
<th>output shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>FP16</td>
<td>16</td>
<td>(16,16,512,512)</td>
<td>(16,16,512,512)</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>FP16</td>
<td>16</td>
<td>(256,512,16,16)</td>
<td>(256,512,16,16)</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>FP32</td>
<td>16</td>
<td>(30522,1024)</td>
<td>(30522,1024)</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>FP32</td>
<td>16</td>
<td>(1024,1024)</td>
<td>(1024,1024)</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>FP16</td>
<td>16</td>
<td>(64,1,16,16)</td>
<td>(64,1,16,16)</td>
</tr>
</tbody>
</table>
### Summary of the subgraphs.

<table>
<thead>
<tr>
<th>no.</th>
<th># of ops</th>
<th>precision</th>
<th>batch size</th>
<th>input shape</th>
<th>output shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>FP16</td>
<td>16</td>
<td>(16,16,512,512)</td>
<td>(16,16,512,512)</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>FP16</td>
<td>16</td>
<td>(256,512,16,16)</td>
<td>(256,512,16,16)</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>FP32</td>
<td>16</td>
<td>(30522,1024)</td>
<td>(30522,1024)</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>FP32</td>
<td>16</td>
<td>(1024,1024)</td>
<td>(1024,1024)</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>FP16</td>
<td>16</td>
<td>(64,1,16,16)</td>
<td>(64,1,16,16)</td>
</tr>
</tbody>
</table>

#### Performance of subgraphs (higher is better).

![Graph showing performance comparison of subgraphs](image)
### Results of Subgraphs

#### Summary of the subgraphs.

<table>
<thead>
<tr>
<th>no.</th>
<th># of ops</th>
<th>precision</th>
<th>batch size</th>
<th>input shape</th>
<th>output shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>FP16</td>
<td>16</td>
<td>(16,16,512,512)</td>
<td>(16,16,512,512)</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>FP16</td>
<td>16</td>
<td>(256,512,16,16)</td>
<td>(256,512,16,16)</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>FP32</td>
<td>16</td>
<td>(30522,1024)</td>
<td>(30522,1024)</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>FP32</td>
<td>16</td>
<td>(1024,1024)</td>
<td>(1024,1024)</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>FP16</td>
<td>16</td>
<td>(64,1,16,16)</td>
<td>(64,1,16,16)</td>
</tr>
</tbody>
</table>

#### Performance of subgraphs (higher is better).

- AKG produces an average speedup of $1.3 \times$ and $5.6 \times$ over the adapted TVM and CCE opt.
Performance of end-to-end workloads (higher is better).
Results of End-to-end Workloads

- CCE opt only optimizes one end-to-end workload (ResNet-50).
- AKG performs similarly to the adapted TVM for ResNet-50, MobileNet and AlexNet, but outperforms the latter by 20.2% on Bert and SSD.
- The manual approaches take days to weeks to optimize a workload, but AKG only requires minutes to hours.
AKG carefully handles the interplay between tiling and fusion using a reverse strategy [15], a platform-neutral transformation.

AKG adopts a hierarchical fusion approach that can be adapted to other NPU architectures [6].

AKG automates the domain-specific transformations of convolution. While the fractal tiling [16] is Ascend-specific, the img2col transformation [5] can be used as a general method.

AKG also extends the expressiveness of the schedule tree representation, sharing the same objective (i.e., delivering domain-specific knowledge) with MLIR [7].
Questions & Answers

The paper is available at

![QR Code](image1)

The code of AKG is available at

![QR Code](image2)

Thank you!

Any Questions?
References

   A practical automatic polyhedral parallelizer and locality optimizer.

   Tvm: An automated end-to-end optimizing compiler for deep learning.

   Polly–performing polyhedral optimizations on a low-level intermediate representation.
   *Parallel Processing Letters* 22, 04 (2012), 1250010.

   Polyhedral ast generation is more than scanning polyhedra.

   Opencl caffe: Accelerating and enabling a cross platform machine learning framework.
   In *Proceedings of the 4th International Workshop on OpenCL* (New York, NY, USA, 2016), IWOCL’16, ACM.
References


In-datacenter performance analysis of a tensor processing unit.


Mlir: Scaling compiler infrastructure for domain specific computation.

[8] Liao, H., Tu, J., Xia, J., and Zhou, X.

Davinci: A scalable architecture for neural network computing.


Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
References

The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically.

Isl: An integer set library for the polyhedral model.

Polyhedral parallel code generation for cuda.

[13] **Verdooldaege, S., and Janssens, G.**
Scheduling for ppcg.

[14] **Zhao, J., and Cohen, A.**
Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation.

[15] **Zhao, J., and Di, P.**
Optimizing the memory hierarchy by compositing automatic transformations on computations and data.

Cambricon-f: Machine learning computers with fractal von neumann architecture.