Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically

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

Cédric Bastoul
Huawei Technologies France SASU
Paris, France
cedric.bastoul@huawei.com

Yanzhi Yi
Huawei Technologies Co., Ltd.
Beijing, China
yiyanzhi@huawei.com

Jiahui Hu
Huawei Technologies Co., Ltd.
Beijing, China
hujiahui8@huawei.com

Wang Nie
Huawei Technologies Co., Ltd.
Beijing, China
peter.nie@huawei.com

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

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

Chong Li
Huawei Technologies France SASU
Paris, France
ch.l@huawei.com

Thibaut Tachon
Huawei Technologies France SASU
Paris, France
thibaut.tachon@huawei.com

Zhiliang Gan
Huawei Technologies Co., Ltd.
Shenzhen, China
ganzhiliang@huawei.com

ABSTRACT

Due to the missing of a good orchestration of loop transformations, existing optimizing compilers for deploying neural networks on GPU either parallelize reductions ineffectively or miss the fusion opportunities with other operators. Neural network models thus exhibit sub-optimal performance on GPU. We present a practical approach called Panamera for the effective parallelization of reductions in neural networks on GPU. Panamera first leverages loop coalescing to flatten the loop dimensions of reductions, converting all reduction operators into canonical forms eligible for the polyhedral model. Next, Panamera uses polyhedral transformations to reduce the data movements caused by unfused reductions and perform multi-block hardware binding not considered by many compilers. Finally, Panamera embeds a highly optimized routine implemented using GPU atomic instructions, further improving the performance of neural network models while guaranteeing the correctness of parallel reductions. The experimental results demonstrate the effectiveness of our approach: for single operators our code obtains a mean speedup of 33.7×, 3.5×, 5.4× and 9.6× over cuDNN, CUB, TVM and Ansor, for sub-graphs our approach outperforms cuDNN, TVM and Ansor by 9.5×, 2.6× and 2.7×, and for end-to-end workloads, a tensor compiler integrated with our approach outperforms them by 122.5%, 19.3% and 15.2%.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; **Translator writing systems and compiler generators**.

KEYWORDS

deep learning, reduction, GPU, polyhedral compilation

ACM Reference Format:
Jie Zhao, Cédric Bastoul, Yanzhi Yi, Jiahui Hu, Wang Nie, Renwei Zhang, Zhen Geng, Chong Li, Thibaut Tachon, and Zhiliang Gan. 2022. Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically. In *PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT), October 10–12, 2022, Chicago, IL.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Deep learning (DL) applications demand for extraordinary computing power. As NVIDIA GPU nowadays still dominates the market for DL accelerators, effectively deploying DL models on GPU is an important research topic [9, 54]. A DL model is composed of many operators, among which the most compute-intensive ones are convolution and matrix multiplication. Existing methods [6, 9, 23, 54] are effective to improve the execution performance of DL models by deeply optimizing such compute-intensive operators.

*Zhen Geng was a technical expert of AI Compiler at Huawei when doing this work. He is now with the Parallel Computing Software Team at Alibaba, Hangzhou, China.

PACT '22, October 10–12, 2022, Chicago, IL

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

There also exist many operators like Softmax, ReLU, Batch Normalization, *etc.*, that are less compute-intensive compared to convolution and matrix multiplication but also important to the execution performance of a DL workload. The overall execution performance of a DL workload can be sub-optimal if such operators cannot be effectively optimized. The commonness of these operators is that they all involve *reduction*, which applies a binary operator to each element of an input vector and reduces the vector to a single value. Parallelizing reductions is thus important for DL workloads. Unfortunately, many optimizing compilers either parallelize this specific computation pattern ineffectively [9, 47, 54, 64] or lose the optimization opportunities for operator fusion [15, 23].

Research on parallel reductions on GPU has a long history. The investigation of Harris [27] explained in detail how parallel reductions can be performed on GPU. More recent work [20, 22, 49] further enhanced and generalized the approach of Harris. However, these methods did not consider the challenges faced by parallelizing reductions in DL workloads, and few of them are integrated into optimizing compilers for DL applications. Parallel reductions in DL workloads are handled using either of the following two ways.

First, compilation techniques for DL models [9, 47, 54] use language constructs and loop transformations to parallelize reductions. They fuse multiple network layers and decompose the fusion results by abstracting away the architectural features of GPU. However, *a fused operator is ineffectively decomposed* when nested reductions over multiple variables that possess several smaller reduced dimensions are involved: while decomposing only one of these small nested reduced dimensions to a single GPU block results in a waste of hardware resources, dispatching multiple of them to more than one blocks (though not supported by these approaches) has to sacrifice the parallelism of other fully parallelizable dimensions.

The second approach for parallel reductions is making use of CUDA libraries like Thrust [5], cuDNN [11] and CUB [41]. Different from the aforementioned approaches, these highly tuned libraries are written by GPU experts and can enable multi-block parallelism for reductions. Yet *these libraries do not scale with the diverse data types or tensor shapes*, as will be demonstrated in our experiments. Some optimizing compilers like XLA [23] and Diesel [15] map a reduction to these CUDA routines, but their implementation is not compatible with profitable loop transformations and misses the fusion opportunities with other operators. *The resulting data movement across the memory hierarchy of GPU cancels out the performance improvement brought by CUDA libraries.*

In this paper, we present Panamera, a practical approach to **PA**rallelize **N**eur**A**l network **M**odels **E**ffectively on GPU by implementing **R**eductions **A**tomically. To deal with nested reductions over multiple variables that take place frequently in DL, Panamera carefully implements *loop coalescing* [45] in an intelligent way: it neither designs custom schedule primitives [53] nor models the transformation as a black-box optimization [7, 54, 58]; instead, it isolates this transformation as a pre-processing step from the polyhedral schedulers [7, 16] and uses loop coalescing to normalize nested reductions in DL workloads into three canonical forms. Such a handling of reductions brings the first insight to existing optimizing compilers: it simplifies the scheduling algorithms, which are used by Panamera and many other polyhedral compilers [54, 64], by avoiding the need to introduce complicated constraints to enable/disable this undesired transformation [59]. Importantly, as this pre-processing step is implemented on top of the HalideIR [47], it can be easily implemented in Tensor Comprehensions (TC) [54] and TVM [9] that also (at least originally) use HalideIR.

Next, Panamera performs polyhedral loop fusion and tiling on the canonical reductions and binds the transformed loop nests to GPU hardware. While these standard transformations are not new, performing them on the three canonical forms contributes the second insight to the compiler community by bringing about two benefits. (1) The canonical forms flatten multiple small reduced dimensions into larger one, allowing a compiler to decompose the larger reduced dimension across multiple thread blocks, which was not considered by TVM [9]. (2) As each canonical form decreases the number of nested loop dimensions, the numbers of tile sizes and block/thread configurations are also reduced, which can tighten the tuning space of an autotuner [65] for DL compilers.

Finally, Panamera embeds highly tuned routines to appropriate positions of its generated code by making use of GPU *atomic instructions* [22, 37], allowing Panamera to scale with complex scenarios. Traditionally, a reduction can be fused with its dependent operators through loop fusion, which is implemented during the polyhedral transformations of Panamera, but two independent reductions cannot be merged because the polyhedral model exploit fusion based on dependences between two operators and there exist no dependences between them. We make it possible to fuse two independent reductions in our code generator by carefully selecting the identical hardware configuration in the highly tuned routines, increasing the fusion possibilities of reductions in DL workloads. Besides, we provide the code templates in this paper, which other developers can easily integrate to their systems. This is the third insight offered by Panamera to the compilation techniques.

In the experiments, we first demonstrate that Panamera can outperform cuDNN [11], CUB [41], TVM [9] and Ansor [65] by 33.7×, 3.5×, 5.4× and 9.6×, respectively, for single operators. We then use sub-graphs to show the compound effect of libraries and loop transformations, resulting in an average speedup of 9.5×, 2.6× and 2.7× over cuDNN, TVM and Ansor. The results of end-to-end workloads are finally reported, with a mean improvement of 122.5%, 19.3% and 15.2% achieved by a tensor compiler that has been integrated with Panamera over MindSpore [32] backed by cuDNN and cuBLAS [40], TVM and Ansor.

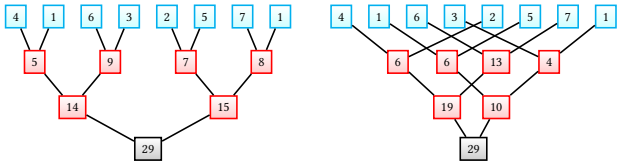
In summary, this work makes the following contributions.

- Panamera canonicalizes reductions in DL not considered before [9, 64], making it possible to effectively decompose various reductions and fully harness GPU hardware resources.
- Panamera implements a good orchestration of loop transformations for reductions, avoiding the need to introduce complex constraints in polyhedral schedulers [59] and decreasing the tuning space size of DL reductions [9, 65].
- Panamera exhibits a much better scalability to data types and tensor shapes than many CUDA libraries [11, 41], rendering a compiler applicable to various reduction scenarios.
- Panamera enables fusion of independent reductions, further improving the fusion possibilities and validating that there still exists space for optimizing reductions.

The paper is organizes as follows. Section 2 introduces the background. Section 3 explains dimension flattening. Section 4 presents loop fusion and tiling. Section 5 describes the library implementation, followed by the potentials and limitations discussed in Section 6, experimental results reported in Section 7 and related work compared in Section 8. Section 9 concludes the paper.

2 BACKGROUND AND OVERVIEW

We consider reduction of *associative* and *commutative* operators. These computational properties authorize the parallel execution of this operator by reorganizing the computational order between the input numerical elements. Figure 1 depicts two variants of the parallel summation using a binary tree structure when given a list of (blue) inputs with size n. These implementations reduce the frequency to compute the (red) partial results from $n-2$ to $\log_2 n - 1$ when given a set of (blue) inputs with size n. As the summations of the (red) partial results along the same horizontal level are independent, parallel executions can be performed using $n/2$ parallel processors. This makes GPU suitable for this task with the growth of input size, but the parallelization of reductions on GPU is non-trivial.



(a) Interleaved addressing. **(b) Sequential addressing.**

Figure 1: The tree-based parallel reductions.

2.1 Parallel Reductions on GPU

The research on parallel reduction on GPU [27] revealed the optimization criteria. For example, sequential addressing (Figure 1b) can outperform interleaved addressing (Figure 1a) due to the absence of bank conflicts on shared memory. A careful craft of the code is also necessary to avoid the thread idleness while data loading from global memory. As this optimization is usually performed by hand, manually tuned libraries [5, 11, 41] are still competitive candidates for parallel reduction on GPU.

GPU abstracts the streaming multiprocessors (SMs) as blocks and the CUDA cores as threads. The maximal number of the allocatable threads within a block is limited. Parallel reduction can be executed within a block by dispatching one or more loop iterations to each thread, but one should decompose a reduction into multiple parallel blocks with the growth of the input data size for improving performance. However, GPU does not provide global synchronization across blocks; decomposing a reduction across multiple blocks is thus non-trivial. Existing solutions break down a reduction into multiple GPU kernels [27], which is impractical for DL models since multiple kernels should be generated for one reduction operator.

The manual routines [5, 11, 41] are also incompatible with profitable loop transformations, especially (1) *loop coalescing* [45] that is performance-critical for DL reduction as will be demonstrated in this work, and (2) *loop fusion* [38], which can be used to create more intermediate variables that can be allocated on shared memory [62]. Optimizing parallel DL reduction using vendor libraries alone thus misses many opportunities to benefit from faster shared memory. A promising but also very challenging solution is to combine the high-performance implementations with compilation approaches capable of managing various loop transformations systematically.

2.2 Polyhedral Parallel Reductions

Hardware binding and loop transformations can be implemented using the polyhedral model [17]. It performs loop fusion using heuristics that are integrated into the scheduling algorithms [7, 59], which in turn compute a combination of auxiliary loop transformations beneficial to loop fusion by solving integer linear programming (ILP) problems. An ILP problem is established using dependences between statement instances; the scheduling algorithms and the fusion heuristics can thus guarantee the validity of the modeled transformations. The loop transformations can also be managed using the polyhedral representation [25], on top of which hardware binding can be conducted.

Parallel reduction in the polyhedral model is handled by relaxing the induced reduction dependences between loop iterations [14, 49, 51, 56]. This allows the model to perform loop tiling [33] using a given fusion configuration. More specifically, a tiling transformation groups the iterations of a loop nest into smaller active working sets, with outer parallel dimensions (tile loops) iterating between these working sets and inner parallel dimensions (point loops) within a working set. Hardware binding is achieved by relating tile loops to GPU blocks and point loops to GPU threads.

However, such approaches cause a waste of hardware resources when parallelizing only one small reduced dimension of nested reductions over multiple variables, which take place frequently in DL. While not yet studied, decomposing multiple small reduced dimensions of such cases are often achievable by sacrificing the parallelism of other fully parallelizable dimensions, because GPU offers at most 3D parallelism. Hence, a loop transformation, *i.e.*, loop coalescing [45], that reduces the dimensionality of the loop nests where nested reductions over multiple variables happen should be used. Unfortunately, as loop coalescing is harmful to the simplification criterion of many existing polyhedral schedulers [7, 16], it is an undesired transformation, though it can be produced in some rare cases by introducing complicated scheduling constraints [59]. As such, modeling loop coalescing as a black-box transformation using the polyhedral schedulers is non-trivial.

Even loop coalescing can be produced by a polyhedral scheduler and a fattened reduced dimension can be dispatched to multiple blocks, polyhedral parallel reduction still has to address the global synchronization between them. This issue, however, is ineffectively addressed by privatizing the partial results [14], missing the opportunity to harness the low-level atomic instructions. An alternative to the privatization strategy and code generation of the polyhedral approaches is to wrap highly tuned routines, like what TC [54] did by wrapping CUB [41], but this approach is restricted to innermost loops, which is not sufficient for the diverse DL reduction scenarios. Moreover, the ineffective handling of partial tiles [35] with irregular number of loop iterations produced by loop tiling in TC also leads to inferior performance in practice.

2.3 Overview of Our Approach

PANAMERA borrows the domain-specific language (DSL) of TVM to rewrite a fused sub-graph as tensor expressions. A sub-graph is generated by the graph engine of AKG [63], which can import a deep neural network expressed using popular DL frameworks [1, 44].

PANAMERA first performs loop coalescing [45], which flattens the loop nest of a reduction operator into a two-dimensional (2D) loop nest or a single (1D) loop. This allows us to focus on 1D or 2D reductions, covering all types of DL reductions. Isolating loop coalescing as pre-processing simplifies the polyhedral scheduling by avoiding the need for complicated scheduling constraints [59], but it comes at the price of losing the expressiveness for loop interchange [2] in the later. We will well address this side effect in Section 3.1.

The flattened input code is lowered to the polyhedral representation [25], on top of which the *isl* scheduler [59] is used to perform loop fusion and tiling, possibly with the combination of auxiliary loop transformations. The reduction dependences are ignored when performing hardware binding, which allows for the decomposition of a reduced dimension across multiple thread blocks.

To generate a single kernel, we use atomic instructions to write back each result of a block to global memory. We introduce our library implementation to an appropriate position within each block, which tunes the parallel execution of a reduction by considering the optimizations not expressible in polyhedral compilation. We finally generalize the approach for more complex reduction cases.

3 DIMENSION FLATTENING

A DL model solves complex problems using abundant data of multiple dimensions. The data of a DL model is expressed as tensors or multi-dimensional arrays, the operations of which are usually encompassed by deeply nested loops. For instance, an image classification model usually takes 4D tensors as input and performs operations on these input tensors. The enclosing loop nest of each tensor operator is composed of at least four dimensions. As such, *nested reductions over multiple variables* may happen in tensor reduction operators, which encourages us to consider more complicated reduction patterns than those covered by existing approaches [11, 27, 37].

Nested reductions over multiple variables also complicate the polyhedral scheduling algorithms and thus discourage many loop transformations. We introduce a dimension flattening optimization as a pre-processing step to generalize the various DL reduction patterns. To achieve this purpose, we refine *loop coalescing* [45], a loop transformation that combines nested loops into a single loop.

3.1 Loop Coalescing

We first assume there exists only one reduction operator in a subgraph, and will discuss the handling of multiple reduction operators in Section 5.3. A reduction operator only induces dependences along the reduced loop dimensions, allowing us to characterize each loop of a reduction operator as either *reduced* or *parallel*. We flatten a reduction operator's loop nest to a 1D loop when reductions are performed over all loop variables; otherwise, we combine all reduced dimensions into one reduced loop and all parallel dimensions to another, resulting in a 2D loop nest.

Loop coalescing is always valid because it only changes the loop structure but not the order of computation [24] by specializing the code using a different way to reduce the control overhead. It can be applied without further modifications when flattening all reduced dimensions in Figure 2a into a 1D reduced loop; it can also be performed safely when both parallel dimensions and reduced dimensions are continuously nested, as shown in Figure 2b and 2c.

Yet one cannot directly flatten the pattern shown in Figure 2d but has to resort to an *interchange* transformation [2] to make the pattern align with either of those shown in Figure 2b and 2c. It is always valid to transform the interleaved pattern shown in Figure 2d into Figure 2b or Figure 2c, since the permutation always happens between a parallel dimension and a reduced loop without violating any dependences.

Loop interchange may be harmful to *memory coalescing*, an optimization featured by GPU to compensate long access overhead by combining multiple memory requests from parallel threads to adjacent locations into a single memory transaction. Fortunately, data layout is easy to reason about in a DSL: tensor transpositions can be introduced by reshaping tensors to guarantee that the permuted loop dimensions always scan consecutive memory addresses. Updating data layout before dimension flattening is important because the polyhedral model will not be able to perform loop interchange between the flattened dimensions, overcoming the weakness of the isolation of loop coalescing from the polyhedral model. PANAMERA maximizes the opportunity to benefit from memory coalescing, though long access latency may still be unavoidable in some corner cases where conflicting demands caused by reshaping different tensors take place. We did not observe such cases in our experiments.

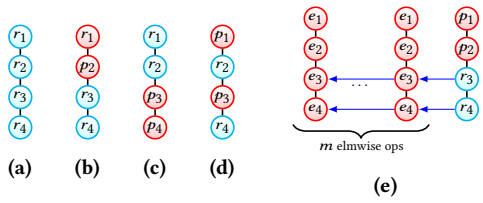


Figure 2: The loop nested patterns: (a) Reductions over all loop dimensions; (b) and (c) Both the (red) parallel dimensions and the (blue) reduced dimensions are continuous; (d) The parallel dimensions and the reduced dimensions are interleaved; (e) When coupled with elementwise operators. *r*, *p* and *e* are short for reduction, parallel and elementwise, respectively. Blue arrows denote dependence propagation.

As a result, loop coalescing can always transform these nested patterns into one of the canonical forms shown in Figure 3. We use R to represent the reduction statement. The nested pattern shown in Figure 2a can be flattened into the form shown in Figure 3a, which we refer to as *all*-reduce. The patterns shown in Figure 2b and 2c can be transformed into the code shown in Figure 3b and 3c, respectively, and we use *x*-reduce and *y*-reduce to represent these versions.

```
reduced for j=0 to N
  R(j_1, ..., j_r);
```
(a) *all*-reduce.

```
parallel for i=0 to M
  reduced for j=0 to N
    R(i_1, ..., i_p, j_1, ..., j_r);
```
(b) *x*-reduce.

```
reduced for j=0 to N
  parallel for i=0 to M
    R(j_1, ..., j_r, i_1, ..., i_p);
```
(c) *y*-reduce.

Figure 3: The canonical forms after dimension flattening.

The parameters of the codes can be determined using

$$\begin{cases} M = \prod_{x=1}^{p} s_x = s_1 \times \cdots \times s_p,\ N = \prod_{y=1}^{r} t_y = t_1 \times \cdots \times t_r; \\ i_a = \left\lfloor i \Big/ \prod_{x=a+1}^{p} s_x \right\rfloor \bmod s_a : (1 \le a < p),\ i_p = i \bmod s_p; \\ j_b = \left\lfloor j \Big/ \prod_{y=b+1}^{r} t_y \right\rfloor \bmod t_b : (1 \le b < r),\ j_r = j \bmod t_r; \end{cases} \quad (1)$$

where p and r represent the numbers of parallel and reduced loop dimensions, respectively, and each s_x $(1 \le x \le p)$ or t_y $(1 \le y \le r)$ is used to denote the number of parallel or reduced loop iterations. We allow at most one s_x and one t_y to be symbolic constants such that M and N can be written as affine expressions. In addition, we have to recover the original subscripts of the reduction statement R from the coalesced loop dimensions. The last two sets of Formula (1) are meant to perform this recovery.

We use Figure 4a to illustrate the effect of loop coalescing. The (underlined) reduced dimensions are separated by a parallel dimension w, which triggers the loop interchange of our pre-processing steps. The reduced dimensions then become continuous and are flattened into an y-reduce form shown in Figure 4b. The loop extents and the tensor subscripts are also updated according to Formula (1).

```
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        E(h,w,x,y);
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        R(h,w,x,y);
```

(a)

```
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        E(h,w,x,y);
for i=0 to 20
  for j=0 to 40*10*5
    R(i,(j/(10*5))%40,
      (j/5)%10,j%5);
```

(b)

```
parallel for i=0 to 20
  parallel for j=0 to 40*10*5
    E(i,(j/(10*5))%40,
      (j/5)%10,j%5);
parallel for i=0 to 20
  reduced for j=0 to 40*10*5
    R(i,(j/(10*5))%40,
      (j/5)%10,j%5);
```

(c)

Figure 4: An example to illustrate the effect of dimension flattening. (a) The original code; (b) Flatten the reduction operator; (c) Propagate reductions to the elementwise operator.

Reasoning about reduction dependences using polyhedral compilation [14, 49, 51, 56] is impossible here, because the compilation flow has not yet been lowered to the polyhedral representation. Instead, a reduced dimension in DL models can be inferred using DSL [9, 54], and the bounds of a loop are always (symbolic) constants. They together make it possible to automate loop coalescing.

3.2 Reduction Propagation

Loop coalescing invalidates the originally possible fusion between a reduction operator and its preceding elementwise operators. Figure 2e depicts a reduction operator preceded by m elementwise operators. Due to the perfect dimension matching between the loop nests, these operators can be fused, but loop coalescing loses this property by changing the reduction loop nest into an x-reduce form.

A reduction is not allowed to be followed by elementwise operators in a sub-graph, since loop tiling (Section 4.2) will prevent the fusion between them. Such a requirement can be feedback to the high-level graph compiler [63] to refine its rules, which manages the interaction between a tensor optimizer and a graph compiler [23].

To make the fusion with these preceding elementwise operators still possible, we also coalesce each elementwise operator in Figure 2e. An elementwise operator never induces dependences; we can thus assume that each elementwise dimension is parallel. One may obtain a 1D parallel loop if he/she combines the enclosing loops of an elementwise operator, which does not match the x-reduce pattern.

We propagate the reduction dependences to each elementwise operator and use the blue arrows to represent such a propagation. Each pair of e_3 and e_4 dimensions of an elementwise operator will thus be considered as reduced, and our compiler can apply the same coalescing strategy to each of the m elementwise operators. A more intuitive example is shown in Figure 4c, where an elementwise operator (the first loop nest) is coalesced according to its succeeding reduction operator (the second loop nest).

4 POLYHEDRAL TRANSFORMATIONS

The polyhedral model requires affine loop and tensor subscript expressions [17] for a given program. Our pre-processing steps make the three canonical reduction forms eligible for polyhedral compilation. First, the loop parameters M and N are affine expressions due to the constraints on s_x and t_y. Second, each tensor subscript inferred using Formula (1) only involves multiplications, integer divisions and the modulo arithmetic, which can also be perfectly modeled in polyhedral compilation. One can easily lower the output of Section 3 into the polyhedral representation [25].

4.1 Loop Fusion

Loop fusion is applied by respecting each dependence, minimizing the producer-consumer relations between the DL operators and thereby maximizing the temporal locality. Loop tiling and hardware binding can then be performed based on a loop fusion configuration. We enforce *outer parallelism* in the *isl* scheduler, which always permutes a parallel loop to an outer position when possible. In other words, our *isl* scheduler transforms the loop nest of a y-reduce case shown in Figure 3c into the nested pattern shown in Figure 3b. It makes it possible to always bind a parallel loop to the outer dimension of GPU blocks and a reduced loop to the inner, minimizing the overhead of global synchronizations.

4.2 Loop Tiling and Hardware Binding

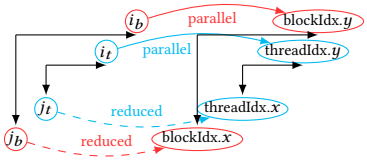
Given a fusion configuration, the polyhedral model applies loop tiling to the composed 2D loop nests or 1D loops. Each y-reduce case is converted into x-reduce by the *isl* scheduler, and *all*-reduce can be viewed as a special case of that shown in Figure 3b with M set as 0. We thus use the x-reduce pattern to illustrate how loop tiling and hardware binding are performed. Loop tiling is valid if two smaller blocks of a loop nest's iterations produced by this transformation can be executed without mutual dependences [33]. This prerequisite for the validity of loop tiling is also guaranteed by the schedulers of *isl*. Loop tiling transforms an x-reduce reduction that has been fused with elementwise operators into the code shown in Figure 5a. We use i_b,j_b to represent the tile loops and i_t,j_t for point loops.

Loop tiling is performed to align with the multi-level parallelism of GPU hardware. As shown on the left of Figure 5b is the hierarchy of the tiled loop nest; on the right is the 2D GPU blocks and threads. The curved arrows represent the binding relations between loop

dimensions and block/thread indexes. The i_b and i_t loops can be mapped to their counterparts safely, since they both are parallel. Due to the reduction dependences, the j_b and j_t loops, however, are originally not allowed to distribute across multiple blocks or threads. Considering the associativity of a reduction, we ignore the reduction dependences, which will be recovered later, and transform j_b and j_t loops into parallel. These two dimensions can now be decomposed into multiple blocks, as the dashed arrows show.

```
/* Tile sizes are 32×4. */
parallel for i_b=0 to M/32
  reduced for j_b=0 to N/4
    parallel for i_t=0 to 32
      reduced for j_t=0 to 4
        m elmwise stmts;
        // marked reduce stmt
        R(i_1, ···, i_p, j_1, ···, j_r);
```

(a) The tiled code.



(b) Hardware binding.

Figure 5: Loop tiling and hardware binding of x-reduce.

The binding strategy is implemented by manipulating the internal representation [25] using its rich set of node types, which was also employed by existing polyhedral tools [54, 58]. We go one step further by mapping the reduced loop dimensions to the inner dimensions (`blockIdx.`x/`threadIdx.`x) of the block/thread parameters. We enforce this binding strategy for two reasons. First, binding a reduced tile loop to the inner block dimension minimizes the amount of the global synchronizations across multiple blocks. Second, such a binding strategy benefits for memory coalescing thanks to our memory access pattern discussed in Section 3.1.

4.3 Orchestration Effects of Transformations

The combination of loop fusion and tiling follows the traditional way used by many existing polyhedral compilers [7, 54, 58], but loop coalescing is no longer computed by the polyhedral schedulers. Instead, performing loop coalescing in an isolated way makes it possible to obtain the three canonical forms in Figure 3, which eases the hardware binding in Section 4.2 without sacrificing the parallelism of other fully parallelizable dimensions. Without these three canonical forms, the dimenisonality of tunable loop dimensions can vary greatly, and an autotuner [65] has to search a much larger space of tile sizes and thread/block configurations.

5 CODE GENERATION AND OPTIMIZATION

To resume the ignored reduction dependences, we attach a special mark to each reduction statement, as the comment before the reduction statement `R` shown in Figure 5a. The attachment of such a mark is also done by manipulating the internal representation [25]. This mark delivers a request to the code generator, which will deal with a reduction statement using a special scheme.

5.1 Code Generation

Code generation in polyhedral compilation is trivial for element-wise operators. The code generator substitutes each tensor index variable with a tiled expression, which is instanced using the built-in `blockIdx/threadIdx` variables according to the mapping relations for hardware binding. The generation of reduction statements is a bit more complicated. For the sake of simplicity, we take the parallel summation operator (*reduce_sum*) as an example to illustrate the code generation of reductions, which is shown in Figure 6.

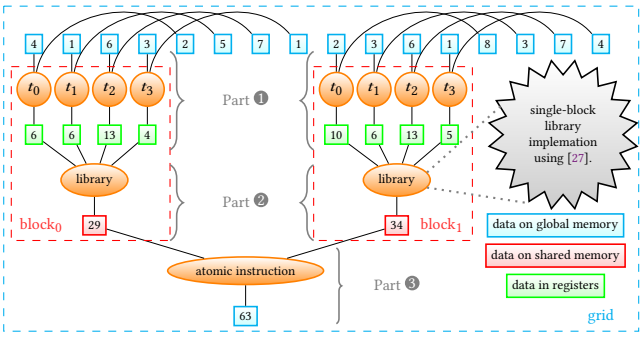


Figure 6: Parallel reductions using atomic instructions.

We suppose that *reduce_sum* is performed over 16 (blue) data elements. We also assume that two (dashed red) blocks are used to execute this reduction, each of which is configured using four (orange) threads, t_0 to t_3. This configuration results in the execution of multiple reductions within each thread. The top curved lines represent the relations between additional data elements and the threads. In our example, each thread produces a local summation of two input elements. This constitutes the first execution part (Part ❶), which automatically implements the "first add during global load" optimization, *i.e.*, the sequential addressing optimization in Figure 1b, of the kernel decomposition approach [27]. Part ❶ also maximizes the opportunity to enable sequential addressing and allow for the fusion of a faster parallel reduction with other operators, which was not considered by highly tuned libraries [11, 27, 41].

Part ❶ computes a (green) local summation using a thread that requires further reductions. These local summations cannot be generated by the polyhedral model since no corresponding statement exists in the internal representation. We introduce an invocation of a parallel reduction routine, Part ❷, with both sequential addressing and loop unrolling exploited by [27] considered, reducing the local summations of each thread to a (red) partial result for each block. Part ❷ guarantees the high performance of the generated code, addressing the ineffectiveness of stand-alone compile-time transformations [14, 66]. The maximum allocatable number of threads per block is limited, but we dispatch more computations to one thread in Part ❶ and execute the local reduction of a thread in parallel with others', minimizing the number of blocks involved.

A reduction over all (red) partial results is finally added automatically following the library invocation. We use Part ❸ to represent this final reduction and leverage the low-level GPU atomic instructions to guarantee the global synchronization. Using atomic instructions always generates a single kernel, but it enforces the sequential updates from the (red) partial results to the final (blue) summation. We thus always try to minimize the number of blocks, which is mitigated by Part ❶, to guarantee the high performance. The benefits of using atomic instructions include the avoidance of multiple kernels [27] and the much better performance over the code generated by *isl* [57].

The above orchestration generates the code depicted in Figure 7. We show the code as a template for different kinds of reductions, with various data types fully supported. OP can be instanced using an associative and commutative binary operator. The supported binary operators in our implementation include *summation* (+), *product* (×), *min, max, logical AND* (&&) and *logical OR* (||). T is used to represent the type of the input data, which can be one of *double, float32, float16, bool, long long int* and *int*, covering all reduction scenarios we have seen so far in the domain of deep neural networks. One can easily complement other reduction and/or data types when necessary, to which our approach is still applicable.

```
__global__ void reduce(int len, T *input, T *output, int num, OP op){
  T local_sum=0;
  __shared__ T shared_buf[4];
  __shared__ T block_sum[1];
  /* Part ❶, automatically generated using polyhedral compilation. */
  for(int k=0; k< num; k++)
    if(threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*num<len)
      op(local_sum,input[threadIdx.x+k*blockDim.x
        +blockIdx.x*blockDim.x*num]);
  __synchthreads();
  /* Part ❷, automatic invocation of library routines. */
  Parallel_Reduce<T,OP,4,all>(op,&block_sum[0],shared_buf,local_sum);
  __synchthreads();
  /* Part ❸, automatic global sychronization using atomics. */
  if(threadIdx.x==0)
    Atomic_Return<T,OP>(block_sum[0],&output[0],op);
}
```

Figure 7: Automatically generated kernel for reduce_sum.

Parallel_Reduce and Atomic_Return are the interfaces to our library and low-level atomic instructions, which will be instanced using a data type T and a reduction operator OP. The third argument of Parallel_Reduce represents the number of threads within each reduced block, and the last argument indicates the reduction pattern (Figure 3) handled by this function.

A further optimization can be applied to the loop of Part ❶, which can rewrite the loop as

```
for(int k=threadIdx.x+blockIdx.x*blockDim.x*num;k<len;k+=blockDim.x)
  op(local_sum,input[k]);
```

and remove the *if* conditional within the kernel. In addition, a larger stride of `blockDim.x*gridDim.x` can also be used to maximize the opportunity to optimize global memory coalescing. Generating loops with non-unitary strides using polyhedral compilation is straightforward, but the internal representation of the polyhedral model has to be lowered to HalideIR [47] to emit CUDA kernel in our DL compiler, which currently does not support such a functionality.

One may notice that **__synchthreads()** has also been introduced at the correct positions in the code to perform synchronizations. The variables that should be allocated on the shared memory have also been declared with the **__shared__** attribute. Like existing polyhedral compilers for GPU [54, 58], the generation of thread synchronizations and the memory promotion to shared memory are both implemented by manipulating the polyhedral representation [25], which is also used to determine which variables are local to a single block and promote it to shared memory/registers. Thread synchronizations can be introduced as long as a Parallel_Reduce or Atomic_Return invocation is generated.

5.2 Generalizing the Library

We now generalize our approach for an irregular input size n. Specifically, we divide the input elements into two groups, with one of size 2^k and the other consisted of the remaining. k should be selected such that 2^k is equal to the greatest power of two among those smaller than n. We then perform a local reduction over the input data to reduce the number of elements to 2^k. An irregular input size is thus transformed into a form eligible for our library, with affordable *if* conditionals used during the added local reductions.

While this optimization is designed for parallel execution within a single block, *i.e.*, for Part ❷ of Figure 6, it can also be used to optimize irregular input sizes across multiple blocks. As we explained in Section 4.2, the polyhedral model performs loop tiling to fit for the GPU memory hierarchy. One cannot always find tile sizes that can divide the input sizes of a reduction operator. Partial tiles, usually with an irregular number of loop iterations, are encountered when tile sizes cannot divide the input sizes. Existing approaches like TC [54] fail to benefit from the CUB library [41] in such cases.

Another difficulty is the limited set of data types supported by atomic instructions of GPU devices with compute capability 8.x and higher. They only support data types of 2 bytes, 4 bytes and 8 bytes [42], making Part ❸ not suitable to handle *logical AND/OR*. Fortunately, data of the *bool* type (1 byte) can always be processed efficiently, no matter when accessed from memories or used by computation. The maximum representable value of this type also implies that the input size will not be too large. As a result, a single block is sufficient to handle reductions of this type.

5.3 Handling Multiple Reductions

We now discuss the handling of multiple reductions. As explained in Section 3, PANAMERA only allows the fusion of a reduction with its preceding elementwise operators. An ideal scenario that takes place frequently in DL applications is composed of multiple independent reduction operators, which may share one or multiple elementwise operators. One can still fuse multiple reduction operators when the numbers of their enclosing loop nests are identical and they are performing reductions along the same set of loop dimensions. PANAMERA is still applicable in this case by embedding one library invocation for each reduction. Figure 8 is an example of such cases, where all templated objects haven been instantiated.

However, one cannot obtain a single kernel when different sets of reduced dimensions of multiple independent reduction operators do not match exactly. This happens when given a sub-graph composed of multiple dependent reduction operators due to the failure of fusion between them. Our solution is to feedback to the upstream graph engine of AKG [63] to decompose such a sub-graph into smaller ones such that the prerequisite of our approach can be satisfied, and our approach can work as normal. In summary, our library is implemented in more than 1,000 lines of CUDA C++ code, with various data types, diverse reduction scenarios and different interfaces all supported. Debugging and testing of the approach cost roughly one month.

6 DISCUSSIONS

We now discuss the potentials and limitations of PANAMERA.

6.1 Generality and Potentials

As explained before, the data of DL reductions is usually organized in a deeper nested manner than the available hardware parallelism

```
__global__ void reduce(float *input0, float *input1, float *input2,
                                                  float *output0, float *output1){
  float local_sum=0; float local_max=-3.40282e+38f;
  __shared__ float shared_buf[128]; __shared__ float block_sum[1];
  __shared__ float block_max[1];
  /* Fuse the addition operator with reduce_sum. */
  for(int k=0; k< 8; k++)
    if(threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*8<1024){
      float agg_local = input0[threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*8]
        + input1[threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*8];
      Sum(local_sum, agg_local);
    }
  __synchthreads();
  Parallel_Reduce<float,Sum,128,all>(Sum,&block_sum[0],shared_buf,local_sum);
  __synchthreads();
  if(threadIdx.x==0)
    output0[0] = block_sum[0];
  __synchthreads();
  /* Fuse two reductions through identical hardware configuration. */
  for(int k=0; k< 17; k++)
    if(threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*17 < 2176)
      Max(local_max, input2[threadIdx.x+k*blockDim.x+blockIdx.x*blockDim.x*17]);
  __synchthreads();
  Parallel_Reduce<float,Max,128,all>(Max,&block_max[0],shared_buf,local_max);
  __synchthreads();
  if(threadIdx.x==0)
    output1[0] = block_max[0];
}
```

Figure 8: A code example of a fused operator that is composed of one addition and two reductions. It first sums `input0` and `input1`, both of which are 1D tensors of size 1024, and outputs `output0` through a *reduce_sum*. Another 1D tensor `input2` of size 2176 is reduced (*reduced_max*) to `output2`.

dimensions but the data size along each loop is relatively smaller, resulting in the ineffective use of GPU threads. On the one hand, dimension flattening can be used to address this issue but it misses the fusion opportunities. On the other hand, performing loop fusion without the help of loop coalescing fails to maximize the utilization of hardware resources. Existing techniques thus cannot model the conflicting demands between GPU hardware parallelism enabled by loop coalescing and the optimization of memory hierarchy exploited by loop fusion, though each of them was commonly used before. Panamera resolves this trade-off for DL reductions by orchestrating these techniques systematically to achieve their best composition.

We use the functionality of *isl* to compute new schedules and perform fusion and tiling, but we carefully manipulate the internal representation of *isl* to realize the appropriate hardware binding. Dimension flattening and library embedding are implemented by ourselves. Note that loop tiling and coalescing are expressible using *affine relations* [4, 34] in the polyhedral model, but the state-of-practice polyhedral scheduling algorithms [7, 16, 59] are only able to compute *affine functions* that do not expand or collapse loop nest dimensions. Loop transformations like tiling and fusion falling into this category are thus performed in an isolated manner. For instance, Pluto [7] and PPCG [58] focus on enabling tiling first and then apply the transformation as a post scheduling pass. Panamera is the first work that uses loop coalescing as pre-processing before scheduling for DL reductions. Also note that the idea presented in this paper was not restricted to polyhedral compilation: as loop tiling for reductions and their fusion with elementwise operators after our preparation are always legal, one can easily integrate our approach into other tensor compilers like TVM.

We did not set a threshold on the number of fused operators. Instead, the criterion to make fusion decisions is determined by avaliable hardware resources. The reduction operators are given with the higher priority when faster memory are (close to) saturation in the case of aggressive fusion.

Our approach harnesses the domain-specific properties of DL applications, but it is not only applicable to deep neural networks. All optimizations related to the domain-specific properties of DL models can be turned off to generalize the approach to more application domains. We always deal with Figure 3a using one GPU, since DL reductions usually possess smaller data sizes than those of high-performance computing, which rarely exceed the handling power of a single GPU. For Figure 3b and Figure 3c, we evenly decompose the parallel for loop to multiple GPUs and let each generate one kernel to avoid synchronizations between them.

Panamera is also applicable to matrix multiplication, since it can also be considered as a reduction operator. Nonetheless, we do not encourage to optimize matrix multiplication using Panamera in most cases, since many DL compilers have more sophisticated optimization strategies for such operators by fully incorporating with specific hardware support.

For example, AKG has its specific strategy to optimize matrix multiplication or map it to tensor cores using the same approach presented in [6]. We compare the performance of Panamera and the specific handling of matrix multiplication in AKG in Table 1. The performance of Panamera is lower than the special handling backed by tensor cores. However, Panamera can also surpass the later by 2.71× when the reduced dimension is very large and parallel dimensions are small (a strange shape that rarely but probably happen in practice), where the atomic instructions are not the performance bottleneck.

Table 1: Performance comparison of matrix multiplication when optimized using Panamera and tensor cores in AKG. We report execution time in microseconds.

MNK shape	*K*-dim config	Panamera	tensor cores	matching percent
128 × 32 × 64	2 blocks	24.044	4.381	18.22%
128 × 32 × 1024	16 blocks	21.378	57.882	270.75%
1024 × 512 × 1024	16 blocks	183.18	78.623	42.92%

6.2 Limitations

Panamera currently suffers from some limitations. First, using atomic instructions probably results in the non-determinism issue. Addressing this problem using compilation techniques [39, 43] is possible, but these methods may miss the specific features of atomic instructions. We believe the recent hardware scheme for deterministic atomic buffering [12] is the best solution. Using atomic instructions prevents Panamera from being extend to associative but non-communicative reductions, but it is the price to pay for using such hardware primitives. Second, mapping to the templated routines during code generation still needs manual configurations, making Panamera not fully automated. We believe automating the generation of these routines is possible but calls for much effort due to the diversity of different fused scenarios. Similar innermost optimization can also be automated, like the automatic generation of the innermost kernel for general matrix multiplication [52]. We leave addressing these two limitations as our future work.

7 EXPERIMENTAL RESULTS

Panamera is implemented in AKG [64] that takes as input a DL model and generates CUDA code for GPU, with a templated C++ interface to *isl*-0.21 [57] used for polyhedral transformations. The code repository is available at https://gitee.com/mindspore/akg. AKG was used for NPUs but it can also generate CUDA code. A DL model is first converted into sub-graphs, each of which is transformed into tensor computations by the DSL of TVM. Panamera lowers a tensor program to generate CUDA code for an NVIDIA Tesla V100 GPU. The CUDA code is compiled using the *nvcc* compiler version 10.1 with the *-O3* flag enabled for each program.

We conduct experiments on single operators, sub-graphs and end-to-end workloads. The CUDA code generated by TVM (v0.6) [9], Ansor [65], cuDNN (v7.6.4) and CUB (v1.8) [11] are considered for comparison. A single operator is written using the DSL of TVM, which can be taken as input by Panamera, TVM and Ansor. The code variants and the GPU hardware configurations are optimized using the auto-tuners of each approach, with both the optimal tile sizes and the GPU grid/block parameters fully tuned. We pass the appropriate arguments to the interfaces of cuDNN and CUB such that they can be used in the experiment. The geometric mean of 10 executions is reported to minimize the effect of performance noise. The optimal tile sizes used for each code evaluated in the experiments are fully tuned by their own auto-tuners.

7.1 Results of Single Operators

We use three reduction operators including *reduce_sum* (summation), *reduce_max* (maximum) and *reduce_and* (logical *AND*) to evaluate the scalability. The results of *reduce_mul* (product), *reduce_min* (minimum) and *reduce_or* (logical *OR*), follow similar trends as the three operators considered here, respectively; we will thus not show their results. We consider two factors, input tensor configurations and data types, in this experiment. The data types used in this evaluation include *float*32, *float*16, *int* and *bool*. The results of the *double* and *long long int* are similar to those of *float*32 and *int*. The original tensor shape configurations are listed below the bar charts of *reduce_sum*, and the flattened shape configurations are shown below each plot of *reduce_max* with each reduced dimension underlined. Figure 9 and 10 show the comparison of the execution times. We use the flattened shape configurations for explanation.

When given *float*32 and *int* types, TVM performs poorly with larger input sizes, especially in the *all*-reduce scenarios, due to its ineffective hardware binding strategy. As it can also work with dimension flattening introduced in Section 3, TVM performs better under *x*- and *y*-reduce shape configurations but still falls behind Panamera due to the improper use of GPU blocks.

Panamera outperforms TVM slightly when given *float*16 data, because we did not perform reductions over too many elements. The maximum representable value of IEEE 754 half-precision floating-point numbers is 65504. A greater size of *float*16 numbers may lead to an overflow error of the partial or final reduction results. Considering the insufficient numbers of input elements, we did not span the reduction to multiple blocks. The performance improvement comes from our library. Similarly, one cannot specify a much larger shape configuration to *reduce_and*, which takes as input *bool* data. In summary, Panamera obtains a mean speedup of 5.4× over TVM. Note that reductions over *float*16 values are not supported by TVM, which has to first convert the input into *float*32 and then transform the result back to *float*16 to allow for the comparison.

We also report the execution times (the violet bars) of Panamera with only the multi-block functionality disabled. This version performs similarly to Panamera when given smaller shape configurations, but it suffers from severe degradation under larger and/or complex shape configurations where the block-level parallelism is crucial to performance improvement. The performance difference between this version and TVM is due to the different tuned tile sizes. The purpose of this experiment is to isolate the effects of Part ❷ and ❸, which have to be used together for correctness. We did not evaluate the effect of Part ❶, since it has been studied in [18, 27].

Ansor advanced TVM by automatically generating schedule templates using a sampling strategy, but it did not optimize TVM's single-block parallelism. Due to the randomness of the sampling strategy, its performance follows the similar trend as TVM's by sometimes outperforming and sometimes falling behind the later. Ansor's tuner searches towards a direction which it supposes can find a better schedule, but it quits with a failure information thrown out under the last *y*-reduce configuration, where its tuner cannot find better solutions. Panamera outperforms Ansor by 9.6×.

cuDNN exhibits the worst scalability among all approaches to both factors we considered. First, it seems that cuDNN does not support reduction operators over integer numbers, since it throws out a `Bad_Parameter` error when handling reductions of integer numbers. The execution times of the *int* type are thus missing in Figure 9 and 10. Second, when given floating-point numbers, cuDNN scales well under the *all*-reduce cases, but its performance declines severely under an *x*- or *y*-reduce configuration.

It is hard to exactly explain the reasons why cuDNN suffers from such degradation, since its algorithmic implementation is not publicly accessible. Based on a profiling analysis, we guess the possible reasons may be as follows. First, it is likely that cuDNN dose not perform loop coalescing, which results in the ineffective hardware binding between parallel/reduced dimensions and GPU blocks in the case of nested reductions over multiple variables. Second, it seems like an identical 3D thread configuration «8,16,1» within each block is used by default by this library, though multiple blocks are allowed. Finally, it might not consider the pattern shown in Figure 2d when given a tensor shape configuration. Conversely, these optimization strategies have all been integrated into our approach, leading to a mean speedup of 33.7× over cuDNN.

We also collect the data of CUB [41]. CUB does not support reductions over *float*16 data. We thus report the results of this type by performing the same type conversion approach as what we did to TVM, *i.e.*, first converting the input data type into *float*32 and then changing the result back to *float*16. CUB falls behind our approach due to the overhead of data type conversion, though it performs similarly to Panamera when given *float*32 or *int* data.

CUB also has many restrictions. Similar to cuDNN, CUB cannot handle the non-continuous reduction dimensions like Figure 2d. We feed the flattened shape configurations to this library. In addition, it can only take as input reduction operators whose reduced dimensions are along the inner loops, which requires an auxiliary transpose operator to permute the reduced dimensions to inner

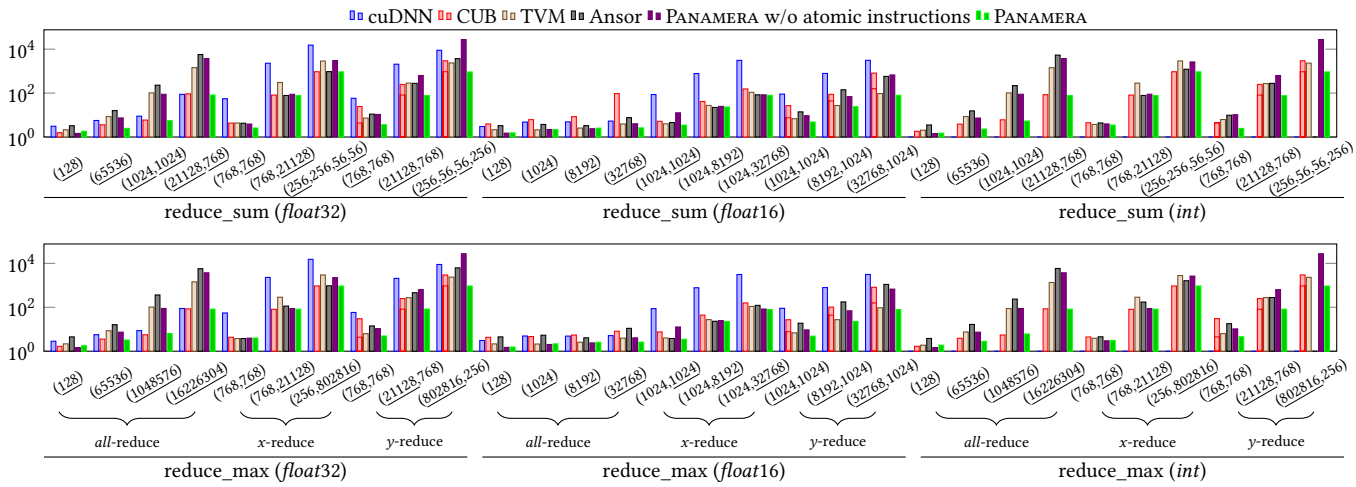


Figure 9: Execution times of a single reduction operator under different data types (*reduce_sum* *x* axis: original shape configurations; *reduce_max* *x* axis: flattened shape configurations; *y* axis: log scaled execution time in μs; lower is better).

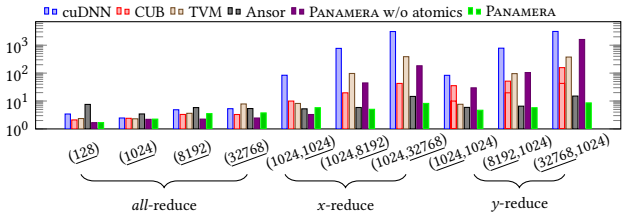


Figure 10: Execution times of *reduce_and* (*x* axis: shape configurations; *y* axis: log scaled time in μs; lower is better). The original and flattened shape configurations are identical.

positions when given a *y*-reduce case. This auxiliary transpose operator introduces performance penalty, which is represented by the upper part of each stacked (red) bar under the *y*-reduce configurations, and the execution time of CUB is always longer than that of PANAMERA under *y*-reduce inputs. On average, PANAMERA surpasses CUB by 3.5× and outperforms the latter by 2.1× when the performance penalty of a transpose operator is not considered.

7.2 Results of Sub-graphs

We collect 12 sub-graphs obtained from the high-level graph engine [63] and compare the performance with TVM, Ansor and cuDNN. Table 2 summarizes these sub-graphs, with each input configuration denoted by the data type followed the tensor shape in a bracket and reduced dimensions underlined. A sub-graph is composed of two to six elementwise and reduction operators, and each of its branches is terminated by a reduction operator. These sub-graphs can be taken as input by TVM [9], Ansor [65] and our approach. Figure 11 shows the execution times of each approach. The result of cuDNN is missing in some cases due to the failure of supporting *type casting* operators and *shape reshaping* operators. This illustrates that cuDNN is rarely scaling with divergent elementwise operators. PANAMERA produces an average 9.5× speedup over cuDNN due to the optimization on reductions and the saved memory access latency thanks to loop fusion.

Table 2: Summary of Sub-graphs. *f* for *float*; cast16 converts an *f*32 tensor into *f*16 and cast32 performs the reverse process; r_sum represents the reduce_sum operator.

no.	input config.	op_1	op_2	op_3	op_4	op_5	op_6
1	*f*32 [64,2]	cast16	cast32	cast16	r_sum	-	-
2	*f*32 [1280,21128]	cast16	r_sum	-	-	-	-
3	*f*16 [64,768]	cast32	r_sum	-	-	-	-
4	*f*32 [1280,21128]	mul	r_sum	-	-	-	-
5	*f*32 [1280]	neg	mul	r_sum	-	-	-
6	*f*32 [3072]	mul	mul	r_sum	-	-	-
7	*f*32 [64,128,768]	add	mul	_sum	-	-	-
8	*f*32 [64,128,768]	add	mul	r_sum	add	mul	r_sum
9	*f*32 [8192,768]	r_sum	r_sum	-	-	-	-
10	*f*16 [64,128,12,64]	reshape	cast32	r_sum	-	-	-
11	*f*16 [64,128,768]	reshape	cast32	r_sum	-	-	-
12	*f*16 [64,20]	reshape	r_sum	-	-	-	-

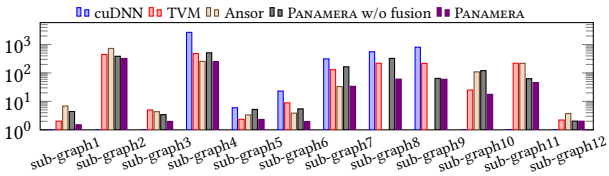


Figure 11: Execution times of sub-graphs (*y* axis: log scaled execution time in μs; lower is better).

The manual fusion of TVM also benefits from the faster local memories of GPU and obtains comparable performance to that of PANAMERA under *all*-reduce cases like sub-graphs 5 and 12, but it underperforms when the benefit of multi-block parallelism (sub-graphs 4, 6, 10 and 11) or reduction propagation (sub-graphs 1, 2, 3 and 7) is significant. In addition, the interaction with the upstream graph engine allows for the aggressive fusion of reduction operators

(sub-graphs 8 and 9) where the two reductions on different branches do not depend on each other but perform reductions along the same set of loop dimensions. Such information is committed to the upstream graph engine [63], which will then combine these two reductions. This scenario is not considered by TVM. Our approach brings about a mean speedup of 2.6× over TVM.

Ansor's performance is inferior when the reduced dimensions are relatively smaller (sub-graphs 1, 2, 5 and 12) or superior to TVM in other cases (sub-graphs 4 and 6). This approach fails to handle sub-graphs 8 and 9. Panamera outperforms Ansor by 2.7× on average. The performance gap between Panamera and these baselines may be further widened when given larger shape configurations.

The dark gray bars represent the execution times of Panamera with loop fusion disabled. The benefit of loop fusion is lightweight when given an *all*-reduce case and the number of operators is small (sub-graph 12). This version also performs similarly to Panamera for sub-graph 9, because the fusion between two independent reduction operators is exploited by the graph engine, which is enabled in both versions. The effect of loop fusion is notable in other cases.

7.3 Results of End-to-end Workloads

Panamera was integrated into AKG, which performs a large number of domain-specific optimizations for convolutions and (batched) matrix multiplications, without which the deployment of a DL model onto GPU would be impossible. We first compare the execution times of the codes generated by AKG with and without our approach, which is used to illustrate how much improvement Panamera can bring about to a tensor compiler.

We next compare the performance with MindSpore [32], TVM and Ansor. MindSpore is a DL framework backed by cuDNN/cuBLAS. The purpose is to demonstrate that Panamera can help AKG exceed other approaches. The optimizations for convolutions, (batched) matrix multiplications, *etc.* are also enabled by TVM and cuBLAS [40]. We consider BERT [13], Wide&Deep [10], VGG-16 [50], MobileNet-v3 [31], Transformer-large [55] and GPT-3 [8] in this experiment, with each expressed using MindSpore. The model configurations can be retrieved from the model zoo of MindSpore at https://gitee.com/mindspore/models.

BERT [13] is composed of 110×10^6 parameters and used for natural language processing. It is also one of the models in MLPerf [48]. The dataset of this model is composed of 4000 words and we use mixed precision to experiment the workload. Wide&Deep [10] is a model for recommendation system and click predication area. Its dataset is extracted from [26] which include 9.56 GB data. VGG [50] is also extracted from MLPerf [48] and used for large-scale image recognition. MobileNet [31] takes as input images from the same set of dataset of VGG and performs a combination of hardware-aware network architecture search. Transformer-large [55] is designed for natural language processing, which we instantiate using the WMT English-to-German translation task, with mixed-precision enabled. Similarly, GPT-3 [8] is an auto-regressive language model created by OpenAI, for which we use the openwebtext dataset. Each end-to-end workload is expressed using the MindSpore framework. Note that the data of the GPT-3 model is collected on an NVIDIA Tesla A100 GPU due to the limited time, and this hardware is used by each code version of this model in this experiment.

Table 3 reports execution time in milliseconds. The rightmost column records the number of operators fused by DL reductions that enabled by Panamera. The preceding column of the rightmost reports the improvement (21.2% on average) of Panamera over AKG. Our approach always improves the performance of AKG due to the compound effect of faster parallel reductions and loop fusion. The profiling results shown in Figure 9, Figure 10 and Figure 11 also apply to these end-to-end workloads. The next preceding three columns list the improvements of AKG integrated with our approach over MindSpore backed by CUDA libraries, TVM and Ansor, respectively. The libraries perform worst because cuDNN/cuBLAS does not consider fusion across network layers. Integrating Panamera into AKG produces a mean improvement of 122.5% over the library routines. AKG itself is usually competitive to or falls behind TVM and Ansor, but its performance exceeds the later two by 19.3% and 15.2% on average thanks to our approach.

Table 3: Results of end-to-end workloads.

Workloads	MindSpore	TVM	Ansor	AKG	Panamera	Improvement over				number of fused ops
						MindSpore	TVM	Ansor	AKG	
BERT	352.2	138.0	120.3	124.0	111.0	+217%	+24%	+8%	**+12%**	304
Wide&Deep	22.4	12.5	12.8	12.6	11.0	+104%	+14%	+16%	**+15%**	74
VGG	70.4	65.7	66.3	67.6	64.2	+10%	+2%	+3%	**+5%**	39
MobileNet	151.4	133.0	129.4	136.8	131.5	+15%	+1%	-2%	**+4%**	52
Transformer	157.8	132.4	126.5	136.8	79.2	+99%	+67%	+60%	**+73%**	746
GPT-3	483.0	133.9	131.3	146.2	123.7	+290%	+8%	+6%	**+18%**	409
average						**+122.5%**	**+19.3%**	**+15.2%**	**+21.2%**	

Note that the performance improvements are over the optimized code generated by AKG, which has highly optimized matrix multiplication and convolution operators that consume most (usually 50%-90% or more) of the execution time of an end-to-end workload. As such, the performance improvements of Panamera seem modest for VGG and MobileNet. Let us assume this portion be 90%, and this part does not contribute to the improvements of Panamera over AKG, because they both parallelize these operators. Suppose the remaining 10% be composed of only reduction operators. The theoretical speedup brought by Panamera over AKG is $\frac{1}{0.9+\lim\limits_{x\to\infty}(0.1/x)} = \frac{1}{0.9} = 1.11$, where we presume the speedup achieved by Panamera for reductions is x. In practice, the portion of reduction operators may be smaller because there also exist many other kinds of operators. Hence, the results on these two workloads are not insignificant. For other workloads, Panamera can achieve favorable improvements (up to 1.73× for Transformer-large).

7.4 Compilation Overhead

We now discuss the compilation overhead of our approach. We collect the compilation time for each single operator. The results show that Panamera does not introduce too heavy overhead (1.6-2.1× slower) compared to TVM, the compilation time of which for each single reduction operator is ranging from 0.35 to 0.45 seconds. Such a lightweight cost does not aggravate the compilation overhead when experimenting with sub-graphs and end-to-end networks. One of the reasons of this lightweight compilation overhead is the isolation of loop coalescing from the polyhedral model. Besides, reduction propagation guarantees the matching between the loop dimensions of a reduction operator and elementwise operators,

simplifying the polyhedral fusion heuristic and thus mitigating the polyhedral scheduling time.

8 RELATED WORK

Vendor libraries [5, 11, 41] are a common approach to parallelize reductions on GPU. We showed in our experiments that a carefully designed implementation by considering multiple metrics described in Section 5.1 can achieve better performance than cuDNN and CUB for many scenarios. Unlike our solution, other approaches rely on kernel decomposition [27] to optimize the parallel reductions on GPU, which is impractical for DL reductions as discussed in Section 2.

Cooperative groups (CGs) [28] are used to group threads and can synchronize block-level reductions. We used CGs in the early stages of Panamera. CGs achieved competitive execution performance to atomic instructions in many cases, but we finally use the current approach due to two reasons. First, handling reductions using CGs provides is less flexible than our current solution: one has to always guarantee the perfect matching between the configurations of thread blocks, threads and SMs; an error indicating "too many blocks in cooperative launch" is otherwise thrown. Our current solution does not have such a limitation. Second, CGs also restrict the number of launched thread blocks within an SM, which makes the performance of CGs' generated code inferior to our current solution when handling reductions with larger input data sizes. We summarize some of such cases in Table 4.

Table 4: Input sizes of *reduce_sum* (*float*32) for which the performance of CGs falls behind that of Panamera. Execution time is reported in milliseconds.

shape configuration	cooperative groups	Panamera		improvement over cooperative groups
		w/o atomic	w. atomic	
(16226304)	93.70	1530.80	78.04	20.07%
(1024,131072)	626.46	624.51	601.17	4.21%
(131072,1024)	657.45	607.85	609.96	7.79%
(1048576,512)	2534.8	2446.8	2424.6	4.55%

Compilation approaches exploit the combined effect of reductions and other operators; they usually boil down to three stages. Some of them [20, 66] focused on the detection of reduction dependences, some studied the scheduling of reductions [14], and some [49, 51, 56] used the associativity and commutativity of a reduction to study its parallelism. None of these techniques consider the domain-specific properties of DL models or the GPU atomic instructions. Whether Reduction Drawing [49] uses atomic instructions was not explicitly described. It seems their work still uses kernel decomposition [27] when multiple thread blocks are involved. The comprehensive study [14] on much earlier parallel reductions showed that most of much earlier methods suffered from similar limitations to the compilation approaches discussed here.

Recent compilation frameworks [3, 46, 54] for deep neural networks take into account the domain-specific knowledge of DL models. TC [54] is also integrated with CUB [41] to enhance the performance of the generated code. However, the reduction scenarios covered by TC is a subset of those handled by our approach. The imperfect handling of partial tiles in TC also makes their performance inferior to our technique. Futhark is an optimizing compiler for a functional, array programming language. Similar to Panamera, it studies parallel reduction for GPU by supporting fusion between an elementwise-like producer and reduction consumer [30] and also between independent reductions [29]. Its parallelization strategies [36] for the latter two canonical forms in Figure 3 can be evaluated by a tensor compiler's autotuner, which helps Panamera select the best-performing grid/block configurations. However, Futhark requires an expensive transposition operation when dealing with the y-reduce scenarios, which are well addressed in this work.

Compilation approaches can also simplify the algorithmic complexity of reductions [19] by reusing the intermediate results computed during reductions. This idea is adopted by AlphaZ [61] and extended to handle dependent reductions [60]. Panamera differs from these approaches by making use of the commutativity of reductions and GPU hardware resources rather than optimizing the algorithmic complexity. In particular, Panamera is also applicable to dependent reductions by performing reductions using a single thread block. Each additional statement that introduces a backward dependence to the reduction statements is executed by one block.

Language specifications and extensions excel at providing domain-specific knowledge to compilers. Representative DSLs include Halide [47] for image processing and TVM [9] for DL models. Halide's extension [53] allows for the refactoring of reductions, with the transformations still managed by hand. Another work [21] that supports Halide generalizes various types of operators including reductions and integrates its compilation flow with cuBLAS [40], but no specialized libraries for reductions were considered. Atomic instructions was also integrated into high-level kernel synthesis frameworks [22], with loop transformations not considered. As our work demonstrated, exploiting the reuse of intermediate variables created by loop fusion is essential to improve the performance.

9 CONCLUSION

We studied parallel reductions on GPU and proposed a combined library and polyhedral approach to optimize such programs for deep neural networks. By fully considering the domain-specific properties of DL models, we implemented loop coalescing as a preprocessing optimization and propagated the reduction dependences. These preparations allow us to focus on three canonical forms of reductions, which are then delivered to the polyhedral model for exploiting loop fusion and tiling. With the well-designed hardware binding strategy in polyhedral compilation, the code generator is able to automatically produce high-performance programs by wrapping a highly tuned library and embedding low-level atomic instructions. The results demonstrated that a careful orchestration of well-known techniques can achieve better performance than the state of the art. We will address the automatic generation of the highly optimized routine and determinism issues in the future.

ACKNOWLEDGMENTS

We feel thankful for the constructive comments of the anonymous reviewers that improve the quality of this paper. Jie Zhao's work is partially supported by the National Natural Science Foundation of China under Grant No. U20A20226.

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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987), 491–542. https://doi.org/10.1145/29873.29875

[3] 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 *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[4] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO'16)*. Association for Computing Machinery, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[5] Nathan Bell and Jared Hoberock. 2012. Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

[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, 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

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (Boston, MA, USA) *(DLRS 2016)*. Association for Computing Machinery, New York, NY, USA, 7–10. https://doi.org/10.1145/2988450.2988454

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE]

[12] Yuan Hsi Chou, Christopher Ng, Shaylin Cattell, Jeremy Intan, Matthew D. Sinclair, Joseph Devietti, Timothy G. Rogers, and Tor M. Aamodt. 2020. Deterministic Atomic Buffering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*. 981–995. https://doi.org/10.1109/MICRO50266.2020.00083

[13] 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

[14] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions. In *5th International Workshop on Polyhedral Compilation Techniques* (Amsterdam, The Netherlands) *(IMPACT 2015)*. 11 pages.

[15] 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

[16] 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.

[17] 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

[18] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: EXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC'19)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/3307681.3326606

[19] Gautam and Sanjay Rajopadhye. 2006. Simplifying Reductions. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL'06)*. Association for Computing Machinery, New York, NY, USA, 30–41. https://doi.org/10.1145/1111037.1111041

[20] Philip Ginsbach and Michael F. P. O'Boyle. 2017. Discovery and exploitation of general reductions: A constraint based approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 269–280. https://doi.org/10.1109/CGO.2017.7863746

[21] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS'18)*. Association for Computing Machinery, New York, NY, USA, 139–153. https://doi.org/10.1145/3173162.3173182

[22] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 73–84. https://doi.org/10.1109/CGO.2019.8661187

[23] Google. 2017. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[24] Tobias Grosser. 2014. *A Decoupled Approach to High-level Loop Optimization: Tile shapes, Polyhedral Building Blocks and Low-level Compilers*. Ph.D. Dissertation. Université Pierre et Marie Curie-Paris VI.

[25] 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

[26] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine Based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 1725–1731.

[27] Mark Harris. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 1–39.

[28] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups

[29] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Santa Barbara, CA, USA) *(ARRAY 2016)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/2935323.2935326

[30] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 Graph-Reduction Approach to Fusion. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC'13)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2502323.2502328

[31] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 1314–1324. https://doi.org/10.1109/ICCV.2019.00140

[32] Huawei. 2020. MindSpore. https://www.mindspore.cn/en

[33] François Irigoin and Rémi 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

[34] Wayne Kelly and William Pugh. 1995. *A Unifying Framework for Iteration Reordering Transformations*. Technical Report. USA.

[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] Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. Association for Computing Machinery, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

[37] Justin Luitjens. 2014. Faster Parallel Reductions on Kepler. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler

[38] 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

[39] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. 2019. Lazy Determinism for Faster Deterministic Multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 879–891. https://doi.org/10.1145/3297858.3304047

[40] Nvidia. 2013. cuBLAS. https://developer.nvidia.com/cublas

[41] Nvidia. 2018. CUB Documentation. https://nvlabs.github.io/cub/

[42] Nvidia. 2020. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[43] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/1508244.1508256

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

[45] Constantine D Polychronopoulos. 1987. Loop coalescing: A Compiler Transformation for Parallel machines. In *1987 16th International Conference on Parallel Processing (ICPP 1987)*. 235–242.

[46] Benoît Pradelle, Benoît Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2019. Polyhedral Optimization of TensorFlow Computation Graphs. In *Programming and Performance Visualization Tools*, Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz (Eds.). Springer International Publishing, Cham, 74–89.

[47] 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

[48] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[49] Chandan Reddy, Michael Kruse, and Albert Cohen. 2016. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/2967938.2967950

[50] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[51] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A Framework for Enhancing Data Reuse via Associative Reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI'14)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2594291.2594342

[52] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic generation of fast BLAS3-GEMM: A portable compiler approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 122–133. https://doi.org/10.1109/CGO.2017.7863734

[53] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel associative reductions in Halide. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 281–291. https://doi.org/10.1109/CGO.2017.7863747

[54] 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

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[56] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-Affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO'14)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/2544137.2544141

[57] 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

[58] 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

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

[60] Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying Dependent Reductions in the Polyhedral Model. *Proc. ACM Program. Lang.* 5, POPL, Article 20 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434301

[61] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Proceedings of the 2012 International Workshop on Languages and Compilers for Parallel Computing (LCPC 2012)*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-642-37658-0_2

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

[63] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. In *Proceedings of Machine Learning and Systems*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.), Vol. 4. 1–19.

[64] 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106

[65] 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

[66] Yun Zou and Sanjay Rajopadhye. 2012. Scan Detection and Parallelization in "Inherently Sequential" Nested Loop Programs. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO'12)*. Association for Computing Machinery, New York, NY, USA, 74–83. https://doi.org/10.1145/2259016.2259027

A ARTIFACT

This is the artifact description of the paper #78 entitled "Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically". We offer the description for reproducing the results of single operators and sub-graphs due to the following two considerations. First, the result of an end-to-end workload is the sum of its sub-graphs, so we believe reproducing the results of sub-graphs is sufficient. Second, all of the workloads used in the paper are written using MindSpore [32], which is a DL framework developed by Huawei. The artifact reviewers may be more familiar with TensorFlow [1], Pytorch [44] or other popular frameworks but not MindSpore. Reproducing the results of end-to-end workloads requires the artifact reviewers first to practice with MindSpore and next play with its eco-system software, *e.g.*, its autotuner, to obtain the best results. Hence, fully reproducing the end-to-end results may take much longer time and heavier engineering effort. Of course, we can provide the reproducing steps to those who are interested in the end-to-end experiments, and they can let us know if they wish, although this would be a non-trivial task.

A.1 Preparation

We opened a repository for this artifact evaluation at https://gitee.com/yaozhujia/panamera-artifact. To clone the artifact materials, `git` should already been installed. We use the version 2.17. The data sets and examples used in the paper can be obtained from this repository. The mandatory hardware is an NVIDIA Tesla V100 GPU. The operating system we used is Ubuntu 16.04 LTS. One can also try on other Linux distributions. The code was compiled by NVIDIA CUDA Toolkit version 10.1 when we wrote the paper, but 11.1 is also acceptable. Once installed, the Profiler of the CUDA Toolkit can be used to reproduce the results. Python 3.7.5 or higher versions are required, with the package manager `pip` installed. The environment is summarized in Table 5.

Table 5: Experiment environment.

Hardware	An NIVDIA Tesla V100 GPU
Operating System	Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64) or higher
CUDA Toolkits	version 10.1 or 11.1
Python (`pip` installed)	version 3.7.5 or higher
`git`	version 2.7 or higher

Once the above requirements have been met, the artifact reviewers can fetch the artifact materials using

```
$ git clone git@gitee.com:yaozhujia/panamera-artifact.git
```

Listing 1: Cloning the repository of artifact materials.

In the following context, a command starting with $ can be executed in the Unix terminal, and # denotes a comment. "/path-to-panamera-artifact/" must be replaced by an artifact reviewer's environmental location. For example, one can replace it using "/home/jack/Desktop" if the user name is "jack" and he/she clones the artifact materials into the Desktop directory of his/her computer. All materials for reproducing the results are put into the directory named "reproduction". We describe the artifact in a tool-by-tool way.

A.2 Reproducing the Results of PANAMERA

A.2.1 Installation. To reproduce the results of PANAMERA, one first has to install AKG [64], into which PANAMERA has been integrated, using the provided Python wheels, depending on the CUDA Toolkit versions. For example, if the Toolkit version is 10.1, the artifact reviewers can use the following commands to install AKG.

```
$ cd /path-to-panamera-artifact/install/cuda10
$ pip install akg-1.2.0-cp37-cp37m-linux_x86_64.whl
```

Listing 2: Installing the AKG compiler.

A.2.2 Execution. One can now reproduce the results of PANAMERA. The artifact reviewers can change into the directory named "Panamera" and follow the instructions of README. The commands for reproducing the results of PANAMERA are as follows.

```
$ cd /path-to-panamera-artifact/reproduction/Panamera
# First, one can reproduce the results of single operators
# 1.1 reproduce the results of reduce_sum
$ nvprof python reduce_sum.py
# 1.2 reproduce the results of reduce_max
$ nvprof python reduce_max.py
# 1.3 reproduce the results of reduce_and
$ nvprof python reduce_and.py
# Second, one can reproduce the results of sub-graphs or "composite cases"
# 2.1 eproduce the results of composite cases
$ nvprof python test_composite_info.py -af ./composite/1.info
$ nvprof python test_composite_info.py -af ./composite/2.info
$ nvprof python test_composite_info.py -af ./composite/3.info
$ nvprof python test_composite_info.py -af ./composite/4.info
$ nvprof python test_composite_info.py -af ./composite/5.info
$ nvprof python test_composite_info.py -af ./composite/6.info
$ nvprof python test_composite_info.py -af ./composite/7.info
$ nvprof python test_composite_info.py -af ./composite/8.info
$ nvprof python test_composite_info.py -af ./composite/9.info
$ nvprof python test_composite_info.py -af ./composite/10.info
$ nvprof python test_composite_info.py -af ./composite/11.info
$ nvprof python test_composite_info.py -af ./composite/12.info
```

Listing 3: Commands for reproducing PANAMERA's results.

Listing 3 can reproduce the results of all single operators and the 12 sub-graphs in Table 2 of the paper. For single operators, the default data type is *float*32 and the default shape configuration is [1024,1024]. To change the data type, one can open, *e.g.*, the `reduce_sum.py` file and change the parameters (line 77 for shape and line 78 for data type) in the main function at the end of the file.

A.2.3 Profiling Results. During the execution, AKG outputs the results to `stdout`, which is similar to "gpu(0): exec = xxx ms/op".

A.3 Reproducing the Results of TVM

A.3.1 Installation. AKG was developed based on TVM version 0.6 [9]. Hence, the TVM version 0.6 has already been installed when installing the Python wheels file.

A.3.2 Execution. The artifact reviewers can change into the directory named "tvm" and follow the instructions of the README file. The commands for reproducing the results of PANAMERA are the same as Listing 3 except that the first command is replaced by

```
$ cd /path-to-panamera-artifact/reproduction/tvm
```

Listing 4: Commands for changing directory to tvm.

Changes of data types and/or shape configurations are the same as the execution of PANAMERA.

A.3.3 Profiling Results. During the execution, AKG outputs the results to `stdout`, which is similar to "gpu(0): exec = xxx ms/op". Note that the script in the "tvm" directory has disabled the polyhedral scheduler of AKG, which falls back to TVM version 0.6.

A.4 Reproducing the Results of Ansor

A.4.1 Installation. To install Ansor [65], the artifact reviewers can change into the directory named "ansor" using

```
$ cd /path-to-panamera-artifact/reproduction/ansor
```

Listing 5: Commands for changing directory to ansor.

and follow the instructions of README. Ansor is the autotuner of TVM, so one can follow the installation instructions of TVM at https://tvm.apache.org/docs/install/index.html. In particular, Ansor requires the TVM version has to be later than 0.8.

A.4.2 Execution. The commands for reproducing the results of Ansor are as follows.

```
# First, one can reproduce the results of single operators
$ python single-op/single_op.py
# Second, one can reproduce the results of sub-graphs or "composite cases"
$ python composite-op/case1.py
$ python composite-op/case2.py
$ python composite-op/case3.py
$ python composite-op/case4.py
$ python composite-op/case5.py
$ python composite-op/case6.py
$ python composite-op/case7.py
$ python composite-op/case8(fail).py
$ python composite-op/case9(fail).py
$ python composite-op/case10.py
$ python composite-op/case11.py
$ python composite-op/case12.py
```

Listing 6: Commands for reproducing the results of Ansor.

For single operators, changes of data types and/or shape configurations can be achieved by modifying the "single_op.py" file in the "single-op" directory. The default shape configuration and data type are [768, 21128] and *float*32, respectively. They can be modified at line 27 and 24 of the "single_op.py" file, respectively.

A.4.3 Profiling Results. During the execution, Ansor outputs the results to `stdout`, which looks like "Execution time of this operator: xxx ns". Note that the script has been rewritten using Ansor's DSL and the search trail has been set to 1000, which can find a good code variant according to our experience.

A.5 Reproducing the Results of cuDNN

A.5.1 Installation. To reproduce the results of cuDNN reported in the paper, one needs to install MindSpore [32] first. The artifact reviewers can change into the directory named "cudnn" using

```
$ cd /path-to-panamera-artifact/reproduction/cudnn
```

Listing 7: Commands for changing directory to cudnn.

and follow the instructions of the README file. Specifically, the framework can be installed from https://www.mindspore.cn/install/en. One can select version 1.8.1, and the hardware platform should be either GPU CUDA 10.1 or 11.1. Operation system is Linux-x86_64 by default, and Python can be the version 3.8.0. Installation mode can choose `pip`.

A.5.2 Execution. Once MindSpore is installed, one can reproduce the results of cuDNN using

```
$ nvprof python sample.py
```

Listing 8: Commands for reproducing the results of cuDNN.

Changes of data types and/or shape configurations can be achieved by modifying the "input_x" parameter of the "sample.py" file. The default setting uses shape [768,768] and data type *float*64. The "ReduceSum" operator at line 9 can be replaced by "ReduceMax" and "ReduceAnd". As cuDNN does not support fused operators, the result of a sub-graph is the sum of multiple invocations of cuDNN library calls, so we did not provide examples for sub-graphs.

A.5.3 Profiling Results. During the execution, MindSpore outputs the results to `stdout`, indicated by the command of `nvprof`. Note that a reduction operator implemented using cuDNN may call many times of different kernels, *e.g.*, "reduce_tensor_kernel_free" and "op_tensor_kernel_alpha2_zero". The result of a reduction operator executed by cuDNN should be the sum of these kernels.

A.6 Reproducing the Results of CUB

A.6.1 Installation. The code repository of CUB is https://github.com/NVIDIA/cub. The artifact reviewers can first change into the directory named "CUB" and clone CUB using

```
$ cd /path-to-panamera-artifact/reproduction/CUB
$ git clone https://github.com/NVIDIA/cub
$ cd cub/tree/main/examples/block
```

Listing 9: Commands for changing the directory to CUB.

A.6.2 Execution. The examples used to reproduce the results of CUB are offered in its repository. In particular, the two CUDA files "example_block_reduce.cu" at https://github.com/NVIDIA/cub/tree/main/examples/block/example_block_reduce.cu for the all-reduce patterns and "example_device_reduce.cu" at https://github.com/NVIDIA/cub/blob/main/examples/device/example_device_reduce.cu for the x- and y-reduce patterns are used in the experiments. The command to execute the fetched CUDA C++ code of CUB can be executed using

```
$ nvcc -arch=sm_70 example_x_reduce.cu -I../.. -lcudart -O3
```

Listing 10: Commands for reproducing the results of CUB.

where "sm_70" is specified for the NVIDIA V100 GPU and "x" should be replaced by block or device.

As CUB is a templated library, changing data type, operator type and shape configurations requiring some manual efforts. The artifact reviewers can use the two examples to reproduce results of *reduce_sum* with data type *int* if they do not want to change data/operator types or shape configurations. Otherwise, the artifact reviewers can follow the instructions below.

To change data type, one has to change all involved arrays in the file into other data types. For example, the data types of `h_in`, `h_reference` and `d_in` arrays should be change into other ones. The shape configuration is defined by the `num_items` parameter. For "example_device_reduce.cu", it can be manually defined at line 110. For "example_block_reduce.cu", `num_items` is implicitly defined (line 142) as the product of one can change the parameters of the `BLOCK_THREADS` and `ITEMS_PER_THREAD`, which in turn are defined when invoking the `Test` subroutine (line 280). For example, one can invoke the `Test` subroutine using `Test(1024,64,BLOCK_REDUCE_WARP_REDUCTIONS)()` when executing an all-reduce pattern of shape 65536. The operator type can be changed in different ways. For "example_block_reduce.cu", one can change `Sum` at line 96 into *e.g.*, `Max` to reproduce the results of `reduce_max`. For "example_device_reduce.cu", the `Sum` at line 161 can be altered into `Max` or `And` to obtain other operator types.

A.6.3 Profiling Results. During the execution, CUB outputs the results to `stdout`, which is similar to "Average kernel millis: xxx".