Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically

Jie Zhao[1] Cédric Bastoul[2] Yanzhi Yi[3]
Jiahui Hu[3] Wang Nie[3] Renwei Zhang[3] Zhen Geng[3†]
Chong Li[2] **Thibaut Tachon**[2] Zhiliang Gan[3]

[1]State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou
[2]Huawei Technologies France SASU, Paris
[3]Huawei Technologies Co. Ltd., Hangzhou, Beijing and Shenzhen
[†]Now is with the Parallel Computing Software Team at Alibaba, Hangzhou

31st International Conference on Parallel Architectures and Compilation Techniques (PACT'22)

October 12, 2022, Chicago, Illinois, USA

Outline

1. Introduction
2. Dimension Flattening
3. Polyhedral Loop Transformations
4. Code Generation and Optimization
5. Experimental Results
6. Conclusion

What is reduction and why do we study reduction

What is reduction and why do we study reduction

Definition

Reduction is a binary operator $\circledast$ that applies to each element of an input vector V and reduces V to a single value r. Formally,

$$r = \begin{pmatrix} \circledast_{i=1}^{n} \left(v_i^{(1)}\right) \\ \vdots \\ \circledast_{i=1}^{n} \left(v_i^{(d)}\right) \end{pmatrix} = \begin{pmatrix} v_1^{(1)} \circledast v_2^{(1)} \circledast \cdots \circledast v_n^{(1)} \\ \vdots \\ v_1^{(d)} \circledast v_2^{(d)} \circledast \cdots \circledast v_n^{(d)} \end{pmatrix}$$

where the subscript iterates between n vectors and the (parenthesized) superscript between d dimensions of a vector v.

What is reduction and why do we study reduction

Definition

Reduction is a binary operator $\circledast$ that applies to each element of an input vector V and reduces V to a single value r[1]. Formally,

$$r = \begin{pmatrix} \circledast_{i=1}^{n} \left(v_i^{(1)} \right) \\ \vdots \\ \circledast_{i=1}^{n} \left(v_i^{(d)} \right) \end{pmatrix} = \begin{pmatrix} v_1^{(1)} \circledast v_2^{(1)} \circledast \cdots \circledast v_n^{(1)} \\ \vdots \\ v_1^{(d)} \circledast v_2^{(d)} \circledast \cdots \circledast v_n^{(d)} \end{pmatrix}$$

where the subscript iterates between n vectors and the (parenthesized) superscript between d dimensions of a vector v.

[1] r is a constant with respect to V. In particular, it can also be an element of another vector.

What is reduction and why do we study reduction

Definition

Reduction is a binary operator $\circledast$ that applies to each element of an input vector V and reduces V to a single value r[1]. Formally,

$$r = \begin{pmatrix} \circledast_{i=1}^{n}\left(v_i^{(1)}\right) \\ \vdots \\ \circledast_{i=1}^{n}\left(v_i^{(d)}\right) \end{pmatrix} = \begin{pmatrix} v_1^{(1)} \circledast v_2^{(1)} \circledast \cdots \circledast v_n^{(1)} \\ \vdots \\ v_1^{(d)} \circledast v_2^{(d)} \circledast \cdots \circledast v_n^{(d)} \end{pmatrix}$$

where the subscript iterates between n vectors and the (parenthesized) superscript between d dimensions of a vector v.

[1] r is a constant with respect to V. In particular, it can also be an element of another vector.

- Reduction is involved in less compute-intensive operators (e.g., SoftMax, ReLU, BachNorm) of neural network models.

What is reduction and why do we study reduction

Definition

Reduction is a binary operator $\circledast$ that applies to each element of an input vector V and reduces V to a single value r[1]. Formally,

$$r = \begin{pmatrix} \circledast_{i=1}^{n} \left(v_i^{(1)}\right) \\ \vdots \\ \circledast_{i=1}^{n} \left(v_i^{(d)}\right) \end{pmatrix} = \begin{pmatrix} v_1^{(1)} \circledast v_2^{(1)} \circledast \cdots \circledast v_n^{(1)} \\ \vdots \\ v_1^{(d)} \circledast v_2^{(d)} \circledast \cdots \circledast v_n^{(d)} \end{pmatrix}$$

where the subscript iterates between n vectors and the (parenthesized) superscript between d dimensions of a vector v.

[1] r is a constant with respect to V. In particular, it can also be an element of another vector.

- Reduction is involved in less compute-intensive operators (e.g., SoftMax, ReLU, BachNorm) of neural network models.
- Ineffective parallelization of reductions can hamper the performance of such operators, which can in turn result in sub-optimal performance.

What is reduction and why do we study reduction

Definition

Reduction is a binary operator $\circledast$ that applies to each element of an input vector V and reduces V to a single value r[1]. Formally,

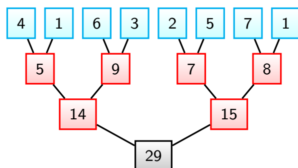
$$r = \begin{pmatrix} \circledast_{i=1}^{n}\left(v_i^{(1)}\right) \\ \vdots \\ \circledast_{i=1}^{n}\left(v_i^{(d)}\right) \end{pmatrix} = \begin{pmatrix} v_1^{(1)} \circledast v_2^{(1)} \circledast \cdots \circledast v_n^{(1)} \\ \vdots \\ v_1^{(d)} \circledast v_2^{(d)} \circledast \cdots \circledast v_n^{(d)} \end{pmatrix}$$

where the subscript iterates between n vectors and the (parenthesized) superscript between d dimensions of a vector v.

[1] r is a constant with respect to V. In particular, it can also be an element of another vector.

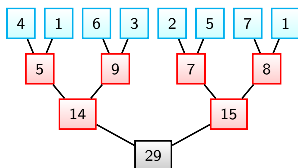
- Reduction is involved in less compute-intensive operators (e.g., SoftMax, ReLU, BachNorm) of neural network models.
- Ineffective parallelization of reductions can hamper the performance of such operators, which can in turn result in sub-optimal performance.
- Optimizing reduction is thus important for parallelizing neural network models but not well studied before.

Why do we target GPU



parallel execution of a reduction $29 = (((((((4+1)+6)+3)+2)+5)+7)+1)$

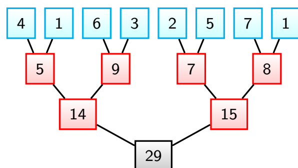
Why do we target GPU



parallel execution of a reduction $29 = (((((((4+1)+6)+3)+2)+5)+7)+1)$

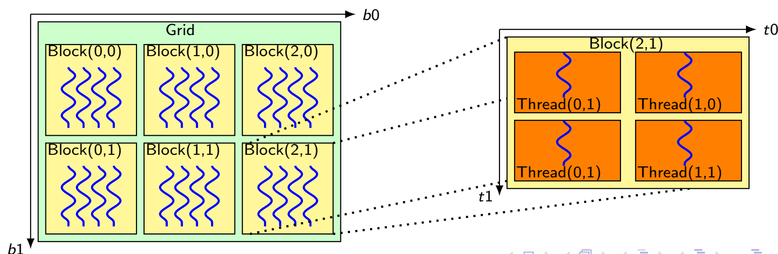
- Parallelism in reduction makes GPU an attractive and suitable target.

Why do we target GPU

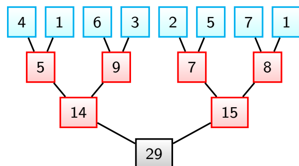


parallel execution of a reduction $29 = (((((((4 + 1) + 6) + 3) + 2) + 5) + 7) + 1)$

- Parallelism in reduction makes GPU an attractive and suitable target.
- GPU abstracts the streaming multiprocessors as blocks and CUDA cores as threads. The number of threads within a block is limited.



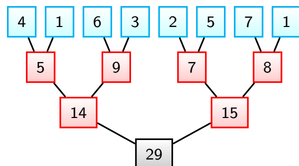
Limitations of prior work on parallel reduction



- Parallel Reductions on GPU

- Polyhedral Parallel Reductions

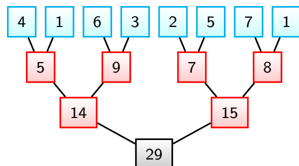
Limitations of prior work on parallel reduction



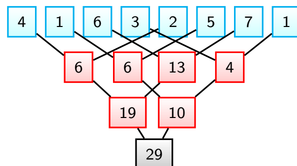
- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
- Polyhedral Parallel Reductions

[1] Mark Harris. "Optimizing parallel reduction in CUDA". *Nvidia developer technology* 2.4 (2007), pp. 1–39.

Limitations of prior work on parallel reduction



interleaved addressing

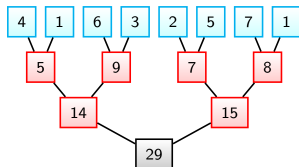


sequential addressing

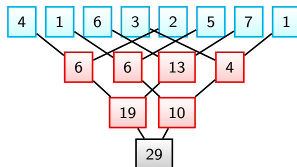
- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
- Polyhedral Parallel Reductions

[1] Mark Harris. "Optimizing parallel reduction in CUDA". *Nvidia developer technology* 2.4 (2007), pp. 1–39.

Limitations of prior work on parallel reduction



interleaved addressing

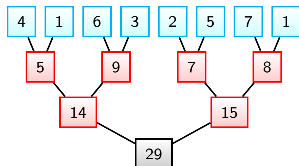


sequential addressing

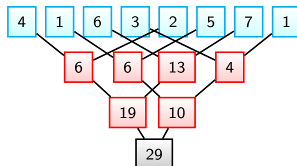
- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
  - Elements can be dispatched to multiple threads, but have to be decomposed into multiple blocks when the number of elements grows
- Polyhedral Parallel Reductions

[1] Mark Harris. "Optimizing parallel reduction in CUDA". *Nvidia developer technology* 2.4 (2007), pp. 1–39.

Limitations of prior work on parallel reduction



interleaved addressing

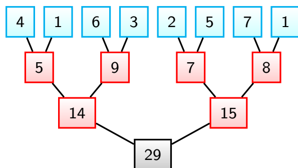


sequential addressing

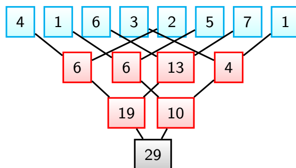
- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
  - Elements can be dispatched to multiple threads, but have to be decomposed into multiple blocks when the number of elements grows
  - Non-trivial due to the missing of synchronization across blocks
  - Incompatible with loop transformations, e.g., fusion, coalescing
- Polyhedral Parallel Reductions

[1] Mark Harris. "Optimizing parallel reduction in CUDA". *Nvidia developer technology* 2.4 (2007), pp. 1–39.

Limitations of prior work on parallel reduction



interleaved addressing



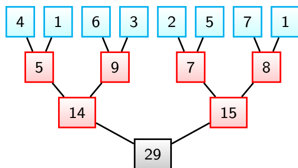
sequential addressing

- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
  - Elements can be dispatched to multiple threads, but have to be decomposed into multiple blocks when the number of elements grows
  - Non-trivial due to the missing of synchronization across blocks
  - Incompatible with loop transformations, e.g., fusion, coalescing
- Polyhedral Parallel Reductions
  - Polyhedral compilation[2] easily composes loop transformations

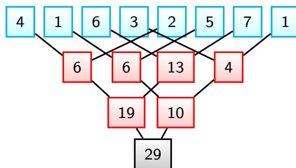
[1] Mark Harris. “Optimizing parallel reduction in CUDA”. *Nvidia developer technology* 2.4 (2007), pp. 1–39.

[2] Paul Feautrier et al. “Polyhedron Model”. *Encyclopedia of Parallel Computing*. Ed. by David Padua. 2011, pp. 1581–1592.

Limitations of prior work on parallel reduction



interleaved addressing



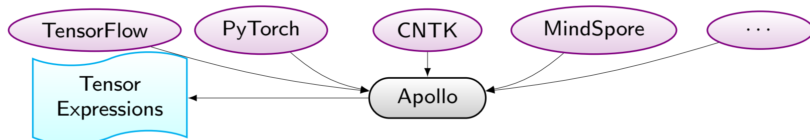
sequential addressing

- Parallel Reductions on GPU
  - Harris[1] revealed many optimization useful for library-based methods
  - Elements can be dispatched to multiple threads, but have to be decomposed into multiple blocks when the number of elements grows
  - Non-trivial due to the missing of synchronization across blocks
  - Incompatible with loop transformations, e.g., fusion, coalescing
- Polyhedral Parallel Reductions
  - Polyhedral compilation[2] easily composes loop transformations
  - Wastes GPU resources when handling multiple, small reduction dims
  - Ineffective handling of global synchronization through privatization

[1] Mark Harris. "Optimizing parallel reduction in CUDA". *Nvidia developer technology* 2.4 (2007), pp. 1–39.

[2] Paul Feautrier et al. "Polyhedron Model". *Encyclopedia of Parallel Computing*. Ed. by David Padua. 2011, pp. 1581–1592.

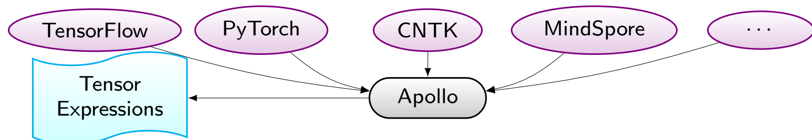
Architecture of PANAMERA



- Takes as input a sub-graph generated by our graph engine Apollo[1]

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

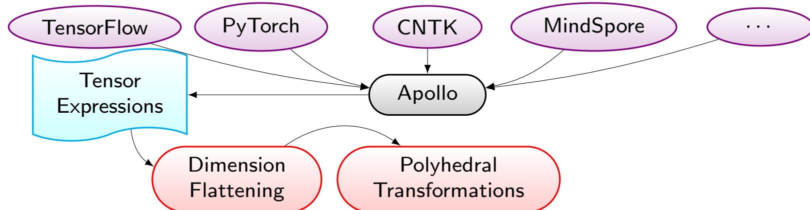
Architecture of PANAMERA



- Takes as input a sub-graph generated by our graph engine Apollo[1], supporting various deep learning frameworks

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

Architecture of PANAMERA

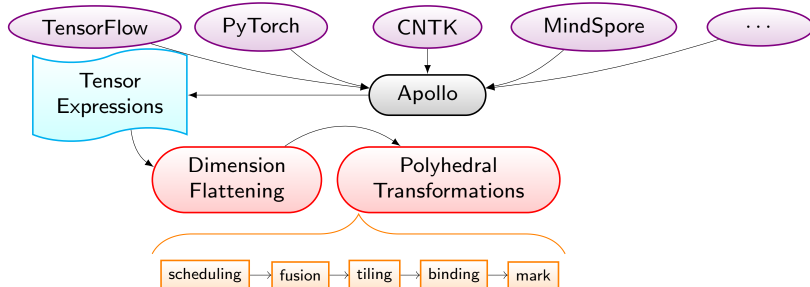


- Takes as input a sub-graph generated by our graph engine Apollo[1], supporting various deep learning frameworks
- Built on top of our polyhedral tensor compiler AKG[2]

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

Architecture of PANAMERA

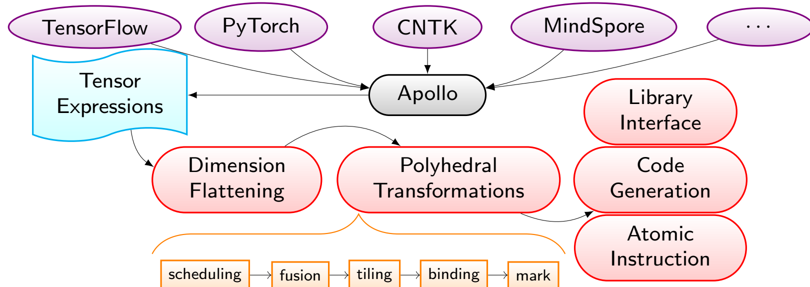


- Takes as input a sub-graph generated by our graph engine Apollo[1], supporting various deep learning frameworks
- Built on top of our polyhedral tensor compiler AKG[2], automatically managing loop transformations and hardware binding

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

Architecture of PANAMERA

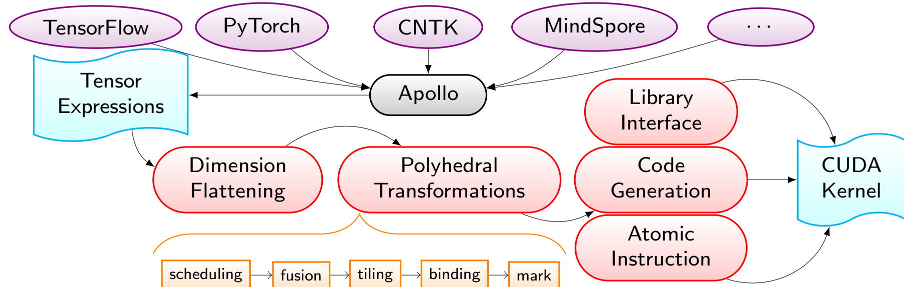


- Takes as input a sub-graph generated by our graph engine Apollo[1], supporting various deep learning frameworks
- Built on top of our polyhedral tensor compiler AKG[2], automatically managing loop transformations and hardware binding
- Wraps self-developed, high-performance libraries

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

Architecture of PANAMERA



- Takes as input a sub-graph generated by our graph engine Apollo[1], supporting various deep learning frameworks
- Built on top of our polyhedral tensor compiler AKG[2], automatically managing loop transformations and hardware binding
- Wraps self-developed, high-performance libraries, fully utilizing low-level hardware instructions

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent
- Calls for loop coalescing[1] to flatten the small reduction dims

(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.

[1] Constantine D Polychronopoulos. "Loop coalescing: A Compiler Transformation for Parallel machines". ICPP 1987, pp. 235–242.

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent
- Calls for loop coalescing[1] to flatten the small reduction dims

(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.

- (a) can be flattened into *all*-reduce

```
reduced for j=0 to N
  R(j_1, ··· , j_r);
```

all-reduce.

```
parallel for i=0 to M
  reduced for j=0 to N
    R(i_1, ··· , i_p, j_1, ··· , j_r);
```

x-reduce.

```
reduced for j=0 to N
  parallel for i=0 to M
    R(j_1, ··· , j_r, i_1, ··· , i_p);
```

y-reduce.

[1] Constantine D Polychronopoulos. "Loop coalescing: A Compiler Transformation for Parallel machines". ICPP 1987, pp. 235–242.

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent
- Calls for loop coalescing[1] to flatten the small reduction dims

(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.

- (a) can be flattened into *all*-reduce; (b) and (c) can be flattened into *x*- and *y*-reduce

```
reduced for j=0 to N
  R(j1, ··· , jr);
```

all-reduce.

```
parallel for i=0 to M
  reduced for j=0 to N
    R(i1, ··· , ip, j1, ··· , jr);
```

x-reduce.

```
reduced for j=0 to N
  parallel for i=0 to M
    R(j1, ··· , jr, i1, ··· , ip);
```

y-reduce.

[1] Constantine D Polychronopoulos. "Loop coalescing: A Compiler Transformation for Parallel machines". ICPP 1987, pp. 235–242.

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent
- Calls for loop coalescing[1] to flatten the small reduction dims

(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.

- (a) can be flattened into *all*-reduce; (b) and (c) can be flattened into *x*- and *y*-reduce; (d) needs loop interchange

```
reduced for j=0 to N
  R(j1, ··· , jr);
```

all-reduce.

```
parallel for i=0 to M
  reduced for j=0 to N
    R(i1, ··· , ip, j1, ··· , jr);
```

x-reduce.

```
reduced for j=0 to N
  parallel for i=0 to M
    R(j1, ··· , jr, i1, ··· , ip);
```

y-reduce.

[1] Constantine D Polychronopoulos. "Loop coalescing: A Compiler Transformation for Parallel machines". ICPP 1987, pp. 235–242.

Flattening multiple dimensions through loop coalescing

- *Nested reductions over multiple variables* are frequent
- Calls for loop coalescing[1] to flatten the small reduction dims

(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.

- (a) can be flattened into *all*-reduce; (b) and (c) can be flattened into *x*- and *y*-reduce; (d) needs loop interchange (always valid)

```
reduced for j=0 to N
  R(j1, ··· , jr);
```

all-reduce.

```
parallel for i=0 to M
  reduced for j=0 to N
    R(i1, ··· , ip, j1, ··· , jr);
```

x-reduce.

```
reduced for j=0 to N
  parallel for i=0 to M
    R(j1, ··· , jr, i1, ··· , ip);
```

y-reduce.

[1] Constantine D Polychronopoulos. "Loop coalescing: A Compiler Transformation for Parallel machines". ICPP 1987, pp. 235–242.

Flattening multiple dimensions through loop coalescing

- Transformation formula of tensor indexes

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

Flattening multiple dimensions through loop coalescing

- Transformation formula of tensor indexes

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

- At most one s_x and one t_y can be symbolic constants, making the flattend dimensions amenable to polyhedral complication

Flattening multiple dimensions through loop coalescing

- Transformation formula of tensor indexes

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

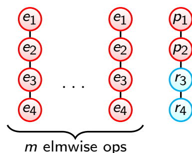
- At most one s_x and one t_y can be symbolic constants, making the flattend dimensions amenable to polyhedral complication
- An example of dimension flattening

```
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);
```

```
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);
```

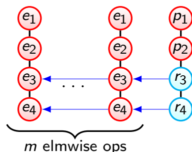
Propagating reduction dependences to enable fusion

- Loop coalescing invalidates the originally possible fusion



Propagating reduction dependences to enable fusion

- Loop coalescing invalidates the originally possible fusion



- Propagate the dependences along the reduced dims

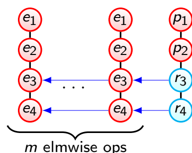
```
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        E(h,w,x,y);
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        R(h,w,x,y);
```

```
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);
```

```
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);
```

Propagating reduction dependences to enable fusion

- Loop coalescing invalidates the originally possible fusion



- Propagate the dependences along the reduced dims

```
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);
```

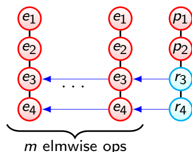
```
for h=0 to 40
  for w=0 to 20
    for x=0 to 10
      for y=0 to 5
        E(h,w,x,y);
for i=0 to 20
  for j=0 to 40*10*5
    R(i,(j/(10*5))%40,(j/5)%10,j%5);
```

```
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);
```

- Recover the fusion possibility

Propagating reduction dependences to enable fusion

- Loop coalescing invalidates the originally possible fusion



- Propagate the dependences along the reduced dims

```
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);
```

```
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);
```

```
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);
```

- Recover the fusion possibility
- Fusion with follow-up elementwise operators is handled by Apollo[1]

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

Polyhedral loop fusion

- Transformation formula of tensor indexes guarantees the "static affine control" requirement of polyhedral compilation

Polyhedral loop fusion

- Transformation formula of tensor indexes guarantees the "static affine control" requirement of polyhedral compilation
- Polyhedral loop fusion is the default heuristic of *isl*[1], reinforced by the post-tiling fusion strategy[2] embedded in AKG when necessary

[1] Sven Verdoolaege. "Isl: An Integer Set Library for the Polyhedral Model". ICMS'10, pp. 299–302.

[2] Jie Zhao et al. "Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data". MICRO'20, pp. 427–441.

Polyhedral loop fusion

- Transformation formula of tensor indexes guarantees the “static affine control” requirement of polyhedral compilation
- Polyhedral loop fusion is the default heuristic of *isl*[1], reinforced by the post-tiling fusion strategy[2] embedded in AKG when necessary
- Always guarantee outer parallelism (possibly by converting a *y*-reduce into an *x*-reduce pattern)

[1] Sven Verdoolaege. “Isl: An Integer Set Library for the Polyhedral Model”. ICMS’10, pp. 299–302.

[2] Jie Zhao et al. “Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data”. MICRO’20, pp. 427–441.

Polyhedral loop fusion

- Transformation formula of tensor indexes guarantees the "static affine control" requirement of polyhedral compilation
- Polyhedral loop fusion is the default heuristic of *isl*[1], reinforced by the post-tiling fusion strategy[2] embedded in AKG when necessary
- Always guarantee outer parallelism (possibly by converting a y-reduce into an x-reduce pattern)
- bind a parallel loop to outer GPU block dims and a reduced loop to inner

[1] Sven Verdoolaege. "Isl: An Integer Set Library for the Polyhedral Model". ICMS'10, pp. 299–302.

[2] Jie Zhao et al. "Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data". MICRO'20, pp. 427–441.

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* 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);
```

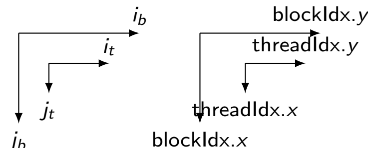
The tiled code.

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* 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);
```

The tiled code.



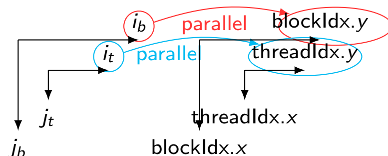
Hardware binding.

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* 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);
```

The tiled code.



Hardware binding.

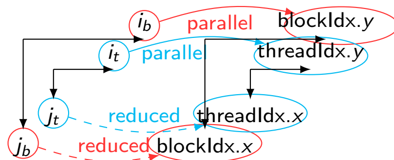
- The outer parallel loops can be bound safely

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* Tile sizes are 32 × 4. */
parallel for ib=0 to M/32
  reduced for jb=0 to N/4
    parallel for it=0 to 32
      reduced for jt=0 to 4
        m elmwise stmts;
        // marked reduce stmt
        R(i1, ··· , ip, j1, ··· , jr);
```

The tiled code.



Hardware binding.

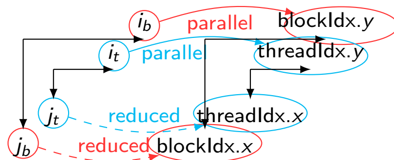
- The outer parallel loops can be bound safely
- The inner reduced loops are bound by ignoring reduction dependences

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* Tile sizes are 32 × 4. */
parallel for ib=0 to M/32
  reduced for jb=0 to N/4
    parallel for it=0 to 32
      reduced for jt=0 to 4
        m elmwise stmts;
        // marked reduce stmt
        R(i1, ··· , ip, j1, ··· , jr);
```

The tiled code.



Hardware binding.

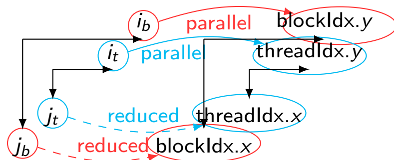
- The outer parallel loops can be bound safely
- The inner reduced loops are bound by ignoring reduction dependences
- This enables the possibility to decompose a reduction operator across multiple blocks when handling large reduction dimensions

Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```
/* Tile sizes are 32 × 4. */
parallel for ib=0 to M/32
  reduced for jb=0 to N/4
    parallel for it=0 to 32
      reduced for jt=0 to 4
        m elmwise stmts;
        // marked reduce stmt
        R(i1, ··· , ip, j1, ··· , jr);
```

The tiled code.



Hardware binding.

- The outer parallel loops can be bound safely
- The inner reduced loops are bound by ignoring reduction dependences
- This enables the possibility to decompose a reduction operator across multiple blocks when handling large reduction dimensions
- Ignored dependences will be resumed during code generation

Orchestration effects of loop transformations

- Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.

[1] Sven Verdoolaege et al. "Scheduling for PPCG". *Report CW* 706 (2017).

Orchestration effects of loop transformations

- Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.
- This isolation can mitigate the polyhedral scheduling overhead, allowing us to optimize reductions with a reasonable cost.

[1] Sven Verdoolaege et al. "Scheduling for PPCG". *Report CW* 706 (2017).

Orchestration effects of loop transformations

- Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.
- This isolation can mitigate the polyhedral scheduling overhead, allowing us to optimize reductions with a reasonable cost.
- Loop interchange before the polyhedral transformations can be harmful to memory coalescing; we avoid this risk by reasoning about tensor layouts using tensor expression language.

[1] Sven Verdoolaege et al. "Scheduling for PPCG". *Report CW* 706 (2017).

Orchestration effects of loop transformations

- Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.
- This isolation can mitigate the polyhedral scheduling overhead, allowing us to optimize reductions with a reasonable cost.
- Loop interchange before the polyhedral transformations can be harmful to memory coalescing; we avoid this risk by reasoning about tensor layouts using tensor expression language.
- Isolating loop coalescing also makes it possible to canonicalize reduction patterns, as shown before.

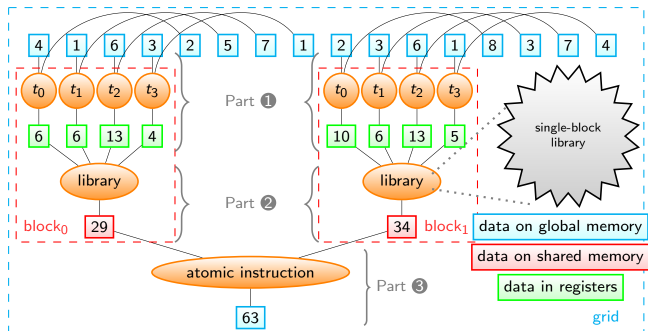
[1] Sven Verdoolaege et al. "Scheduling for PPCG". *Report CW* 706 (2017).

Orchestration effects of loop transformations

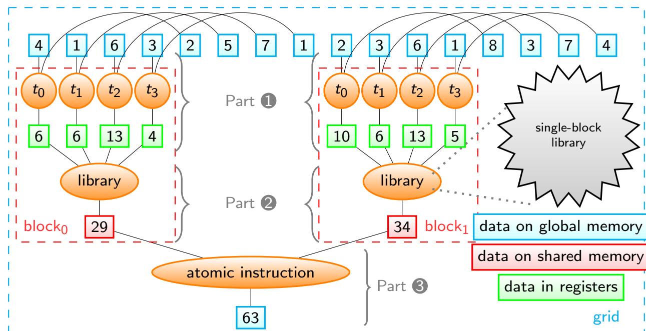
- Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.
- This isolation can mitigate the polyhedral scheduling overhead, allowing us to optimize reductions with a reasonable cost.
- Loop interchange before the polyhedral transformations can be harmful to memory coalescing; we avoid this risk by reasoning about tensor layouts using tensor expression language.
- Isolating loop coalescing also makes it possible to canonicalize reduction patterns, as shown before.
- Our three canonical reduction forms simplify hardware binding strategies and compress the search space of tile sizes.

[1] Sven Verdoolaege et al. "Scheduling for PPCG". *Report CW* 706 (2017).

A self-developed library using atomic instructions

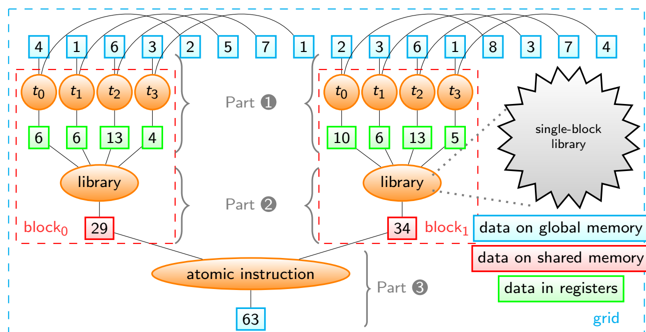


A self-developed library using atomic instructions



- Part ❶ enables sequential addressing and fusion with other operators

A self-developed library using atomic instructions

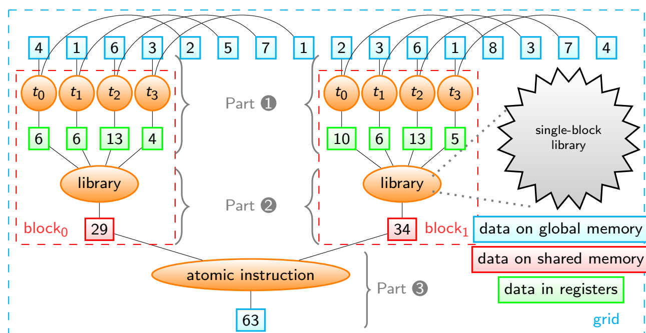


- Part ❶ enables sequential addressing and fusion with other operators
- Part ❷ ensures higher performance than stand-alone compilation approaches[1][2] and minimizes the number of involved blocks

[1] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". OSDI 2018, pp. 578–594.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

A self-developed library using atomic instructions



- Part ➊ enables sequential addressing and fusion with other operators
- Part ➋ ensures higher performance than stand-alone compilation approaches[1][2] and minimizes the number of involved blocks
- Part ➌ carries out global synchronization using atomic instructions, avoiding the need to invoke multiple kernels for neural network models

[1] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". OSDI 2018, pp. 578–594.

[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

Example templated code

```
__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 1, 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 2, automatic invocation of library routines. */
  Parallel_Reduce<T,OP,4,all>(op,&block_sum[0],shared_buf,local_sum);
  __synchthreads();
  /* Part 3, automatic global sychronization using atomics. */
  if(threadIdx.x==0)
    Atomic_Return<T,OP>(block_sum[0],&output[0],op);
}
```

Example templated code

```
__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 1, 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 2, automatic invocation of library routines. */
  Parallel_Reduce<T,OP,4,all>(op,&block_sum[0],shared_buf,local_sum);
  __synchthreads();
  /* Part 3, automatic global sychronization using atomics. */
  if(threadIdx.x==0)
    Atomic_Return<T,OP>(block_sum[0],&output[0],op);
}
```

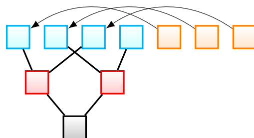
- OP can be instanced using *summation*, *product*, *min*, *max*, *logical AND* and *logical OR*
- T can be one of *double, float32, float16, bool, long long int* and *int*
- Parallel_Reduce and Atomic_Return are interfaces to our library and low-level atomic instructions
- **__synchthreads()** is automatically inserted

Generalizing the templated code generator

- Handling an irregular input size n

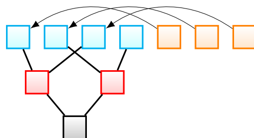
Generalizing the templated code generator

- Handling an irregular input size n
  - Divide n into 2^k (greatest power of two less than n) and $n - 2^k$
  - Perform a local reduction to convert n into an irregular size 2^k



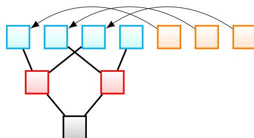
Generalizing the templated code generator

- Handling an irregular input size n
  - Divide n into 2^k (greatest power of two less than n) and $n - 2^k$
  - Perform a local reduction to convert n into an irregular size 2^k
  - Designed for Part ➋, this optimization is also useful for irregular sizes across multiple blocks



Generalizing the templated code generator

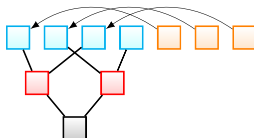
- Handling an irregular input size n
  - Divide n into 2^k (greatest power of two less than n) and $n - 2^k$
  - Perform a local reduction to convert n into an irregular size 2^k
  - Designed for Part ➋, this optimization is also useful for irregular sizes across multiple blocks



- Generalization to multiple reductions

Generalizing the templated code generator

- Handling an irregular input size n
  - Divide n into 2^k (greatest power of two less than n) and $n - 2^k$
  - Perform a local reduction to convert n into an irregular size 2^k
  - Designed for Part ➋, this optimization is also useful for irregular sizes across multiple blocks



- Generalization to multiple reductions
  - Still fusible when their enclosing loop nests are identical and reductions patterns are the same
  - More complicated scenarios can be feedback to the upstream graph compiler[1] for exploiting more fusion opportunities

[1] Jie Zhao et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". Vol. 4. MLSys'22. 2022, pp. 1–19.

Example code for fused reductions

```
__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];
}
```

Fusing 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 `output1`.

Experimental setups

Hardware	NIVDIA Tesla V100 GPU
Operating system	Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)
CUDA toolkit	version 10.1, -O3 option
Python	version 3.7.5
Neural network framework	MindSpore[1], version 1.8.1
Baselines	AKG[2], TVM (v0.6)[3], Ansor[4], cuDNN (v7.6.4) and CUB (v1.8)[5]
Reported time	Geometric mean of 10 executions

[1] Huawei. *MindSpore*. 2020. URL: https://www.mindspore.cn/en.

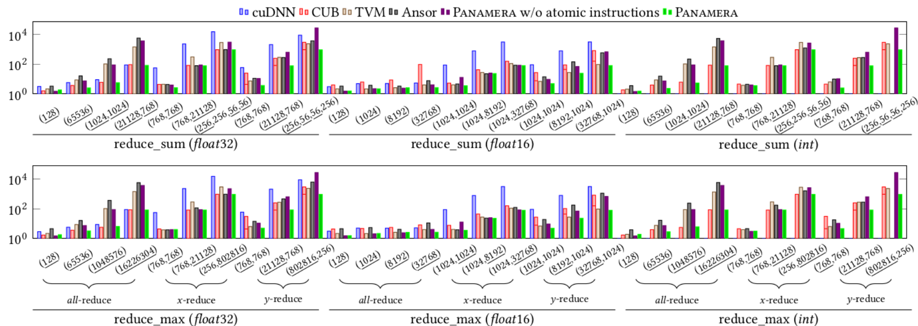
[2] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". PLDI 2021, pp. 1233–1248.

[3] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". OSDI 2018, pp. 578–594.

[4] Lianmin Zheng et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning". OSDI 2020, pp. 863–879.

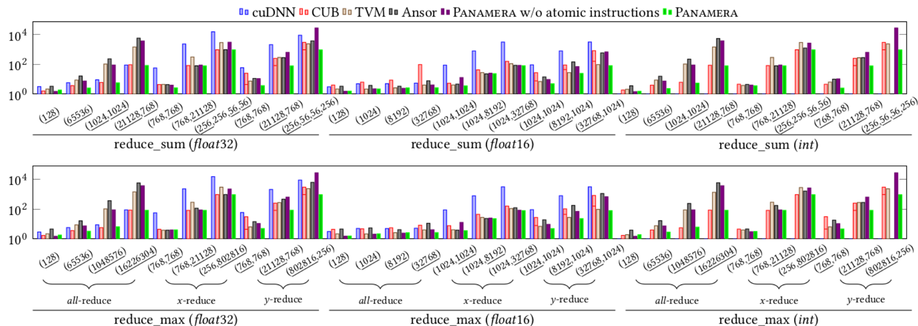
[5] Nvidia. *CUB Documentation*. 2018. URL: https://nvlabs.github.io/cub/.

Results of single operators



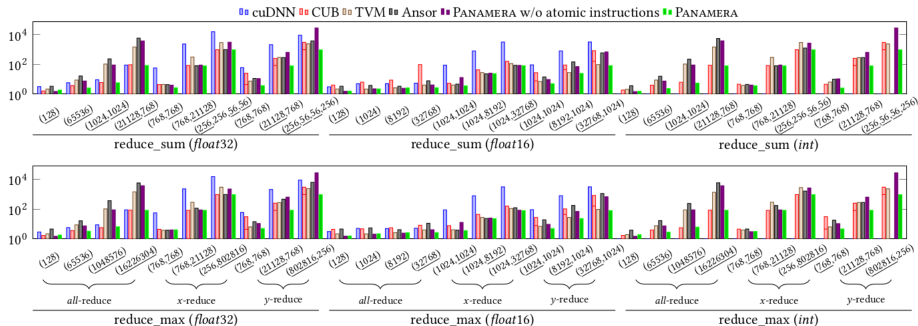
- 3 operators: *reduce_sum*, *reduce_max* and *reduce_and*
- 4 data types: *float*32, *float*16, *int* and *bool*
- 10 different input shape configurations (*reduce_sum x* axis: original shapes; *reduce_max x* axis: flattened shapes)
- *y* axis: log scaled execution time in microseconds; lower is better

Results of single operators



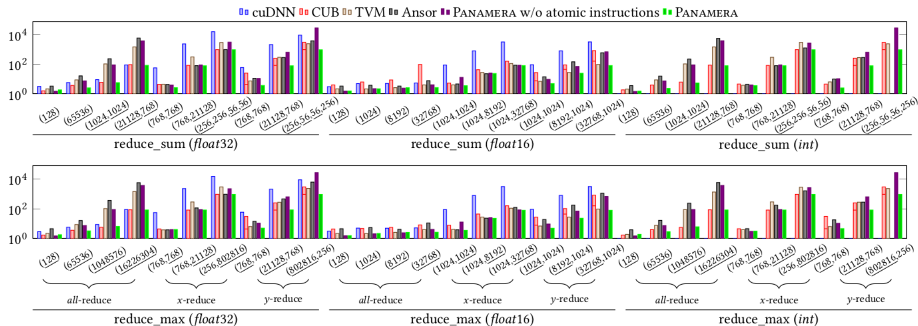
- 3 operators: *reduce_sum*, *reduce_max* and *reduce_and*
- 4 data types: *float*32, *float*16, *int* and *bool*
- 10 different input shape configurations (*reduce_sum x* axis: original shapes; *reduce_max x* axis: flattened shapes)
- *y* axis: log scaled execution time in microseconds; lower is better
- Please refer to the paper for the result of *reduce_and*

Results of single operators



- TVM performs poorly under larger sizes, especially for *all*-reduce
- Ansor sometimes quits when handling *y*-reduce
- cuDNN may not perform loop coalescing and always uses an identical 3D thread configuration $<< 8, 16, 1 >>$
- CUB seems more suitable for reductions along the inner loops

Results of single operators



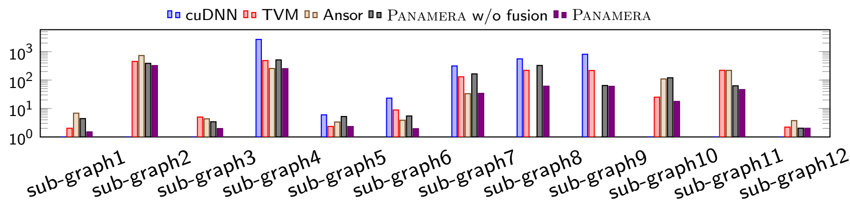
- TVM performs poorly under larger sizes, especially for *all*-reduce
- Ansor sometimes quits when handling *y*-reduce
- cuDNN may not perform loop coalescing and always uses an identical 3D thread configuration $<< 8, 16, 1 >>$
- CUB seems more suitable for reductions along the inner loops
- PANAMERA outperforms cuDNN, CUB, TVM and Ansor by 33.7×, 3.5×, 5.4× and 9.6×, respectively

Results of fused operators

Summary of fused operators. 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	-	-	-	-

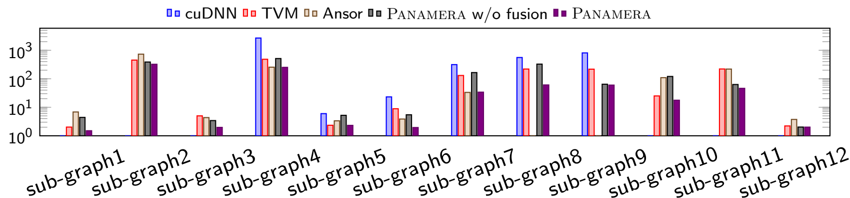
Results of fused operators



y axis: log scaled execution time in microseconds; lower is better.

- While not exploiting fusion, cuDNN does not support *type casting* or *reshaping* operators
- TVM/Ansor under-performs when multi-block parallelism is essential for performance

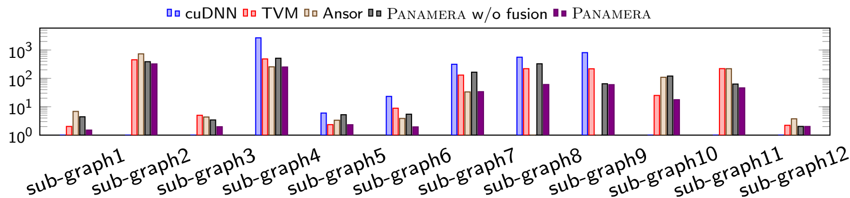
Results of fused operators



y axis: log scaled execution time in microseconds; lower is better.

- While not exploiting fusion, cuDNN does not support *type casting* or *reshaping* operators; PANAMERA exhibits a better scalablity by handling more scenarios
- TVM/Ansor under-performs when multi-block parallelism is essential for performance; PANAMERA can better handle the multi-block parallelism

Results of fused operators



y axis: log scaled execution time in microseconds; lower is better.

- While not exploiting fusion, cuDNN does not support *type casting* or *reshaping* operators; PANAMERA exhibits a better scalablity by handling more scenarios
- TVM/Ansor under-performs when multi-block parallelism is essential for performance; PANAMERA can better handle the multi-block parallelism
- On average, PANAMERA outperforms cuDNN, TVM and Ansor by 9.5×, 2.6× and 2.7×, respectively

Results of end-to-end workloads

Execution time in milliseconds (GPT-3 is executed on a Tesla A100 GPU)

Workloads	MindSpore	TVM	Ansor	AKG	PANAMERA	Improvement over				number of fused ops
						MindSpore	TVM	Ansor	AKG	
BERT[1]	352.2	138.0	120.3	124.0	111.0	+217%	+24%	+8%	+12%	304
Wide&Deep[2]	22.4	12.5	12.8	12.6	11.0	+104%	+14%	+16%	+15%	74
VGG-16[3]	70.4	65.7	66.3	67.6	64.2	+10%	+2%	+3%	+5%	39
MobileNet-v3[4]	151.4	133.0	129.4	136.8	131.5	+15%	+1%	-2%	+4%	52
Transformer[5]	157.8	132.4	126.5	136.8	79.2	+99%	+67%	+60%	+73%	746
GPT-3[6]	483.0	133.9	131.3	146.2	123.7	+290%	+8%	+6%	+18%	409
average						+122.5%	+19.3%	+15.2%	+21.2%	

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". NAACL 2019, Volumn 1, pp. 4171–4186.

[2] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". DLRS 2016, pp. 7–10.

[3] Karen Simonyan et al. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[4] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[5] Ashish Vaswani et al. "Attention is All You Need". NIPS'17, pp. 6000–6010.

[6] Tom Brown et al. "Language Models are Few-Shot Learners". NeurIPS 2020, pp. 1877–1901.

Results of end-to-end workloads

Execution time in milliseconds (GPT-3 is executed on a Tesla A100 GPU)

Workloads	MindSpore	TVM	Ansor	AKG	PANAMERA	Improvement over				number of fused ops
						MindSpore	TVM	Ansor	AKG	
BERT[1]	352.2	138.0	120.3	124.0	111.0	+217%	+24%	+8%	+12%	304
Wide&Deep[2]	22.4	12.5	12.8	12.6	11.0	+104%	+14%	+16%	+15%	74
VGG-16[3]	70.4	65.7	66.3	67.6	64.2	+10%	+2%	+3%	+5%	39
MobileNet-v3[4]	151.4	133.0	129.4	136.8	131.5	+15%	+1%	-2%	+4%	52
Transformer[5]	157.8	132.4	126.5	136.8	79.2	+99%	+67%	+60%	+73%	746
GPT-3[6]	483.0	133.9	131.3	146.2	123.7	+290%	+8%	+6%	+18%	409
average						+122.5%	+19.3%	+15.2%	+21.2%	

- number of operators fused by PANAMERA in a workload

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". NAACL 2019, Volumn 1, pp. 4171–4186.

[2] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". DLRS 2016, pp. 7–10.

[3] Karen Simonyan et al. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[4] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[5] Ashish Vaswani et al. "Attention is All You Need". NIPS'17, pp. 6000–6010.

[6] Tom Brown et al. "Language Models are Few-Shot Learners". NeurIPS 2020, pp. 1877–1901.

Results of end-to-end workloads

Execution time in milliseconds (GPT-3 is executed on a Tesla A100 GPU)

Workloads	MindSpore	TVM	Ansor	AKG	Panamera	Improvement over				number of fused ops
						MindSpore	TVM	Ansor	AKG	
BERT[1]	352.2	138.0	120.3	124.0	111.0	+217%	+24%	+8%	+12%	304
Wide&Deep[2]	22.4	12.5	12.8	12.6	11.0	+104%	+14%	+16%	+15%	74
VGG-16[3]	70.4	65.7	66.3	67.6	64.2	+10%	+2%	+3%	+5%	39
MobileNet-v3[4]	151.4	133.0	129.4	136.8	131.5	+15%	+1%	-2%	+4%	52
Transformer[5]	157.8	132.4	126.5	136.8	79.2	+99%	+67%	+60%	+73%	746
GPT-3[6]	483.0	133.9	131.3	146.2	123.7	+290%	+8%	+6%	+18%	409
average						+122.5%	+19.3%	+15.2%	+21.2%	

- number of operators fused by Panamera in a workload
- Panamera enhances the performance of AKG by 21.2% on average

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". NAACL 2019, Volumn 1, pp. 4171–4186.

[2] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". DLRS 2016, pp. 7–10.

[3] Karen Simonyan et al. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[4] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[5] Ashish Vaswani et al. "Attention is All You Need". NIPS'17, pp. 6000–6010.

[6] Tom Brown et al. "Language Models are Few-Shot Learners". NeurIPS 2020, pp. 1877–1901.

Results of end-to-end workloads

Execution time in milliseconds (GPT-3 is executed on a Tesla A100 GPU)

Workloads	MindSpore	TVM	Ansor	AKG	PANAMERA	Improvement over				number of fused ops
						MindSpore	TVM	Ansor	AKG	
BERT[1]	352.2	138.0	120.3	124.0	111.0	+217%	+24%	+8%	+12%	304
Wide&Deep[2]	22.4	12.5	12.8	12.6	11.0	+104%	+14%	+16%	+15%	74
VGG-16[3]	70.4	65.7	66.3	67.6	64.2	+10%	+2%	+3%	+5%	39
MobileNet-v3[4]	151.4	133.0	129.4	136.8	131.5	+15%	+1%	-2%	+4%	52
Transformer[5]	157.8	132.4	126.5	136.8	79.2	+99%	+67%	+60%	+73%	746
GPT-3[6]	483.0	133.9	131.3	146.2	123.7	+290%	+8%	+6%	+18%	409
average						+122.5%	+19.3%	+15.2%	+21.2%	

- number of operators fused by PANAMERA in a workload
- PANAMERA enhances the performance of AKG by 21.2% on average
- AKG + PANAMERA outperforms MindSpore (backed by CUDA libraries), TVM and Ansor by 122.5%, 19.3% and 15.2%, respectively

[1] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". NAACL 2019, Volumn 1, pp. 4171–4186.

[2] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". DLRS 2016, pp. 7–10.

[3] Karen Simonyan et al. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[4] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[5] Ashish Vaswani et al. "Attention is All You Need". NIPS'17, pp. 6000–6010.

[6] Tom Brown et al. "Language Models are Few-Shot Learners". NeurIPS 2020, pp. 1877–1901.

Summary of the contributions

- PANAMERA canonicalizes reductions in DL not considered before, making it possible to effectively decompose various reductions and fully harness GPU hardware resources.

Summary of the contributions

- PANAMERA canonicalizes reductions in DL not considered before, 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 and decreasing the tuning space size of DL reductions.

Summary of the contributions

- PANAMERA canonicalizes reductions in DL not considered before, 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 and decreasing the tuning space size of DL reductions.
- PANAMERA exhibits a much better scalability to data types and tensor shapes than many CUDA libraries, rendering a compiler applicable to various reduction scenarios.

Summary of the contributions

- PANAMERA canonicalizes reductions in DL not considered before, 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 and decreasing the tuning space size of DL reductions.
- PANAMERA exhibits a much better scalability to data types and tensor shapes than many CUDA libraries, 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.

Potentials and limitations

+ No threshold on the number of fused operators, which is only determined according to the available hardware resources

Potentials and limitations

+ No threshold on the number of fused operators, which is only determined according to the available hardware resources
+ x- and y-reduce patterns can be executed on multiple GPUs, with parallel for loops decomposed evenly across devices

Potentials and limitations

- No threshold on the number of fused operators, which is only determined according to the available hardware resources
- x- and y-reduce patterns can be executed on multiple GPUs, with parallel for loops decomposed evenly across devices
- Applicable to matrix multiplication but not encouraged

Performance comparison of matrix multiplication when optimized using PANAMERA and tensor cores in AKG. Execution time is in microseconds.

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

Potentials and limitations

+ No threshold on the number of fused operators, which is only determined according to the available hardware resources
+ x- and y-reduce patterns can be executed on multiple GPUs, with parallel for loops decomposed evenly across devices
+ Applicable to matrix multiplication but not encouraged

Performance comparison of matrix multiplication when optimized using PANAMERA and tensor cores in AKG. Execution time is in microseconds.

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

- non-determinism issue of atomic instructions
- (slight) manual effort required to configure templated routines in the generated code

Potentials and limitations

+ No threshold on the number of fused operators, which is only determined according to the available hardware resources
+ x- and y-reduce patterns can be executed on multiple GPUs, with parallel for loops decomposed evenly across devices
+ Applicable to matrix multiplication but not encouraged

Performance comparison of matrix multiplication when optimized using PANAMERA and tensor cores in AKG. Execution time is 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%

- non-determinism issue of atomic instructions; the hardware scheme for deterministic atomic buffering[1] is a solution
- (slight) manual effort required to configure templated routines in the generated code; fully automation is under construction

[1] Yuan Hsi Chou et al. "Deterministic Atomic Buffering". MICRO 2020, pp. 981–995.

Questions & Answers

Thank you!

Any Questions?