Parallelizing Neural Network Models Effectively on GPU by Implementing Reductions Atomically

Jie Zhao¹  Cédric Bastoul²  Yanzhi Yi³
Jiahui Hu³  Wang Nie³  Renwei Zhang³  Zhen Geng³†
Chong Li²  Thibaut Tachon²  Zhiliang Gan³

¹State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou
²Huawei Technologies France SASU, Paris
³Huawei 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
What is reduction and why do we study reduction?

**Definition**

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

$$r = \left[ \left( v^{(1)}_1 \ast v^{(1)}_2 \ast \cdots \ast v^{(1)}_n \right) \ldots \ldots \left( v^{(d)}_1 \ast v^{(d)}_2 \ast \cdots \ast v^{(d)}_n \right) \right]$$

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

**Reduction is involved in less compute-intensive operators (e.g., SoftMax, ReLU, BatchNorm) 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.**
What is reduction and why do we study reduction

**Definition**

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

$$r = \left( \otimes_{i=1}^{n} \left( v_i^{(1)} \right) \right) = \left( v_1^{(1)} \otimes v_2^{(1)} \otimes \ldots \otimes v_n^{(1)} \right)$$

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 ⊗ that applies to each element of an input vector $V$ and reduces $V$ to a single value $r^{[1]}$. Formally,

$$r = \left( \bigotimes_{i=1}^{n} (v_i^{(1)}) \right) = \left( \begin{array}{c}
    v_1^{(1)} \otimes v_2^{(1)} \otimes \cdots \otimes v_n^{(1)} \\
    \vdots \\
    v_1^{(d)} \otimes v_2^{(d)} \otimes \cdots \otimes v_n^{(d)}
\end{array} \right)$$

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.
**Definition**

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

$$r = \left( \begin{array}{c}
\otimes_{i=1}^{n} (v_i^{(1)}) \\
\vdots \\
\otimes_{i=1}^{n} (v_i^{(d)})
\end{array} \right) = \left( \begin{array}{c}
v_1^{(1)} \otimes v_2^{(1)} \otimes \cdots \otimes v_n^{(1)} \\
\vdots \\
v_1^{(d)} \otimes v_2^{(d)} \otimes \cdots \otimes v_n^{(d)}
\end{array} \right)$$

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 \( \otimes \) that applies to each element of an input vector \( V \) and reduces \( V \) to a single value \( r \)[1]. Formally,

\[
\begin{align*}
    r &= \left( \otimes_{i=1}^{n} \left( v_i^{(1)} \right) \right) = \left( \begin{array}{c}
        v_1^{(1)} \\
        \vdots \\
        v_n^{(1)}
    \end{array} \right) \\
    &= \left( \otimes_{i=1}^{n} \left( v_i^{(d)} \right) \right) = \left( \begin{array}{c}
        v_1^{(d)} \\
        \vdots \\
        v_n^{(d)}
    \end{array} \right)
\end{align*}
\]

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, BatchNorm) 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 $\odot$ that applies to each element of an input vector $V$ and reduces $V$ to a single value $r^{[1]}$. Formally,

$$ r = \left( \begin{array}{c}
\odot_{i=1}^{n} \left( v^{(1)}_i \right) \\
\vdots \\
\odot_{i=1}^{n} \left( v^{(d)}_i \right)
\end{array} \right) = \left( \begin{array}{c}
v^{(1)}_1 \odot v^{(1)}_2 \odot \cdots \odot v^{(1)}_n \\
\vdots \\
v^{(d)}_1 \odot v^{(d)}_2 \odot \cdots \odot v^{(d)}_n
\end{array} \right) $$

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, BatchNorm) 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.
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
Parallel Reductions on GPU
- Harris\cite{Harris} revealed many optimization useful for library-based methods

Polyhedral Parallel Reductions
Limitations of prior work on parallel reduction

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

---

Limitations of prior work on parallel reduction

- 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

---

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

Limitations of prior work on parallel reduction

- Parallel Reductions on GPU
  - Harris\cite{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\cite{2} easily composes loop transformations

---


Limitations of prior work on parallel reduction

- Parallel Reductions on GPU
  - Harris\textsuperscript{[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\textsuperscript{[2]} easily composes loop transformations
  - Wastes GPU resources when handling multiple, small reduction dims
  - Ineffective handling of global synchronization through privatization


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

• Takes as input a sub-graph generated by our graph engine Apollo\(^1\), supporting various deep learning frameworks

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]

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

Built on top of our polyhedral tensor compiler AKG\textsuperscript{[2]}, automatically managing loop transformations and hardware binding


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


Takes as input a sub-graph generated by our graph engine Apollo\cite{1}, supporting various deep learning frameworks

Built on top of our polyhedral tensor compiler AKG\cite{2}, automatically managing loop transformations and hardware binding

Wraps self-developed, high-performance libraries, fully utilizing low-level hardware instructions

---


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.

Flattening multiple dimensions through loop coalescing

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

\[
\begin{array}{cccc}
  \text{r}_1 & \text{r}_2 & \text{r}_3 & \text{r}_4 \\
  \text{p}_1 & \text{p}_2 & \text{p}_3 & \text{p}_4
\end{array}
\]

(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

\[
\begin{align*}
\text{parallel for } i=0 \text{ to } M \\
\text{reduced for } j=0 \text{ to } N \\
R(j_1, \cdots, j_r, i_1, \cdots, i_p);
\end{align*}
\]

\[
\begin{align*}
\text{all-reduce.} \quad \text{x-reduce.}
\end{align*}
\]

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

\[
\begin{align*}
\text{reduced for } j=0 \text{ to } N & \quad R(j_1, \ldots, j_r); \\
\text{all}-reduce. & \\
\text{parallel for } i=0 \text{ to } M & \quad R(i_1, \ldots, i_p, j_1, \ldots, j_r); \\
\text{x}-reduce. & \\
\text{reduced for } j=0 \text{ to } N & \quad R(j_1, \ldots, j_r, i_1, \ldots, i_p); \\
\text{y}-reduce. &
\end{align*}
\]

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 \textit{all}-reduce; (b) and (c) can be flattened into \textit{x}- and \textit{y}-reduce; (d) needs loop interchange

\begin{align*}
\text{reduced for } j=0 \text{ to } N & \quad \text{parallel for } i=0 \text{ to } M \quad \text{reduced for } j=0 \text{ to } N \\
& \quad R(j_1, \ldots, j_r); \quad \text{parallel for } i=0 \text{ to } M \quad R(i_1, \ldots, i_p, j_1, \ldots, j_r); \\
\textit{all}-reduce. & \quad \textit{x}-reduce. \quad \textit{y}-reduce.
\end{align*}

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)

\[
\text{reduced for } j=0 \text{ to } N \\
R(j_1, \ldots, j_r);
\]

\[
\text{parallel for } i=0 \text{ to } M \\
\text{reduced for } j=0 \text{ to } N \\
R(i_1, \ldots, i_p, j_1, \ldots, j_r);
\]

\[
\text{all-reduce.}
\]

\[
\text{x-reduce.}
\]

\[
\text{y-reduce.}
\]

---

Transforming tensor index notation

\[
M = \prod_{x=1}^{p} s_x = s_1 \times \cdots \times s_p, \quad N = \prod_{y=1}^{r} t_y = t_1 \times \cdots \times t_r;
\]

\[
i_a = \left[ \frac{i}{\prod_{x=a+1}^{p} s_x} \right] \mod s_a : (1 \leq a < p), \quad i_p = i \mod s_p;
\]

\[
j_b = \left[ \frac{j}{\prod_{y=b+1}^{r} t_y} \right] \mod t_b : (1 \leq b < r), \quad j_r = j \mod t_r;
\]
Flattening multiple dimensions through loop coalescing

- Transformation formula of tensor indexes

\[
M = \prod_{x=1}^{p} s_x = s_1 \times \cdots \times s_p, \quad N = \prod_{y=1}^{r} t_y = t_1 \times \cdots \times t_r;
\]

\[
i_a = \left\lfloor \frac{i}{\prod_{x=a+1}^{p} s_x} \right\rfloor \mod s_a : (1 \leq a < p), \quad i_p = i \mod s_p;
\]

\[
\left\lfloor \frac{j}{\prod_{y=b+1}^{r} t_y} \right\rfloor \mod t_b : (1 \leq b < r), \quad j_r = j \mod t_r;
\]

- At most one \( s_x \) and one \( t_y \) can be symbolic constants, making the flattened dimensions amenable to polyhedral complication
Transformation formula of tensor indexes

\[
\begin{align*}
M &= \prod_{x=1}^{p} s_x = s_1 \times \cdots \times s_p, \\
N &= \prod_{y=1}^{r} t_y = t_1 \times \cdots \times t_r; \\
i_a &= \left\lfloor \frac{i}{\prod_{x=a+1}^{p} s_x} \right\rfloor \mod s_a : (1 \leq a < p), i_p = i \mod s_p; \\
j_b &= \left\lfloor \frac{j}{\prod_{y=b+1}^{r} t_y} \right\rfloor \mod t_b : (1 \leq b < r), j_r = j \mod t_r;
\end{align*}
\]

At most one \(s_x\) and one \(t_y\) can be symbolic constants, making the flattened 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

Loop Diagram:

- $e_1$ connected to $e_2$, $e_3$, $e_4$
- $e_1$ connected to $e_2$, $e_3$, $e_4$
- $p_1$ connected to $p_2$, $r_3$, $r_4$

$m$ elmwise ops
Propagating reduction dependences to enable fusion

- Loop coalescing invalidates the originally possible fusion

- Propagate the dependences along the reduced dims

```plaintext
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);
```

```plaintext
parallal 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

```c
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);
```

- Recover the fusion possibility

```c
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

  ```c
  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 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 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]

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

---


\[2\] 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)

---


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


Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```c
/* 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(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**

```c
/* Tile sizes are 32 x 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(i_1, \ldots, i_p, j_1, \ldots, j_r);
```

The tiled code.

Hardware binding.
Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration

```c
/* 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.

- The outer parallel loops can be bound safely
Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration
  
  ```c
  /* Tile sizes are 32 x 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(\(i_1, \cdots, \hat{i}, \hat{j}, \cdots, j_r\));
  ```

  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

```c
/* 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
  \(R(\ldots, i_p, \ldots, j_r, \ldots);\)
```

The tiled code.

- 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

Hardware binding.
Tiling and binding reduced dimensions

- Tiling is performed on top of a fusion configuration
  
  /* Tile sizes are $32 \times 4$. */
  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
  $m$ elwise stmts;
  // marked reduce stmt
  $R(i_1, \cdots, i_p, j_1, \cdots, j_r)$;

  The tiled code.

- 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

Hardware binding.
Loop coalescing is an achievable but undesired transformation in polyhedral compilation[1], we isolates it as a preprocessing step in dimension flattening.
Orchestration effects of loop transformations

- Loop coalescing is an achievable but undesired transformation in polyhedral compilation\cite{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.

\cite{1} Sven Verdoolaege et al. “Scheduling for PPCG”. Report CW 706 (2017).
Loop coalescing is an achievable but undesired transformation in polyhedral compilation\cite{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.

\cite{1} Sven Verdoolaege et al. “Scheduling for PPCG”. Report CW 706 (2017).
Loop coalescing is an achievable but undesired transformation in polyhedral compilation\cite{Verdoolaege_2017}, 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.

\cite{Verdoolaege_2017} Sven Verdoolaege et al. “Scheduling for PPCG”. *Report CW 706 (2017).*
Loop coalescing is an achievable but undesired transformation in polyhedral compilation\cite{Verdoolaege2017}, 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.

A self-developed library using atomic instructions

Part 1
- Enables sequential addressing and fusion with other operators

Part 2
- Ensures higher performance than stand-alone compilation approaches and minimizes the number of involved blocks

Part 3
- Carries out global synchronization using atomic instructions, avoiding the need to invoke multiple kernels for neural network models

Code Generation and Optimization
Self-developed Library
October 12, Chicago, Illinois
A self-developed library using atomic instructions

- Part 1 enables sequential addressing and fusion with other operators

![Diagram of library and atomic instructions with grid, block, data on global memory, data on shared memory, data in registers]
A self-developed library using atomic instructions

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

A self-developed library using atomic instructions

• Part 1 enables sequential addressing and fusion with other operators
• Part 2 ensures higher performance than stand-alone compilation approaches\[1\][2] and minimizes the number of involved blocks
• Part 3 carries out global synchronization using atomic instructions, avoiding the need to invoke multiple kernels for neural network models

Example templated code

```c
__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
ParallelReduce and AtomicReturn are interfaces to our library and low-level atomic instructions.
synchthreads() is automatically inserted.
Example templated code

```c
__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 synchronization 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
Handling an irregular input size $n$
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 $2^k$
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 2, 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 2, 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 2, 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

Example code for fused reductions

```c
__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

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


Results of single operators

- 3 operators: `reduce_sum`, `reduce_max` and `reduce_and`
- 4 data types: `float32`, `float16`, `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

- **3 operators:** `reduce_sum`, `reduce_max` and `reduce_and`
- **4 data types:** `float32`, `float16`, `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`
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.
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
Summary of fused operators. cast16 converts an $f32$ tensor into $f16$ and cast32 performs the reverse process; r_sum represents the reduce_sum operator.

<table>
<thead>
<tr>
<th>no.</th>
<th>input config.</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>op5</th>
<th>op6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$f32$ [64,2]</td>
<td>cast16</td>
<td>cast32</td>
<td>cast16</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$f32$ [1280,21128]</td>
<td>cast16</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>$f16$ [64,768]</td>
<td>cast32</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>$f32$ [1280,21128]</td>
<td>mul</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>$f32$ [1280]</td>
<td>neg</td>
<td>mul</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>$f32$ [3072]</td>
<td>mul</td>
<td>mul</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>$f32$ [64,128,768]</td>
<td>add</td>
<td>mul</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>$f32$ [64,128,768]</td>
<td>add</td>
<td>mul</td>
<td>r_sum</td>
<td>add</td>
<td>mul</td>
<td>r_sum</td>
</tr>
<tr>
<td>9</td>
<td>$f32$ [8192,768]</td>
<td>r_sum</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>$f16$ [64,128,12,64]</td>
<td>reshape</td>
<td>cast32</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>$f16$ [64,128,768]</td>
<td>reshape</td>
<td>cast32</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>$f16$ [64,20]</td>
<td>reshape</td>
<td>r_sum</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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
While not exploiting fusion, cuDNN does not support *type casting* or *reshaping* operators; Panamera exhibits a better scalability 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

While not exploiting fusion, cuDNN does not support type casting or reshaping operators; PANAMERA exhibits a better scalability 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 $9.5 \times$, $2.6 \times$ and $2.7 \times$, respectively.
### Execution time in milliseconds (GPT-3 is executed on a Tesla A100 GPU)

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

---


# Results of end-to-end workloads

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

<table>
<thead>
<tr>
<th>Workloads</th>
<th>MindSpore</th>
<th>TVM</th>
<th>Ansor</th>
<th>AKG</th>
<th>PANAMERA</th>
<th>Improvement over</th>
<th>number of fused ops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MindSpore</td>
<td>TVM</td>
</tr>
<tr>
<td>BERT$^1$</td>
<td>352.2</td>
<td>138.0</td>
<td>120.3</td>
<td>124.0</td>
<td>111.0</td>
<td>+217%</td>
<td>+24%</td>
</tr>
<tr>
<td>Wide&amp;Deep$^2$</td>
<td>22.4</td>
<td>12.5</td>
<td>12.8</td>
<td>12.6</td>
<td>11.0</td>
<td>+104%</td>
<td>+14%</td>
</tr>
<tr>
<td>VGG-16$^3$</td>
<td>70.4</td>
<td>65.7</td>
<td>66.3</td>
<td>67.6</td>
<td>64.2</td>
<td>+10%</td>
<td>+2%</td>
</tr>
<tr>
<td>MobileNet-v3$^4$</td>
<td>151.4</td>
<td>133.0</td>
<td>129.4</td>
<td>136.8</td>
<td>131.5</td>
<td>+15%</td>
<td>+1%</td>
</tr>
<tr>
<td>Transformer$^5$</td>
<td>157.8</td>
<td>132.4</td>
<td>126.5</td>
<td>136.8</td>
<td>79.2</td>
<td>+99%</td>
<td>+67%</td>
</tr>
<tr>
<td>GPT-3$^6$</td>
<td>483.0</td>
<td>133.9</td>
<td>131.3</td>
<td>146.2</td>
<td>123.7</td>
<td>+290%</td>
<td>+8%</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+122.5%</td>
<td>+19.3%</td>
</tr>
</tbody>
</table>

- number of operators fused by **PANAMERA** in a workload

---


## Results of end-to-end workloads

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

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

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

---

## Results of end-to-end workloads

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

<table>
<thead>
<tr>
<th>Workloads</th>
<th>MindSpore</th>
<th>TVM</th>
<th>Ansor</th>
<th>AKG</th>
<th>PANAMERA</th>
<th>Improvement over</th>
<th>AKG</th>
<th>number of fused ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT(^1)</td>
<td>352.2</td>
<td>138.0</td>
<td>120.3</td>
<td>124.0</td>
<td>111.0</td>
<td>+217%</td>
<td>+12%</td>
<td>304</td>
</tr>
<tr>
<td>Wide&amp;Deep(^2)</td>
<td>22.4</td>
<td>12.5</td>
<td>12.8</td>
<td>12.6</td>
<td>11.0</td>
<td>+104%</td>
<td>+15%</td>
<td>74</td>
</tr>
<tr>
<td>VGG-16(^3)</td>
<td>70.4</td>
<td>65.7</td>
<td>66.3</td>
<td>67.6</td>
<td>64.2</td>
<td>+10%</td>
<td>+5%</td>
<td>39</td>
</tr>
<tr>
<td>MobileNet-v3(^4)</td>
<td>151.4</td>
<td>133.0</td>
<td>129.4</td>
<td>136.8</td>
<td>131.5</td>
<td>+15%</td>
<td>+4%</td>
<td>52</td>
</tr>
<tr>
<td>Transformer(^5)</td>
<td>157.8</td>
<td>132.4</td>
<td>126.5</td>
<td>136.8</td>
<td>79.2</td>
<td>+99%</td>
<td>+73%</td>
<td>746</td>
</tr>
<tr>
<td>GPT-3(^6)</td>
<td>483.0</td>
<td>133.9</td>
<td>131.3</td>
<td>146.2</td>
<td>123.7</td>
<td>+290%</td>
<td>+18%</td>
<td>409</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+122.5%</td>
<td>+15.2%</td>
<td></td>
</tr>
</tbody>
</table>

- 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

---


\(^5\) Ashish Vaswani et al. “Attention is All You Need”. NIPS’17, pp. 6000–6010.

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.
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.

<table>
<thead>
<tr>
<th>$MNK$ shape</th>
<th>$K$-dim config</th>
<th>Panamera</th>
<th>Tensor cores</th>
<th>Matching percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 32 \times 64$</td>
<td>2 blocks</td>
<td>24.044</td>
<td>4.381</td>
<td>18.22%</td>
</tr>
<tr>
<td>$128 \times 32 \times 1024$</td>
<td>16 blocks</td>
<td>21.378</td>
<td>57.882</td>
<td>270.75%</td>
</tr>
<tr>
<td>$1024 \times 512 \times 1024$</td>
<td>16 blocks</td>
<td>183.18</td>
<td>78.623</td>
<td>42.92%</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>MNK shape</th>
<th>K-dim config</th>
<th>Panamera</th>
<th>Tensor cores</th>
<th>Matching percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 × 32 × 64</td>
<td>2 blocks</td>
<td>24.044</td>
<td>4.381</td>
<td>18.22%</td>
</tr>
<tr>
<td>128 × 32 × 1024</td>
<td>16 blocks</td>
<td>21.378</td>
<td>57.882</td>
<td>270.75%</td>
</tr>
<tr>
<td>1024 × 512 × 1024</td>
<td>16 blocks</td>
<td>183.18</td>
<td>78.623</td>
<td>42.92%</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>$MNK$ shape</th>
<th>$K$-dim config</th>
<th>Panamera</th>
<th>Tensor Cores</th>
<th>Matching Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 32 \times 64$</td>
<td>2 blocks</td>
<td>24.044</td>
<td>4.381</td>
<td>18.22%</td>
</tr>
<tr>
<td>$128 \times 32 \times 1024$</td>
<td>16 blocks</td>
<td>21.378</td>
<td>57.882</td>
<td>270.75%</td>
</tr>
<tr>
<td>$1024 \times 512 \times 1024$</td>
<td>16 blocks</td>
<td>183.18</td>
<td>78.623</td>
<td>42.92%</td>
</tr>
</tbody>
</table>

- 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.

Thank you!

Any Questions?