# Effectively Scheduling Computational Graphs of Deep Neural Networks toward Their Domain-Specific Accelerators

Jie Zhao[1] Siyuan Feng[2] Xiaoqiang Dan[3]

Fei Liu[3] Chengke Wang[3] Sheng Yuan[3] Wenyuan Lv[3] Qikai Xie[3]

[1]Information Engineering University, Zhengzhou

[2]Shanghai Jiao Tong University, Shanghai

[3]Stream Computing Inc., Hangzhou

*The 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*

July 12, 2023, Boston, MA, USA

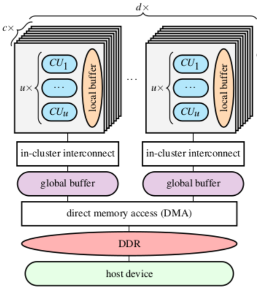
# Outline

# A Deep Neural Network (DNN) DSA Abstraction

- Moore's Law $\downarrow$ $\rightsquigarrow$ Domain-specific Architecture (DSA) $\uparrow$

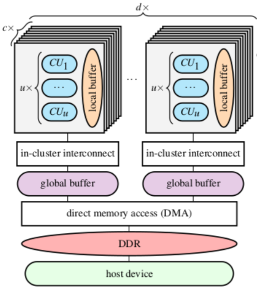
# A Deep Neural Network (DNN) DSA Abstraction

- Moore's Law $\downarrow \rightsquigarrow$ Domain-specific Architecture (DSA) $\uparrow$
- A DSA Abstraction has formed after several years of investigations



# A Deep Neural Network (DNN) DSA Abstraction

- Moore's Law $\downarrow \rightsquigarrow$ Domain-specific Architecture (DSA) $\uparrow$
- A DSA Abstraction has formed after several years of investigations



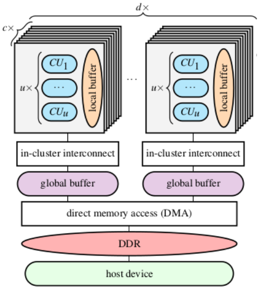

$$\text{Goya}\begin{cases} d \leftarrow 1; c \leftarrow 9; u \leftarrow 1 \\ LB \leftarrow \text{Local Memory or N/A} \\ GB \leftarrow \text{Shared Memory} \\ CU_1 \leftarrow \text{GEMM engine/TPC} \end{cases}$$

$$\text{Ascend}\begin{cases} d \leftarrow 1; c \leftarrow 8; u \leftarrow 3 \\ LB \leftarrow \text{Unified/L1 Buffer} \\ GB \leftarrow \text{on-chip Buffer} \\ CU_1 \leftarrow \text{scalar unit} \\ CU_2 \leftarrow \text{vector unit} \\ CU_3 \leftarrow \text{cube unit} \end{cases}$$

$$\text{IPU}\begin{cases} d \leftarrow 2; c \leftarrow 1216; u \leftarrow 1 \\ LB \leftarrow \text{Local Memory} \\ GB \leftarrow \text{N/A} \\ CU_1 \leftarrow \text{core} \end{cases}$$

# A Deep Neural Network (DNN) DSA Abstraction

- Moore's Law $\downarrow \rightsquigarrow$ Domain-specific Architecture (DSA) $\uparrow$
- A DSA Abstraction has formed after several years of investigations




$$\text{Goya}\begin{cases} d \leftarrow 1;\, c \leftarrow 9;\, u \leftarrow 1 \\ LB \leftarrow \text{Local Memory or N/A} \\ GB \leftarrow \text{Shared Memory} \\ CU_1 \leftarrow \text{GEMM engine/TPC} \end{cases}$$

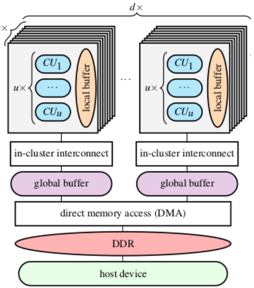
$$\text{Ascend}\begin{cases} d \leftarrow 1;\, c \leftarrow 8;\, u \leftarrow 3 \\ LB \leftarrow \text{Unified/L1 Buffer} \\ GB \leftarrow \text{on-chip Buffer} \\ CU_1 \leftarrow \text{scalar unit} \\ CU_2 \leftarrow \text{vector unit} \\ CU_3 \leftarrow \text{cube unit} \end{cases}$$

$$\text{IPU}\begin{cases} d \leftarrow 2;\, c \leftarrow 1216;\, u \leftarrow 1 \\ LB \leftarrow \text{Local Memory} \\ GB \leftarrow \text{N/A} \\ CU_1 \leftarrow \text{core} \end{cases}$$

- Scheduling DNNs for this DSA abstraction is thus important!

# A Deep Neural Network (DNN) DSA Abstraction

- Moore's Law $\downarrow \rightsquigarrow$ Domain-specific Architecture (DSA) $\uparrow$
- A DSA Abstraction has formed after several years of investigations



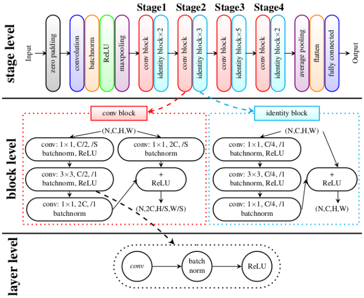

Goya $\begin{cases} d \leftarrow 1; c \leftarrow 9; u \leftarrow 1 \\ LB \leftarrow \text{Local Memory or N/A} \\ GB \leftarrow \text{Shared Memory} \\ CU_1 \leftarrow \text{GEMM engine/TPC} \end{cases}$

Ascend $\begin{cases} d \leftarrow 1; c \leftarrow 8; u \leftarrow 3 \\ LB \leftarrow \text{Unified/L1 Buffer} \\ GB \leftarrow \text{on-chip Buffer} \\ CU_1 \leftarrow \text{scalar unit} \\ CU_2 \leftarrow \text{vector unit} \\ CU_3 \leftarrow \text{cube unit} \end{cases}$

IPU $\begin{cases} d \leftarrow 2; c \leftarrow 1216; u \leftarrow 1 \\ LB \leftarrow \text{Local Memory} \\ GB \leftarrow \text{N/A} \\ CU_1 \leftarrow \text{core} \end{cases}$

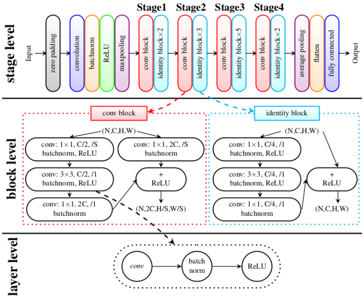
- Scheduling DNNs for this DSA abstraction is thus important!
- But existing approaches cannot fully exploit its computing power...

# Limitations of Prior Work




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

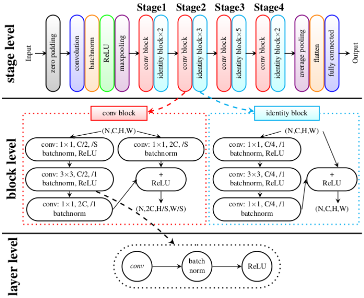
# Limitations of Prior Work




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

- Prior work groups nodes by obscuring hardware architectures, producing more kernels and requiring more in-between, off-core data movements;

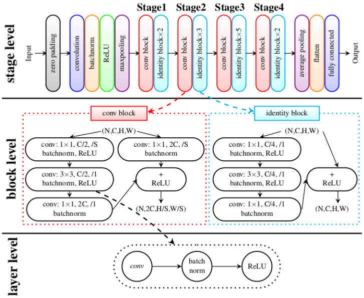
# Limitations of Prior Work




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

- Prior work groups nodes by obscuring hardware architectures, producing more kernels and requiring more in-between, off-core data movements;
- Grouping nodes within a layer generates fine-grained sub-graphs, missing the across-layer instruction scheduling opportunities;

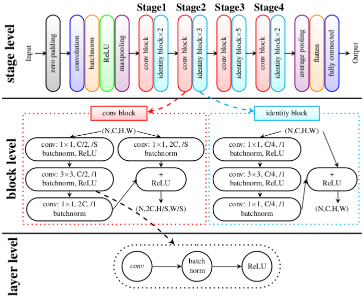
# Limitations of Prior Work




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

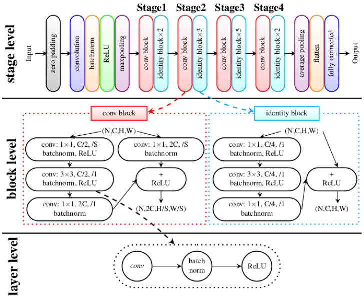
- Prior work groups nodes by obscuring hardware architectures, producing more kernels and requiring more in-between, off-core data movements;
- Grouping nodes within a layer generates fine-grained sub-graphs, missing the across-layer instruction scheduling opportunities;
- Prior work did not expose/exploit the imbalanced memory usage distribution[1], under-utilizing the faster local memory.

[1] Ji Lin et al. "Memory-efficient Patch-based Inference for Tiny Deep Learning". *NeurIPS*. vol. 34. 2021, pp. 1–13.




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

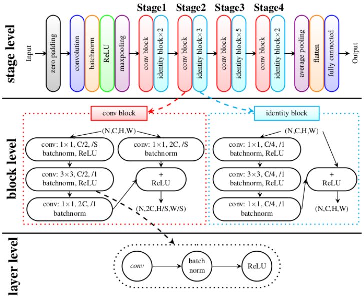
# Our Solution




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

- Construct coarser-grained sub-graphs, generating larger kernels and coverting data movements from off-core to on-core;

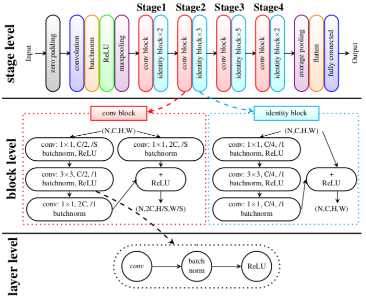
# Our Solution




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

- Construct coarser-grained sub-graphs, generating larger kernels and coverting data movements from off-core to on-core;
- Sub-graphs should cover layers or blocks, better hiding memory latency and exploiting the parallelism across CUs;

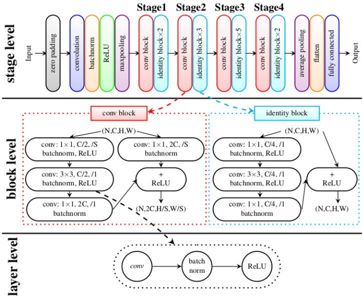
# Our Solution




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

- Construct coarser-grained sub-graphs, generating larger kernels and coverting data movements from off-core to on-core;
- Sub-graphs should cover layers or blocks, better hiding memory latency and exploiting the parallelism across CUs;
- Consider the internal relations between coarser-grained sub-graphs, better utilizing the faster local memory.

# Our Solution




- layer: nodes connected in a straight line, with at most one containing parameters learner using gradients of loss.
- block: a layer or a group of layers used recursively
- stage: a logical, high-level abstraction used in a computational graph

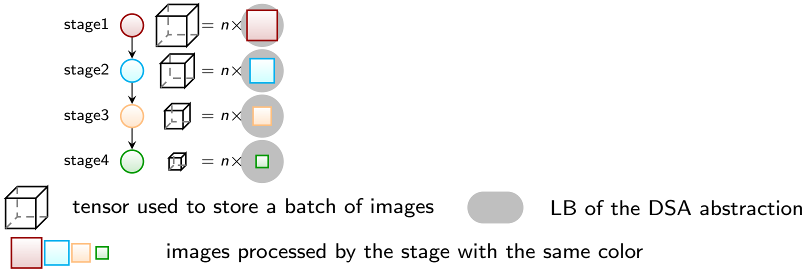
- Construct coarser-grained sub-graphs, generating larger kernels and coverting data movements from off-core to on-core;
- Sub-graphs should cover layers or blocks, better hiding memory latency and exploiting the parallelism across CUs;
- Consider the internal relations between coarser-grained sub-graphs, better utilizing the faster local memory.
- These solutions form our new scheduler for DSA – GraphTurbo.

# Core Idea of GraphTurbo

- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.

# Core Idea of GraphTurbo

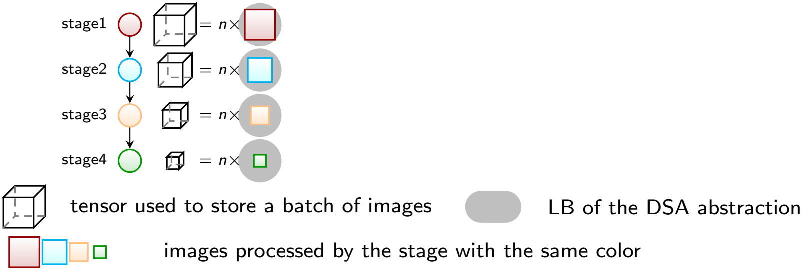
- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.




- Each cluster processes 8 images; each stage reduces an image by half.

# Core Idea of GraphTurbo

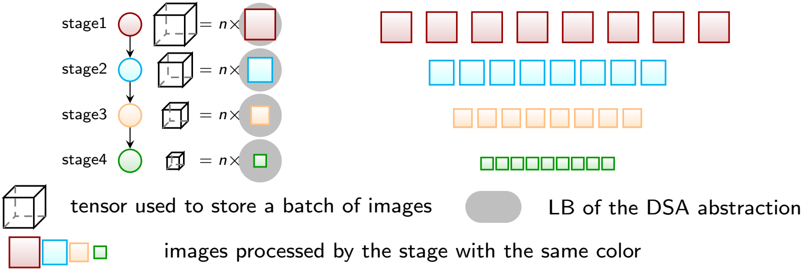
- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.




- Each cluster processes 8 images; each stage reduces an image by half.
- Construct larger sub-graph for each stage.

# Core Idea of GraphTurbo

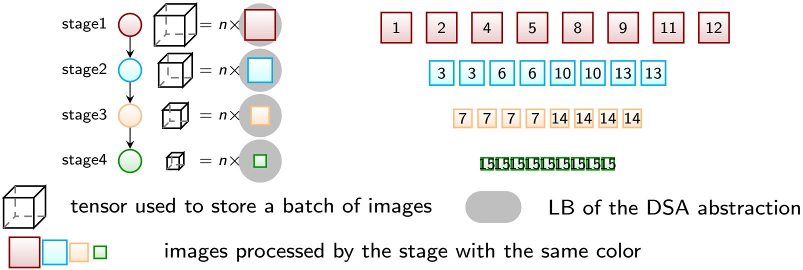
- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.




- Each cluster processes 8 images; each stage reduces an image by half.
- Construct larger sub-graph for each stage.
- Split each sub-graph into 8, 4, 2, and 1 instance(s), respectively.

# Core Idea of GraphTurbo

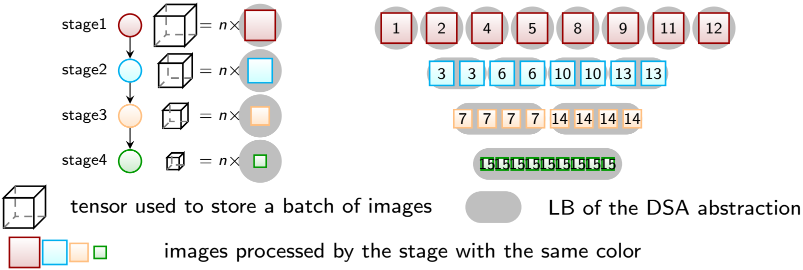
- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.




- Each cluster processes 8 images; each stage reduces an image by half.
- Construct larger sub-graph for each stage.
- Split each sub-graph into 8, 4, 2, and 1 instance(s), respectively.
- Schedule sub-graph instances in this order.

# Core Idea of GraphTurbo

- maximally preserve the input tensors in LB to convert as many off-core data movements as possible into on-core data exchanges.




- Each cluster processes 8 images; each stage reduces an image by half.
- Construct larger sub-graph for each stage.
- Split each sub-graph into 8, 4, 2, and 1 instance(s), respectively.
- Schedule sub-graph instances in this order.
- Saturate LB while exploiting the parallelism across cores.

# Collect Splitting Information

**Algorithm 1:** Compute `SplitInfo`

`SplitInfo` $\leftarrow \varnothing$;

**foreach** $d$ **in** $[1, \cdots, depth \leftarrow$ **dimof** $(output$ of $SG)]$ **do**

  $n_d \leftarrow 0$; $split_d \leftarrow 0$; $f_d \leftarrow \infty$;

  **foreach** $v$ **in** $[1, 2, 4, 8, 9, \cdots, size^{(d)}_{output}]$ **do**

    **if** $\lceil \frac{peak}{v} \rceil \leq$ **sizeof** (LB) **then**

      $n_d \leftarrow n_d + 1$; $split_d \leftarrow 1$; $f_d \leftarrow v$; **break**;

  **foreach** $t$ **in** $intermediates$ **do**

    **if** $split_d = 1$ **then**

      $n_d \leftarrow n_d$ + **num_of_op**$(t)$;

      **if** **match_dim**$(t, d)$ **and** $size^{(d)}_t \% f_d = 0$ **then**

        `SplitInfo` $\leftarrow$ `SplitInfo` $\cup \{(split_d, n_d, f_d, d)\}$;

# Collect Splitting Information

**Algorithm 1:** Compute `SplitInfo`

`SplitInfo` $\leftarrow \varnothing$;
**foreach** $d$ **in** $[1, \cdots, depth \leftarrow$ **dimof** $(output$ of $SG)]$ **do**
  $n_d \leftarrow 0$; $split_d \leftarrow 0$; $f_d \leftarrow \infty$;
  **foreach** $v$ **in** $[1, 2, 4, 8, 9, \cdots, size^{(d)}_{output}]$ **do**
    **if** $\lceil \frac{peak}{v} \rceil \leq$ **sizeof** (LB) **then**
      $n_d \leftarrow n_d + 1$; $split_d \leftarrow 1$; $f_d \leftarrow v$; **break**;
  **foreach** $t$ **in** $intermediates$ **do**
    **if** $split_d = 1$ **then**
      $n_d \leftarrow n_d +$ **num_of_op**$(t)$;
      **if** **match_dim**$(t, d)$ **and** $size^{(d)}_t \% f_d = 0$ **then**
        `SplitInfo` $\leftarrow$ `SplitInfo` $\cup \{(split_d, n_d, f_d, d)\}$;

- Collect hardware information for constructing larger sub-graphs.
- `SplitInfo` includes split loop dimension, factor, etc.

# Collect Splitting Information

**Algorithm 1:** Compute `SplitInfo`

`SplitInfo` $\leftarrow \varnothing$;
**foreach** $d$ **in** $[1, \cdots, depth \leftarrow$ **dimof** $(output$ of $SG)]$ **do**
  $n_d \leftarrow 0$; $split_d \leftarrow 0$; $f_d \leftarrow \infty$;
  **foreach** $v$ **in** $[1, 2, 4, 8, 9, \cdots, size^{(d)}_{output}]$ **do**
    **if** $\lceil \frac{peak}{v} \rceil \leq$ **sizeof** (LB) **then**
      $n_d \leftarrow n_d + 1$; $split_d \leftarrow 1$; $f_d \leftarrow v$; **break**;
  **foreach** $t$ **in** *intermediates* **do**
    **if** $split_d = 1$ **then**
      $n_d \leftarrow n_d +$ **num_of_op**$(t)$;
      **if match_dim**$(t, d)$ **and** $size^{(d)}_t \% f_d = 0$ **then**
        `SplitInfo` $\leftarrow$ `SplitInfo` $\cup \{(split_d, n_d, f_d, d)\}$;

- Collect hardware information for constructing larger sub-graphs.
- `SplitInfo` includes split loop dimension, factor, etc.
- Each sub-graph *SG* is initialized by an *op*.
- Each *op* include only one output tensor and multiple input tensors.

# Collect Splitting Information

**Algorithm 1:** Compute `SplitInfo`

`SplitInfo` $\leftarrow \varnothing$;
**foreach** $d$ **in** $[1, \cdots, depth \leftarrow$ **dimof** $(output$ of $SG)]$ **do**
    $n_d \leftarrow 0$; $split_d \leftarrow 0$; $f_d \leftarrow \infty$;
    **foreach** $v$ **in** $[1, 2, 4, 8, 9, \cdots, size^{(d)}_{output}]$ **do**
        **if** $\lceil \frac{peak}{v} \rceil \leq$ **sizeof** (LB) **then**
            $n_d \leftarrow n_d + 1$; $split_d \leftarrow 1$; $f_d \leftarrow v$; **break**;
    **foreach** $t$ **in** $intermediates$ **do**
        **if** $split_d = 1$ **then**
            $n_d \leftarrow n_d$ + **num_of_op**$(t)$;
            **if match_dim**$(t, d)$ **and** $size^{(d)}_t \% f_d = 0$ **then**
                `SplitInfo` $\leftarrow$ `SplitInfo` $\cup \{(split_d, n_d, f_d, d)\}$;

- Collect hardware information for constructing larger sub-graphs.
- `SplitInfo` includes split loop dimension, factor, etc.
- Each sub-graph *SG* is initialized by an *op*.
- Each *op* include only one output tensor and multiple input tensors.
- Compute `SplitInfo` for the output and propagate it to inputs.

# Collect Splitting Information

**Algorithm 1:** Compute `SplitInfo`

`SplitInfo` $\leftarrow \varnothing$;<br>
**foreach** $d$ **in** $[1, \cdots, depth \leftarrow$ **dimof** $(output$ of $SG)]$ **do**<br>
    $n_d \leftarrow 0$; $split_d \leftarrow 0$; $f_d \leftarrow \infty$;<br>
    **foreach** $v$ **in** $[1, 2, 4, 8, 9, \cdots, size^{(d)}_{output}]$ **do**<br>
        **if** $\lceil \frac{peak}{v} \rceil \leq$ **sizeof** (LB) **then**<br>
            $n_d \leftarrow n_d + 1$; $split_d \leftarrow 1$; $f_d \leftarrow v$; **break**;<br>
    **foreach** $t$ **in** *intermediates* **do**<br>
        **if** $split_d = 1$ **then**<br>
            $n_d \leftarrow n_d +$ **num_of_op**$(t)$;<br>
            **if match_dim**$(t, d)$ **and** $size^{(d)}_t \% f_d = 0$ **then**<br>
                `SplitInfo` $\leftarrow$ `SplitInfo` $\cup \{(split_d, n_d, f_d, d)\}$;

- Collect hardware information for constructing larger sub-graphs.
- `SplitInfo` includes split loop dimension, factor, etc.
- Each sub-graph *SG* is initialized by an *op*.
- Each *op* include only one output tensor and multiple input tensors.
- Compute `SplitInfo` for the output and propagate it to inputs.
- Define three metrics, and use

$$\text{lexmax}_{\forall d \in \texttt{SplitInfo}} (n_d, -f_d, -d)$$

to select the a loop dimension of a tensor to be split.

# Group Sub-graphs with the aid of SplitInfo

**Algorithm 2:** Group sub-graphs

$SG[1,\cdots,g] \leftarrow$ **topological_order** $(G)$; $b \leftarrow g$;
**foreach** $i$ **in** $[1,\cdots,g]$ **do**
    SplitInfo$[i] \leftarrow$ Algo.1 $(SG[i])$;
    BestSplit$[i] \leftarrow$ Eq. (1) $(SG[i]$, SplitInfo$[i])$;
**repeat**
    $\{G,s\} \leftarrow$ **straight_merge**$(SG[1,\cdots,b]$, SplitInfo$[1,\cdots,b])$;
    **foreach** $i$ **in** $[1,\cdots,s]$ **do**
        BestSplit$[i] \leftarrow$ Eq. (1) $(SG[i]$, SplitInfo$[i])$;
    $\{G,d\} \leftarrow$ **diamond_merge**$(SG[1,\cdots,s]$, SplitInfo$[1,\cdots,s])$;
    **foreach** $i$ **in** $[1,\cdots,d]$ **do**
        BestSplit$[i] \leftarrow$ Eq. (1) $(SG[i]$, SplitInfo$[i])$;
    $\{G,b\} \leftarrow$ **branch_merge**$(SG[1,\cdots,d]$, SplitInfo$[1,\cdots,d])$;
    **foreach** $i$ **in** $[1,\cdots,b]$ **do**
        BestSplit$[i] \leftarrow$ Eq. (1) $(SG[i]$, SplitInfo$[i])$;
**until** $s$, $d$ and $b$ all do not decrease;

# Group Sub-graphs with the aid of `SplitInfo`

**Algorithm 2:** Group sub-graphs

```
SG[1, ..., g] ← topological_order (G); b ← g;
foreach i in [1, ..., g] do
    SplitInfo[i] ← Algo.1 (SG[i]);
    BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
repeat
    {G, s} ← straight_merge(SG[1, ..., b], SplitInfo[1, ..., b]);
    foreach i in [1, ..., s] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
    {G, d} ← diamond_merge(SG[1, ..., s], SplitInfo[1, ..., s]);
    foreach i in [1, ..., d] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
    {G, b} ← branch_merge(SG[1, ..., d], SplitInfo[1, ..., d]);
    foreach i in [1, ..., b] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
until s, d and b all do not decrease;
```

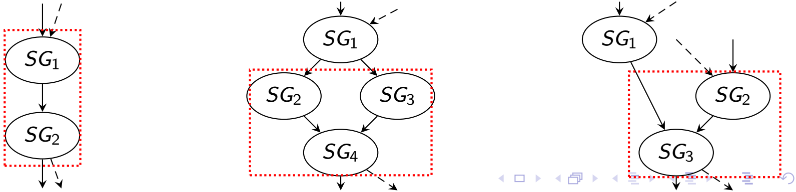
- Sort a graph $G$ in a topological order, each node denoting an $SG$.

# Group Sub-graphs with the aid of SplitInfo

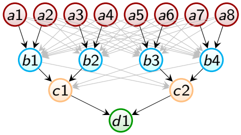
**Algorithm 2:** Group sub-graphs

```
SG[1,···,g] ←topological_order (G); b ← g;
foreach i in [1,···,g] do
    SplitInfo[i] ← Algo.1 (SG[i]);
    BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
repeat
    {G,s} ←straight_merge(SG[1,···,b], SplitInfo[1,···,b]);
    foreach i in [1,···,s] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
    {G,d} ←diamond_merge(SG[1,···,s], SplitInfo[1,···,s]);
    foreach i in [1,···,d] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
    {G,b} ←branch_merge(SG[1,···,d], SplitInfo[1,···,d]);
    foreach i in [1,···,b] do
        BestSplit[i] ← Eq. (1) (SG[i], SplitInfo[i]);
until s, d and b all do not decrease;
```

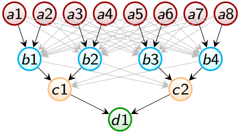
- Sort a graph $G$ in a topological order, each node denoting an $SG$.
- Group $SG$ by repeatedly considering three patterns



# Order Sub-graph Instances



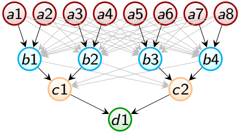
# Order Sub-graph Instances




- GraphTurbo uses either a BFS heuristic to order these sub-graph instances,

  a1 a2 a3 a4 a5 a6 a7 a8 b1 b2 b3 b4 c1 c2 d1

# Order Sub-graph Instances




**Algorithm 3:** Schedule Sub-graph Instances

```
visit ← get_subgraph_instances(G);
while visit ≠ ∅ do
    foreach indegree(SGI) = 0 in visit do
        ready ← ready.push(SGI); visit ← visit \ SGI;
    while ready ≠ ∅ do
        p ← sizeof(ready);
        while indegree(SGI_p) ≠ 0 do
            p ← p − 1;
        order ← order.push(SGI_p); ready ← ready \ SGI_p;
        foreach SGI in visit and ready do
            if SGI_p is a producer of SGI then
                remove_producer(SGI, SGI_p);
                indegree(SGI) ← indegree(SGI) − 1;
                ready ← ready.push(SGI); visit ← visit \ SGI;
```

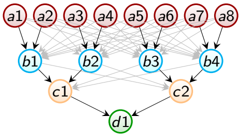
- GraphTurbo uses either a BFS heuristic to order these sub-graph instances,

  a1 a2 a3 a4 a5 a6 a7 a8 b1 b2 b3 b4 c1 c2 d1

- or a DFS heuristic, which simplifies the algorithmic design.

  a8 a7 b4 a6 a5 b3 c2 a4 a3 b2 a2 a1 b1 c1 d1

# Order Sub-graph Instances




**Algorithm 3:** Schedule Sub-graph Instances

```
visit ← get_subgraph_instances(G);
while visit ≠ ∅ do
    foreach indegree(SGI) = 0 in visit do
        ready ← ready.push(SGI); visit ← visit \ SGI;
    while ready ≠ ∅ do
        p ← sizeof(ready);
        while indegree(SGI_p) ≠ 0 do
            p ← p − 1;
        order ← order.push(SGI_p); ready ← ready \ SGI_p;
        foreach SGI in visit and ready do
            if SGI_p is a producer of SGI then
                remove_producer(SGI, SGI_p);
                indegree(SGI) ← indegree(SGI) − 1;
                ready ← ready.push(SGI); visit ← visit \ SGI;
```

- GraphTurbo uses either a BFS heuristic to order these sub-graph instances,

  a1 a2 a3 a4 a5 a6 a7 a8 b1 b2 b3 b4 c1 c2 d1

- or a DFS heuristic, which simplifies the algorithmic design.

  a8 a7 b4 a6 a5 b3 c2 a4 a3 b2 a2 a1 b1 c1 d1

- An ILP-based heuristic is under construction and will be released soon.

# Infer Core Binding and Buffer Scopes

**Algorithm 4:** Infer Core Binding and Buffer Scopes

```
visit ← DFS_visit_reverse_order(O); size ← sizeof(visit);
bind[1, ··· ,size] ← {[]}; scope[1, ··· ,size] ← {LB};
foreach i in [1, ··· ,size] do
    if bind[i] = [] or scope[i] ≠ LB then
        bind[i] ← plain_binding (output of visit[i]);
        if infer_binding (bind[i]) = [] or is invalid then
            continue;
        foreach producer[j] in visit do
            if bind[j] = [] then
                bind[j] ← infer_binding (bind[i]);
            else if bind[j] ≠ infer_binding (bind[i]) then
                scope[j] ← GB;
            else
                continue;
    else
        bind[i] ← update_binding (bind[i]) uses more cores than
          plain_binding (output of visit[i]) ? update_binding
          (bind[i]) : plain_binding (output of visit[i]);
```

# Infer Core Binding and Buffer Scopes

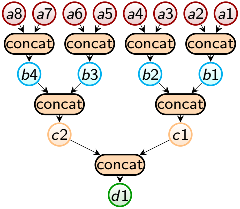
**Algorithm 4:** Infer Core Binding and Buffer Scopes

```
visit ← DFS_visit_reverse_order(O); size ← sizeof(visit);
bind[1, ··· ,size] ← {[]}; scope[1, ··· ,size] ← {LB};
foreach i in [1, ··· ,size] do
    if bind[i] = [] or scope[i] ≠ LB then
        bind[i] ← plain_binding (output of visit[i]);
        if infer_binding (bind[i]) = [] or is invalid then
            continue;
        foreach producer[j] in visit do
            if bind[j] = [] then
                bind[j] ← infer_binding (bind[i]);
            else if bind[j] ≠ infer_binding (bind[i]) then
                scope[j] ← GB;
            else
                continue;
    else
        bind[i] ← update_binding (bind[i]) uses more cores than
          plain_binding (output of visit[i]) ? update_binding
          (bind[i]) : plain_binding (output of visit[i]);
```

- Visit the scheduling result of sub-graph instances in a reverse order.
- Either initialize binding information using a plain strategy and the buffer scope using LB,
- or infer the binding strategy from the output tensor.
- A better strategy is selected if both inferred and initialized binding information exist.

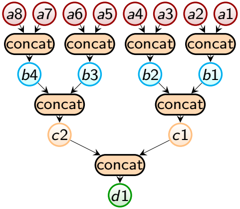
# Concatenate the Outputs of Sub-graph Instances

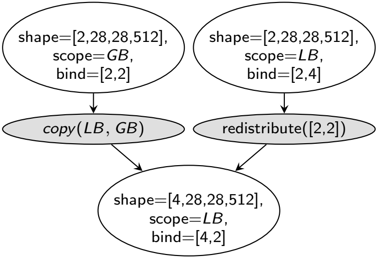
- Detect fine-grained dependencies between sub-graph instances and introduce a lightweight concatenation *op* when necessary.



# Concatenate the Outputs of Sub-graph Instances

- Detect fine-grained dependencies between sub-graph instances and introduce a lightweight concatenation *op* when necessary.

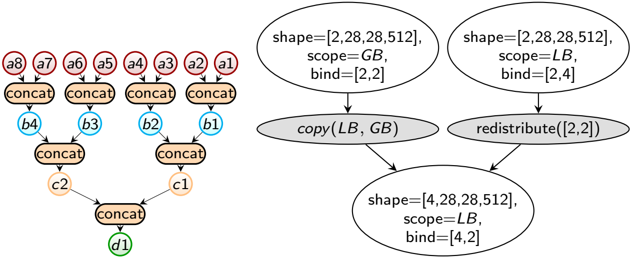



- Insert additional *op*s, e.g., copy, redistribute, for moving data across the memory hierarchy if the binding strategies and memory scopes of a concatenation *op* are different from each other.

# Concatenate the Outputs of Sub-graph Instances

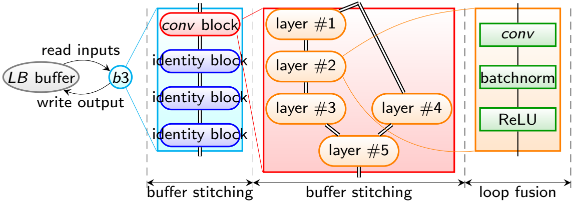
- Detect fine-grained dependencies between sub-graph instances and introduce a lightweight concatenation *op* when necessary.




- Insert additional *op*s, e.g., copy, redistribute, for moving data across the memory hierarchy if the binding strategies and memory scopes of a concatenation *op* are different from each other.
- How the approach is generalied to handle a sub-graph of multiple output tensors and other cases is discussed in the paper.

# Loop Fusion within Layers

- Generate one kernel for a sub-graph instance by expanding it as



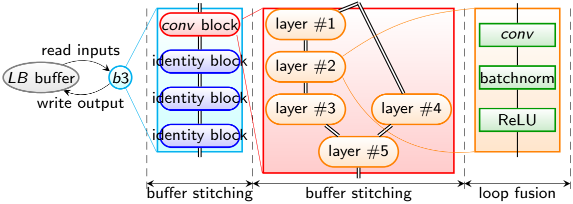

buffer stitching is performed between the components connected by

loop fusion is performed between the components connected by

an *op* that can be expressed using loop nests of arithmetic operations

# Loop Fusion within Layers

- Generate one kernel for a sub-graph instance by expanding it as

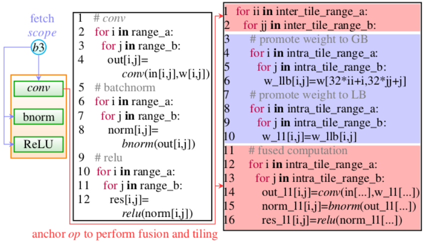



buffer stitching is performed between the components connected by

loop fusion is performed between the components connected by

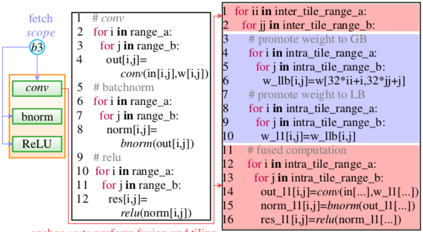
an *op* that can be expressed using loop nests of arithmetic operations

- Perform loop fusion within each layer



# Buffer Stitching across Layers/Blocks

- Remain the outputs of a layer in LB, e.g., *res_l*1, instead of spilling it to slower global memory
- Consider both compute- and memory-intensive *ops*.



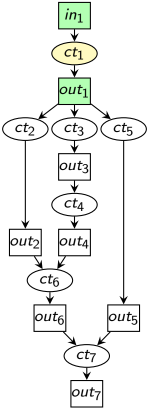

anchor *op* to perform fusion and tiling

# Memory Allocation and Reuse

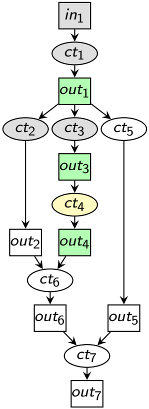
- Release the space consumed by an output tensor as early as possible.

# Memory Allocation and Reuse

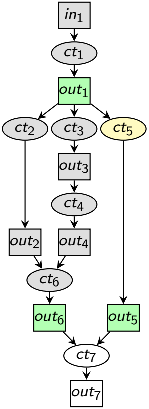
- Release the space consumed by an output tensor as early as possible.
- The space with the logest liveness across multiple computation tasks is first spilled in case LB cannot hold all tensors.



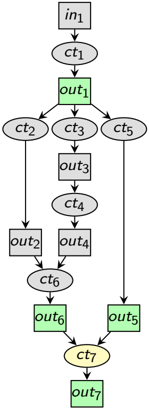

Execute $ct_1$.




Execute $ct_4$.



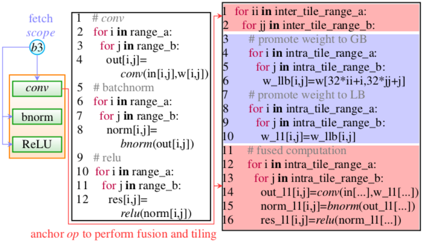

Execute $ct_5$.




Execute $ct_7$.

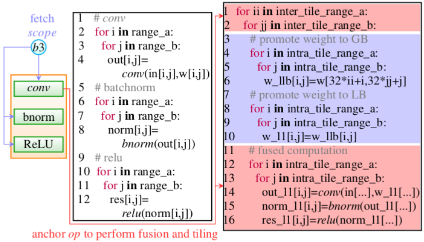
# Across-layer Instruction Scheduling

- Weight tensors can be promoted as early as possible.

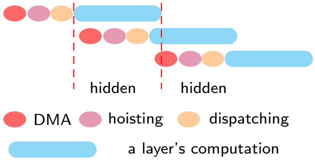


# Across-layer Instruction Scheduling

- Weight tensors can be promoted as early as possible.

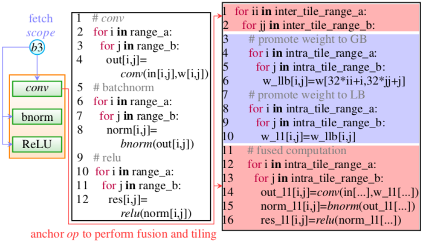



- The latency of these promotion statements behind computation tasks.

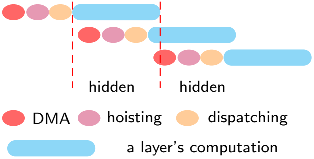


# Across-layer Instruction Scheduling

- Weight tensors can be promoted as early as possible.



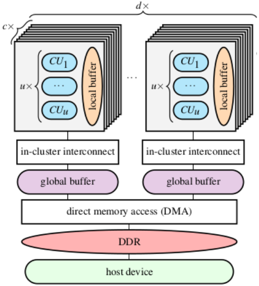

- The latency of these promotion statements behind computation tasks.




- Enable across-layer memory latency hiding.

# Environments and Setup

- The experiment platform is STCP920[1]

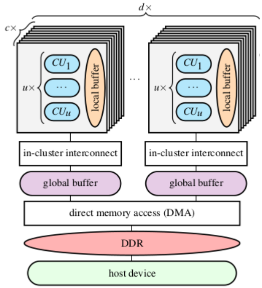



$$\begin{cases} d \leftarrow 4;\, c \leftarrow 8;\, u \leftarrow 3 \\ LB \leftarrow \text{64 KB L1} \\ GB \leftarrow \text{8MB last local buffer (LLB)} \\ CU_1 \leftarrow \text{vector core; } CU_2 \leftarrow \text{VME; } CU_3 \leftarrow \text{MME} \end{cases}$$

[1] Rongkai Zhan et al. "NeuralScale: A RISC-V Based Neural Processor Boosting AI Inference in Clouds". *Fifth Workshop on Computer Architecture Research with RISC-V*. CARRV. Virtual, 2021.

# Environments and Setup

- The experiment platform is STCP920[1]




$$\begin{cases} d \leftarrow 4; c \leftarrow 8; u \leftarrow 3 \\ LB \leftarrow \text{64 KB L1} \\ GB \leftarrow \text{8MB last local buffer (LLB)} \\ CU_1 \leftarrow \text{vector core}; CU_2 \leftarrow \text{VME}; CU_3 \leftarrow \text{MME} \end{cases}$$

- DNN models: ResNet-50 v1.5, BERT, DLRM, MobileNet v2, Vision_Transformer, DenseNet, Conformer
- DNN frameworks: Pytorch v1.81.1 for DLRM, and TensorFlow v1.13 for all others
- Compare with TVM, AStitch, and a vendor-crafted implementation

[1] Rongkai Zhan et al. "NeuralScale: A RISC-V Based Neural Processor Boosting AI Inference in Clouds". *Fifth Workshop on Computer Architecture Research with RISC-V*. CARRV. Virtual, 2021.

# Performance Comparison

- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

# Performance Comparison

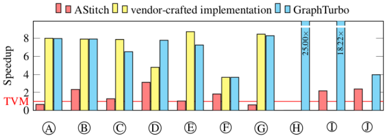
- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

| label | model | batch size | batches per cluster TVM | batches per cluster GraphTurbo | throughput unit | TVM's result |
|---|---|---|---|---|---|---|
| Ⓐ | ResNet-50 | 64 | 2 | 16 | images/s | 1064 |
| Ⓑ | BERT-128 | 32 | 4 | 8 | sentences/s | 512 |
| Ⓒ | BERT-256 | 16 | 2 | 4 | sentences/s | 412 |
| Ⓓ | BERT-384 | 8 | 1 | 2 | sentences/s | 36 |
| Ⓔ | BERT-512 | 8 | 1 | 2 | sentences/s | 324 |
| Ⓕ | DLRM | 1024 | 64 | 256 | queries/s | 131000 |
| Ⓖ | MobileNet-v2 | 128 | 2 | 32 | images/s | 1416 |
| Ⓗ | Vision_Transformer | 32 | 4 | 8 | images/s | 40 |
| Ⓘ | DenseNet | 32 | 4 | 8 | images/s | 456 |
| Ⓙ | Conformer | 12 | 1 | 3 | sentences/s | 184 |

# Performance Comparison

- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

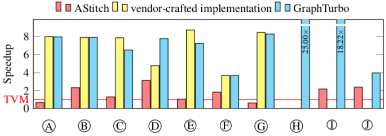
| label | model | batch size | batches per cluster | | throughput unit | TVM's result |
|---|---|---|---|---|---|---|
| | | | TVM | GraphTurbo | | |
| Ⓐ | ResNet-50 | 64 | 2 | 16 | images/s | 1064 |
| Ⓑ | BERT-128 | 32 | 4 | 8 | sentences/s | 512 |
| Ⓒ | BERT-256 | 16 | 2 | 4 | sentences/s | 412 |
| Ⓓ | BERT-384 | 8 | 1 | 2 | sentences/s | 36 |
| Ⓔ | BERT-512 | 8 | 1 | 2 | sentences/s | 324 |
| Ⓕ | DLRM | 1024 | 64 | 256 | queries/s | 131000 |
| Ⓖ | MobileNet-v2 | 128 | 2 | 32 | images/s | 1416 |
| Ⓗ | Vision_Transformer | 32 | 4 | 8 | images/s | 40 |
| Ⓘ | DenseNet | 32 | 4 | 8 | images/s | 456 |
| Ⓙ | Conformer | 12 | 1 | 3 | sentences/s | 184 |



# Performance Comparison

- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

| label | model | batch size | batches per cluster | | throughput unit | TVM's result |
|---|---|---|---|---|---|---|
| | | | TVM | GraphTurbo | | |
| A | ResNet-50 | 64 | 2 | 16 | images/s | 1064 |
| B | BERT-128 | 32 | 4 | 8 | sentences/s | 512 |
| C | BERT-256 | 16 | 2 | 4 | sentences/s | 412 |
| D | BERT-384 | 8 | 1 | 2 | sentences/s | 36 |
| E | BERT-512 | 8 | 1 | 2 | sentences/s | 324 |
| F | DLRM | 1024 | 64 | 256 | queries/s | 131000 |
| G | MobileNet-v2 | 128 | 2 | 32 | images/s | 1416 |
| H | Vision_Transformer | 32 | 4 | 8 | images/s | 40 |
| I | DenseNet | 32 | 4 | 8 | images/s | 456 |
| J | Conformer | 12 | 1 | 3 | sentences/s | 184 |

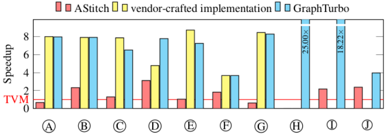



- TVM fuses *ops* within a sub-graph, producing kernels that exchange data via DDR.
- AStitch neither orders sub-graph instances nor considers compute-intensive *ops*.

# Performance Comparison

- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

| label | model | batch size | batches per cluster | | throughput unit | TVM's result |
|---|---|---|---|---|---|---|
| | | | TVM | GraphTurbo | | |
| Ⓐ | ResNet-50 | 64 | 2 | 16 | images/s | 1064 |
| Ⓑ | BERT-128 | 32 | 4 | 8 | sentences/s | 512 |
| Ⓒ | BERT-256 | 16 | 2 | 4 | sentences/s | 412 |
| Ⓓ | BERT-384 | 8 | 1 | 2 | sentences/s | 36 |
| Ⓔ | BERT-512 | 8 | 1 | 2 | sentences/s | 324 |
| Ⓕ | DLRM | 1024 | 64 | 256 | queries/s | 131000 |
| Ⓖ | MobileNet-v2 | 128 | 2 | 32 | images/s | 1416 |
| Ⓗ | Vision_Transformer | 32 | 4 | 8 | images/s | 40 |
| Ⓘ | DenseNet | 32 | 4 | 8 | images/s | 456 |
| Ⓙ | Conformer | 12 | 1 | 3 | sentences/s | 184 |

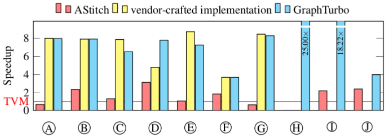



- TVM fuses *ops* within a sub-graph, producing kernels that exchange data via DDR.
- AStitch neither orders sub-graph instances nor considers compute-intensive *ops*.
- On average, GraphTurbo outperforms TVM by 11.15×, AStitch by 6.16×, and the vendor-crafted implementation by 1.04×.

# Performance Comparison

- We report the performance by selecting the optimal numbers of batches per cluster.
- How these optimal numbers are selected is discussed in the paper.

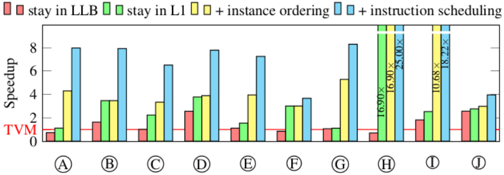
| label | model | batch size | batches per cluster | | throughput unit | TVM's result |
|---|---|---|---|---|---|---|
| | | | TVM | GraphTurbo | | |
| Ⓐ | ResNet-50 | 64 | 2 | 16 | images/s | 1064 |
| Ⓑ | BERT-128 | 32 | 4 | 8 | sentences/s | 512 |
| Ⓒ | BERT-256 | 16 | 2 | 4 | sentences/s | 412 |
| Ⓓ | BERT-384 | 8 | 1 | 2 | sentences/s | 36 |
| Ⓔ | BERT-512 | 8 | 1 | 2 | sentences/s | 324 |
| Ⓕ | DLRM | 1024 | 64 | 256 | queries/s | 131000 |
| Ⓖ | MobileNet-v2 | 128 | 2 | 32 | images/s | 1416 |
| Ⓗ | Vision_Transformer | 32 | 4 | 8 | images/s | 40 |
| Ⓘ | DenseNet | 32 | 4 | 8 | images/s | 456 |
| Ⓙ | Conformer | 12 | 1 | 3 | sentences/s | 184 |




- TVM fuses *ops* within a sub-graph, producing kernels that exchange data via DDR.
- AStitch neither orders sub-graph instances nor considers compute-intensive *ops*.
- On average, GraphTurbo outperforms TVM by 11.15×, AStitch by 6.16×, and the vendor-crafted implementation by 1.04×.
- Compilation overhead of different approaches is reported in the paper.

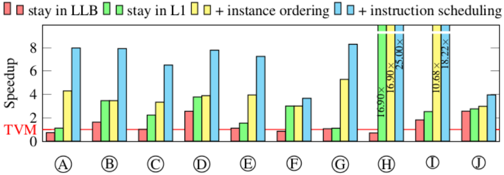
# Performance Breakdown

- Evaluate how different factors of GraphTurbo contribute to the overall speedup using four variants:



## Performance Breakdown

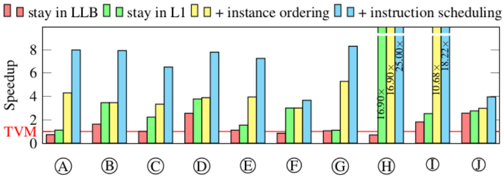
- Evaluate how different factors of GraphTurbo contribute to the overall speedup using four variants:




- Variant 1: maximally keeps outputs in LLB

# Performance Breakdown

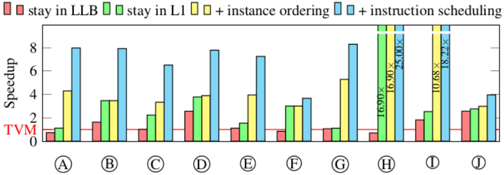
- Evaluate how different factors of GraphTurbo contribute to the overall speedup using four variants:




- Variant 1: maximally keeps outputs in LLB
- Variant 2: maximally keeps outputs in L1; outperforms Variant 1 by 3.67×. (demonstrating the importance of utilizing L1, i.e., the LB of the DSA abstraction)

# Performance Breakdown

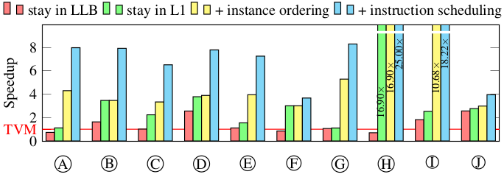
- Evaluate how different factors of GraphTurbo contribute to the overall speedup using four variants:




- Variant 1: maximally keeps outputs in LLB
- Variant 2: maximally keeps outputs in L1; outperforms Variant 1 by 3.67×. (demonstrating the importance of utilizing L1, i.e., the LB of the DSA abstraction)
- Variant 3: Variant 2 + schedule sub-graph instance; outperforms Variant 1 by 2.20×.

# Performance Breakdown

- Evaluate how different factors of GraphTurbo contribute to the overall speedup using four variants:




- Variant 1: maximally keeps outputs in LLB
- Variant 2: maximally keeps outputs in L1; outperforms Variant 1 by 3.67×. (demonstrating the importance of utilizing L1, i.e., the LB of the DSA abstraction)
- Variant 3: Variant 2 + schedule sub-graph instance; outperforms Variant 1 by 2.20×.
- Variant 4: Variant 3 + across-layer instruction scheduling; outperforms Variant 1 by 1.72×.

# Hardware Utilization

- We report the frequencies of each memory level.

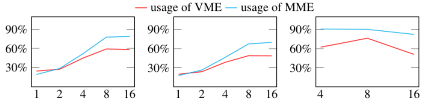
| label | DDR | | | LLB | | | L1 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo |
| Ⓐ | 58 | 1 | 1 | 0 | 11 | 11 | 0 | 291 | 284 |
| Ⓑ | 242 | 2 | 1 | 0 | 0 | 0 | 0 | 304 | 305 |
| Ⓒ | 242 | 2 | 1 | 0 | 25 | 110 | 0 | 401 | 240 |
| Ⓓ | 515 | 2 | 1 | 0 | 49 | 75 | 0 | 968 | 967 |
| Ⓔ | 242 | 2 | 1 | 0 | 25 | 76 | 0 | 474 | 337 |
| Ⓕ | 76 | 1 | 0 | 0 | 0 | 0 | 0 | 75 | 76 |
| Ⓖ | 56 | 1 | 0 | 0 | 7 | 3 | 0 | 619 | 608 |
| Ⓗ | 214 | - | 24 | 0 | - | 60 | 0 | - | 340 |
| Ⓘ | 247 | - | 0 | 0 | - | 3 | 0 | - | 389 |
| Ⓙ | 1054 | - | 4 | 0 | - | 813 | 0 | - | 250 |

# Hardware Utilization

- We report the frequencies of each memory level.

| label | DDR | | | LLB | | | L1 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo |
| A | 58 | 1 | 1 | 0 | 11 | 11 | 0 | 291 | 284 |
| B | 242 | 2 | 1 | 0 | 0 | 0 | 0 | 304 | 305 |
| C | 242 | 2 | 1 | 0 | 25 | 110 | 0 | 401 | 240 |
| D | 515 | 2 | 1 | 0 | 49 | 75 | 0 | 968 | 967 |
| E | 242 | 2 | 1 | 0 | 25 | 76 | 0 | 474 | 337 |
| F | 76 | 1 | 0 | 0 | 0 | 0 | 0 | 75 | 76 |
| G | 56 | 1 | 0 | 0 | 7 | 3 | 0 | 619 | 608 |
| H | 214 | - | 24 | 0 | - | 60 | 0 | - | 340 |
| I | 247 | - | 0 | 0 | - | 3 | 0 | - | 389 |
| J | 1054 | - | 4 | 0 | - | 813 | 0 | - | 250 |

- We also report how VME and MME are utilized.



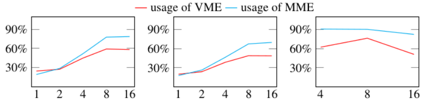

(a) ResNet-50 (FP16) (b) ResNet-50 (INT8) (c) BERT-128

# Hardware Utilization

- We report the frequencies of each memory level.

| label | DDR | | | LLB | | | L1 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo | TVM | crafted | GraphTurbo |
| Ⓐ | 58 | 1 | 1 | 0 | 11 | 11 | 0 | 291 | 284 |
| Ⓑ | 242 | 2 | 1 | 0 | 0 | 0 | 0 | 304 | 305 |
| Ⓒ | 242 | 2 | 1 | 0 | 25 | 110 | 0 | 401 | 240 |
| Ⓓ | 515 | 2 | 1 | 0 | 49 | 75 | 0 | 968 | 967 |
| Ⓔ | 242 | 2 | 1 | 0 | 25 | 76 | 0 | 474 | 337 |
| Ⓕ | 76 | 1 | 0 | 0 | 0 | 0 | 0 | 75 | 76 |
| Ⓖ | 56 | 1 | 0 | 0 | 7 | 3 | 0 | 619 | 608 |
| Ⓗ | 214 | - | 24 | 0 | - | 60 | 0 | - | 340 |
| Ⓘ | 247 | - | 0 | 0 | - | 3 | 0 | - | 389 |
| Ⓙ | 1054 | - | 4 | 0 | - | 813 | 0 | - | 250 |

- We also report how VME and MME are utilized.




(a) ResNet-50 (FP16) (b) ResNet-50 (INT8) (c) BERT-128

- The scalability to GPU is demonstrated in the paper using ResNet18-Tailor, which outperforms the CUTLASS implementations with and without convolution fusion by 1.06× and 1.23×.

# Contributions

- We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.

# Contributions

- We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.
- This synergy reduces off-core data movements, better saturates the valuable local memory, and empowers across-layer instruction scheduling.

# Contributions

+ We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.
+ This synergy reduces off-core data movements, better saturates the valuable local memory, and empowers across-layer instruction scheduling.
+ We design and implement a novel scheduling approach GraphTurbo, addressing the deployment of DNNs on DSA chips and offering insight to other platforms.

# Contributions

- We recognize the importance of considering hardware architecture at the graph partitioning level, enabling the synergy between network and hardware architectures.
- This synergy reduces off-core data movements, better saturates the valuable local memory, and empowers across-layer instruction scheduling.
- We design and implement a novel scheduling approach GraphTurbo, addressing the deployment of DNNs on DSA chips and offering insight to other platforms.
- The experimental results demonstrate that GraphTurbo can outperform two state-of-the-art tools and achieve performance comparable to the vendor-crafted code.

# Questions & Answers


We acknowledgment the TVM community led by Tianqi Chen, without whose work this paper would be impossible.