Exact Scheduling to Minimize Off-Chip Data Movement for Deep Learning Accelerators

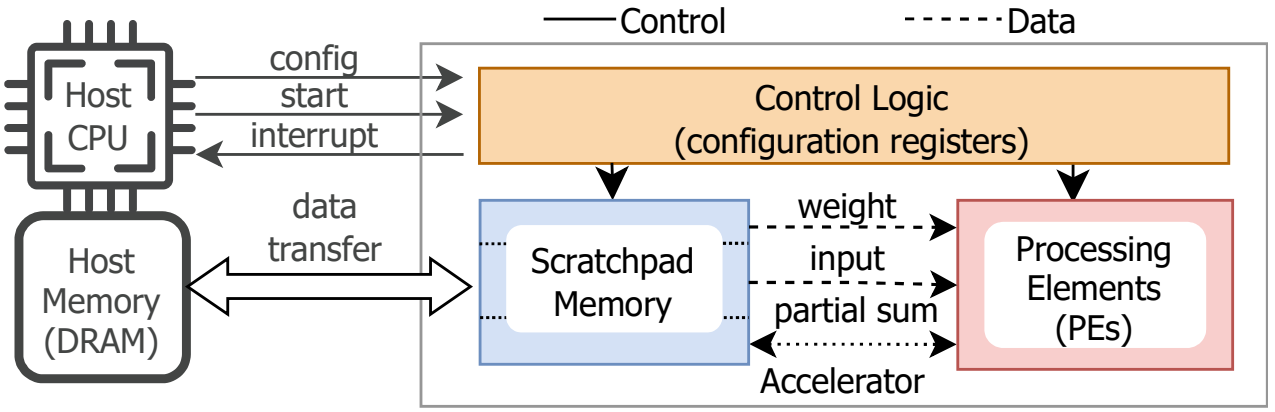
Yi Li, Aarti Gupta, Sharad Malik

Princeton University

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA, and by a Qualcomm Innovation Fellowship

Coarse-grain HW Accelerators

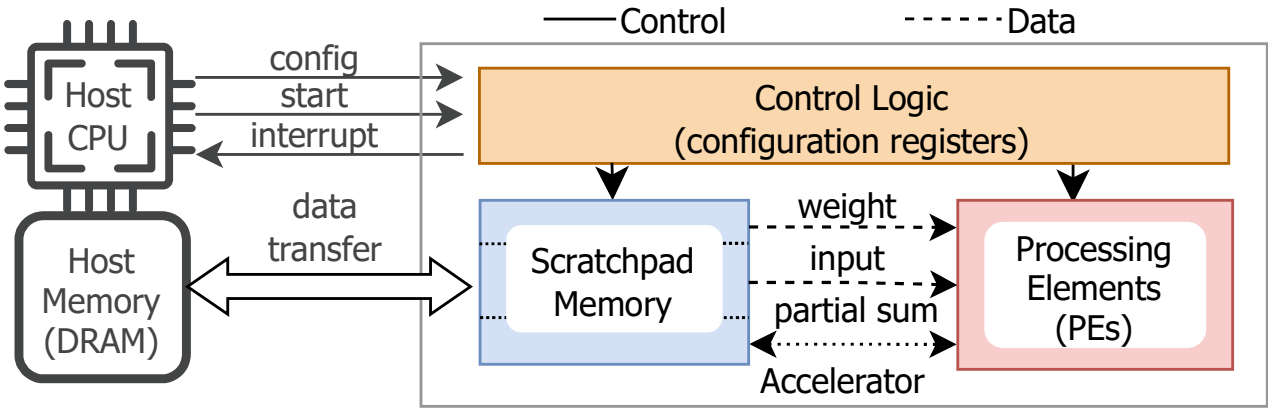
- Growing trend of “coarse-grain” HW accelerators to provide highly efficient computation for DNNs
  - Coarse granularity of HW functions, e.g., an entire Convolutional layer or an LSTM layer
  - Higher HW specialization, better efficiency



Typical coarse-grain deep learning accelerator design

Coarse-grain HW Accelerators

- Programmers specify operators' size parameters, and accelerator computes entire layers
  - Offloading follows a "data copy in – configure – trigger – data copy out" pattern



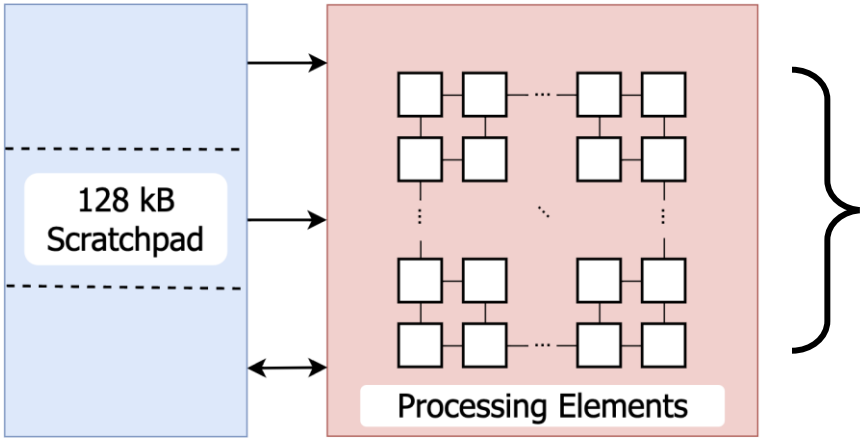
Typical coarse-grain deep learning accelerator design

Application-Level Operator (app-op) to Accelerator-Level Operator (acc-op)

E.g., mapping a 2D Convolution layer to HW Conv2D function:

```
# for input size of 14x14, requires ≈ 370 kB for all data (16-bit)
nn.Conv2D(in_channels=384,out_channels=576, kernel_size=1)
```

App-Op



HW Micro-Arch Constraints on the Acc-Op:

Max.InChans = 256

Max.OutChans = 256

Scratchpad Size = 128kB

- Often there are mismatches between app-ops (from DNNs) and acc-ops (HW function calls)
- Decomposition of app-ops (through operator tiling) is needed to fit acc-op

App-Op to Acc-Op Mapping

- App-op is mapped to loops of acc-ops by *loop tiling along dimensions of the operator*
  - Each acc-op invocation still follows the “data copy– configure – trigger – data copy” pattern

```
nn.Conv2D(large_input, large_wgt)
```

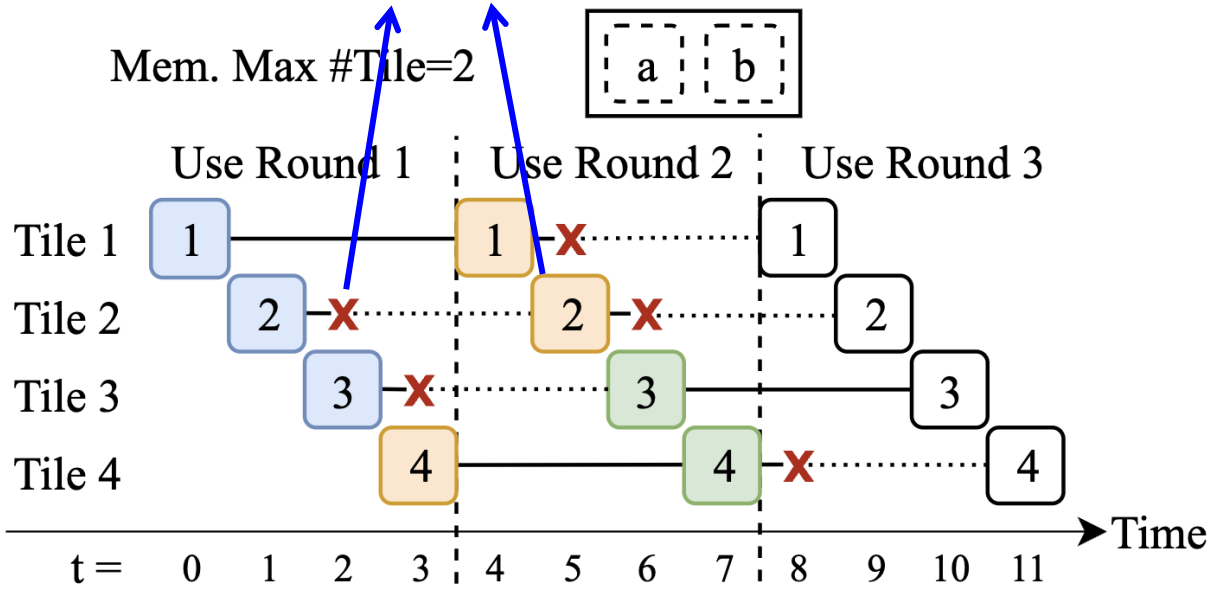
```
for t_oc in h_tiles:
  for t_ic in c_tiles:
    t_inp = getTile(inp, t_ic)
    t_wgt = getTile(wgt, t_ic, t_oc)
    out = accumulate(
                acc.Conv2D_small(t_inp, t_wgt)
    )
```

Example of mapping an acc-op to loops of acc-ops

App-Op to Acc-Op Mapping

- "Non-compulsory" off-chip data access from loops of Acc-Ops
  - Data reuse if same tensor tile resides in the accelerator
  - Data eviction if not enough space for incoming tensors

"Non-compulsory" off-chip data accesses from data eviction and later retrieval of the data from host memory



Example of data access pattern for a loop of 4 tile ops

App-Op to Acc-Op Mapping

- "Non-compulsory" data accesses for a given app-op/acc-op mapping is determined by
  - *Loop Tiling:* tile size and loop bounds
  - *Loop Ordering:* the order of the loops of different tiling dimensions
  - *Memory Partition:* partition of accelerator's memory for different tensors

```
nn.Conv2D(large_input, large_wgt)
```

⇩

```
for t_oc in h_tiles:
  for t_ic in c_tiles:
    t_inp = getTile(inp, t_ic)
    t_wgt = getTile(wgt, t_ic, t_oc)
    out = accumulate(
                acc.Conv2D_small(t_inp, t_wgt)
    )
```

Example of mapping an acc-op to loops of acc-ops

Collectively, these decisions are termed as **schedule** of app-op/acc-op mapping

App-Op to Acc-Op Mapping

- **Enormous** search space of valid schedules
  - A typical Conv2D layer with (C, K, P, Q)* of (64, 64, 224, 224) has $64^2 \times 224^2$ tiling choices and 4! different loop orders
  - Mapping to an accelerator operating on 16-byte vector with 128kB scratchpad [Whatmough et al. 2019] has $\approx 3.4 \times 10^6$ different partition for different tensors
  - 1.6×10^{16} different schedules in total

- **Significant** difference of total data access volume between a good and a bad schedule
  - For the above example, a bad schedule may result in **384x** more off-chip data access than the optimal one
  - Off-chip data access consumes significantly more energy than accelerator computation and increases latency [Horowitz et al. 2014]

* C, K, P, Q refers to input/output channel dimension and output height/width dimension

Prior Works on Scheduling Optimization

- Manual Tuning (e.g., Halide [Ragan-Kelly et al. 2013], TVM [Chen et al. 2018])
  - Requires significant manual effort from experienced performance engineers
- Auto-tuning (e.g., AutoTVM [Chen et al. 2018])
  - Requires substantial execution data points and training time, not feasible for early-stage accelerator designs
- Architectural Mapping (e.g., Timeloop [Parashar et al. 2019], CoSA [Huang et al. 2021])
  - Focus on mapping computation to the accelerator's spatial resources, leading to much larger search space
  - Rely on approximate memory model to estimate off-chip data access

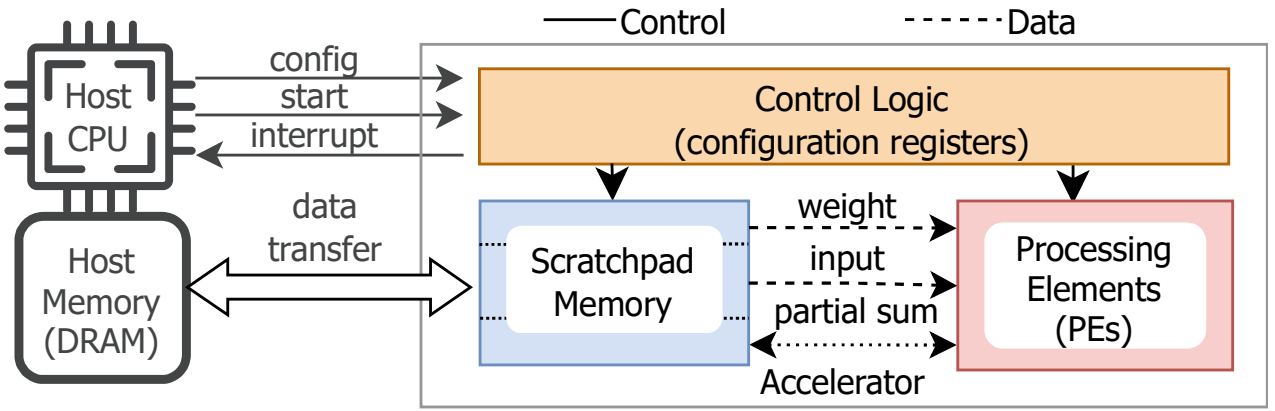
This Work:

Shoehorn: An optimal scheduler for minimizing data-access

Uses exact memory model

Problem Definition

- **Application-level and accelerator-level operators**
  - *app-op's* behavior can be statically determined
  - *app-op* can be mapped to an *acc-op* or loops of *acc-ops*
- **Accelerator Template**
  - Coarse-grain Accelerator
  - SW-controlled scratchpad
- **Schedule Template**
  - Loop Tiling - L^{Tiling}
  - Loop Order - L^{Order}
  - Memory Partition - M^{P}



Typical coarse-grain deep learning accelerator design

Problem Definition

- **Problem Definition**
  - Given an *acc-op* P^{app} and an accelerator target P^{acc}
  - find a schedule $(L^{Tiling}, L^{Order}, M^{P})$ that minimizes the sum of data access X_d for all tensor d

$$\underset{L^{Tiling},\, L^{Order},\, M^{P}}{\text{minimize}} \sum_{d \in D} X_d(P^{app}, P^{acc}, L^{Tiling}, L^{Order}, M^{P})$$

  - Large solution space size, e.g., 10^{16}
  - Need to compute data accesses for each point in solution space
    - Simulation is too slow for this

Analytical Model to Calculate Off-Chip Data Access

- **Overcome the speed limitation of simulation**
  - fast evaluation of a data point in the search space without expensive memory simulation

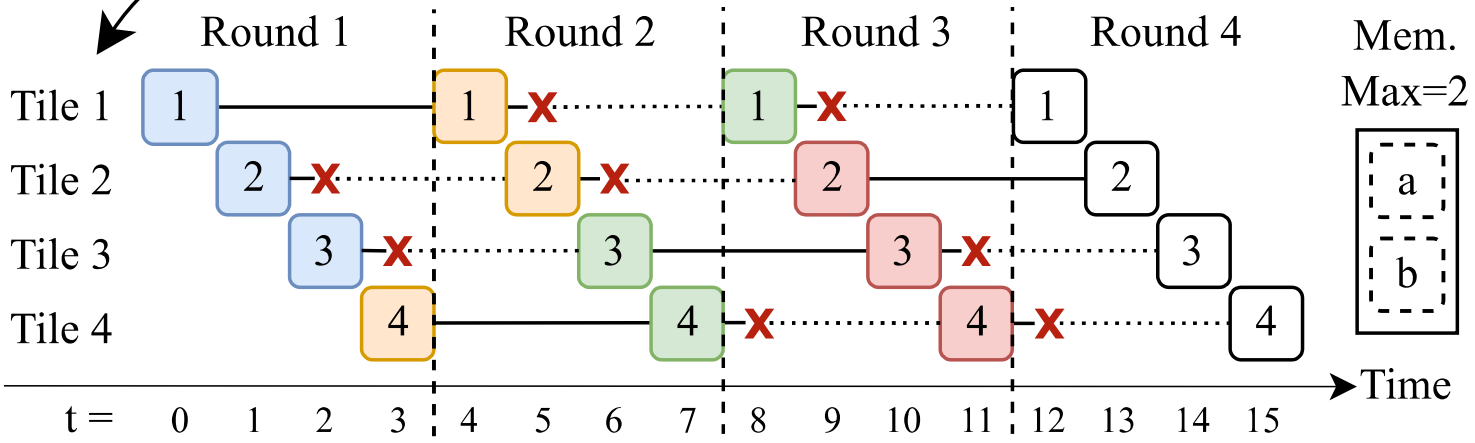
- **Possible as computation is statically determined**
  - statically determined computation of DNN operator
  - fully SW controlled scratchpad
  - Belady's algorithm to determine the optimal data eviction decision for least amount of data access
    - Replace the tile which has the furthest reuse

L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.

Analytical Model to Calculate Off-Chip Data Access

- **Derive analytical model from cyclic pattern**
  - cyclic access patterns of tensor tiles
  - cyclic eviction patterns of tensor tiles when Belady's algorithm used

```
1: for toc in range(num_toc):      # num. outChan tile = 4
2:   for tic in range(num_tic):    # num. inChan tile = 1
3:     for th in range(num_th):    # num. height tile = 2
4:       for tw in range(num_tw): # num. width tile = 2
5:         # get tiled input based on the loop indices
6:         tiled_inp = get_tiled_inp(input, tic, th, tw)
7:         ...
```

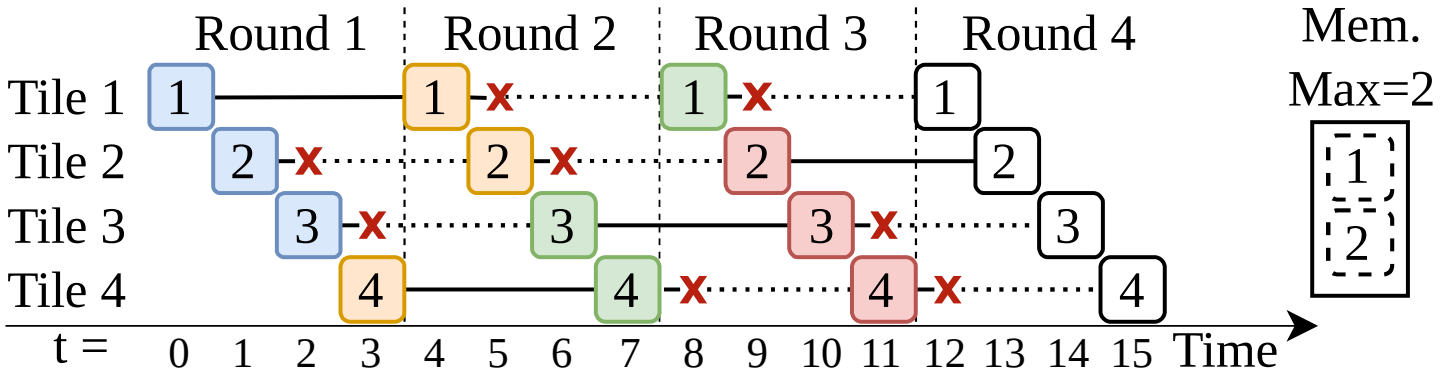


Analytical Model to Calculate Off-Chip Data Access

- **Calculate exact data access from the cyclic patterns**
  - In essence, given a schedule $(L^{Tiling}, L^{Order}, M^{P})$, we can directly calculate the total data access
  - please refer to the paper for more details…

$$S_i = \alpha_d \times \frac{T_i^g \times (R_i - 1)}{T_i^g - 1} \times (T_i^{total} - N_d^{avail}) \times M_d^t \qquad (3)$$

$$S_i = R_i \times T_i^g \times S_{i-1} + \alpha_d \times T_i^g \times T_{i-1}^{total} \times (R_i - 1) \times M_d^t \qquad (4)$$

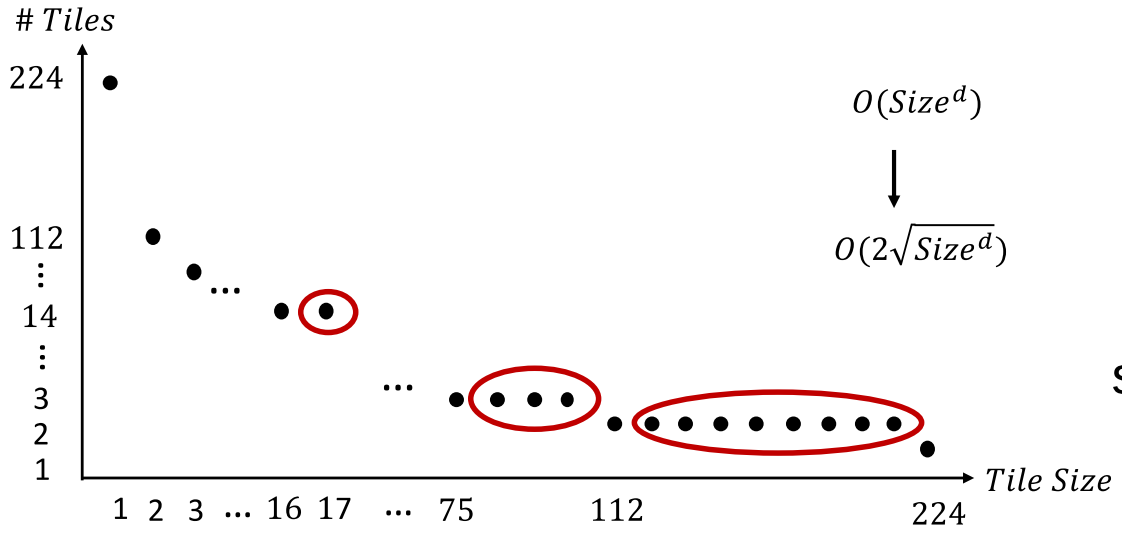


Pruning Techniques

- Even with the analytical model, the search space is still too large
- Back to the previous example:
  - A Conv2D layer with (C, K, P, Q) of (64, 64, 224, 224) has ≈1.6 x 10^{16} different schedules in total
  - ≈10^{8} hours to explore with a modern CPU
- We develop a set of pruning techniques to reduce the search space to a manageable size

Pruning Techniques

- Pruning on tiling choices
  - *Pruned by HW parameters*
    - e.g., HW vector size, #processing elements
  - *Pruned by analytical model*
    - for the tile sizes leading to the same number of tiles, keep the smallest to reduce unnecessary padding in the last tiles



Don't need to consider all 64^2 x 224^2 tiling choices!

For the same problem, search space is pruned from **1.6 x 10^{16}** to **5.1 x 10^{12}**

Tile size pruning by analytical model

Figure based on dissertation of Qi Nie (2020)

Pruning Techniques

- Derive the optimal memory partition instead of evaluating every allocation
  - Based on the "sensitivity" of data access changes from memory allocation - $\delta_d = \Delta S_d \,/\, \Delta M_d$
  - Insight: assign memory to tensor that reduces data accesses most

$$S_i = \alpha_d \times \frac{T_i^g \times (R_i - 1)}{T_i^g - 1} \times (T_i^{total} - N_d^{avail}) \times M_d^t \qquad (3)$$

$$S_i = R_i \times T_i^g \times S_{i-1} + \alpha_d \times T_i^g \times T_{i-1}^{total} \times (R_i - 1) \times M_d^t \qquad (4)$$

$$\min_{M_d} \sum_{d \in D} S_d(L^{Tiling}, L^{Order}, M_d)$$

$$s.t. \sum_{d \in D} M_d \leq M$$

Problem for optimal memory partition

For the same example, search space is pruned from **5.1 x 10^{12}** to **1.5 x 10^6**

Manageable by exhaustive search!

Additional Speedup

- Programming efficiency
  - Multi-threading
  - Caching prior data access calculations
- Additional tiling size pruning heuristic
  - Shoehorn-H
  - Reduce the tiling size selection to only factors of the original dimensions
  - e.g., for dimension size of 16, reducing tile sizes from [1, 2, 3, 4, 6, 8, 16] to only [1, 2, 4, 8, 16]

In practice, the search space for our example is further pruned from **1.6×10^6** to **9.7×10^4** , another 15x pruning

Exhaustive Search for Optimal Solution

- The pruning techniques facilitate exhaustive search for optimal solution
- Furthermore, the search can be speeded up by
  - Multi-threading
  - Caching prior data access calculations

Algorithm 3 Exhaustive Search for Optimal Schedule

1: **function** GETOPTIMALSCHEDULE(P_{app}, P_{acc})
2: U = getPrunedSearchSpace(P_{app}, P_{acc})
3: $bestSch = None$
4: **for** (L^{order}, L^{tiling}) in U **do**
5: M^P = getMemPartition(L^{order}, L^{tiling}, P_{app}, P_{acc})
6: $\{S_d\}$ = {getCLA (L^{order}, L^{tiling}, M_d) for d in D}
7: $bestSch$ = update($bestSch$, getCost(L^{order}, L^{tiling}, M^P, $\{S_d\}$))
8: **return** $bestSch$ ▷ optimal schedule: (L^{order}, L^{tiling}, M^P)

Cost function as exact amount of data access

Extensible to more complex cost functions

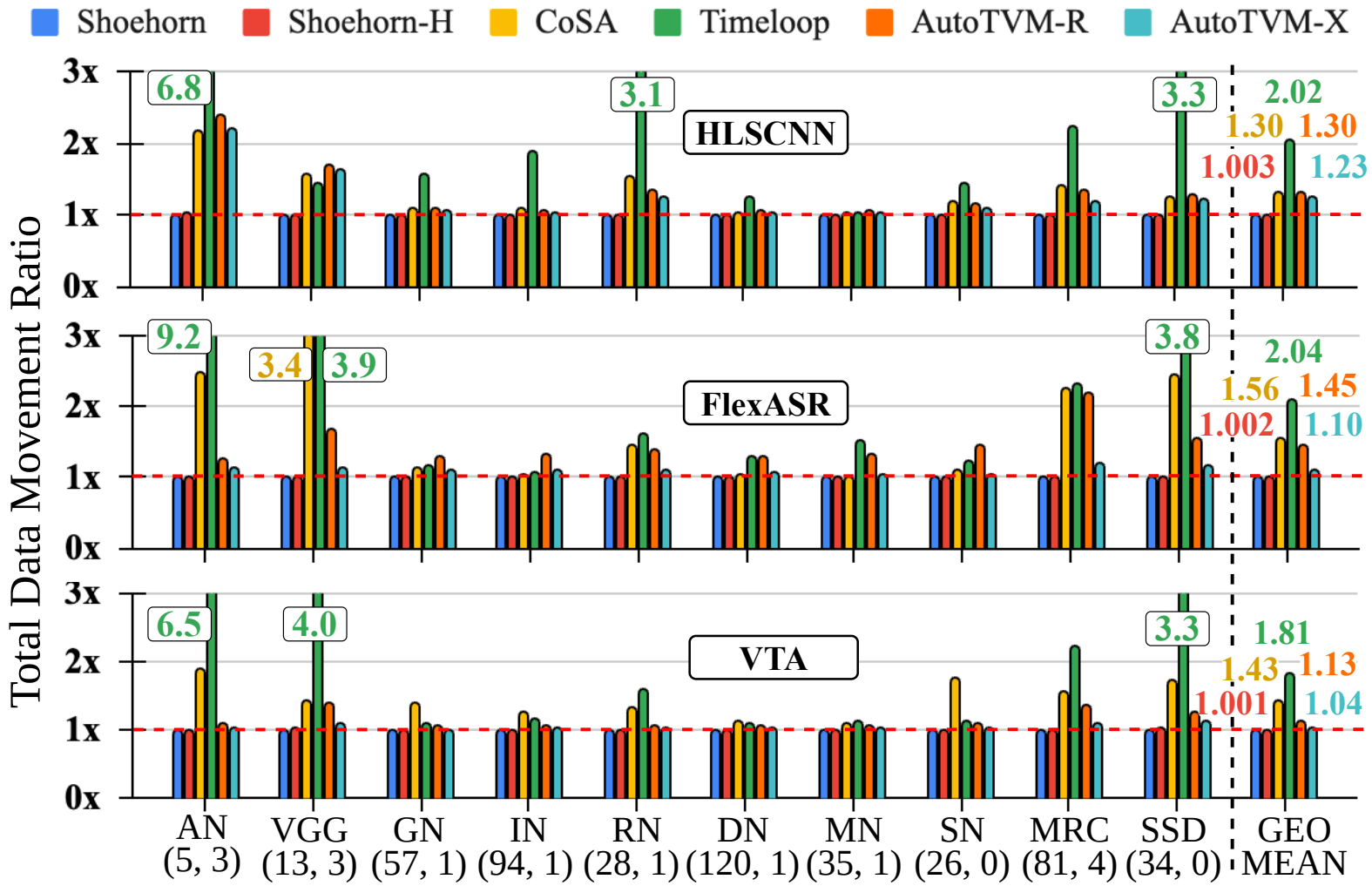
Exhaustive search algorithm

Evaluation Setup

- **Target Accelerators**
  - FlexASR [Tambe et al. 2021]
  - HLSCNN [Whatmough et al. 2019]
  - VTA [Moreau et al. 2019]
- **Target Applications**
  - 10 widely used DNNs
  - All contain linear layers and Conv2D layers to be offloaded to the accelerators
- **Target Scheduler tools**
  - Shoehorn and Shoehorn-H (with heuristic-based pruning)
  - CoSA [Huang et al. 2021]: constraint-solving-based with approximate memory model
  - Timeloop [Parashar et al. 2019]: Heuristic-based scheduler with approximate memory model
  - AutoTVM [Chen et al. 2018]: autotuning based with random sampling (AutoTVM-R) and with learning-based model (AutoTVM-X)

Evaluation Results

- Total data movement comparison relative to Shoehorn



Evaluation Results

- Time-to-solution for different scheduling tools
  - Shoehorn achieve the minimal data movement in **sub-seconds** on average
  - AutoTVM achieves the next best results after Shoehorn, at the cost of 100x ~ 10000x more solution time
  - Shoehorn-H achieves near-optimal results (within 0.1% ~ 0.3%) and is ≈7x faster than Shoehorn for HLSCNN

Schedulers	HLSCNN	FlexASR	VTA
Shoehorn	1.07s	0.07s	0.10s
Shoehorn-H	0.14s	0.06s	0.07s
CoSA	0.34s	0.32s	0.49s
Timeloop	7.75s	2.20s	2.19s
AutoTVM-R	107.51s	113.62s	112.81s
AutoTVM-X	1072.00s	849.64s	880.18s

∗ All experiments run on AMD EPYC-7532 CPU.

Summary

- Shoehorn, a scheduling optimizer to minimize off-chip data accesses for mapping DNN operators to accelerator
  - An analytical model to accurately calculates the memory access for a given schedule
  - Efficient pruning based on accelerator parameters, tile-size analysis and automatic memory partitioning
- Shoehorn can generate optimal schedules in sub-seconds for a variety of DNNs on different accelerators
- Pruning heuristic to further reduce time-to-solution with near-optimal result