Flexer: Out-of-Order Scheduling for Multi-NPUs

Hyemi Min
hyemi@csap.snu.ac.kr
Seoul National University
Seoul, South Korea

Jungyoon Kwon
jungyoon@csap.snu.ac.kr
Seoul National University
Seoul, South Korea

Bernhard Egger
bernhard@csap.snu.ac.kr
Seoul National University
Seoul, South Korea

Abstract

Recent neural accelerators often comprise multiple neural processing units (NPUs) with shared cache and memory. The regular schedules of state-of-the-art scheduling techniques miss important opportunities for memory reuse. This paper presents Flexer, an out-of-order (OoO) scheduler that maximizes instruction-level parallelism and data reuse on such multi-NPU systems. Flexer employs a list scheduling algorithm to dynamically schedule the tiled workload to all NPUs. To cope with the irregular data access patterns of OoO schedules, several heuristics help maximize data reuse by considering the availability of data tiles at different levels in the memory hierarchy. Evaluated with several neural networks on 2 to 4-core multi-NPUs, Flexer achieves a speedup of up to 2.2x and a 1.2-fold reduction in data transfers for individual layers compared to the best static execution order.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Compilers**; • **Theory of computation → Shared memory algorithms**.

Keywords: Parallel AI accelerators, out-of-order operation scheduling, memory management

ACM Reference Format:
Hyemi Min, Jungyoon Kwon, and Bernhard Egger. 2023. Flexer: Out-of-Order Scheduling for Multi-NPUs. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579990.3580025

1 Introduction

In recent years, deep learning has reshaped the computing landscape. To accommodate the structural and computationally complex workloads of deep neural networks (DNNs), more and more embedded systems contain multi-core neural processing units (multi-NPUs) [7, 8, 18, 19, 24].

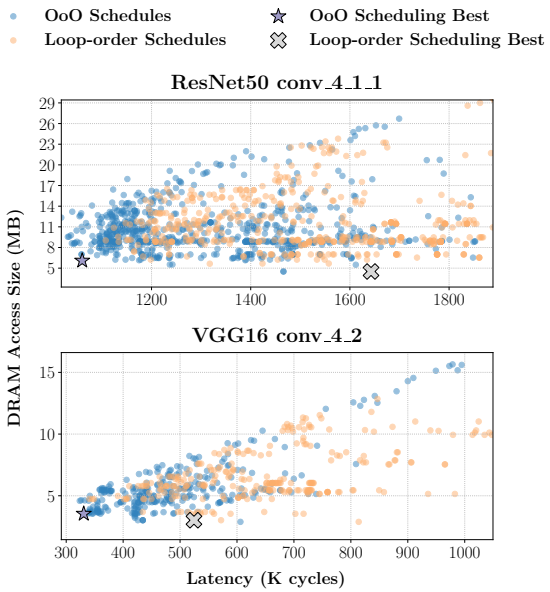


Figure 1. Performance comparison of fixed dataflow versus OoO schedules for various tiling sizes on a two-NPU system

To match the resource constraints of a mobile NPU, the computation of a DNN layer is partitioned into smaller operations. The size of these so-called tiles, the dimensions along which the data is tiled, and the execution order of the individual tiles significantly affect the execution time, the number of data accesses, and the energy efficiency of an execution schedule. Several techniques to find the optimal schedule for a certain layer and a given NPU have been proposed. Depending on which data is kept on-chip, these techniques can be classified into input, output, and weight-stationary schedules [3]. Optimization techniques such as traditional loop optimization [9, 15, 16, 23], machine learning [2, 13, 26], genetic algorithms [14, 25], language models [22] or mathematical solvers [11, 21] are used to find a good data tiling and loop ordering of a DNN layer.

On multi-NPUs architectures, these loop transformations are applied to a data partitioning of a layer [4, 7, 27]. For multiple cores with shared on-chip memory, such loop order-based schedules miss opportunities for data reuse, resulting in sub-optimal performance. Figure 1 plots the latency and the of accessed off-chip memory of two layers from ResNet50 and VGG16 on a two-NPU system. For all valid data tilings, the yellow dot represent the best fixed loop ordering while

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580025

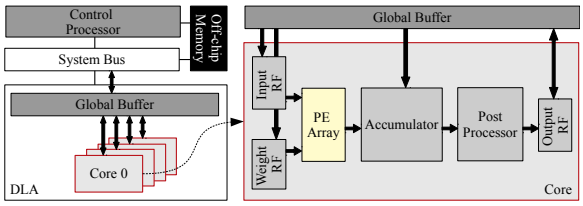


Figure 2. High-level accelerator structure

the blue dots represent the schedules obtained with the presented out-of-order scheduler. The results show that OoO scheduling has the potential to achieve significantly better performance than fixed loop order-based schedules.

In this work, we present *Flexer*, an execution schedule generator for multi-NPUs. *Flexer* operates similar to a list instruction scheduler for multi-issue machines where each NPU represents an FU. It schedules the operations of a tiled workload represented in a data-flow graph (DFG). Data tiles are assigned to variable-sized "registers" and brought on/off-chip on-the-fly by greedy allocation heuristics that maximize data reuse. A priority function selects the operations that minimize data transfers and latency from the ready-to-execute operations. Unlike loop order-based schedules that use pre-determined regions for each type of data (weights, input and output activations), the irregular data allocation sequence of an OoO schedule can lead to memory fragmentation. *Flexer* employs heuristics to prioritize data movement operations that minimize fragmentation.

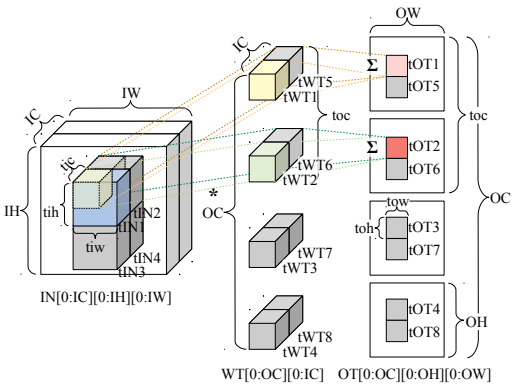
Flexer is evaluated on a parameterizable multi-NPU architecture with four DNNs. Compared to the best loop-order schedule, the out-of-order schedules generated by *Flexer* achieve a speedup of up to 1.67 and a data transfer size reduction of 1.26 for individual layers.

The remainder of this paper is organized as follows. Section 2 describes a general multi-core NPU architecture and provides some background on the execution of DNNs on NPU systems. Section 3 introduces the high-level operation of *Flexer*. Section 4 discusses optimizations for memory management and priority functions to generate efficient schedules. Section 5 presents the results obtained for different multi-NPU systems and deep neural networks. Section 6 discusses related work, and Section 7 concludes this paper.

2 Background

2.1 Parameterizable Accelerator Architecture

Flexer assumes a general multi-NPU DNN accelerator as shown in Figure 2. A shared on-chip scratchpad memory, labeled *Global Buffer*, facilitates data reuse between cores and reduces the amount of data loaded from external memory. *Flexer* is configurable in the number of NPUs and the size of the global buffer. While the organization of the NPUs is not of particular importance, a cycle-accurate performance model must be available to compute the latency of operations for given data (tile) sizes.



(a) Tiled convolution

```
for ow=0 to OW, ow+=tow do                    : Loop1
  for oh=0 to OH, oh+=toh do                  : Loop2
    for ic=0 to IC, ic+=tic do                : Loop3
      for oc=0 to OC, oc+=2*toc do            : Loop4
tCONV N:    OT[ow:ow+tow][oh:oh+toh][oc:oc+toc]
              +=IN[ic:ic+tic][iw:iw+tiw][ih:ih+tih]
               * WT[oc:oc+toc][ic:ic+tic]
tCONV N+1:  OT[ow:ow+tow][oh:oh+toh][oc+toc:oc+2*toc]
              +=IN[ic:ic+tic][iw:iw+tiw][ih:ih+tih]
               * WT[oc+toc:oc+2*toc][ic+tic:ic+2*tic]
      end
    end
  end
end
```

(b) Tiled loop of a convolution

Loop order: Loop1 – Loop2 – Loop3 - Loop4

1st:	tCONV1	tOT1	tIN1	tWT1	
	tCONV2	tOT2	tIN1	tWT2	
2nd:	tCONV3	tOT3	tIN1	tWT3	
	tCONV4	tOT4	tIN1	tWT4	
3rd:	tCONV5	tOT1	tIN2	tWT5	tOT1
	tCONV6	tOT2	tIN2	tWT6	tOT2
4th:	tCONV7	tOT3	tIN2	tWT7	tOT3
	tCONV8	tOT4	tIN2	tWT8	tOT4
5th:	tCONV9	tOT5	tIN3	tWT1	
...					

(c) Operation sequence for a given loop order

Figure 3. Tiled execution of a convolution

2.2 Tiling and Scheduling on Multi-NPU Architectures

The computation of a single layer in a DNN, such as a convolution, has to be split into several operations that operate on smaller data tiles because the total data size a layer is too large to be brought into the NPU at once [1]. This process is called *tiling*.

The latency of an operation depends on the dimension and direction of a tiling. Figure 3 (a) illustrates the tiling of a convolution layer. The three-dimensional tiles for input, weight, and output data, IN, WT, and OT, are parameterized by height, width, and channel depth denoted by $t?w$, $t?h$, and $t?c$ where $? \in \{IN, WT, OT\}$. Figure 3 (b) shows the resulting nested loop structure. The innermost loop is unrolled n times

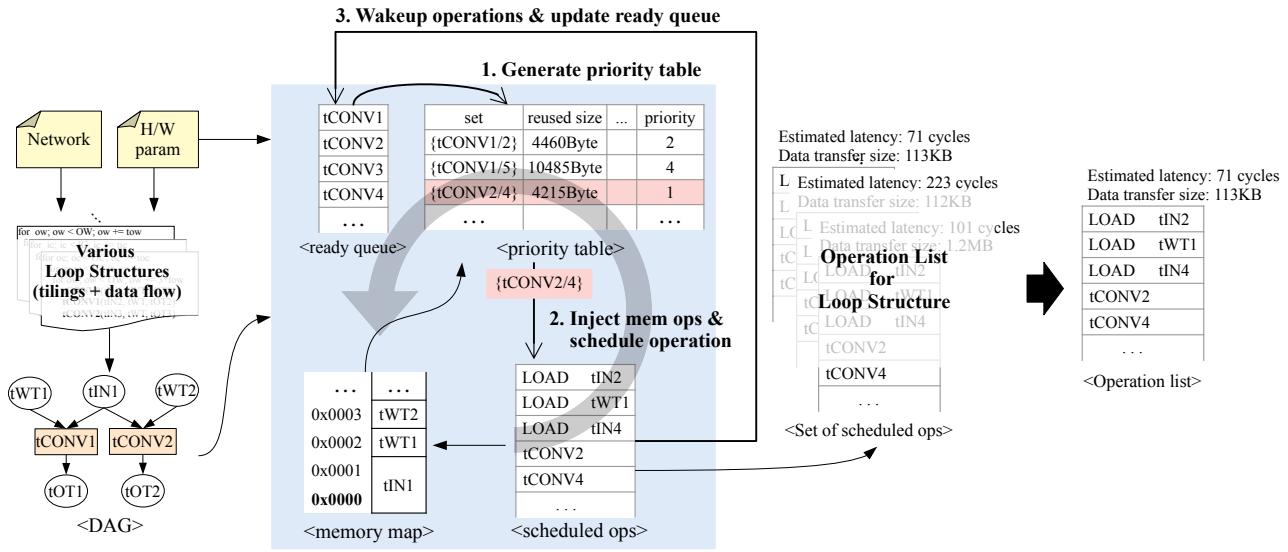


Figure 4. Organization and operation of *Flexer*

on a multi-NPU with n NPUs. In Figure 3 (c), the execution of the loop is shown as a sequence of convolution operations

$$tCONV\ OT \leftarrow IT, WT[, PS]$$

where each operation generates a (partial) output tile OT from an input tile IT, a weight tile WT, and an optional partial sum PS.

The state-of-the-art scheduling techniques determine the tiling size and the loop order to minimize a given goal. As shown in Figure 1, static approaches can miss opportunities for better schedules on multi-NPU architectures. Consider the situation as shown in Figure 3 (c) where the first two convolutions, `tCONV1` and `tCONV2`, have been executed. `tCONV1` generates `tOT1` which allows `tCONV5` to run; similarly, the result `OT2` generated by `tCONV2` allows execution of `tCONV6`. Other independent convolutions such as `tCONV9` are also ready to be executed. *Flexer* is able to achieve better execution plans by considering the dataflow graph of tiled convolutions and scheduling operations that maximize data reuse and minimize the latency of ready-to-run operations.

3 Overview of Flexer

The organization and operation of *Flexer* is shown in Figure 4. *Flexer* schedules operations similar to a list instruction scheduler for multi-issue machines where each NPU represents an FU. *Flexer* assumes a "register-to-register" model, i.e., only computational operations are represented in the DFG of a layer. The assignment of SPM to data tiles is implemented in a greedy manner and similar to a linear scan register allocator. Data movements are treated like spill operations: the required memory operations to move data tiles (input, weight, partial sum and output) from DRAM to SPM and vice-versa are inserted on-the-fly. For a given DAG of a layer, all operations whose inputs are available are inserted into the ready queue from where they are selected for execution on the individual NPUs. The latency of an operation o for a given tile size is assumed to be available through a model. If not all required input tiles are currently present in the SPM, a load l (and, potentially, a prior spill s) are inserted and the latency of the operation is updated ($lat(total) = lat(o)+lat(p)+lat(l)$).

Algorithm 1 details the operation of *Flexer*. For a given network and hardware configuration, *Flexer* generates an OoO schedule for all viable tilings and data flows (i.e., loop orderings) and returns the best schedule (lines 2–11). The metric to rank the schedules by is configurable; in this work, we use $latency \times transferred\ data$ (line 5).

The function `GetSchedule` computes an OoO schedule for a given tiling and data flow (lines 12–27). After generating the data-flow graph, the ready queue is populated with all operations that have no unsatisfied data dependencies (lines 13–15). Scheduling is performed by generating all possible sets of parallelizable tiled operations and calculating the data transfer size and total latency to determine the priority of each set. *Flexer* then schedules the set with the highest priority (line 19- 20). *Flexer* checks whether the data of all input operands is available in the global buffer and reserves space for the generated result. If necessary, operations are generated to spill (write-back) to, resp. load required data from off-chip memory (line 21). After all memory operations have been issued, the operations of the set are scheduled to the individual NPUs (line 22). For each operation, *Flexer* records the start and end time of the operation considering data dependencies and the availability of the resources. Operations that become ready as a result of executing the current set are woken up and placed in the ready queue (line 24).

Algorithm 1: Scheduling

```
Input  : network configuration, hardware configuration
1  min_metric ← ∞, best_schedule ← NULL
2  foreach tiling_factor do
3      foreach dataflow do
4          schedule ← GetSchedule(tiling_factor, dataflow)
5          metric ← schedule.latency × schedule.data_transfer_size
6          if metric < min_metric then
7              best_schedule ← schedule
8              min_metric ← metric
9          end
10     end
11 end
Output : latency, data_transfer_size, scheduled_ops
12 Function GetSchedule(tiling_factor, dataflow):
13     DAG ← GenerateDAG(tiling_factor, dataflow)
14     op_list ← {OP | OP ∈ DAG}
15     ready_op_list ← {OP_ready | OP_ready ∈ DAG
                        , OP_ready has no dependency}
16     scheduled_ops ← ∅
17     data_transfer_size ← 0
18     while ready_op_list != ∅ or op_list != ∅ do
19         sets ← MakeCombination(ready_op_list, num_cores)
20         selected_ops ← Get1stPrioritySet(sets)
21         mem_ops ← GenerateMemOPs(selected_ops)
22         scheduled_ops ← scheduled_ops
                    + SchedulingOPs(selected_ops, mem_ops)
23         data_transfer_size ← data_transfer_size
                    + GetDataTransferSize(mem_ops)
24         ready_op_list ← ready_op_list
                    + WakeupOPs(selected_ops, mem_ops)
25     end
26     latency ← scheduled_ops.last_element.end_time
27     return Schedule(latency, data_transfer_size, scheduled_ops)
28 end
```

Flexer iterates over all possible data tilings and computes a schedule for each DFG. The complexity of computing a schedule for a given DFG is dominated by the number of data allocations which is exponential in the number of NPUs. Overall, this leads to a huge number of schedules to be calculated. *Flexer*'s single-threaded code requires about 20 hours to find a schedule for a 4-core NPU and ResNet50 on a recent computer. If runtime is a concern, implementing a memory function to remember the best tiling could significantly reduce the runtime of the scheduler. Also, schedules for different data allocations could be computed in parallel.

4 Memory Management and Prioritization

Flexer creates sets of n operations that execute in parallel on an multi-NPU system comprising of n individual NPUs. Each set is composed of n operations from the ready queue and is assigned a priority that is computed according to the priority function. *Flexer* aims to maximize data reuse in the on-chip memory; the priority thus reflects the impact of an operation set on the amount of transferred data and the execution latency of the set.

Another important concern is the management of the global on-chip buffer space. The impact on the amount of data transferred and the latency of a set depend on the order of execution and the memory overhead which, in turn, depend on a good memory management policy. The memory management policy decides where to place data that is generated or loaded from off-chip memory and what data to evict when insufficient on-chip memory is available.

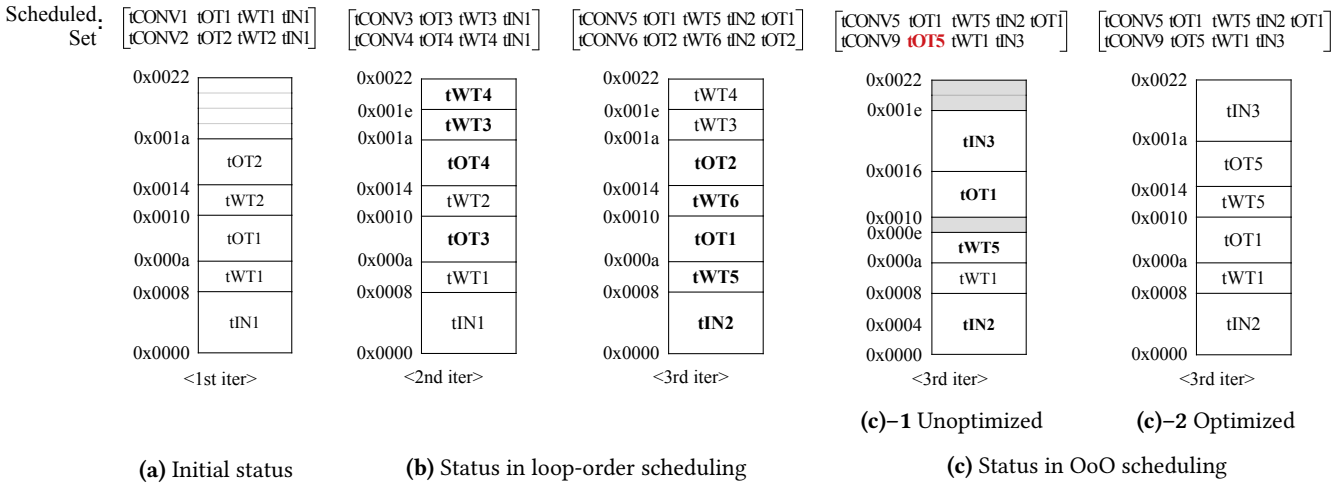
In state-of-the-art fixed-order schedulers, memory management is relatively simple because the ordering of data transfers is regular and the sizes of the regions reserved for input, weight, and output data are known. The irregular schedules generated by *Flexer*'s OoO scheduler, however, call for more elaborate memory management policies.

Simple memory allocation policies such as first fit placement result in a heavily fragmented on-chip memory and cause a large number of data reload and spill operations. A fragmented on-chip memory prevents the OoO scheduler from generating schedules with a high computational load because the NPUs are stalled by memory operations. The decision about which data to spill is equally important because it impacts both the data reuse and amount of data transferred (Section 4.1).

Another consideration is the large number of sets of parallel operations. While the goal of *Flexer* is to demonstrate the potential of OoO scheduling and not speed, evaluating all possible sets along with determining how to manage the on-chip memory would cause too much overhead in the scheduler. Section 4.3 details how the priority of the sets is computed, how the on-chip memory is managed, and how sets are ignored that represent identical data flows with different operations.

4.1 Memory Management

For regular loop-order schedules, the size of the memory regions to hold input, weight, and output data do not change once their size has been determined. Figure 5 illustrates the differences of memory management with fixed loop-order scheduling versus out-of-order scheduling. Figure 5 (a) illustrates the memory organization after the first operation set has been scheduled according to the schedule from Figure 3 (c), an *input-stationary* schedule that shares the input data among NPUs and maximizes the reuse of the on-chip input data. Weight and output data differ from set to set and



(c)–1 Unoptimized **(c)–2** Optimized

(a) Initial status **(b)** Status in loop-order scheduling **(c)** Status in OoO scheduling

Figure 5. On-chip memory management

are brought in/out of the on-chip global buffer on demand. Figure 5 (a) illustrates the memory organization after the first operation set has been scheduled. Figure 5 (b) shows memory management for loop-order schedules. The second set reuses the input data and new weight data is allocated in a free memory block to maximize memory utilization. New output data is allocated to the same memory region as the previously generated partial sums `tOT1`, `tOT2` by spilling (e.g., writing back) the data back to off-chip memory. Similarly, in the third iteration, `tIN1` is not used anymore and replaced by `tIN2`; other data is also allocated at fixed locations and replaces existing data. Such an allocation is only possible because the data-flows observed with loop-order scheduling are regular, allowing the scheduler to pre-compute the memory addresses for the data to be used by each operation.

Flexer's out-of-order scheduler, however, does not generate regular dataflows. Figure 5 (c) shows one possible memory state where {`tCONV5, tCONV9`} are selected for execution after the second iteration from Figure 5 (b). Since the on-chip memory is full, some data needs to be spilled. The weight tile `tWT1` reused by `tCONV9`, and `tIN1` can be overwritten by the same-sized `tIN2` because `tIN1` is no longer needed. Spilling needs to evict one of the remaining data elements [`tOT3, tWT2, tOT4, tWT3, tWT4`]. With a first-fit policy that traverses the on-chip memory and spills the first spillable block that is big enough to hold the requested data, memory fragmentation occurs as shown in Figure 5 (c)–1: `tOT3` is spilled and `tWT5` is allocated in its place, resulting a small free memory fragment at `0x000e`~`0x0010`. Continuing with first-fit spilling, `tWT2, tOT4` are spilled and replaced by `tOT1` and `tWT3` is spilled for `tIN3`, creating another free memory fragment at `0x001e`~`0x0022`. This fragmentation prevents the allocation of `tOT5` despite sufficient free memory being available in the on-chip memory.

Avoiding such fragmentation is critical to achieving a good performance. A fragmented on-chip memory leads to low utilization and reuse and causes unnecessary spills. Another important point is the selection of the data tiles to be spilled. While the remaining reuse count of a block can be determined by inspecting the DFG of the layer, the execution order of operations affects their reuse distance. *Flexer* employs a heuristical algorithm that minimizes memory fragmentation and maximizes reuse through careful tile placement and spilling. In-place replacements of equally-sized blocks where the allocated block is no longer needed and can be replaced by an incoming block are handled first, then free memory blocks are allocated with a best-fit strategy. If there are still required tiles left, some allocated blocks need to be spilled. *Flexer*'s spilling heuristics are illustrated in Algorithm 2.

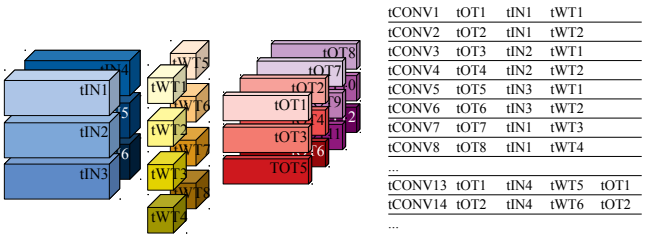
The spilling algorithm is invoked once for each allocated tile that doesn't fit into the on-chip memory. The goal of the heuristics is first to minimize fragmentation and then to maximize data reuse. The minimal sequence of consecutive blocks that is sufficiently large to hold the requested but causes the least fragmentation and highest data reuse is found by a loop nest iterating over all possible start and end positions (lines 9 and 11). The conditions under which a sequence is considered to be a new best sequence of blocks to spill (*best_spill_blks*) have been separated for better readability: first, a sequence is selected if it causes less fragmentation than the so-far best sequence (line 18). Second, a sequence is selected if it causes the same amount of fragmentation but is expected to lead to more data reuse than the so-far best sequence (line 20). Finally, a sequence is selected if it causes the same amount of fragmentation and data reuse but consists of fewer blocks than the currently best sequence (line 23). A newly identified best sequence is remembered (lines 28–31) and finally returned when all possible sequences of blocks have been explored (line 36).

Algorithm 2: Finding victim blocks for spilling

```
1  struct {
2      Data data
3      int size
4      int start_addr
5      int end_addr
6  } Block;
   Input  : required_size
7  memory_map ← list of allocated blocks
8  best_spill_blks ← ∅, min_disadv ← ∞, min_frag_size ← ∞
9  for start = 0 ... sizeof(memory_map) − 1 do
10     spill_blks ← ∅, disadv ← 0, spill_size ← 0, pick ← false
11     for end = start ... sizeof(memory_map) do
12         blk ← memory_map[end]        // get j_th block
13         spill_blks ← spill_blks + blk
14         spill_size ← spill_size + blk.size
15         disadv ← disadv + blk.size × blk.data.remain_uses
16         if spill_size ≥ required_size then
17             frag_size ← spill_size − required_size
18             if (frag_size < min_frag_size) then
19                 pick ← true
20             else if frag_size == min_frag_size and
21                     disadv < min_disadv then
22                 pick ← true
23             else if frag_size == min_frag_size and
24                     disadv == min_disadv and
25                     len(spill_blks) < len(best_spill_blks)) then
26                 pick ← true
27             end
28             if pick == true then
29                 best_spill_blks ← spill_blks
30                 min_disadv ← disadv
31                 min_frag_size ← frag_size
32             end
33             break
34         end
35     end
36 end
   Output : best_spill_blocks
```

4.2 Pruning

While the main purpose of this work is to explore and demonstration the potential of out-of-order scheduling and thus the runtime of the employed algorithms has largely been ignored, a minimal number optimizations had to be implemented to reduce *Flexer*'s runtime to a reasonable level. *Flexer* considers every possible combination of ready operations to be issued as an operation set. For o operations in the ready queue and n NPU cores, the number of possible combinations is given by ${}_oC_n$ which quickly leads to an unattainable number of combinations to consider. With 100 operations in the ready queue and four NPUs, for example, ${}_{100}C_4 \approx 3.92 \times 10^6$. We observe that the vast majority of these sets lead to identical



(a) Tiling convolution diagram **(b)** Operation sequence

Figure 6. Example of a tiled convolution

data-flow patterns with regards to the number and type of data movement operations. Since the data flow is the main factor when deciding the priority of an operation set, such duplicates are ignored.

Duplicates are identified by classifying an operation set's data flow by the number of newly allocated and re-used tiles of each type (`IN`,`WT`,`OT`). The dataflow map in Figure 7 (c) illustrates the concept. Operation set pruning reduces the number of operation sets to the order of $\prod_{i=1}^{n} P(i)$ where $P(x)$ denotes the number of distinct integer partitions of x.

4.3 Operation Set Priority

In each scheduling step, *Flexer* selects the operation set with the highest priority for scheduling (Algorithm 1, line 20). Similar to spilling, the priority function assigns a higher priority to operations sets that lead to higher memory utilization by considering the data reuse, the amount of data brought on-chip, and the amount of spilled data. In particular, the priority of an operation set is determined as follows:

- **Prioritize data reuse**
  Flexer minimizes the amount of data transferred between off- and on-chip memory by computing a so-called *memory benefit* that considers the amount of reused and spilled data of an operation set. The memory benefit is calculated as follows:

$$\textbf{memory benefit} = reused\ data - spilled\ data$$

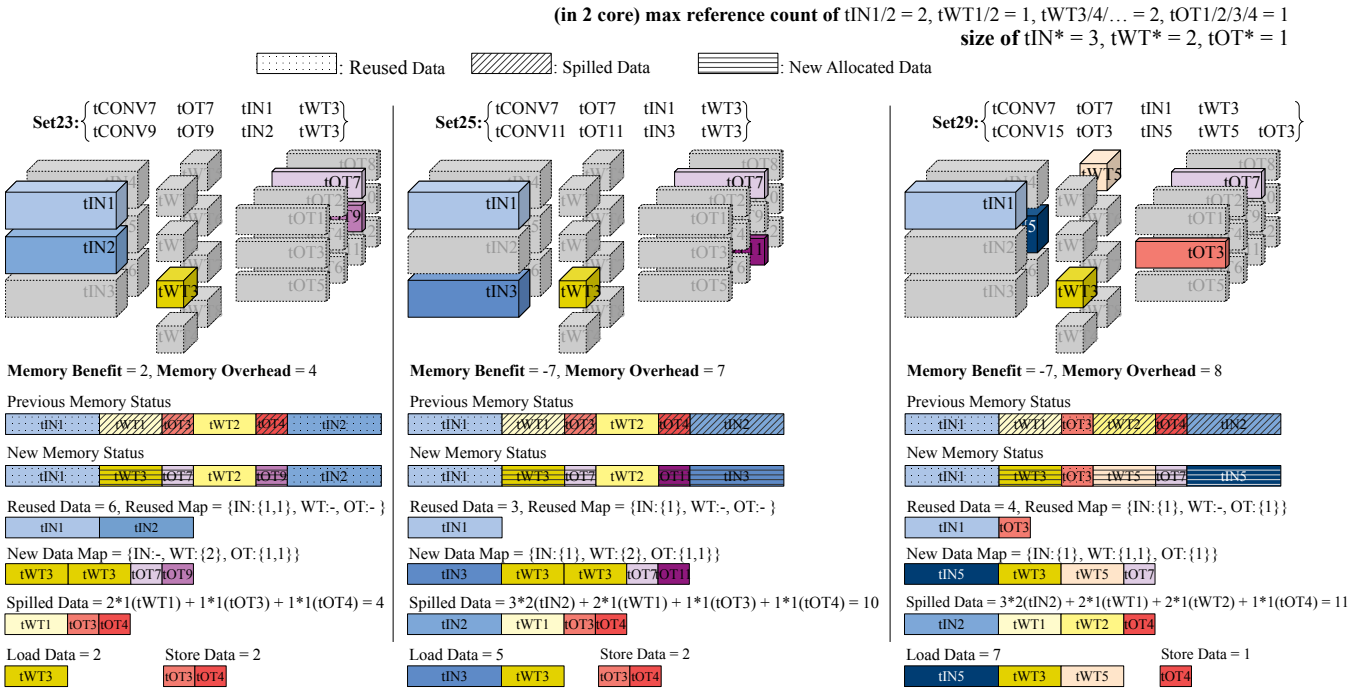
$$\textbf{reused data} = \sum_{rdata_{OP}} size(rdata_{OP})$$

$$\textbf{spilled data} = \sum_{sdata_{OP}} size(sdata_{OP}) * max\ ref\ count(sdata_{OP})$$

$$\textbf{max ref count} = min \begin{cases} \#\ of cores, \\ remain.\ uses(data) \end{cases}$$

$$where: \begin{array}{ll} rdata_{OP} & = \text{data reused by OP} \\ sdata_{OP} & = \text{data spilled by OP} \\ OP & = \text{operations in set} \end{array}$$

  When calculating value of *spilled data*, *Flexer* considers no only the size but also the remaining maximum reference count of the spilled data to keep data with a high reuse count in on-chip memory. As an example,



(a) Measuring memory benefit and memory overhead of sets

tCONV5	tOT5	tIN3	tWT1	
tCONV6	tOT6	tIN3	tWT2	
tCONV7	tOT7	tIN1	tWT3	
tCONV8	tOT8	tIN1	tWT4	
tCONV9	tOT9	tIN2	tWT3	
tCONV10	tOT10	tIN2	tWT4	
tCONV11	tOT11	tIN3	tWT3	
…				
tCONV13	tOT1	tIN4	tWT5	tOT1
tCONV14	tOT2	tIN4	tWT6	tOT2
tCONV15	tOT3	tIN5	tWT5	tOT3
…				

(b) Operations

Set		Memory Benefit	Memory Overhead	Utility	Priority	Dataflow map	
ID	Operations					Reuse Map	New Data Map
set1	tCONV5/6	-4	5	1	4	{IN:-, WT: {1,1}, OT:-}	{IN:{2},WT:-,OT:{1,1}}
set2	tCONV5/7	-5	7	1	5	{IN:{1},WT:{1},OT:-}	{IN:{1},WT:{1},OT:{1,1}}
set3	tCONV5/8	-5	7	1		{IN:{1},WT:{1},OT:-}	{IN:{1},WT:{1},OT:{1,1}}
…			…				…
set23	tCONV7/9	2	4	1	1	{IN:{1,1},WT:-,OT:-}	{IN:-,WT:{2},OT:{1,1}}
set24	tCONV7/10	0	6	1	2	{IN:{1,1},WT:-,OT:-}	{IN:-,WT:{1,1},OT:{1,1}}
set25	tCONV7/11	-7	7	1	6	{IN:{1},WT:-,OT:-}	{IN:{1},WT:{2},OT:{1,1}}
…			…				…
set28	tCONV7/14	-3	13	1	3	{IN:{1},WT:-,OT:-}	{IN:{1},WT:{1,1},OT:{1,1}}
set29	tCONV7/15	-7	8	1	7	{IN:{1},WT:-,OT:{1}}	{IN:{1},WT:{1,1},OT:{1}}
…			…				…

(c) Priority table

Figure 7. Operation set selection and memory management of *Flexer*

consider the situation shown in Figure 7 (b) showing the operation list after `tCONV1-4` have been scheduled. The *max ref count* for `tIN1,2` and `tWT3,4` is 2, while the value for `tWT1,2` and `tOT1-4` is 1 since these blocks are only referenced once more.

- **Prioritize higher memory utilization**
  If the memory benefit of two operation sets is identical, *Flexer* selects the set that achieves a higher memory utilization because a better packed on-chip memory offers more opportunities for data reuse for later operations.
- **Prioritize shorter memory operations**
  If memory benefit and utilization are identical for two operation sets, *Flexer* selects the set with the lower latency to spill and load the required data to eliminated or minimize idle time of the NPUs. For example, consider the operation sets `set25` and `set29` in Figure 7 (a) that have an identical memory benefit and memory utilization. In this case, `set25` is assigned a higher priority since its memory overhead is smaller than that of `set29`.

Figure 7 illustrates the entire process for a two-NPU architecture after the operations `tCONV1-4` from the tiling example in Figure 6 have been scheduled. Figure 7 (a) visualizes the data flow and memory allocation for the three operation sets `set23`, `set24` and `set29`. Details about the priority calculation are listed the priority table (Figure 7 (c)). The classification of the data-flows of `set2` and `set3` are identical, hence `set3` is pruned. For the remaining sets, the table shows the dataflow maps and the resulting memory benefit, overhead, and the final priority of the operation sets. Operation set `set23` is assigned the highest priority of 1 and thus selected as the next set to be scheduled. Also note that this is not an operation order that is generated by one of the static loop-order scheduling techniques.

5 Evaluation

Flexer is evaluated with four DNNs and a total of eight different configurations of a parameterizable multi-NPU accelerator developed by Samsung Research. The organization or the accelerator is identical to the general multi-NPU architecture from Section 2.1. Each NPU core contains a compute array comprising 32×32 PEs running at 1 GHz. The architecture is parameterizable in the size of the on-chip memory, the available memory bandwidth, and the number of NPU cores.

The accelerator itself is part of an SoC that includes general-purpose cores and other accelerators. The entire development toolchain comprising a compiler, a cycle-accurate simulator, and an RTL generator adapts to a specific architecture instance as defined by a configuration file listing the hardware parameters of the accelerator.

We compare the performance of the presented out-of-order scheduling technique to the the best static loop-order schedule with the four networks *VGGNet-16* [20], *ResNet-50* [10], *SqueezeNet* [12], *YOLOv2* [17] on eight multi-NPU architectures as shown in Table 1. The baseline is found through exhaustive search among all schedules with different data stationary models and viable tiling sizes. Figure 8 shows the baseline-normalized performance of schedules generated by *Flexer* in terms of execution time (inference) and the amount of transferred for the different networks and hardware configuration. *Flexer* achieves an end-to-end (entire network) speedup of up to 1.67× and a reduction of up to 1.26× in the amount of transferred data.

The end-to-end network performance from Figure 8 blurs the full potential of OoO scheduling because for some of the many layers, the best static schedule is optimal. In Figure 9, we compare the performance of *Flexer* with the baseline for a given network and hardware on a layer-by-layer basis. Figure 9 (a) reveals that *Flexer* achieves up to a 2.17-fold speedup and a 1.53-fold reduction in transferred data for a single layer. On the other hand, we also observe that *Flexer* produces less efficient schedules in terms of data transfers

Table 1. Hardware configurations used in the evaluation

	# of cores	on-chip memory size	bandwidth
arch1	2	256	32
arch2	2	256	64
arch3	2	512	32
arch4	2	512	64
arch5	4	256	32
arch6	4	256	64
arch7	4	512	32
arch8	4	512	64

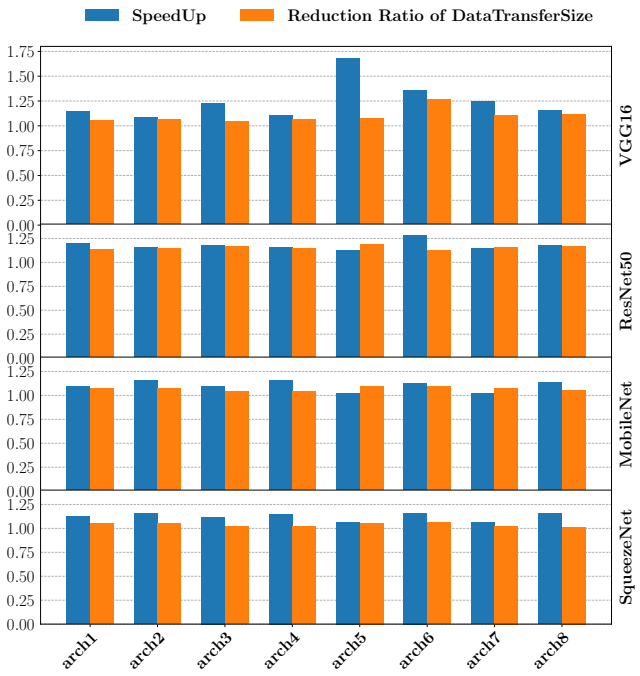
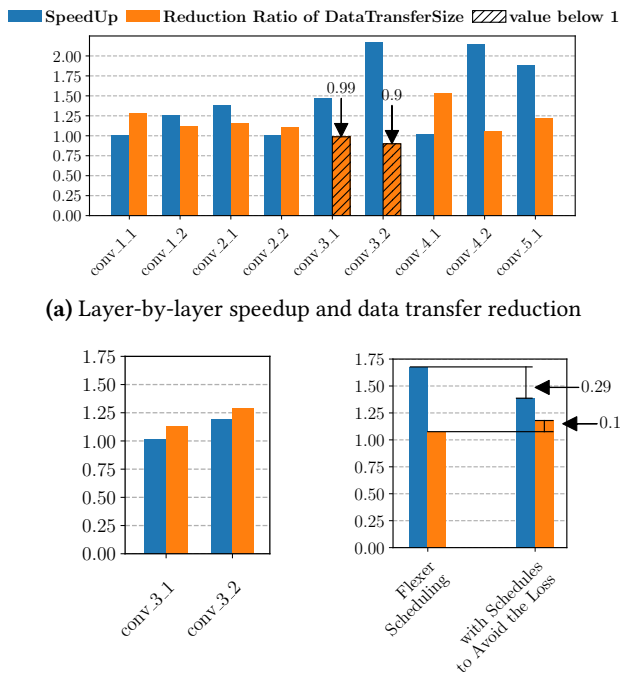


Figure 8. Speedup and data transfer reduction of *Flexer* compared to the best static loop-order schedule

compared to the baseline. This result is caused by the performance metric used during scheduling which is defined as $performance = latency \times datatransfersize$. With this metric, the large benefit of lower-latency schedules comes at a (slight) increase in the amount of transferred data. Note, however, that the performance metric in the priority-based list scheduler can easily be adjusted to particular goals. As an example, Figure 9 (b) shows the generated schedules for the two layers `conv_3_1/2` with weighting reductions in data transfers higher than performance improvements. The resulting schedules do achieve a significant reduction in the amount of data transferred at the expense of a much diminished performance improvement. Also for the end-to-end network performance, the focus on data transfers only does not seem desirable (Figure 9 (c)).



(a) Layer-by-layer speedup and data transfer reduction

(b) Schedules minimizing data transfers

(c) *Flexer* default vs minimal data transfer policy

Figure 9. Speedup and data transfer reduction of *Flexer* compared to the best loop-order schedule for VGG16 on arch5

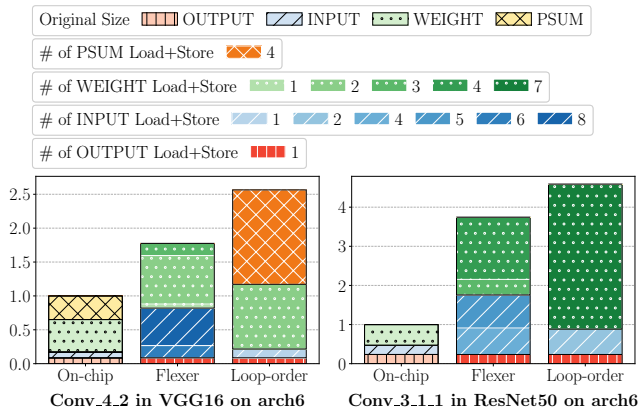


Figure 10. Amount of transferred data and reload count for each data type for on-chip, *Flexer*, and the best static loop-order schedule

To understand how *Flexer* manages on-chip memory, Figure 10 visualizes the data movements between the off- and on-chip memory for one layer from VGG16 and ResNet50, respectively on `arch6`. The different types of data (input, output, weights, and partial sums) are colored differently. The saturation of the color indicates the number of times a tile is moved from one memory to another. The bar *on-chip* represents the best schedule for an unlimited on-chip memory, i.e., all data tiles are moved at most once. The height of the bars represent the amount of data transferred.

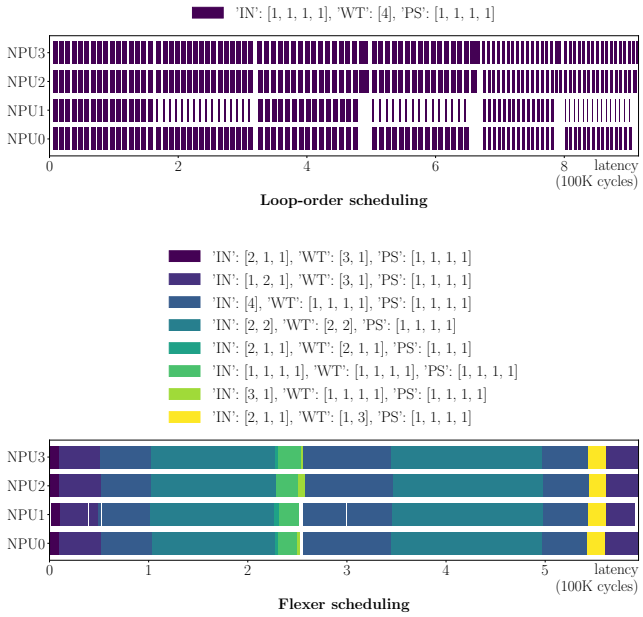


Figure 11. Visualization of spatial data reuse between NPUs of a single layer

We observe how the static loop-order schedules keep one type of data on-chip: the schedule for `conv_4_2` from VGG16 uses an input-stationary schedule, while the schedule for `conv_3_1_1` from ResNet50 keeps the partial sum on-chip. Consequently, the other types of memory exhibit significant data transfers with high reload counts. The regular structure of the loop also dictates that all tiles of a given type are reloaded the same number of times, i.e., there is no reload variation for a given data type. The out-of-order schedules generated by *Flexer*, on the other hand, contain different data flow patterns that result in different reload counts for the same type of data. More specifically, while single-core NPUs only benefit from temporal reuse (from one operation to the next) in the on-chip memory, multi-NPU architectures allow data reuse both in the temporal and spatial (between NPUs) dimension. The out-of-order schedules generated by *Flexer* are able to exploit the best data reuse patterns for any given situation, causing various reload pattern during execution of a single layer.

Figure 11 visualizes spatial data reuse, i.e., simultaneous reuse of data by multiple NPUs. Unsurprisingly, only one type of data (`WT`) is shared in the stationary data flow. *Flexer*'s out-of-order scheduler generates a number of data reuse patterns. The different patterns reveal that different types of data are shared during the execution of a single layer; a characteristic of OoO schedules that is not possible with loop-order schedules.

By modifying the priority function, list schedulers can be tailored towards a given optimization goal. Figure 12 shows the effect of four priority functions listed in Table 2 with respect to the metric $latency \times amount\ of\ transferred\ data$

Table 2. Different priority functions and memory management policies

Priority1	Min transfer: Select the set that causes the minimal amount of data movements
Priority2	Min spilling: Select the set that causes the lowest amount of spilled data
MemPolicy1	First-fit spill: spill the first data block large enough to hold the requested data
MemPolicy2	Small spill: repeatedly spill the smallest data blocks until sufficient memory is available

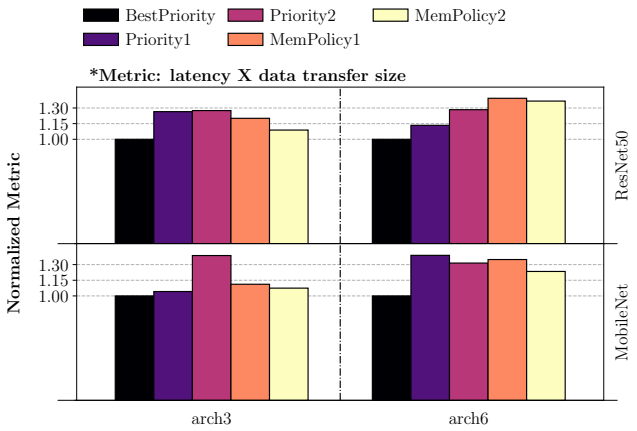


Figure 12. Performance of different priority and memory management policies

(lower is better). All results are normalized to *Flexer*'s default priority function discussed in Section 4. The results show (1) that good memory management has a higher impact than operation set selection and (2) that the default priority function of *Flexer* achieves overall good results for different networks and hardware configurations.

6 Related Work

As illustrated by Figure 1, the amount of data transferred between off- and on-chip memory and the latency of a schedule vary significantly on same hardware in dependence of the execution schedule. Consequently, many works aim to find the fastest schedule for a given hardware. Exploiting the loop representation of a DNN layer, many prior works try to find the best DNN schedule by selecting the optimal tiling size and loop permutations.

A number of approaches employ self-tuning algorithms to solve the scheduling problem. While ConfuciuX [13], FlexTensor [26] and AutoTVM [2] use machine learning algorithm, Ansor [25] and GAMMA [14] rely on genetic algorithms to find the optimal schedule. Compared to genetic algorithms, machine learning technique are better able to avoid getting stuck in local minima; however, pruning the enormous design space may cause the technique to miss the best schedule. Genetic algorithm, on the other hand, are better at covering the entire search space and are less at risk of missing the best schedules compared to machine learing and heuristics-based pruning. Interstellar [22], DMazeRunner [6] and PolySA [5] explore more flexible loop structures through various loop representations of DNN layers. CoSA [11] and Vaidya et al. [21] generate an execution schedule for a DNN in a relatively short time thanks to a constrained optimization solver. All of the above approches, however, are based on fixed loop-order schedules that risk missing better schedule opportunities. The presented OoO scheduling technique is not bound by a loop order and schedules operations based on an analysis of the data-flow graph of a layer.

Several works present scheduling techniques for multi-NPUs or scalable accelerators. In TANGRAM [7], the authors propose an algorithm that reduces redundant operations and data duplication between cores, however, the generated schedules still adhere to loop-order schedules. On the other hand, Zheng et al. [27] propose an atomic dataflow in which the tiled convolution is scheduled in accordance to a pre-defined priority rule to maximize data reuse while minimizing latency. However, Zheng et al. [27] does not explore all possible tilings and this risks missing better schedules. Additionally, Zheng et al. [27] base the priority of operations with heuristics while our work considers the actual memory status and overhead.

In contrast, the presented out-of-order scheduler combined with its memory management techniques and operation set pruing allow *Flexer* to explore new, irregular execution schedules that achieve significant performance improvements over fixed loop-order schedules.

7 Conclusion

This work presented *Flexer*, a DNN scheduling technique based on out-of-order scheduling that can exploit opportunity missed by regular loop-structure schedules. *Flexer* selects the best combination of parallel operations from a list of sets based on a priority function similar to traditional priority-based list schedulers. Results with various deep neural networks on different multi-NPU architectures show that *Flexer* is able to generate significantly more efficient execution schedules than state-of-the-art loop-based schedulers.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was funded, in parts, by the Neural Processing Research Center (NPRC) of the Samsung Advanced Institute of Technology (SAIT), and by the Korean National Research Foundation through grants 2022R1F1A1074967, 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), and 10077609 (MOTIE/KEIT). ICT at Seoul National University provided research facilities for this study. Bernhard Egger is the corresponding author for this work.

References

[1] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

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

[3] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan 2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[4] Antonio Cipolletta and Andrea Calimera. 2021. Dataflow Restructuring for Active Memory Reduction in Deep Neural Networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 114–119. https://doi.org/10.23919/DATE51398.2021.9473965

[5] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240838

[6] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. DMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 70 (oct 2019), 27 pages. https://doi.org/10.1145/3358198

[7] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 807–820. https://doi.org/10.1145/3297858.3304014

[8] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. 2020. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 681–697. https://doi.org/10.1109/MICRO50266.2020.00062

[9] Barend Harris, Inpyo Bae, and Bernhard Egger. 2018. Architectures and algorithms for on-device user customization of CNNs. *Integration* (2018). https://doi.org/10.1016/j.vlsi.2018.11.001

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[11] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 554–566. https://doi.org/10.1109/ISCA52012.2021.00050

[12] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[13] S. Kao, G. Jeong, and T. Krishna. 2020. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 622–636. https://doi.org/10.1109/MICRO50266.2020.00058

[14] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[15] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 343–348. https://doi.org/10.23919/DATE.2018.8342033

[16] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/3020078.3021736

[17] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016). arXiv:1612.08242 http://arxiv.org/abs/1612.08242

[18] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3352460.3358302

[19] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

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

[21] Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2022. Comprehensive Accelerator-Dataflow Co-design Optimization for Convolutional Neural Networks. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 325–335. https://doi.org/10.1109/CGO53902.2022.9741281

[22] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. *Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators*. Association for Computing Machinery, New York, NY, USA, 369–383. https://doi.org/10.1145/3373376.3378514

[23] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[24] Yipeng Zhang, Bo Du, Lefei Zhang, and Jia Wu. 2020. Parallel DNN Inference Framework Leveraging a Compact RISC-V ISA-Based Multi-Core System. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 627–635. https://doi.org/10.1145/3394486.3403105

[25] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. *Ansor: Generating High-Performance Tensor Programs for Deep Learning*. USENIX Association, USA.

[26] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

[27] Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2022. Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 475–489. https://doi.org/10.1109/HPCA53966.2022.00042

Received 2022-09-02; accepted 2022-11-07