# Optimal Model Partitioning with Low-Overhead Profiling on the Heterogeneous Platform for Deep Learning Inference

JAEWOOK LEE, SEOK YOUNG KIM, YOONAH PAIK, CHANG HYUN KIM, WON JUN LEE, and SEON WOOK KIM*, Korea University, Korea

The heterogeneous platform, including CPUs, GPUs, accelerators, and even recently introduced Processing-in-Memory (PIM), is used in many areas to achieve energy-efficient and high-performance computation. In most Deep Learning (DL) frameworks, a user manually partitions a model's computational graph onto the computing devices by considering the devices' capability and the data transfer between the devices. The Deep Neural Network (DNN) models are becoming more and more complex for improving accuracy; thus, it is exceptionally challenging to determine the partitioning for achieving the best performance, especially on a heterogeneous platform.

This paper proposes two novel algorithms for the DL inference to resolve the challenge: a low-overhead profiling algorithm and an optimal model partitioning algorithm. First, we reconstruct a computational graph (CG) by considering the devices' capability to represent all the possible scheduling paths on the platform. Second, we develop a profiling algorithm to find the required minimum profiling paths to measure all the node and edge costs of the reconstructed CG. Finally, we devise the model partitioning algorithm to get the optimally minimum execution time using the dynamic programming technique with the profiled data. We evaluated our work by executing the BERT, RoBERTa, and GPT-2 models on the ARM multicores with the PIM-modeled FPGA platform with a batch size 1 and a sequence length of 8 to 64. For three computing devices in the platform, i.e., CPU serial, CPU parallel, and PIM executions, we could find all the costs only in four profile runs, three for node costs and one for edge costs. Also, our model partitioning algorithm achieved a speedup of 1.1x~2.1x and 1.1x~3.0x compared to the execution with manually assigned device priority of (CPU parallel, PIM, and CPU serial) and (PIM, CPU parallel, and CPU serial), respectively.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Neural networks**.

Additional Key Words and Phrases: PIM-based heterogeneous platform, Optimal scheduling, Profiling, Computational Graph

**ACM Reference Format:**
Jaewook Lee, Seok Young Kim, Yoonah Paik, Chang Hyun Kim, Won Jun Lee, and Seon Wook Kim. 2022. Optimal Model Partitioning with Low-Overhead Profiling on the Heterogeneous Platform for Deep Learning

*Jaewook Lee and Seok Young Kim contributed equally to this study.

This paper was the result of the research project supported by SK Hynix Inc and the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean goverment (MSIP) (No.2022-0-00050, Development of PIM Computing Architecture based on Data-Flow).

The corresponding author: Seon Wook Kim.
Authors' address: Jaewook Lee, jake8542@korea.ac.kr; Seok Young Kim, tjrdud513@korea.ac.kr; Yoonah Paik, yoonpaik@korea.ac.kr; Chang Hyun Kim, huterkch@korea.ac.kr; Won Jun Lee, rriiaa@korea.ac.kr; Seon Wook Kim, seon@korea.ac.kr, Korea University, 145, Anam-ro, Seongbuk-gu, Seoul, 02841, Korea.

1084-4309/2022/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Inference. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 1 (January 2022), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The heterogeneous platform, accommodating various computing devices, such as CPUs, GPUs, ASICs, FPGAs, and so on, has been widely used for energy-efficient and high-performance computation [7, 11, 31, 34] of a broad class of applications [10, 29, 33]. Recently, with the emergence of data-intensive and low locality applications in Deep Learning (DL), e.g., LSTM [13] and RNN [14], Processing-in-Memory (PIM) [16, 19] is also adopted to the platform, thus increasing heterogeneity.

To achieve the advantage by fully utilizing the heterogeneity, we should consider both each device's computation performance and the data transfer overhead incurred when a device accesses data in other devices at the scheduling. For example, CPU and GPU generally have their own memory space on a platform, requiring an explicit data copy from one memory to another to maintain data consistency and sometimes resulting in significant overall performance degradation due to the slow PCIe interface [20]. It is really challenging to schedule, i.e., partition the computation optimally for each device on a heterogeneous platform while considering its computation capability and associated data transfers because of many possible execution path candidates.

The previous partitioning studies took two approaches: cost model-based partitioning [2, 3, 18, 22, 30] and ML-based partitioning [8, 9, 17]. The first approach measured the costs by profiling the applications and partitions the execution using the cost model. The cost was often estimated because too many possible profile execution paths exist. The estimation approximated the computation cost through curve-fitting and the data transfer cost by dividing the data size by memory bandwidth. The cost model used the computation and data transfer costs and partitioned the application by determining whether to stay on one device or move to another. The second approach built the dataset and trained the ML model. The dataset was obtained by extracting features from the static code analysis and profiling with different input data sizes and possible device combinations. Then, the ML model was trained using the datasets to obtain the optimal partition. Finally, the ML model provided the optimal partition for a given application.

The recent DNN models' high complexity in computation makes it more challenging to profile the programs [6, 21, 28], thus making it difficult to find the optimal model partition for the best performance. For example, ONNX Runtime [26] represents the computation as a computational graph of CG($V_c$,$E_c$), consisting of $V_c$ nodes representing operators and $E_c$ edges describing tensors, and CG is a Directed Acyclic Graph (DAG). The BERT-small model [6] is composed of 222 nodes and 262 edges in CG: The complex CG involves many possible profiling paths, increasing exponentially as the number of devices increases. Therefore, it would not be applicable in polynomial time to profile all possible scheduling paths for measuring all the node and edge costs and identify an optimal scheduling path from the profiled costs. In most current DL frameworks [1, 27], users manually partition the execution, so it is hard to obtain the best performance; for example, by marking which nodes to map with which devices in PyTorch [27] and TensorFlow [1] and defining the computing device's priority and capability for each operator in ONNX Runtime.

*This paper proposes two novel polynomial-time algorithms for optimally running an inference DNN model on a heterogeneous platform: one for profiling to recognize the minimum number of execution paths to measure all the costs and the other for partitioning to achieve the best execution performance with the profiled costs.*

Our approach consists of the following three steps. First, we build a Device-mapped Computational Graph, DCG($V_d$,$E_d$) from CG($V_c$,$E_c$), to represent that a node shows a pair of an operator and its device capability, and each edge does the tensor's data transfer between devices, where

$|V_d| = O(V_c \times n)$, $|E_d| = O(E_c \times n^2)$, and $n$ is the number of computing devices on the target platform.

Second, we profile the DCG to measure all the node and edge costs. The node cost represents the computation time of an operator on a device, and the edge cost implies the data transfer time between devices. After measuring all the costs, we substitute the partitioning problem with finding the minimum cost path from a start to an end node in DCG. Measuring all the node costs needs $n$ profile runs in DCG. However, it is challenging to measure all the edge costs by profiling since there are $n^{V_d}$ possible execution paths, leading to exponential application runs. Our polynomial-time profiling algorithm tracks the edges instead of execution paths since the number of edges in DCG is polynomial. Furthermore, we reduce the number of profiled edges by considering edge attributes. The data transfer between devices usually uses DMA; thus, we classify all the edges by three attributes, occurring the same data transfer cost: a source device, a destination device, and a data transfer size. Then, we develop a profiling algorithm to identify the minimum number of execution paths to include all the distinct attribute edges, resulting in the polynomial-time complexity of $O(V_d + E_d^2)$.

Finally, we apply a dynamic programming technique to find the optimal model partitioning from the profiled costs for achieving the best execution performance. We prove that our partitioning problem is the same as the Assembly Line Scheduling (ALS) problem [5], and therefore, the complexity of our partitioning algorithm is $O(V_d)$.

We implemented our profiling and partitioning algorithms on the ONNX Runtime framework [26]. We used a PIM-modeled FPGA [16] and ARM Cortex-A53 as our experimental heterogeneous platform, i.e., three computing devices to be scheduled as the CPU serial, the CPU parallel on the multicores, and the PIM execution for targeting memory-intensive and low locality applications. To the best of our knowledge, our work is the first to address the optimal model partitioning in the PIM-based heterogeneous platform. We evaluated the performance by running three Transformer-based models, i.e., BERT [6], RoBERTa [21], and GPT-2 [28].

We needed at least three profiling runs to measure all the node costs for our computing platform, i.e., three devices. Additionally, our edge profiling algorithm identified only one execution path to find all the edge costs in all the models. We analyzed the operator-by-operator performance of the models and found that PIM outperformed the others in most operators. However, the PIM incurred the data transfer cost; thus, we should carefully assign the operators to devices to achieve the best overall performance. For the detailed performance analysis, we manually made two execution priority orders of (CPU parallel, PIM, and CPU serial) and (PIM, CPU parallel, and CPU serial). Using the profiled costs and applying our optimal model partitioning algorithm, we achieved a speedup of 1.1x~2.1x and 1.1x~3.0x compared to the execution with manually assigned device priority orders, respectively.

The remainder of the paper is organized as follows: Section 2 introduces background about our experimental platform, including the DL framework and the PIM computing device. Section 3 proposes our low-overhead profiling and optimal model partitioning algorithms. Section 4 presents the performance evaluation. Section 5 describes the related work, and Section 6 concludes the paper.

## 2 BACKGROUND

This section reviews our experimental platform's ONNX Runtime framework and the PIM computing device.

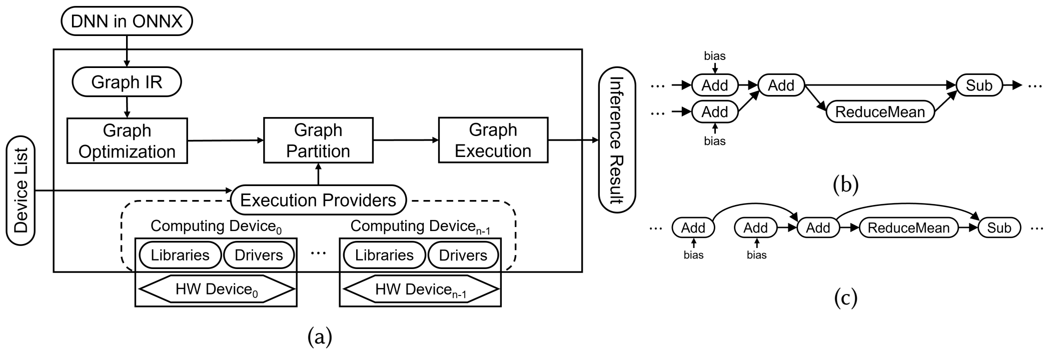


Fig. 1. (a) Execution flow of the ONNX Runtime framework. (b) A computational graph (CG). (c) A topologically sorted CG.

## 2.1 ONNX Runtime Framework

Open Neural Network Exchange (ONNX) [25] is an open format that represents a DNN model for providing interoperability between DL frameworks, such as TensorFlow [1] and PyTorch [27]. The DNN model implemented in one framework can be exported to the ONNX format and used in another. ONNX Runtime is a framework that supports and runs the ONNX format on multiple devices by adopting an execution provider interface, allowing us to conveniently integrate various devices by abstracting a computing device and its execution environment, including the libraries and drivers.

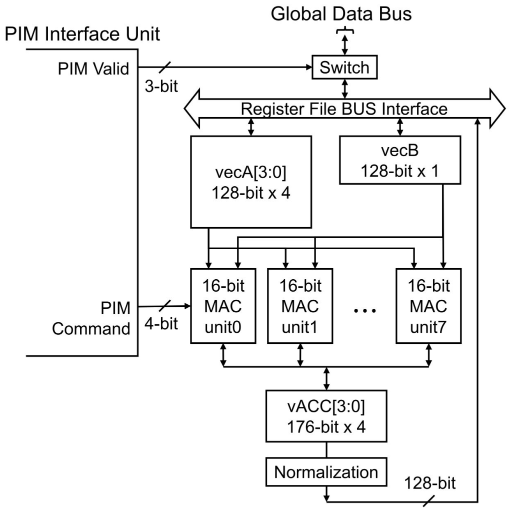
Fig. 1(a) shows the execution flow of ONNX Runtime deploying multiple computing devices. ONNX Runtime transforms the ONNX format of a DNN model into Graph IR, i.e., a computational graph (CG) and its topologically sorted CG, as shown in Fig. 1(b) and (c). A node of CG represents an operator, and an edge implies a tensor representing data movement (dependence) between nodes in the form of a multi-dimensional array or a vector. ONNX Runtime performs graph optimization, partition, and execution in order. The graph optimizer applies various hardware-dependent and independent optimizations to CG. The graph partitioner maps each node to one of the computing devices in a device list by considering the user-defined device priority and capability. It also inserts a memory copy node if a device uses the output from other devices. Finally, the graph execution stage traverses the CG nodes in topological order and assigns them to the scheduled computing devices for execution.

A user specifies a list of available computing devices in the framework, i.e., device capability. Also, the user prioritizes the computing devices in the execution provider list for specifying the device execution preference. All the specification solely depends on the user's knowledge of the devices.

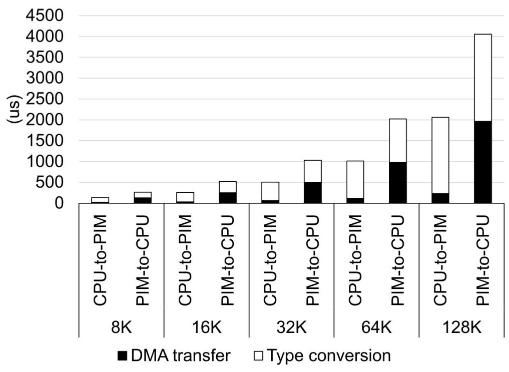
## 2.2 PIM Device: Silent-PIM

We used Silent-PIM as the PIM device on our experimental heterogeneous platform [16], which satisfies the standard memory interface [15] and performs the PIM execution using the standard memory request.

Fig. 2(a) shows the Silent-PIM's datapath, supporting bfloat16 8-way vector addition, subtraction, multiplication, and MAC operations. There are 128-bit×4 *vecA* and 128-bit×1 *vecB* general registers and *vACC* accumulator registers. The *vecA* register holds the 4-cycle burst data, i.e., 64 bytes, and the *vecB* stores data at every cycle during the burst. Whenever *vecB* stores 16-byte at every



(a)



(b)

Fig. 2. (a) Silent-PIM datapath [16]. (b) Data transfer cost in time from CPU to PIM and PIM to CPU devices when varying the element sizes.

cycle, Silent-PIM performs the PIM operations with 8-way vector units. Silent-PIM performs a matrix-matrix (MM) multiplication, $A \times B = C$, by repeating a vector-matrix (VM) multiplication column-wise by the number of rows of $A$. The element-wise vector/matrix execution is more straightforward than the VM multiplication.

In most DNN models, the VM multiplication and element-wise operation read source operands only once, thus exploiting the low locality and resulting in higher PIM performance than CPU. However, if the source operands are available in caches, the CPU *may* outperform PIM. In the MM multiplication, Silent-PIM repeats the VM multiplication, so we prefer to use the CPU over PIM due to data locality. However, if source operands are available only in memory, PIM *may* deliver higher performance than CPU. In addition, PIM does not support the computation of complex functions, such as *exp*, *tanh*, and so on. In summary, the performance depends on the source operand location and the operator type. Therefore, we should carefully offload functions to devices, i.e., partitioning workloads, to achieve the best performance on the heterogeneous platform.

## 2.3 Data Transfer Cost

Our experimental heterogeneous platform includes two computing devices, CPU and Silent-PIM, as shown in Fig. 7, and uses DMA for data transfer between them.

The DMA data transfer time is affected by the following four factors: a data type, a transfer size, a source device, and a destination device. Silent-PIM uses bfloat16, and CPU uses float32 types: therefore, we need a data type conversion before the data transfer between PIM and CPU. The DMA data transfer time is proportional to the data size [23]. Also, we configure the PIM memory as uncacheable and the CPU memory as cacheable [16]. Due to memory ordering, uncacheable memory access takes much longer than cacheable access.

Fig. 2(b) shows the data transfer cost from CPU to PIM and vice versa when varying element sizes, consisting of the type conversion time and the DMA transfer time. The type conversion in the two cases took a similar time since the CPU performed the conversion. However, the DMA

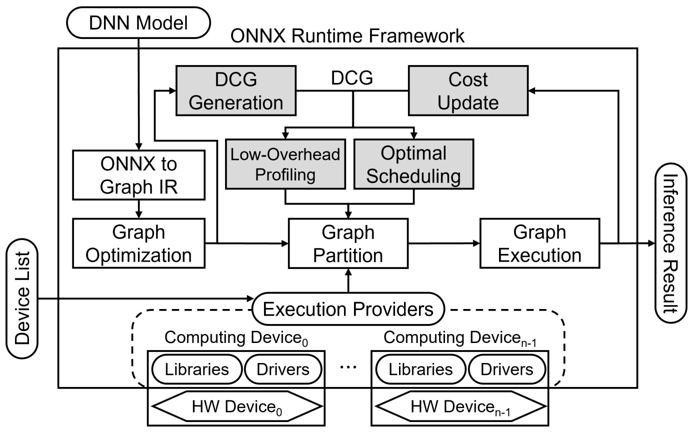


Fig. 3. The extension of the ONNX Runtime framework from Fig. 1. We added the gray-colored components for our work.

transfer time differed depending on the source and destination devices, and CPU-to-PIM performed faster than PIM-to-CPU. The CPU reads the source data from fast cacheable memory, and the PIM does from slow uncacheable memory. It should be noted that the data transfer size is the same after the type conversion, i.e., 2 bytes per element.

## 3 LOW-OVERHEAD PROFILING AND OPTIMAL MODEL PARTITIONING

Fig. 3 shows an overall architecture of the ONNX Runtime by adding the gray-colored components to Fig. 1 for realizing our low overhead profiling and optimal DL inference model partitioning on the heterogeneous platform.

After the graph optimization, we generate DCG from CG using the computing device's capability defined in execution providers. Our low-overhead profiling algorithm finds the minimum number of execution paths to measure all the node and edge costs in DCG, thus resulting in the lowest profile overhead. Then, from the profile runs, we apply our optimal partitioning algorithm to find an execution path to guarantee the minimum execution time. The rest of this section explains each step in detail.

### 3.1 Reconstructing Computational Graph for Heterogeneous Computing

The original graph partitioner allocates each CG node to one of the computing devices by considering the user-defined priority, i.e., device preference, without considering the device's operator cost and data transfer cost. Therefore, the performance heavily relies on the user's experience, which would fail to provide the desired performance. Also, allocating a node's successor to a different device requires inserting a memory copy node between them.

In order to involve the computing nodes on a heterogeneous platform for the model partitioning, we develop DCG from CG, representing the devices' operator capability and their data dependencies, as shown in Fig. 4 from CG in Fig. 1(a) with $n$ computing devices. The solid arrow edge ($\longrightarrow$) represents the data dependence in the same device, and the dashed arrow edge ($\dashrightarrow$) shows the data dependence between different devices, requiring explicit data transfer. We also add a start node and an end node for providing a single entry and exit in DCG.

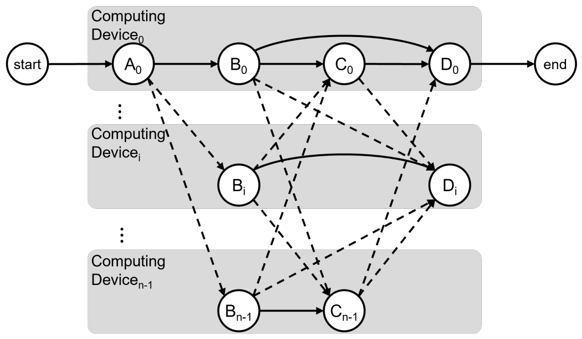


Fig. 4. DCG from Fig. 1(b) for $n$ computing devices on a heterogeneous computation.

## 3.2 Profiling DCG with the Lowest Overhead

Without the loss of generality, we assign zero cost to the start and the end nodes because they are dummy operators and to the solid edges because the tensors are in the same device. We acquire the costs of the rest of the nodes and the dashed edges by profile runs.

We classify the profiling run into node cost profiling and edge cost profiling. The node cost profiling is straightforward; we can identify all the node costs with the profiling runs as many as the number of devices on our target platform, i.e., $n$, by assigning the highest priority to each computing device in the device list one by one. For example, in Fig. 4, we need $n$ profile executions for acquiring all the node costs, i.e., $A_0B_iC_0D_i$ by assigning the highest priority to device 0 to $n-1$. From the node cost profiling, we can measure some edge costs. However, the node profiling runs cannot measure all the edge costs.

Profiling all the edge costs is problematic because the exponential number of execution paths exists from the start to the end nodes, i.e., $O(n^{|V_d|})$ paths in DCG. In this case, we cannot finish the process in the polynomial-time even if we develop a polynomial-time algorithm due to the exponential input size. We may use a book-keeping approach to reduce the working path size since many paths involve the same subpaths. However, the path size still grows exponentially since all the nodes may have $n$ children. Therefore, instead of managing paths, we track edges since the number of edges in DCG is $O(n^2 \times V_c) = O(n \times V_d)$, i.e., a polynomial input size. Then, we topologically visit each DCG node to find the minimum number of execution paths covering all the edges.

*3.2.1 Reducing the Problem Size.* We reduce the problem size, i.e., the number of edges to be profiled, by identifying distinct attribute (DA) edges. In general, we use the DMA for data transfer between devices; thus, the data transfer cost is determined by four attributes, as discussed in Section 2.2: a data type, a data size, a source device, and a destination device. Therefore, we do not need to profile the edges with the same attribute repeatedly; we profile each DA edge only once.

Table 1 shows the DCG's number of edges and DA edges in the transformer-based models for two devices (CPU, PIM). For example, the DCG edges($E_d$) in BERT, RoBERTa, and GPT-2 decreased from 221, 628, and 527 to only 8, 8, and 7 DA edges, respectively. We observed that most nodes' operand dimensions are constant, thus implying the transfer size is constant and allowing us to have the significantly small amount of DA edges from DCG edges. The number of *ReduceMean* and $SoftMax$ nodes to reduce the data dimension is small.

*3.2.2 Edge Profiling Algorithm.* We use Fig. 5 as an example for explaining the edge profiling algorithm with two computing devices, device 0 and 1. A subgraph of the initial DCG appears in

Table 1. The number of edges and DA edges in DCG from the transformer-based models for two devices.

| | BERT | RoBERTa | GPT-2 |
|---|---|---|---|
| DCG edges ($E_d$) | 221 | 628 | 527 |
| DA edges | 8 | 8 | 7 |

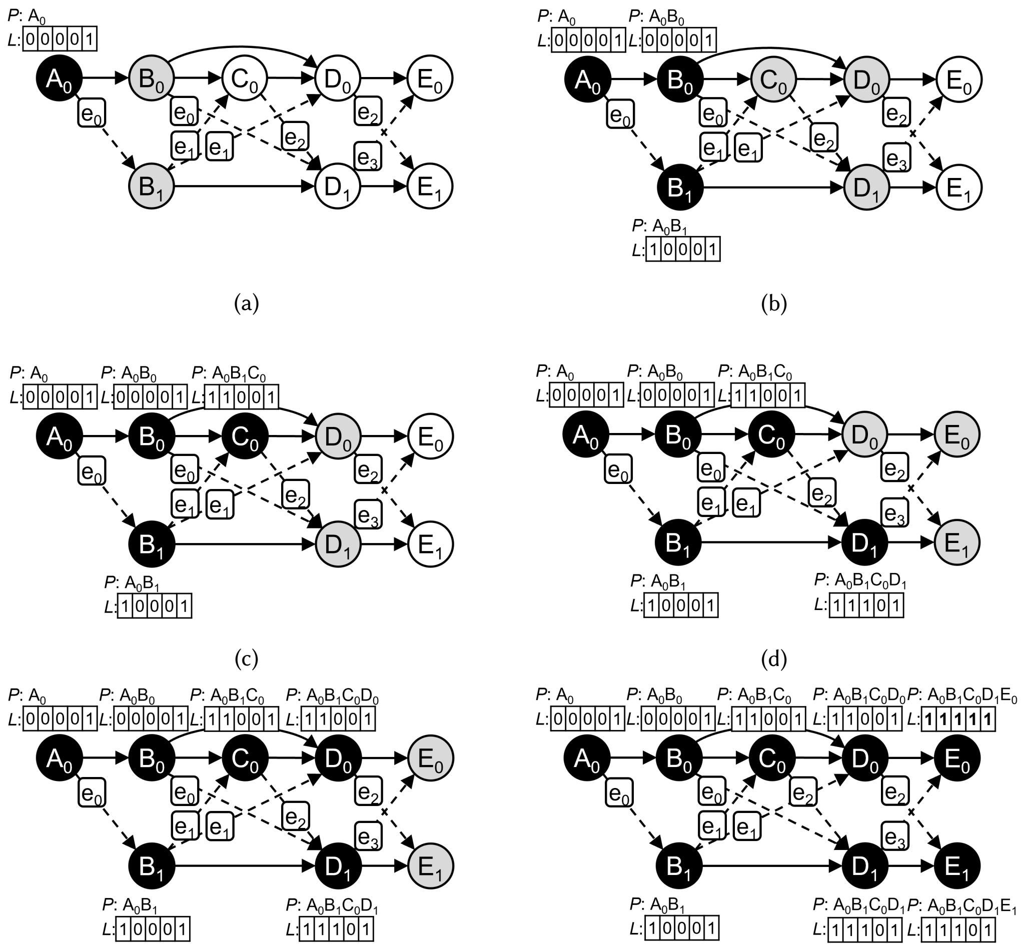


Fig. 5. An example of low-overhead edge profiling applied to DCG with a profiled edge list ($L$) and its profiled path ($P$). The topological order is $[A_0, (B_1, B_0), C_0, (D_1, D_0), (E_1, E_0)]$, and the parentheses represent the siblings. (a) Initial at $T_0$ (b) $T_1$ and $T_2$ (c) $T_3$ (d) $T_4$ (e) $T_5$ (f) $T_6$ and $T_7$.

Fig. 5(a), where there are five DA edges ($e_0 \sim e_4$), and operators $A$ and $C$ are incapable for device 1. The node profiling identifies the edge cost of $e_4$.

For the edge profiling algorithm, we augment two data structures into each node: a profiled edge list ($L$) and its associated path, i.e., the profiled path ($P$). The edge list is the list of the distinct edges in DCG, and we construct it when building DCG. In the figure, the list represents $[e_0, e_1, e_2, e_3, e_4]$.

We initialize each node's field in the profiled edge list with 0's and set a field associated with the incoming edges during the node visit. The profiled path is a path that contains the marked DA edges in a profiled edge list to be executed from the start node. A node with more than one incoming edge may have multiple pairs of a profiled edge list and a profiled path.

We show Algorithm 1 to identify the minimum profiling paths for recognizing all the node and edge costs in DCG. The profiling path identification algorithm performs the topological sort of DCG (Line 1) and the node cost profiling (Lines 4~8). We can sort DCG without any overhead since we can use CG's available topological sort information; we need only to consider the devices as siblings in the order. The algorithm visits the DCG nodes in the topological order for the edge cost profiling (Line 10). At every node visit, the algorithm generates the node's profiled edge list and its profiled path by considering the DA edges in the explored path with the predecessor's information (Lines 15~25). When we mark all the DA edges, we stop the algorithm execution (Lines 12~14) and use all the collected profiled paths for profile runs (Line 20).

In the example figure, we initially color all the nodes as white, a dequeued one from the topological FIFO queue as black, and currently processed nodes as gray. In Fig. 5(a) at $T_0$, we assume that we already visited the $A_0$ node, thus dequeuing $A_0$ and having $[0, 0, 0, 0, 1]$ in the profile edge list ($L(A_0)$) and $A_0$ in the profile path ($P(A_0)$). Remember that we profiled $e_4$ from the node profiling.

Fig. 5(b) shows two steps to dequeue $B_1$ and $B_0$: At $T_1$, we dequeue $B_1$ (Line 11) and generate the $[1, 0, 0, 0, 1]$ profiled edge list ($L(B_1)$) by adding the explored $e_0$ DA edge to the $A_0$'s profiled edge list and $A_0B_1$ of the profiled path ($P(B_1)$) (Lines 16~18). Similarly, at $T_2$, we dequeue $B_0$ and generate the $[0, 0, 0, 0, 1]$ profiled edge list ($L(B_0)$) the same as $A_0$'s because there is no DA edge from $A_0$ to $B_0$ (Line 23). We also generate $A_0B_0$ of the profiled path ($P(B_0)$) (Line 16).

In Fig. 5(c) at $T_3$, we dequeue $C_0$ to have two incoming execution paths; thus, we compare the inclusion of the generated profiled edge lists from the paths, i.e., summarize the results. One path from $B_0$, including no new DA edge, generates $[0, 0, 0, 0, 1]$ for $L(C_0)$, $B_0C_0$ for $P(C_0)$. The other path from $B_1$, including a new DA edge of $e_1$, generates $[1, 1, 0, 0, 1]$ for $L(C_0)$, $B_1C_0$ for $P(C_0)$. Since $[1, 1, 0, 0, 1]$ includes $[0, 0, 0, 0, 1]$, i.e., covers more DA edges, we only maintain $[1, 1, 0, 0, 1]$ with its profiled path (Lines 19~21).

In Fig. 5(d) at $T_4$, we dequeue $D_1$ to have only one incoming path from $C_0$, including two DA edges of $e_0$ and $e_2$; thus, we generate $[1, 1, 1, 0, 1]$ for $L(D_1)$, $C_0D_1$ for $P(D_1)$ using $L(C_0)$ and $P(C_0)$. In Fig. 5(e) at $T_5$, we dequeue $D_0$ to have only one incoming path from $C_0$, including one DA edge of $e_1$ to be already registered in the predecessor, $C_0$; thus, $D_0$'s profiled edge list and path are the same as $C_0$, i.e., $[1, 1, 0, 0, 1]$ for $L(D_0)$, $C_0D_0$ for $P(D_0)$.

Fig. 5(f) shows two steps to dequeue $E_1$ and $E_0$: At $T_6$, we dequeue $E_1$ and generate the $[1, 1, 1, 0, 1]$ profiled edge list ($L(E_1)$) by adding the explored $e_2$ DA edge to the $D_0$'s profiled edge list and $D_0E_1$ of the profiled path ($P(E_1)$). Similarly, at $T_7$, we dequeue $E_0$ and generate the $[1, 1, 1, 1, 1]$ profiled edge list ($L(E_0)$) by adding the explored $e_3$ DA edge to the $D_1$'s profiled edge list and $D_1E_0$ of the profiled path ($P(E_1)$). We find all the DA edges, and the algorithm stops (Lines 12~14).

We can determine the inclusion between the profiled edge lists at each node visit in $O(E_d)$ since there are at most $E_d$ different profiled edge lists. Also, we can determine whether a path includes new DA edges in $O(1)$. Therefore, the total complexity of our profiling algorithm is $O(V_d + E_d^2)$, which enables finding the minimum paths that contain all the edge costs in polynomial time.

## 3.3 Optimal Model Partitioning

We apply the famous Assembly Line Scheduling (ALS) problem to the optimal partitioning problem in the heterogeneous platform by showing that the two problems are the same. For simplicity, we use only two assembly lines for manufacturing, i.e., two computing devices on the platform.

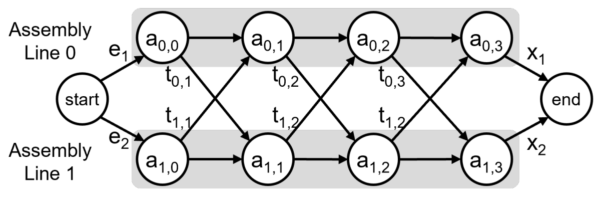
**Algorithm 1:** Profiling Path Identification Algorithm

```
   Input  : Device Mapped Graph($DCG$), Devices($n$)
   Output : ProfileRunPath($P$)
 1 Perform the topological sort of DCG and enqueue the result in Q
 2 /* Identify node cost profiling paths */
 3 Initialize start_node.profiled_edge_list
 4 for all device i do
 5     Set device_priority as [device i, device 0]
 6     Build the execution path and find DA_edges
 7     Add the path to $P$ and DA_edges to start_node.profiled_edge_list
 8 end
 9 /* Identify edge cost profiling paths */
10 while not empty in Q do
11     $u$ = dequeue(Q)
12     if u.profiled_edge_list has all 1's then
13         break /* Done */
14     end
15     for each incoming execution path to u do
16         Add profiled_path to $u$
17         if the path includes new DA_edges then
18             Add profiled_edge_list to $u$ /* Register new DA */
19             if u has more than one profiled_edge_list then
20                 Determine their inclusion and summarize u.profiled_edge_list
21             end
22         else
23             Add the predecessor's profiled_edge_list to $u$ /* Inherit from predecessors */
24         end
25     end
26 end
27 Add u.profiled_path to $P$
```

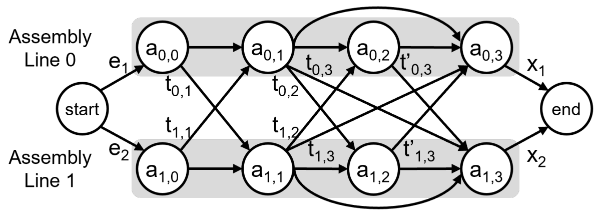
Fig. 6(a) shows the ALS problem in determining which nodes to be selected from assembly lines to minimize the manufacturing time. In the graph, $a_{i,j}$ denotes a cost of the $j$th node on $i$th assembly line, $t_{i,j}$ denotes a transfer cost from the $j-1$th node on $i$th assembly line, and $e_i$ and $x_i$ are an entry cost and an exit cost on the $i$th assembly line. The fastest time to get a chassis from the start to the $j$th node on $i$th assembly line, $f_i[j]$, is expressed by Eq. (1) for $j \geq 1$ where $f_0[0] = e_1 + a_{0,0}$ and $f_1[0] = e_2 + a_{1,0}$.

$$\begin{aligned} f_0[j] &= \min(f_0[j-1] + a_{0,j}, f_1[j-1] + t_{1,j} + a_{0,j}) \\ f_1[j] &= \min(f_1[j-1] + a_{1,j}, f_0[j-1] + t_{0,j} + a_{1,j}) \end{aligned} \tag{1}$$

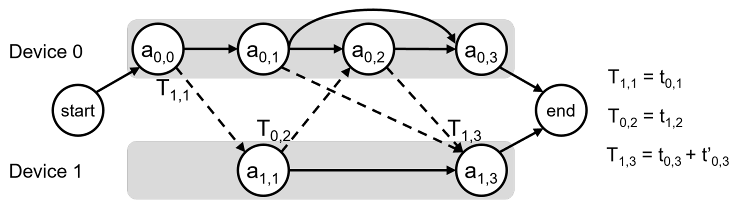
We modified the ALS graph from Fig. 6(a) to Fig. 6(b) by allowing nodes to have more than one incoming edge from a different assembly line (or more than one outgoing edge to a different


Assembly Line 0
Assembly Line 1
start
end
$e_1$
$e_2$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$t_{0,1}$ $t_{0,2}$ $t_{0,3}$
$t_{1,1}$ $t_{1,2}$ $t_{1,2}$
$x_1$
$x_2$

(a)


Assembly Line 0
Assembly Line 1
start
end
$e_1$
$e_2$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$t_{0,1}$ $t_{0,2}$ $t_{0,3}$ $t'_{0,3}$
$t_{1,1}$ $t_{1,2}$ $t_{1,3}$ $t'_{1,3}$
$x_1$
$x_2$

(b)


Device 0
Device 1
start
end
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,3}$
$T_{1,1}$ $T_{0,2}$ $T_{1,3}$
$T_{1,1} = t_{0,1}$
$T_{0,2} = t_{1,2}$
$T_{1,3} = t_{0,3} + t'_{0,3}$

(c)

Fig. 6. Comparison of the ALS and the DCG problems. (a) ALS graph. (b) Modified ALS graph. (c) DCG.

assembly line). Then, Eq. (1) is changed to the following:

$$\begin{aligned} f_0[j] &= \min(f_0[j-1] + a_{0,j}, f_1[j-1] + T_{1,j} + a_{0,j}) \\ f_1[j] &= \min(f_1[j-1] + a_{1,j}, f_0[j-1] + T_{0,j} + a_{1,j}) \end{aligned} \quad (2)$$

$T_{i,j}$ denotes a sum of a transfer cost from all the data-dependent predecessors of $j$th node on $i$th assembly line. For example, $T_{1,3} = t_{0,3} + t'_{0,3}$. The equation still holds the optimal solution since 1) the transfer cost of all the edges across the assembly lines is used only once for finding the optimal value and 2) $t_{i,j}$ represents the transfer cost from the other assembly lines, and therefore, we can replace all the transfer costs from all data dependent predecessors of the $j$th node on $i$th assembly line by $T_{i,j}$.

We can easily map the modified ALS graph to DCG by removing incapable nodes and processing the ALS in topological order. Fig. 6(c) shows an example: The 1st and 3rd operators are not supported by computing device 1; therefore, we remove them from DCG.

## 4 PERFORMANCE EVALUATION

### 4.1 Experimental Environment and Methodology

We developed the heterogeneous platform, including the ARM multicores and Silent-PIM [16], as shown in Fig. 7; thus, we could define three computing devices such as `CPU_S` (CPU serial execution),

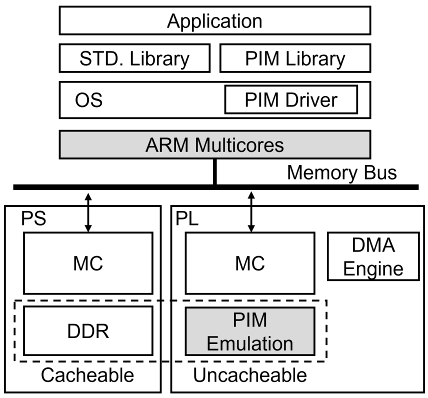


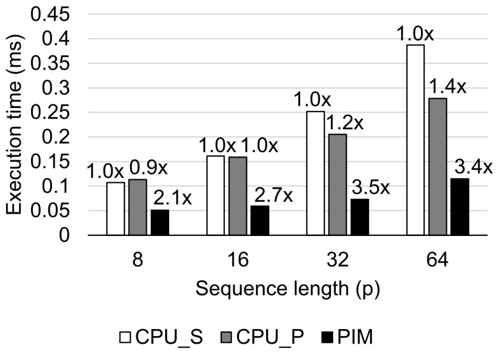
Fig. 7. Our experimental platform on FPGA.

Table 2. PIM executable operators in the transformer-based models.

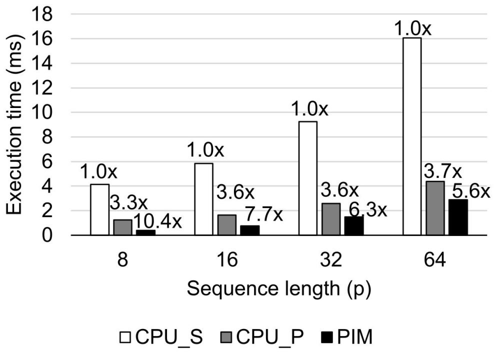
| Operation | Description | | Models | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | BERT | | RoBERTa | | GPT-2 | |
| | | | q | r | q | r | q | r |
| Gemm | (1) | $(p,q) \times (q,r) + (r)$ | - | | - | | 3072 | 768 |
| | | | | | | | 768 | 2304 |
| | | | | | | | 768 | 3072 |
| | | | | | | | 768 | 768 |
| | (2) | $(p,q) \times (r,q)^T + (r)$ | 512 | 2 | 768 | 2 | - | |
| | | | 512 | 512 | 768 | 768 | | |
| MatMul | (1) | $(p,q) \times (q,r)$ | 512 | 2048 | 768 | 3072 | - | |
| | | | 512 | 512 | 768 | 768 | | |
| | | | 2048 | 512 | 3072 | 768 | | |
| Element-wise | (1) | $(p,r) + (p,r)$ | - | 512 | - | 768 | - | 3072 |
| | (2) | $(p,r) + (r)$ | | 1/512/2048 | | 1/768/3072 | | 1/768 |
| | (3) | $(p,r) + (1)$ | | 2048 | | 3072 | | 3072 |
| | (4) | $(p,r) \times (p,r)$ | | 2048 | | 3072 | | 3072 |
| | (5) | $(p,r) \times (r)$ | | 512 | | 768 | | 768 |
| | (6) | $(p,r) \times (1)$ | | 2048 | | 3072 | | 3072 |
| | (7) | $(p,r) - (p)$ | | 512 | | 768 | | 768 |

CPU_P (CPU parallel execution), and PIM (PIM execution) for the experiment. We emulated Silent-PIM on an FPGA board (Xilinx Zynq UltraScale+ board/HTG-Z920) [12]. The ARM SoC of the board consists of two areas: Processing System (PS) and Programmable Logic (PL). The host CPU, Quad-core CPU (ARM Cortex-A53 MPCore/1.5GHz), was on the PS side, while the PIM device was implemented on the PL side. Due to the different operating frequencies between the areas, we scaled the performance. The PS side memory was managed as cachable, while the PL side memory was managed as uncachable. The DMA engine was used for the CPU to offload PIM tasks.

We evaluated the performance of our profiling and partitioning algorithms on the platform, using three state-of-the-art transformer-based models, BERT [6], RoBERTa [21], and GPT-2 [28]. We can exploit CPU_P through the ONNX Runtime library and parallelizable operators. Table 2 shows the operators supported by PIM in the models.



(a)



(b)

Fig. 8. The operators' speedup of the CPU parallel and PIM execution with respect to the serial CPU execution when varying the sequence length ($p$) from 8 to 64. (a) Element-wise addition: (p $\times$ 512) + (p $\times$ 512). (b) Matrix-matrix multiplication: (p $\times$ 512) $\times$ (512 $\times$ 512).

## 4.2 Operator-by-Operator Performance

Before showing the performance of our partitioning algorithm, we compared the operators' performance of CPU_S, CPU_P, and PIM. We chose two operators from Table 2: element-wise addition as ($p \times 512$) + ($p \times 512$) and matrix-matrix multiplication as ($p \times 512$) $\times$ ($512 \times 512$), where $p$ stands for a sequence length. It should be noted that the execution time of all the element-wise operations, such as addition, multiplication, and subtraction, are the same. Also, the execution time of Gemm is a sum of execution time in a matrix-matrix multiplication and an element-wise addition, and the multiplication dominates the overall time.

In the element-wise addition of Fig. 8(a), PIM shows a speedup of 2.1x~3.4x compared to CPU_S and 2.2x~2.8x compared to CPU_P. Since the element-wise addition has no data reuse, all the execution have the same amount of memory requests to DRAM. PIM reads source operands and computes them simultaneously; therefore, its execution is always faster than the others. Also, the execution time of PIM is linearly proportional to $p$, i.e., the matrix size. The performance of CPU_P linearly increases with $p$ due to the large task granularity of threads. For $p = 8$ and 16, the speedup is less or similar to 1.0x due to overhead from parallel code overhead and small task granularity.

PIM also shows higher performance than the others in the matrix-matrix multiplication, i.e., a speedup of 5.6x~10.4x compared to the CPU_S and 1.5x~3.1x to CPU_P, as shown in Fig. 8(b). As discussed in Section 2, PIM performs the matrix-matrix multiplication by repeating the VM multiplication, which requires reading the ($512 \times 512$) matrix by $p$ times and results in performance degradation as $p$ increases. On the other hand, the CPU can fully exploit the locality of the second source operand in a cache, thus acquiring the ideal speedup close to the number of cores, 4. As a result, as $p$ increases, the performance gap between CPU and PIM gets smaller.

## 4.3 Optimal Model Partitioning

We compared the performance using our optimal partitioning with the following two manually prioritizing device orders with respect to the serial execution:

① CPU_S only: CPU serial execution only.
② (CPU_P,PIM,CPU_S): Prioritizing node allocation in the order of CPU parallel, PIM, and CPU serial, the same as assigning all the thread-parallelizable operators to the CPU parallel

execution and the rest to the CPU serial execution. It is because all the PIM operators can be executed by CPU parallel.

③ `(PIM,CPU_P,CPU_S)`: Prioritizing node allocation in the order of PIM execution, CPU parallel, and CPU serial.

④ `OPT`: Our optimal execution using CPU serial, CPU parallel and PIM execution.

Fig. 9 shows the execution time and speedup of running the three transformer-based models while varying a sequence length from 8 to 64 with respect to `CPU_S`. Our optimal execution `OPT` shows a speedup of 1.1x~2.1x and 1.1x~3.0x compared to `(CPU_P,PIM,CPU_S)` and `(PIM,CPU_P,CPU_S)`, manually assigning nodes based on priorities. The speedup of `OPT` is higher than the others in all the test cases, which shows the strength and robustness of our approach.

The speedup of `OPT` decreases as the sequence length increases because the performance gap between `CPU_P` and `PIM` reduces; the execution time of `PIM` is linearly proportional to the sequence length, but the `CPU` exploits the higher cache locality. Therefore, as the sequence length $p$ increases, the less performance gap between `CPU_P` and `PIM` results in a performance decrease in `OPT`. The speedup of `CPU_P,PIM,CPU_S)` is higher than `(PIM,CPU_P,CPU_S)` on BERT and RoBERTa because Gemm requires input weight transpose, which is not a friendly operation for `PIM`.

*4.3.1 BERT and RoBERTa.* Fig. 10 shows the partitioned result of `(PIM,CPU_P,CPU_S)` and `OPT` in BERT and RoBERTa with a sequence length of 16. The `OPT` partitioning assigns the Gemm and the element-wise operators to `CPU_P` instead of `PIM` due to the node and edge costs, respectively. `PIM` takes more time than `CPU_P` at the Gemm execution in Table 2 (2), requiring transposing an input. Also, the division and the erf operators unsupported by PIM incur high data transfer costs for their successors' PIM execution. However, the MatMul performance in `PIM` is superior to the others; thus, the assignment is unchanged.

As a result, `OPT` decreased both the computation time and the data transfer time by 54% and 27% in BERT and 34% and 20% in RoBERTa compared to `(PIM,CPU_P,CPU_S)`, as shown in Fig. 11.

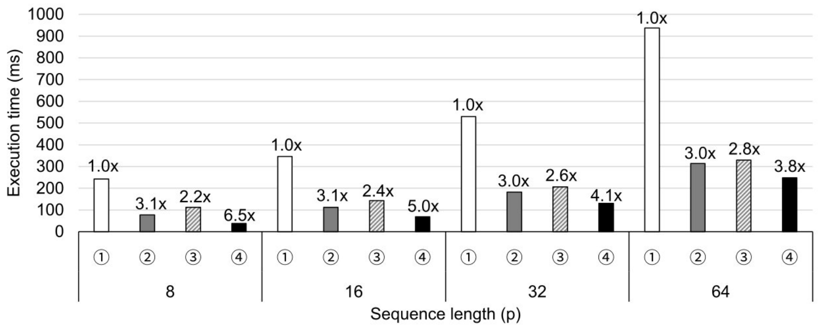
*4.3.2 GPT-2.* Fig. 12 shows the partitioned result of GPT-2 for `(PIM,CPU_P,CPU_S)` and `OPT` when the sequence length is 16. There are two Gemm operators in the decoder that dominates the execution time. Two of them are different in size, where the operator on the bottom is more compute-intensive. The partitioning algorithm assigns Gemm on the top to `CPU_P` and the other to `PIM`. Also, the optimal partitioning assigns the element-wise operations to `CPU_P` to decrease the data transfer.

As shown in Fig. 11(c), their computation times are similar, but data transfer time decreases by 64%, which leads to the optimal execution time.

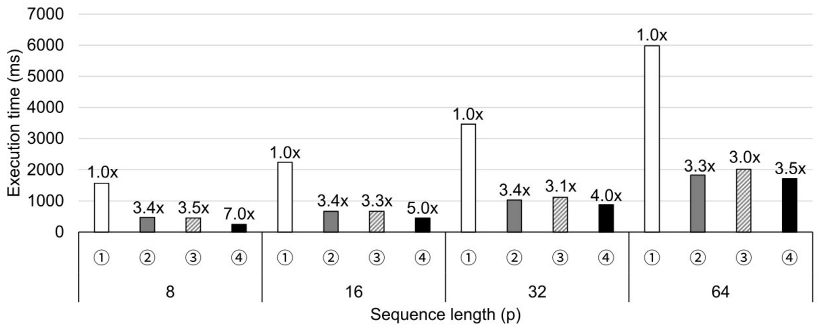
## 4.4 Profiling and Model Partitioning Overhead

Table 3 shows the DCG characteristics and the algorithm overhead. The CGs of BERT, RoBERTa, and GPT-2 have 222, 616, and 1934 nodes and 262, 765, and 2424 edges, respectively. With our experimental platform with three devices, the DCGs built from each model have 469, 1317, and 2593 nodes, 686, 1951, and 1723 edges, respectively. We could reduce them to only 20, 20, and 24 DA edges, significantly reducing the profiling overhead for finding all the edge costs. Moreover, as we identified the cost of the DA edges in the node cost profiling, only four DA edges were left to discover for all three models for the edge cost profiling.

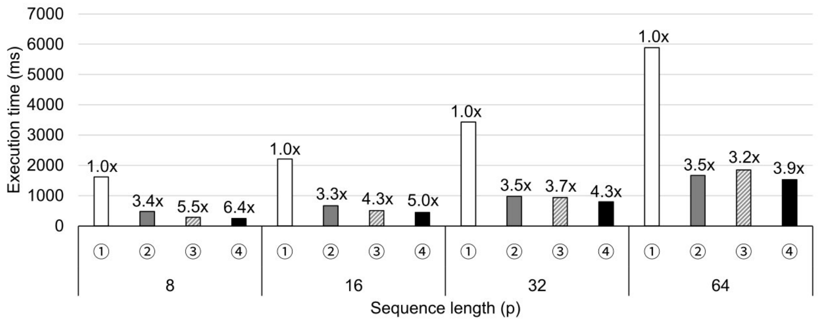
The graph traversal for finding those four edges took only 0.04s, 0.04s, and 0.07s in BERT, RoBERTa, and GPT-2, respectively. BERT and RoBERTa took a similar time due to their similar CG characteristics. We could need three profile runs for the node cost profiling and only one run for



(a)



(b)



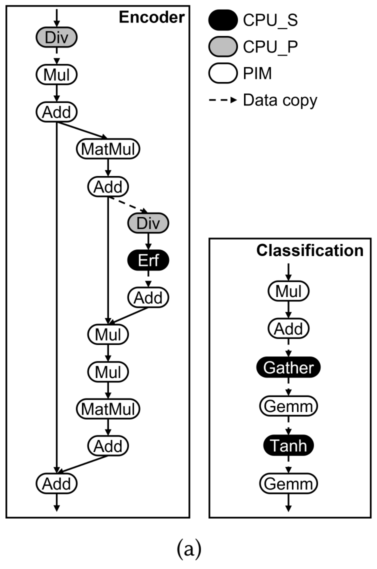
(c)

Fig. 9. Execution time and speedup running the transformer-based models when varying the sequence length ($p$) from 8 to 64 with respect to `CPU_S only`. (a) BERT (b) RoBERTa (c) GPT-2

the edge cost profiling. The time spent on optimal partitioning was negligible and proportional to the graph complexity.

## 5 RELATED WORKS

Previous research on workload partitioning for heterogeneous platforms has focused on the profiling and partitioning on the CPU-GPU systems with data-parallel kernels written in OpenCL [32] or CUDA [24]. Even with the higher throughput of GPU than CPU, the low PCIe bandwidth degrades



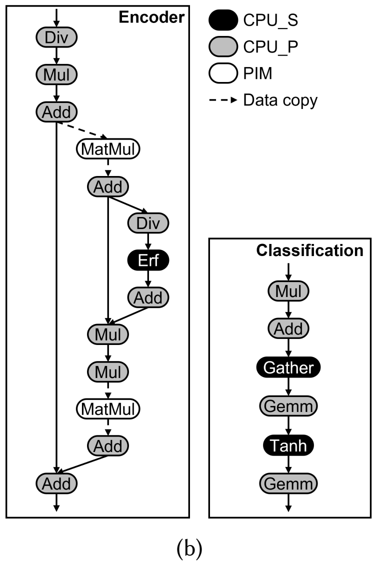
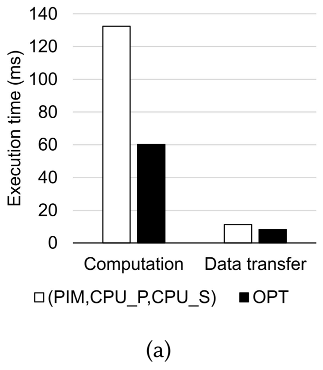
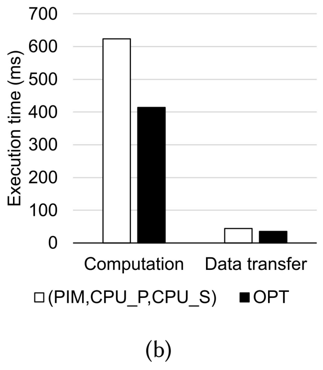


Fig. 10. The partitioned results of BERT and RoBERTa ($p = 16$). (a) (PIM,CPU_P,CPU_S) (b) OPT





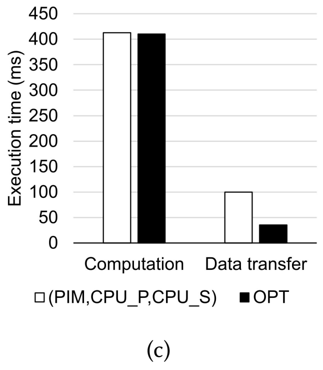


Fig. 11. Computation cost and data transfer cost in the models ($p$=16). (a) BERT (b) RoBERTa (c) GPT-2

Table 3. The number of nodes/edges according to the model and its profiling/partitioning overhead.

| | | Models | | |
|---|---|---|---|---|
| | | BERT | RoBERTa | GPT-2 |
| DCG characteristics | CG nodes/edges ($V_c/E_c$) | 222/262 | 616/725 | 1934/2424 |
| | DCG nodes/edges ($V_d/E_d$) | 469/686 | 1317/1951 | 2593/1723 |
| | DA edges | 20 | 20 | 24 |
| | PIM capable nodes | 108 | 308 | 245 |
| Algorithm overhead | Graph traversal | 0.04s | 0.04s | 0.07s |
| | Required profile runs (vertex/edge) | 3/1 | 3/1 | 3/1 |
| | Optimal partitioning | 0.02s | 0.07s | 0.21s |

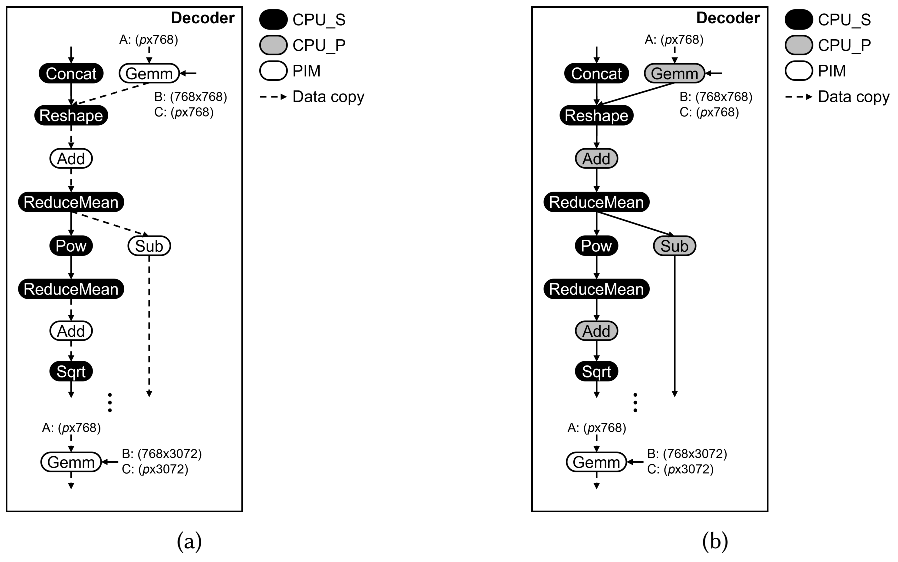


Fig. 12. The partitioned results of GPT-2 ($p = 16$). (a) `(PIM,CPU_P,CPU_S)` (b) `OPT`

the overall performance due to the data transfer overhead. Therefore, accurate profiling and partitioning determine the overall performance.

The first approach *statically* partitions the model by profiling applications and building a cost model using the profiled data. Luk et al. [22] developed an analytical performance model to predict the execution time on CPU and GPU through curve-fitting the profiled data. However, it did not distinguish the data transfer time from the computation time, leading to non-optimal partitioning. Albayrak et al. [2] proposed an inter-kernel greedy mapping algorithm from profiling kernels on devices. The algorithm iterates through $n$ kernels in topological order and assigns them to devices by comparing CPU and GPU costs, where costs are execution time on each device and the data transfer time from the source and sink devices. The algorithm would not provide optimal partitioning when resolving the complex data dependencies between kernels. Shen et al. [30] proposed an intra-kernel partitioning algorithm targeting a single kernel. When running each kernel with the CPU and GPU, the execution time of a kernel can be expressed as $\max(T_G + T_D, T_C)$, where $T_G$, $T_D$, and $T_C$ denote the GPU kernel computation time, the data transfer time, and the CPU kernel computation time, respectively. After partial profiling, they achieved the two metrics, a relative hardware throughput between CPU and GPU and the ratio of GPU throughput to data transfer bandwidth. Then, they substituted $T_G$, $T_D$, and $T_C$ with the two metrics and achieved the optimal partitioning when $T_G + T_D = T_C$.

Lee et al. [18] proposed partitioning multiple kernels of an application to multiple devices. It first builds a regression model based on profiled data collected by executing kernels with various input sizes for predicting the execution time of computing devices. Then, it constructs a data dependency graph of kernels. The kernels are listed in topologically sorted order of the graph and mapped with the device based on the priority, the predicted execution time, and data transfer cost. It further decomposes the kernel into sub-kernels and maps across multiple devices. Our work does not predict the execution time but profiles with very low cost, thus providing more accurate partitioning.

The second approach uses *runtime* information for optimal partitioning. Belviranli et al. [3] proposed a dynamically partitioning algorithm that works in two phases. The first phase learns the computational performance of each hardware by assigning a small amount of the workload. Then, the rest of the workload is assigned according to the hardware performance in the next adaptive stage. Boyer et al. [4] decomposed a kernel into chunks, such as primary computing workloads. They executed a small number of chunks on each device at the initial execution. Then, using the execution time, they partitioned the remaining work to balance the load between devices, leading to optimal partitioning.

The third approach uses the machine learning (ML) technique to predict the optimal partitioning for the given workload. Grewe et al. [9] proposed optimal partitioning based on an ML model trained with static information from a compiler's AST IR analysis. Similarly, Ghose et al. [8] additionally used control flow, mainly depending on the thread-id as an important deciding factor for optimal partitioning. Kofler et al. [17] proposed optimal partitioning by the ML model that was trained with static information from compile-time and dynamic information, e.g., data transfer size, from runtime.

## 6 CONCLUSION

In this paper, we proposed the optimal model partitioning method for achieving the best performance of the DL inference execution on the heterogeneous platform. First, we proposed the device-mapped computational graph (DCG) to represent the device capability by restructuring the existing computational graph. Then, we introduced the algorithm to find the minimum profile runs to obtain all node and edge costs in DCG in polynomial time. Also, we classified the DCG edges into the necessarily profiled distinct attribute edges, significantly reducing the problem size. Finally, our dynamic programming technique adopted from the ALS approach provided the optimal model partitioning from DCG.

We have evaluated our method on the PIM modeled FPGA platform with the ARM multicores by running three transformer-based models, varying a sequence length $p$ from 8 to 64. Our partitioning algorithm achieved a speedup of 1.1x~2.1x and 1.1x~3.0x compared to the execution with manually assigned device priority orders (CPU parallel, PIM, and CPU serial) and (PIM, CPU parallel, and CPU serial), respectively. Our low-cost edge profiling algorithm found only one profile run for obtaining all the edge costs, which took 0.04s, 0.04s, and 0.07s for each model with $p = 16$, which was negligible. Also, the partitioning algorithm took little time. We will apply our work to higher heterogeneity platforms, including GPUs and application-specific accelerators.

## ACKNOWLEDGMENTS

This paper was the result of the research project supported by SK Hynix Inc and the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean goverment (MSIP) (No.2022-0-00050, Development of PIM Computing Architecture based on Data-Flow).

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhen. 2016. Tensorflow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA, 265–283.

[2] Omer Erdil Albayrak, Ismail Akturk, and Ozcan Ozturk. 2012. Effective Kernel Mapping for OpenCL Applications in Heterogeneous Platforms. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW'12)*. Pittsburgh, PA, USA, 81–88.

[3] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. 2013. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 1–20.

[4] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. 2013. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Proceedings of the 10th ACM International Conference on Computing Frontiers (CF'13)*. Ischia, Italy, 1–10.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd. ed.). MIT Press.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (Nov. 2018).

[7] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. Minneapolis, MN, USA, 345–354.

[8] Anirban Ghose, Soumyajit Dey, Pabitra Mitra, and Mainak Chaudhuri. 2016. Divergence Aware Automated Partitioning of OpenCL Workloads. In *Proceedings of the 9th India Software Engineering Conference (ISEC'16)*. Goa, India, 131–135.

[9] Dominik Grewe and Michael F.P. O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction (CC'11)*. Saarbrücken, Germany, 286–305.

[10] Rajesh K. Gupta and Giovanni De Micheli. 1993. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers* 10, 3 (Sep. 1993), 29–41.

[11] Everton Hermann, Bruno Raffin, Francois Faure, Thierry Gautier, and Jeremie Allard. 2010. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Proceedings of the 16th European Conference on Parallel Processing (Euro-par'10)*. Berlin, Heidelberg, Germany, 235–246.

[12] Hitech Global. 2017. HTG-Z920. https://www.xilinx.com/products/boards-and-kits/1-qwrzuv.html. accessed: 2022-08-07.

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[14] Lakhmi Jain and Larry Medsker. 1999. *Recurrent Neural Networks: Design and Applications* (1st ed.). CRC Press, Inc.

[15] JEDEC Solid State Technology Association and others. 2012. JEDEC Standard: DDR4 SDRAM. *JESD79-4, Sep* (2012).

[16] Chang Hyun Kim, Won Jun Lee, Yoonah Paik, Kiyong Kwon, Seok Young Kim, Il Park, and Seon Wook Kim. 2022. Silent-PIM: Realizing the Processing-in-Memory Computing With Standard Memory Requests. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (Feb. 2022), 251–262.

[17] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS'13)*. New York, NY, USA, 149–160.

[18] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2015. Orchestrating multiple data-parallel kernels on multiple devices. In *Proceedings of the 24th International Conference on Parallel Architecture and Compilation Techniques (PACT'15)*. San Francisco, CA, USA, 256–366.

[19] Sukhan Lee, Shin-Haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Hyunsung Kim, O Seongil, Anand Lyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. Worldwide Event, 43–56.

[20] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (Jan. 2020), 94–110.

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* (July 2019).

[22] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. New York, NY, USA, 45–55.

[23] Microchip. 2008. Direct Memory Access (DMA) (Part III). http://ww1.microchip.com/downloads/en/devicedoc/70215c.pdf. accessed: 2022-01-02.

[24] NVIDIA Corporation. 2020. CUDA. https://developer.nvidia.com/cuda-toolkit. accessed: 2022-01-02.

[25] ONNX Developers. 2019. Open Neural Network Exchange (ONNX). https://onnx.ai/. accessed: 2021-12-22.

[26] ONNX Runtime Developers. 2021. ONNX Runtime. https://onnxruntime.ai/. accessed: 2021-12-22.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning

Library. In *Proceedings of the 32th Neural Information Processing Systems (NeurIPS'19)*. Vancouver, Canada, 8026–8037.

[28] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[29] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. 1993. Architecture of Field-Programmable Gate Arrays. *Proc. IEEE* 81, 7 (Jul. 1993), 1013–1029.

[30] Jie Shen, Ana Lucia Varbanescu, Peng Zou, and Henk Sips. 2016. Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (Dec. 2016), 2766–2780. https://doi.org/10.1109/TPDS.2015.2509972

[31] Fengguang Song, Stanimire Tomov, and Jack Dongarra. 2012. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and Multi-GPU Systems. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. San Servolo Island, Venice, Italy, 365–376.

[32] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[33] Chirs J. Thompson, Sahngyun Hahn, and Mark Oskin. 2002. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *Proceedings of the ACM/IEEE 35th Annual International Symposium on Computer Architecture (ISCA'02)*. Istanbul, Turkey, 306–317.

[34] Jing Wu and Joseph Jaja. 2013. High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. Cambridge, MA, USA, 115–125.