Efficient Scheduling Algorithm for Large-scale Models on Heterogeneous Mobile Systems

Jinyoung Kim
jinyoungkim97@yonsei.ac.kr
Yonsei University
Seoul, South Korea

Yongjun Kim
yjuun.kim@samsung.com
Samsung Electronics
Hwaseong, South Korea

Minseong Kim
min.s.kim@samsung.com
Samsung Electronics
Hwaseong, South Korea

Yongjun Park
yongjunpark@yonsei.ac.kr
Yonsei University
Seoul, South Korea

Abstract

Real-time AI inference on mobile devices increasingly demands the use of SoCs integrating heterogeneous accelerators, such as NPUs and DSPs. However, simultaneously optimizing operation parallelism and data transfer within constrained SRAM resources remains a significant challenge. Existing research often prioritizes parallelism at the expense of excessive DRAM access or suffers from high computational complexity, limiting scalability for large-scale models.

To address these issues, this paper proposes a novel dynamic programming-based scheduling framework optimized for heterogeneous mobile environments. The proposed framework explores near-optimal schedules by applying priority that reflects data transfer efficiency. Furthermore, we optimize the search space to ensure the algorithm operates within polynomial time and introduce a mechanism to prevent schedule deadlocks. Experimental results demonstrate that the proposed framework achieves an average performance improvement of 1.16× over list scheduling, proving its efficacy by enhancing data transfer efficiency and operation parallelism by 1.18× and 1.26×, respectively.

CCS Concepts

• **Software and its engineering** → *Compilers*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; System on a chip; • **Theory of computation** → **Scheduling algorithms**; **Dynamic programming**.

Keywords

scheduling, dynamic programming, heterogeneous SoC, NPU, compiler optimization, on-device inference

1 Introduction

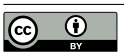
With the increasing demand for on-device AI [15, 44, 50, 54, 55], SoC-based hardware acceleration is being researched to run deep learning models efficiently within limited resources [4, 25, 43]. This SoC forms a heterogeneous computing environment by integrating various accelerators, such as GPUs, digital signal processors (DSPs) and NPUs including tensor engine (TE) and vector engine (VE), along with a standard CPU on a single chip. These specialized accelerators process key operations such as matrix multiplications and vector operations in parallel at the hardware level, shortening inference time and providing high power efficiency [18, 31, 49, 53, 57, 58].

A heterogeneous SoC shares DRAM as global memory and loads all necessary weights and input data into the DRAM at the start of inference. When a specific operation is executed, the required data is moved from the DRAM to the corresponding accelerator, and the finished result is stored back in the DRAM. The compiler controls this data transfer and execution to find the best schedule that minimizes the total inference time.

Since these data transfer and operation execution use different hardware resources, they can be performed at the same time [38, 41, 51]. Also, operations on different accelerators can run in parallel. For this, the compiler analyzes the model's operation graph to find tasks without data dependencies. If they require different hardware resources, the compiler schedules them to run simultaneously. Through this scheduling, the SoC minimizes idle resources and maximizes throughput to reduce inference time.

Additionally, the NPU has an internal SRAM shared by the TEs and VEs. The results from the TE or VE can stay in the SRAM instead of being sent to the DRAM, allowing them to be reused later. The SoC sets up this shared SRAM as a scratchpad memory, letting the compiler directly control data transfer to maximize data reuse. In mobile environments with limited resources, frequent DRAM accesses cause more delay and power consumption than the computation itself, so this technique greatly improves the inference time [20, 29, 45, 56].

[1. Complexity of scheduling with limited resources] However, simultaneously applying these optimization techniques requires substantial SRAM resources, which is often infeasible in resource-constrained mobile environments. Although maximizing parallel execution between the TE and VE requires loading large amounts of data into the SRAM, minimizing data transfer requires retaining data in the SRAM even when it is not immediately needed. This creates a situation similar to cache eviction in traditional CPUs caused by capacity limitations. Especially in environments where the compiler must explicitly manage data transfer, deriving an



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846863

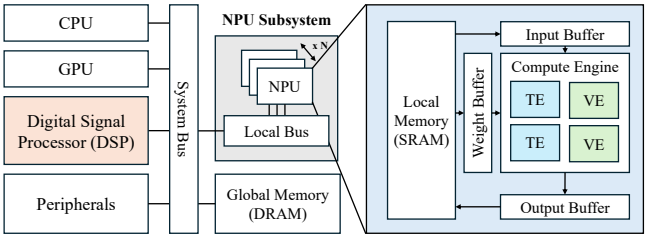


Figure 1: An overview of a mobile SoC architecture.

optimal schedule while considering these constraints is highly challenging [21].

Previous studies either fail to fully address this scheduling complexity or do not target heterogeneous SoC environments. List scheduling (LS) [2, 13], a representative algorithm in heterogeneous computing, first sorts operations without data dependency problems according to user-defined priorities and schedules those with higher priorities first. While LS has low computational complexity, its greedy approach often fails to appropriately balance operation parallelism with data transfer. Additionally, proposed integer linear programming (ILP) algorithms [35, 47] assume GPU cluster environments with independent resources, making them unsuitable for heterogeneous SoCs with shared resources.

[2. Computational complexity of the algorithm] Another issue is computational complexity. With the rising demand for large-scale models such as large language models [1, 14, 48] and large vision models [32, 42], scheduling algorithms with non-polynomial complexity often face exponential increases in compile time as the number of operations grows, making them impractical for real-world use. ILP algorithms exhibit exponential complexity as the number of constraints and variables increases with model size. Similarly, existing dynamic programming (DP) [27, 47] algorithms also fail to guarantee polynomial time complexity when extended to heterogeneous SoC environments, even with the application of proposed pruning techniques. Consequently, previous studies have limitations in applying large-scale models to mobile SoCs.

To overcome these limitations, this paper proposes a new Pruning-based Dynamic Programming scheduling framework optimized for mobile heterogeneous environments, named ***PDP***. The proposed framework first tunes scheduling priorities that reflect data transfer efficiency to derive a near-optimal schedule by balancing operation parallelism and data transfer. Additionally, a new pruning technique is introduced to ensure the algorithm operates within polynomial time, allowing it to be quickly applied to large-scale models. In particular, this paper presents a deadlock avoidance mechanism to prevent issues that may arise during the pruning process, guaranteeing schedule generation even for complex model structures.

Experimental results demonstrate that PDP significantly outperforms LS, achieving a maximum speedup of 1.63×. Specifically, by enhancing data transfer efficiency and operation parallelism by 1.18× and 1.26× respectively, the framework achieves an average performance improvement of 1.16× over LS. These results suggest that the proposed framework can effectively optimize the execution performance of large-scale models on heterogeneous mobile platforms.

The contributions of this paper are as follows:

- This paper proposes a new pruning-based DP scheduling framework that precisely reflects the SRAM sharing between the TE and VE within the NPU and the parallel execution in heterogeneous SoC environments. Through this, the framework derives near-optimal scheduling results in actual heterogeneous mobile SoC environments.
- This paper presents a technique to integrate scheduling priorities considering data transfer efficiency into the DP scheduling logic. This allows for a near-optimal balance between operation parallelism and data transfer, effectively shortening the overall inference time of the model.
- Furthermore, this paper presents a new pruning technique to solve the exponential complexity problem of existing DP algorithms. By limiting the algorithm's complexity to polynomial time, we designed it to enable efficient schedule generation even during the compilation of recent large-scale and complex models.
- Finally, this paper proposes a mechanism to systematically prevent deadlock problems that can occur when pruning. This guarantees that the framework can stably generate schedules across various models and demonstrates practical usability and applicability in actual compiler environments.

2 Background and Motivation

2.1 Heterogeneous Mobile SoC Architecture

A heterogeneous mobile SoC consists of various accelerators, including CPUs, GPUs, DSPs, and NPUs specialized for AI workloads [19, 39]. Figure 1 illustrates an architectural overview of such a heterogeneous mobile SoC. These accelerators exchange data through shared global DRAM. The NPU is composed of a TE, specialized for matrix operations, and a VE, which handles vector operations such as Softmax. The TE and VE share an internal SRAM within the NPU, allowing them to exchange data directly without passing through the DRAM. This shared SRAM is configured as scratchpad memory (SPM), designed to allow the compiler to explicitly manage data transfer between the DRAM and SRAM [5, 6, 11]. While data transfer between the DRAM and SRAM is explicitly managed by the compiler through load/store operations, data transfer in the lower-level buffers adjacent to the NPU's compute engines is automatically controlled by hardware logic.

At the beginning of model inference, all weights and input data are loaded into the DRAM. When an operation is performed on a specific accelerator, the necessary data is loaded from the DRAM to that accelerator, and the resulting output is stored back in the DRAM after execution. Thus, the execution of an individual operation involves three stages: 1) data loading from the DRAM to the accelerator, 2) operation execution on the accelerator, and 3) output data storing from the accelerator to the DRAM. This process is controlled by the compiler. In the case of the NPU, the first and third data transfer stages can be bypassed by maintaining data in the SRAM for reuse instead of evicting it to the DRAM.

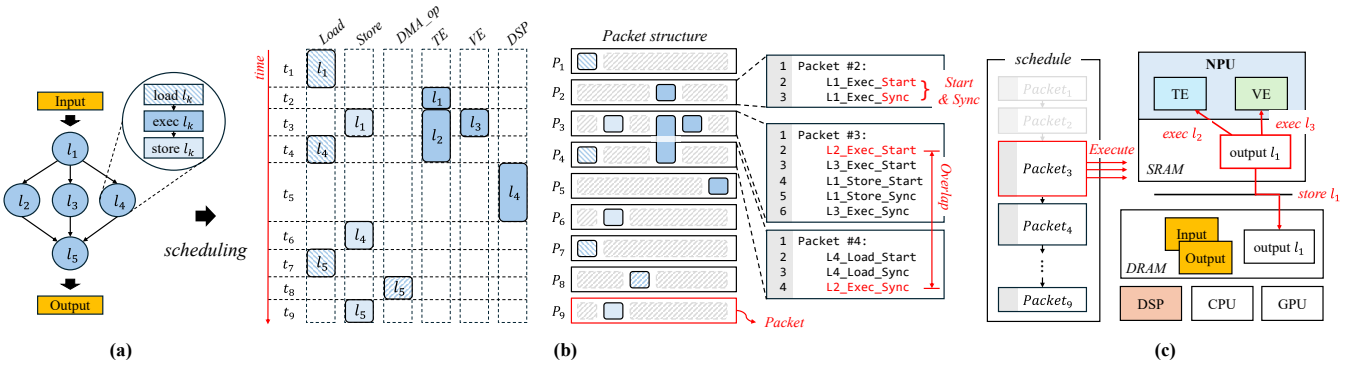


Figure 2: (a) Deep learning model graph, (b) compiler representation of the scheduling result, and (c) execution of the schedule.

2.2 Execution Model for Heterogeneous SoC Architectures

Figure 2 illustrates the process of how a deep learning model schedule is represented in the compiler. Figure 2(a) shows a model graph, where l_k denotes the k-th layer of the model. As mentioned previously, the execution of a layer consists of three stages: load, exec, and store. Load fetches the input feature map (IFM) and weights into the accelerator, while store saves the resulting output feature map (OFM) to the DRAM. Exec includes operation execution on the TE, VE, or DSP, as well as DMA operations such as layout conversion performed on the SRAM.

Figure 2(b) shows a scheduling result. After l_1 is executed on the TE, l_2 and l_3 proceed on the TE and VE, respectively, allowing them to run immediately without additional data transfer. In contrast, l_4 is executed on the DSP and requires data sharing through the DRAM. Consequently, the result of l_1 is stored and then loaded into the DSP. Subsequently, the output is stored and loaded again for l_5 to execute on the NPU's SRAM.

The compiler represents the schedule by indicating only the start and end timings of each operation. The right side of Figure 2(b) shows the scheduling result as represented by the compiler. The compiler performs each operation using start instructions to initiate hardware actions and sync instructions to wait for completion. These are grouped into time intervals, and instructions within the same group are executed in parallel by the hardware. Such a group is called a *packet*.

In Figure 2(b), the time intervals t_1 to t_9 correspond to the packets P_1 to P_9. While t_1 is a single interval containing both the start and sync instructions for l_1, it could also be divided into two intervals, each containing one instruction. In this figure, as many instructions as possible are included in the same time interval to minimize the number of packets. P_3 and P_4 show how the parallel execution of operations is represented. The start instruction of l_2 is placed in P_3 and the sync instruction in P_4, demonstrating that it not only runs concurrently with multiple operations but also continues to run in parallel with another operation after others have finished.

Consequently, the compiler represents the entire schedule by grouping start and sync instructions into packets and sequencing them. Figure 2(c) shows the execution of the schedule on the hardware. The hardware executes packets sequentially from the top, processing instructions within the same packet simultaneously. Execution proceeds to the next packet only after all instructions in the current packet have finished. This example shows the execution of packet P_3, where l_2 and l_3 bypass the data loading and execute immediately because the required data is already present in the SRAM.

2.3 Data Transfer Optimization

By using the shared SRAM as SPM within NPUs, the compiler can minimize data transfer to the DRAM by omitting unnecessary load/store operations. This technique is called *feature map forwarding*. Such optimization can be implemented between TE and VE, explicitly retaining data in the SRAM for future use, or adjusting the tiling to maximize these opportunities. Various techniques have been studied, and stratum chain [20, 26] is a representative method that utilizes tiling.

When two layers are executed sequentially, each layer is typically tiled considering the small SRAM size. The tiles in the second layer begin only after all tiles in the first layer are completed. This is because a single OFM generated by the first layer's tile is insufficient to be used as the IFM for the second layer's tile and thus must gather multiple tile results. This forces synchronization between layers, which incurs delays in inference time. Furthermore, since the total OFM size of tiles frequently exceeds the SRAM capacity, the results must be stored in the DRAM and loaded again, causing performance degradation.

To eliminate such unnecessary DRAM accesses, tiling should be carefully applied so that the data generated in a preceding layer tile can be immediately used for the subsequent layer tile. This removes synchronization and redundant DRAM accesses, while enabling direct data sharing between inter-layer tiles on the SRAM. The structure that links tiles across layers without such synchronization is called *stratum chain*. Due to this chain structure, even models with a sequential structure gain various opportunities for parallel execution. By concurrently executing operations across different chains, stratum chain not only minimizes data transfer but also maximizes operation parallelism, effectively reducing inference time.

2.4 Complexity of Balancing Parallelism and Data Transfer Optimization

Many opportunities exist to optimize both operation parallelism and data transfer within deep learning models. Although applying both optimizations is ideal for improving inference performance, conflicts often arise in mobile environments with limited SRAM capacity [7, 40]. Maximizing operation parallelism requires sufficient compute units and SRAM space to accommodate the input data of concurrent operations. In contrast, minimizing data transfer requires retaining data in the SRAM for future use even when it is not immediately needed, which also demands additional memory space. Since both optimizations compete for substantial SRAM resources, situations similar to cache eviction due to capacity misses frequently occur in mobile environments. Especially in an environment with the SRAM used as SPM where the compiler explicitly controls data transfer, deriving an optimal schedule while balancing these competing optimizations is highly challenging [21].

Figure 3 compares the performance of the LS algorithm [2, 13] with an Expert schedule manually optimized by an expert. The results are normalized to the performance of the Expert, where DMA Eff. represents the degree of DRAM access efficiency and Compute Over. denotes the level of compute overlap. As a widely adopted algorithm for heterogeneous systems, LS schedules operations without data dependency problems according to user-defined priorities. Although LS has low computational complexity and ease of implementation, its greedy approach to scheduling operations as soon as resources become available concentrates execution at the beginning of the schedule, which increases SRAM pressure. This often prevents LS from finding an optimal solution in scenarios where the balance between parallelism and data transfer must be considered.

As shown in Figure 3, LS fails to balance these two factors for well-known models. While LS initially appears to enhance parallelism, the excessive concentration of operations leads to data saturation in the SRAM. When execution is bound by limited functional units, the SRAM becomes saturated, eventually degrading parallelism. A representative case is when excessive load operations are executed initially, filling the SRAM with data waiting for the TE, which creates performance bottlenecks. In extreme cases, the algorithm may reach a schedule deadlock where no further operations can be scheduled because, although input data is ready, there is insufficient space for partial sums. LS resolves such deadlocks by prematurely evicting data from the SRAM to the DRAM, thereby losing opportunities for data transfer optimization.

Due to these shortcomings, LS shows lower performance than the expert schedule, particularly in EDSR [30]. EDSR consists of sequential residual blocks, yet the application of stratum chain creates significant opportunities for parallelism. Because performance is heavily dependent on the success of feature map forwarding for skip connections within the residual blocks, LS attempts to execute up to four chains concurrently, but fails to implement feature map forwarding due to SRAM shortages. In contrast, the expert schedule limits concurrency to a maximum of two chains to secure SRAM space, successfully achieving feature map forwarding for the skip connections. Consequently, the expert schedule achieves 2.43× higher inference performance for EDSR and an average of

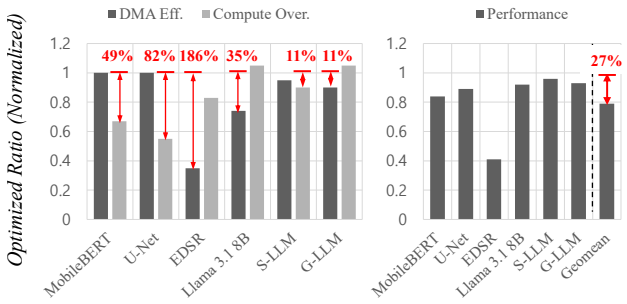


Figure 3: Performance comparison of LS and Expert schedules. S-LLM [8] and G-LLM [48] denote proprietary on-device LLMs from Samsung and Google, respectively.

1.27× higher performance across all models compared to LS. As demonstrated, balancing operation parallelism and data transfer optimization is the key factor that determines improved inference performance.

2.5 Computational Complexity of the Algorithm

Beyond the CNN-based super-resolution models primarily used in mobile environments, there is an increasing demand for generative AI, such as large language models and large vision models. Deploying these large-scale models on resource-constrained mobile devices requires device-specific optimizations, which require significant manual effort from experts. To enhance optimization efficiency, an environment that allows experts to rapidly iterate through various trials is essential. Therefore, maintaining short compile times to quickly verify optimization results is a critical factor in accelerating model deployment. Since scheduling is a key part of such expert-led optimizations, ensuring that scheduling algorithms complete within polynomial time is essential for practicality as model sizes continue to grow [28].

Existing ILP-based algorithms [16, 35, 47] model the scheduling problem using constraints and objective functions to derive optimal solutions. These algorithms define layer execution timing, hardware resource usage, and data dependencies as variables for a solver. However, ILP is NP-complete and fails to derive solutions within polynomial time. Consequently, compile time increases exponentially as model size grows, making it impractical for real-world applications.

Alternatively, DP-based algorithms [27, 47] derive optimal schedules by combining solutions to subproblems. However, these proposed DP algorithms either fail to complete within polynomial time or utilize hardware models that are incompatible with the characteristics of heterogeneous mobile SoCs. In the latter case, the algorithms assume a hardware model where all concurrently initiated operations must complete before subsequent operations can begin. Consequently, they fail to generate optimal schedules, such as l_2 in packets P_3 and P_4 of Figure 2(b), where multiple operations are executed in parallel, thereby falling short of achieving optimal performance. Therefore, to ensure the practicality of the algorithm, it must guarantee completion within polynomial time.

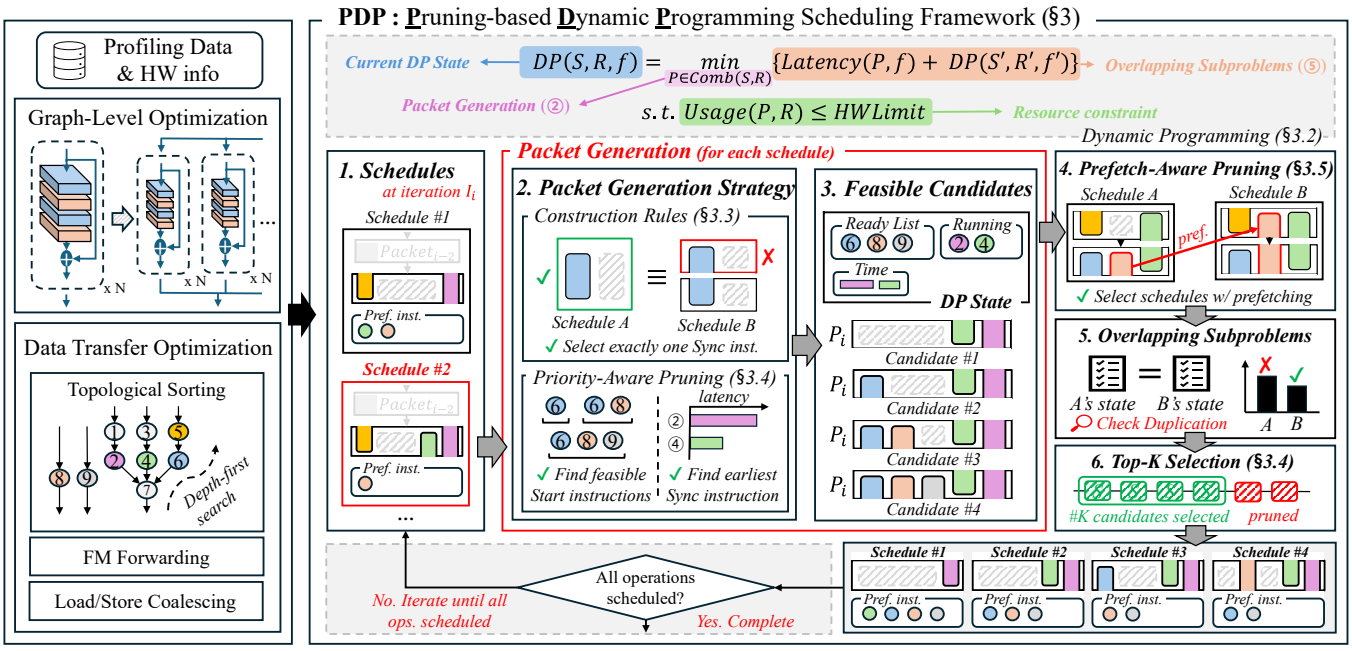


Figure 4: An overview of Pruning-based Dynamic Programming scheduling framework.

This requirement is becoming increasingly crucial given the current trend of growing model sizes.

3 Pruning-based Dynamic Programming Scheduling Framework

This section presents the PDP scheduling framework, which balances operation parallelism and data transfer optimization to derive near-optimal schedules within polynomial time. Figure 4 illustrates an overview of the proposed PDP scheduling framework. First, we define the essential terms and definitions that form the foundation of the algorithm. We then describe the process of identifying available scheduling candidates from the current state and determining the optimal schedule. Furthermore, we define search rules that restrict candidate exploration to avoid redundant searches and provide a detailed discussion of the pruning techniques used to ensure polynomial time complexity. Finally, we present a mechanism to prevent schedule deadlocks that may arise from the application of these pruning techniques.

3.1 Terms and Definitions

Definition 1. v_i denotes the i-th node in a directed acyclic graph (DAG) representing the deep learning model, where each node corresponds to an individual operation within the model.

Definition 2. t_i denotes the execution time of node v_i within the model, which is obtained by profiling on the target device or calculated using a cost model.

Definition 3. P_i denotes the i-th packet within the schedule generated by the compiler. Packets are executed sequentially on the hardware starting from P_1, while all instructions within the same packet are executed concurrently.

Definition 4. $Resource(v_i)$ denotes the amount of hardware resources required to execute node v_i, and $HWLimit$ denotes the total hardware capacity of the target device. These resources include the number of TE or VE units and the size of SRAM.

Definition 5. $starts(P_i)$ and $syncs(P_i)$ denote the sets of operations triggered by the start instructions and completed by the sync instructions contained in packet P_i, respectively.

Definition 6. S denotes the set of operations that have not yet been executed, R represents the set of operations currently running, and f is a function defining the remaining execution time for each operation. These variables are used as key components to represent the scheduling state in the DP algorithm.

Definition 7. $Usage(P, R)$ denotes the total usage of hardware resources when executing packet P in the state defined by R. This value is calculated by adding the resource requirements of the operations newly started in P and those currently running in R, and is defined as:

$$Usage(P, R) = \sum_{v \in starts(P)} Resource(v) + \sum_{v \in R} Resource(v) \quad (1)$$

Definition 8. $Latency(P, f)$ denotes the latency of packet P in the state defined by f. Since the start instructions in the packet just trigger the beginning of operations, the packet's latency is determined by the time required for operations to be completed via sync instructions. As all operations within a packet are executed in parallel, the latency of P is determined by the maximum remaining execution time among these operations, and is defined as:

$$Latency(P, f) = \max_{v \in syncs(P)} \{f(v)\} \quad (2)$$

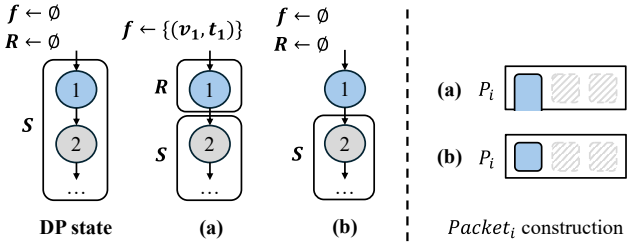


Figure 5: DP states in serial execution.

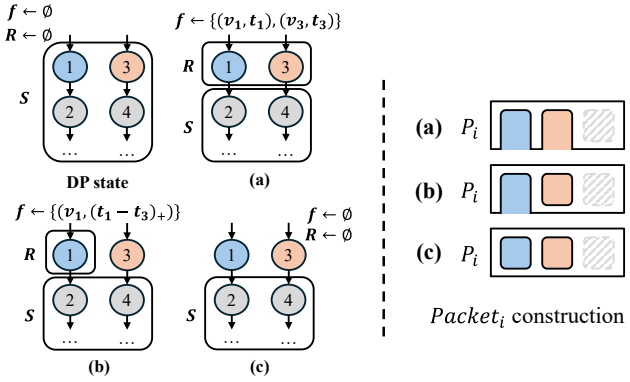


Figure 6: DP states in parallel execution.

Definition 9. $Comb(S, R)$ denotes the set of all possible packet combinations that can be generated by combining executable instructions in the state defined by S and R.

3.2 Scheduling Problem Formulation

The scheduling algorithm proposed in this paper targets static scheduling performed at compile time, assuming that graph-level optimizations such as tiling have already been applied to the deep learning model. We also assume that the required hardware resources and execution units for each operation within the model are pre-determined. The algorithm takes as input a DAG representing the dependencies between operations and their respective execution times. In this DAG, nodes denote individual operations and directed edges represent data dependencies. Finally, the algorithm produces a near-optimal scheduling result as output.

The DP state is represented by a combination of the set of operations that have not yet been executed (S), the set of operations currently running (R), and a function f that defines the remaining execution time for each operation. Each entry in the DP table, $DP(S, R, f)$, denotes the total execution time of the optimal scheduling that can be achieved from that state. The algorithm explores possible packet combinations that can be scheduled in the current state and searches for the optimal solution by comparing the execution times of the different schedules resulting from each choice.

$$\begin{aligned} &DP(S, R, f) = \min_{P \in Comb(S,R)} \{Latency(P, f) + DP(S', R', f')\} \\ &\text{s.t. } Usage(P, R) \leq HWLimit \end{aligned} \tag{3}$$

The main equation above illustrates the update process for the DP entries. First, the algorithm explores executable packet candidates based on the current state ($P \in Comb(S, R)$), where each packet must satisfy the hardware resource limits ($Usage(P, R) \leq HWLimit$). For all possible packet combinations, the algorithm calculates the sum of the packet's latency ($Latency(P, f)$) and the optimal execution time of the subsequent state ($DP(S', R', f')$). The packet that minimizes this sum is selected to update $DP(S, R, f)$, and this selection is recorded as the optimal scheduling decision.

This update process optimizes redundant computations by leveraging the overlapping subproblems in DP.

$$\begin{aligned} S' &= S \setminus starts(P) \\ R' &= (R \cup starts(P)) \setminus syncs(P) \\ f'(v) &= \max(0, \tau(v) - Latency(P, f)) \\ \text{where } \tau(v) &= \begin{cases} t_v & \text{if } v \in starts(P) \setminus syncs(P) \\ f(v) & \text{if } v \in R \setminus syncs(P) \end{cases} \end{aligned} \tag{4}$$

The equation above defines the state transition process following a packet selection. The subsequent state S' is the set obtained by excluding the newly started operations from the existing set S. R' is defined as the union of the previously running operations and the newly started ones, excluding those that are completed within the selected packet. The remaining time function f' is updated as follows: for newly started operations that have not yet completed ($v \in starts(P) \setminus syncs(P)$), the value is recorded by subtracting the packet's latency from the operation's total execution time. For operations previously running but not yet completed ($v \in R \setminus syncs(P)$), the function is updated by subtracting the packet's latency from their existing remaining execution time.

The following describes the packet exploration process of the algorithm in detail. Figure 5 illustrates an example of the DP process on a serial graph, with the resulting packet shown on the right. Assuming that operation v_1 is ready for scheduling and no operations are currently running, two possible packet combinations exist. In Figure 5(a), only the start instruction for v_1 is selected. In this case, v_1 transitions to the running state, being removed from S and added to R, and its execution time t_1 is recorded in f ($f \leftarrow \{(v_1, t_1)\}$). Figure 5(b) represents the case where both the start and sync instructions for v_1 are selected. Here, since v_1 is started and completed within the same packet, it is removed from S while R remains an empty set. As the execution of v_1 is complete, no remaining time is recorded in f.

Figure 6 illustrates an example where two operations are executed in parallel. In a state where v_1 and v_3 are ready and no operations are currently running, there are four possible packet combinations for their parallel execution. In Figure 6(a), only the start instructions for both operations are selected, and thus both v_1 and v_3 transition to the running state. Figure 6(b) shows the case

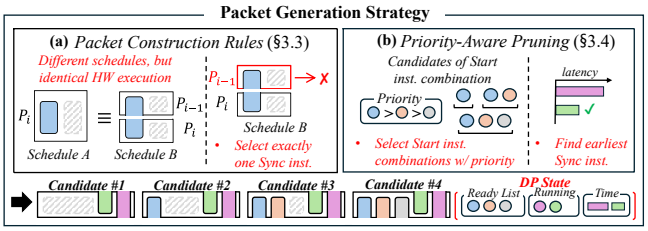


Figure 7: An overview of packet generation strategy.

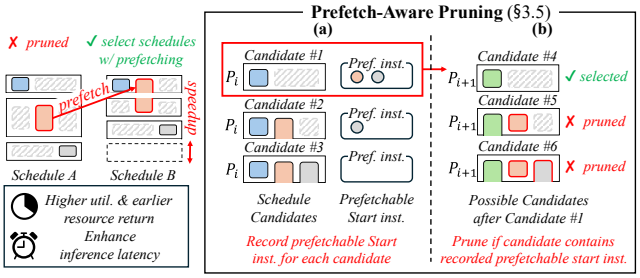


Figure 8: Mechanism of prefetch-aware pruning.

where start instructions for both are selected, but only the sync instruction for v_3 is included. v_1, for which no sync instruction was selected, remains in the running state after the packet execution, and its remaining execution time is updated in f. Since v_3 completes its execution within this packet, the packet's latency is determined by t_3, during which v_1 is also executed in parallel. Consequently, the value $(t_1 - t_3)_+$ is recorded in f ($f \leftarrow \{(v_1, (t_1 - t_3)_+)\}$). Here, $(t_1 - t_3)_+$ denotes $max(0, t_1 - t_3)$ to ensure that the remaining execution time does not become negative. Similar to Figure 6(b), it is also possible to select only the sync instruction of v_1. Finally, Figure 6(c) represents the case where sync instructions for both operations are selected, completing their execution simultaneously.

3.3 Packet Construction Rules

However, the previous algorithm can cause redundant exploration due to the structural issues of packet-based scheduling. Since start instructions do not consume execution time, a packet consisting only of start instructions can be merged with subsequent packets without changing the hardware execution result. During hardware execution, operations running in parallel at a specific time can be represented by multiple packets, and these packets can be merged with each other to form alternative schedules with the same result. To avoid redundancy, the resulting packet sequence must not be convertible into another form through merging. For this, the algorithm applies a rule to construct a packet by combining zero or more start instructions and exactly one sync instruction. Figure 7(a) shows an example of applying this rule.

The conditions for merging two consecutive packets are as follows: 1) the preceding packet consists only of start instructions, or 2) the following packet consists only of sync instructions. Assume that the construction rules of this algorithm allow for redundancy. Every packet generated according to the rules contains exactly one sync instruction. For two consecutive packets to merge, the following packet must consist only of sync instructions according to condition 2). However, the merged packet would then contain a total of two sync instructions, which violates the rule of having exactly one sync instruction per packet. Therefore, such merged packets cannot be created during the search process, which shows that the rule effectively prevents redundant exploration.

These packet construction rules also improve the spatial complexity of the algorithm. Since every packet contains exactly one sync instruction, schedules with the same number of packets have an identical count of completed operations. Consequently, all schedules reaching the same DP state always have the same number of selected packets. Based on this property, the algorithm can select one packet at each iteration and maintain only the results of the current iteration in the DP table, effectively reducing spatial complexity.

3.4 Complexity Reduction via Pruning

However, the proposed algorithm still has a complexity limit that makes it difficult to operate in polynomial time. To solve this, this paper introduces pruning techniques, designed to efficiently derive schedules even for large-scale models.

3.4.1 Priority-Aware Pruning. First, a pruning technique based on user-defined priorities is applied to efficiently reduce the search space. Figure 7(b) shows the process of applying this technique. When constructing a packet, pending operations are sorted by priority, and a lower-priority operation is restricted to be included in a combination only if higher-priority operations are already selected. For example, when the priorities are $v_1 > v_2 > v_3$, the possible combinations are limited to $\{\}, \{v_1\}, \{v_1, v_2\}$ and $\{v_1, v_2, v_3\}$ to reduce the search space. These priorities are set to prioritize operations with high data transfer efficiency and those on the critical path.

For data transfer efficiency, PDP topologically sorts the model to establish an execution order that maximizes data reuse, and then applies data transfer optimization on this order. Operations with a lower topological index receive a higher priority. For the critical path, we use the slack time of each operation, defined as the difference between its latest start time (LST) and earliest start time (EST). Operations with smaller slack lie closer to the critical path and are therefore assigned higher priorities. Additionally, the one sync instruction included per packet targets that of the operation with the shortest remaining execution time among those currently running. If an operation that completes earlier is ignored and another is selected instead, the release of hardware resources is delayed, resulting in an inefficient schedule. Therefore, the algorithm prioritizes the operation with the minimum remaining time to maximize resource efficiency and exclude unnecessary search space.

3.4.2 Top-K Selection. The algorithm constructs schedule candidates based on the selected packets and finally performs top-K selection to retain only the top K candidates. In this paper, schedules with relatively lower parallelism are prioritized to limit operation parallelism. This criterion is intended to prevent deadlocks that may occur during the scheduling process, and the specific reasons for this are detailed in the following section.

Algorithm 1: Parallel Execution Scheduling

Input: Directed acyclic graph of input model G;
Heuristic value for top-k selection K
Output: Optimal schedule S_{opt}

```
Function runScheduling(G, K):
    C ← ∅
    cand_init ← initialize(G)
    C.insert(cand_init)
    while not isAllScheduled(C, G) do
        C_next ← ∅
        for cand in C do
            tryToSchedule(cand, C_next)
        pruneByPrefetch(C_next)
        mergeOverlapStates(C_next)
        selectTopK(C_next, K)
        C ← C_next
    S_opt ← getBestSchedule(C)
    return S_opt
```

3.4.3 Complexity. Through these pruning techniques, the algorithm derives a schedule in polynomial time. To complete all operations, the sync instruction for each operation must be scheduled. Since the proposed packet construction rule selects exactly one sync instruction at each iteration, the final schedule consists of a number of packets equal to the total number of operations, requiring n iterations in total. At each iteration, the algorithm maintains k candidates, where k is the top-K selection parameter, and each candidate generates $l + 1$ new candidates. Here, l denotes the average maximum number of operations that can be executed in parallel. Therefore, the time complexity is defined as $O(nlk)$, and it becomes $O(n^2lk)$ in the worst case when the cost of maintaining the ready queue, including sorting and dependency updates, is taken into account. Spatial complexity is also managed efficiently. As previously mentioned, schedules reaching the same DP state always contain the same number of packets. By maintaining only the schedules with the same number of packets in the DP table, spatial complexity can be effectively reduced, which is further improved by pruning techniques. Consequently, the spatial complexity is defined as $O(lk)$.

3.5 Deadlock Prevention Mechanism

The DP-based algorithm theoretically guarantees a valid schedule by exploring the entire search space. However, applying the proposed pruning may lead to deadlocks if the reduced space excludes all valid solutions. Therefore, this paper proposes a mechanism to effectively prevent deadlocks even within a reduced search space. Generally, hardware resources are released upon operation completion. However, when data is kept in SRAM for data transfer optimization, the SRAM remains occupied until the operation that finally consumes the data is executed. This can result in a deadlock where the scheduler waits for a specific operation without releasing resources, even when no operations are currently running. While the priority-based packet generation does not induce deadlocks, top-K selection may increase the risk if it restricts candidates solely to minimize inference time. Since higher parallelism increases this risk, the proposed mechanism prioritizes schedules that minimize parallelism to prevent deadlocks. However, this approach may require an excessively large K to find the optimal schedule. To address this, we further propose a technique that identifies prefetchable candidates and excludes unnecessary ones.

Algorithm 2: Apply Prefetch-Aware Pruning

Input: Schedule candidates C_{next}
Output:

```
Function pruneByPrefetch(C_next):
    for cand in C_next do
        if any cand.starts in cand.P_ops and
          isPrunable(cand) then
            C_next.remove(cand)
    for cand in C_next do
        cand_most ← getMostParallel(cand)
        cand.P_ops ← getDifference(cand_most.starts,
          cand.starts)
```

3.5.1 Prefetch-Aware Pruning. Prefetching executes operations earlier when hardware resources are available, advancing the timing of resource release. This secures available resources for subsequent operations and ultimately reduces inference time. Since schedules before and after applying prefetching maintain the same validity, preserving only the prefetch-applied schedules effectively maintains near-optimal candidates without the risk of deadlocks.

Figure 8 illustrates an example of the proposed prefetch-aware pruning technique. At each iteration, the algorithm selects packet candidates and records prefetchable operations. Figure 8(a) shows the available packet combinations when three executable operations exist at iteration i. In this example, it is assumed that the first packet combination (Candidate #1), consisting only of the start and sync instructions of the blue operation, is selected. Given that Candidate #3 executes up to three operations simultaneously, it is evident that all three candidates have sufficient hardware resources to execute these operations. Therefore, in the case of Candidate #1, where the orange and gray operations are not executed despite sufficient resources, these operations are recorded as prefetchable. Similarly, the unexecuted gray operation in Candidate #2 is recorded as prefetchable. Figure 8(b) shows the process of continuing the scheduling from Candidate #1 at iteration $i + 1$. Assuming that green, orange, and gray operations are executable at this iteration, the orange and gray ones have already been recorded as prefetchable in the previous iteration. Consequently, the algorithm excludes packet combinations containing these operations from the search. By leveraging prefetchable operation information in this manner, inefficient schedules where prefetching is not applied can be effectively excluded.

4 Implementation

Algorithm 1 shows the pseudocode of the proposed PDP scheduling algorithm. The algorithm receives a DAG G and a parameter

Algorithm 3: Overlap Subproblem Schedules

```
   Input: Schedule candidates C_next
   Output:
1  Function mergeOverlapStates(C_next):
2      for cand in C_next do
3          for E_ops in cand.E_list do
4              if all cand.starts not in E_ops then
5                  C_next.remove(cand)
6      state_to_cands ← groupByDPState(C_next)
7      for state, C in state_to_cands do
8          cand_opt ← findOptimal(C)
9          C_sorted ← sortBySchedId(C)
10         E_list ← ∅
11         for cand in C_sorted do
12             cand.schedule ← cand_opt.schedule
13             cand.E_list ← E_list
14             E_list.insert({cand.P_ops})
```

K for top-K selection as input and outputs the optimal schedule S_{opt}. First, an initial schedule is generated based on G and inserted into the candidate set C (lines 2–4). Then, iteration continues until scheduling is completed (line 5). At each iteration, packet combinations are selected by combining executable operations following the packet generation strategy, and new candidates are generated to be stored in the set for the next iteration (C_{next}) (lines 6–8). Generated candidates that have lower parallel execution are assigned lower schedule IDs. Subsequently, prefetch-aware pruning, overlapping subproblems, and top-K selection are applied (lines 9–12). When scheduling is complete, the algorithm finally returns one near-optimal schedule as a result (lines 13–14).

Algorithm 2 shows the pseudocode for the prefetch-aware pruning technique. This algorithm removes candidates that can be prefetched from C_{next} and updates prefetchable operation set (P_{ops}) for the next iteration. First, the algorithm checks the P_{ops} of each candidate. If a candidate's packet at the current iteration includes the start instructions of those operations, the candidate is removed (lines 2–4). The algorithm uses the isPrunable function to make an exception for the candidate with the most parallel execution, as its prefetched schedules might have already been pruned. Then, among the child candidates derived from the same parent, the algorithm calculates the set difference between the start instructions of the candidate with the highest parallelism ($cand_{most}$) and those of the current candidate. This set difference is recorded as the P_{ops} for the next iteration (lines 5–7).

Algorithm 3 shows the pseudocode for the overlapping subproblems technique. Due to the previously described prefetch-aware pruning, candidates in the same DP state can possess different P_{ops}. This requires a modification to the conventional approach, which simply removed non-optimal candidates. The algorithm copies the optimal schedule of a specific DP state to other candidates in the same state and defines an essential operation list (E_{list}) to prevent search redundancy. E_{list} consists of the P_{ops} of candidates with lower schedule IDs within the same DP state. Candidates with higher IDs must include at least one operation from each P_{ops} of the candidates with lower IDs. Because each candidate's search space within the same DP state differs according to its own P_{ops}, this mutually exclusive approach avoids redundant searches.

First, the algorithm removes candidates from the candidate set C_{next} that do not satisfy the condition mentioned above (lines 2–5). Subsequently, candidates are grouped by their DP states (line 6). Within each group, the algorithm identifies the optimal schedule, which has the shortest execution time (findOptimal) (lines 7–8). Then, it sorts the candidates in ascending order of their IDs (sortBySchedId) (line 9). Following this order, the schedule of each candidate is replaced by the optimal schedule, and its E_{list} is updated with the P_{ops} sets accumulated from all preceding candidates. Finally, the candidate's own P_{ops} is added to the E_{list} for the next candidate (lines 10–14).

5 Evaluation

Experiments were evaluated on the Samsung Exynos 2600 [9], a heterogeneous mobile platform. The SoC features NPU and DSP accelerators, where the NPU comprises a structure in which the TE and VE share a common SRAM. Six models, as listed in Table 1, were employed for the evaluation. To assess the performance of PDP, we compared it against list scheduling (LS), utilized in the O2 optimization of production-level compilers, and an expert schedule manually optimized based on profiling data (Expert). The expert schedule can be regarded as near-theoretical optimal, as it additionally accounts for memory fragmentation and hardware-specific issues. All results represent average values obtained from 10 repeated runs on the target device. In PDP, priority was applied by considering the model topology to maximize data transfer efficiency. Furthermore, we varied the parameter K from 1 to 64 and selected the schedule that achieved the highest inference performance. We analyze overall inference performance for each model, data transfer optimization and operation parallelism, the effect of parameter K on the output schedule, and the effect of DP state memoization.

Table 1: Summary of input models for evaluation.

Name	Application
MobileBERT [46]	NLP
U-Net [42]	Image Segmentation
EDSR [30]	Image Super-Resolution
Llama 3.1 8B [14]	On-device LLM
Samsung proprietary LLM (S-LLM) [8]	Generative AI (Language)
Google proprietary LLM (G-LLM) [48]	On-device LLM

5.1 Overall Inference Performance

Figure 9(a) presents a performance comparison between LS and the proposed PDP framework. Overall, PDP outperforms LS, achieving a maximum speedup of 1.63× in EDSR. However, PDP shows slightly lower performance than the Expert, which is attributed to hardware execution characteristics that are difficult to identify at the compiler level or constraints not modeled during the scheduling phase. For instance, in MobileBERT, despite deriving a schedule comparable to the Expert, stalls occurred due to false dependency

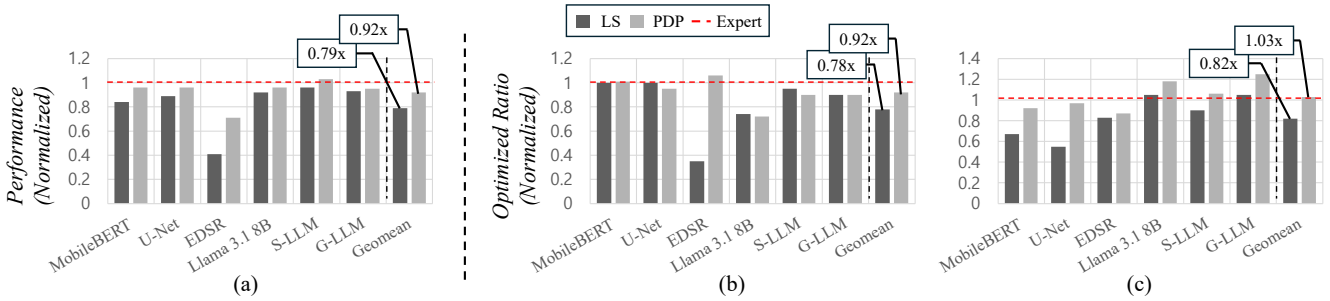


Figure 9: Experimental results normalized to Expert: (a) inference performance, (b) DMA efficiency, and (c) compute overlap.

in SRAM which is subtle to detect and mitigate during actual hardware execution. In the case of EDSR, the schedule was modified during the subsequent memory offset allocation process due to fragmentation, preventing it from reaching the expected performance. In an isolated analysis of segments without fragmentation, PDP was confirmed to achieve approximately 0.90× the performance of the Expert. Despite these compiler modeling limitations, PDP recorded an average performance of 0.92× compared to the Expert and 1.16× compared to LS across all models.

5.2 Breakdown Analysis of Data Transfer and Operation Parallelism

Figures 9(b) and (c) present the analysis results of data transfer efficiency and operation parallelism for each model, respectively. Regarding data transfer efficiency, PDP achieved the most outstanding performance in EDSR, recording 1.06× higher efficiency compared to the Expert. Figure 10 shows the profiling results of LS, PDP, and the Expert for EDSR. While LS attempts to maximize computational parallelism by executing multiple branches simultaneously, it fails to optimize data transfer for the skip connections within each branch. In contrast, PDP and the Expert successfully

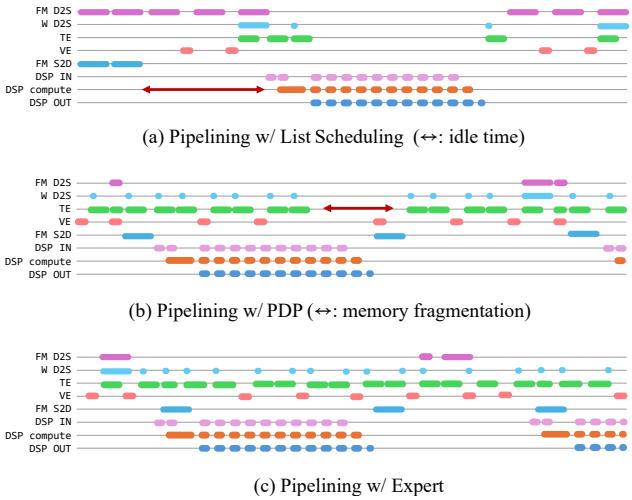


Figure 10: Scheduling timelines of EDSR.

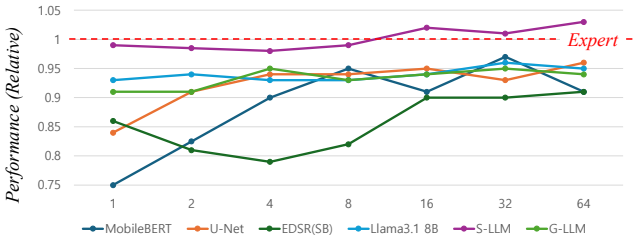


Figure 11: Impact of parameter K on inference performance (normalized to Expert).

apply data transfer optimization for skip connections by limiting the number of parallel branches to a maximum of two. As observed in Figure 10(a), LS incurs frequent data transfer (FM D2S) due to the failure of such optimization. Consequently, in terms of data transfer efficiency, PDP recorded an average of 0.92× compared to the Expert, demonstrating approximately 1.18× better performance than LS.

Regarding operation parallelism, significant differences were observed in MobileBERT and U-Net. In the case of MobileBERT, LS missed opportunities to achieve higher parallelism due to its greedy scheduling approach. Specifically, LS overlapped the TE operation with a preceding VE operation that had a shorter execution time, whereas PDP and the Expert maximized execution efficiency by overlapping the TE with a subsequent VE operation with a longer execution time. However, as previously mentioned, PDP failed to reach the expected performance in EDSR due to fragmentation, despite deriving a schedule comparable to the Expert. Figure 10(b) shows that the execution of TE is delayed due to fragmentation. Nevertheless, PDP outperformed LS across all models, recording an operation parallelism of 1.03× relative to the Expert. As a result, PDP achieved approximately 1.26× higher operation parallelism compared to LS.

5.3 Performance Analysis with Respect to Parameter K

Figure 11 presents the inference performance for each model relative to variations in the parameter K. For MobileBERT and S-LLM, the values at $K = 2$ were calculated using interpolation due to compilation errors. EDSR(SB) represents the results measured for a

Table 2: Performance and compile time across different top-k selection parameters.

Model	$k=1$		$k=2$		$k=4$		$k=8$		$k=16$		$k=32$		$k=64$	
	Perf.	Time	Perf.	Time	Perf.	Time	Perf.	Time	Perf.	Time	Perf.	Time	Perf.	Time
MobileBERT	0.75x	1.44s	-	-	0.90x	2.39s	0.95x	2.76s	0.91x	2.72s	0.96x	2.70s	0.91x	2.83s
U-Net	0.84x	37.41s	0.91x	81.66s	0.94x	145.5s	0.94x	290.8s	0.95x	593.4s	0.93x	1172s	0.96x	2513s
EDSR	0.67x	0.12s	0.63x	0.22s	0.69x	0.39s	0.71x	0.78s	0.69x	1.59s	0.67x	3.14s	0.65x	5.82s
Llama 3.1 8B	0.93x	7.75s	0.94x	12.25s	0.93x	20.54s	0.93x	40.64s	0.94x	87.32s	0.96x	190.9s	0.95x	495s
S-LLM	0.99x	8.41s	-	-	0.98x	30.15s	0.99x	60.51s	1.02x	134s	1.01x	290.3s	1.03x	630.3s
G-LLM	0.91x	2.40s	0.91x	3.38s	0.95x	6.13s	0.93x	12.75s	0.94x	26.86s	0.95x	59.7s	0.94x	123.9s

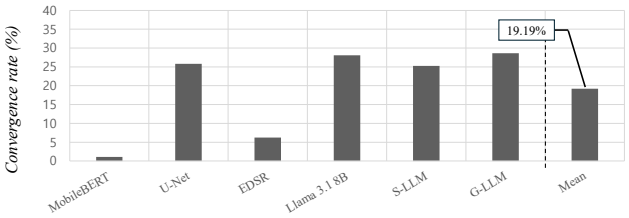


Figure 12: Average rate of convergence to the same DP state.

single block where fragmentation does not occur. Table 2 details the specific compilation times and inference performance for each K value. In general, inference performance tended to improve as K increased. The performance gains were particularly significant in models with abundant opportunities for parallel execution, such as MobileBERT, U-Net, and EDSR(SB). For instance, MobileBERT's performance improved from 0.75× of the Expert at $K = 1$ to a maximum of 0.96× at $K = 32$. While compilation time generally increased in proportion to K, it tended to saturate once the schedule approached a near-optimal solution. In the case of MobileBERT, the compilation time began to saturate at $K = 8$, maintaining around 2.7 seconds. Conversely, for U-Net, the compilation time increased continuously to approximately 42 minutes at $K = 64$, with other models also showing a gradual increase in proportion to K.

5.4 Effect of DP State Memoization

To evaluate the effectiveness of overlapping subproblems, we conducted additional experiments with $K = 64$. Figure 12 shows the ratio of candidates that converge to the same DP state for each model. On average, 19.19% of the candidates reached an identical DP state, with G-LLM recording the highest ratio of 28.62%. This demonstrates that memoization is effective in eliminating redundant search paths and suppressing the growth of the candidate set. Memoization also affects the quality of the derived schedules. While Llama 3.1 8B showed little difference depending on whether memoization was applied, U-Net suffered a 7% performance drop without it. This trend was more pronounced in models with a higher convergence ratio and greater sensitivity to K. Meanwhile, the compilation time remained stable regardless of memoization, since top-K selection already bounds the exponential growth of the candidate set.

6 Related Works

Extensive research has proposed various scheduling algorithms to optimize the inference performance of deep learning models. While some studies [16, 35, 47] modeled the scheduling problem using integer linear programming, they face limitations as their time complexity increases exponentially with model scale. Another framework [37] employs genetic algorithms, and others [12, 33, 34] have explored reinforcement learning. However, these approaches also struggle with large-scale models because they do not guarantee convergence within polynomial time.

Regarding dynamic programming-based research, Tarnawski et al. [47] implemented polynomial time scheduling through topological sorting based on a Hamiltonian path. However, this work targets GPU clusters with independent resources, making it unsuitable for heterogeneous mobile environments that utilize shared resources. Jung et al. [27] introduced a heuristic variable called the process rate to restrict the search space and solve the problem in polynomial time, but it specifically targets a single precision-scalable NPU. Consequently, it fails to account for the asynchronous execution characteristics of heterogeneous environments, leading to suboptimal schedules.

Furthermore, numerous studies [10, 17, 22–24, 36] address parallel execution in distributed accelerator environments, but they generally assume high-end server systems with independent memory resources. For mobile SoCs, Xu et al. [52] address the prefill bottleneck caused by the static shape support of NPUs by splitting prompts into chunks and offloading operations to the CPU and GPU, while Chen et al. [3] propose tensor partitioning tailored to the prefill and decoding phases with unified-memory synchronization. However, both focus on tensor-level offloading across accelerators. Neither accounts for the scratchpad SRAM shared by the TE and VE, and thus for the SRAM shortages and schedule deadlocks that arise in resource-constrained mobile environments.

7 Conclusion

This paper proposes a dynamic programming-based scheduling framework optimized for heterogeneous mobile environments. By applying priority that reflects data transfer efficiency, PDP balances computational parallelism and data transfer. To solve the exponential complexity, we introduced a pruning technique that keeps scheduling within practical polynomial time, together with a deadlock prevention mechanism for stable schedule generation. Experimental results show that PDP reduces DMA overhead and increases computational parallelism over conventional list scheduling,

providing a practical solution for large-scale models on resource-constrained heterogeneous mobile systems.

Acknowledgments

We would like to thank Seokwon Kang for his valuable comments and suggestions. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (RS-2020-II201361), and by the National Research Foundation of Korea (NRF) grant (RS-2025-00560614) funded by the Korea government (MSIT). This work was also supported by Samsung Electronics Co., Ltd (IO240426-09742-01). Yongjun Park is the corresponding author.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

[2] Hamid Arabnejad and Jorge G. Barbosa. 2014. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2014), 682–694. doi:10.1109/TPDS.2013.57

[3] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 359–374. doi:10.1145/3731569.3764808

[4] 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. doi:10.1145/2541940.2541967

[5] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. 2020. A Survey of Accelerator Architectures for Deep Neural Networks. *Engineering* 6, 3 (2020), 264–274. doi:10.1016/j.eng.2020.01.007

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, Seoul, Republic of Korea, 367–379. doi:10.1109/ISCA.2016.40

[7] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (June 2020), 48–57. doi:10.1145/3361682

[8] Samsung Electronics. 2026. Artificial Intelligence. https://research.samsung.com/artificial-intelligence Accessed: 2026-09-09.

[9] Samsung Electronics. 2026. Exynos 2600. https://semiconductor.samsung.com/processor/mobile-processor/exynos-2600/ Accessed: 2026-09-09.

[10] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 431–445. doi:10.1145/3437801.3441593

[11] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 751–764. doi:10.1145/3037697.3037702

[12] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholm, Sweden, 1676–1684. https://proceedings.mlr.press/v80/gao18a.html

[13] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429. doi:10.1137/0117039

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[15] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Berkeley, CA, USA, 705–719. https://www.usenix.org/conference/nsdi21/presentation/guo

[16] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. 2021. Towards optimal placement and scheduling of DNN operations with Pesto. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 39–51. doi:10.1145/3464298.3476132

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[18] Hanan Hussain, P. S. Tamizharasan, and C. S. Rahul. 2022. Design possibilities and challenges of DNN models: a review on the perspective of end devices. *Artificial Intelligence Review* 55, 7 (2022), 5109–5167. doi:10.1007/s10462-022-10138-z

[19] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. 2021. Sparsity-aware and re-configurable NPU architecture for samsung flagship mobile SoC. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, Piscataway, NJ, USA, 15–28. doi:10.1109/ISCA52012.2021.00011

[20] Hyuk-Jin Jeong, JiHwan Yeo, Cheongyo Bahk, and JongHyun Park. 2023. Pin or Fuse? Exploiting Scratchpad Memory to Reduce Off-Chip Data Transfer in DNN Accelerators. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/3579990.3580017

[21] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (Portland, Oregon) *(MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 235–247. doi:10.1145/3498361.3538948

[22] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholm, Sweden, 2274–2283. https://proceedings.mlr.press/v80/jia18a.html

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[24] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. mlsys.org, Stanford, CA, USA, 1–13. https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3079856.3080246

[26] Hanwoong Jung, Hexiang Ji, Alexey Pushchin, Maxim Ostapenko, Wenlong Niu, Ilya Palachev, Yutian Qu, Pavel Fedin, Yuri Gribov, Heewoo Nam, et al. 2023. Accelerating Deep Neural Networks on Mobile Multicore NPUs. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 236–248. doi:10.1145/3579990.3580015

[27] Kiung Jung, Seok Namkoong, Hongjun Um, Hyejun Kim, Youngsok Kim, and Yongjun Park. 2024. Orchestrating Multiple Mixed Precision Models on a Shared Precision-Scalable NPU. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Copenhagen, Denmark) *(LCTES 2024)*. Association for Computing Machinery, New York, NY, USA, 72–82. doi:10.1145/3652032.3657571

[28] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[29] Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation & Test in Europe Conference &*

Exhibition (DATE). IEEE, Dresden, Germany, 343–348. doi:10.23919/DATE.2018.8342033

[30] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. 2017. Enhanced Deep Residual Networks for Single Image Super-Resolution. arXiv:1707.02921 [cs.CV] https://arxiv.org/abs/1707.02921

[31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Berkeley, CA, USA, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[32] Sachin Mehta and Mohammad Rastegari. 2022. MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer. arXiv:2110.02178 [cs.CV] https://arxiv.org/abs/2110.02178

[33] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. In *International Conference on Learning Representations*. OpenReview.net, Vancouver, BC, Canada, 11 pages. https://openreview.net/forum?id=Hkc-TeZ0W

[34] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, Sydney, NSW, Australia, 2430–2439. https://proceedings.mlr.press/v70/mirhoseini17a.html

[35] Seok Namkoong, Taehyeong Park, Kiung Jung, Jinyoung Kim, and Yongjun Park. 2025. SortingHat: System Topology-aware Scheduling of Deep Neural Network Models on Multi-GPU Systems. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 339–354. doi:10.1145/3721145.3728490

[36] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3341301.3359646

[37] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2020. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. In *International Conference on Learning Representations*. OpenReview.net, Addis Ababa, Ethiopia, 24 pages. https://openreview.net/forum?id=rkxDoJBYPB

[38] Jun-Seok Park, Taehee Lee, Heonsoo Lee, Changsoo Park, Youngsang Cho, Mookyung Kang, Heeseok Lee, Jinwon Kang, Taeho Jeon, Dongwoo Lee, et al. 2025. 16.3 An on-Device Generative AI Focused Neural Processing Unit in 4nm Flagship Mobile SoC with Fan-Out Wafer-Level Package. In *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 68. IEEE, San Francisco, CA, USA, 286–288. doi:10.1109/ISSCC49661.2025.10904722

[39] Jun-Seok Park, Changsoo Park, Suknam Kwon, Taeho Jeon, Yesung Kang, Heonsoo Lee, Dongwoo Lee, James Kim, Hyeong-Seok Kim, YoungJong Lee, et al. 2022. A Multi-Mode 8K-MAC HW-Utilization-Aware Neural Processing Unit with a Unified Multi-Precision Datapath in 4nm Flagship Mobile SoC. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, San Francisco, CA, USA, 246–248. doi:10.1109/ISSCC42614.2022.9731639

[40] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for Convolutional Neural Networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, Asheville, NC, USA, 13–19. doi:10.1109/ICCD.2013.6657019

[41] Qualcomm. 2026. Qualcomm Hexagon NPU. https://www.qualcomm.com/processors/hexagon Accessed: 2026-09-09.

[42] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi (Eds.). Springer International Publishing, Cham, 234–241. doi:10.1007/978-3-319-24574-4_28

[43] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, et al. 2025. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms. *ACM Comput. Surv.* 57, 11, Article 286 (June 2025), 39 pages. doi:10.1145/3729215

[44] Shriyank Somvanshi, Md Monzurul Islam, Gaurab Chhetri, Rohit Chakraborty, Mahmuda Sultana Mimi, Sawgat Ahmed Shuvo, Kazi Sifatul Islam, Syed Javed, Sharif Ahmed Rafat, Anandi Dutta, et al. 2025. From Tiny Machine Learning to Tiny Deep Learning: A Survey. *ACM Comput. Surv.* 58, 7, Article 168 (Dec. 2025), 33 pages. doi:10.1145/3776588

[45] Rafael Sousa, Byungmin Jung, Jaehwa Kwak, Michael Frank, and Guido Araujo. 2021. Efficient Tensor Slicing for Multicore NPUs using Memory Burst Modeling. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, Virtual Conference, 84–93. doi:10.1109/SBAC-PAD53543.2021.00020

[46] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 2158–2170. doi:10.18653/v1/2020.acl-main.195

[47] Jakub Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of DNN graph operators. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1296, 13 pages. https://proceedings.neurips.cc/paper_files/paper/2020/hash/b14680dec683e744ada1f2fe08614086-Abstract.html

[48] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2025. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[49] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. https://www.usenix.org/conference/osdi22/presentation/unger

[50] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025. Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models. *ACM Comput. Surv.* 57, 9, Article 228 (April 2025), 39 pages. doi:10.1145/3724420

[51] Bartłomiej Wróblewski, Gioele Gottardo, and Anastasios Zouzias. 2025. Parallel Scan on Ascend AI Accelerators. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Milan, Italy, 1290–1292. doi:10.1109/IPDPSW66978.2025.00216

[52] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 445–462. doi:10.1145/3669940.3707239

[53] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. arXiv:2105.04663 [cs.DC] https://arxiv.org/abs/2105.04663

[54] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 81–93. doi:10.1145/3458864.3467882

[55] Ruichen Zhang, Guangyuan Liu, Yinqiu Liu, Changyuan Zhao, Jiacheng Wang, Yunting Xu, Dusit Niyato, Jiawen Kang, Yonghui Li, Shiwen Mao, et al. 2026. Toward Edge General Intelligence With Agentic AI and Agentification: Concepts, Technologies, and Future Directions. *IEEE Communications Surveys & Tutorials* 28 (2026), 4285–4318. doi:10.1109/COMST.2026.3651702

[56] 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. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages. https://www.usenix.org/conference/osdi20/presentation/zheng

[57] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[58] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu