Optimizing Compute Core Assignment for Dynamic Batch Inference in AI Inference Accelerator

Anonymous Author(s)

Abstract

Modern AI inference accelerators offer high-performance and power-efficient computations for machine learning models. Most accelerators employ static inference to enhance performance, which requires models to be compiled with predetermined input batch sizes and intermediate tensor shapes. However, static inference can lead to program failures or inefficient execution when processing batched data of varying sizes, a scenario known as dynamic batch inference. This work addresses this challenge by focusing on the emerging multicore AI inference accelerators that offer flexible compute core assignment. We propose to dynamically partition the input batch data into smaller batches, and create multiple model instances to process each partition in parallel. The challenge lies in how to determine the optimal number of model instances, the proper batch size for each handling model, and the assignment of compute cores among the models, to minimize the inference time. To solve the problem, we construct an accurate profiling-based cost model and devise a dynamic programming algorithm to determine the best configuration. Experimental results indicate that our method achieves 3.05× higher throughput on average in multi-person pose estimation benchmarks, compared to the EdgeTPU inference strategy.

Keywords

Deep learning, AI accelerator, dynamic inference, dynamic programming

ACM Reference Format:

Anonymous Author(s). 2025. Optimizing Compute Core Assignment for Dynamic Batch Inference in AI Inference Accelerator. In *Proceedings of ACM SAC Conference (SAC'25).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 Introduction

Recent advancements in deep learning have demonstrated outstanding performance in various fields, such as computer vision [23, 33], natural language processing [4, 29], and many others. These advancements are driven by the increased model sizes, aimed at learning complex patterns from large datasets. The vast number of model parameters also leads to significant computational costs. For example, YOLOv5-large [30] requires 109.1 billion floating-point operations (FLOPs) for inference, while LLaMA-7B [29] demands approximately 1.7 trillion FLOPs for an input length of 128. This trend of increasing model sizes and computational requirements is expected to continue in the future [11].

The significant computational cost poses a critical challenge for real-time inference. To tackle this issue, many high-performance and power-efficient *AI inference accelerators* have been introduced to enhance model inference speed [1, 2, 7, 9, 12, 17, 22, 26, 32].

To maximize efficiency, many AI inference accelerators, e.g., Google EdgeTPU [26] and Intel NPU [12], employ *static* inference. They require compiled models to be bound to *fixed* tensor shapes, including batch size and dimensions. Unlike training where models are often handled with dynamic shapes to accommodate various training conditions (e.g., variable text lengths), inference with static shapes allows the compiler to make assumptions about the data and optimize the computation ahead of time. Compilers can enable more efficient memory management and optimizations, such as operation fusion [5, 6, 8, 20, 31], constant folding [34], and loop optimizations [3, 6, 16], thus resulting in better utilization of the hardware. These benefits are crucial for achieving high performance and power efficiency in AI inference tasks.

However, restricting models to fixed shapes presents challenges when the size of input batches varies during inference—a scenario known as *dynamic batch inference*. For example, dynamic batch inference can occur in the downstream tasks of object detection, e.g., pose estimation and object tracking/recognition. Figure 1 depicts the issue using the two-stage multi-person pose estimation (MPPE) model architecture. In the first stage, an object detection model identifies and crops all individuals. In the second stage, a pose estimation model determines the posture of each person. Since the number of people can vary over time, the pose estimation model may fail to run on the accelerator if the detected samples do not match the batch size of the compiled model. Therefore, a solution to tackle this issue while speeding up inference is essential.

Recently, multicore architecture has been adopted in AI accelerators to meet the growing computational demands of modern workloads. In addition to utilizing static inference for efficient computation, these accelerators (e.g., Qualcomm Cloud AI 100 [22]) offer the flexibility to execute multiple models in parallel and bind workloads to compute cores. These new features provide opportunities to enable high-performance inference with dynamic batches.

To tackle the problem of variable input sizes, we propose to dynamically partition the input data into subsets. Then, we create multiple model instances to process the partitioned data in parallel. Hence, the challenge becomes how to properly partition the input data, and how to determine the optimal number of model instances, the batch size each model should handle, and the distribution of compute cores among the models. To this end, we first develop a cost model by profiling the inference time of models with varying batch sizes and compute core assignments. Using this cost model, we then derive a dynamic programming algorithm to schedule the input workloads and to further find the optimal mapping of compute cores and batch sizes. Addressing the large search space of this problem,

SAC'25, March 31 –April 4, 2025, Sicily, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/xx.xxx/xxx_x

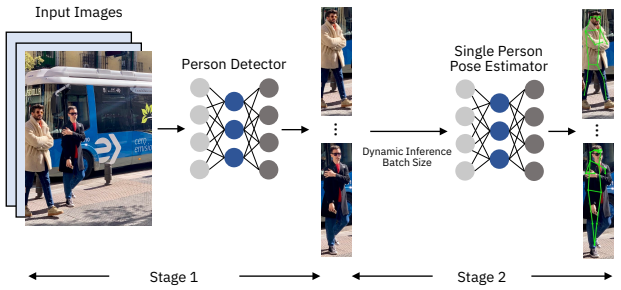

Figure 1: Architecture of the two-stage multi-person pose estimation (MPPE). Stage 1: Input images from multiple cameras are processed by a person detection model to identify and crop human bodies. Stage 2: The cropped persons are fed into single-person pose estimation models (SPPE) to estimate the pose of each individual. The number of input samples for SPPE models are dynamic, varying based on the number of human bodies detected in stage 1.

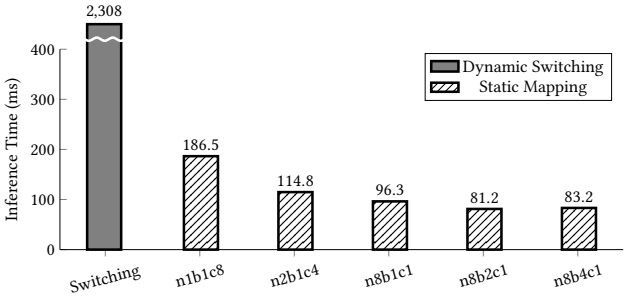

Figure 2: MMPose(ResNet50) inference time to process eight input data of the sample sizes {4,8,12,16,20,24,28,32}. The result indicates (1) dynamic switching results in significantly longer inference time than static mapping, due to high model loading/unloading overhead; (2) different mappings can affect the inference times significantly. Notation nXbYcZ denotes: X model(s) are executed in parallel, with each model assigned with batch size Y and Z compute core(s).

we introduce an analytical method to reduce the problem size based on performance observations. Finally, we evaluate our approach on multi-person pose estimation benchmarks, demonstrating its effectiveness in real-world scenarios.

In summary, this paper makes the following contributions:

- We develop a dynamic programming algorithm to determine the optimal configuration for efficient dynamic batch inference on multicore AI inference accelerators.
- We present a systematic method to narrow down the search space by leveraging the insights from performance profiles.
- Experimental results demonstrate that our method achieves an average of 3.05× higher throughput in multi-person pose estimation, compared to the EdgeTPU inference strategy.

The rest of this paper is organized as follows. Section 2 describes related works and the initial experiment that motivates this work. Section 3 formally defines the compute core assignment problem and presents our dynamic programming algorithm. Section 4 evaluates the efficiency and effectiveness of our method. Section 5 concludes the paper.

2 Motivation and Related Work

2.1 Motivation

This work focuses on the Qualcomm Cloud AI 100 accelerator due to its widespread use in robotic platforms and AI PCs. Qualcomm Cloud AI 100 is a configurable multicore AI accelerator that supports static inference only. Before running models on the accelerator, users must compile the models into executables with two pre-defined parameters: (a) the *batch size* of the model and (b) the *number of compute cores* the model will use. During inference, the runtime system assigns compute core(s) to the model based on the pre-defined value. Moreover, the assigned compute cores are *exclusively* used by the model. A model can fail to run if the number of available cores is insufficient during compute core allocation, or if there is a mismatch between the input data size and the batch size a compiled model is designed to handle.

One possible approach to tackle dynamic batch inference on the accelerator is to prepare multiple models, each compiled with a possible input size. At runtime, the system dynamically switches to the appropriate model based on the input size. Since only one model is executed at a time, we can compile each model to utilize all compute cores. Another approach, employed by many accelerators such as Google EdgeTPU, is to compile the model with a batch size of "one." This allows the system to handle arbitrary input data sizes by processing the input samples one at a time. In this approach, we can also map all compute cores to the model.

In contrast to the above-mentioned methods, we propose to split the input data into several partitions, spawning multiple model instances for parallel processing, and having the models process the partitions over one or multiple iteration(s), without switching models during execution.

We conduct an initial experiment on the Qualcomm Cloud AI 100 to compare the inference time of the aforementioned approaches, using the MMPose (ResNet50) single-person pose estimation model with a dynamic number of input samples. Since our approach requires selecting the following parameters: the number of parallel models n, the batch size for each model b, and the number of cores allocated to each model c, we choose several valid configurations for comparison. The experiment processes eight input data with different sizes, and eight compute cores are used.

Figure 2 shows the inference time results. Several observations are drawn from the results. (1) The dynamic switching approach takes significantly longer time than the static mapping approach. Though this approach can process each input data in a single run, it incurs significant overhead from model loading/unloading. (2) The EdgeTPU mapping strategy (i.e., n1b1c8) performs poorly compared to other mapping strategies. Though this approach is easy to implement in both software and hardware, it is inadequate for achieving optimal performance on multicore accelerators. (3) Different configurations can significantly influence the execution time.

These observations emphasize the necessity of finding optimal configurations when accelerating models with dynamic batch inference on the multicore accelerators, thereby motivating this paper to design an efficient algorithm to achieve the goal.

2.2 Dynamic Neural Network Inference

Many machine learning tasks involve intrinsic features of dynamic inference. Dynamic inference can be categorized into three types: (a) dynamic batch size, (b) dynamic tensor shape, and (c) dynamic execution flow. Similar to multi-person pose estimation, downstream tasks in video object detection, such as trajectory prediction and multi-object tracking, face dynamic batch sizes during inference due to the variable number of objects detected. Dynamic shapes, which arise from input samples of varying shapes, are commonly seen in graph neural networks (GNNs) that process graph data of different structures [27]. Dynamic execution flow occurs when model computation involves control flow structures such as conditional statements, loops, and recursion [28]. This research focuses on optimizing the inference performance of dynamic batch sizes.

Several tools [18, 19, 21, 35] implement automatic dynamic batching during inference, which combines inference requests to enhance throughput. Instead of targeting AI inference accelerators, these tools are primarily designed for the more general-purpose processors such as CPUs and GPGPUs, which support model execution with dynamic shapes. In contrast, our work focuses on optimizing dynamic batch inference while preserving the performance advantages of static inference. Our method can potentially improve performance of these works on AI inference accelerators.

2.3 AI Accelerators

AI accelerators have become integral components in cloud services, edge devices, and recently, AI-enabled personal computers (PCs). In clouds, Amazon EC2 DL2q adopts Qualcomm Cloud AI 100 for inference services, while Google Cloud Platform provides TPUs [13] for high-performance neural network training and inference. In edge devices, AI accelerators are primarily used for real-time inference, with the additional benefits of reliability and privacy. Google EdgeTPU [26] and Intel NCS2 [12] are two popular accelerators designed to enhance AI workloads at the edge. Additionally, many robotic platforms are powered by the Qualcomm Cloud AI 100 Edge version, which features nine neural processing cores and can offer up to 70 trillion operations per second (TOPS) while consuming only 15 watts [22]. Current AI accelerators are trending toward multicore architectures, thus making our proposed method for optimizing dynamic batch inference potentially impactful.

3 Method

Accelerating neural networks with dynamic batch inference presents several challenges. First, accelerators prefer static inference, while the input data sizes can be dynamic in real-world scenarios. Second, although modern inference accelerators provide flexibility in assigning compute cores, different mappings can lead to significant variations in execution time, as shown in our initial experiments. To address these issues, we propose dynamically partitioning input samples into smaller subsets and creating multiple model instances to process each partition efficiently in parallel. In this section, we formally define the compute core assignment problem and introduce a two-step algorithm to solve it. We focus on the multicore accelerators, where multiple models can be executed in parallel and the compute core(s) are used exclusively for each model.

3.1 Problem Definition

The objective of this work is to develop an algorithm that optimally assigns compute cores and batch sizes to models for processing input samples of dynamic sizes, with the aim of minimizing inference time. This problem can be decomposed into two subproblems: (1) identifying the valid candidate compute core/batch size/model mappings, and (2) finding the optimal mapping that minimize the inference time. By solving these two subproblems, we can determine the minimal inference time for any given numbers and sizes of input samples.

We define the following parameters:

- D: The total number of available compute cores.
- $B = (b_1, b_2, \ldots, b_k)$: The set of distinct, possible batch sizes that can be mapped to the models.
- $N = (n_1, n_2, \ldots, n_k)$: The number of models for each batch size $b_i \in B$ within a mapping.
- $C = (c_1, c_2, \ldots, c_k)$: The number of compute cores allocated for the models n_i of batch size b_i within a mapping.

Since we will partition input samples into small batches, we first define B, which is the set of distinct, possible batch sizes that can be mapped to the models. The elements of B can be any combination chosen by the user. A simple approach is to set B as consecutive positive integers, i.e., $B = (1, 2, \ldots, k)$, where, for example, $k = 16$.

Next, we determine the number of models n_i and the number of compute cores c_i mapped to each batch size b_i. Note that the values of n_i and c_i within a mapping could be zero, indicating the absence of models with those batch sizes. In addition, each model must be assigned at least one compute core. Thus, a mapping must meet the constraints in Equation 1:

$$D = \sum_{i=1}^{k} c_i \,, \quad c_i \geq n_i \geq 0, \tag{1}$$

where D represents the total number of available compute cores.

In order to handle any input sample sizes, we always put the batch size *one* in B and assign at least one model to it, thus guaranteeing that the system can always establish a valid schedule.

For the first subproblem, the objective is to identify all valid mappings S that satisfy the constraints in Equation 1. Given that the number of such mappings is typically large, we quantitatively analyze S in Section 3.2 and then eliminate suboptimal mappings in Section 3.3. For the second subproblem, the objective is to first find the best execution schedule for models within each candidate mapping, and then determine the optimal mapping with minimal execution time. The algorithm to solve the problem is presented in Section 3.4.

Now we formally define the problem as follows. We have a set of batch sizes B and the total number of compute cores D. We want to first find the candidate mappings of batch sizes, compute cores, and models. Then we optimize the execution schedule for models according to the mappings, and find the one with minimal execution time.

3.2 Search Space

To find the possible mappings of batch sizes, compute cores, and models, we break this problem into two steps: (1) Allocating the available compute cores D to different batch sizes b_i, which determines the values of c_i. (2) Determining the number of models n_i and the allocation of compute cores c_i to the models. The mappings must satisfy the constraints in Equation 1.

For the first step, it is a combinatorial problem that counts non-negative integer to the equation $D = \sum_{i=1}^{k} c_i$, where k is the number of distinct batch sizes and D is the total number of compute cores. The number of combinations is given by the combinatorial function $H(k, D)$.

In the second step, the assignment of n_i models to c_i compute cores is equivalent to solving an integer partition problem. Table 1 shows an example of possible assignments with four compute cores. We define the partition function $p(c_i)$, which returns the number of possible ways to partition c_i compute cores among n_i models.

Therefore, the total number of mappings S is:

$$|S| = H(k, D) \cdot \prod_{i=1}^{k} p(c_i) \quad \forall c_i \neq 0 \tag{2}$$

Table 1: Five different mappings of models across a total number of four compute cores: $p(c_i = 4) = 5$.

Number of models n_i	Core mapping
4	(1,1,1,1)
3	(1,1,2)
2	(1,3), (2,2)
1	(4)

3.3 Narrowing the Search Space

After determining the number of possible mappings, we can proceed to schedule the models for each mapping. However, we observe that the number of mappings can be substantial. For instance, when $D = 16$, $k = 8$, and $c_i = 2\ \forall i \in [1, 8]$, the total number of mappings is approximately 60 million from Equation 2. Therefore, it is crucial to narrow down the search space to reduce the algorithm's runtime.

We can reduce the search space by leveraging insights from the performance profiles. First, we can limit the maximum number of cores a model can use in c_i, thereby reducing the term $\prod_{i=1}^{k} p(c_i)$. For example, we measure the MMPose throughputs with varying numbers of compute cores by fixing the batch size. Figure 3 shows the results normalized against one core. The result indicates that the throughput increases and gradually saturates with the number of compute cores. After $c_i = 6$, adding more cores does not improve the throughput. Therefore, it is reasonable to impose a limit on the maximum number of cores for the models. This method prevents unnecessary exploration of the combinations where a model uses an excessive number of cores.

Second, we can impose a limit on the potential models' batch sizes, thereby reducing k and the term $H(k, D)$. Figure 4 shows the normalized throughputs of different batch sizes against batch size one by fixing the number of cores used. The result indicates

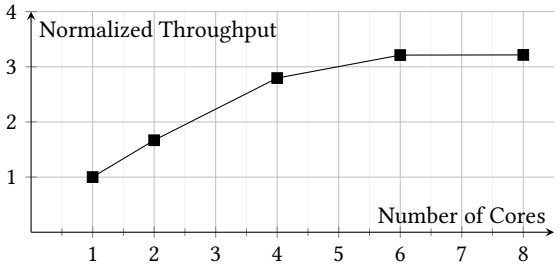

Figure 3: MMPose throughputs of varying numbers of compute cores by fixing the batch size (normalized with one core).

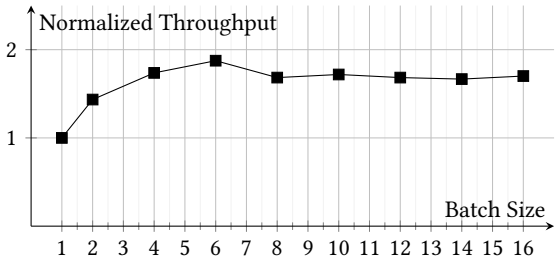

Figure 4: MMPose throughputs of varying batch sizes by fixing the number of cores used (normalized with batch size one). Each model is mapped to two compute cores.

that while the throughput increases with the batch size, this improvement also saturates after a certain point, which can be set as the maximum batch size. Furthermore, using larger batch sizes can lead to idle resources if the size of input data is smaller than the model's batch size. Hence, this method effectively eliminates the search for unnecessarily large batch sizes. By applying the aforementioned two methods, we can significantly reduce the total number of mappings in S.

3.4 Model Execution Schedule

After narrowing down the candidate mappings, the next step is to determine, according to the mapping, how to partition input data into smaller batches and schedule the models to process the partitioned batches. Note that the compute cores are used exclusively. Hence, the models are executed in parallel during inference without resource contention.

To support our algorithm, we first construct a profiling-based inference time table. We collect all configurations (i.e., batch size and compute core assignments) from the candidate mappings and compile the models accordingly. Then, we measure the inference times of the models to form the time table. Figure 5 (left) illustrates an example of the table. This table is also utilized to narrow down the candidate mappings, as described in Section 3.2 and 3.3.

Based on the time table, the goal is to partition the input data into smaller batches for each candidate mapping and schedule the models to process the partitioned batches with the shortest execution time. To this end, we develop a dynamic programming algorithm to solve the problem. We define the following parameters:

- $M = (m_1, m_2, \ldots, m_l)$: Model instances of the candidate mapping $s \in S$.
- e_i: Inference time of model m_i running for one iteration.

We introduce a function $T_s(x)$, which represents the shortest execution time for the mapping s, given the input data of size x. In addition, we define $d_s(i, x)$, which is the total number of batches processed by model m_i for the input size x. Since all models are run in parallel, the overall execution time is determined by the slowest model instance, as in Equation 3:

$$T_s(x) = \max_{1 \le i \le l} \Big(d_s(i, x) \cdot e_i \Big) \tag{3}$$

where e_i is the inference time of model m_i running for one iteration.

Now we build the recurrence from $d_s(i, x)$ and $T_s(x)$ to solve the problem. We initialize $d_s(i, 0)$ and $T_s(0)$ to zero, as there is no input data.

Next, we enumerate the input size x from 1 to $x_{\max}$ and iterate over all models. If an input size is less than a model's batch size, $d_s(i, x) = 0$ because there is insufficient data. Otherwise, we try to partition the input data, and we can derive $d_s(i, x)$ from $d_s(i, x - h_i)$, where h_i is the batch size of model m_i.

The key idea is that assuming we assign h_i data to the model m_i, the model will process one additional batch compared to the count of $d_s(i, x - h_i)$. The execution time to process these batches is

$$\Big(d_s(i, x - h_i) + 1 \Big) \cdot e_i$$

Now we identify the model instance $i_{\min}$ that has the shortest execution time, as Equation 4.

$$i_{\min} = \underset{1 \le i \le l \,\wedge\, x \ge h_i}{\arg\min} \Big((d_s(i, x - h_i) + 1) \cdot e_i \Big) \tag{4}$$

By selecting the model with the shortest execution time, we ensure that the overall inference time $T_s(x)$ is minimized.

Based on the result in Equation 4, we update $d_s(i, x)$ for each model with Equation 5.

$$d_s(i, x) = \begin{cases} d_s(i, x - h_i) + 1 & \text{if } i = i_{\min} \\ d_s(i, x - h_i) & \text{otherwise} \end{cases} \tag{5}$$

Then, we can compute $T_s(x)$ using Equation 3.

The pseudo-code of the dynamic programming algorithm for model execution scheduling is in Algorithm 1. Using this algorithm, we can determine the optimal execution schedule for a candidate mapping and compute the optimal execution time $T_s(x)$ for any input data size $x \ge 1$. Figure 5 (right) illustrates an example of the execution schedule. Based on the time table, our dynamic programming efficiently partitions inputs of variable sizes across the model instances, and minimizes the overall execution time as a result.

Finally, we apply Algorithm 1 to all candidate mappings. The result with the best schedule among these candidate mappings is selected as the final solution.

Algorithm 2 lists the pseudo-code of the overall compute core assignment algorithm. The algorithm begins by building the inference time table (line 3-7). Based on the profiled model performance, we reduce the size of candidate mappings (line 8-9). Then all candidate mappings are scheduled with the dynamic programming algorithm to determine the best configuration (line 10-14).

Algorithm 1 Model Execution Scheduling

Input: Mapping s, Input data sizes $X = (1, 2, \ldots, x_{\max})$.
Output: T_s, d_s
$M = (m_1, m_2, \ldots, m_l)$ ▷ Model instances in mapping s
$H = (h_1, h_2, \ldots, h_l)$ ▷ Batch size of model instances
$E = (e_1, e_2, \ldots, e_l)$ ▷ Inference time of model instances
for $x \leftarrow 1$ to $x_{\max}$ **do**
  $i_{\min} = \underset{1 \le i \le l \,\wedge\, x \ge h_i}{\arg\min} \Big((d_s(i, x - h_i) + 1) \cdot e_i \Big)$
  for $i \leftarrow 1$ to l **do** ▷ Update d_s
    if i is equal to $i_{\min}$ **then**
      $d_s(i, x) = d_s(i, x - h_i) + 1$
    else
      $d_s(i, x) = d_s(i, x - h_i)$
    end if
  end for
  $T_s(x) = \max_{1 \le i \le l} \Big(d_s(i, x) \cdot e_i \Big)$
end for
Return T_s, d_s

Algorithm 2 Optimal Compute Core Assignment

1: **Input:** Total number of cores D, Set of distinct batch sizes $B = \{b_1, b_2, \ldots b_k\}$, Number of input samples $X = (1, 2, \ldots, x_{\max})$.
2: **Output:** Optimal mapping s_{best}

3: **for** $b_i \in B$ **do** ▷ Establish the profiling-based cost model
4: **for** $c_j \in [1, D]$ **do**
5: Profile the model and fill in the time table entry $e(b_i, c_j)$
6: **end for**
7: **end for**
8: Compute candidate mappings S based on constraints in Eq. 1
9: Narrow down candidate mappings S with profiled performance

10: **for** $s \in S$ **do** ▷ Dynamic programming for scheduling
11: Initialize $T_s(0) \leftarrow 0$
12: Initialize $d_s(i, 0) \leftarrow 0 \quad \forall i \in [1, l]$
13: T_s, d_s = ModelExecutionScheduling(s, X)
14: **end for**

15: Initialize $s_{\text{best}} \leftarrow$ null, $t_{\min} \leftarrow \infty$
16: **for** $s \in S$ **do** ▷ Determine the best schedule among S
17: Compute t as the average $T_s(x)$ for x from 1 to $x_{\max}$
18: **if** $t < t_{\min}$ **then**
19: $t_{\min} \leftarrow t$, $s_{\text{best}} \leftarrow s$
20: **end if**
21: **end for**
22: **Return** s_{best}

Lines 15-21 in Algorithm 2 present a naive example assuming the sizes of input data are uniformly distributed between 1 and $x_{\max}$; thus, the execution time of each candidate mapping is averaged across this input size range. Users can also apply other constraints, such as selecting a mapping that maximizes the number of objects processed in real-time scenarios.

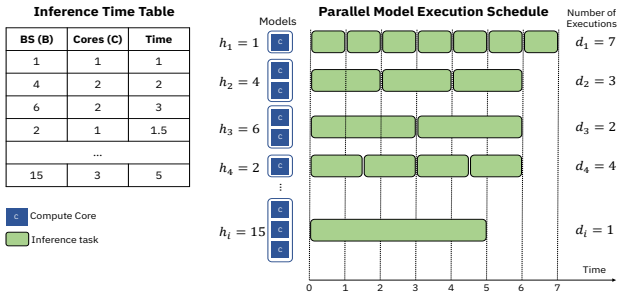

Figure 5: Parallel model execution schedule. Our dynamic programming algorithm efficiently distributes the dynamic batch inference tasks across the models based on a profiling-based cost model.

4 Experiment

In this section, we present a comprehensive evaluation of the proposed compute core assignment approach. We start by detailing the experimental settings. Subsequently, we compare the inference throughput across various mapping strategies, examine the real-time performance, and assess the accuracy of our cost model.

4.1 Experimental Settings

4.1.1 Models and Benchmark Datasets.

We evaluate our approach using the two-stage multi-person pose estimation (MPPE) application. Since MPPE shares a similar architecture with other downstream tasks in object detection, we consider it a representative target to demonstrate the advantages of our method. To further generalize the evaluation, we use models with varying sizes and backbones. Specifically, for the person detection model, we select YOLOv5 [30] in two sizes: YOLOv5m and YOLOv5l, which have 21.1 million and 46.5 million parameters, respectively. For the single-person pose estimation model, we use MMPose [25] with three different backbones: MobileNetV2 [24], ResNet50, and ResNet101 [10], with 9.6, 34, and 53 million parameters, respectively.

We compare different mapping strategies using two benchmark datasets for MPPE: MS-COCO [15] and CrowdPose [14]. The input resolutions are resized to 640×640 for YOLOv5, and 256×192 for MMPose. All models are compiled using Qualcomm AIC compiler v1.10 and quantized to 8-bit integers.

4.1.2 Implementation.

The evaluation is performed on the Qualcomm Robotics RB6 Platform, which consists of an Edge version of the Qualcomm Cloud AI 100 accelerator [22]. The accelerator includes nine neural processing cores operating at a frequency of 600 MHz. All compute cores share 8 GB of device DRAM.

We implement a pipelining strategy to overlap the execution of YOLOv5 and MMPose. In addition, we assign one core and two cores to YOLOv5m and YOLOv5l, respectively. The remaining cores are allocated for MMPose. We ensure that MMPose has a longer inference time than YOLOv5 to facilitate a more appropriate comparison of the mapping methods. In the following experiments, we use combinations of the YOLOv5 and MMPose names (e.g., YOLOv5m-MobileNetV2) to denote the hybrid model. We use the notation nXbYcZ to denote the creation of X models, each mapped with a batch size of Y and Z compute cores.

4.1.3 Evaluation Metrics.

We evaluate mapping strategies using two metrics: **inference throughput** and **real-time performance**. The inference throughput is measured as the number of people processed per second by the MMPose models, using the MS-COCO and CrowdPose datasets. We use 10% of the images as the tuning set to select optimal compute core assignment with our method, and the remaining 90% is used for throughput evaluation. As these datasets focus on pose estimation accuracy with relatively few human bodies per image, we randomly group 16 images per iteration (without repetition) to create a broader distribution of input sizes. The total number of human bodies across the 16 images represent one input size of an iteration. As a result, there are 6,819 iterations for the MS-COCO dataset (Avg: 34.95 people, SD: 16.38) and 658 iterations for CrowdPose (Avg: 67.01 people, SD: 14.21). The input size distributions of the two datasets enable us to assess performance under both low (i.e., MS-COCO) and high (i.e., CrowdPose) input size conditions.

We assess the real-time performance by setting an execution time constraint of 30 frames per second (FPS), considering that most inputs for downstream tasks in object detection are video feeds. Under this constraint, we compare the maximum number of people the MMPose models can process while ensuring real-time performance.

4.2 Evaluation of Inference Throughput

This section compares the inference throughput of our mapping algorithm against two baseline mapping strategies across different model backbones. **Baseline 1** follows the mapping approach used in Google EdgeTPU, where a single instance of the MMPose model with a batch size of "one" is created, and all available cores are allocated to it (i.e., batch-one-core-all). Baseline 1 processes an arbitrary number of input samples sequentially, handling one sample at a time. **Baseline 2**, on the other hand, allocates only one core per model instance but increases parallelism by running multiple instances of the model, each on a separate core (i.e., batch-one-core-one). We evaluate the performance of three MMPose backbones: MobileNetV2, ResNet50, and ResNet101, all of which use YOLOv5m for person detection. Larger models, such as the ResNet backbones, provide higher accuracy but at the cost of longer inference times.

Figure 6 shows the performance comparison across the three mapping approaches. In this experiment, our algorithm derives the mapping {n1b1c1, n7b2c1} for MobileNet, which creates one model with a batch size of one and seven models with a batch size of two, each running on one core. For both ResNet50 and ResNet101, the optimal mapping is {n1b1c1, n4b2c1, n3b3c1}.

Compared to Baseline 1, Baseline 2 and our method achieve 2.33× and 3.05× higher inference throughput on average, respectively, due to enhanced parallelism and hardware utilization. This shows the importance of exploiting core-level parallelism on multicore AI inference accelerators, which are gaining widespread adoption. Running a single model with a batch size of one on all cores, as used in EdgeTPU, results in poor core utilization.

Additionally, our approach outperforms Baseline 2 with an average of 1.31× higher throughput, due to its ability to identify more

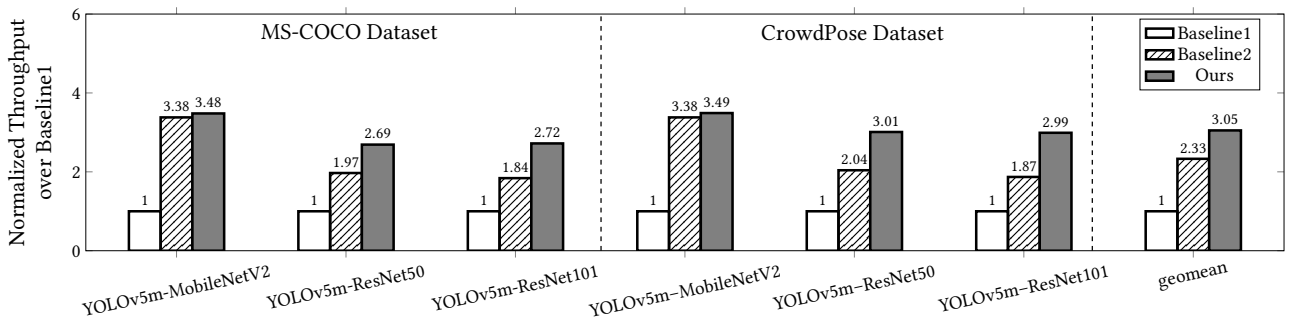

Figure 6: Throughput improvements over EdgeTPU strategy for MS-COCO and CrowdPose datasets with different backbones.

sophisticated compute core assignments. Notably, the benefits of our method are more pronounced with deeper architectures like ResNet50 and ResNet101, achieving an average throughput improvement of 1.41× and 1.53× over Baseline 2, respectively, compared to only a 3.2% improvement observed with MobileNetV2. This is because ResNet's deeper and more complex architecture has more computations that can be parallelized, leading to better utilization of compute cores when using larger batch sizes. Although Baseline 2's "batch-one-core-one" mapping is adequate, it fails to fully utilize compute cores for ResNet models. Combined with the cost model, our dynamic programming approach identifies better mappings of batch sizes to compute cores, resulting in enhanced throughput.

Furthermore, we observe that our approach achieves greater inference throughput on CrowdPose than MS-COCO. This is due to the higher average number of people per iteration in the CrowdPose dataset. In MS-COCO, the smaller crowd sizes in input data sometimes lead to under-utilization of compute cores. In contrast, the denser scenes in CrowdPose provide more workloads, allowing our approach to efficiently distribute data across the handling models and achieve better core utilization.

Overall, our approach achieves an average of 3.05× throughput improvement over the EdgeTPU mapping strategy (Baseline 1) in multi-person pose estimation across different model configurations. Additionally, our method shows an average of 1.31× higher throughput than Baseline 2, which uses a straightforward mapping and parallelization approach. These results demonstrate the effectiveness of our dynamic programming algorithm in optimizing compute core assignments and improving throughput.

4.3 Evaluation of Real-Time Performance

In this experiment, we evaluate two YOLOv5 model sizes (YOLOv5m and YOLOv5l) along with three MMPose backbones: MobileNetV2, ResNet50, and ResNet101. To meet the 30-FPS real-time constraint, YOLOv5m requires one compute core on the Qualcomm accelerator, while YOLOv5l which offers improved accuracy needs two compute cores due to its larger model size.

Figure 7 illustrates the maximum number of input samples that the MMPose models can process while adhering to the real-time constraint. On average, our method can handle 3.41× more input samples than Baseline 1 and 1.39× more than Baseline 2 across various model configurations. These results further emphasize the effectiveness of our compute core assignment strategy in real-time scenarios.

4.4 Validation of Execution Time Predictions

Our scheduling of parallel model execution relies on an accurate cost model. The cost model is built by compiling the models with varying batch sizes and compute core allocations, and then measuring the inference time of each model. Since each model is profiled individually, it is necessary to verify whether we can accurately estimate the execution time when multiple models are run at the same time.

Figure 8 shows the comparison between the actual inference time and our predictions of the YOLOv5m-MobileNetV2 model with different number of input samples. We observe that the inference time result resembles a step function. The reason is that the inference time is determined by the longest execution time among the parallel models (e.g., the first model in Figure 5). Increasing the number of input samples can take the same time if a model can handle the additional data without extending the execution time of the slowest model. The result in Figure 8 indicates a relative error of 2.7%. Across different model backbones, the relative error remains within 3.8%, demonstrating the robustness and reliability of our prediction.

5 Conclusion and Future Work

AI inference accelerators, like the Qualcomm Cloud AI 100, are specialized for static inference to achieve maximum performance. However, this creates challenges for accelerating AI applications with a dynamic number of input samples. In this work, we propose a novel approach that dynamically partitions input samples, optimally assigns compute cores, and efficiently schedules model execution using dynamic programming to accelerate dynamic batch inference. Additionally, we present a systematic approach to reduce the search space using insights from performance profiles. Experimental results demonstrate significant improvement in inferencing multi-person pose estimation applications, reaching 3.05× higher throughputs on benchmark datasets and 3.41× more input samples under real-time constraints, compared to the EdgeTPU inference strategy.

The future plan includes optimizing compute core assignment for dynamic shape inference such as graph neural networks, and models with dynamic execution flow. Additionally, our proposed method shows potential for applications on emerging accelerators, e.g., compute-in-memory chips, which also feature multiple processing units.

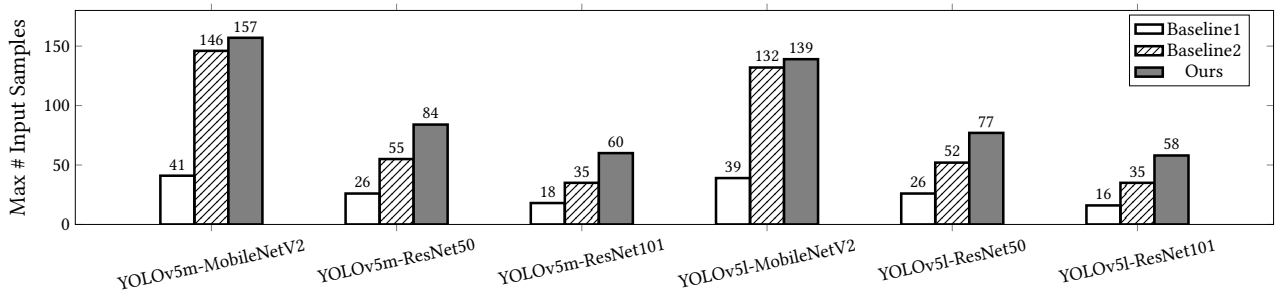

Figure 7: Maximum number of input samples MMPose can handle under real-time constraint with different backbones.

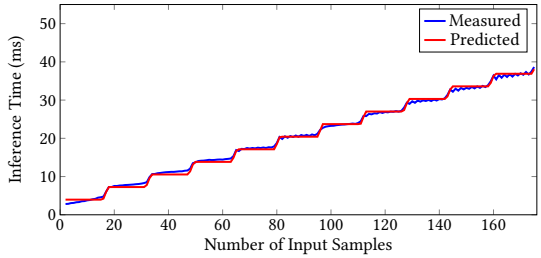

Figure 8: Comparison between predicted and measured inference time for different number of input samples.

References

[1] Inc. Advanced Micro Devices. 2023. AMD Alveo V70 AI Accelerator. https://www.xilinx.com/applications/data-center/v70.html.

[2] Amazon. 2023. Inferentia. aws.amazon.com/machine-learning/inferentia/.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.* 43, 6 (jun 2008), 101–113.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1877–1901.

[5] Xuyi Cai, Ying Wang, and Lei Zhang. 2022. Optimus: An Operator Fusion Framework for Deep Neural Networks. *ACM Trans. Embed. Comput. Syst.* 22, 1, Article 1 (oct 2022), 26 pages.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX OSDI*. 578–594.

[7] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[8] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058* (2018).

[9] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, and et al. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *International Symposium on Computer Architecture (ISCA '23)*. Article 80.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. arXiv:1712.00409

[12] Intel. 2017. Neural Compute Stick. https://software.intel.com/content/www/us/en/develop/articles/intel-movidius-neural-compute-stick.html

[13] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Annual International Symposium on Computer Architecture*. 1–12.

[14] Jiefeng Li, Can Wang, Hao Zhu, Yihuan Mao, Hao-Shu Fang, and Cewu Lu. 2018. CrowdPose: Efficient Crowded Scenes Pose Estimation and A New Benchmark. *arXiv preprint arXiv:1812.00324* (2018).

[15] Tsung-Yi Lin, Michael M. Yeh, Scott Antol, David B. J. Poole, Ross Girshick, and Piotr Dollaŕ. 2014. Microsoft COCO: Common Objects in Context. In *European Conference on Computer Vision (ECCV)*.

[16] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *International Symposium on Field-Programmable Gate Arrays*. 45–54.

[17] Eitan Medina and Eran Dagan. 2020. Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor. *IEEE Micro* 40, 2 (2020), 17–24.

[18] Microsoft. 2023. Batch Inference. https://github.com/microsoft/batch-inference.

[19] AWS Neuron. 2024. NeuronCore Batching. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-features/neuroncore-batching.html.

[20] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Programming Language Design and Implementation*. 883–898.

[21] Nvidia. 2024. Concurrency and Dynamic Batching on Jetson. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/examples/jetson/concurrency_and_dynamic_batching/README.html.

[22] Qualcomm. 2023. Qualcomm® Cloud AI 100 (Edge Version). https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qualcomm_cloud_ai_100_announcment_deck_.pdf.

[23] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis With Latent Diffusion Models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10684–10695.

[24] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[25] Arindam Sengupta, Feng Jin, Renyuan Zhang, and Siyang Cao. 2020. mm-Pose: Real-Time Human Skeletal Posture Estimation Using mmWave Radars and CNNs. *IEEE Sensors Journal* 20, 17 (2020), 10032–10044.

[26] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. 2022. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. In *IEEE IISWC*. 79–91.

[27] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Association for Computational Linguistics (ACL)*. 1556–1566.

[28] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *23rd International Conference on Pattern Recognition (ICPR)*. 2464–2469.

[29] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971

[30] Ultralytics. 2021. YOLOv5: A state-of-the-art real-time object detection system. https://docs.ultralytics.com.

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[32] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, et al. 2021. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *International Symposium on Computer Architecture*. 153–166.

[33] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. 2021. Deep High-Resolution Representation Learning for Visual Recognition. *Transactions on Pattern Analysis and Machine Intelligence* 43, 10 (2021), 3349–3364.

[34] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. In *ACM PACT*. 31–42.

[35] Ziyang Zhang, Yang Zhao, Huan Li, and Jie Liu. 2024. BCEdge: SLO-Aware DNN Inference Services with Adaptive Batch-Concurrent Scheduling on Edge Devices. *IEEE Transactions on Network and Service Management* (2024), 1–1.