Memory-aware Hierarchical Scheduling of ML Tasks in Heterogeneous Clusters of Edge Computing Devices with GPUs

Saumya Mathkar[*] Pinki Pinki[*] Vinayak Naik[*#]
[*]*Department of CSIS* [#]*Department of EEE*
BITS Pilani, Goa, India
{p20210017, 2022proj011, vinayak}@goa.bits-pilani.ac.in

***Abstract*—Machine Learning (ML) workloads represent a major share of edge computing tasks, where efficient scheduling is critical to exploit heterogeneous edge devices' capabilities in terms of their GPUs. We find that conventional schedulers allocate resources opportunistically, focusing on availability rather than optimizing the overall execution time of ML jobs. This results in inefficient utilization of heterogeneous resources. An example is a heuristic-based strategy of First Come First Serve (FCFS), combined with Least Loaded First. In this approach, jobs are assigned in an FCFS manner and placed on the least loaded resourceful device. This allocation blocks subsequent larger tasks requiring high resources, thereby increasing the completion time of all tasks, known as the makespan [1]. To alleviate the problem, we propose MemBatch ILP, an Integer Linear Programming (ILP)-based scheduler that performs memory-aware batching to minimize the makespan of ML workloads while reducing the idling of GPUs. We employ a hierarchical scheduling technique – memory-aware batching at the cluster level and fine-grained task-splitting at the device level, enabling parallel execution and balanced utilization of heterogeneous resources. We compare our MemBatch ILP with a baseline ILP scheduler and FCFS-Least Loaded First on a testbed of heterogeneous Jetson edge devices. Our results show that MemBatch ILP reduces the makespan by up to 48% compared to FCFS-Least Loaded First and 61% compared to baseline ILP.**

***Index Terms*—Edge Computing**

I. Introduction

The rapid growth of ML applications on the network edge has reshaped the way computational workloads are deployed and managed. Applications such as real-time video analytics, predictive maintenance, and autonomous navigation require strict latency guaranties. Scheduling tasks to meet them is challenging because edge clusters are heterogeneous, with devices differing in GPU architectures, processing speeds, and available memory.

Ray [2] is a widely adopted framework for distributed ML workloads. It offers low-latency scheduling in heterogeneous and resource-constrained edge environments by allocating tasks across various devices in the cluster. Large platforms, such as Netflix, employ Ray to build heterogeneous CPU–GPU training clusters [3]. Despite its widespread use, Ray's default scheduler, referred to here as First Come First Serve–Least Loaded First, exhibits notable limitations. It schedules the job in an FCFS manner from the task side and places tasks on the least loaded devices on the resource side. The idea is that by leveraging the least loaded device, we will minimize the total execution time, which is the sum of every task's execution time. As a result, smaller jobs may occupy resourceful devices, preventing larger, resource-intensive tasks from being executed efficiently. This misallocation slows down subsequent workloads and increases makespan.

To address this, we propose MemBatch ILP, a memory-aware ILP-based scheduling algorithm that shifts the optimization focus from minimizing total execution time to minimizing the makespan. Minimizing total execution time focuses on reducing the sum of task durations across all devices in the cluster. In scheduling terms, when GPU cores work in parallel, the makespan corresponds to the maximum completion time, among all cores, of the one that takes the longest to finish its assigned tasks. Thus, minimizing makespan ensures a balanced workload distribution.

MemBatch ILP aims to optimally utilize every device in the cluster. It tries to ensure that all cores finish execution at approximately the same time, improving parallel execution and making better use of available computational resources. It uses memory-aware batching to speed up the calculation of the schedule. Instead of taking all tasks and running ILP on that, MemBatch ILP first groups the tasks into multiple batches at *cluster level*, based on the total memory available on all devices. Each batch contains as many tasks as fits within the cluster memory, that is, the sum of the memory of all devices in the cluster (e.g. 64 GB in our testbed). Our algorithm generates multiple batches of the subset of the task set T, which are then processed sequentially. By batching tasks, the ILP no longer needs to handle all tasks together, reducing the number of decision variables per optimization and lowering scheduler overhead.

The computational complexity of the ILP-based scheduler depends on the number of decision variables, which scales with the product of the number of tasks and the number of devices. For a workload of N tasks and D devices, the baseline ILP solves a problem with $N \times D$ decision variables, resulting in an approximate complexity of $\mathcal{O}\big((N \cdot D)^2\big)$. MemBatch ILP reduces this complexity by partitioning the N tasks into B batches ($B \leq N$), where each batch b contains T_b tasks such

We acknowledge BioCyTiH Foundation for sponsoring the project.

that $\sum_{b=1}^{B} T_b = N$. The complexity per batch is $\mathcal{O}\big((T_b \cdot D)^2\big)$. The leading to the overall complexity is

$$\sum_{b=1}^{B} \mathcal{O}\big((T_b \cdot D)^2\big) \ll \mathcal{O}\big((N \cdot D)^2\big), \quad \textit{when } T_b \ll N$$

Thus, dividing tasks into smaller batches significantly reduces the size of the ILP problem, making the approach tractable for large-scale scheduling within the memory constraints of the devices. We give each batch of tasks to the task splitting module [4], which divides oversized tasks into smaller sub-tasks that fit within the memory of the smallest device. Following this hierarchy of cluster-level batching and device-level task splitting ensures better utilization of GPU and memory resources while maintaining the novelty of the approach.

We compare MemBatch ILP with the following two techniques.

1) **Baseline ILP Scheduling** It minimizes the total computation time of all tasks executed on a cluster of edge devices. The algorithm [4] formulates the task assignment problem as a binary optimization problem, where each task can be assigned or left unassigned. Unlike makespan minimization, this formulation may yield imbalanced schedules in which some devices complete early while others remain heavily loaded, thereby increasing the overall makespan despite minimizing the overall computation time.
2) **FCFS-Least Loaded First Scheduling** It [2] sends tasks in arrival order, following an *FCFS* policy at the queue level. Once the tasks are ready for placement, the scheduler selects a set of top k devices. It first sorts the devices to prioritize those that already have tasks scheduled, helping to maintain the locality. Then, it favors devices with low resource utilization to improve load balancing. The variable k refers to the number of top devices that the scheduler considers when deciding where to schedule a task. The scheduler calculates k as the maximum between two values: a fraction of the total size of the cluster or a fixed minimum number of devices. From the top candidates k, the scheduler then selects a device at random. In general, the FCFS-Least Loaded First technique is characterized by the following two key components.
    a) Task Dispatch Order: Tasks are enqueued and dispatched in the order of arrival (*First Come First Serve*).
    b) Task Placement Strategy: When allocating a task, the following three-step procedure is applied.
        i) Locality: Prioritizes devices that already run related tasks.
        ii) Load Balancing: Selects devices with low resource utilization.
        iii) Randomization: Breaks ties by randomly selecting among the top k candidates.

    Although FCFS-Least Loaded First is efficient and simple, it does not account for heterogeneity between cores or detailed resource utilization. Consequently, it leads to a load imbalance, with some devices becoming heavily loaded, while others remain underutilized.

II. Related work

We divide related work into two subsections – (1) scheduling frameworks and (2) scheduling techniques. The former considers the scheduling software that one can use and the latter provides algorithms for scheduling.

A. Scheduling Frameworks

Scheduling frameworks are the backbone for task execution and scheduling in large-scale environments. These frameworks manage task execution and scheduling mechanisms to coordinate resources.

Hadoop and Spark are widely used frameworks for handling large data volumes. Spark performs better for real-time delay-sensitive applications than Hadoop, making it more suitable [5]. Hadoop supports scheduling such as FIFO, fair scheduling, and capacity policies using the external scheduler YARN, and Spark has its built-in task scheduler, which provides FIFO and fair scheduling. These are coarse-grained and assume abundant resources, making them unsuitable for heterogeneous edge clusters with limited resources. Their high overhead makes them a poor fit for resource-constrained, latency-sensitive edge environments.

Dean and Porter [6] compare distributed execution frameworks such as Spark, Flink, and their distinct trade-offs between scheduling granularity and execution performance. Spark is used for coarse-grained, data-parallel computations, benefiting from data locality. While the authors of [7] and [6] focused more on the locality of the data, this focus increases the total execution time and reduces the effectiveness of fine-grained workloads. Flink is more suitable for heterogeneous and resource-constrained edge environments. Both frameworks prioritize low-latency scheduling (low scheduler time) and fine-grained task execution. However, this leads to longer overall job completion times or longer overall execution time. These trade-offs highlight the need for adaptive scheduling mechanisms to balance overall computation time and workload granularity in heterogeneous environments. This also tells us that there is a need for a scheduler that can balance the scheduler time and the overall computation time of the job.

Shi et al. [8] introduced a GPU-aware batching approach for energy-constrained mobile devices. By partitioning workloads and aggregating similar sub-tasks, their method reduces GPU energy consumption with energy efficiency as the primary objective. Although effective in lowering power consumption, this approach focuses on energy minimization, making it less suitable for performance-critical deployments. Cang et al. [9] proposed a joint batching and scheduling framework that optimizes the number of batches, batch initiation times, and task-batch assignments. This approach outperforms traditional sequential batching and accounts for heterogeneous task requirements and device capacities, offering greater adaptability in diverse computing environments.

Liu et al. [10] grouped tasks into batches based on feature vectors, allowing the model to reuse its weights across tasks. The batched tensor layout reduces memory-access frequency and bandwidth demand. Their approach combines batching with an early-exiting algorithm to shrink batch size across layers, freeing memory and improving throughput while preventing out-of-memory failures. Zhang et al. [1] groups subtasks of DNN that work on the same data to leverage data parallelism. Our work groups tasks from different ML tasks acting on different data to fit the GPU memory capacity of each edge computing device which is different from grouping to increase data parallelism.

B. Scheduling Techniques

Classical scheduling algorithms include First-Come-First-Serve, Shortest Job First, Round Robin, and Priority-based scheduling [11]. Among them, the Shortest Job First reduces waiting time compared to First-Come-First-Serve and Round Robin, while the Highest Response Ratio Next [12] outperforms Round Robin [13]. Round Robin is widely used in time-sharing systems, whereas embedded systems adopt real-time schedulers such as Earliest Deadline First and Least Slack Time [13]. However, these methods remain resource-agnostic and are less effective in heterogeneous environments.

Recently, many researchers have focused on integrating scheduling with various batching techniques, as batching enables grouping tasks to reduce scheduling overhead, improve throughput, and exploit resource sharing. In contrast, scheduling ensures timely and efficient allocation across heterogeneous devices. Early batching techniques typically rely on fixed-size or execution-time-aware batching [14]. However, these methods are inefficient in dynamic workload environments where task sizes, execution times, and resource requirements vary significantly. In the HPC domain, Zrigui et al. [15] proposed a batch scheduling approach that classifies jobs as 'small' or 'large', and prioritizes small jobs to reduce the average bounded slowdown. Although effective in queue management and throughput improvement, this approach remains batch-queue-oriented and does not address fine-grained task-to-device scheduling in heterogeneous clusters.

Gonthier et al. [7] addressed the locality of the data for GPU scheduling. They proposed task ordering strategies to reduce data movements and improve execution time. Li et al. [16] proposed thread batching mechanisms consisting of two parts, thread-enabled batch memory partitioning and thread batch-aware scheduling to improve locality and parallelism in GPU systems. Although the strategies mentioned in [7] and [16] work well for workloads with reusable data, they may not work well for highly diverse tasks and GPUs. Mary and Alyosius [17] focus on mobility and execution time–aware task offloading in mobile cloud computing. Since executing tasks directly on mobile devices quickly drains battery and memory, tasks are instead offloaded to cloud resources. Their model decides whether to migrate a task based on its remaining execution time and device mobility. However, the approach focuses on managing tasks in different cloudlets; the performance is highly dependent on cloudlet availability and connectivity, making the model less effective in regions with limited infrastructure. To address device diversity, studies mentioned in [4] minimize the total computation time between devices. While this reduces cluster execution time, it can overload faster devices, leaving slower ones underutilized, thereby increasing delays since the slowest device determines completion.

Research Gaps

- Distributed computing frameworks such as Spark, Hadoop, Ray, and Flink employ locality-aware strategies like FIFO, FCFS, and fair scheduling; however, these approaches overlook execution time, optimality, and device heterogeneity factors critical in resource-constrained heterogeneous environments.
- Most batching approaches are queue-centric or data-centric, largely ignoring device heterogeneity. As a result, they struggle to adapt to dynamic workloads and optimally match tasks with devices in edge clusters.
- Existing studies on edge computing focus on energy efficiency or memory optimization, while critical objectives such as minimizing overall time with heterogeneity-aware task-device allocation remain underexplored.

To overcome these gaps, we propose MemBatch ILP, which combines memory-aware batching, task splitting, and ILP-based scheduling for heterogeneous edge clusters. Tasks are batched based on per-task profiles and device memory availability; oversized tasks are split into memory-compliant subtasks for parallel execution. An ILP-based scheduler then assigns tasks to devices optimally, minimizing the makespan of the cluster while respecting memory and compute constraints. Unlike heuristic methods, MemBatch ILP ensures balanced task allocation and achieves a lower makespan even on resource-constrained devices such as Jetson Nanos.

III. Methodology

We formulate the problem, define the metrics used to evaluate the effectiveness of the solution, and mention the design of the proposed MemBatch ILP technique.

A. Problem Statement

We consider the problem of scheduling a set of ML tasks T in a heterogeneous edge cluster composed of devices with varying GPU architectures, processing speeds, and memory capacities. Each task has been profiled for the requirements for GPU utilization, memory footprint, and execution time on all categories of edge devices.

The goal of scheduling is to assign tasks to devices so that the makespan, the time from the start of execution until the last task finishes, is minimized. This differs from baseline ILP, which minimized only the total execution time but did not explicitly account for the balance of workload between devices. Minimizing makespan ensures that all devices in the cluster finish execution at the same time, thus improving parallel execution and resource utilization.

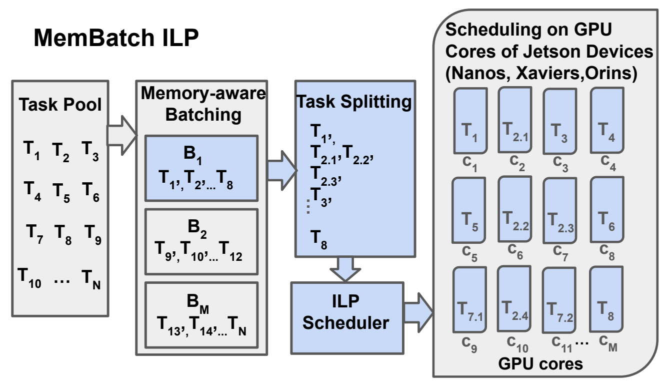


Fig. 1: Workflow of MemBatch ILP. The memory-aware batching module groups the tasks into batches without exceeding cluster memory. The task splitting module splits oversized tasks if needed. The ILP scheduler assigns batches of tasks to Jetson GPU cores by minimizing the makespan.

B. Evaluation Metrics

We evaluate the solutions using the following metrics.

- Makespan: The total time elapsed from the beginning of the first task to the completion of the last task. A lower makespan indicates better parallelism and load balancing. We assess scalability by examining how the makespan varies under different task volumes in multi-device clusters.
- GPU Utilization: The maximum percentage of GPU capacity, among all cores, from the beginning to the completion of task execution.
- Scheduler Runtime: The time required by the scheduling algorithm to assign a batch of tasks to the devices in the cluster. As ILP is an NP-complete problem, we will measure the time it takes to schedule the tasks compared to the tasks' execution.

C. Proposed Solution: MemBatch ILP

Our proposed solution, MemBatch ILP, schedules a set of tasks T, each with profiled GPU and memory requirements, in a heterogeneous cluster. The algorithm proceeds in three iterative stages until all tasks are completed, which are memory-aware batching, task splitting, and ILP-based scheduling, as shown in Figure 1.

1) **Memory-aware batching** The first stage, memory-aware batching, forms groups of tasks such that the combined memory footprint of each batch does not exceed the total memory of the cluster. Algorithm 1 begins with all tasks marked as unscheduled and repeatedly forms batches in FCFS order. Tasks are incrementally added to a batch until the aggregate footprint approaches the memory limit of the cluster. This is repeated, forming multiple batches until all tasks are put in batches. We give these batches to the task splitting and ILP scheduling module sequentially. We apply task splitting, if necessary, to fit oversized tasks onto available devices. After the task splitting module creates batches, the ILP scheduler minimizes the makespan for each while considering memory, GPU, and parallelism constraints, using task-device profiling data. Scheduled tasks are then removed from the unscheduled list before the next batch is processed. The scheduling will continue until all tasks are scheduled. In addition to optimality, batching reduces the overhead of the ILP formulation by limiting the number of tasks considered in each optimization round.

2) **Task splitting**
The second stage addresses tasks whose resource requirements exceed the available memory capacity of individual devices. The task splitting algorithm partitions tasks that exceed a predefined GPU memory threshold into smaller sub-tasks. These sub-tasks are executed in parallel across multiple GPU cores, ensuring parallelism

Algorithm 1 Memory-Aware Batching

```
Input:
    $T$ = Set of all tasks
    $M[i]$ = Memory required by task $i$
    $D$ = Set of all devices in the cluster
    $Mem\_D[j]$ = Available memory on device $j$
Initialize: $Unscheduled\_Tasks \leftarrow T$
$Batches \leftarrow \emptyset$
$total\_cluster\_memory \leftarrow \sum_{j \in D} Mem\_D[j]$
$current\_batch \leftarrow \emptyset$
$current\_batch\_memory \leftarrow 0$
for each task $i$ in $T$ do
    if $current\_batch\_memory + M[i] \leq total\_cluster\_memory$ then
        Add $i$ to $current\_batch$
        $current\_batch\_memory \leftarrow current\_batch\_memory + M[i]$
    else
        Add $current\_batch$ to $Batches$
        $current\_batch \leftarrow \{i\}$
        $current\_batch\_memory \leftarrow M[i]$
    end if
end for
if $current\_batch \neq \emptyset$ then
    Add $current\_batch$ to $Batches$
end if
for each $Batch$ in $Batches$ do
    $Remaining\_Tasks \leftarrow Batch$
    while $Remaining\_Tasks \neq \emptyset$ do
        Run Task Splitting on $Remaining\_Tasks$
        Run ILP Scheduler on $Remaining\_Tasks$
        Remove scheduled tasks from $Remaining\_Tasks$
    end while
end for
```

Algorithm 2 MemBatch ILP Scheduling to Minimize Makespan with Penalty for Unassigned Tasks

```
1: Input:
     • $T = \{T_1, T_2, \ldots, T_N\}$: Set of ML tasks
     • $C = \{C_1, C_2, \ldots, C_M\}$: Set of GPU cores
     • $Mem\_C[j]$: Available memory on core $C_j$
     • $Comp[i][j]$: Computation time of task $T_i$ on core $C_j$
     • $Mem[i][j]$: Memory required for task $T_i$ on core $C_j$
     • $CoreReq[i][j]$: Percentage of GPU capacity of core $C_j$ required by task $T_i$
     • $CoreCap[j]$: Total available GPU capacity of core $C_j$ (in % or core units)
2: Output:
     • A set of task-to-core assignments $(C_j, T_i)$
     • A list of unassigned tasks (if any)
3: ILP Formulation:
     • Binary decision variable $x_{ij} \in \{0,1\}$: $x_{ij} = 1$ if task $T_i$ is assigned to core $C_j$
     • Binary variable $p_i \in \{0,1\}$: $p_i = 1$ if task $T_i$ is unassigned
     • Continuous variable $makespan \geq 0$
4: Objective: Minimize makespan
5: Constraints
     • Task Scheduling Constraint: Each task is either assigned to exactly one core or marked unassigned: $\sum_{j=1}^{M} x_{ij} + p_i = 1, \quad \forall i \in \{1,\ldots,N\}$
     • Memory Usage Constraint: Memory usage for each GPU core should not exceed the limit: $\sum_{i=1}^{N} Mem[i][j] \cdot x_{ij} \leq Mem_C[j], \quad \forall j \in \{1,\ldots,M\}$
     • GPU Core Constraint: GPU usage for each core should not exceed the limit: $\sum_{i=1}^{N} CoreReq[i][j] \cdot x_{ij} \leq CoreCap[j], \quad \forall j \in \{1,\ldots,M\}$
6: Solve the ILP formulation using a ILP solver:
7:   if There are unassigned tasks ($x_{ij} = 0$ for some $i$) then
8:     Update the task list $T$ with the unassigned tasks
9:     Return to solve the updated ILP problem
10:  else
11:    Return the list of tasks assigned to cores
12:  end if
13: End when all tasks are assigned
```

under resource constraints. By decomposing oversized tasks, the algorithm not only avoids memory violations but also improves parallelism, as parallel execution of smaller sub-tasks on multiple cores outperforms processing of a single large task on one core. After splitting a task into sub-tasks, all tasks and sub-tasks from the batch are given to the ILP scheduler, assigning each a unique ID.

3) **MemBatch ILP scheduling** In the final stage, each batch of tasks is scheduled using an ILP formulation in Algorithm 2 to minimize the makespan. Since multiple devices in the cluster execute groups of tasks in parallel, effective parallelism is crucial for overall performance. An unbalanced workload—where one device is overloaded while others remain underutilized directly increases the makespan and degrades system efficiency. After task decomposition through the task splitting module, the objective function minimizes the makespan. This design ensures that the scheduling solution remains both time-efficient and resource-complete. The scheduling is subject to the following constraints.

a) Task: Each task must be assigned once or marked as unassigned.
b) Memory: The total memory requirements of all tasks must not exceed the available memory capacity.
c) GPU: The total GPU requirements of all tasks assigned to a core must not exceed the GPU capacity available to that core.

IV. Experimental Setup

In this section, we describe the experimental setup used to evaluate the proposed scheduler, MemBatch ILP, compared to the baseline ILP and FCFS-Least Loaded First. We first present the cluster configuration and task set, followed by a description of the experiments conducted. ‘

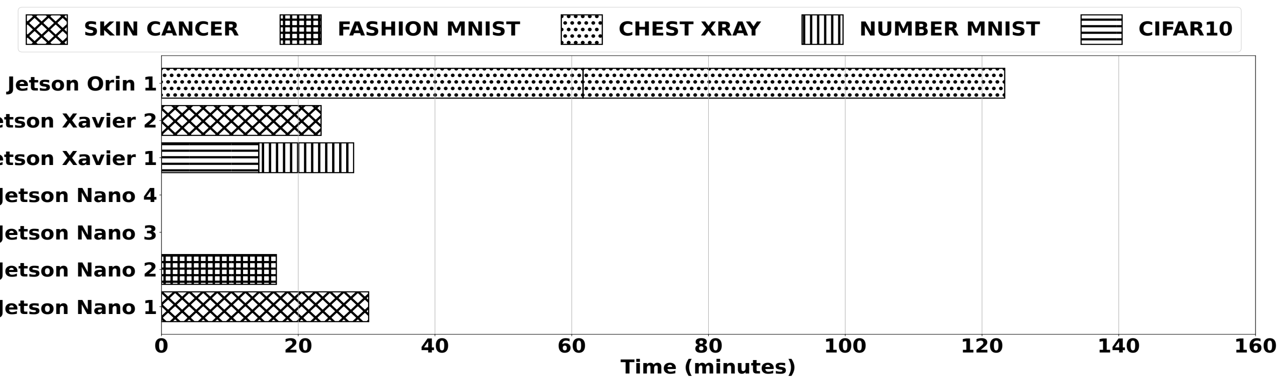
A. Cluster Configuration

To evaluate the effectiveness of scheduling strategies in heterogeneous edge environments, we deploy MemBatch ILP on a real-world cluster comprising NVIDIA Jetson devices with varying memory and GPU capabilities. The cluster configuration is summarized in Table I.

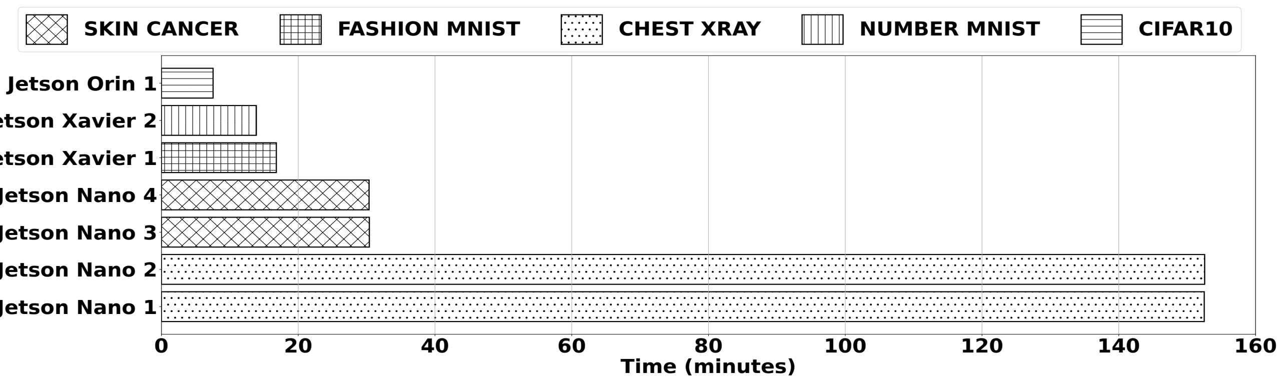
TABLE I: Edge Cluster Configuration

Device Type	**Units**	**# of CUDA Cores Per Unit**	**RAM (GB)**
Jetson Nano	4	128	4
Jetson Xavier NX	2	384	16
Jetson Orin NX	2	1024	16

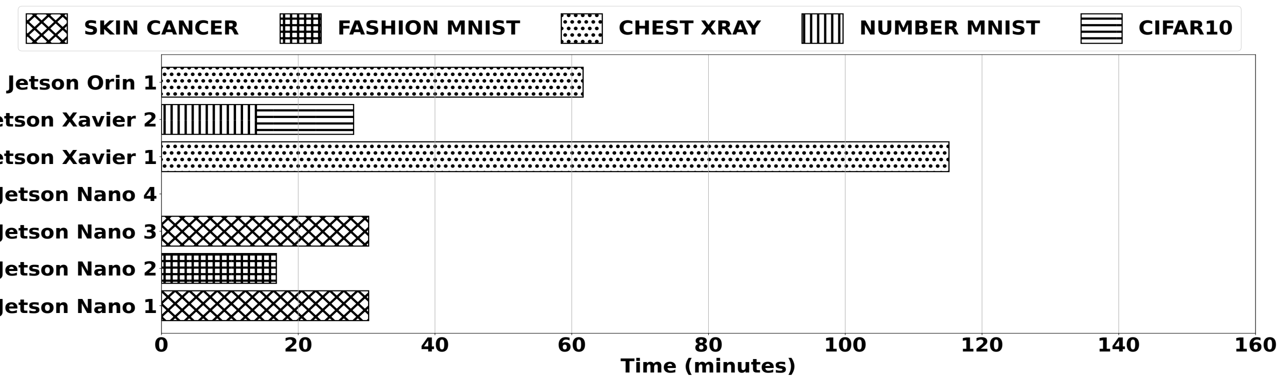
The cluster comprises multiple worker devices and a single head device. The head device, a Jetson Orin NX, provides high GPU performance and large memory capacity, enabling the execution of the ILP MemBatch scheduler. The worker devices include four Jetson Nanos, two Jetson Xavier NX units, and one Jetson Orin NX. The Jetson Nanos, with limited memory and modest GPU resources, are low-cost, entry-level devices that represent weaker edge devices, adding complexity to scheduling. The Xavier NX devices represent mid-tier performance, while the Orin NX provides high computational capacity. This heterogeneous mix offers a realistic and challenging testbed for evaluating scheduling strategies across devices of varying capabilities.



(a) Baseline ILP



(b) FCFS-Least Loaded First



(c) MemBatch ILP

Fig. 2: Makespan comparison of three schedulers in a heterogeneous Jetson cluster. (a) Baseline ILP assigns both Chest X-ray sub-tasks to Orin, resulting in one waiting and a makespan of 123 minutes. (b) FCFS-Least Loaded First maps both sub-tasks to Nanos by treating GPUs as identical, yielding the longest makespan of 155 minutes. (c) MemBatch ILP exploits device heterogeneity by distributing sub-tasks across Orin and Xavier, achieving the shortest makespan of 115 minutes.

B. Task Set

We evaluate five representative workloads drawn from publicly available datasets Cifar 10 (190 MB) [18], Fashion Mnist (83 MB) [19], Number Mnist (60 MB) [20], Chest X-ray (4.8 GB) [21], and Skin Cancer detection (6 GB) [22]. These datasets vary in image size, number of classes, and task complexity, ranging from lightweight classification to computationally demanding medical image analysis. We classify these datasets into lightweight and heavy. Lightweight workloads refer to small datasets of size $\leq 500\,$MB with low memory and GPU demand, whereas heavy workloads involve large datasets of size $> 500\,$MB, typically in GBs, that require substantial GPU memory and RAM. Each workload involves dataset loading, preprocessing, and model training using PyTorch. Smaller datasets (Number-MNIST, Fashion-MNIST, and CIFAR-10) demand relatively low memory requirements but exhibit different GPU utilization patterns. They form a lightweight load. In contrast, large datasets of Chest X-ray and Skin Cancer Detection demand substantial GPU memory and RAM, making them challenging to place on resource-constrained devices. They form a heavy workload. This mix of workloads ensures that the evaluation reflects all scenarios.

In ILP-based scheduling, each task is profiled to determine the following parameters.

- Execution time per device: Estimated runtime of each task across devices.
- GPU usage: Percentage of GPU capacity required per task on each device (e.g., 90% on Jetson Nano and 60% on Jetson Orin).
- Memory usage: Size of memory in GB for each task.

When a new task arrives without any prior profiling data, the profiler switches to a default configuration mode. In this mode, the scheduler assigns the averaged profile values derived from the existing profiled tasks.

C. Experiments

We perform experiments on a heterogeneous Jetson cluster described in Table I. To evaluate performance under varying system loads, tasks are submitted in volumes ranging from five to a hundred, covering both lightweight- and heavy-load scenarios. Each experiment is repeated ten times, and we report the average values to account for the variance caused by the execution of the stochastic task. The proposed MemBatch ILP scheduler is implemented in Python, with the scheduler's runtime measured separately to quantify scheduling overhead. All workloads are trained using PyTorch with CUDA acceleration, ensuring that tasks effectively exploit GPU resources across the cluster.

V. Results

We evaluate the proposed solution on the metrics of makespan, GPU utilization, and scheduler runtime.

A. Makespan

The makespan results for the three schedulers are summarized in Figure 2. For the baseline ILP 2a, both sub-tasks of the Chest X-ray workload are assigned to Jetson Orin. Although the first sub-task consumes 64% of Orin's GPU capacity, the second sub-task is placed on the same device. This behavior arises because the baseline ILP minimizes the total computation time across the cluster without considering device-level parallelism, resulting in a makespan of 123 minutes. The FCFS-Least Loaded First scheduler (see Figure 2b) assigns the heaviest workload of Chest X-ray, to Jetson Nanos, treating all GPUs as identical. Consequently, the slowest devices execute the heaviest workload, leading to the longest makespan of 155 minutes. Its earliest-job-first technique causes heavy tasks to wait on slower devices once faster ones are occupied.

In contrast, MemBatch ILP (see Figure 2c) accounts for device-specific characteristics, including GPU speed, memory capacity, and task execution time. It distributes the Chest X-ray sub-tasks across the fastest devices, Jetson Orin and Jetson Xavier, even though the task appears last in the queue. Orin completes its sub-task in approximately 70 minutes, while Xavier finishes in 115 minutes. The MemBatch ILP scheduler achieves the shortest makespan of 115 minutes, outperforming FCFS-Least Loaded First (155 minutes), and baseline ILP (123 minutes). Taking into account the heterogeneity of the device and the task requirements, it avoids overloads and ensures efficient task placement. In contrast, FCFS-Least Loaded First favors lightly utilized devices for earlier tasks, causing later heavy tasks to experience resource contention. The total computation times of all tasks were 221 minutes, 282 minutes, and 404.04 minutes for the baseline ILP, MemBatch ILP, and FCFS-Least Loaded First. The baseline ILP focuses on minimizing the total computation time, which limits parallelism and prolongs the makespan.

To evaluate performance under increasing load, we gradually increase the number of tasks from five to hundred and measure the resulting makespan, as shown in Figure 4. The baseline ILP scales poorly, reaching 64 hours for a hundred tasks due to limited parallelism. FCFS-Least Loaded First assigns long-running tasks to slower devices, causing delays and under-utilization of faster devices, with a makespan of 48 hours. MemBatch ILP consistently achieves the lowest makespan of 25 hours, a 61% improvement over the baseline ILP and 48% over FCFS-Least Loaded First, demonstrating its effectiveness in balancing task allocation and leveraging device heterogeneity as workload increases.

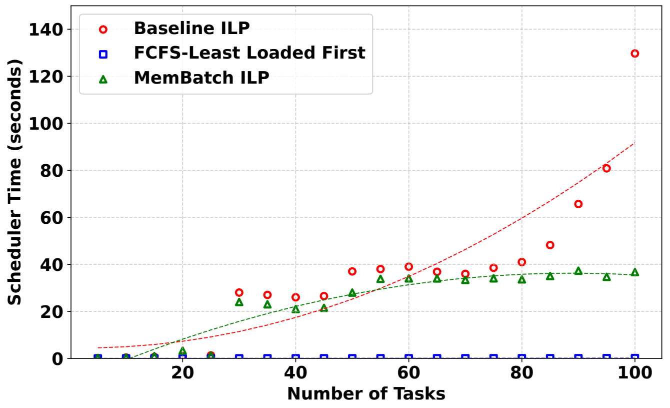


Fig. 3: Scheduler runtime comparison across varying task counts. The baseline ILP runtime grows rapidly beyond eighty tasks, FCFS-Least Loaded First maintains millisecond-level performance, and MemBatch ILP keeps bounded scheduling time below thirty-eight seconds through memory-aware batching. Since FCFS-Least Loaded First use FCFS, it has the least time complexity. However, the tasks take far more time to complete, thereby eroding the time saved in the long run.

B. GPU Utilization

We show the usage of GPU in Jetson devices for the three schedulers, highlighting how each technique uses heterogeneous GPU resources during the execution of tasks, in

Figure 5. The FCFS–Least Loaded First strategy achieves the highest GPU utilization, since it aggressively assigns tasks to available devices without considering task–device compatibility. This allocation leads to uneven scheduling and the longest makespan, as many small-capacity devices become overloaded with heavy tasks, while high-capacity devices get light tasks. In contrast, the baseline ILP scheduler shows the lowest GPU utilization because it concentrates more tasks on the fastest devices to minimize the total execution time, leaving slower devices underutilized. This imbalance reduces overall GPU usage across the cluster, even though the makespan is shorter than FCFS-Least Loaded First. MemBatch ILP, on the other hand, achieves intermediate GPU utilization by evenly distributing tasks between devices. This balanced allocation reduces the makespan, avoids overloading the devices, and provides an improved makespan compared to FCFS-Least Loaded First and the ILP baseline.

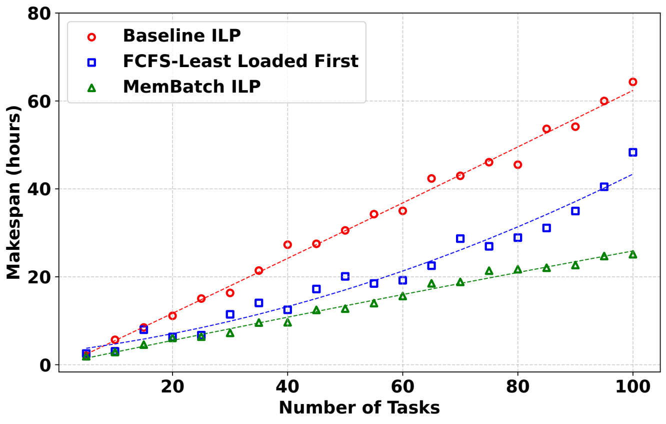


Fig. 4: Makespan comparison under three schedulers as task count increases. (a) Baseline ILP reaches 64 hours, (b) FCFS-Least Loaded First assigns heavy tasks to slower devices, resulting in 48 hours, and (c) MemBatch ILP achieves the shortest makespan of 25 hours, representing improvements of 61% over the baseline ILP and 48% over FCFS-Least Loaded First for a hundred tasks.

C. Scheduler Runtime

Figure 3 presents the time that each scheduler requires to assign tasks to devices, reflecting the computational overhead of the scheduling algorithm. Baseline ILP and MemBatch ILP solve an optimization problem to minimize makespan while considering device heterogeneity, memory constraints, and GPU usage. As a result, their runtime is higher than that of FCFS-Least Loaded First, which uses a simple earliest-job-first heuristic. For up to eighty tasks, the baseline ILP and MemBatch ILP perform similarly; beyond this, the baseline ILP's scheduler time increases drastically. For a hundred tasks, the baseline ILP requires 130 seconds, MemBatch ILP 36.7 seconds, and FCFS-Least Loaded First 0.1089 seconds. MemBatch ILP's memory-aware batching ensures that scheduler time remains constant even for a hundred tasks. When the requirement is fast scheduling irrespective of makespan, FCFS-Least Loaded First is the best choice. However, when optimality and minimization of execution time are the main objectives, MemBatch ILP proves to be the most effective approach.

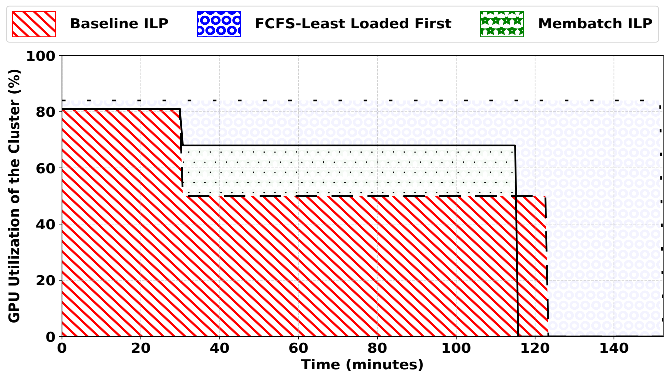


Fig. 5: GPU utilization across Jetson devices for the three schedulers. FCFS–Least Loaded First shows the highest utilization by aggressively assigning tasks. However, insufficient task–device matching leads to overloading the GPUs and the longest makespan. The baseline ILP yields the lowest GPU utilization by allocating more tasks on the fastest devices, leaving slower ones idle, and suboptimal makespan. MemBatch ILP shows a balance, distributing tasks evenly to improve utilization and reduce makespan compared to FCFS and the baseline ILP.

VI. Conclusion and Future Work

Our work presented MemBatch ILP, an ILP-based distributed scheduler designed for heterogeneous edge clusters that execute ML workloads. By combining memory-aware batching with a makespan minimization objective, MemBatch ILP achieves improved workload balance and efficient resource utilization for parallel execution. Our exhaustive experimental evaluation on a Jetson-based edge testbed demonstrates that MemBatch ILP reduces the makespan by up to 48% compared to FCFS-Least Loaded First and 61% compared to baseline ILP while maintaining high utilization across devices and ensuring comparable execution times on all devices.

Despite these gains, the current scheduler treats CPU and GPU resources independently, requiring explicit task-to-resource mapping. Future work will focus on developing an automatic task characterization mechanism to distinguish CPU- and GPU-intensive workloads and allocate them accordingly. We plan to explore Reinforcement Learning (RL)-based scheduling strategies that can predict task profiles for unseen workloads and adaptively decide CPU–GPU allocation, further improving efficiency and generalization in dynamic edge environments.

References

[1] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang, "Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 366–379.

[2] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, 2018, pp. 561–577.

[3] S. Goyal, "Heterogeneous training cluster with ray at netflix," Anyscale Blog, 2023, accessed: August 22, 2025.

[4] S. Mathkar, S. Aiyer, Y. Bapat, P. Pinki, A. K. Paul, and V. Naik, "Scheduling big machine learning tasks on clusters of heterogeneous edge devices," in *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*. Bengaluru, India: IEEE, Jan. 2025, pp. 439–447.

[5] N. M. Mirza, A. Ali, and M. K. Ishak, "The scheduling techniques in the hadoop and spark of smart cities environment: a systematic review," *Bulletin of Electrical Engineering and Informatics*, vol. 13, no. 1, pp. 453–464, Feb 2024.

[6] P. Dean and B. Porter, "Emergent scheduling of distributed execution frameworks," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, Jun 2019, pp. 240–242.

[7] M. Gonthier, L. Marchal, and S. Thibault, "Locality-aware scheduling of independent tasks for runtime systems," in *European Conference on Parallel Processing*. Cham: Springer International Publishing, Aug. 2021, pp. 5–16.

[8] W. Shi, S. Zhou, Z. Niu, M. Jiang, and L. Geng, "Multiuser co-inference with batch processing capable edge server," *IEEE Transactions on Wireless Communications*, vol. 22, no. 1, pp. 286–300, Jul 2022.

[9] Y. Cang, M. Chen, and K. Huang, "Joint batching and scheduling for high-throughput multiuser edge ai with asynchronous task arrivals," *IEEE Transactions on Wireless Communications*, vol. 23, no. 10, pp. 13 782–13 795, May 2024.

[10] Z. Liu, Q. Lan, and K. Huang, "Resource allocation for multiuser edge inference with batching and early exiting," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 4, pp. 1186–1200, Feb 2023.

[11] P. P. Reddy, K. Trisha, N. Sree, S. Bhaskaran, and N. Sampath, "Enhanced task scheduling in cloud computing: A comparative analysis and hybrid algorithm implementation using cloudsim," in *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, Jun 2024, pp. 1–5.

[12] S. Abubakar, S. Yusuf, A. Obiniyi, and A. Mohammed, "Modified round robin with highest response ratio next cpu scheduling algorithm using dynamic time quantum," *Sule Lamido University Journal of Science & Technology*, vol. 6, no. 1&2, pp. 87–99, Mar 2023.

[13] L. Li, "Advances in cpu scheduling algorithms in operating systems," *Highlights in Science, Engineering and Technology*, vol. 120, pp. 8–13, Dec. 2024.

[14] N. Tang and A. M. Kordon, "A fixed-parameter algorithm for scheduling unit dependent tasks with unit communication delays," in *European Conference on Parallel Processing*. Cham: Springer International Publishing, Aug 25 2021, pp. 105–119.

[15] S. Zrigui, R. de Camargo, A. Legrand, and D. Trystram, "Improving the performance of batch schedulers using online job runtime classification," *Journal of Parallel and Distributed Computing*, vol. 164, pp. 83–95, Jun 2022.

[16] B. Li, M. Mao, X. Liu, T. Liu, Z. Liu, W. Wen, Y. Chen, and H. Li, "Thread batching for high-performance energy-efficient gpu memory design," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–21, 2019.

[17] J. Mary and A. Alyosius, "Mobility and execution time aware task offloading in mobile cloud computing," *International Journal of Interactive Mobile Technologies*, vol. 16, no. 15, pp. 30–45, Aug 2022.

[18] A. Krizhevsky, "Cifar dataset," https://www.cs.toronto.edu/ kriz/cifar.html, 2009.

[19] Z. Research, "Fashion mnist dataset," https://www.kaggle.com/datasets/zalando-research/fashionmnist, 2021.

[20] H. K., "Mnist dataset," https://www.kaggle.com/datasets/hojjatk/mnist-dataset, 2021.

[21] NIH Clinical Center, "Nih chest x-rays," https://www.kaggle.com/datasets/nih-chest-xrays/data, n.d.

[22] Kaggle user kmader, "Skin cancer mnist: Ham10000," https://www.kaggle.com/datasets/kmader/skin-cancer-mnist-ham10000, n.d.