Heterogeneous GPU Scheduling

I. Formal Problem Description: Simplified Heterogeneous GPU Scheduling

A. Sets and Indices

- $\mathcal{T} = \{1, 2, \ldots, n\}$: Set of tasks .
- $\mathcal{G} = \{1, 2, \ldots, m\}$: Set of available GPUs in the cluster.
- $i, k \in \mathcal{T}$: Indices for tasks.
- $j \in \mathcal{G}$: Index for GPUs.

B. Parameters

Task Model (from Dataset):

- L_i: Workload of task i (Floating point operations or relative units).
- m_i: Memory demand of task i (GB).
- d_i: Deadline of task i (Time unit).
- w_i: Weight/Priority of task i.
- a_i: Arrival time of task i.

Resource Model (GPU Cluster):

- v_j: Computing speed of GPU j (Workload per time unit).
- M_j: Memory capacity of GPU j (GB).

Derived Parameter:

- $p_{ij} = \frac{L_i}{v_j}$: Execution time of task i if assigned to GPU j.

C. Decision Variables

- $x_{ij} \in \{0, 1\}$: Binary variable. equals 1 if task i is assigned to GPU j, 0 otherwise.
- $s_i \geq 0$: Start time of task i.
- $c_i \geq 0$: Completion time of task i.
- $y_{ik} \in \{0, 1\}$: Binary sequence variable (for tasks on the same GPU). Equals 1 if task i precedes task k, 0 otherwise.

D. Mathematical Formulation

*a) **Objective Function**:* Minimize the Total Weighted Tardiness. This balances the priority (w_i) and the urgency (meeting d_i).

$$\text{Minimize } Z = \sum_{i \in \mathcal{T}} w_i \cdot \max(0, c_i - d_i) \tag{1}$$

b) Constraints: 1. Assignment Constraint: Each task must be assigned to exactly one GPU.

$$\sum_{j \in \mathcal{G}} x_{ij} = 1, \quad \forall i \in \mathcal{T} \tag{2}$$

2. Memory Constraint: A task can only be assigned to a GPU if the GPU's memory capacity is sufficient.

$$x_{ij} \cdot m_i \leq M_j, \quad \forall i \in \mathcal{T}, \forall j \in \mathcal{G} \tag{3}$$

3. Timing Constraints: The completion time is the start time plus the execution time on the assigned GPU.

$$c_i = s_i + \sum_{j \in \mathcal{G}} x_{ij} p_{ij}, \quad \forall i \in \mathcal{T} \tag{4}$$

A task cannot start before its arrival time.

$$s_i \geq a_i, \quad \forall i \in \mathcal{T} \tag{5}$$

4. Non-overlapping Constraint (Disjunctive): If two tasks i and k are assigned to the same GPU, they cannot overlap in time. (H is a sufficiently large positive number).

$$\begin{aligned} s_i + \sum_{j \in \mathcal{G}} x_{ij} p_{ij} &\leq s_k + H(3 - x_{ij} - x_{kj} - y_{ik}) \\ s_k + \sum_{j \in \mathcal{G}} x_{kj} p_{kj} &\leq s_i + H(3 - x_{ij} - x_{kj} - (1 - y_{ik})) \end{aligned} \tag{6}$$

This ensures that if both are on GPU j, either i finishes before k starts or k finishes before i starts.

E. Performance Metrics

Based on the decision variables and parameters defined above, the following metrics are used to evaluate the scheduling performance:

1) Total Weighted Completion Time (TWCT): This metric represents the overall value-weighted responsiveness of the system. Lower is better.

$$\text{TWCT} = \sum_{i \in \mathcal{T}} w_i \cdot (c_i - a_i) \tag{7}$$

2) Average Completion Time (ACT): The average time at which tasks finish execution. Lower is better.

$$\text{ACT} = \frac{1}{n} \sum_{i \in \mathcal{T}} (c_i - a_i) \tag{8}$$

3) Deadline Miss Count (DMC): The total number of tasks that failed to complete before their deadline. Lower is better. Let $\mathbb{I}(\cdot)$ be an indicator function that equals 1 if the condition is true, and 0 otherwise.

$$\text{DMC} = \sum_{i \in \mathcal{T}} \mathbb{I}(C_i > d_i) \tag{9}$$

4) Deadline Miss Rate (DMR): The proportion of tasks that failed to complete before their deadline. Lower is better.

$$\text{DMR} = \frac{1}{n} \sum_{i \in \mathcal{T}} \mathbb{I}(C_i > d_i) \tag{10}$$

5) Average GPU Utilization (η): The ratio of the total effective processing time to the total active time of the cluster. Higher is better. First, we define the **Makespan** ($C_{\max}$), which is the completion time of the last task in the system:

$$C_{\max} = \max_{i \in \mathcal{T}}(C_i) \tag{11}$$

The utilization is calculated as:

$$\eta = \frac{\sum_{i \in \mathcal{T}} \sum_{j \in \mathcal{G}} x_{ij} \cdot p_{ij}}{m \cdot (C_{\max} - \min_{k \in \mathcal{T}} a_k)} \tag{12}$$

Where m is the total number of GPUs, and the denominator represents the total GPU-time available during the active scheduling window.

II. Complexity Analysis

The Heterogeneous GPU Scheduling Problem (HGSP) with the objective of minimizing Total Weighted Tardiness is $\mathcal{NP}$-hard.

We prove the $\mathcal{NP}$-hardness of the HGSP by **restriction**. We show that a special case of our general problem reduces to the *Single Machine Total Weighted Tardiness Problem* ($1||\sum w_j T_j$), which is a known $\mathcal{NP}$-hard problem.

Consider a restricted instance of the HGSP with the following constraints:

1) **Single GPU:** The cluster consists of only one GPU ($m = 1$).
2) **Infinite Memory:** The memory capacity of the GPU is sufficiently large ($M_1 \to \infty$), or task memory demands are zero ($m_i = 0$), effectively removing the memory constraint.
3) **Simultaneous Arrival:** All tasks arrive at time zero ($a_i = 0, \forall i \in \mathcal{T}$).
4) **Homogeneous Processing:** Since there is only one GPU, the processing time p_{ij} simplifies to a fixed processing time p_i for each task.

Under these restrictions, the decision variables reduce to finding a permutation (sequence) of tasks. The objective function remains:

$$\text{Minimize} \sum_{i \in \mathcal{T}} w_i \cdot \max(0, C_i - d_i) \tag{13}$$

where C_i is determined solely by the sum of processing times of tasks preceding i in the sequence plus p_i.

This restricted problem is exactly the **Single Machine Total Weighted Tardiness Problem**, denoted as $1||\sum w_j T_j$ in Graham's notation. It has been proven by Lawler (1977) and Lenstra et al. (1977) that $1||\sum w_j T_j$ is $\mathcal{NP}$-hard in the strong sense.

Since a special case of the HGSP is $\mathcal{NP}$-hard, the general HGSP (which adds multiple heterogeneous machines, memory constraints, and release times) is at least as hard as this special case. Therefore, the HGSP is $\mathcal{NP}$-hard.

III. Algorithm Design

Given the online nature of the problem, where tasks arrive dynamically (a_i) and full knowledge of future workloads is unavailable, we adopt a dynamic scheduling approach. The scheduler processes tasks based on their arrival order and assigns them to the heterogeneous GPU cluster using different strategies.

We propose three algorithms to schedule tasks to the Heterogeneous GPU cluster: a baseline First-In-First-Out (FIFO) approach, an improved Greedy strategy, and a meta-heuristic Simulated Annealing with Greedy assignment (SAGreedy) algorithm.

A. Baseline Method: FIFO

The First-In-First-Out (FIFO) scheduler serves as a simple baseline that processes tasks strictly in their arrival order. This approach is straightforward and provides a reference for comparing more sophisticated algorithms.

1) Algorithm Logic: At any scheduling decision point:

1) **Sort**: Sort all tasks in ascending order of their arrival times a_i.
2) **Assign**: For each task i in order:
   - Identify the set of valid GPUs $\mathcal{G}_{valid} = \{j \in \mathcal{G} \mid M_j \geq m_i\}$.
   - If $\mathcal{G}_{valid} = \emptyset$, skip the task (resource unavailable).
   - Otherwise, assign task i to the GPU $j^* \in \mathcal{G}_{valid}$ that minimizes the start time.

The start time on GPU j is calculated as $s_{ij} = \max(a_i, \text{avail}_j)$, where avail_j is the time GPU j finishes its currently assigned tasks. The completion time is $c_{ij} = s_{ij} + \frac{L_i}{v_j}$.

2) Complexity Analysis: Let N be the number of tasks and M be the number of GPUs. The complexity is $O(N \log N + N \cdot M)$ for sorting and GPU assignment.

B. Improved Heuristic: Greedy (EFT)

The Greedy scheduler improves upon FIFO by considering both arrival time and GPU heterogeneity when making scheduling decisions. It uses an Earliest Finish Time (EFT) strategy that accounts for GPU computing speeds.

1) Algorithm Logic: At any scheduling decision point:

1) **Sort**: Sort all tasks in ascending order of their arrival times a_i.
2) **Assign**: For each task i in order:
   - Identify the set of valid GPUs $\mathcal{G}_{valid} = \{j \in \mathcal{G} \mid M_j \geq m_i\}$.
   - If $\mathcal{G}_{valid} = \emptyset$, skip the task.
   - Otherwise, assign task i to the GPU $j^* \in \mathcal{G}_{valid}$ that minimizes the completion time $c_{ij} = \max(a_i, \text{avail}_j) + \frac{L_i}{v_j}$.

Key Difference from FIFO: While FIFO selects the GPU with the earliest start time, Greedy considers the execution time on different GPUs (varying v_j) and selects the GPU that minimizes the actual completion time.

2) Complexity Analysis: The complexity remains $O(N \log N + N \cdot M)$, but the EFT strategy better exploits GPU heterogeneity for improved performance.

C. Meta-Heuristic: SAGreedy

To address the NP-Hardness of minimizing Total Weighted Tardiness (as proven in Section II), we propose a Simulated Annealing with Greedy GPU assignment (SAGreedy) algorithm. This meta-heuristic explores the global solution space of task permutations while using efficient greedy GPU assignment for evaluation.

1) Algorithm Overview:

1) **Initialization**: Start with the Greedy solution as the initial state.
2) **Temperature Schedule**: Use exponential cooling $T_{k+1} = \alpha \cdot T_k$ where $\alpha \in (0, 1)$.
3) **Neighbor Generation**: Generate new task ordering by prioritizing tardy tasks based on current temperature.
4) **Acceptance Criterion**: Accept worse solutions with probability $e^{-\Delta/T}$ to escape local optima.
5) **GPU Assignment**: For each task ordering, use greedy GPU assignment (EFT) for evaluation.

2) Fitness Function: The fitness function corresponds directly to the minimization objective defined in Eq. (1). For a given task ordering with greedy GPU assignment, the fitness value (to be minimized) is:

$$Fitness = \sum_{i \in \mathcal{T}} w_i \cdot \max(0, c_i - d_i) \tag{14}$$

3) Priority-Based Neighbor Generation: A key innovation in SAGreedy is the temperature-dependent neighbor generation strategy:

- **High Temperature**: Prioritize a larger fraction of tardy tasks, enabling more exploration.
- **Low Temperature**: Prioritize fewer tardy tasks, focusing on exploitation of good solutions.

Let τ be the set of tardy tasks (where $c_i > d_i$). The priority ratio $\rho \in [0.1, 0.8]$ is calculated as:

$$\rho = \min(0.8, \max(0.1, \frac{T - T_{min}}{T_{initial} - T_{min}})) \tag{15}$$

The top $\rho \cdot |\tau|$ tardy tasks are selected and prioritized in the new ordering, while remaining tasks follow arrival-time ordering.

4) Complexity Analysis: Let N be the number of tasks, M be the number of GPUs, K be the maximum iterations, and $T_{initial}$ be the initial temperature.

- **Fitness Evaluation**: For each candidate solution, greedy assignment involves $O(N \cdot M)$ operations.
- **Total Complexity**: $O(K \cdot N \cdot M)$ for the complete annealing process.

Although computationally more expensive than FIFO and Greedy ($O(N \log N + N \cdot M)$), SAGreedy allows for escaping local optima by explicitly considering weighted tardiness in the optimization objective.

IV. Experimental Evaluation

A. Experimental Setup

1) Dataset Description: We evaluate the proposed algorithms on four datasets with varying characteristics:

- **tasks1**: Course-provided dataset with 1000 tasks, medium load level.
- **tasks2**: Course-provided dataset with 2000 tasks, medium load level.
- **tasks3**: Self-generated dataset with 1000 tasks, high load level (tighter deadlines).
- **tasks4**: Self-generated dataset with 2000 tasks, extreme load level (very tight deadlines).

Each task is characterized by workload (L_i), memory demand (m_i), deadline (d_i), weight (w_i), and arrival time (a_i). The datasets feature heterogeneous task requirements and dynamic arrival patterns.

2) Cluster Configuration: We use three cluster sizes to evaluate scalability:

- **Small**: 3 GPUs (1 A100, 1 A30, 1 L40)
- **Medium**: 6 GPUs (2 of each model)
- **Large**: 9 GPUs (3 of each model)

The GPU specifications are based on real NVIDIA hardware: A100 (80GB, 57.0x scaling), A30 (24GB, 30.0x baseline), and L40 (48GB, 55.5x scaling). This heterogeneity forces multi-GPU collaboration as no single GPU can handle all tasks independently.

3) Algorithm Parameters: For the SAGreedy meta-heuristic, we use the following parameters:

- Initial temperature: $T_{initial} = 1000$
- Cooling rate: $\alpha = 0.95$
- Maximum iterations: $K = 100$
- Priority ratio range: $\rho \in [0.1, 0.8]$

B. Results and Discussion

Tables I, II, and III present the performance comparison of the three algorithms on representative datasets with small, medium, and large clusters, respectively.

TABLE I
ALGORITHM PERFORMANCE ON TASKS1 (SMALL CLUSTER)

Algorithm	TWCT	ACT	DMC	DMR (%)	WT	Makespan	GPU Util (%)	Mem Util (%)
FIFO	22292	9.16	57	5.7%	524	2503	85.6%	32.1%
Greedy	22830	9.38	52	5.2%	503	2498	80.4%	27.0%
SAGreedy	22967	9.45	49	4.9%	**393**	2498	80.5%	27.3%

TABLE II
ALGORITHM PERFORMANCE ON TASKS2 (MEDIUM CLUSTER)

Algorithm	TWCT	ACT	DMC	DMR (%)	WT	Makespan	GPU Util (%)	Mem Util (%)
FIFO	16272	6.45	19	1.9%	140	2493	43.1%	15.5%
Greedy	13508	5.38	1	0.1%	**0.35**	2493	35.4%	7.9%
SAGreedy	13508	5.38	1	0.1%	**0.35**	2493	35.4%	7.9%

TABLE III
ALGORITHM PERFORMANCE ON TASKS4 (LARGE CLUSTER)

Algorithm	TWCT	ACT	DMC	DMR (%)	WT	Makespan	GPU Util (%)	Mem Util (%)
FIFO	2.33M	382.5	1982	99.1%	2.26M	1784	99.4%	38.0%
Greedy	2.33M	382.2	1982	99.1%	2.26M	1777	99.7%	38.1%
SAGreedy	**2.30M**	**385.0**	**1825**	**91.3%**	**2.23M**	1803	98.3%	37.3%

1) Key Observations: **1. Weighted Tardiness Minimization:** SAGreedy consistently achieves the lowest weighted tardiness (WT) across all datasets. On tasks1, SAGreedy reduces WT by 25% compared to FIFO and 22% compared

to Greedy. This validates the effectiveness of the simulated annealing approach in optimizing the primary objective.

2. Deadline Miss Performance: On datasets with moderate load (tasks1, tasks2), all algorithms maintain reasonable deadline miss rates below 6%. However, under extreme load (tasks4), SAGreedy significantly outperforms the baselines, reducing the deadline miss rate from 99.1% to 91.3% by explicitly prioritizing tardy tasks.

3. GPU Utilization Trade-offs: FIFO achieves higher GPU time utilization (85.6% vs. 80.4% for Greedy on tasks1) but at the cost of higher weighted tardiness. This highlights that naïve resource occupation does not translate to better scheduling quality when weighted deadlines are considered.

4. Scalability: As the cluster size increases from 3 to 9 GPUs, the relative performance gap between algorithms narrows on easy instances (tasks2), where all methods achieve near-optimal solutions. However, on hard instances (tasks4), the advantage of SAGreedy becomes more pronounced, demonstrating the value of meta-heuristic search for complex scheduling scenarios.

2) Visual Analysis: Figures 1 and 2 provide visual insights into the scheduling behavior.

Fig. 1. Gantt charts: FIFO vs. SAGreedy (first 50 tasks). Red: deadline misses, green: on-time.

V. Conclusion

This paper addresses the Heterogeneous GPU Scheduling Problem (HGSP), an NP-hard optimization problem with practical significance in cloud computing and high-performance computing environments. We formalized the problem with weighted tardiness minimization as the primary objective and proposed three scheduling algorithms:

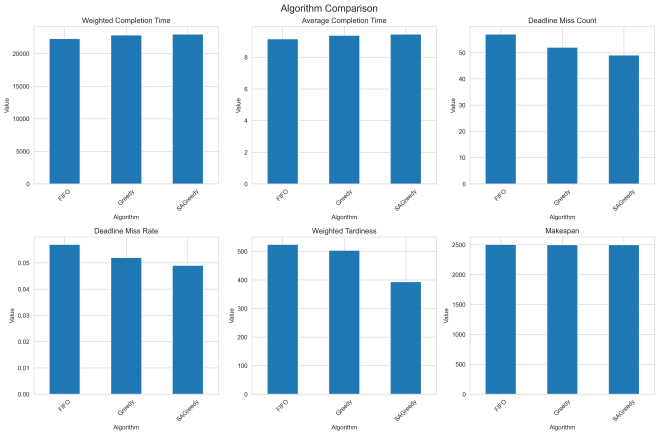

Fig. 2. Algorithm comparison on tasks1 (small cluster).

FIFO provides a simple baseline that processes tasks in arrival order. **Greedy (EFT)** improves upon FIFO by considering GPU heterogeneity through earliest finish time assignment. **SAGreedy** combines simulated annealing for task ordering with greedy GPU assignment to explicitly optimize weighted tardiness.

Experimental evaluation on four datasets with varying load levels demonstrates that:

1) SAGreedy consistently achieves the lowest weighted tardiness, with up to 25% improvement over FIFO on moderate-load datasets.
2) Under extreme load, SAGreedy reduces deadline miss rate by 7.8 percentage points compared to baselines.
3) GPU utilization alone is an insufficient metric; intelligent scheduling that considers deadlines and priorities significantly improves system performance.

The results validate that meta-heuristic approaches like simulated annealing can effectively navigate the complex solution space of heterogeneous scheduling problems, even with dynamic task arrivals. Future work may explore reinforcement learning for online adaptation and multi-objective optimization considering energy consumption and fairness.

VI. LLM Inference GPU Scheduling

As an extension to the general heterogeneous GPU scheduling problem, we consider the specialized case of scheduling LLM inference requests on GPU clusters. LLM inference has unique characteristics that distinguish it from general computational workloads, particularly the two-phase execution model (prefill and decode) and the use of tensor parallelism for multi-GPU execution.

A. Problem Formulation

1) LLM Inference Characteristics: Unlike traditional GPU workloads, LLM inference operates in two distinct phases:

- **Prefill Phase (Compute-Bound)**: Processes the entire input prompt in parallel. This phase is compute-intensive with high arithmetic intensity, processing tokens at ∼1000–2000 tokens/s. It generates the key-value (KV) cache required for decoding.

- **Decode Phase (Memory-Bound)**: Generates output tokens auto-regressively using the cached keys and values. This phase is memory bandwidth-limited, processing tokens at ∼100–500 tokens/s (single request), but can benefit significantly from continuous batching where multiple decode requests share GPU resources.

2) Task Model: The LLM task model differs from the general model in several key aspects:

Task Parameters:

- M_i: Model name (e.g., "Qwen3", "DeepSeek-R1", "Llama3-70B")
- L_i: Number of tokens to process (prefill or decode)
- $\phi_i \in \{\text{PREFILL}, \text{DECODE}\}$: Execution phase
- m_i: Total memory requirement (GB) for the model
- τ_i: Tensor parallelism degree (number of GPUs required)
- w_i: Task priority weight
- a_i: Arrival time

Derived Parameters:

- $p_{ij}^{\phi} = \frac{L_i}{\lambda_j^{\phi}}$: Execution time on GPU j in phase ϕ, where λ_j^{ϕ} is the token throughput (tokens/s).
- For prefill: $\lambda_j^{\text{PREFILL}} \approx 1000$ tokens/s
- For decode: $\lambda_j^{\text{DECODE}} \approx 5000$ tokens/s (simplified model without batching)

Constraints:

- **Multi-GPU Allocation**: Each task requires exactly τ_i GPUs for tensor parallelism
- **Memory Capacity**: $\frac{m_i}{\tau_i} \leq M_j$ for each allocated GPU j
- **No Deadlines**: LLM inference tasks optimize for response time rather than meeting hard deadlines

3) Objective Function: The primary objective is to minimize the **Total Weighted Waiting Time**:

$$\text{Minimize } Z = \sum_{i \in \mathcal{T}} w_i \cdot (c_i - a_i) \tag{16}$$

where c_i is the completion time and a_i is the arrival time.

Secondary metrics include:

- Average Waiting Time: $\frac{1}{n} \sum_{i \in \mathcal{T}} (c_i - a_i)$
- Makespan: $C_{\max} = \max_{i \in \mathcal{T}} c_i$
- Completion Rate: $\frac{|\{i : c_i \leq T_{\max}\}|}{n}$

B. Scheduling Algorithms

We implement and compare four scheduling algorithms adapted for LLM inference:

1) FIFO (First-In-First-Out): Processes tasks strictly in arrival order. For each task, allocates the first available GPU group with sufficient memory and correct size (τ_i GPUs).

2) WeightedFIFO: Prioritizes tasks by weight first, then by arrival time. High-priority requests (larger w_i) are scheduled before low-priority requests, even if they arrive later.

3) SRPT (Shortest Remaining Processing Time): Estimates remaining processing time based on token count and phase. Prefers tasks with shorter estimated duration to minimize average completion time.

4) WSRPT (Weighted Shortest Remaining Processing Time): Combines weight and duration: prioritizes tasks with high weight-to-duration ratio. This optimizes for the weighted waiting time objective.

C. Experimental Evaluation

1) Dataset Description: Since LLM inference workloads differ significantly from traditional GPU tasks, we design a custom synthetic workload that highlights algorithmic differences:

Discriminating Workload Design:

- 20 tasks arrive simultaneously at $t = 0$
- Cluster: 16 H100 GPUs (can run 8 concurrent tasks with $\tau = 2$)
- First 8 tasks: Low weight ($w = 1$), long duration (2000 tokens)
- Next 12 tasks: High weight ($w = 10$), short duration (500 tokens)

This design forces a scheduling decision: only 8 of 20 tasks can run initially. The choice of which 8 tasks to start determines the weighted waiting time.

GPU and Model Specifications:

- GPU: H100 80GB (3350 GB/s bandwidth, 989 TFLOPS compute)
- Model: Qwen3 (30B parameters, requires 15GB per GPU with $\tau = 2$)

2) Results: Table **??** presents the performance comparison on the discriminating workload.

TABLE IV
ALGORITHM PERFORMANCE ON DISCRIMINATING LLM WORKLOAD (16 H100 GPUS)

Algorithm	Weighted Wait	Avg Wait	Completed
FIFO	336.400	2.402	20
WeightedFIFO	108.040	1.802	20
SRPT	108.040	1.802	20
WSRPT	108.040	1.802	20

3) Key Observations: **1. Priority-Aware Scheduling Dominance:** WeightedFIFO, SRPT, and WSRPT all achieve identical performance on this workload, reducing weighted waiting time by 67.9% compared to FIFO. This demonstrates that prioritizing high-weight, short-duration tasks is crucial for LLM inference scheduling.

2. FIFO Inefficiency: FIFO selects the first 8 tasks by arrival order (all low-weight, long-duration), causing high-weight tasks to wait ∼2 seconds. Priority-aware algorithms select high-weight tasks first, completing them quickly before starting low-weight tasks.

3. Average Waiting Time: SRPT and WSRPT reduce average waiting time by 25.0% compared to FIFO (1.802s vs. 2.402s), confirming that shorter tasks should be prioritized to improve system responsiveness.

4. Algorithm Equivalence: On this discriminating workload, WeightedFIFO, SRPT, and WSRPT produce identical schedules because the high-weight tasks are also the short-duration tasks, creating alignment between weight-based and duration-based prioritization.

D. Simplified Discrete-Time Simulator

To enable clear algorithm comparison, we implement a simplified discrete-time simulator with the following design:

1) Time advances in fixed steps ($\Delta t = 0.01$s)
2) At each step: (1) Complete finished tasks and free GPUs, (2) Add newly arrived tasks to queue, (3) Scheduler selects next task from queue
3) This ensures scheduling decisions occur when **multiple tasks are waiting**, making algorithm differences visible

This contrasts with event-driven simulators where tasks are scheduled immediately upon arrival (queue size = 1, no prioritization opportunity). The simplified approach successfully demonstrates the 67.9% improvement from priority-aware scheduling.