High-Performance Task Scheduling for Heterogeneous Systems

Jingxuan Zhang jingxua7@andrew.cmu.edu
Peiran Hou peiranh@andrew.cmu.edu

We developed a task scheduler for Directed Acyclic Graphs (DAGs) that optimizes execution across heterogeneous CPU and GPU resources. Our work explores both static and dynamic scheduling strategies, along with static, dynamic, and learned cost models, compares cudaStream and cudaGraph, and evaluates their performance.

1 Background

1.1 Heterogeneous system

Heterogeneous computing consists of applications running on a platform that has more than one computational unit with different architectures, such as a multi-core CPU and a many-core GPU. Although kernels are often optimized for GPUs, exclusive GPU scheduling causes bottlenecks. In some cases, a better scheduling decision runs some kernels on the CPU, and even though they take longer than they would if run on the GPU, they still finish faster than if they were to wait for the GPU to be free. By executing kernels on available CPUs rather than waiting for GPU availability, the system avoids idle time and has a better performance than a static schedule that runs each kernel on the fastest device.

1.2 DAG Task Scheduler

A directed acyclic graph (DAG) is a directed graph with no directed cycles. It is commonly denoted as $G = (V, E)$, where V is a set of vertices and E is a set of directed edges. In task scheduling, each vertex represents a task, and each directed edge (u, v) represents a precedence constraint, meaning that task v can only start after task u has completed. Because a DAG contains no cycles, there is no circular dependency among tasks, and at least one valid topological ordering of the vertices exists. Such an ordering provides a legal sequential execution order, although many tasks may still be executed in parallel when their dependencies have been satisfied. The longest dependency chain in the DAG is often called the critical path, and it provides a lower bound on the total execution time because tasks on this path must be executed in sequence.

A DAG task scheduler is responsible for mapping the tasks in a DAG onto available computing resources while respecting all precedence constraints. In heterogeneous systems, such as CPU–GPU platforms, this problem becomes more challenging because different tasks may have different execution times on different devices, and transferring data between devices may introduce additional communication cost. Therefore, an effective scheduler must consider task readiness, processor availability, computation cost, communication cost, and the critical path structure. The overall objective is typically to minimize the makespan, which is the completion time of the final task in the DAG.

1.3 Parallelism Opportunities and Challenges

The DAG structure of task-based workloads exposes several levels of parallelism that our scheduler takes advantage of. At the task level, any set of nodes with no dependency between them can be dispatched at the same time, and the execution engine can make use of this by maintaining a ready queue that keeps releasing tasks as their predecessors finish. At the device level, CPU threads and GPU streams run in parallel: while the CPU thread pool handles lighter, shorter tasks, multiple CUDA streams run heavier compute kernels at the same time on the GPU, so the two resources work together rather than one waiting on the other. Finally, for workloads with repeated subgraph patterns, CUDA graph capture

can reduce the cost of launching GPU kernels repeatedly, improving GPU efficiency without changing the underlying schedule.

Achieving good parallel performance in heterogeneous DAG scheduling comes with several practical challenges. First, while the scheduler aims to use both CPU and GPU at the same time, routing tasks to the CPU can sometimes hurt rather than help. When the GPU already has enough work to stay busy, the extra cost of data transfers and synchronization between devices can cancel out any gain from using the CPU. Besides, the quality of any schedule depends heavily on how accurately the cost model predicts task execution times and data transfer costs. In practice, GPU kernel times vary with memory access patterns and cache state, and PCIe transfer latency behaves non-linearly with transfer size and contention, making precise prediction difficult.

2 Architecture

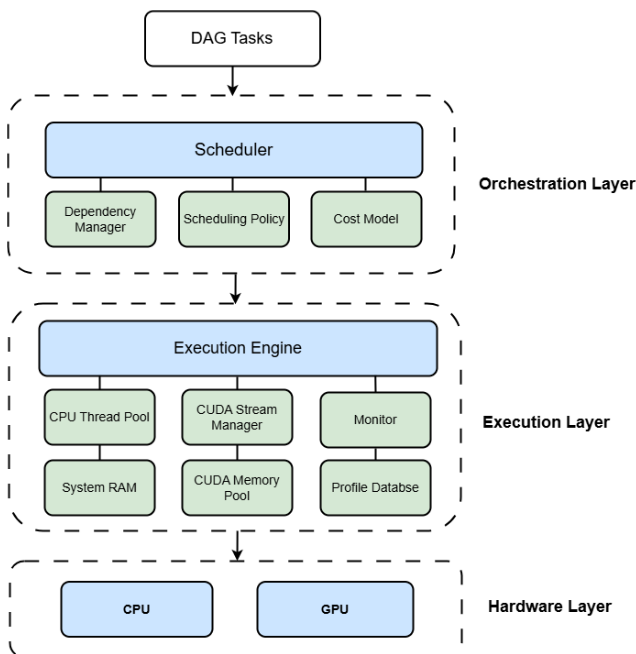


Figure 1: *System Architecture*

Figure 1 presents the system as a layered architecture. At the top, DAG tasks are passed into the orchestration layer, where the scheduler combines dependency management, scheduling policy, and the cost model to decide task order and device assignment. These decisions are then executed by the execution layer, which manages CPU threads, CUDA streams, memory resources, and runtime monitoring. Finally, the hardware layer performs the actual computation on CPU and GPU, while profiling data is collected and stored for later analysis and model refinement.

The workflow graph (Figure 2) starts from a benchmark DAG, which is converted into a `TaskGraph` with workload and dependency information. The graph is then evaluated by one of three cost-modeling methods: analytical, learned, or online calibration. Based on these estimates, the system follows either a static path, where a full schedule is computed before execution, or a dynamic path, where ready tasks are dispatched at runtime. During execution, profiling data such as measured runtime and actual device placement are written to the profiling database, which is later used for offline training and online calibration. Finally, the system outputs execution results for performance evaluation and visualization.

3 Approach

This project explores DAG scheduling for heterogeneous CPU–GPU execution from three main aspects. First, we study cost modeling, starting from a hardware-aware analytical model and progressively improving it with offline learned models and online calibration. Second, we compare scheduling strategies, including Static schedulers such as HEFT and MCT, as well as a Dynamic greedy scheduler that makes runtime decisions based on task readiness and estimated finish time. Third, we evaluate execution backends, using CUDA streams for flexible Dynamic execution and CUDA Graphs for more efficient submission of statically planned GPU work. We started our work on a repository on Github [9].

3.1 Cost Models

These three cost-modeling methods reflect an incremental refinement process. We begin with the static analytical model because the scheduler needs an initial estimate before any profiling data exists. This model is simple, deterministic, and portable across workloads, but it can only approximate real execution behavior because hardware effects such as launch overhead, memory hierarchy behavior, PCIe transfer latency, and task-type-specific efficiency are difficult to model exactly. To reduce this gap, we introduce the Static learned model, which uses profiling data from the target machine to learn the relationship between workload features and observed execution time. This makes the prediction more machine-specific

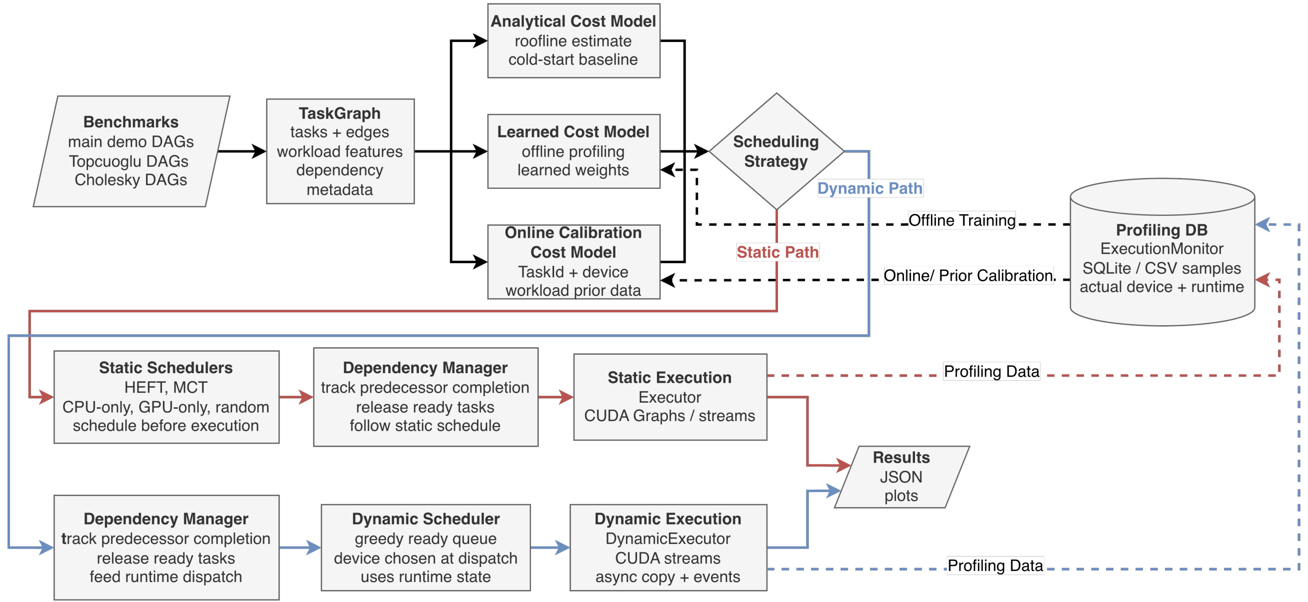


Figure 2: *System Workflow*

and better aligned with the actual runtime, while still allowing scheduling decisions to be made before execution. However, a learned model based only on general workload features may still miss task-specific behavior, especially when the same task appears repeatedly or when two tasks with similar features behave differently in practice. Therefore, we further add online calibration, which uses measured runtime feedback to refine predictions at two granularities: exact local observations for the same $(\texttt{TaskId}, \texttt{device})$ pair, and generalized prior data for similar workloads described by $(\texttt{TaskType}, \texttt{flops}, \texttt{in_bytes}, \texttt{out_bytes}, \texttt{device})$. In this progression, the analytical model provides a cold-start baseline, the learned model incorporates offline empirical knowledge, and online calibration adapts the estimates to repeated and newly observed runtime behavior.

Static Analytical Model The analytical cost model provides a hardware-aware baseline estimate without requiring prior profiling data. For each task, it estimates the execution time on CPU and GPU from the task's computational work and memory footprint, using a roofline-style model that considers peak compute throughput, memory bandwidth, cache behavior, and device-specific efficiency for different task types. Given a task with F floating-point operations and memory traffic B, the compute time on device d is estimated as

$$T_{\text{comp}}(t, d) = \max\left(\frac{F_t}{P_d \cdot \eta_{t,d}}, \frac{B_t}{M_d}\right), \quad (1)$$

where P_d is the peak compute throughput of device d, M_d is its memory bandwidth, and $\eta_{t,d}$ is a task- and device-specific efficiency factor. For GPU execution, the model also adds a fixed kernel launch overhead:

$$T_{\text{gpu}}(t) = T_{\text{comp}}(t, \text{GPU}) + T_{\text{launch}}. \quad (2)$$

This overhead is important for small tasks where launch latency can dominate execution time. This Static model is deterministic and can be used directly by schedulers before any runtime measurements are available.

Static Learned Model The learned cost model improves the analytical estimates by fitting prediction functions from profiling data collected on the target machine. Instead of relying only on theoretical hardware parameters, it learns the relationship between task features and observed execution time on each device. For a task t, we represent its workload by a feature tuple

$$x_t = (\texttt{TaskType}_t, F_t, B_t^{\text{in}}, B_t^{\text{out}}, d), \quad (3)$$

where F_t is the amount of computation, B_t^{in} and B_t^{out} are the input and output data sizes, and d is the target device. The learned model estimates execution time as

$$\hat{T}_{\text{learned}}(t, d) = f_{\theta_{\tau,d}}(F_t, B_t^{\text{in}}, B_t^{\text{out}}), \quad (4)$$

where $\tau = \texttt{TaskType}_t$ and $\theta_{\tau,d}$ denotes the trained parameters for that task type and device. The

model is trained offline using recorded profiling samples, producing separate predictors for different task types and devices. Transfer costs are also learned from measured host-to-device and device-to-host copy times:

$$\hat{T}_{\mathrm{comm}}(u,v) = g_{\phi_\delta}(S_{uv}), \tag{5}$$

where S_{uv} is the edge data size and δ is the transfer direction. During scheduling, the system uses the learned prediction whenever sufficient trained weights are available, and falls back to the analytical model otherwise.

Online Calibration The calibration mechanism is organized at two levels. First, for repeated executions of the same task, the system uses local calibration keyed by (`TaskId`, `device`). This offline/local data captures the measured runtime of a specific task on a specific device, so later executions of the same DAG can reuse task-specific observations instead of depending only on static estimates. Second, for different tasks with similar workload characteristics, the system uses prior-data calibration keyed by (`TaskType`, `flops`, `in_bytes`, `out_bytes`, `device`). This online/generalized path allows previously collected profiling data to inform predictions for unseen tasks that share similar computational and memory features. In combination, exact local calibration improves repeated-task accuracy, while workload-level prior calibration improves generalization across tasks of the same type.

3.2 Schedulers

3.2.1 Static Schedulers

We implement five static schedulers that compute a full device assignment before execution begins. **CPU-Only** and **GPU-Only** assign all tasks to a single device in topological order and serve as homogeneous baselines. **Random** assigns each task to CPU or GPU with fixed probability. The two primary schedulers are:

HEFT (Heterogeneous Earliest Finish Time [1]) assigns each task to the device minimizing its earliest finish time, guided by a pre-computed priority rank. It proceeds in two phases. Phase 1 computes an upward rank for every task:

$$\mathrm{rank}_u(t) = \bar{w}(t) + \max_{s\in\mathrm{succ}(t)}\big(\bar{c}(t,s) + \mathrm{rank}_u(s)\big) \tag{6}$$

where t denotes the current task, $s \in \mathrm{succ}(t)$ denotes a successor of t, $\bar{w}(t)$ is the average compute cost of t across all devices, and $\bar{c}(t,s)$ is the average communication cost on edge (t,s). Phase 2 processes tasks in descending rank order and assigns each to the device minimizing its Earliest Finish Time:

$$\mathrm{EFT}(t,d) = \mathrm{EST}(t,d) + w(t,d), \tag{7}$$

$$\mathrm{EST}(t,d) = \max_{p\in\mathrm{pred}(t)}\big(\mathrm{EFT}(p,d_p) + c(p,t,d_p \to d)\big) \tag{8}$$

where $w(t,d)$ is the compute cost of task t on device d, d_p is the device assigned to predecessor p, and $c(p,t,d_p \to d)$ is the communication cost of transferring data from p on device d_p to t on device d.

MCT (Minimum Completion Time [2]) replaces the global rank sort with a dynamic ready queue ordered by each task's earliest possible start time $r(t) = \max_{p\in\mathrm{pred}(t)} \mathrm{EFT}(p)$. Tasks are dequeued in order of $r(t)$ and assigned greedily by the same EFT criterion, so the scheduling order adapts to the simulated execution timeline rather than a pre-computed static rank.

3.2.2 Dynamic Greedy Scheduler

When using static task allocation and scheduling, a drift is observed when runtime uncertainty and unpredictable execution behavior occur[3], which is a source of load imbalance and idle time. So we decided to explore Dynamic scheduling which makes assignment decisions at runtime as tasks become ready. The Dynamic scheduler we implemented maintains a main FIFO queue of ready tasks and per-device sub-queues, and on each scheduling pass applies four rules in order:

1. **Sub-queue dispatch**: if a device is free and its sub-queue is non-empty, immediately dispatch the next task from that sub-queue.
2. **EFT comparison**: estimate finish times on both devices and assign the task to whichever finishes sooner:

$$\mathrm{free_finish} = t_{\mathrm{now}} + \hat{w}(t,\, d_{\mathrm{free}}), \tag{9}$$

$$\mathrm{busy_finish} = t_{\mathrm{free}}(d_{\mathrm{busy}}) + \hat{w}(t,\, d_{\mathrm{busy}}) \tag{10}$$

   If the busy device wins, the task is parked in its sub-queue.
3. **Stop**: if both devices are fully loaded, halt the pass until the next task completion event.

3.3 Execution

We use two execution backends because Dynamic and Static scheduling place different requirements on the runtime. Dynamic scheduling requires an online executor that can react as tasks become ready, overlap GPU kernels with data transfers, and enforce dependencies without blocking the CPU scheduler. CUDA streams and events provide this flexibility. Static scheduling, on the other hand, produces a fixed schedule before execution, which allows eligible GPU work to be represented as CUDA Graphs and submitted through a graph-based backend. The following two execution modes therefore target different tradeoffs: CUDA streams prioritize runtime flexibility, while CUDA Graphs exploit the predictability of static schedules.

Dynamic Execution with CUDA Streams For Dynamic scheduling, GPU execution uses a fixed pool of non-blocking CUDA streams managed by the `StreamManager`. When the scheduler assigns a ready task to the GPU, the execution engine acquires a stream and enqueues input setup, asynchronous data transfers, and the kernel launch on that stream. Cross-device CPU-to-GPU transfers are issued with `cudaMemcpyAsync`, so data movement becomes part of the stream-ordered GPU work instead of blocking the host before submission. The implementation also uses event-based dependency chaining: after a GPU task launches, the engine records a completion event on its stream and stores it by task id; successor GPU tasks can then call `cudaStreamWaitEvent` on their own streams to wait for predecessor completion without blocking the CPU dispatch thread. The task promise is resolved after the GPU work has been enqueued and the completion event has been recorded, while a background wait keeps the stream from returning to the pool before the event is reached. This design allows independent GPU tasks and transfers to overlap across streams while preserving DAG precedence constraints.

Static Execution with CUDA Graphs For Static scheduling, the device assignment and estimated timing of every task are computed before execution, and the executor follows this precomputed schedule at runtime. It uses the dependency manager to release tasks after their predecessors finish, submits each task to the scheduled device, wires successor inputs through copies or buffer borrowing, and records actual timing for evaluation. When CUDA Graph execution is enabled, eligible GPU tasks are executed through per-task CUDA graph capture: the engine checks that the input buffer is already prepared and that the task type supports capture, then captures the GPU function on a CUDA stream, instantiates the captured graph, and launches it with `cudaGraphLaunch`. This preserves the Static scheduling policy because CPU/GPU placement decisions do not change at runtime; CUDA Graphs only modify the GPU submission path by representing eligible GPU work as a fixed executable graph, which can reduce host-side scheduling and launch overhead.

3.3.1 CPU Thread Pool and GPU Stream Manager

The runtime uses a CPU thread pool (8 threads as default) to execute CPU-assigned tasks in parallel without repeatedly creating and destroying worker threads. Ready CPU tasks are submitted to the pool, where persistent worker threads fetch tasks from a shared queue and execute them concurrently. This design reduces host-side scheduling overhead and provides a simple execution backend for CPU tasks.

For GPU execution, the system uses a `StreamManager` that maintains a pool of reusable CUDA streams (4 streams as default). When a task is assigned to the GPU, the runtime acquires a stream, enqueues asynchronous memory copies and kernel launches on that stream, and then releases the stream after completion. The stream manager also supports CUDA event operations, which allows the runtime to coordinate dependencies across streams and overlap GPU computation with communication more effectively.

4 Testing

Our heterogeneous system consists of a CPU and a GPU. Most of our tests were conducted on the GHC machine with an NVIDIA GeForce RTX 2080 and an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz, while the granularity test was performed on a local machine equipped with an NVIDIA GeForce GTX 1650 Ti and an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. Our test cases are AI-generated.

4.1 Evaluation: Random DAG Generator

To evaluate scheduler performance more broadly, we adopt the parametric random DAG generator

from Topcuoglu et al. [1], which has become a standard benchmark methodology in heterogeneous task scheduling research. The generator produces weighted DAGs whose structural and computational properties are controlled by the following parameters.

- **Number of tasks** v: The total number of nodes in the DAG.
- **Shape parameter** α: Controls the aspect ratio of the graph. The DAG's number of levels is drawn from a uniform distribution with mean $\sqrt{v}/\alpha$, and the width of each level is drawn with mean $\alpha\sqrt{v}$. A value $\alpha \gg 1$ produces a wide, shallow graph with high parallelism; $\alpha \ll 1$ produces a tall, narrow graph with low parallelism.
- **Mean out-degree**: The expected number of successors per node, sampled uniformly. This controls the density of inter-task dependencies.
- **Communication-to-computation ratio (CCR)**: The ratio of average edge communication cost to average task computation cost.
- **Heterogeneity factor** β: Controls the spread of per-processor compute costs. The execution time of task n_i on processor p_j is sampled uniformly from:

$$\bar{w}_i\left(1 - \frac{\beta}{2}\right) \;\le\; w_{i,j} \;\le\; \bar{w}_i\left(1 + \frac{\beta}{2}\right)$$

  where $\bar{w}_i$ is the task's mean compute cost. A high β causes significant variation across processors; $\beta = 0$ gives a homogeneous system.

In our implementation, edge weights are sampled to match the target CCR via a nominal inter-device bandwidth. The per-task workload descriptor is derived from the average compute cost using a nominal effective throughput, allowing the existing cost model to consume the generated graphs without modification.

Test Configuration

Table 1 lists the parameter values used in our evaluation sweep.

For each parameter combination, multiple independent DAGs are generated using different seeds, and each DAG is executed multiple times to reduce timing variance. Unlike the Cholesky benchmark, these synthetic workloads do not have a closed-form correctness check; instead, we evaluate scheduler quality by comparing makespan across different scheduling strategies.

Parameter	Values	Effect
Task count v	8, 12, 16, 24	DAG size
Shape α	1.0	Balanced height/width
Mean out-degree	2, 3	Edge density
CCR	0.5, 1.0	Comm. vs. compute
β (fixed)	0.5	Moderate heterogeneity

Table 1: *Parameter sweep for the Topcuoglu random DAG evaluation.*

4.2 Evaluation: Different DAG Structures

To evaluate whether the schedulers behave consistently across different dependency patterns, we test several representative synthetic DAG structures. These DAGs are not tied to a specific application kernel; instead, they are used to isolate how graph topology affects scheduling decisions. As shown in Figure 3, the benchmark includes a chain DAG, a fork-join DAG, a layered DAG, a random DAG, and a reduction-tree DAG. The chain DAG has almost no task-level parallelism and is useful for testing dependency serialization and data-transfer overhead. The fork-join and layered DAGs expose more parallelism, allowing us to evaluate whether the scheduler can keep multiple devices busy. The random DAG represents irregular dependency patterns, while the reduction tree models workloads with many independent inputs that gradually merge into a final result.

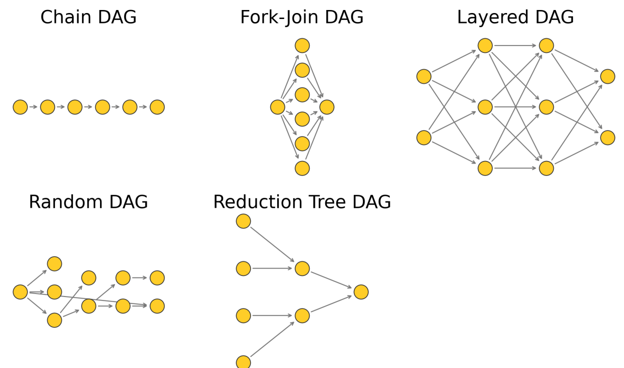


Figure 3: *Representative synthetic DAG structures used in the evaluation.*

Task	Operation	Indices	Device
POTRF	$A(k,k) = LL^\top$	—	CPU
TRSM	$A(i,k)L(k,k)^{-\top}$	$i > k$	GPU
SYRK	$A(i,i) - L(i,k)L(i,k)^\top$	$i > k$	GPU
GEMM	$A(i,j) - L(i,k)L(j,k)^\top$	$i > j > k$	GPU

Table 2: *Task types and device assignments.*

4.3 Validation: Cholesky Decomposition

To validate the correctness of our scheduler, we needed a workload with a mathematically verifiable output. Blocked Cholesky factorization is an ideal choice: given a symmetric positive definite (SPD) input matrix $\mathbf{A}$, the factorization must produce a lower-triangular factor $\mathbf{L}$ such that $\mathbf{A} = \mathbf{L}\mathbf{L}^\top$. Any violation of task ordering will corrupt the factor and produce a measurable numerical error.

4.3.1 The Cholesky DAG

The blocked Cholesky algorithm partitions an $n \times n$ matrix into $n_t \times n_t$ tiles, each of dimension $b \times b$ elements. Each factorization step over panel k generates four types of tasks with explicit data dependencies, summarized in Table 2.

4.3.2 Input Matrix Generation

The SPD input matrix is generated deterministically using a low-rank construction:

$$A(i,j) = \delta_{ij}\,\alpha + \sum_{t=0}^{T-1} f(i,t)\,f(j,t)$$

where $f(i,t) = \sin(0.013\,i\,t + \varphi) + 0.5\cos(\cdots)$ is a sinusoidal latent feature, $\alpha = 128.0$ is a diagonal bias ensuring positive definiteness, and the summation spans $T = 8$ low-rank terms. The random seed controls the phase offsets, enabling fully reproducible test matrices.

4.3.3 Correctness Metric

After the scheduler completes all tasks, we reconstruct the dense factor $\mathbf{L}$ and compute the relative Frobenius residual:

$$\varepsilon_{\text{rel}} = \frac{\|\mathbf{A} - \mathbf{L}\mathbf{L}^\top\|_F}{\|\mathbf{A}\|_F}$$

A run is considered correct if $\varepsilon_{\text{rel}} < 10^{-8}$ and the maximum absolute element-wise residual is below 10^{-6}.

4.3.4 Test Variables

We tested different problem sizes, as shown in Table 3.

Variable	Values	Effect
Tile count n_t	4,6,8,10,12	DAG size and parallelism
Tile size b	64,96,128,192,256,384	Per-task compute and data volume
Runs per case	3	Accounts for runtime variability

Table 3: *Parameter sweep for the Cholesky validation benchmark.*

5 Results

5.1 Cost Model

Estimation Error vs Task Number We test three different types of cost model on Static HEFT scheduler using the random-generated DAG tasks and get the cost estimation error when the number of tasks grows. The result is shown in Fig4.

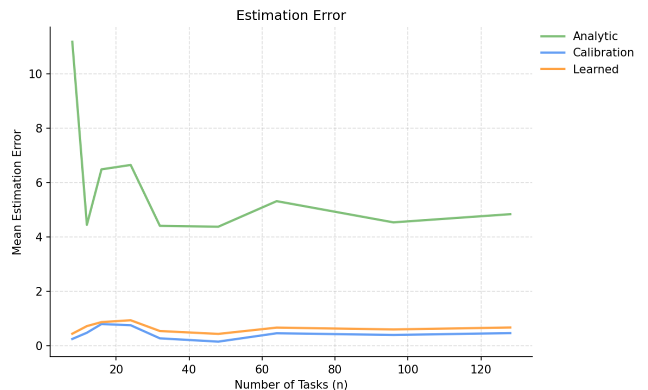


Figure 4: *Cost Estimation Error vs N*

As shown in Fig. 4, the analytic cost model has significantly higher mean estimation error than the calibration and learned models. This is because the analytic model relies on simplified roofline-style assumptions and hardware parameters. Although it captures the rough effects of computation, memory traffic, and GPU launch overhead, it cannot fully model task-type-specific behavior, cache effects, kernel implementation details, PCIe overhead, or runtime scheduling overhead. These factors are especially important for small tasks, where fixed costs such as launch latency and data movement can dominate execution time. As the DAG size

increases, the analytic error becomes more stable, but it remains consistently higher.

In contrast, both the calibration and learned cost models achieve much lower error across all task scales. The learned model uses profiling data from the target machine to capture empirical relationships between workload features and actual runtime, while the calibration model further refines predictions using measured feedback from previously observed tasks or similar workloads. As a result, both empirical methods keep the mean error below roughly one in most cases, showing that machine-specific runtime data is essential for accurate scheduling in heterogeneous CPU/GPU systems.

Estimation Error vs CCR This test is also using Static HEFT scheduler and the random-generated DAG tasks. Figure 5 shows that the mean estimation error is relatively insensitive to changes in the communication-to-computation ratio (CCR). The analytic model remains consistently inaccurate across the full CCR range. This suggests that its main source of error is not only the relative amount of communication, but also the simplified assumptions used to model both computation and transfer costs. In particular, fixed overheads, cache effects, kernel behavior, and PCIe latency are not fully captured by the analytical formula, so increasing or decreasing CCR does not eliminate the gap.

In contrast, the learned and calibration models maintain low and nearly flat error curves. Since these methods are based on measured runtime data, they can better capture both device execution behavior and data-transfer overheads on the target machine. The stability across CCR values indicates that the empirical models generalize well as the workload shifts from more compute-dominated to more communication-dominated cases.

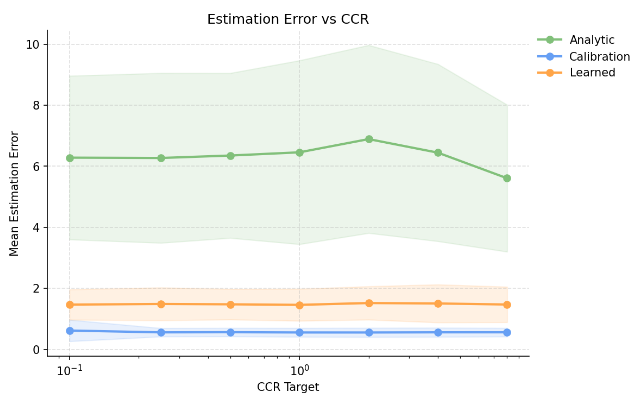
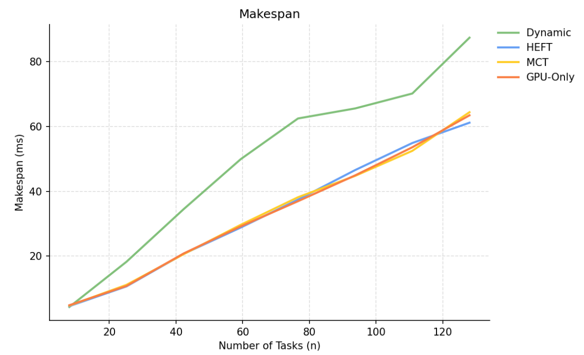
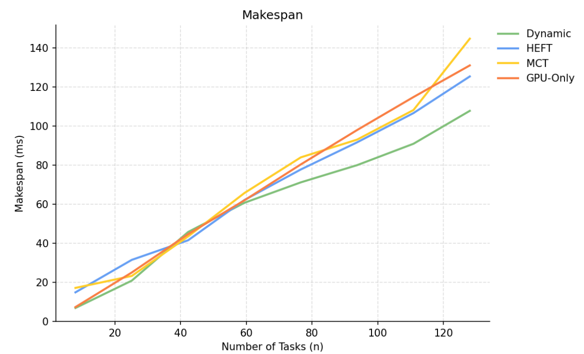


Figure 5: *Cost Estimation Error vs CCR*

5.2 Scheduling



(a) *Makespan without noise*



(b) *Makespan with noise*

Figure 6: *Makespan comparison: (a) without noise and (b) with noise*

Makespan vs Task Number We test the performance by adding noise to task execution. Experiments are conducted on randomly generated DAGs and learned cost model. In our study, noise is implemented as a random sleep duration injected into each task's execution to simulate the unpredictable timing fluctuations and delays found in real-world heterogeneous environments.

From Figure6 (a) we can find that, without noise, Static scheduling algorithms such as HEFT and MCT show better performance. When task durations are more predictable, these methods can calculate an optimal global plan that leads to a significantly lower makespan than the Dynamic approach. In this scenario, the Dynamic method performs poorly because it makes local, immediate decisions without the benefit of a long-term execution strategy, leading to less efficient resource allocation as the number of tasks scales.

However, the introduction of system noise in Figure (b) reverses these results. The performance of planning-based algorithms like HEFT and MCT degrades sharply because their rigid schedules cannot

adapt to unexpected timing fluctuations, causing accumulated delays across the DAG. In contrast, the Dynamic scheduler proves to be more robust, achieving the lowest makespan under uncertainty. The speedup over GPU-only execution is about 1.3×. By assigning tasks based on real-time resource availability rather than a fixed schedule, the Dynamic approach effectively absorbs system noise and gets better makespan.

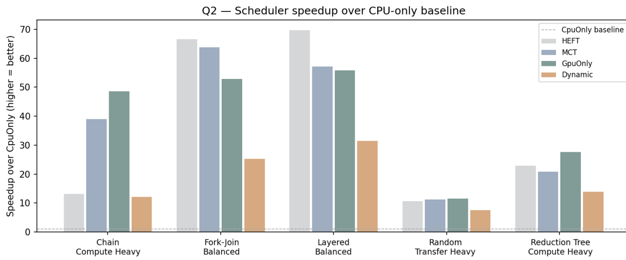


Figure 7: *Speedup on Different DAGs*

Speedup vs DAG Type Figure 7 evaluates the speedup of various schedulers across different DAG topologies. In balanced structures like Fork-Join and Layered DAGs, HEFT and MCT achieve the highest performance gains. This is because these topologies offer high parallelism, allowing smart schedulers to efficiently distribute tasks between the CPU and GPU while prioritizing the critical path. Their ability to balance computation and communication is key to maximizing throughput in these complex scenarios.

However, the Dynamic scheduler consistently shows significantly lower speedups compared to its static counterparts. Because it makes reactive, local decisions at runtime, it lacks the look-ahead capability required to optimize for the DAG's critical path in a stable environment. In these noise-free tests, the Dynamic scheduler cannot match the execution efficiency of static planners that pre-calculate an optimal global schedule.

The effectiveness of these algorithms changes greatly in other scenarios. For Chain structures, GpuOnly outperforms both HEFT and Dynamic because the lack of parallelism makes cross-device communication a liability rather than an asset; keeping the data on the GPU avoids unnecessary transfer overhead. In Transfer Heavy Random DAGs, all algorithms show a sharp decline in speedup. This indicates that when data movement becomes the primary bottleneck, the computational advantages of the GPU are minimized, proving that the hardware's interconnect speed becomes the limiting factor.

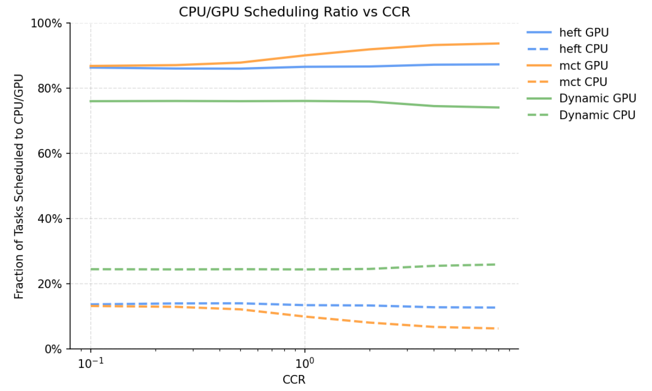


Figure 8: *CPU/GPU Ratio vs CCR*

GPU/CPU Task Ratio vs CCR Figure 8 shows the GPU and CPU scheduling ratio of different scheduling algorithims as CCR increases. The results are measured on random DAG workloads with varying CCR. HEFT and MCT maintain a high GPU allocation by prioritizing the earliest finish time and keeping successor tasks on the GPU to ensure zero transfer overhead, while Dynamic scheduling shows a higher CPU ratio because it schedules from the ready queue at runtime, offloading tasks to the CPU to hide expensive communication wait times.

As communication portion increases, HEFT maintains a remarkably stable allocation ratio, whereas MCT and Dynamic scheduling exhibit opposed trends. MCT's GPU allocation rises while Dynamic scheduling's GPU share declines. Driven by the logic of assigning tasks to whichever device finishes them fastest, MCT increasingly keeps successor tasks on the GPU to ensure zero transfer overhead as communication costs become prohibitive. In contrast, Dynamic Greedy schedules tasks based on the ready queue at runtime, opting to increase the CPU load to offset and hide expensive communication wait times rather than waiting for the GPU bus. Meanwhile, HEFT remains stable because its static, rank-based decisions are rooted in the global DAG topology, making it less sensitive to local latency trade-offs than the other two strategies.

5.3 Execution

Figure 9 compares the makespan of the static HEFT scheduler with and without CUDA Graph execution as the number of tasks increases. The results are evaluated on randomly generated DAGs. For small DAGs, the two methods perform similarly because the number of GPU submissions is limited, so kernel launch and host-side scheduling overhead do not

dominate the total runtime. In this range, the execution time is mainly determined by the task computation itself, and the extra cost of graph capture and instantiation can reduce the visible benefit of CUDA Graphs.

As the DAG becomes larger, CUDA Graph execution achieves lower makespan than the normal HEFT execution path. The gap becomes more obvious at high task counts because repeated GPU kernel submissions accumulate significant launch overhead in the stream-based backend. CUDA Graphs reduce this overhead by representing eligible GPU work as executable graph instances and launching them through `cudaGraphLaunch`, while preserving the same static scheduling decisions. This shows that CUDA Graphs are most beneficial for task-rich DAGs where the execution structure is fixed and GPU submission overhead becomes a larger part of total runtime.

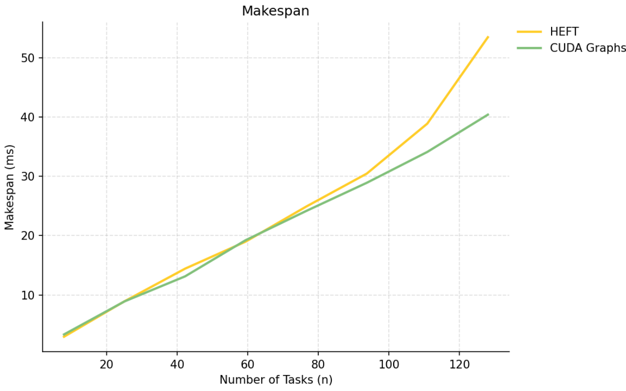


Figure 9: *Makespan CUDA Graph*

5.4 Task Granularity

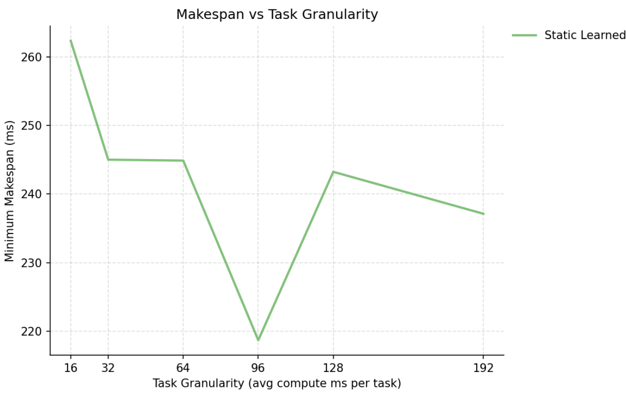


Figure 10: *Makespan across Different Task Granularity*

This experiment evaluates the effect of task granularity using the Static scheduler(HEFT) with learned cost model. Experiments are conducted on random DAGs. To isolate granularity, the total expected amount of computation is kept approximately fixed while the average computation per task is varied. Therefore, larger granularity values correspond to fewer but heavier tasks, while smaller granularity values correspond to more fine-grained tasks. As shown in Fig. 10, the makespan is not monotonic with task granularity. Very coarse-grained tasks, such as the 192 ms case, reduce scheduling overhead but expose less parallelism. Since there are fewer ready tasks at each stage of the DAG, the scheduler has fewer opportunities to overlap CPU and GPU work, which can increase the final makespan.

On the other hand, very fine-grained tasks also do not necessarily give the best performance. Although smaller tasks create more parallelism, they introduce more scheduling, dependency-management, and GPU launch/transfer overhead. The best result appears around the middle granularity range, especially near 96 ms per task, where the workload has enough tasks to expose useful parallelism but each task is still large enough to amortize runtime overhead. This suggests that for Static learned scheduling, there is a practical granularity sweet spot: tasks should be fine enough to keep heterogeneous devices busy, but not so fine that overhead dominates useful computation.

5.5 Cholesky Decomposition

Table 4 shows that the implementation is numerically stable overall, with 119 out of 120 validation runs passing. Minor discrepancies in a small number of runs are expected in parallel heterogeneous execution due to floating-point rounding errors and differences in execution order across devices. These variations do not indicate incorrect behavior, as the numerical error remains within acceptable tolerance.

Method	Correct	Incorrect
Static Learned	30/30	0/30
Static Online Calib.	30/30	0/30
Dynamic Learned	29/30	1/30
Dynamic Online Calib.	30/30	0/30
Overall	119/120	1/120

Table 4: *Correctness results for the Cholesky validation benchmark. Each method is evaluated on 30 runs.*

Figure 11 compares the makespan of Cholesky decomposition across different problem sizes. As

the tile count and tile size increase, the makespan increases sharply because the DAG contains more tasks and each task performs more dense linear algebra work. The Learned Static and Dynamic

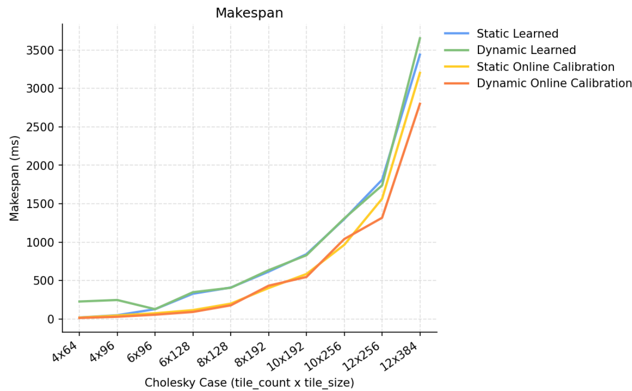


Figure 11: *Makespan across different sizes*

methods follow similar scaling trends, but the Dynamic learned scheduler is often slower. This is because Dynamic scheduling makes local runtime decisions based on ready tasks and device availability, while Cholesky decomposition has strong panel dependencies and a sensitive critical path. When the Dynamic scheduler assigns more tasks to the CPU to avoid waiting for the GPU, those slower tasks can delay later dependent work. The Static scheduler, by contrast, computes a complete schedule before execution and can better account for dependency structure.

Online calibration reduces makespan for both Static and Dynamic scheduling, especially on larger cases. This indicates that measured runtime feedback helps correct inaccuracies in the learned model, particularly for Cholesky kernels whose actual cost depends on library overheads, GPU launch behavior, and data transfers. The Dynamic online calibration method performs best on the largest case, showing that calibration can make online decisions more effective when the workload becomes sufficiently large.

6 Work Distribution

Both team members contributed substantially to the design, implementation, evaluation, and writing of the project. While each member took primary responsibility for different components, many parts of the system were developed collaboratively through joint design discussion, debugging, and experimental analysis. We therefore assign the final credit evenly between the two contributors.

Student	Main Contributions	Credit
Jingxuan Zhang	Project code structure design, Cost model development, schedulers, CUDA stream improvements, main test cases, testing and report writing	50%
Peiran Hou	Cost model development, Schedulers, Topcuoglu and Cholesky benchmarking, CUDA graph, profiling database support, testing and report writing	50%

Table 5: *Work distribution and credit allocation for the project.*

References

[1] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar. 2002, doi: 10.1109/71.993206.

[2] T. D. Braun, H. J. Siegel, N. Beck, et al., "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810–837, 2001.

[3] N. Grinsztajn, O. Beaumont, E. Jeannot, and P. Preux, "READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling," in *2021 IEEE Cluster Conference*, Portland, OR, USA, 2021, pp. 1–11.

[4] C. Gregg, M. Boyer, K. M. Hazelwood, and K. Skadron, "Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 209–218.

[5] M. Alaei and F. Yazdanpanah, "A survey on heterogeneous CPU–GPU architectures and simulators," *Concurrency and Computation: Practice and Experience*, vol. 37, no. 1, p. 8318, 2025.

[6] M. W. Convolbo and J. Chou, "Cost-aware DAG scheduling algorithms for minimizing execution cost on cloud resources," *The Journal of Supercomputing*, vol. 72, pp. 985–1012, 2016. doi: 10.1007/s11227-016-1637-7.

[7] D. Sirisha, “Complexity versus quality: a trade-off for scheduling workflows in heterogeneous computing environments,” *The Journal of Supercomputing*, vol. 79, pp. 924–946, 2023. doi: 10.1007/s11227-022-04687-x.

[8] D. Sirisha and S. S. Prasad, “MPEFT: a makespan minimizing heuristic scheduling algorithm for workflows in heterogeneous computing systems,” *CCF Transactions on High Performance Computing*, vol. 5, pp. 374–389, 2023. doi: 10.1007/s42514-022-00116-w.

[9] https://github.com/LessUp/heterogeneous-task-scheduler

[10] https://github.com/Heteroflow/Heteroflow