Accepted to be Published in:
CGO '26: Proceedings of the 24rd ACM/IEEE International Symposium on Code Generation and Optimization, Jan. 31–Feb. 4, 2026, Sydney, Australia

FRUGAL: Pushing GPU Applications beyond Memory Limits

Lingqi Zhang*¶‖, Tengfei Wang‡¶‡‡, Jiajun Huang§‖, Chen Zhuang*†, Ivan R. Ivanov*†, Peng Chen*, Toshio Endo†, Mohamed Wahib*‖

* RIKEN Center for Computational Science, {lingqi.zhang, chen.zhuang, peng.chen, mohamed.attia}@riken.jp

† Institute of Science Tokyo, ivanov.i.e641@m.isct.ac.jp, endo@scrc.iir.isct.ac.jp

‡ Google Cloud Japan, tengfeiwang@google.com

§ University of South Florida, jiajunhuang@usf.edu

***Abstract*—GPUs power modern scientific and AI applications, but their limited memory capacity restricts scalability. Buying GPUs with larger HBM is prohibitively expensive and still bounded by market limits. Existing solutions either exploit application-specific knowledge through out-of-core techniques, which lack generality, or rely on system-level page faulting, which is transparent but inefficient. We propose FRUGAL, an application-agnostic framework and methodology that reduces GPU memory footprint while sustaining high performance. FRUGAL formulates memory management as an optimization over an application's execution graph, encompassing prefetching, kernel execution, and offloading. Using static analysis and profiling, FRUGAL applies a two-phase scheduling and migration strategy, solving an otherwise intractable optimization efficiently. Evaluations on Tiled Cholesky Decomposition, Tiled LU Decomposition, Tiny-CUDA-NN, and QuEST show that FRUGAL significantly reduces maximum GPU memory usage by 80.21%, 80.20%, 64.75% and 60.86% with only a geometric mean of 28.31% slowdown. FRUGAL allows applications to exceed hardware-imposed limits, and maintains strong performance scalability beyond existing GPU memory constraints, without additional hardware cost.**

***Index Terms*—GPU Memory Management, Application-agnostic Optimization, Static Optimization**

I. Introduction

GPUs are widely adopted to accelerate modern scientific and AI applications. However, their limited onboard memory capacity often becomes a critical bottleneck, restricting application scalability. This is an issue in many areas, including quantum simulations [1], machine learning workloads [2], [3], etc. To overcome this memory wall, users are often forced to purchase high-end GPUs equipped with high-bandwidth memory (HBM), which can cost three to five times more than DDR5 [4]. For instance, an NVIDIA A100 GPU with 40GB of HBM2 is priced at $10,500, while the 80GB HBM2e variant costs $17,950—an 80% price increase for merely doubling the memory capacity [5]. Even at such costs, GPU memory remains capped by the maximum configurations available on the market, severely limiting problem sizes in many applications.

One method to mitigate GPU memory constraints is to exploit application-specific knowledge, offloading data from GPU memory (e.g., HBM) to secondary memory (e.g., host DRAM). This requires out-of-core techniques that overlap just-in-time data prefetching and computation. Prior work has explored such methods in deep learning [2], [6], [3], [7], [8], iterative stencils [9], [10], [11], matrix multiplication [12], graph processing [13], [14], sparse tensor contraction [15], [16], and more. While effective, these approaches are not generalizable, as they rely on application-specific knowledge and manual algorithmic changes.

On the other end of the spectrum is pure system-level solutions, such as NVIDIA's Unified Memory (UM) and AMD's Shared Virtual Memory (SVM), which transparently migrate data across memory hierarchies without programmer involvement. These approaches rely on demand-driven page faulting, which avoids application-specific modifications but lacks semantic knowledge of the workload. As a result, their performance often falls short of practical needs [17], [18]. Similarly, other page-fault based system-level techniques [19], [20], [21], [22], [23], [24] also suffer from inefficiency due to the absence of application-awareness.

This motivates a key research question: can we design a methodology that sits in the middle of the two ends of the spectrum explained above: an application-agnostic solution that outperforms pure system-level methods while avoiding the need for hand-crafted, application-specific optimizations. More specifically, could there be a method that requires no prior knowledge from programmers, yet still leverages application characteristics to achieve both reduced memory footprint and satisfactory performance? Addressing this question involves several challenges: (1) abstracting the memory footprint reduction problem into a tractable optimization problem, (2) designing methods to solve this optimization using system- and application-level information without user inputs, and (3) deploying the methodology in real-world applications to increase the effective memory capacity with sound performance.

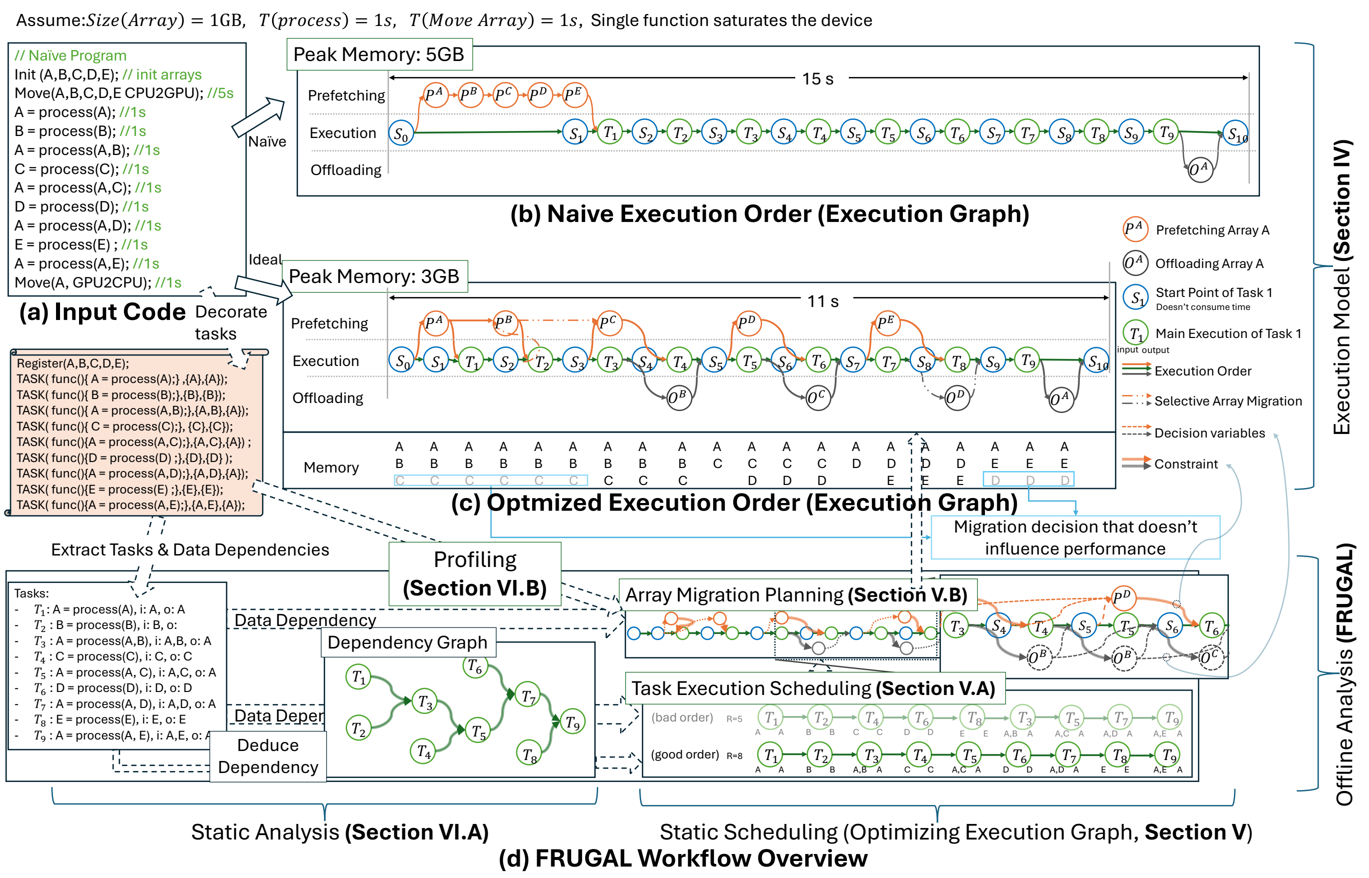
To address these challenges, we propose **FRUGAL**, a framework and methodology that reduces GPU memory footprint while maintaining performance. **FRUGAL** requires lightweight manual annotations of tasks and arrays, but automates all scheduling and data migration decisions—eliminating the need for algorithmic restructuring:

- **Offline Analysis.** FRUGAL operates as an offline methodology, eliminating runtime overhead. This makes it well-suited for production applications with long execution times such as DNN training [25], [26], [27], inference [28], [26], [29], quantum simulations [30], [31], [32], and numerical weather prediction [33], [34], [35]. We demonstrate that this offline strategy achieves satisfactory runtime efficiency.

¶ Equal contribution

‖ Corresponding authors

‡‡ Work done during the author's Master's study at Institute of Science Tokyo


```
// Naive Program
Init (A,B,C,D,E); // init arrays
Move(A,B,C,D,E CPU2GPU); //5s
A = process(A); //1s
B = process(B); //1s
A = process(A,B); //1s
C = process(C); //1s
A = process(A,C); //1s
D = process(D); //1s
A = process(A,D); //1s
E = process(E) ; //1s
A = process(A,E); //1s
Move(A, GPU2CPU); //1s
```

```
Register(A,B,C,D,E);
TASK( func(){ A = process(A);} ,{A},{A});
TASK( func(){ B = process(B);},{B},{B});
TASK( func(){ A = process(A,B);},{A,B},{A});
TASK( func(){ C = process(C);}, {C},{C});
TASK( func(){A = process(A,C);},{A,C},{A}) ;
TASK( func(){D = process(D) ;},{D},{D} );
TASK( func(){A = process(A,D);},{A,D},{A});
TASK( func(){E = process(E) ;},{E},{E});
TASK( func(){A = process(A,E);},{A,E},{A});
```

```
Tasks:
-   T1 : A = process(A), i: A, o: A
-   T2 : B = process(B), i: B, o:
-   T3 : A = process(A,B), i: A,B, o: A
-   T4 : C = process(C), i: C, o: C
-   T5 : A = process(A, C), i: A,C, o: A
-   T6 : D = process(D), i: D, o: D
-   T7 : A = process(A, D), i: A,D, o: A
-   T8 : E = process(E), i: E, o: E
-   T9 : A = process(A, E), i: A,E, o: A
```

Assume: $Size(Array) = 1\text{GB}$, $T(process) = 1s$, $T(Move\ Array) = 1s$, Single function saturates the device
(a) Input Code
Decorate tasks
Naïve
Ideal
Peak Memory: 5GB
15 s
Prefetching
Execution
Offloading
(b) Naive Execution Order (Execution Graph)
Peak Memory: 3GB
11 s
Memory
(c) Optmized Execution Order (Execution Graph)
Prefetching Array A
Offloading Array A
Start Point of Task 1
Doesn't consume time
Main Execution of Task 1
input output
Execution Order
Selective Array Migration
Decision variables
Constraint
Execution Model (Section IV)
Migration decision that doesn't influence performance
Extract Tasks & Data Dependencies
Profiling (Section VI.B)
Array Migration Planning (Section V.B)
Data Dependency
Dependency Graph
Task Execution Scheduling (Section V.A)
(bad order) R=5
(good order) R=8
Deduce Dependency
Offline Analysis (FRUGAL)
Static Analysis (Section VI.A)
Static Scheduling (Optimizing Execution Graph, Section V)
(d) FRUGAL Workflow Overview

Fig. 1: Overview of FRUGAL (through a motivating example)

- **Optimization formulation.** We model the memory footprint reduction problem as an optimization over the *execution graph*, consisting of prefetching, execution, and offloading tasks. This abstraction enables a unified treatment of diverse applications, removing the need for custom solutions.
- **Static scheduling and analysis.** We employ static scheduling, first scheduling task execution based on data reuse, and then optimizing data migration via mixed-integer programming (MIP). To enable effective scheduling, we extract both data and control dependencies through static analysis, augmented with profiled performance data.
- **Complexity reduction.** The joint optimization problem naively has an intractable search space of $O(n! \cdot 2^{mn^2})$ for n tasks and m arrays. FRUGAL introduces a two-phase approach—task execution scheduling and array migration planning—that significantly reduces computational complexity to a practical level. FRUGAL can finish optimization within 3 seconds for 32-task graphs and 7.5 seconds for 1,152-task graphs through subgraph decomposition.
- **Empirical validation.** We evaluate FRUGAL on real-world applications and demonstrate substantial reductions in maximum memory footprint, while sustaining high performance. Moreover, FRUGAL enables applications to exceed the original GPU memory capacity, supporting larger problem sizes and effectively pushing the limits of hardware constraints. FRUGAL enables execution beyond GPU memory capacity. For instance, we successfully run a 225 GB Tiled Cholesky Decomposition on a 96 GB GH200 (2.34× memory oversubscription) with only 4% performance degradation.

II. Motivating Example

To illustrate how FRUGAL works, consider the simple motivating code example shown in Fig. 1(a). A common problem arises when programmers overlook memory management and task scheduling during application development. As a result, they typically allocate all memory at program initialization, or at particular stages in the program, and defer all task scheduling decisions to the runtime system. This convenience for the programmer leads to the suboptimal scheduling order shown in Fig. 1(b), where tasks execute without considering efficient use of memory.

FRUGAL addresses this problem by leveraging two key pieces of information: ❶ data dependencies, collected from static analysis and user annotations, and ❷ task execution times, obtained through profiling. Rather than relying solely on runtime scheduling decisions, FRUGAL uses this information to statically schedule the program and strategically insert memory migration operations.

The impact of this optimization is significant. As shown in Fig. 1(c) and Fig. 1(d), FRUGAL's optimized execution order achieves a 40% reduction in peak memory footprint and a 27% improvement in performance. While careful manual optimization might achieve similar results, FRUGAL automates this complex process, making such optimizations accessible without requiring expert knowledge.

In this section, we use a simplified example for illustration. A detailed step-by-step walkthrough of a real-world example is provided in Appendix C.

A. Practical Applications of Memory Savings

Memory savings by FRUGAL enables three key use cases:

1) **Enable previously infeasible computations:** Run applications that exceeded available GPU memory, such as larger neural networks or higher-resolution simulations.
2) **Scale up problem sizes:** Use freed memory to increase computational scale, e.g. higher mesh resolutions, more qubits in quantum simulations, or larger batch sizes in ML training.
3) **Reduce hardware costs:** Deploy on GPUs with smaller memory capacity (e.g., H100 80GB instead of H100 94GB), or co-locate multiple applications for better hardware utilization.

These use cases directly correspond to our evaluation scenarios in Section Section VIII.

III. Overview of FRUGAL

As shown in Fig. 1, FRUGAL automatically combines static dependency analysis with dynamic profiling to optimize the use of GPU memory without manual intervention. This optimization is enabled by our execution model.

Execution Model. At the core of FRUGAL is our execution model: an execution graph in which concurrent lanes execute simultaneously. This model (Section IV-D) builds on practical architectural (Section IV-B) and performance (Section IV-C) assumptions, and serves as the foundation for our scheduling decisions.

Scheduling the Execution Graph. FRUGAL's main contribution is formulating memory footprint reduction as a static scheduling problem. We first schedule task execution based on data reuse metrics (Section V-B), then we optimize data migration placement using Mixed Integer Programming (Section V-C).

Offline Analysis for Scheduling Support. To enable effective scheduling, we perform *static analysis* that extracts data dependencies and control dependencies (Section VI-B), while incorporating *profiled performance information* (Section VI-C).

Limitations FRUGAL handles input size variations without replanning when computation scales proportionally with input size. However, applications with input-dependent control flow require replanning. We note that many scientific HPC applications exhibit input-independent behavior [36], [37], [38].

IV. Execution Model: Execution Graph in FRUGAL

Memory-aware scheduling and migration present a vast and complex search space that makes finding optimal solutions challenging. FRUGAL addresses this by first constraining the search space through practical assumptions (this section), then developing efficient optimization techniques to make the problem tractable (Section V). This section establishes the architectural (Section IV-B) and performance assumptions (Section IV-C) that reduce the exponential complexity.

A. Problem Abstraction

FRUGAL operates on key concepts: *Task*, *Array*, and *Dependency*:

- **Task:** Is an atomic scheduling unit representing GPU computation. Each task consists of one or more GPU operations (kernels, library calls, or fused operations) executed as a unit.
- **Array:** A contiguous memory region residing in either CPU or GPU memory space, accessed for read and/or write by one or more tasks. Arrays establish data dependencies between tasks, serving as inputs consumed and outputs produced during *Task* execution. FRUGAL manages each *Array* as a memory migration unit to optimize the memory footprint.
- **Dependency:** Execution ordering constraints between tasks, encompassing both data dependencies (derived from array usage) and control dependencies that can be deduced from the data dependency graph to enforce execution order beyond explicit data flow requirements.

B. Architecture Model

FRUGAL targets CPU-GPU memory hierarchies with the following characteristics. While designed for this architecture, the approach can be adapted to other heterogeneous memory systems by modifying these assumptions:

- **Memory Hierarchy:** Two-level physically separated memory system comprising *main memory* (GPU memory) and *secondary memory* (CPU memory).
- **Infinite Secondary Memory:** Applications are unlikely to exceed *secondary memory* capacity limits. If they do, then that is subject to orthogonal solutions that go out-of-core from CPU memory to Flash memory, local storage etc
- **Bidirectional Data Transfer:** Data transfers between main and secondary memory in one direction do not affect bandwidth in the opposite direction. Examples include PCIe, Nvidia NVLink, and AMD Infinity Fabric architectures.

C. Performance Assumptions and Practical Considerations

To manage design complexity while maintaining practical applicability, we establish performance assumptions and discuss their implications and limitations.

1) Core Assumptions: Our execution model relies on two fundamental assumptions:

a) Saturation: We assume both device and memory bus should be driven by the scheduled tasks to reach saturation:

- **Device Saturation:** Each *Task* fully utilizes computational resources, achieving maximum occupancy and throughput. Consequently, concurrent tasks must serialize as no additional resources remain. For two tasks T_1 and T_2, the total runtime is invariant across execution orders: sequential ($T_1 \rightarrow T_2$/$T_2 \rightarrow T_1$) or parallel launch ($T_1 \parallel T_2$).
- **Memory Bus Saturation:** Each *Array* migration fully utilizes available bandwidth during transfer.

This saturation assumption transforms the complex space of concurrent execution into serialized queues, significantly simplifying scheduling analysis.

b) Independence: Memory migration and task execution operate on independent hardware resources without mutual interference. This enables separate optimization of schedules for execution, prefetching, and offloading without cross-component performance impacts.

2) Auxiliary Assumptions:

- Task rescheduling preserves individual task performance
- Memory allocation/deallocation overhead is negligible

3) Practical Considerations and Limitations:

a) Independence Assumption Accuracy: While the independence assumption holds well for compute-bound kernels, memory-bound tasks may experience interference. Our microbenchmarks on NVIDIA GH200 (4 TB/s memory bandwidth, 0.45 TB/s migration bandwidth) show:

- Compute-bound tasks: negligible impact from concurrent migration
- Memory-bound tasks: up to 22.5% slowdown under concurrent migration

Despite the potential for 20% error in worst-case scenarios, this assumption enables tractable analysis while providing reasonable accuracy for mixed workloads.

b) Saturation Assumption Validity: The saturation assumption requires well-optimized kernels—a standard expectation in HPC. When kernels underutilize resources:

- Our model provides performance upper bounds
- Potential concurrent execution opportunities are missed

However, modern HPC applications typically achieve near-saturation performance through careful optimization [39], [40], [41], [42], validating this assumption for our target domain.

c) Model Applicability: These assumptions work for:

- Well-optimized HPC applications
- Workloads with mixed compute/memory-bound tasks
- Systems with dedicated migration hardware (PCIe, NVLink)

D. Execution Graph

We model execution using an Execution Graph derived from a Directed Acyclic Graph (DAG). This model consists of multiple concurrent lanes that execute simultaneously.

- **Execution:** Nodes in the DAG representing *Task* execution.
- **Migration:** Nodes in the DAG representing *Array* migration. Two types exist: *Prefetching* (CPU to GPU) and *Offloading* (GPU to CPU).
- **Edges:** Directed edges in the DAG representing execution order. Edges must preserve all dependencies.
- **Lanes:** Concurrently executing queues corresponding to our architecture model. Our approach relies on three assumptions: ❶ tasks execute on GPU, ❷ bidirectional memory migration (Section IV-A), and ❸ independence of migration and execution. Based on these, we establish three lanes: *Execution*, *Offloading*, and *Prefetching*, which operate simultaneously without interference (Section IV-C).

These three lanes form the foundation of our scheduling approach: FRUGAL overlaps computation with bidirectional data movement while preserving task dependencies.

E. Constructing the Execution Graph

This section describes the construction of an execution graph $G' = (V', E')$ from a dependency graph $G = (V_{task}, E_{dep})$ through the following steps.

1) Input Specification: The inputs are the tasks' logical dependencies and the arrays' data dependencies.

The task dependency graph is a DAG $G = (V_{task}, E_{dep})$ where:

- $V_{task} = \{t_1, t_2, ..., t_n\}$ represents the set of tasks
- $E_{dep} \subseteq V_{task} \times V_{task}$ represents dependency edges, where $(t_i, t_j) \in E_{dep}$ indicates task t_i must be complete before task t_j
- Each task $t_i \in V_{task}$ has an execution time $w_i \in \mathbb{R}^+$

Data dependencies are characterized by:

- $\mathcal{A} = \{a_1, a_2, ..., a_m\}$: the set of arrays
- Each array $a_k \in \mathcal{A}$ has size $|a_k| \in \mathbb{N}$
- For each task $t_i \in V_{task}$:
  - $\mathcal{I}(t_i) \subseteq \mathcal{A}$: input arrays
  - $\mathcal{O}(t_i) \subseteq \mathcal{A}$: output arrays
  - $\mathcal{D}(t_i) = \mathcal{O}(t_i) \cup \mathcal{I}(t_i)$: all arrays for task t_i

2) Task Begin Node Insertion: We add a task start node s_i before each task execution, with weight $w(s_i) = 0$. Additionally, we insert sentinel nodes s_0 and s_{n+1} at the beginning and end of the program, respectively. The execution lane nodes become:

$$V_{exec} = \{s_0, s_1, t_1, s_2, t_2, ..., s_n, t_n, t_{n+1}\} \tag{1}$$

These zero-weight synchronization nodes serve two purposes: (1) they provide explicit points for coordinating data transfers with task execution, and (2) they enable clear visualization of array lifetimes in the execution graph, as shown in Fig. 1.

3) Data Migration Node Insertion: We insert two types of data transfer nodes:

a) Prefetching nodes: For each array a_k that initial CPU-to-GPU transfer before task v_i and needs to finalized before task v_j:

$$V_{prefetch} = V_{prefetch} \cup \{p^k_{i,j}\} \tag{2}$$

with weight $w(p^k_i) = |a_k|/B_P$, where B_P is the prefetch bandwidth.

b) Offloading nodes: For each array a_k that initial GPU-to-CPU transfer after task v_i completes and before task v_j begins:

$$V_{offload} = V_{offload} \cup \{o^k_{i,j}\} \tag{3}$$

with weight $w(o^k_{i,j}) = |a_k|/B_O$, where B_O is the offload bandwidth.

4) Execution Graph Construction: The complete execution graph contains:

$$V' = V_{exec} \cup V_{offload} \cup V_{prefetch} \tag{4}$$

The edge set E' enforces:

- **Task dependencies:** $(t_i, s_j) \in E'$ if $(v_i, v_j) \in E_{dep}$

- **Task decomposition:** $(s_i, t_i) \in E'$ for all i
- **Prefetch completion:** $(p^k_{i,j'}, s_{j'}) \in E'$ for all $a_k \in \mathcal{I}(v_j)$ where $j' \leq j$ and $p^k_{i,j'}$ represents prefetching array a_k from CPU to GPU between tasks t_i and t'_j, essentially before task t_j.
- **Offload initiation:** $(t_{i'}, o^k_{i',j}) \in E'$ for all $a_k \in \mathcal{O}(v_i)$ that must be offloaded, where $o^k_{i',j}$ represents offloading array a_k from GPU to CPU after task t_i completes
- **Data coherence:** $(o^k_{i,j}, p^k_{j',j''}) \in E'$ where $j' \geq j$, ensuring that if array a_k is offloaded by $o^k_{i,j}$ and later needed by task $t_{j''}$, it must be prefetched again via $p^k_{j',j''}$

As such, any edge set satisfying these constraints forms a valid execution graph.

5) Lane Assignment and Serialization: Define the lane assignment function $\lambda : V' \rightarrow \{exec, offload, prefetch\}$. Within each lane, nodes are serialized:

- **Execution lane:** Any chronological order respecting E_{dep}
- **Prefetch lane:** Ordered by source task index, then destination task, then arbitrarily
- **Offload lane:** Ordered by source task, then destination task, then arbitrarily

F. Performance Projection

1) Execution Time: We set the projection of the the execution time of the DAG to be that of the tasks in its critical path. In our execution model, we compute the longest path from s_0 to s_{n+1}, accounting for parallel execution across lanes between synchronization points in the *execution lane*.

Let $L(v)$ denote the longest path from s_0 to vertex v. We compute:

$$L(v) = \max_{u \in \text{pred}(v)} \{L(u) + w(u)\} \tag{5}$$

where pred(v) denotes predecessors of v in the execution graph and $w(u)$ represents the node weight.

The total execution time is:

$$T = L(s_{n+1}) = \max_{\text{path } p \in \mathcal{P}} \sum_{v \in p} w(v) \tag{6}$$

where $\mathcal{P}$ represents all paths from s_0 to s_{n+1} in the execution graph, accounting for parallel execution in prefetch, execution, and offload lanes.

2) Memory Footprint: Memory migration operations affect memory usage as follows: prefetching increases memory usage when the operation begins (memory allocation), while offloading decreases memory usage only after the operation completes (memory deallocation).

We can determine whether array a_k occupies memory during task t_i using the indicator variable:

$$y^k_i = \sum_{u=0}^{i} \sum_{v=u+1}^{n} p^k_{u,v} - \sum_{u=0}^{i-1} \sum_{v=u+1}^{i} o^k_{u,v} \tag{7}$$

where $p^k_{u,v} \in \{0, 1\}$ indicates whether array a_k is prefetched between tasks u and v, and similarly $o^k_{u,v} \in \{0, 1\}$ indicates whether array a_k is offloaded. The total memory usage at task t_i is:

$$Mem_i = \sum_{k \subseteq \mathcal{A}} |a_k| \, y^k_i \tag{8}$$

where $|a_k|$ is the size of array a_k.

G. Visualized Example

The naive execution graph in Figure 1(b) and optimized graph in Figure 1(c) both satisfy dependency constraints yet achieve vastly different performance, demonstrating the importance of strategic memory migration placement. In Section V, we illustrate how we identify the optimized execution graph.

V. Optimizing Execution Graph to Reduce Peak Memory Usage

Having established a constrained execution model in the previous section, which still presents $O(n! \times 2^{n^2 m})$ complexity, this section presents our optimization approach that makes the problem computationally tractable, while finding execution graphs with minimal memory footprint.

A. Overview of Two-Phase Optimization

Leveraging the independence assumption between memory migration and task execution (Section IV-C), we decompose the optimization into two sequential phases: task execution scheduling (Section V-B) and array migration planning (Section V-C).

1) Complexity of Joint Optimization: It is worth noting that, a joint optimization would require exploring all possible combinations of execution orders and migration decisions, resulting in a computationally intractable search space of $O(n! \cdot 2^{mn^2})$ for n tasks and m arrays. Our two-phase approach reduces this complexity by first fixing the task execution order to maximize data reuse opportunities (Phase 1), then determining optimal migration points within that fixed schedule (Phase 2). While this decomposition may sacrifice global optimality, it enables practical solution times while achieving significant memory footprint reductions.

2) Phase 1: Task Execution Scheduling: This phase determines a task execution order that maximizes data reuse between consecutive tasks. By grouping tasks that access common arrays, we create opportunities for memory savings that Phase 2 can exploit. The output is a valid topological ordering of the task dependency graph that prioritizes data locality.

3) Phase 2: Array Migration Planning: Given the fixed task execution order from Phase 1, this phase determines when to prefetch arrays to GPU memory and when to offload them back to CPU memory. We formulate this as a Mixed Integer Programming (MIP) problem that minimizes peak memory usage while respecting data dependencies. The phase supports multiple optimization objectives, allowing users to prioritize either memory footprint or runtime performance.

B. Phase 1: Task Execution Scheduling

1) Problem Definition & Metric: Given a task dependency graph $G = (V_{task}, E_{dep})$, we seek a topological ordering that maximizes data reuse between nearby tasks. This reduces array migrations by keeping frequently-accessed data in GPU memory.

We define a reuse metric considering both immediate and second-order task (tasks separated by one intervening task) relationships:

$$R(\sigma) = \sum_{i=2}^{n} \sum_{a_k \in \mathcal{D}(\sigma(i-1)) \cap \mathcal{D}(\sigma(i))} |a_k| + \frac{1}{2} \sum_{i=3}^{n} \sum_{a_k \in \mathcal{D}(\sigma(i-2)) \cap \mathcal{D}(\sigma(i))} |a_k| \quad (9)$$

where $\mathcal{D}(t_i) = \mathcal{I}(t_i) \cup \mathcal{O}(t_i)$ represents all arrays accessed by task t_i. The first term captures immediate reuse between adjacent tasks, while the second term (weighted by 0.5) rewards data persistence across one intervening task.

2) Schedule Finding Methods: We employ Beam Search (Algorithm 1 in Appendix A) to find valid topological orderings [43], [44], [45]:

The algorithm maintains the top K promising partial schedules at each level, expanding only the most promising candidates based on accumulated data reuse. When $K = \infty$, beam search performs exhaustive enumeration finding the optimal solution (i.e. becomes breadth-first search).

3) Complexity Analysis: The computational complexity of beam search varies by method parameter: $O(n \cdot K \cdot b)$ where b is the average branching factor (typically $|E_{dep}|/n$). The beam width K provides a tunable trade-off between solution quality and computation time, enabling FRUGAL to handle graphs of varying sizes within practical time constraints. As shown in Appendix B-A, $K = 100$ provides a reasonably good candidate with acceptable preprocessing time.

C. Phase 2: Array Migration Planning

1) Problem Definition: Given the fixed task execution order σ from Phase 1, array sizes, and task runtimes, we determine when to migrate arrays between CPU and GPU memory. We formulate the array migration problem as the Mixed Integer Program shown in Figure 2.

2) Solution Method (MIP Formulation): We formulate array migration as the Mixed Integer Program shown in Figure 2.

a) Decision Variables: Our model uses two types of binary decision variables:

- $p^k_{i,j} \in \{0, 1\}$: Whether to initiate prefetch of array a_k before task t_i until task t_j. We only create variables when migration is actually launched, with completion timing enforced through dependency constraints (when an array is needed).
- $o^k_{i,j} \in \{0, 1\}$: Whether to offload array a_k after task t_i until before task t_j, with initialization timing enforced through dependency constraints (when an array is used).

Note that in practice we simplify from paired variables $p^k_{i,j}/o^k_{i,j}$ to single-index p^k_i/o^k_j, as constraints already enforce completion and starting timing respectively. This eliminates redundant variables without loss of expressiveness. But in this section we keep the pair to simplify the explanation.

b) Objective Function: We minimize a weighted combination of two metrics:

$$\text{Minimize: } \alpha \cdot \text{Time} + \beta \cdot \text{Mig} \quad (11)$$

These metrics are explained as follows:

- **Time**: Total execution time including migration overhead (see Eq. (6)).

Minimize

$$\alpha \cdot \text{Time} + \beta \cdot \text{Mig} \quad (10)$$

Subject to

$$\forall j \in [1, n], k \in [1, m] : \sum_{i=0}^{j-1} p^k_{i,j} \leq 1 \quad \text{(C1.1)}$$

$$\forall i \in [1, n], k \in [1, m] : \sum_{j=i+1}^{n} o^k_{i,j} \leq 1 \quad \text{(C1.2)}$$

$$\forall i \in [1, n], a_k \in \mathcal{D}(t_i) : y^k_i = 1 \quad \text{(C2)}$$

$$\forall i \in [0, n-1], \forall j \in [i+1, n], a_k \notin \mathcal{D}(t_j) : p^k_{i,j} = 0 \quad \text{(C3.1)}$$

$$\forall i \in [1, n], \forall j \in [i+1, n], a_k \notin \mathcal{D}(t_i) : o^k_{i,j} = 0 \quad \text{(C3.2)}$$

$$M_{\text{min}} \leq \text{Mem} \leq M_{\text{max}} \quad \text{(C4)}$$

$$\text{Time} \leq T \quad \text{(C5)}$$

$$L(s_0) = 0 \quad \text{(C6)}$$

Parameters

n	Number of tasks
m	Number of arrays
M_{max}	Memory upper bound (user-specified, optional)
T	Execution time bound (user-specified, optional)
$w(t_i)$	Execution time of task t_i
$w(p^k)$	Execution time of prefetching array a_k
$w(o^k)$	Execution time of offloading array a_k
$\|a_k\|$	Size of array a_k
$\mathcal{D}(t_i)$	Arrays accessed by task t_i
α, β	Objective weights

Decision Variables

$p^k_{i,j}$	Binary: prefetch a_k after the begin of task t_i, complete before t_j
$o^k_{i,j}$	Binary: offload a_k after task t_i, complete before the begin of t_j

Note: All arrays initially reside in CPU memory

Derived Variables

x^k_i	$= \sum_{\substack{u \in [0,n-1], v \in [1,n] \\ v \leq i}} p^k_{u,v} - \sum_{\substack{u \in [1,n-1], v \in [1,n] \\ u < i, v \geq i}} o^k_{u,v}$ (Binary: array a_k in GPU memory at start of t_i)
y^k_i	$= x^k_i \wedge \neg(\exists j < i : o^k_{j,i} = 1)$ (Binary: array a_k accessible at start of t_i, not undergoing offloading)
$L(v)$	Earliest start time of node v in execution graph
M_{min}	$= \max_{i \in [1,n]} \sum_{a_k \in \mathcal{D}(t_i)} \|a_k\|$: Theoretical lower bound
Mem	$= \max_{i \in [1,n]} \sum_{k=1}^{m} \|a_k\| \cdot x^k_i$: Peak memory footprint

Objective Components

Time	$= L(s_{n+1})$ (total execution time)
Mig	$= \sum_{i,j,k} p^k_{i,j} + \sum_{i,j,k} o^k_{i,j}$ (migration count)

Fig. 2: Mixed Integer Program for array migration planning. Given fixed task order from Phase 1, determines optimal prefetch/offload schedule.

- **Mig**: Number of migration operations ($\sum_{ijk} p^k_{i,j} + \sum_{ijk} o^k_{i,j}$).

The weights α and β trade off runtime against migration count. The migration penalty helps avoid excessive data movement that could cause performance overhead not captured by the model.

We support two approaches for multi-objective optimization:

Weighted optimization: Users tune α, β to achieve desired trade-offs. Setting $\alpha > \beta$ prioritizes runtime reduction, while $\beta > \alpha$ prioritizes minimal memory migration.

Hierarchical optimization: Users specify strict priorities through sequential optimization: first minimize runtime, then minimize migrations (Time $\leq$ T (C5), where T=minimum runtime found). This lexicographic approach guarantees optimal runtime while minimizing migrations within runtime bound.

c) Key Constraints: The formulation enforces:

- **Migration consistency:** For any given task that uses the

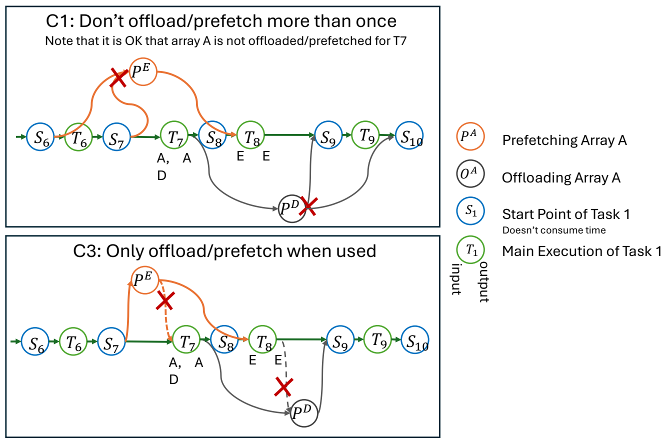

C1: Don't offload/prefetch more than once
Note that it is OK that array A is not offloaded/prefetched for T7
P^E
S_6 T_6 S_7 T_7 S_8 T_8 S_9 T_9 S_{10}
A, A
D
E E
P^D
C3: Only offload/prefetch when used
P^A Prefetching Array A
O^A Offloading Array A
S_1 Start Point of Task 1
Doesn't consume time
T_1 Main Execution of Task 1
input
output

Fig. 3: Visualization of constraints C1 (no duplicate migrations) and C3 (migrate only when used). Invalid operations are marked with $\times$.

arrays, each array can be prefetched at most once and each array can be offloaded at most once (C1, C3), as illustrated in Fig. 3. The same array could be prefetched and offloaded again for another task.

- **Data availability:** Arrays must be accessible in GPU memory when accessed by tasks (C2).
- **Memory bounds:** Peak memory usage must not exceed the user-specified maximum and stay above the theoretical minimum (C4).
- **Temporal ordering:** The execution graph maintains proper task dependencies and migration timing through earliest start time computation (C6, derived variables x_i^k, y_i^k).

Figure 2 presents the complete formulation with all constraints and derived variables.

3) Complexity & Practical Considerations: The MIP formulation is NP-hard with complexity dominated by $O(nm)$ binary decision variables, where migration decisions can span up to n tasks ahead. We further reduce the search space through lookahead window constraints:

Lookahead Window Constraints To further reduce complexity, we limit how far ahead migration decisions can span. Since migrations must synchronize with task execution, we only need to consider decisions within a reasonable horizon:

- **Hard limit**: Migration decisions span at most $S = 30$ tasks, reducing complexity to $O(m)$ variables per task
- **Adaptive limit**: Based on anticipated migration time and execution time. For instance, if offloading all arrays takes time T_{off}, we only consider tasks within $\alpha \cdot T_{off}$ execution time, where $\alpha > 1$ provides a safety margin for estimation errors

This windowing approach exploits the fact that long-range migration decisions are rarely optimal—prefetching too early wastes memory, while offloading too late delays memory release. As shown in Appendix B-B, $S = 30$ and $\alpha = 50$ provide a reasonably good candidate with acceptable preprocessing time. Note that tighter parameters might make the memory reduction constraint infeasible, but do not necessarily influence the runtime of the scheduling result.

VI. Application Analysis

A. Analysis Input - User Input

FRUGAL requires marking task ranges and data dependencies at array granularity as input for application analysis. When tasks correspond to individual kernels, task ranges and data dependencies can often be extracted using existing static analysis tools (e.g., LLVM [46]). However, manual task registration via FRUGAL's API enables grouping multiple kernels into coarser-grained tasks, providing greater flexibility in controlling optimization granularity (e.g., excluding specific kernels from the execution graph or including calls to third-party libraries that would otherwise be omitted).

Arrays are registered through FRUGAL's API:

```
mmg.registerMemory(A); mmg.registerMemory(B);
mmg.registerMemory(C); mmg.registerMemory(X);
mmg.registerMemory(Y); mmg.registerMemory(Z);
```

Tasks are registered with their input and output arrays:

```
tmg.registerTask(kernelA, {A}, {B}, "taskA");
tmg.registerTask(kernelB, {B}, {C}, "taskB");
tmg.registerTask(cublasGemm, {X, Y}, {Z}, "gemm");
```

B. Static Analysis

From the input/output specifications in Section VI-A, FRUGAL automatically constructs the task dependency graph. A dependency edge (t_i, t_j) is added when tasks share arrays with potential data hazards (RAW, WAW, WAR). This automatic dependency extraction works uniformly for custom kernels and library calls. Our compiler analysis can automatically infer input/output sets for direct kernel calls, reducing annotation effort. However, this optimization is not fundamental to FRUGAL's operation—the core system only requires input/output specifications through the API.

C. Offline Profiling

While Phase 1 scheduling requires only dependencies, Phase 2 migration planning needs runtime characteristics. FRUGAL automatically profiles applications using Nvidia CUPTI to capture task execution times $w(t_i)$ for migration planning by inserting marker kernels at task boundaries. It also collects memory footprints for validating FRUGAL's memory reduction. We use microbenchmarks to measure memory transfer costs between CPU and GPU.

D. Hybrid Analysis Benefits

Static analysis provides the dependency structure needed for Phase 1 scheduling, while dynamic profiling supplies the performance metrics essential for Phase 2 migration planning. This separation aligns with our two-phase optimization: structural information guides scheduling, while performance data drives migration decisions.

VII. Implementation

A. CUDA Graph Backend

We use CUDA Graph [47] to execute the optimized memory management plans. CUDA Graph reduce kernel launch overhead for iterative workloads by capturing a sequence of dependent GPU operations into a static Directed Acyclic Graph (DAG) that can be executed with a single driver call. Consequently, CUDA Graphs are widely used across industrial and academic domains, e.g., PyTorch [48], vLLM [49], and GROMACS [50], [51]. Utilizing CUDA Graphs in FRUGAL requires handling several technical issues:

Memory Pool Management. CUDA Graph cannot handle dynamic memory allocation. We implement a memory pool that maps user pointers to pool addresses. When Phase 2 decides to migrate data, we update these mappings while keeping the graph structure static.

Static Scheduling in CUDA Graph. The optimized execution graph from Phases 1 and 2 provides a complete static schedule with three lanes (execution, prefetch, offload). This schedule already incorporates all necessary dependencies, both within lanes and between lanes. We construct the CUDA Graph following this predetermined order, where inter-lane dependencies (e.g., prefetch completing before task execution) are enforced by the static schedule. This eliminates runtime scheduling overhead since all ordering decisions are made during optimization.

B. Offline Profiling Implementation

We construct a CUDA Graph for profiling. To enable fine-grained timing, we insert marker kernels at task boundaries and maintain a mapping between tasks and their corresponding CUDA Graph nodes. During profiling execution, we use CUPTI to capture precise start and end timestamps for each node. We then compute the execution time (maximum span) for each task, accounting for possible intra-task kernel concurrency. This approach provides accurate per-task timing that reflects CUDA Graph execution behavior.

C. Limitations and Further Automation

Our current implementation requires users to explicitly annotate tasks and their associated arrays. Yet FRUGAL can be further automated by utilizing compiler analysis in LLVM [46]. Each kernel can represent a single task, and the compiler analysis can be used to obtain memory effects and deduce input and output arrays.

VIII. Evaluation

A. Basic Evaluation Setting

We evaluate FRUGAL on the NVIDIA GH200 Grace Hopper Superchip [52], featuring 96 GB of HBM3 memory with 4 TB/s bandwidth. The CPU-GPU interconnect provides asymmetric bandwidth: 450 GB/s theoretical and 381 GB/s measured (CPU to GPU) via microbenchmarking. We use Gurobi [53] v11.0.2 to solve the MIP formulation for migration planning.

1) Evaluated Applications: We evaluate FRUGAL on four representative applications that demonstrate seamless integration with the CUDA ecosystem:

❶ **Tiled Cholesky Decomposition (TCD)** and ❷ **Tiled LU Decomposition (TLD)**: These fundamental linear algebra routines leverage cuBLAS and cuSOLVER libraries, representing the large class of scientific computing applications built on NVIDIA's optimized math libraries.

❸ **Tiny-CUDA-NN (TCNN)** [54]: A neural network framework utilizing CUTLASS for tensor operations, demonstrating FRUGAL's compatibility with modern ML workloads and template-based CUDA libraries.

❹ **QuEST** [1]: A quantum simulation framework where we optimize the Quantum Fourier Transform (QFT) routine [55], showcasing FRUGAL's applicability to emerging computational domains.

These applications represent a key strength of FRUGAL: it operates as a drop-in solution for production codebases that rely heavily on CUDA libraries (cuBLAS, cuSOLVER, CUTLASS). Unlike approaches requiring code rewriting or custom kernels [56], [57], FRUGAL preserves existing library calls and optimizations while reducing the memory footprint. This compatibility with the established CUDA ecosystem (from legacy scientific codes to cutting-edge ML frameworks) demonstrates FRUGAL's practical deployability without disrupting existing development workflows.

B. Reducing Memory Footprint

We first conduct a comprehensive comparison of FRUGAL with selected schemes on GH200.

1) Baselines: FRUGAL is a framework that automatically applies memory swapping to reduce memory footprint. Though not directly optimizing for the same goal, we compare FRUGAL with these out-of-core solutions:

- **Default [in-core GPU]**: Original CUDA implementation, with data initialized in GPU.
- **CUDA UM [out-of-core]**: Using Unified Memory, a page-faulting solution provided by the vendor.
- **CPU**: Optimized CPU implementations using nvPL [58] for **TCD/TLD**
- **FRUGAL [out-of-core]**: Our proposed approach. We set a limit on max memory usage and set the objective to be *Runtime-First*

In the [in-core] scenario, we assume that all data resides on the GPU. We ensure this by calling cudaMemCopy before launching cudaGraph. In the [out-of-core] scenario, we assume that all data resides on the CPU initially.

We also examined the ATS [59] feature provided by the GH200 system; however, its out-of-core performance was suboptimal (e.g., TCD achieved 0.57 TFLOPS and TLD achieved 0.59 TFLOPS). By comparison, the CPU can exceed 2 TFLOPS (2.1 TFLOPS for TLD and 2.3 TFLOPS for TCD). Additionally, we also examined other supposedly general solutions targeting GPUs [56], [57], however they require intrusive kernel modifications that are incompatible with production CUDA ecosystem libraries. We document the required intrusive kernel

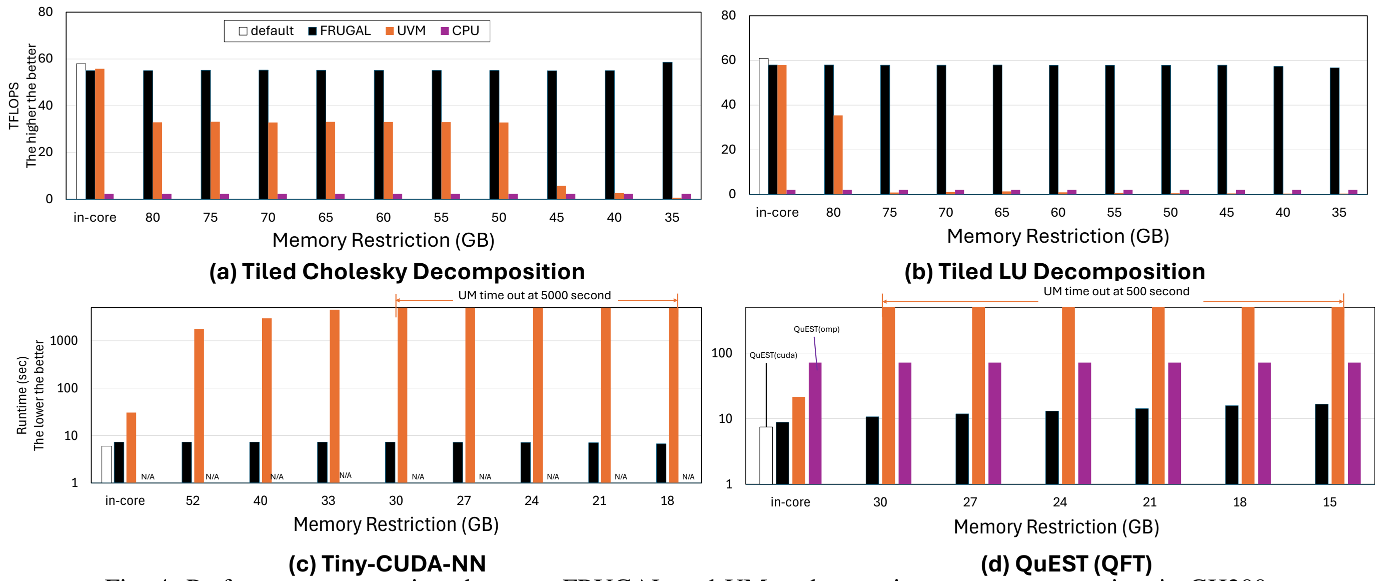


Fig. 4: Performance comparison between FRUGAL and UM under varying memory constraints in GH200.

modifications in the artifact description appendix. Additionally, although many data prefetching memory hierarchy related efforts exist, most of them are restricted to specific hardware and are not based on CUDA, e.g. Persistent Memory [60], [61], [19], [20], [23], [62], CXL [22], [63], Xeon Phi [64], distributed memory [65], or do not provide an implementation [21].

2) Performance Analysis: Fig. 4 presents a performance comparison between FRUGAL and UM under varying memory constraints. In both TCD and TLD, UM exhibits a noticeable performance degradation, likely due to additional overhead from the runtime system occupying GPU memory. As memory constraints become tighter, we observe an immediate performance drop in TLD, TCNN and QFT, while TCD begins to further degrade around 45 GB. This difference arises because tiled Cholesky only requires the lower (or upper) triangular part of a symmetric matrix. Note that when the memory necessary for the solver is more than the memory restriction, UM performance is worse than CPU counterparts, making it unprofitable to utilize the GPU anymore. In contrast, FRUGAL demonstrates remarkable efficiency across all applications. For TCD, TLD, and TCNN, performance degradation remains within 12% of the in-core GPU in-core baseline over all evaluation settings in this evaluation. QuEST shows higher degradation at 2.3× slower at its minimum configuration (15 GB), but this still maintaining 4.3× better performance than the CPU OpenMP implementation.

These results demonstrate FRUGAL's superiority over both UM and CPU-based solutions across diverse applications, demonstrating its generality and performance benefits.

C. Theoretical and Practical Upper-bound on Memory Optimization

While Section VIII-B demonstrates FRUGAL's effectiveness in preserving performance, this section establishes the theoretical and practical upper-bound of memory optimization by setting $Mem = M_{\min}$. Note that since FRUGAL's optimization granularity is at the array level, the theoretical minimum memory footprint equals the maximum memory required by any task (as Constraint C4 shows).

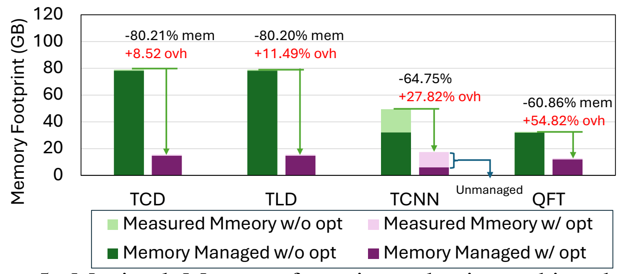


Fig. 5: Maximal Memory footprint reduction achieved by FRUGAL and its corresponding overhead (comparing with in-core).

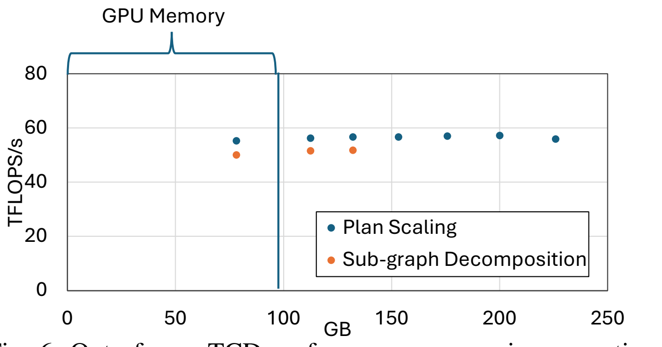


Fig. 6: Out-of-core TCD performance: comparing execution plan scaling versus subgraph decomposition approaches.

However, practical measurements include system memory overheads from the runtime, library calls, and unmanaged memory, resulting in measured memory usage exceeding this theoretical minimum.

As shown in Fig. 5, FRUGAL achieves significant memory reduction even considering the unmanaged system memory across all benchmarks.

D. Case Studies: Beyond Device Memory Capacity

Having validated FRUGAL's effectiveness for performance preservation and memory reduction, this sections explore

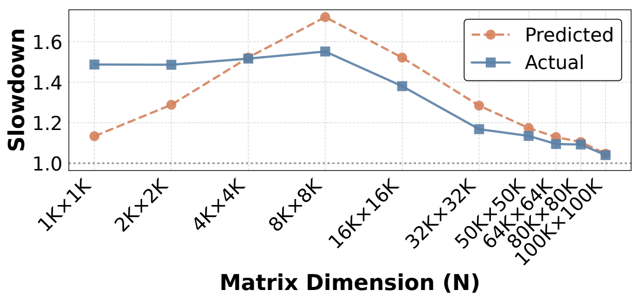


Fig. 7: FRUGAL's predicted vs. actual slowdown across matrix dimensions in TCD (baseline: in-core).

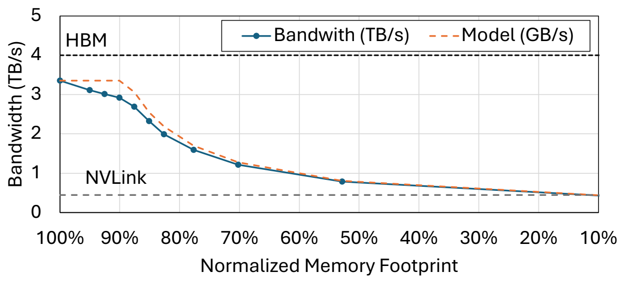


Fig. 8: FRUGAL's performance model predicted vs. actual memory bandwidth for STREAM kernel under varying migration pressure.

specific use cases that capitalize on these memory savings:

1) Case 1: Scaling-up Pre-Optimized Execution Plans: For applications with regular structure, we can profile on smaller inputs and scale the execution plan to larger problems. We demonstrate this with TCD, profiling at 128 MB and scaling up to 200 GB—over 2× the device capacity.

Method: Profile TCD with a 4096×4096 matrix (128 MB), generate the optimized migration schedule, then apply the same pattern to larger matrices by replicating the tile-level schedule.

Preprocessing Time: Initial profiling and optimization requires 0.5–0.9 seconds for the 128 MB problem. This one-time cost amortizes across all subsequent executions at any scale. For perspective, TCD execution at $174,080 \times 174,080$ ($\approx$ 220 GB) takes 31 seconds total—even if preprocessing were included in runtime, it would represent less than 3% overhead.

Results: Figure 6 shows TCD maintaining 55 TFLOPS/s up to 200 GB problem sizes (2× device capacity) with roughly 5% performance degradation compared to hypothetical in-memory execution (57 TFLOPS/s).

2) Case 2: Subgraph Decomposition for Variable Structures: When task graphs change with input size—common in quantum simulation and deep learning—we decompose large problems into memory-fitting subgraphs.

Method: Partition the computation into sequential subgraphs, optimize each independently with FRUGAL, then execute with intermediate state checkpointing.

Preprocessing Time: Preprocessing time scale with the problem size. For a 130 GB problem size, total preprocessing time is approximately 30 seconds.

Results: Using the same TCD benchmark for comparison, subgraph decomposition enables execution up to $\approx$ 130 GB, though with 10% additional runtime overhead from subgraph coordination.

These case studies demonstrate FRUGAL's ability to execute arbitrarily large problems, with preprocessing costs that amortize over multiple runs and long execution times.

E. Performance Assumptions: Sensitivity and Applicability

This section validates the two key assumptions that make FRUGAL's search space tractable.

1) Task Saturation Assumption: We evaluate TCD with FRUGAL across matrix dimensions from $1K \times 1K$ to $100K \times 100K$. Fig. 7 compares FRUGAL's predicted slowdown against actual measurements (relative to in-core baseline) when minimizing runtime under minimal memory constraints.

For small domains ($1K$–$4K$), predictions underestimate slowdown. At $8K \times 8K$, predictions transition to overestimating slowdown. Beyond $16K$ (2 GB), predictions converge with actual performance due to abundant parallelism that saturates the device. In other words, FRUGAL's performance model and actual slowdown converge when the domain size exceeds 2 GB, which we consider a reasonably large problem size for GPU workloads.

2) Task-Migration Independence Assumption: We evaluate the independence assumption using STREAM (performing dot product reductions) at 80 GB problem size (where the saturation assumption holds) while varying memory residency to control migration intensity.

Fig. 8 shows predicted vs. actual bandwidth as memory footprint decreases through migration (x-axis: 100% = no migration, lower = more migration). Reference lines show HBM bandwidth (4 TB/s, upper bound) and NVLink bandwidth (0.45 TB/s, migration capacity).

At 100% memory footprint, the kernel achieves near peak HBM bandwidth. As migration increases, FRUGAL's model (dashed orange) predicts bandwidth degradation assuming complete independence. Actual bandwidth (solid blue) tracks predictions closely with a maximum gap of 13%, indicating the model provides a conservative estimation. Note that in this benchmark, there is no offloading, so the observed maximum prediction error is half of what we anticipated in Section IV-C3.

3) Summary: While both assumptions show systematic deviations under certain conditions—saturation bias at small scales and independence assumption overestimation for memory-bound kernels—FRUGAL's predictions remain sufficiently accurate for guiding optimization decisions by capturing performance trends across different problem configurations.

F. FRUGAL System Design Verification

This subsection validates FRUGAL's key design choices through systematic analysis. We evaluate TCD with a $100K \times 100K$ matrix, optimizing for minimal runtime under minimal memory constraints. Using profiled execution data, we generate and evaluate the top-100 scheduling solutions ranked by FRUGAL's data reuse score, then comparing predicted and actual execution times as shown in Fig. 9. Three key findings emerge:

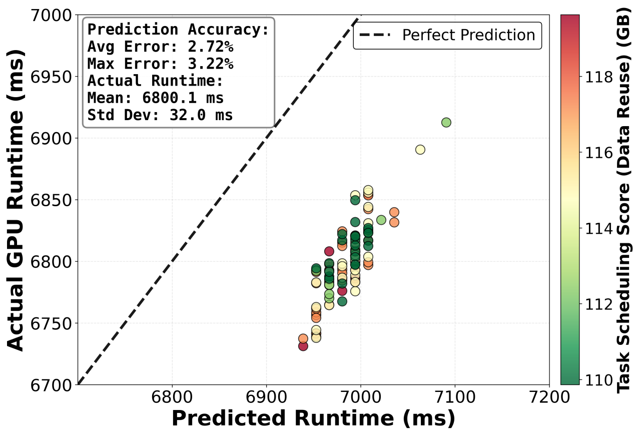


Fig. 9: Predicted vs. actual execution time for top-100 task schedules.

TABLE I: Preprocessing time (seconds) for memory-first optimization

Application	Tasks	Profiling (s)	CUDA Graph (s)	Optimization (s)
TCD	20	12.97	0.006	1.01
TLU	30	23.54	0.009	1.30
TCNN	32	1.25	0.002	2.67
QuEST	1152	3.59	10.43	7.51

- Data reuse scoring does not directly correlate with final runtime performance. However, schedules with lower data reuse (90 GB vs. FRUGAL's 110+ GB) result in worse memory optimization: the MIP solver achieves only 35 GB minimum memory instead of 15 GB, showing that data reuse is critical for memory optimization.
- FRUGAL's performance model accurately predicts actual execution time across all top-100 solutions, validating the model's effectiveness for ranking schedules.
- Performance variance among top solutions is small (mean execution time: 6.80 s, std dev: 32.0 ms), indicating FRUGAL's optimization consistently identifies near-optimal regions.

G. Preprocessing

Table I shows FRUGAL's preprocessing costs, which consist of profiling and optimization phases. CUDA graph construction (<0.1 seconds) is negligible, except for QuEST when the number of tasks is large.

The optimization phase scales well with task count. Applications with up to 32 tasks require 1-3 seconds, while QuEST with 1,152 tasks requires only 7.5 seconds through subgraph decomposition (partitioning into approximately 30-task chunks). This demonstrates that even thousand-task applications can be optimized in under 10 seconds. Profiling time correlates with application runtime rather than task count—TLU requires 23.5 seconds due to its computational intensity, while QuEST needs only 3.6 seconds despite having 40x more tasks.

Importantly, this preprocessing is a one-time cost that amortizes across multiple executions. Even when included in the first run, FRUGAL with preprocessing remains significantly faster than UM's steady-state performance. For instance, QuEST with FRUGAL completes in 37 seconds (including preprocessing) versus UM's 500+ seconds at constrained memory—a more than 13.5× advantage on the first run.

IX. Related Work

Algorithm-Specific Approaches: Many solutions rely on domain-specific out-of-core strategies that prefetch data on-demand using knowledge of the target application. For example, Deep learning frameworks overlap computation with data transfers [2], [3], while techniques for stencils [9], [11], matrix multiplication [12], and graph processing [13] reduce memory usage similarly. These methods can excel in performance but require specialized expertise and often lack generality.

System-Level Approaches: System-level solutions, e.g. CUDA Unified Memory (UM) or AMD SVM, strive for transparency and automatic data management between heterogeneous memory systems. They allow developers to treat CPU and GPU memory as unified but often incur runtime overhead and offer limited performance predictability due to on-demand, page-fault-driven migrations [17], [18]. More advanced frameworks incorporate profiling or performance counters (e.g., TPP [22] and MEMTIS [23]) to optimize placements but not leveraging application-specific structure might limit their performance.

Middle-Ground Approaches: Some efforts seek a compromise by integrating partial application insights while remaining largely automatic. For example, CachedArrays [61] uses static policies plus user annotations, and TrackFM [65] employs compiler analysis for remote memory. Merchandiser [60] combines compiler insights with modeling but targets CPU–persistent memory. These "middle-ground" methods can outperform purely system-level approaches.

Summary and Positioning: FRUGAL, our proposed approach, strategically integrates system-level transparency with targeted profiling and dependency analysis techniques. This balanced methodology achieves substantial memory savings and performance improvements without necessitating extensive domain-specific code modifications, positioning it as an effective compromise between generality and performance.

X. Conclusion

We present FRUGAL, a framework that reduces the GPU memory requirement of general GPU applications by rescheduling kernels and injecting data migration to effectively offload data to the CPU, then prefetch the data back to the GPU when necessary. FRUGAL can greatly reduce the peak memory usage of several widely used workloads with mild overhead. Evaluations on Tiled Cholesky Decomposition, Tiled LU Decomposition, Tiny-CUDA-NN, and QuEST show that FRUGAL significantly reduces maximum GPU memory usage by 80.21%, 80.20%, 64.75% and 60.86% with only a slowdown of 8.52%, 11.49%, 27.82% and 54.82% respectively. Using FRUGAL, we also managed to run the Tiled Cholesky Decomposition of a matrix on NVIDIA GH200 96GB with only 3.5% performance overhead, which originally requires 225.78 GB GPU memory on NVIDIA GH200.

Acknowledgments

This project was supported by JSPS KAKENHI Grant Number JP22H03600 and JP25K15142, JST SPRING JPMJSP2180, and the RIKEN Junior Research Associate Program.

References

[1] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Scientific reports*, vol. 9, no. 1, p. 10736, 2019.

[2] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[3] M. Wahib, H. Zhang, T. T. Nguyen, A. Drozd, J. Domke, L. Zhang, R. Takano, and S. Matsuoka, "Scaling distributed deep learning workloads beyond the memory capacity with karma," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2020.

[4] T. P. Morgan, "He who can pay top dollar for hbm memory controls ai training," 2024.

[5] V. LLC, "Nvidia a100 enterprise pcie 40gb/80gb — vipera - tomorrow's technology today."

[6] H. Zhou, W. Rang, H. Chen, X. Zhou, and D. Cheng, "Deeptm: Efficient tensor management in heterogeneous memory for dnn training," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–17, 2024.

[7] Z. Hu, J. Xiao, Z. Deng, M. Li, K. Zhang, X. Zhang, K. Meng, N. Sun, and G. Tan, "Megtaichi: dynamic tensor-based memory management optimization for dnn training," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[8] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimization towards training A trillion parameter models," *CoRR*, vol. abs/1910.02054, 2019.

[9] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki, "A stencil framework to realize large-scale computations beyond device memory capacity on gpu supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 525–529, 2017.

[10] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Beyond 16gb: Out-of-core stencil computations," in *Proceedings of the Workshop on Memory Centric Programming for HPC*, MCHPC'17, (New York, NY, USA), p. 20–29, Association for Computing Machinery, 2017.

[11] J. Shen, L. Long, X. Deng, M. Okita, and F. Ino, "A compression-based memory-efficient optimization for out-of-core gpu stencil computation," *The Journal of Supercomputing*, vol. 79, no. 10, pp. 11055–11077, 2023.

[12] O. Patil, F. Mueller, L. Ionkov, J. Lee, and M. Lang, "Pears: A performance-aware static and dynamic framework for heterogeneous memory," 2021.

[13] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[14] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[15] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, (New York, NY, USA), p. 318–333, Association for Computing Machinery, 2021.

[16] J. Liu, D. Li, R. Gioiosa, and J. Li, "Athena: high-performance sparse tensor contraction sequence on heterogeneous memory," in *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, (New York, NY, USA), p. 190–202, Association for Computing Machinery, 2021.

[17] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for gpu accelerated computing," SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[18] B. Cooper, T. R. Scogland, and R. Ge, "Shared virtual memory: Its design and performance implications for diverse applications," in *Proceedings of the 38th ACM International Conference on Supercomputing*, ICS '24, (New York, NY, USA), p. 26–37, Association for Computing Machinery, 2024.

[19] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: Breaking gpu memory capacity limits with direct nvm access," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 414–426, 2018.

[20] Y. Liu, Y. Ren, M. Liu, H. Li, H. Guo, X. Miao, X. Hu, and H. Chen, "Optimizing file systems on heterogeneous memory by integrating DRAM cache with virtual memory management," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, (Santa Clara, CA), pp. 71–87, USENIX Association, Feb. 2024.

[21] R. Das, K. Agrawal, M. A. Bender, J. Berry, B. Moseley, and C. A. Phillips, "How to manage high-bandwidth memory automatically," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, (New York, NY, USA), p. 187–199, Association for Computing Machinery, 2020.

[22] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 742–755, 2023.

[23] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 17–34, 2023.

[24] D. Xu, J. Ryu, K. Shin, P. Su, and D. Li, "FlexMem: Adaptive page profiling and migration for tiered memory," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, (Santa Clara, CA), pp. 817–833, USENIX Association, July 2024.

[25] S. Zhao, F. Li, X. Chen, T. Shen, L. Chen, S. Wang, N. Zhang, C. Li, and H. Cui, "Naspipe: high performance and reproducible pipeline parallel supernet training via causal synchronous parallelism," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 374–387, Association for Computing Machinery, 2022.

[26] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel gpu task scheduling for deep learning," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 8343–8354, Curran Associates, Inc., 2020.

[27] H. Oh, J. Lee, H. Kim, and J. Seo, "Out-of-order backprop: an effective scheduling technique for deep learning," in *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, (New York, NY, USA), p. 435–452, Association for Computing Machinery, 2022.

[28] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[29] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, "Welder: Scheduling deep learning memory access via tile-graph," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, (Boston, MA), pp. 701–718, USENIX Association, July 2023.

[30] O. Kyriienko, "Quantum inverse iteration algorithm for programmable quantum simulators," *npj Quantum Information*, vol. 6, no. 1, p. 7, 2020.

[31] I. M. Georgescu, S. Ashhab, and F. Nori, "Quantum simulation," *Reviews of Modern Physics*, vol. 86, no. 1, pp. 153–185, 2014.

[32] Z. Wu, Y. Wu, Y. Liu, H. Shang, Y. Gao, Z. Zhang, Y. Zhang, Y. Long, X. Feng, and H. Cui, "Portable and scalable all-electron quantum perturbation simulations on exascale supercomputers," in *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, (Los Alamitos, CA, USA), pp. 1–14, IEEE Computer Society, nov 2023.

[33] T. Ben-Nun, L. Groner, F. Deconinck, T. Wicky, E. Davis, J. Dahm, O. D. Elbert, R. George, J. McGibbon, L. Trümper, *et al.*, "Productive performance engineering for weather and climate modeling with python," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2022.

[34] S. V. Adams, R. W. Ford, M. Hambley, J. Hobson, I. Kavčič, C. M. Maynard, T. Melvin, E. H. Müller, S. Mullerworth, A. R. Porter, *et al.*, "Lfric: Meeting the challenges of scalability and performance portability in weather and climate models," *Journal of Parallel and Distributed Computing*, vol. 132, pp. 383–396, 2019.

[35] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation,"

ACM Transactions on Architecture and Code Optimization (TACO), vol. 18, no. 4, pp. 1–23, 2021.

[36] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez, "Profiles of upcoming hpc applications and their impact on reservation strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1178–1190, 2021.

[37] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*, (Berlin, Heidelberg), p. 355–373, Springer-Verlag, 2017.

[38] M. Wahib and N. Maruyama, "Automated gpu kernel transformations in large-scale production stencil applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, (New York, NY, USA), p. 259–270, Association for Computing Machinery, 2015.

[39] L. Zhang, M. Wahib, P. Chen, J. Meng, X. Wang, T. Endo, and S. Matsuoka, "Perks: a locality-optimized execution model for iterative memory-bound gpu applications," in *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, (New York, NY, USA), p. 167–179, Association for Computing Machinery, 2023.

[40] L. Zhang, M. Wahib, P. Chen, J. Meng, X. Wang, T. Endo, and S. Matsuoka, "Revisiting temporal blocking stencil optimizations," in *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, (New York, NY, USA), p. 251–263, Association for Computing Machinery, 2023.

[41] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10, p. 16, San Jose, CA, 2010.

[42] V. Volkov, *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[43] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," *SIGPLAN Not.*, vol. 26, p. 241–255, May 1991.

[44] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, p. 406–471, Dec. 1999.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, p. 319–349, July 1987.

[46] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[47] N. Corporation, "CUDA C++ Programming Guidr." https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2025. Accessed: 09-10-2025.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[49] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, (New York, NY, USA), p. 611–626, Association for Computing Machinery, 2023.

[50] J. Witsoe, "A Guide to CUDA Graphs in GROMACS 2023." NVIDIA Technical Blog, 2025. Accessed: 09-10-2025.

[51] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen, "Gromacs: fast, flexible, and free," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1701–1718, 2005.

[52] NVIDIA Corporation, "Nvidia Grace Hopper Superchip architecture white paper," tech. rep., NVIDIA, 2024. White paper.

[53] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024.

[54] T. Müller, "tiny-cuda-nn," 2021.

[55] D. Coppersmith, "An approximate fourier transform useful in quantum factoring," 2002.

[56] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "Gmt: Gpu orchestrated memory tiering for the big data era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, (New York, NY, USA), p. 464–478, Association for Computing Machinery, 2024.

[57] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W.-m. Hwu, "Gpu-initiated on-demand high-throughput storage access in the bam system architecture," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, (New York, NY, USA), p. 325–339, Association for Computing Machinery, 2023.

[58] "Nvidia nvpl." https://developer.nvidia.com/nvpl. Accessed: YYYY-MM-DD.

[59] A. Ltd., *Address Translation Services*. Arm Developer, 2025. Accessed: 09-10-2025.

[60] Z. Xie, J. Liu, J. Li, and D. Li, "Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 204–217, 2023.

[61] M. Hildebrand, J. Lowe-Power, and V. Akella, "CachedArrays: Optimizing data movement for heterogeneous memory systems," in *Proceeding of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.

[62] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "Panthera: Holistic memory management for big data processing over hybrid memories," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 347–362, 2019.

[63] M. Arif, A. Maurya, M. M. Rafique, D. S. Nikolopoulos, and A. R. Butt, "Application-attuned memory management for containerized hpc workflows," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 114–127, 2024.

[64] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Rthms: a tool for data placement on hybrid memory system," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, (New York, NY, USA), p. 82–91, Association for Computing Machinery, 2017.

[65] B. R. Tauro, B. Suchy, S. Campanoni, P. Dinda, and K. C. Hale, "Trackfm: Far-out compiler support for a far memory world," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, (New York, NY, USA), p. 401–419, Association for Computing Machinery, 2024.

Artifact Description

XII. Overview of Contributions and Artifacts

A. Paper's Main Contributions

In this paper, we propose a novel middle-ground strategy, *FRUGAL*, that uses a combination of elaborate profiling, static analysis, and lifting the abstraction of the dependency graph to gain a thorough understanding of the application, hence eliminating the need to develop custom solutions for the underlying algorithms of an application. FRUGAL is an offline approach, which means the optimization is done before deploying the application, and does not incur overhead for calculating the optimization plan after deployment. This offline approach particularly targets applications that would be deployed in production to run for long time periods.

B. Computational Artifacts

A_1 Overall comparison with unified memory

Artifact ID	Related Paper Elements
A_1	Fig. 4

XIII. Artifact Identification

The FRUGAL framework is open source at https://github.com/neozhang307/FRUGAL-base. The code checkpoint related to current submission is in CGO26/master branch.

Relationship To Contributions

We prove that FRUGAL has higher performance compared to solutions that only rely on system runtime information.

Expected Results

FRUGAL might not be optimal in the setting of fully in-core. It outperforms system solutions like Unified Memory (UVM).

Expected Reproduction Time (in Minutes)

Executing FRUGAL over the test sets takes 10 minutes. However, executing the baseline Unified Memory implementation can take up to one day.

Artifact Setup

Hardware

GH200, Nvidia Grace CPU.

Software

S_1 VCPKG: https://learn.microsoft.com/zh-cn/vcpkg/get_started/get-started?pivots=shell-bash

S_2 OR-TOOLS v9.11: https://github.com/google/or-tools

S_3 Gurobi: https://www.gurobi.com/

FRUGAL Configuration

FRUGAL's configuration system can accept input either from the command line or from the $config.json$ file. The main configurable parameters related to this Artifact are:

weightOfTotalRunningTime α in the paper. Controls the weight of running time in the final optimization goal.

weightOfNumberOfMigrations β in the paper. Controls the weight of migrations amount in the final optimization goal.

maxPeakMemoryUsageInMiB Hard limit for max memory usage. It might result in not finding any optimization plan.

prefetchingBandwidthInGB Hardware feature of PCIe and nvLink that can be tested by CUDA sample bandwidthTest. (https://github.com/NVIDIA/cuda-samples.git)

Datasets / Inputs

TCD Tiled Cholesky Decomposition, We use a randomly generated 102400×102400 matrix as input. The tiling is set to 16. We use $N^3/3$ for flops evaluation. Source code available under FRUGAL codebase.

TLD Tiled LU Decomposition. We use a randomly generated 102400×102400 matrix as input. The tiling is set to 16. We use $2N^3/3$ for flops evaluation. Source code available under FRUGAL codebase.

TCNN 8 hidden layers with 4096 neurons. The input layer is encoded with a grid-based encoding backed by a hash table. It learns the RGB color of the pixels of a 2D image, having 2 input dimensions (the coordinates) and 3 output dimensions (the RGB channels). The batch size is set to 2^{18}, and the network is trained for 1 steps. The forward and backward propagation between two layers, the encoding, and the loss evaluation are defined as tasks. Original code is in https://github.com/NVlabs/tiny-cuda-nn.

QuEST The original implementation of QuEST for CUDA back-end allocates a large array to store all the state vector amplitudes, which is not suitable for FRUGAL because FRUGAL migrates data at the granularity of an array. Therefore, we implement partitioning of the state vector amplitudes by equally dividing the amplitudes into 2m shards for the CUDA back-end. We also modify the kernels of quantum gates such that a kernel processes the data of a shard instead of all amplitudes, to lower the theoretical minimum peak memory usage. Each modified kernel is defined as a task. We simulate 31 qubits and divide the amplitudes into 16 shards, which brings 1152 tasks in total and makes FRUGAL's MIP problem too complex to be solved within a reasonable amount of time. Thus, we divide the quantum circuit into 38 stages by grouping every two consecutive gates into a stage, optimize the circuit gates by gate, and combine the optimization plans for gates into a complete plan. The original code is available at https://github.com/QuEST-Kit/QuEST.

Baseline Configuration and Setup

We build FRUGAL based on the assumption that CUDA Graph is efficient in production execution. So our GPU baseline setting utilizes CUDA Graph at the same level. As constructing CUDA Graph would execute the program once, we treat this execution as a program warmup. Then we detail the configuration as follows:

B_1 Default: Original CUDA Graph implementation that utilized cuBLAS and cuSolver

B_2 UVM: Unified Memory on the Default implementation.

B_3 CPU: CPU implementation with NVPL, we additionally add warmup run.

B_3.1 CPU Cholesky Decomposition: NVPL sample code https://github.com/NVIDIA/NVPLSamples/tree/main

B_3.2 CPU LU Decomposition: Our NVPL implementation.

Baseline Failed Attempt

We also examined other baselines:

B_4 ATS: Address Translation Services for Grace System. We change cudaMalloc/cudaMallocManaged to malloc. The performance is so bad that we didn't include it in the paper.

B_5 GMT/BaM: The issue is that we need to change the pointer to array_d_t data structure that makes it impossible to align with cublas, cusolver or cutlass based applications. As an example of code modification, GMT https://github.com/lineagech/GMT/blob/main/benchmarks/bfs/main.cu, line 1227; and BaM https://github.com/ZaidQureshi/bam/blob/master/benchmarks/bfs/main.cu line 1075.

Setting Memory Restriction

FRUGAL : We iterate over every possible memory limitation. Then we use total memory usage to represent the total memory usage, including the memory that FRUGAL could not control, e.g., CUDA runtime, CUDA library, as the result reported.

Baseline : We do a dummy allocation to occupy a certain amount of memory to simulate the situation of forcing memory swapping.

Installation and Deployment

VCPKG can be installed through the shell. Then set the environment as follows:

```
export VCPKG_ROOT=[to directory]
alias vcpkg="\${VCPKG_ROOT}/vcpkg"
export VCPKG_FORCE_SYSTEM_BINARIES=1
```

Listing 1: Setting up VPKG.

OR-TOOLs can be pulled from git and compiled. Details are given in https://developers.google.com/optimization/install/cpp/source_linux. Then set the environment as follows:

```
export ORTOOLS_ROOT=[path-to-install]
```

Listing 2: Setting up OR-TOOLs.

Gurobi can be directly downloaded from the website. Though FRUGAL can support other solvers, Gurobi is the fastest one that we tested. A license is necessary to run Gurobi. Then set the environment as follows:

After every requirement of FRUGAL is installed, reference FRUGAL in the application with:

```
export GUROBI_HOME=[path to installation]
export GRB_LICENSE_FILE=[path to license file]
```

Listing 3: Setting up GUROBI.

```
#include "memopt.hpp"
using namespace memopt;
```

Listing 4: Refering FRUGAL in the application.

Artifact Execution

After running *make config*, *make build*, the binary is created, and parameters can be defined at runtime. As follows:

```
make config
make build
```

Listing 5: Command for build.

Appendix A
Beam Search

We use beam search (Algorithm 1) to find valid topological orderings that maximize data reuse. The algorithm maintains the top K partial schedules at each level, expanding only the most promising candidates.

Algorithm 1 Beam Search for Task Scheduling

```
Require: Task dependency graph $G = (V_{task}, E_{dep})$, reuse matrix $M$, beam width $K$
Ensure: Task execution order $\sigma$ maximizing data reuse
 1: beam ← [empty schedule]
 2: for position $i = 1$ to $n$ do
 3:     candidates ← $\emptyset$
 4:     for each partial schedule $s$ in beam do
 5:         for each ready task $t$ (respecting dependencies) do
 6:             score ← $M[s.\text{last}][t]$                    ▷ Immediate reuse
 7:             if $s$ has 2+ tasks then                         ▷ Second-order reuse
 8:                 score ← score $+0.5 \times M[s.\text{second\_last}][t]$
 9:             end if
10:             candidates ← candidates ∪ {($s$ + $t$, score)}
11:         end for
12:     end for
13:     beam ← top $K$ schedules from candidates by total score
14: end forreturn best complete schedule from beam
```

Appendix B
Parameter Sensitivity Study

A. Phase 1: Beam Search Size

TABLE II: Impact of beam width on FRUGAL's optimization quality and overhead for TCD $100K \times 100K$.

Beam Width	Beam Search Time (ms)	Data Reuse Score (GB)	Predicted Runtime (s)	Real Runtime (s)
1	0.021	117.18	6.95	6.74
10	0.148	119.62	6.69	6.76
50	0.560	119.62	**6.94**	**6.71**
60	0.664	119.62	7.01	6.83
100	1.058	119.62	6.98	6.74

Using the same configuration as Section VIII-F, we evaluate FRUGAL's performance across different beam widths, as shown in Table II. The results reveal: (1) beam search overhead remains negligible even at width 100 (1.06 ms), (2) wider beams provide diminishing returns—widths 10–100 provide identical scores, and (3) the performance variance across beam widths is minimal (a maximum 0.12 s difference, or 1.8%). These findings demonstrate that beam width has minimal impact on runtime performance. However, beam search itself remains essential for identifying task orders that enable effective memory reduction.

B. Phase 2: Scheduling Window Size

Using a randomly selected schedule from the top-K results in Section VIII-F, we isolate Phase 2 by fixing the task order and varying only the migration planning parameters. We evaluate two windowing strategies: static window size (distance parameter) and dynamic window scaling (factor parameter), as shown in Table III.

The distance parameter controls the static scheduling window and exhibits stable behavior across all tested values (1–30), with predicted runtimes varying minimally (6.98–6.99 s) and consistently achieving optimal solutions. In contrast, the factor parameter, which controls dynamic window expansion, shows three distinct regimes: (1) infeasible solutions for values 1–20 due to overly restrictive constraints that eliminate all offload variables, (2) optimal convergence at factor 60, achieving best performance (6.94 s), and (3) oscillating behavior for factors 70–100 with runtimes varying between 6.98 and 6.99 s.

These results demonstrate that static windows can be set conservatively tight without performance penalty, while dynamic factors require sufficient headroom ($\geq$ 30) to maintain feasibility under tight memory reduction constraints. Both parameters incur minimal MIP solve time overhead (0.37–0.41 s).

TABLE III: Impact of MIP solver parameters on optimization quality for TCD $100K \times 100K$.

(a) Distance Parameter

Distance	MIP (s) Time (s)	Predicted Runtime (s)	Real Runtime (s)	Status
1	0.37	6.98	6.74	**Best**
5	0.40	6.98	6.78	
10-30	0.38-0.40	6.99	6.77–6.78	Converged

(b) Dynamic Factor Parameter

Factor	MIP Time (s)	Predicted Runtime (s)	Real Runtime (s)	Status
1–10	0.32–0.33	—	—	Infeasible
20	0.37	6.97	6.75	First feasible
30–50	0.38	6.95–6.98	6.74 –6.77	Oscillating
60	0.40	**6.94**	**6.73**	**Best**
70–100	0.39–0.41	6.98–6.99	6.78–6.79	Oscillating

Appendix C
End-to-End Example

In this section, we present an end-to-end example of how FRUGAL optimizes and generates a migration plan. We set Tile=4, resulting in a total of 19 annotated tasks.

We begin with the annotated pseudocode shown in Fig. 10. FRUGAL analyzes this code, extracts the dependencies, and determines an execution order based on data reuse, as shown in Fig. 11. The MIP step then solves the scheduling problem and generates an optimized migration plan, as shown in Fig. 12.

```
// Phase 1: Allocate and register managed arrays
MemoryManager& memManager = MemoryManager::getInstance();
double* A[T][T];  // Tile pointers

for (int i = 0; i < T; i++) {
  for (int j = 0; j <= i; j++) {  // Lower triangular
    cudaMalloc(&A[i][j], B * B * sizeof(double));
    memManager.registerManagedMemory(A[i][j], B * B * sizeof(double));
  }
}

// Phase 2: Register tasks with data dependencies
TaskManager taskManager;

for (int k = 0; k < T; k++) {
  // POTRF: L[k][k] = cholesky(A[k][k])  // cuSOLVER
  taskManager.registerTask(
    POTRF,
    inputs  = {A[k][k]},
    outputs = {A[k][k]}
  );

  // TRSM: L[i][k] = A[i][k] * inv(L[k][k]^T)  // cuBLAS
  for (int i = k+1; i < T; i++) {
    taskManager.registerTask(
      TRSM,
      inputs  = {A[k][k], A[i][k]},
      outputs = {A[i][k]}
    );
  }

  // SYRK: A[i][i] -= L[i][k] * L[i][k]^T  // cuBLAS
  for (int i = k+1; i < T; i++) {
    taskManager.registerTask(
      SYRK,
      inputs  = {A[i][i], A[i][k]},
      outputs = {A[i][i]}
    );
  }

  // GEMM: A[i][j] -= L[i][k] * L[j][k]^T  // cuBLAS
  for (int i = k+1; i < T; i++) {
    for (int j = k+1; j < i; j++) {
      taskManager.registerTask(
        GEMM,
        inputs  = {A[i][j], A[i][k], A[j][k]},
        outputs = {A[i][j]}
      );
    }
  }
}

// Phase 3: Generate CUDA graph from registered tasks  [OFFLINE]
cudaGraph_t graph = taskManager.generateGraph(stream);

// Phase 4: Profile and optimize memory placement  [OFFLINE]
cudaGraphExec_t optimizedExec;
Optimizer::profileAndOptimize(graph, &optimizedExec);

// Phase 5: Execute optimized graph
cudaGraphLaunch(optimizedExec, stream);
cudaStreamSynchronize(stream);
```

Fig. 10: Complete FRUGAL workflow for tiled Cholesky. (1) Allocate tiles and register with memory manager. (2) Register tasks (cuSOLVER/cuBLAS) specifying input/output arrays. FRUGAL records task IDs in the same order they appear in the original program. (3-4) **Offline:** Generate CUDA graph and optimize memory placement. (5) Launch optimized graph.

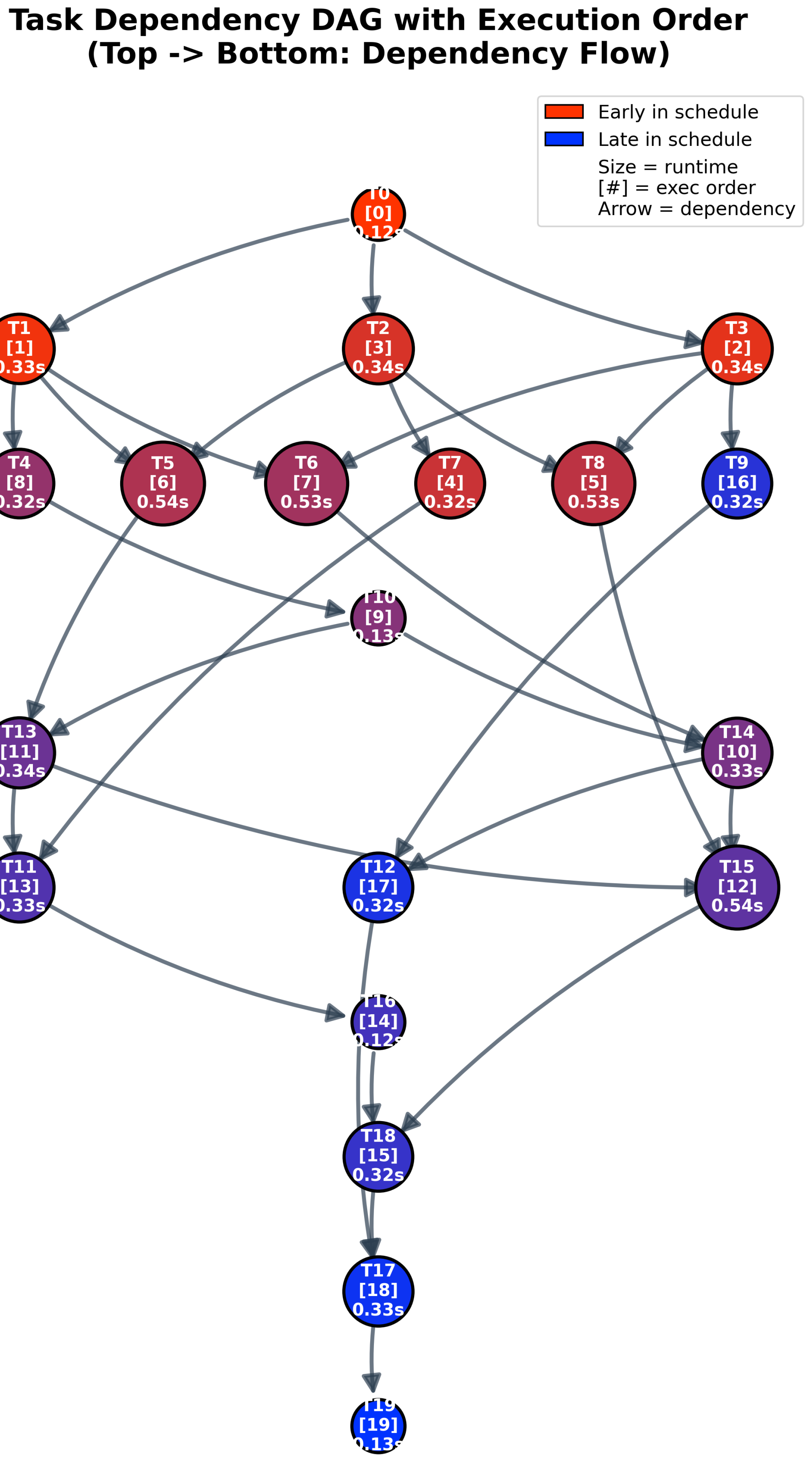


Fig. 11: TCD task dependency and its execution order (phase 1).

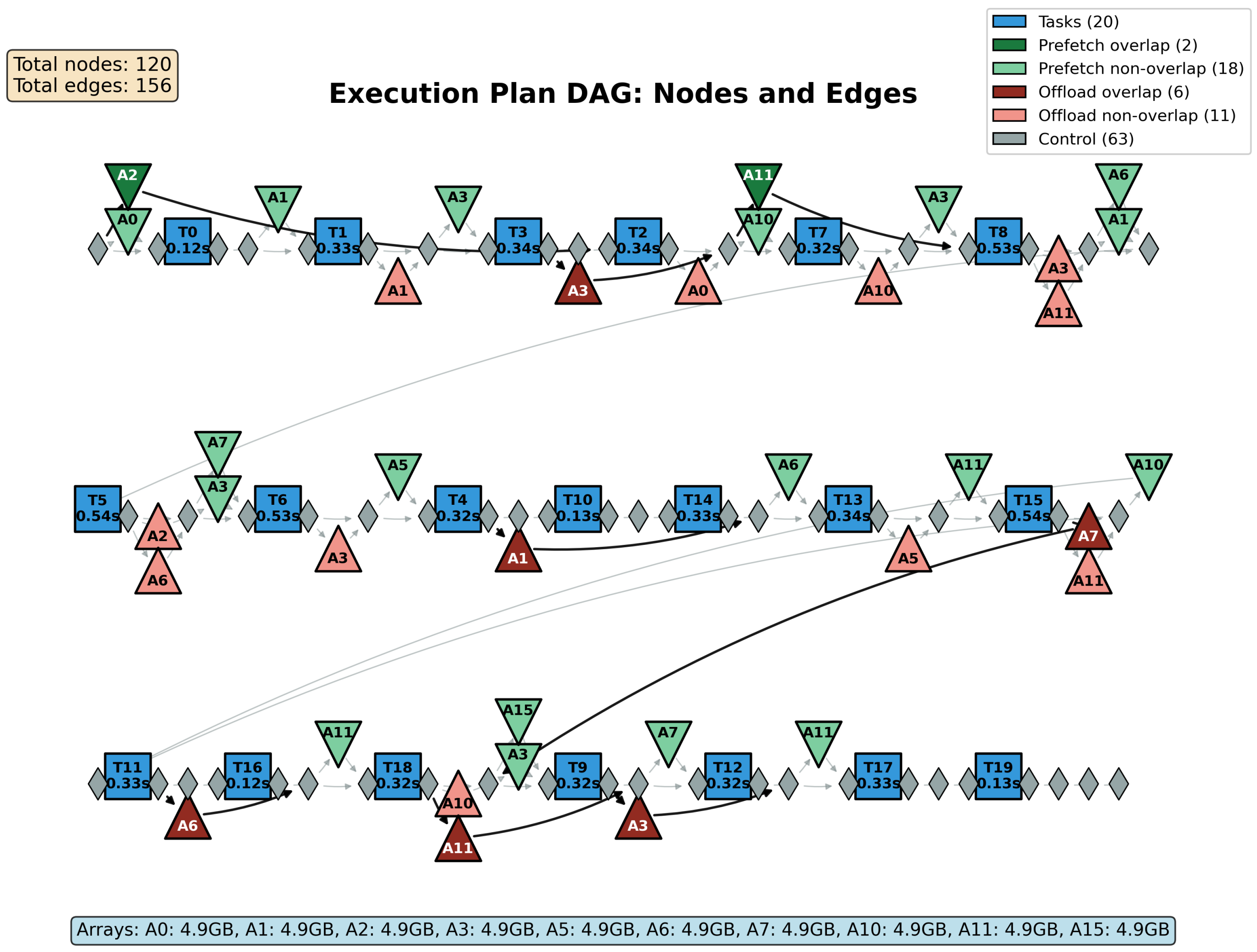


Fig. 12: TCD phase 2 plan