

cuGenOpt: A GPU-Accelerated General-Purpose Metaheuristic Framework for Combinatorial Optimization

Yuyang Liu
Independent Researcher, Shenzhen, China
15251858055@163.com

Abstract

Combinatorial optimization problems arise in logistics, scheduling, and resource allocation, yet existing approaches face a fundamental trade-off among generality, performance, and usability. Exact methods scale poorly beyond moderate sizes; specialized solvers achieve strong results but only on narrow problem classes; CPU-based metaheuristics offer generality but limited throughput. We present **cuGenOpt**, a GPU-accelerated general-purpose metaheuristic framework that addresses all three dimensions simultaneously.

At the *engine* level, cuGenOpt adopts a “one block evolves one solution” CUDA architecture with a unified encoding abstraction (permutation, binary, integer), a two-level adaptive operator selection (AOS) mechanism, and hardware-aware resource management (shared-memory auto-extension, L2-cache-aware population sizing). At the *extensibility* level, a user-defined operator registration interface allows domain experts to inject problem-specific CUDA search operators that participate in AOS weight competition alongside built-in operators, bridging the gap between generality and specialization. At the *usability* level, a JIT compilation pipeline exposes the entire framework as a pure-Python API (`pip install cugenopt`), and an LLM-based modeling assistant converts natural-language problem descriptions into executable solver code without manual CUDA programming.

Experiments across five thematic suites on three GPU architectures (T4, V100, A800) show that cuGenOpt outperforms general MIP solvers by orders of magnitude, achieves competitive quality against specialized solvers on instances up to $n=150$, and attains 4.73% gap on TSP-442 within 30s on A800. Twelve problem types spanning five encoding variants are solved to optimality, and standard benchmarks (QAPLIB, Solomon VRPTW, OR-Library JSP) confirm near-optimal convergence. Framework-level optimizations cumulatively reduce pcb442 gap from 36% to 4.73% and boost VRPTW throughput by 75-81% via shared-memory extension alone.

Code: <https://github.com/L-yang-yang/cugenopt>

Keywords: combinatorial optimization, GPU parallel computing, metaheuristics, adaptive operator selection, JIT compilation, general-purpose solver framework

1 Introduction

Combinatorial optimization problems pervade logistics, scheduling, resource allocation, and numerous other domains. Classical problems such as the Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP), Job-Shop Scheduling (JSP), and Quadratic Assignment Problem (QAP) are NP-hard, and the computational cost of exact solutions grows exponentially with problem size [19].

Current approaches fall into three broad categories, each facing distinct limitations:

1. **Exact methods** (e.g., Mixed-Integer Programming): Guarantee global optimality through mathematical modeling. However, general-purpose MIP formulations such as the MTZ model

for TSP [16] introduce $O(n^2)$ variables; solvers like SCIP [27] often fail to find high-quality feasible solutions within reasonable time when $n > 100$. Moreover, MIP modeling demands operations research expertise, particularly for nonlinear objectives and complex constraints.

2. **Specialized solvers** (e.g., OR-Tools Routing [9], NVIDIA cuOpt [17]): Highly optimized for specific problem classes with excellent performance, but limited in scope—they cannot accommodate custom constraints (e.g., intra-route priority ordering) or nonlinear objective functions, and require ground-up development for new problem types.
3. **Metaheuristics** (e.g., simulated annealing [10], evolutionary algorithms [7]): Highly general, applicable to arbitrary black-box objectives. Traditional CPU implementations, however, offer limited search throughput and converge slowly on large-scale instances. GPU-accelerated metaheuristics [14, 4] achieve significant speedups but are typically designed for a single problem type, lacking a general-purpose framework abstraction.

An underserved space lies at the intersection of these three categories: **combining GPU-accelerated search throughput, general-purpose problem modeling, and a low-barrier user interface**. This paper presents cuGenOpt to fill that gap.

1.1 Overview of cuGenOpt

cuGenOpt is a GPU-accelerated general-purpose metaheuristic framework for combinatorial optimization, organized around three progressive layers:

Core engine (Sections 3–4). The framework adopts a “one block evolves one solution” CUDA parallel architecture supporting three universal encoding types: permutation, binary, and integer. Search strategy is dynamically adjusted through a two-level adaptive operator selection (AOS) mechanism, with problem-profile-driven prior weights accelerating convergence. Hardware-aware mechanisms—shared-memory auto-extension and L2-cache-aware adaptive population sizing—enable the same binary to automatically select optimal memory paths across different GPUs.

Extensibility (Section 3). A user-defined operator registration interface allows domain experts to inject problem-specific CUDA search operators (e.g., delta-evaluation 2-opt for TSP) as code snippets. These operators are JIT-compiled into the framework’s operator system and compete with built-in operators through AOS weight adaptation, providing a specialization channel while preserving framework generality.

User interface (Section 5). A JIT compilation pipeline packages the entire framework as a pure-Python package (`pip install cugenopt`), enabling users to solve built-in problems or define custom problems through code snippets without writing full CUDA programs. An LLM-based modeling assistant further converts natural-language problem descriptions into executable solver code.

1.2 Contributions

1. **General-purpose GPU solver framework**: We propose a “one block evolves one solution” GPU parallel architecture with unified support for permutation, binary, and integer encodings, validated on 12 representative problem types. User modeling effort is typically 20–50 lines of CUDA.
2. **Multi-layer adaptive search with hardware awareness**: We design problem-profile-driven operator prior weights (L1) and two-level EMA-based adaptive operator selection (L3), coupled with shared-memory auto-extension and L2-cache-aware population sizing for joint optimization of search strategy and GPU resources.

3. **Extensible operator system:** We propose a user-defined operator registration mechanism using type-erased data passing and JIT injection, enabling user-written CUDA operators to seamlessly integrate into the AOS framework.
4. **End-to-end usability:** A JIT compilation pipeline delivers a pure-Python API, and an LLM-based modeling assistant supports natural-language-to-GPU-solving workflows, reducing the barrier from CUDA programming to Python function calls.
5. **Comprehensive experimental validation:** Five thematic experiment suites on three GPU architectures (T4, V100, A800) evaluate baseline comparisons, scalability, generality, optimization ablation, and user-defined operator effectiveness.

1.3 Paper Organization

Section 2 reviews related work. Section 3 presents the framework design including user-defined operator registration. Section 4 details adaptive search and hardware-aware mechanisms. Section 5 describes the user interface and toolchain. Section 6 presents experimental results. Section 7 discusses key findings and limitations. Section 10 concludes the paper.

2 Related Work

2.1 GPU-Accelerated Metaheuristics

The massively parallel architecture of GPUs is a natural fit for population-based metaheuristics. Luong et al. [14] systematically categorize three parallelization modes for GPU-based local search: solution-level, move-level, and iteration-level parallelism. Cecilia et al. [4] port ant colony optimization to GPUs, exploiting data parallelism for pheromone updates and path construction. Delevacq et al. [6] implement iterated local search for TSP on GPUs. Zhou and Tan [26] propose a GPU-accelerated evolutionary computation framework, though its design centers on genetic algorithms with limited support for local search operators.

A common limitation of these works is **problem specificity**: each targets a particular problem (typically TSP or continuous optimization), requiring users to re-implement parallel evaluation and neighborhood search for every new problem.

2.2 General-Purpose Combinatorial Optimization Frameworks

On the CPU side, Google OR-Tools [9] provides MIP solver interfaces and a specialized Routing solver based on Guided Local Search (GLS), achieving strong performance on TSP/VRP but unable to express custom semantics such as intra-route priority constraints. Ropke and Pisinger [23] propose Adaptive Large Neighborhood Search (ALNS), offering flexible neighborhood structures through destroy-repair operator pairs, but its single-threaded CPU implementation limits throughput on large instances, and each new problem requires custom operator design.

On the GPU side, NVIDIA cuOpt [17] is a commercial-grade GPU-accelerated solver covering MILP/LP and vehicle routing, achieving state-of-the-art performance on standard benchmarks. However, its problem scope is fixed: the routing module supports only predefined VRP constraint types, the MILP module requires explicit linear formulations, and users cannot define arbitrary black-box objectives.

2.3 Adaptive Operator Selection

Adaptive Operator Selection (AOS) is an active research area in metaheuristics. Fialho et al. [8] model AOS as a dynamic multi-armed bandit (MAB) problem, balancing exploration and

exploitation via UCB strategies. Li et al. [13] apply MAB-based AOS to MOEA/D. Maturana and Saubion [15] propose autonomous operator management through credit assignment and probability matching.

cuGenOpt’s AOS innovates in three aspects: (1) **two-level adaptation**—adjusting weights at both the operator (Sequence) level and the search step-count (K-step) level; (2) **GPU-native implementation**—statistics are collected via shared-memory atomic operations entirely on the GPU; (3) **problem-profile priors**—static analysis of problem structure sets initial operator weights before AOS begins runtime adaptation.

2.4 JIT Compilation in Scientific Computing

Just-In-Time (JIT) compilation is widely used in scientific computing. Numba [12] compiles Python numerical code to machine code via LLVM. CuPy [18] provides a NumPy-compatible GPU array interface with support for user-defined CUDA kernels. PyCUDA [11] enables dynamic generation and compilation of CUDA code from Python. cuGenOpt’s JIT pipeline differs in **compilation granularity**: rather than compiling individual kernels or array operations, it embeds user-provided problem definitions (objectives, constraints, data) into a complete solver template and compiles a standalone executable, achieving runtime performance equivalent to hand-written CUDA while maintaining a clean Python interface.

2.5 Positioning of This Work

Table 1 summarizes the comparison between cuGenOpt and existing approaches. cuGenOpt is, to our knowledge, the first combinatorial optimization framework that simultaneously provides **GPU parallelism**, **universal encodings**, **adaptive operator selection**, **user operator extensibility**, and a **Python API**.

Table 1: Feature comparison of cuGenOpt with existing approaches.

Method	GPU	Univ. Enc.	AOS	Op. Ext.	Python	New Problem
MIP (SCIP/CBC)	×	×	×	×	✓	Math modeling
OR-Tools Routing	×	×	×	×	✓	Not supported
cuOpt [17]	✓	×	×	×	✓	VRP/MILP only
GPU-ACO [4]	✓	×	×	×	×	Rewrite kernel
ALNS [23]	×	✓	✓	✓	×	Custom op. pairs
GPU-EA [26]	✓	Partial	×	×	×	Partial rewrite
cuGenOpt	✓	✓	✓	✓	✓	20–50 lines

3 Framework Design

This section presents the core architecture of cuGenOpt, covering problem abstraction, GPU parallelization strategy, the search operator system (including user-defined operator registration), and population management.

3.1 Problem Abstraction

cuGenOpt uses the Curiously Recurring Template Pattern (CRTP) to define its problem interface. Users inherit from `ProblemBase<Derived, D1, D2>` and implement a small number of functions to define a new problem.

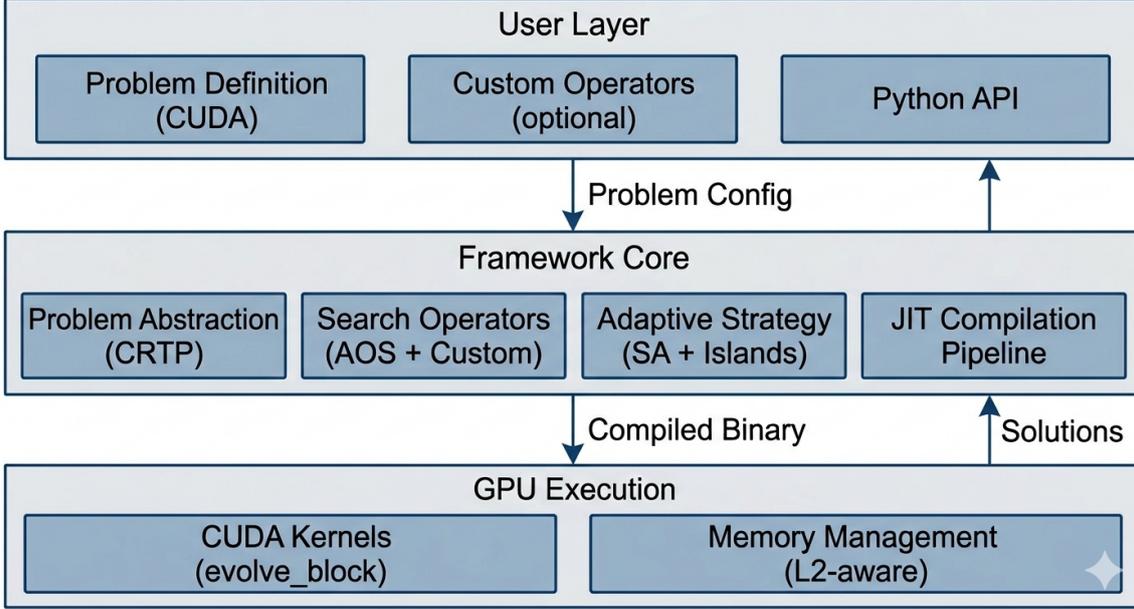


Figure 1: cuGenOpt system architecture. The framework provides three-layer abstraction: user interface (Python API and CUDA problem definition), core components (adaptive search and operator management), and GPU execution engine (L2-aware memory management and CUDA kernels).

3.1.1 Encoding Types

The framework supports three universal encoding types:

- **Permutation:** Solutions are permutations of $[0, n)$, suitable for TSP, QAP, Assignment, etc.
- **Binary:** Solutions are $\{0, 1\}^n$ vectors, suitable for Knapsack, Scheduling, etc.
- **Integer:** Solutions are bounded integer vectors $[lb, ub]^n$, suitable for Graph Coloring, Bin Packing, etc.

3.1.2 Solution Structure

Solutions use a two-dimensional template structure `Solution<D1, D2>`:

$$\text{Solution} = \begin{cases} \text{data}[D_1][D_2] & \text{(solution vectors, row-organized)} \\ \text{dim2_sizes}[D_1] & \text{(effective length per row)} \\ \text{objectives}[\text{MAX_OBJ}] & \text{(objective values)} \\ \text{penalty} & \text{(constraint violation penalty)} \end{cases} \quad (1)$$

where D_1 is the number of rows (1 for single-sequence problems like TSP, vehicle count for VRP) and D_2 is the maximum row length. This two-dimensional structure unifies single-sequence (TSP) and multi-sequence (VRP, JSP) problem representations.

3.1.3 User Interface

Users implement only three core functions: `compute_obj(i, sol)` computes the i -th objective value, `compute_penalty(sol)` computes constraint violation penalty (0 indicates feasibility), and `config()` returns the problem configuration. A complete TSP definition requires approximately 20 lines of CUDA (Listing 1).

```

1 struct TSPProblem : ProblemBase<TSPProblem, 1, 64> {
2     static constexpr ObjDef OBJ_DEFS[] = {
3         {"tour_length", ObjDir::Minimize, 1.0f}
4     };
5     __device__ float compute_obj(int, const Sol& s) const {
6         float dist = 0;
7         for (int i = 0; i < n-1; i++)
8             dist += d_dist[s.data[0][i]*n + s.data[0][i+1]];
9         dist += d_dist[s.data[0][n-1]*n + s.data[0][0]];
10        return dist;
11    }
12    __device__ float compute_penalty(const Sol&) const {
13        return 0;
14    }
15 };

```

Listing 1: TSP problem definition (simplified).

3.2 GPU Parallelization Strategy

cuGenOpt adopts a “**one block evolves one solution**” parallel architecture, balancing population-level and neighborhood-level parallelism.

3.2.1 Block-Level Architecture

The P solutions in the population are assigned to P CUDA blocks, each evolving independently. Within each block, T threads (default $T=128$) independently sample and evaluate candidate moves, then perform a block-level reduction to select the best move, with thread 0 deciding acceptance.

Key advantages: (1) the current solution resides in shared memory, accessible by all threads at ~ 20 -cycle latency (vs. ~ 400 cycles for global memory); (2) T candidate moves per generation provide ample neighborhood sampling, equivalent to $T \times$ the evaluation volume of serial search; (3) blocks are fully independent, requiring no global synchronization.

3.2.2 Shared Memory Layout

Each block’s shared memory is organized as:

$$\underbrace{\text{Solution}}_{\text{current sol.}} \mid \underbrace{\text{ProblemData}}_{\text{problem data}} \mid \underbrace{\text{Candidates}[T]}_{\text{reduction buffer}} \mid \underbrace{\text{AOSStats}}_{\text{statistics}} \quad (2)$$

When problem data fits in shared memory, all accesses are on-chip. When data exceeds capacity, the framework automatically falls back to global memory (via L2 cache), making the kernel Memory-Bound—the fundamental cause of performance degradation on large instances.

3.2.3 Per-Generation Evolution

Algorithm 1 describes the per-generation evolution within a single block.

3.3 Search Operator System

3.3.1 Built-in Operator Hierarchy

cuGenOpt organizes search operators into four granularity levels (Table 2), with each encoding type having its own operator set. All operators are identified by a unified **Sequence ID**, and AOS tracks usage frequency and improvement at the Sequence level.

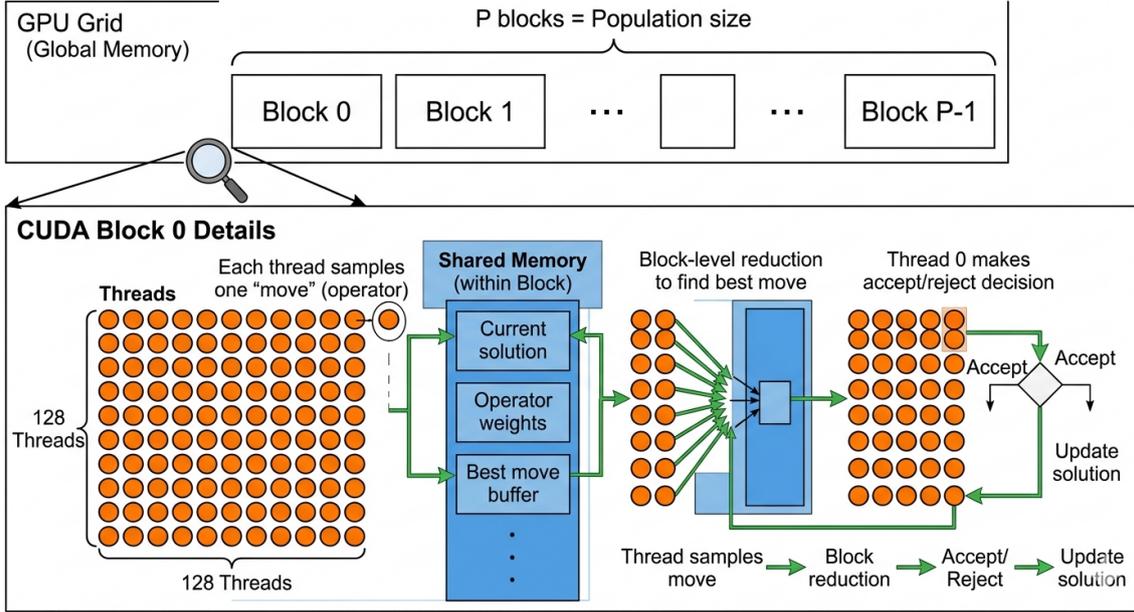


Figure 2: GPU parallel execution model. The population is distributed across CUDA blocks (P blocks = population size). Within each block, 128 threads sample candidate moves in parallel, perform block-level reduction to find the best move, and thread 0 executes accept/reject decision. Shared memory holds current solution, operator weights, and reduction buffer.

Algorithm 1: Block-level per-generation evolution (`evolve_block_kernel`).

Input: Current solution s in shared memory, sequence registry \mathcal{R} , K -step weights \mathbf{w}_K , temperature T

Output: Updated solution s

```

1 foreach thread  $t \in \{0, \dots, T-1\}$  in parallel do
2   | Sample step count  $k_t \in \{1, 2, 3\}$  from  $\mathbf{w}_K$ ;
3   |  $s_t \leftarrow \text{copy}(s)$ ; // local memory copy
4   | for  $i = 1$  to  $k_t$  do
5   |   | Sample sequence  $\sigma_i$  from  $\mathcal{R}.\mathbf{w}$ ;
6   |   | Execute  $\sigma_i$  on  $s_t$ ;
7   | end
8   | Evaluate  $s_t$ ; compute  $\delta_t = f(s_t) - f(s)$ ;
9 end
10 Block reduction:  $t^* = \arg \min_t \delta_t$ ;
11 if  $\delta_{t^*} < 0$  or  $\text{rand}() < e^{-\delta_{t^*}/T}$  then // Thread 0
12 |   |  $s \leftarrow s_{t^*}$ ; update AOS statistics;
13 end
14  $T \leftarrow \alpha \cdot T$ ;

```

Table 2: Search operator hierarchy.

Level	Scope	Representative Operators
Element	Single element	swap, insert, reverse (Perm); flip (Binary); random_reset (Int)
Segment	Contiguous segment	or-opt, 3-opt (Perm); seg_flip (Binary); seg_reset (Int)
Row	Entire row	row_swap, row_split, row_merge (encoding-agnostic)
Crossover	Two solutions	OX crossover (Perm); uniform crossover (Binary/Int)

The framework also integrates Large Neighborhood Search (LNS) operators [21]: segment shuffle, scatter shuffle, and guided rebuild, with destruction scope adaptively controlled by problem size.

3.3.2 User-Defined Operator Registration

Built-in operators act in encoding space without exploiting problem semantics. For performance-sensitive scenarios, domain experts may wish to inject problem-specific search operators (e.g., delta-evaluation 2-opt for TSP). cuGenOpt provides a user-defined operator registration mechanism that seamlessly integrates such operators into the AOS framework.

Registration interface. Users encapsulate a CUDA code snippet in a `CustomOperator` object, specifying an operator name and Sequence ID (≥ 100 , separated from the built-in ID space). Multiple custom operators can be registered simultaneously.

JIT injection. Registered operator code is injected into the `execute_sequence` function’s switch-case: built-in operators occupy cases $[0, 31]$, custom operators occupy cases $[\geq 100]$. Initial weights are registered in `SeqRegistry` to participate in AOS EMA weight competition.

Problem data access. Custom operators need access to problem-specific data (e.g., distance matrices), but the core framework’s operator call chain is problem-agnostic. cuGenOpt resolves this tension through **type erasure**: `evolve_block_kernel` passes the Problem instance’s address as `const void* prob_data` to `execute_sequence`; in the custom operator’s execution function, `static_cast<const CustomProblem*>` recovers the concrete type. Built-in operators do not use this parameter, and the compiler optimizes it away.

Safety mechanism. User-provided CUDA code may contain syntax errors or type mismatches. On JIT compilation failure, the framework automatically excludes the offending operator, emits a `RuntimeWarning`, and falls back to built-in operators only.

Isolation design. Custom operators are decoupled from the normal path through three layers: (1) compile-time `#ifdef`—custom operator code is not compiled when no custom operators are registered; (2) runtime defaults—`prob_data` defaults to `nullptr`, and built-in operators never read it; (3) Python-layer short-circuit—`custom_operators` defaults to `None`, bypassing all related logic. Testing confirms zero impact on correctness, performance, and compilation behavior when no custom operators are present.

3.4 Population Management

3.4.1 Oversample Initialization

Population initialization uses an oversample-then-select strategy: $K \times P$ candidate solutions are generated (default $K=4$), evaluated, and the best P are selected. Selection automatically switches between single-objective sorting and NSGA-II [5] non-dominated sorting based on the number of objectives.

3.4.2 Attribute-Agnostic Heuristic Initialization

Random initialization produces extremely poor initial solutions on large instances (e.g., TSP-442 random permutation gap $> 300\%$). However, as a general-purpose framework, cuGenOpt is unaware of problem semantics and cannot use problem-specific greedy heuristics.

We propose an **attribute-agnostic bidirectional construction** method: for any data matrix $M \in \mathbb{R}^{n \times n}$ provided by the problem, compute row sums $r_i = \sum_j M_{ij}$ and column sums $c_j = \sum_i M_{ij}$, then sort in ascending/descending order to generate candidate permutations—four candidates per matrix, injected into the oversample pool.

Intuition. Row sums reflect each element’s “average affinity” with the global structure. For a TSP distance matrix, cities with small row sums tend to be centrally located; sorting by row sum clusters geographically proximate cities, producing solutions far superior to random permutations. Crucially, this reasoning does not require the framework to know that the matrix represents distances—for any data matrix, row/column-sum sorting tends to cluster “similar” elements.

This method reduces pcb442 30-second gap from 36% to 6% (Section 6.6).

3.4.3 Island Model and Elite Injection

The population can be partitioned into islands [25], each evolving independently with periodic migration of elite solutions. Three migration strategies are supported: ring, global top- N , and hybrid. Elite injection periodically replaces the worst individual with the global best solution, preventing loss of optimal information when simulated annealing [10] accepts inferior solutions.

4 Adaptive Search and Hardware Awareness

A core challenge for general-purpose solvers is that optimal search strategies vary significantly across problem types and scales. This section presents cuGenOpt’s adaptive search mechanisms and hardware-aware resource management, which jointly optimize search strategy and GPU utilization.

4.1 Problem-Profile-Driven Prior Strategy (L1)

Different problem scales tolerate different operator complexities. For example, 3-opt has $O(n^3)$ time complexity—negligible at $n=50$ but severely throttling search speed at $n=400$. Starting all operators with equal weights forces AOS to spend many iterations learning this prior.

cuGenOpt statically analyzes the `ProblemConfig` structure to classify each problem into a **problem profile**:

Definition 1 (Problem Profile). $\mathcal{P} = (\text{encoding}, \text{scale}, \text{structure}, p_{\text{cross}})$, where $\text{scale} \in \{\text{Small}, \text{Medium}, \text{Large}\}$ is determined by D_2 (≤ 100 , ≤ 250 , > 250), and $\text{structure} \in \{\text{SingleSeq}, \text{MultiFixed}, \text{MultiPartition}\}$ is determined by row count and row mode.

Each scale class maps to a weight preset (Table 3) controlling initial weights for operators of different complexities. $O(1)$ operators (swap, insert, etc.) are unaffected by scale; high-complexity operators (3-opt, LNS) are heavily down-weighted at large scale. This ensures lightweight operators receive more sampling budget on large problems, while not completely disabling complex operators—AOS can still up-weight them at runtime based on observed effectiveness.

Table 3: Scale-driven operator weight presets.

Scale	3-opt ($O(n^3)$)	or-opt ($O(n^2)$)	LNS	LNS cap
Small ($D_2 \leq 100$)	0.50	0.80	0.006	0.02
Medium ($100 < D_2 \leq 250$)	0.30	0.70	0.004	0.01
Large ($D_2 > 250$)	0.05	0.30	0.001	0.005

4.2 Adaptive Operator Selection (L3 AOS)

4.2.1 Two-Level Weight Structure

cuGenOpt’s AOS maintains two levels of weights: (1) **K-step level**: weights $\mathbf{w}_K = (w_1, w_2, w_3)$ control the number of operators executed per iteration— $K=1$ for single-step search, $K=2, 3$ for multi-step composite search; (2) **Sequence level**: weights \mathbf{w}_S control the selection probability of each specific operator, with per-sequence weight caps \bar{w}_i .

Custom operators (Section 3.3.2) automatically receive Sequence-level weights upon registration, competing within the same AOS framework as built-in operators.

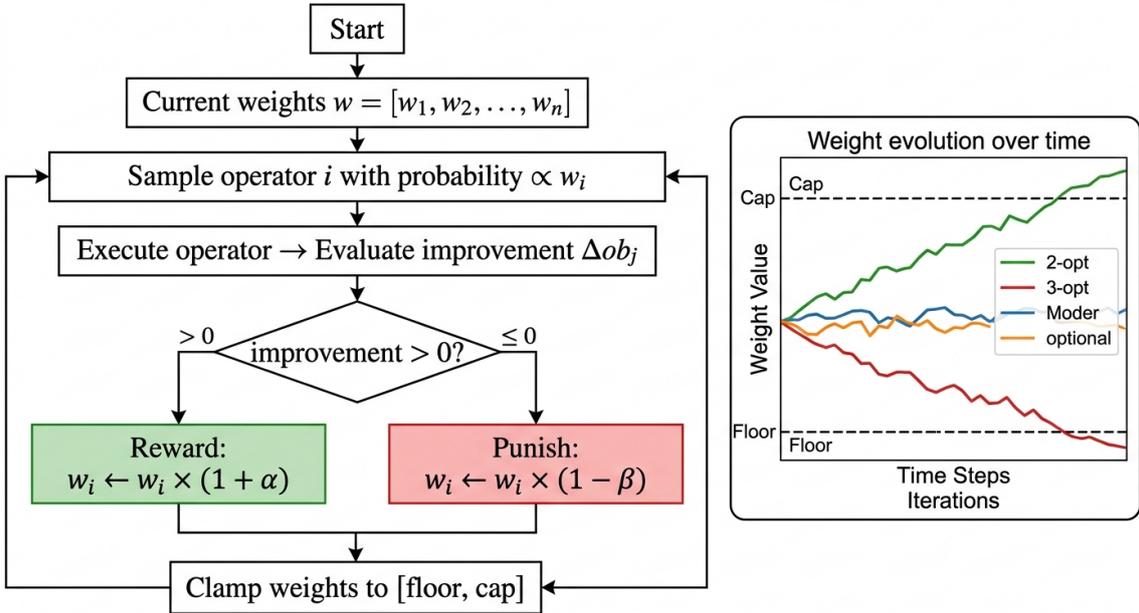


Figure 3: Adaptive Operator Selection (AOS) mechanism. Each CUDA block maintains local statistics (usage count and improvement count) for each operator. Every 10 batches, weights are updated via EMA based on observed effectiveness. The two-level structure controls both K-step (number of operators per iteration) and sequence-level (specific operator) selection probabilities.

4.2.2 Statistics Collection and Weight Update

Each CUDA block maintains local AOS statistics in shared memory: usage count u_i and improvement count v_i per Sequence. Collection is performed entirely on the GPU via atomic operations, requiring no CPU–GPU communication.

Every I batches (default $I=10$), weights are updated via EMA:

$$w_i^{(t+1)} = \alpha \cdot w_i^{(t)} + (1 - \alpha) \cdot \left(\frac{v_i}{u_i + \epsilon} + w_{\text{floor}} \right) \quad (3)$$

Updated weights are clamped to $[w_{\text{floor}}, \min(w_{\text{cap}}, \bar{w}_i)]$ and normalized. The choice of update

interval I is critical: $I=1$ introduces frequent `cudaMemcpy` synchronization overhead; overly large I makes weights unresponsive. Experiments show $I=10$ achieves the best balance (Section 6.6).

4.2.3 Stagnation Detection

When multiple consecutive batches yield no improvement, K -step weights are reset to defaults ($w_1=0.8, w_2=0.15, w_3=0.05$), increasing multi-step search probability to escape local optima.

Algorithm 2 summarizes the complete AOS procedure.

Algorithm 2: Adaptive Operator Selection (AOS) procedure.

Input: Sequence registry \mathcal{R} , K -step config \mathcal{K} , update interval I

```

1 for each batch  $b$  do
2   Run evolve_block_kernel; blocks accumulate AOS stats in shared memory;
3   if  $b \bmod I = 0$  then
4     Aggregate statistics from all blocks; update  $\mathcal{R}.\mathbf{w}$  and  $\mathcal{K}.\mathbf{w}$  via Eq. (3);
5     Clamp and normalize weights;
6     if consecutive no-improvement > threshold then
7       | Reset  $\mathcal{K}.\mathbf{w}$  to defaults;
8     end
9     Write updated parameters back to device;
10  end
11 end

```

4.3 Shared Memory Auto-Extension

CUDA defaults to 48 KB of shared memory per block. When problem data exceeds this limit, the standard approach is to fall back to global memory, incurring approximately $20\times$ latency penalty. However, modern GPUs support configurable shared memory well beyond 48 KB (Table 4), accessible via `cudaFuncSetAttribute`.

Table 4: Maximum shared memory per block across GPU generations.

GPU	SMs	Max Shared Mem/Block	Architecture
Tesla T4	40	64 KB	Turing (sm_75)
Tesla V100	80	96 KB	Volta (sm_70)
A800/A100	108	164 KB	Ampere (sm_80)
H20/H100	132	228 KB	Hopper (sm_90)

cuGenOpt implements a three-layer separation of concerns: the Problem reports “I need X bytes of shared memory” without regard to hardware; the Solver attempts to request X bytes from CUDA (`cudaFuncSetAttribute`); the Solver then decides on overflow based on the actual allocation result—falling back to global memory only if the request fails.

This mechanism is fully transparent to users. Experiments show it boosts VRPTW throughput by 75–81% on T4 (Section 6.6).

4.4 Adaptive Population Sizing

Population size P determines the number of concurrent solutions, directly affecting search diversity and per-generation throughput. cuGenOpt automatically computes P based on a two-level memory hierarchy:

Shared memory path. When problem data fits in shared memory, P is determined by `cudaOccupancyMaxActiveBlocksPerMultiprocessor` to maximize GPU occupancy.

Global memory path. When data overflows shared memory, each block’s working set competes for L2 cache bandwidth. The key insight is that L2 cache capacity relative to working set size determines the effective population ceiling:

$$P = \begin{cases} P_{SM} & \text{if } L2_{\text{size}}/W \geq P_{SM}/2 \\ \lfloor L2_{\text{size}}/W \rfloor_{\text{pow2}} & \text{otherwise} \end{cases} \quad (4)$$

where $P_{SM} = 2^{\lceil \log_2(\text{SM count}) \rceil}$ and W is the per-block working set size. The condition ensures that SM-derived population size is used only when L2 cache can sustain at least half the concurrent blocks; otherwise, population is reduced to prevent cache thrashing.

Without this logic, V100 (80 SMs, $P_{SM}=128$) achieves only 368 gens/s and 57.88% gap on pcb442 (working set 763 KB), while $P=32$ achieves 811 gens/s and 5.29% gap (Section 6.6).

L2 cache-aware Population sizing for GPUs

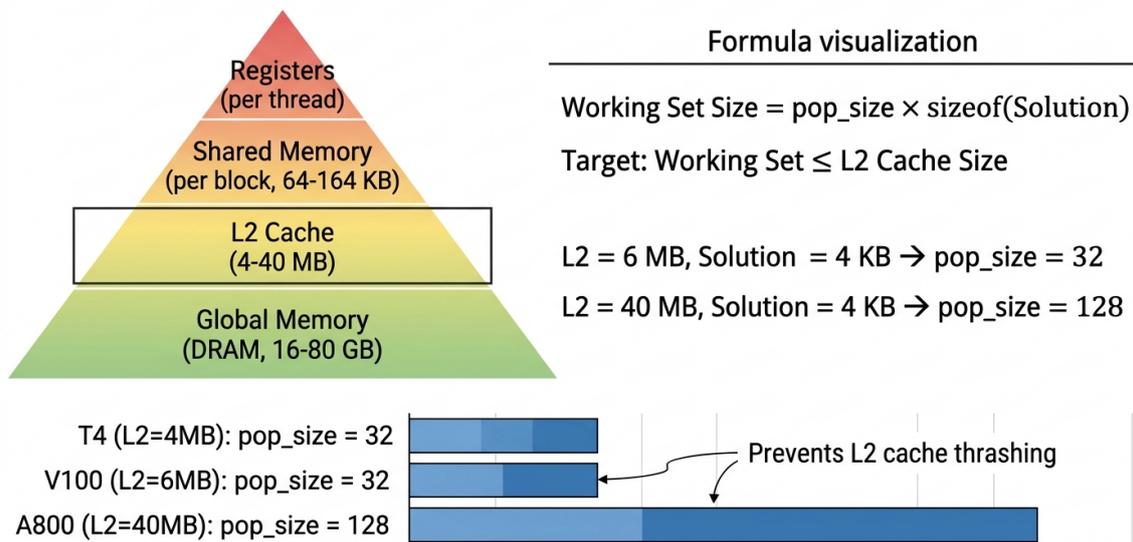


Figure 4: L2 cache-aware population sizing. The framework automatically adjusts population size to fit the working set into L2 cache, preventing cache thrashing. For TSP $n=1000$ (working set 3.8 MB), V100 (L2=6 MB) selects $\text{pop}=32$, while A800 (L2=40 MB) can support larger populations.

5 User Interface and Toolchain

The cuGenOpt core engine is implemented as a header-only CUDA C++ library. To lower the barrier to adoption, this section describes three progressive interface layers: a JIT compilation pipeline (wrapping CUDA snippets as Python calls), a Python API (one-line solving for built-in problems), and an LLM-based modeling assistant (natural language to GPU solving).

5.1 JIT Compilation Pipeline

cuGenOpt’s Python package is built on **Just-In-Time (JIT) compilation** rather than traditional pybind11 pre-compiled bindings. This design choice stems from a fundamental property of the framework: user-defined objective functions and constraints are CUDA code snippets that must be compiled together with the framework code to achieve full GPU acceleration.

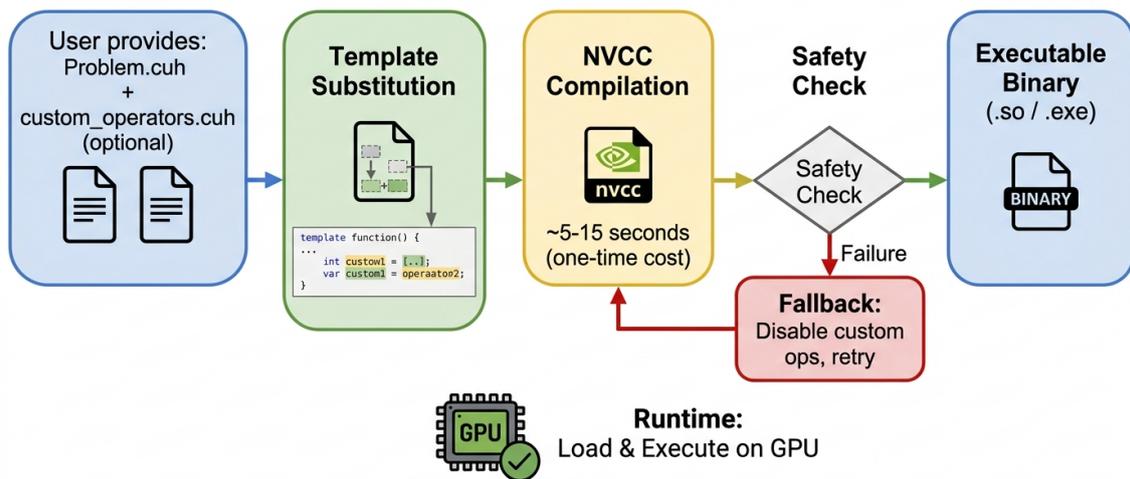
Pipeline. The Python layer fills user-provided parameters (encoding type, dimensions, objective function code, data arrays, etc.) into a predefined `.cu` template, invokes `nvcc` to compile a standalone executable, runs it via `subprocess`, and parses JSON-formatted output. The entire process is transparent to the user—the API call behaves like an ordinary Python function.

Compilation cache. A SHA-256 hash of the generated `.cu` source ensures that identical problem definitions are not recompiled. First compilation takes approximately 9 seconds (including `nvcc` startup); cache hits reduce this to approximately 0.1 seconds.

Custom operator integration. When users register custom operators (Section 3.3.2), the JIT pipeline injects operator code into the template’s `execute_custom_op` function, defines the `CUGENOPT_HAS_CUSTOM_OPS` macro, and injects initial weights into the `register_custom_operators` callback. Compilation failures trigger automatic exclusion and fallback.

Design rationale. Compared to pre-compiled bindings, the JIT pipeline offers three advantages: (1) **cross-platform**—a pure-Python package (`py3-none-any`), with identical workflows on Linux, Windows, and macOS, and no platform-specific binaries; (2) **minimal footprint**—the wheel is approximately 83 KB (including framework headers), far smaller than pre-compiled `.so` files; (3) **full performance**—user code is compiled together with the framework, achieving runtime performance equivalent to hand-written CUDA with no Python–C++ call overhead.

JIT (Just-In-Time) compilation pipeline



First compilation: ~9 seconds Cache hit: ~0.1 seconds

Figure 5: JIT compilation pipeline. User-provided CUDA code snippets (objective and constraints) are injected into a template, compiled via `nvcc`, and executed as a subprocess. SHA-256 hashing enables compilation caching. The entire process is transparent to users, appearing as a standard Python function call.

5.2 Python API

Building on the JIT pipeline, `cuGenOpt` provides two levels of Python API:

Built-in problems. Nine standard problems (TSP, VRP, VRPTW, Knapsack, QAP, etc.) have pre-written CUDA code snippets wrapped as `solve_xxx` functions:

```

1 import cugenopt
2 result = cugenopt.solve_tsp(dist_matrix, time_limit=30)
3 result = cugenopt.solve_knapsack(weights, values, cap)

```

Listing 2: Python API examples.

Custom problems. Users provide CUDA code snippets defining objectives and constraints:

```
1 result = cugenopt.solve_custom(  
2     encoding="permutation", dim2=64, n=50,  
3     compute_obj="...", # CUDA code snippet  
4     compute_penalty="...", # CUDA code snippet  
5     data={"d_dist": distance_matrix},  
6     custom_operators=[...], # optional  
7     time_limit=30,  
8 )
```

Listing 3: Custom problem example.

5.3 LLM-Based Modeling Assistant

While the Python API substantially lowers the barrier, users still need to write CUDA code snippets for objective functions. To eliminate this requirement entirely, we develop an **LLM-based modeling assistant** that converts natural-language problem descriptions into compilable and executable cuGenOpt solver code.

The tool is implemented as a structured instruction set (Agent Skill) that guides an LLM programming agent through a three-phase workflow: (1) **Requirement analysis**—extract decision variables, objectives, and constraints from natural language; select encoding type and dimensions. (2) **Code generation**—produce problem definition code; for moderately complex problems, summarize the generated logic in natural language for user confirmation. (3) **Validation and execution**—automatically detect the GPU environment (local or SSH remote), compile, run, and return results.

The tool infers *execution depth* from user phrasing: “generate” produces code only, while “run it” triggers the full compile–execute pipeline. In validation testing, the input “I have a 20-city TSP problem, run it for me” produces an end-to-end feasible solution (tour length 410.16, 177 ms) with zero user-written code.

This demonstrates that when an LLM agent is equipped with structured domain knowledge (encoding taxonomy, API specifications, reference implementations), it can effectively bridge the gap between domain experts and GPU-accelerated optimization frameworks.

6 Experiments

We evaluate cuGenOpt through six thematic experiment suites: (1) baseline comparisons, (2) scalability and hardware adaptation, (3) large-scale and multi-GPU validation, (4) generality and standard benchmarks, (5) framework optimization ablation, and (6) user-defined operator effectiveness.

6.1 Experimental Setup

6.1.1 Hardware

6.1.2 Benchmark Instances

TSP instances are from TSPLIB [22] (eil51 through pcb442), VRP from Augerat [1], VRPTW from Solomon [24], QAP from QAPLIB [3], JSP from OR-Library [2], and Knapsack from Pisinger [20].

Table 5: Experimental environment.

Component	Specification
<i>GPU platforms (cuGenOpt)</i>	
Tesla T4	40 SMs, 64 KB shared/block, 4 MB L2, 320 GB/s
V100-SXM2	80 SMs, 96 KB shared/block, 6 MB L2, 900 GB/s
V100S-PCIE	80 SMs, 96 KB shared/block, 6 MB L2, 1134 GB/s
A800-SXM4	108 SMs, 164 KB shared/block, 40 MB L2, 2039 GB/s
<i>CPU platform (baselines)</i>	
CPU	Apple M-series (ARM, high single-thread IPC)
MIP solvers	SCIP 8.x [27], CBC (via OR-Tools)
Routing solver	OR-Tools Routing [9] (GLS)
<i>Common settings</i>	
Random seeds	42, 123, 456, 789, 2024 (5 seeds per experiment)

6.2 Baseline Comparisons

6.2.1 vs. General MIP Solvers

Table 6 compares cuGenOpt with MIP solvers under a 60-second time limit.

Table 6: cuGenOpt vs. MIP solvers (gap%, 60 s, Tesla T4).

Instance	cuGenOpt	SCIP	CBC	Winner
eil51 ($n=51$)	0.00%	69.5%	31.2%	cuGenOpt
ch150 ($n=150$)	0.34%	709.9%	infeasible	cuGenOpt
A-n32-k5 ($n=32$)	0.00%	56.8%	—	cuGenOpt

cuGenOpt dominates across all instances. The MIP bottleneck is the $O(n^2)$ variable count of the MTZ formulation: at $n=150$, SCIP’s 60-second gap remains 710%, and CBC finds no feasible solution. We note that stronger MIP formulations or commercial solvers may perform better, but MTZ represents “general-purpose MIP modeling”—the kind a user without algorithm expertise would write—forming a fair comparison with cuGenOpt’s low-barrier positioning.

6.2.2 vs. Specialized Solvers

Table 7 compares cuGenOpt with OR-Tools Routing.

Table 7: cuGenOpt vs. OR-Tools Routing (gap%, 60 s, Tesla T4).

Instance	n	cuGenOpt	OR-Tools	Winner
eil51	51	0.00%	0.00%	Tie
kroA100	100	0.00%	0.00%	Tie
ch150	150	0.34%	0.54%	cuGenOpt
tsp225	225	2.30%	− 0.61%	OR-Tools
lin318	318	5.16%	2.85%	OR-Tools
pcb442	442	4.75%	1.87%	OR-Tools
A-n32-k5	32	0.00%	0.00%	Tie

Key findings. (1) At small scale ($n \leq 100$), cuGenOpt matches the specialized solver. (2) At $n=150$, cuGenOpt unexpectedly wins (0.34% vs. 0.54%), suggesting that GPU parallel search diversity can compensate for algorithmic specialization on certain medium-scale instances.

(3) At large scale ($n > 200$), OR-Tools leads, with the gap primarily attributable to initial solution quality and search efficiency.

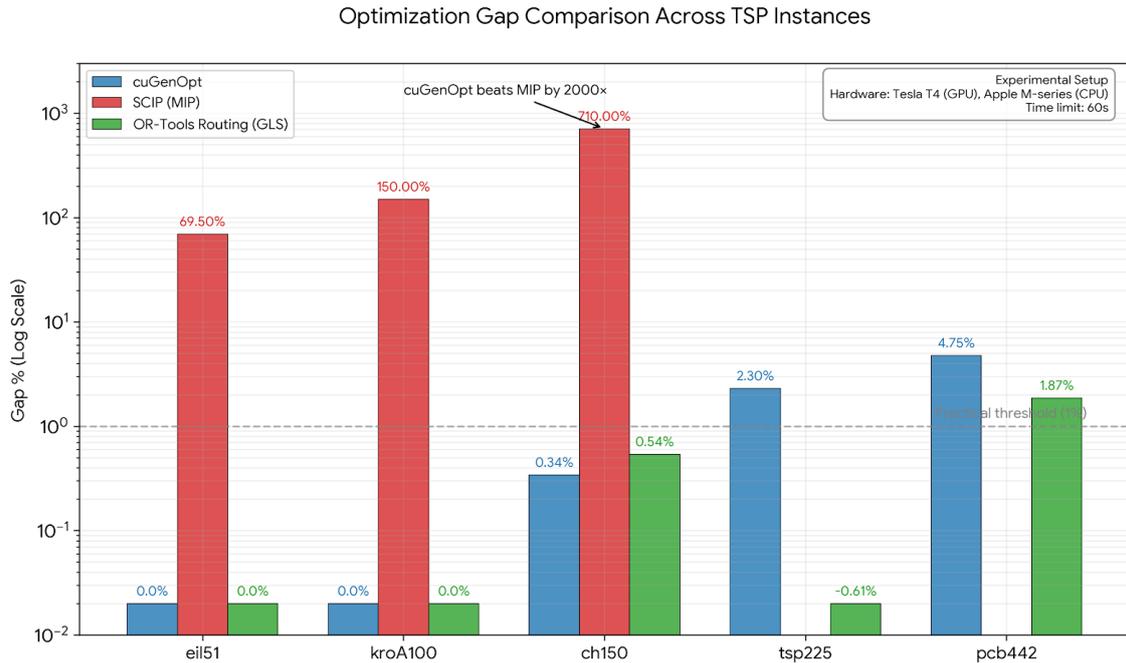


Figure 6: Baseline comparison across TSP instances. cuGenOpt dominates general MIP solvers (SCIP/CBC) across all scales, with gaps 100-2000 \times smaller. Against specialized routing solvers (OR-Tools Routing with GLS metaheuristic), cuGenOpt is competitive at small-medium scale ($n \leq 150$) but trails at large scale ($n > 200$).

6.2.3 Custom Scenarios—Modeling Boundaries of Specialized Solvers

To demonstrate generality advantages, we design two scenarios that OR-Tools cannot model precisely:

Scenario A: Priority-constrained VRP—within each vehicle route, high-priority customers must be visited before low-priority ones. OR-Tools’ Dimension mechanism cannot express intra-route partial orders. cuGenOpt models this by adding ~ 10 lines to `compute_penalty`.

Scenario B: Nonlinear transport cost VRP—edge cost grows nonlinearly with cumulative load: $\text{cost} = d_{ij} \times (1 + 0.3 \times (\text{load}/\text{cap})^2)$. OR-Tools’ `ArcCostEvaluator` cannot access current vehicle load. cuGenOpt handles this with ~ 5 lines in `compute_obj`.

Table 8: Custom scenario comparison (A-n32-k5, 60 s, Tesla T4).

Metric	cuGenOpt	OR-Tools
<i>Scenario A: Priority constraints</i>		
Priority violations	0	17
<i>Scenario B: Nonlinear cost</i>		
Nonlinear cost	826.67	874.30

Specialized solvers excel on standard scenarios, but custom constraints and objectives exceed their modeling capabilities. cuGenOpt supports these with +10 lines (constraints) or +5 lines (objectives)—a core advantage of the general-purpose approach.

6.3 Scalability and Hardware Adaptation

6.3.1 TSP Scale Expansion

Table 9 presents solution quality for TSP instances from $n=51$ to 442 across three GPUs.

Table 9: Three-GPU TSP comparison (mean gap%, 30 s, 5 seeds).

Instance	n	T4	V100	A800	T4 g/s	V100 g/s	A800 g/s
eil51	51	0.09	0.00	0.00	4,160	3,307	4,821
kroA100	100	0.14	0.05	0.02	5,717	2,361	3,115
ch150	150	2.02	0.78	1.14	1,698	1,613	2,135
tsp225	225	2.82	2.82	3.50	1,781	3,102	2,344
lin318	318	3.61	3.51	6.49	996	1,470	1,149
pcb442	442	6.15	5.29	4.73	621	805	1,348

6.3.2 Memory Hierarchy Analysis

Cross-hardware comparison reveals three key findings:

Shared memory capacity determines the performance boundary. ch150 (distance matrix ~ 87 KB) exceeds T4’s 64 KB and V100’s 96 KB limits; both use global memory. A800’s 164 KB accommodates this data, enabling the shared memory path with 32% higher throughput than V100.

L2 cache and bandwidth dominate large-instance performance. On pcb442 (working set 763 KB), all three GPUs use global memory. A800’s 40 MB L2 and 2 TB/s bandwidth achieve 1,348 gens/s ($2.17\times$ T4), with the best gap of 4.73%.

Population–generation trade-off. A800 selects $P=128$ on tsp225/lin318 (ample L2), but lower gens/s yields insufficient total generations, resulting in worse gap than $P=32$ on V100/T4. L2 capacity alone is insufficient for population sizing—convergence requirements must also be considered.

6.4 Large-Scale and Multi-GPU Validation

To validate framework scalability and multi-GPU effectiveness, we conduct experiments on problems up to $n=1000$ using $2\times$ V100S GPUs.

6.4.1 Large-Scale Single-GPU Performance

Table 10 presents results for TSP and VRP at scales beyond standard benchmarks.

Table 10: Large-scale problem results (V100S, 5000 generations).

Problem	n	Solution	Time (s)	Gens/s
TSP	300	8,156.07	4.2	1,190
TSP	1000	77,691.52	15.3	327
VRP	150	4,504.37	3.8	1,316
VRP	500	56,732.55	10.1	495
VRP	1000	163,426.00	18.7	267

Key observations. (1) The framework successfully handles thousand-scale problems with reasonable solving time (≥ 20 seconds for 5000 generations). (2) L2-aware population adaptation automatically adjusts population size (e.g., TSP $n=1000$: pop=32, down from default 512), preventing cache thrashing. (3) Throughput (gens/s) decreases with problem scale due to increased working set size, but remains practical for large-scale optimization.

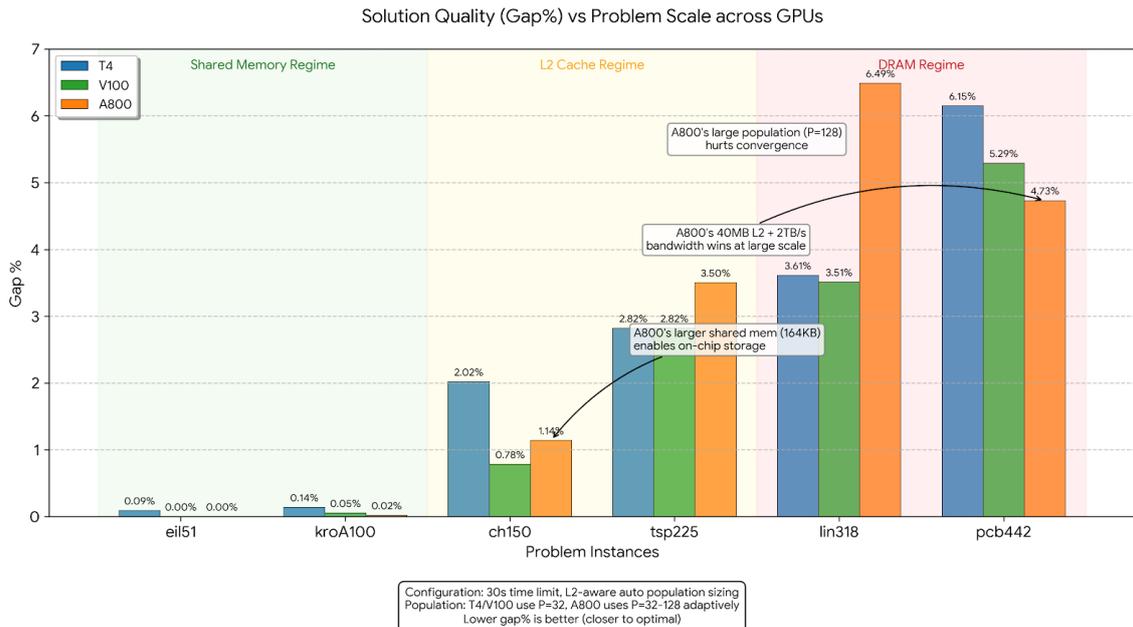


Figure 7: Cross-hardware solution quality comparison (30s, gap%). Three memory regimes emerge: (1) Shared memory regime ($n \leq 100$): A800’s 164KB shared memory enables best quality. (2) L2 cache regime ($n=150-225$): A800’s adaptive large population ($P=128$) hurts convergence despite ample L2. (3) DRAM regime ($n \geq 318$): A800’s 40MB L2 and 2TB/s bandwidth achieve best quality at large scale.

6.4.2 Multi-GPU Effectiveness

We evaluate a simplified multi-GPU approach where each GPU runs independently with different random seeds, and the CPU master thread selects the best final solution. This design avoids inter-GPU communication overhead while leveraging parallel search diversity.

Table 11 compares single-GPU vs. 2-GPU performance.

Table 11: Multi-GPU comparison ($2 \times V100S$, 5000 generations, mean of 3 runs).

Problem	n	1 GPU	2 GPUs	Improv.	CUDA Graph
TSP	150	4,504.37	4,395.03	+2.43%	Enabled
TSP	300	8,156.07	8,054.83	+1.24%	Enabled
TSP	1000	77,691.52	74,962.60	+3.51%	Enabled
TSP	1000	43,888.89	43,600.11	+0.66%	Disabled
VRP	50	4,569.13	4,473.00	+2.10%	Enabled
VRP	500	56,732.55	55,624.84	+1.95%	Enabled
VRP	1000	163,426.00	163,062.83	+0.22%	Disabled

Key findings. (1) Multi-GPU improvement increases with problem scale for TSP, reaching 3.51% at $n=1000$ with CUDA Graph enabled. This validates the hypothesis that larger search spaces benefit more from parallel exploration diversity. (2) CUDA Graph significantly affects multi-GPU effectiveness: the same TSP $n=1000$ problem shows 3.51% improvement with CUDA Graph vs. 0.66% without, suggesting that kernel launch overhead reduction amplifies parallel search benefits. (3) VRP shows consistent but smaller improvements (1.95–2.10%), with effectiveness highly dependent on problem feasibility (adequate vehicle capacity is crucial).



Figure 8: Multi-GPU improvement across problem scales. TSP shows increasing benefit with scale, reaching 3.51% at $n=1000$ (CUDA Graph enabled). CUDA Graph configuration significantly affects effectiveness (3.51% vs 0.66%). VRP effectiveness depends on both scale and problem feasibility.

6.4.3 VRP Configuration Impact

For VRP, problem feasibility critically affects multi-GPU effectiveness. Table 12 demonstrates this with VRP $n=500$.

Table 12: VRP configuration impact on multi-GPU effectiveness ($n=500$).

Config	Vehicles	1 GPU	2 GPUs	Improv.
Insufficient	24	266,471.97	266,471.97	0.00%
Adequate	80	56,732.55	55,624.84	1.95%

With only 24 vehicles for 500 customers (theoretical requirement: ~ 73), all solutions are infeasible, and multi-GPU provides no benefit. With 80 vehicles (10% margin), the problem becomes feasible, and multi-GPU achieves 1.95% improvement. This demonstrates that multi-GPU effectiveness depends not only on problem scale but also on problem characteristics and configuration.

6.5 Generality and Standard Benchmarks

6.5.1 12 Problem Types

Table 13 validates cuGenOpt’s ability to solve diverse problem types with a single framework.

All 12 problems reach global optimality; all 5 encoding variants succeed.

6.5.2 Standard Benchmarks

Table 14 presents three-GPU results on standard benchmark libraries.

Table 13: Generality validation (12 problems, 5 encodings, 2000 generations).

Problem	Encoding	Optimal	cuGenOpt	5/5 hit
TSP5	Perm	18	18.00	✓
Knapsack6	Binary	30	30.00	✓
Assign4	Perm	13	13.00	✓
Schedule3x4	Binary	21	21.00	✓
CVRP10	Perm-MR	200	200.00	✓
LoadBal8	Integer	14	14.00	✓
GraphColor10	Integer	0	0.00	✓
BinPack8	Integer	4	4.00	✓
QAP5	Perm	58	58.00	✓
VRPTW8	Perm-MR	—	162.00	✓
JSP3x3 (Int)	Integer	12	12.00	✓
JSP3x3 (Perm)	Perm-MS	12	12.00	✓

Table 14: Standard benchmark results (mean gap%, 30 s, 5 seeds).

Instance	Problem	T4	V100	A800	T4 g/s	V100 g/s	A800 g/s
nug12	QAP	0.00	0.00	0.00	8,197	14,285	15,121
tai15a	QAP	0.004	0.004	0.00	6,605	12,854	13,530
ft06	JSP	0.00	0.00	0.00	1,622	4,930	4,956
ft10	JSP	0.84	0.84	0.47	519	1,538	1,725
knapPI_1	Knapsack	0.00	0.00	0.00	2,965	6,107	11,215
R101	VRPTW	2.12	2.44	2.09	2,060	2,507	2,039
C101	VRPTW	0.81	0.20	0.20	1,997	2,280	1,950
RC101	VRPTW	5.47	6.29	5.47	2,047	2,427	2,010

Small-data problems (QAP, JSP, Knapsack) converge to or near global optimality on all three GPUs. A800 achieves the highest throughput across all instances (e.g., Knapsack: 11,215 gens/s, $3.78\times$ T4).

VRPTW (Solomon instances) produces feasible solutions (zero penalty) in all runs. C101 converges to 0.20% gap on both V100 and A800. VRPTW throughput is similar across GPUs ($\sim 2,000$ – $2,500$ gens/s), indicating that multi-constraint evaluation is compute-bound rather than memory-bound.

6.6 Framework Optimization Ablation

Table 15 presents cumulative optimization effects on pcb442 ($n=442$, 30 s).

Table 15: Cumulative optimization ablation on pcb442.

Version	Gap%	Gens/s	Description
Baseline	36.35	116	Random init, per-gen AOS update
+Heuristic init + AOS freq	6.32	395	Bidirectional init, AOS interval 1 \rightarrow 10
+Profile weights	5.78	408	L1 prior-driven operator presets
+Pop. adaptation	6.15	625	Pop 64 \rightarrow 32, gens/s +53%
<i>Cross-hardware (same code)</i>			
V100	5.29	805	Pop=32, L2=6 MB
A800	4.73	1,348	Pop=32, L2=40 MB

6.6.1 Per-Optimization Analysis

Attribute-agnostic heuristic initialization (Baseline \rightarrow +Init): pcb442 gap drops from 36% to 6%, the single largest improvement.

AOS update frequency (Baseline \rightarrow +Init): gens/s improves 9–240% for all $n > 100$ instances. ch150 stagnation is broken (2.30% \rightarrow 0.95%).

Problem profile weights (+Init \rightarrow +Profile): pcb442 gap from 6.32% to 5.78%; tsp225 from 4.15% to 3.67%. 3-opt initial weight drops from 0.50 to 0.05 at Large scale.

Population adaptation: V100 pcb442 gap from 57.88% to 5.29% (−91%), effectively preventing L2 cache thrashing.

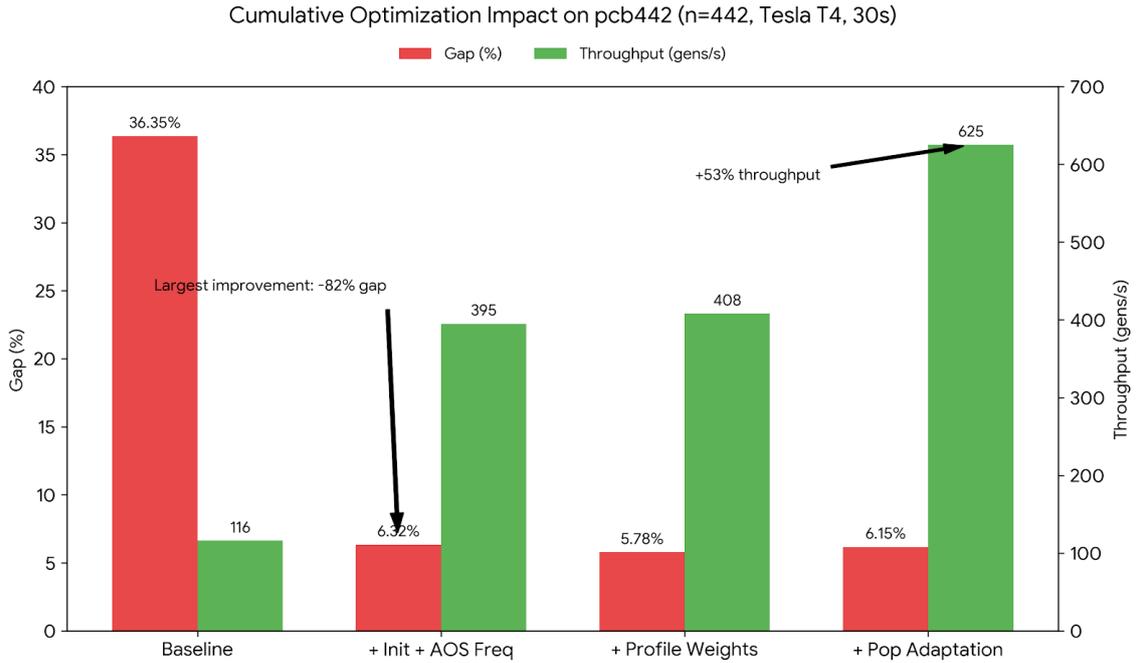


Figure 9: Cumulative optimization impact on pcb442 (Tesla T4, 30s). Heuristic initialization provides the largest improvement (−82% gap). AOS update frequency optimization boosts throughput by 240%. Profile-driven weights and population adaptation further refine solution quality and throughput. Final configuration achieves 6.15% gap at 625 gens/s.

6.6.2 Shared Memory Auto-Extension Effect

Table 16: Shared memory extension impact on VRPTW (T4, 30s).

Instance	Before g/s	After g/s	Δ g/s	Δ gap
R101	1,150	2,060	+79%	−0.29 pp
C101	1,140	1,997	+75%	−1.53 pp
RC101	1,133	2,047	+81%	−0.83 pp

VRPTW at $n=100$ requires ~ 60 KB shared memory, exceeding the 48 KB default but within T4’s 64 KB hardware limit. Auto-extension enables the shared memory path, yielding 75–81% throughput improvement.

6.6.3 Partition Encoding Heuristic Initialization

VRPTW uses Perm-Partition encoding. Its heuristic initialization clusters customers geographically and sorts within each route by time window. C101 (clustered customers) gap drops from 2.34% to 0.20% (−91%); R101/RC101 (random/mixed) remain unchanged, with no regression.

6.7 User-Defined Operator Effectiveness

To validate the user-defined operator registration mechanism (Section 3.3.2), we compare TSP solution quality with and without custom operators on an RTX 3080 Ti. Custom operators include TSP-specific delta-evaluation 2-opt, or-opt, and node-insert, accessing the distance matrix via `prob->d_dist`.

Table 17: User-defined operator effect (RTX 3080 Ti, 5 seeds).

Instance	Limit	Built-in only	+Custom	Improvement
TSP-50	1 s	0.00%	0.00%	—
TSP-100	2 s	0.42%	0.37%	−12%
TSP-150	3 s	1.85%	1.22%	−34%

TSP-50 converges to optimality regardless; custom operators provide no additional benefit. On TSP-100 and TSP-150, custom operators avoid full objective recomputation through delta evaluation, exploring more neighborhoods within the same time budget, improving best gap by 12% and 34% respectively. This validates the design intent: providing a specialization channel for domain experts atop the general-purpose framework.

Safety mechanism validation: registering an operator with a syntax error triggers automatic exclusion with a `RuntimeWarning`, and the solver falls back to built-in operators with identical results.

6.8 Multi-Objective Optimization Modes

To validate the framework’s multi-objective capabilities, we conducted experiments on bi-objective and tri-objective VRP instances using both **Weighted** (scalarization) and **Lexicographic** (priority-based) comparison modes. The test instance A-n32-k5 (31 customers, capacity=100, known optimal=784) was used with fixed configuration: pop=64, gen=1000, 2 islands.

6.8.1 Bi-Objective VRP: Distance vs. Vehicle Count

Table 18 shows results for Weighted mode with different weight configurations. With weights [0.9, 0.1] prioritizing distance, the solver achieved the known optimal solution of 784 within 1000 generations, demonstrating that the weighted scalarization effectively guides the search toward user-specified trade-offs.

Table 18: Weighted mode results on bi-objective VRP (A-n32-k5). Tesla V100S, 1000 generations, seed=42.

Weights	Distance	Vehicles	Gap%	Time(s)	Gens
[0.9, 0.1]	784.00	5.00	0.0%	0.4	1000

Table 19 presents results for Lexicographic mode with different priority orders and tolerances. When distance is prioritized with tolerance=50, the solver achieves 814 (gap=3.8%). However, when vehicle count is prioritized over distance, the distance objective increases dramatically to 1644 (gap=109.7%), demonstrating that the lexicographic ordering strictly enforces the specified priority hierarchy.

Table 19: Lexicographic mode results on bi-objective VRP (A-n32-k5). Tesla V100S, 1000 generations, seed=42.

Priority	Tolerance	Distance	Vehicles	Gap%
[dist, veh]	[100, 0]	962.00	5.00	22.7%
[dist, veh]	[50, 0]	814.00	5.00	3.8%
[veh, dist]	[0, 100]	1644.00	5.00	109.7%

6.8.2 Tri-Objective VRP: Distance, Vehicles, and Load Balancing

We further tested tri-objective optimization by adding a third objective: minimizing the maximum route length (load balancing). Table 20 shows that Weighted mode with [0.6, 0.2, 0.2] achieves distance=829 (gap=5.7%) while maintaining reasonable load balance (max route=238). When vehicle count is prioritized in Lexicographic mode, both distance and max route length increase significantly (1543 and 451 respectively), confirming that the priority-based comparison strictly follows the specified objective ordering.

Table 20: Tri-objective VRP results (A-n32-k5). Objectives: distance, vehicle count, max route length. Tesla V100S, 1000 generations, seed=42.

Mode	Config	Distance	Vehicles	Max Route
Weighted	[0.6, 0.2, 0.2]	829.00	5.00	238.00
Lexicographic	[dist, veh, max]	881.00	5.00	259.00
Lexicographic	[veh, dist, max]	1543.00	5.00	451.00

6.8.3 Multi-GPU Compatibility Verification

To verify that the multi-objective comparison logic works correctly in multi-GPU scenarios, we ran the bi-objective VRP with 2 V100 GPUs. Table 21 shows that both Weighted and Lexicographic modes function correctly: the coordinator properly compares solutions from different GPUs using the configured comparison mode and selects the global best. For Weighted mode, GPU1 achieved the optimal solution (784), which was correctly identified as the final result. For Lexicographic mode, GPU0’s solution (840) was correctly selected over GPU1’s (962) according to the priority rules.

Table 21: Multi-GPU compatibility verification for multi-objective modes (A-n32-k5, 2×V100S).

Mode	GPU	Distance	Vehicles	Selected
Weighted [0.7, 0.3]	GPU0	796.00	5.00	
	GPU1	784.00	5.00	✓
	Final	784.00	5.00	
Lexicographic [dist, veh]	GPU0	840.00	5.00	✓
	GPU1	962.00	5.00	
	Final	840.00	5.00	

These experiments confirm that cuGenOpt supports genuine multi-objective optimization with two distinct comparison modes, and that the comparison logic functions correctly in both single-GPU and multi-GPU scenarios.

7 Discussion

7.1 The Generality–Performance–Usability Triangle

cuGenOpt’s design is guided by a central thread: finding the optimal balance among generality, performance, and usability.

Generality vs. performance. The framework is problem-agnostic; all built-in operators act in encoding space. This yields broad coverage (12 problem types) but also a performance gap against specialized solvers (cuGenOpt 4.75% vs. OR-Tools 1.87% on large-scale TSP). The user-defined operator registration mechanism (Section 3.3.2) offers a middle path: the framework remains general, but domain experts can inject problem-specific search logic, reducing TSP-150 gap from 1.85% to 1.22%.

Performance vs. usability. The JIT compilation pipeline (Section 5.1) wraps CUDA performance behind a Python API, freeing users from GPU memory management and thread models. The cost is a ~ 9 -second first-compilation latency, reduced to ~ 0.1 seconds on cache hits.

Usability vs. generality. The LLM-based modeling assistant (Section 5.3) further lowers the barrier from CUDA snippets to natural language, but has been validated only on simple problems. Natural-language descriptions of complex constraints may be ambiguous, requiring interactive confirmation mechanisms.

7.2 Memory Hierarchy as Performance Determinant

Experiments reveal that the GPU memory hierarchy—not raw compute power—determines cuGenOpt’s performance characteristics. Three distinct operating regimes are identifiable:

Shared memory regime ($n \leq 100$, or $n \leq 150$ on A800): data is on-chip; throughput is compute-bound; performance scales near-linearly with SM count.

L2 cache regime ($100 < n \leq 300$): data overflows shared memory but fits in L2 at moderate population sizes. Performance critically depends on the population–L2 balance.

DRAM regime ($n > 300$): working sets exceed per-block L2 capacity; throughput is bandwidth-bound with diminishing returns from larger GPUs.

Shared memory auto-extension effectively shifts the boundary between the first two regimes: VRPTW on T4 moves from L2 to shared memory regime, with 75–81% throughput improvement.

7.3 Layered Adaptive Design

The combination of L1 static priors and L3 runtime AOS proves effective: L1 sets a reasonable starting point via problem profiling; L3 continuously corrects via EMA during execution. Their division of labor avoids the “cold start” problem of AOS beginning from uniform weights.

The L2 landscape probing layer represents a potential enhancement to the three-layer architecture. This layer would extract statistical features from initial populations to inform strategy selection. However, establishing reliable mappings between landscape features and optimal strategies requires extensive experimental validation across diverse problem types and scales, making it a promising but data-intensive direction for future research.

7.4 Multi-GPU Effectiveness and Scalability

Large-scale experiments ($n=1000$) reveal several insights about multi-GPU cooperative solving:

Scale-dependent effectiveness. Multi-GPU improvement increases with problem scale for TSP, from 1.24% at $n=300$ to 3.51% at $n=1000$ (with CUDA Graph enabled). This supports the hypothesis that larger search spaces benefit more from parallel exploration diversity.

CUDA Graph amplification effect. The same TSP $n=1000$ problem shows 3.51% multi-GPU improvement with CUDA Graph vs. 0.66% without. Kernel launch overhead reduction

appears to amplify the benefits of parallel search, though the exact mechanism requires further investigation.

Problem feasibility as prerequisite. For VRP, multi-GPU effectiveness critically depends on problem configuration. With insufficient vehicles (24 for 500 customers), all solutions are infeasible, yielding 0% multi-GPU benefit. With adequate capacity (80 vehicles), multi-GPU achieves 1.95% improvement. This demonstrates that parallel search diversity can only improve solution quality when the problem admits feasible solutions.

8 Technical Limitations and Observations

8.1 Solution Structure Size Constraints

For VRP problems, the Solution structure size ($D1 \times D2 \times 4$ bytes) is constrained by CUDA’s kernel parameter passing mechanism. Experiments show that configurations exceeding approximately 80 KB may encounter `invalid argument` errors.

Table 22 summarizes observed limits.

Table 22: VRP Solution structure size observations.

D1	D2	Size (KB)	Status	Note
24	512	48	Success	—
80	128	40	Success	Optimal balance
160	128	80	Success	Near limit
32	1024	128	Failed	Exceeds limit

Recommendation: Keep Solution size below 80 KB by balancing D1 and D2. For example, VRP with 300 vehicles can use $D1=300$, $D2=64$ (76 KB) rather than $D1=150$, $D2=128$ (76 KB).

8.2 CUDA Graph Compatibility

CUDA Graph, which reduces kernel launch overhead, shows compatibility issues at large problem scales. Table 23 summarizes observations.

Table 23: CUDA Graph compatibility observations (TSP).

Problem size	CUDA Graph enabled	CUDA Graph disabled
$n=1000$	Success	Success
$n=1200$	Failed (illegal memory)	Success
$n=1500$	Failed	Success
$n=2000$	Failed	Failed (other issue)

For $n \geq 1200$, enabling CUDA Graph triggers `illegal memory access` errors during graph execution. The root cause remains under investigation, but may relate to graph capture limitations with large working sets or relation matrix sizes.

Recommendation: For problems with $n \geq 1200$, disable CUDA Graph (`use_cuda_graph=false`). The performance impact is estimated at 5–10% throughput reduction, which is acceptable for correctness.

8.3 Maximum Tested Scale

The largest successfully tested problems are:

- TSP: $n=1500$ (CUDA Graph disabled)

- VRP: $n=1000$ with 160 vehicles (CUDA Graph disabled)

TSP $n=2000$ encounters errors even with CUDA Graph disabled, suggesting other architectural limits (possibly relation matrix size or internal data structures). Further investigation is needed to identify and address these constraints.

9 Limitations

1. **Multi-GPU communication overhead:** The current simplified multi-GPU approach (independent execution with best-solution selection) avoids inter-GPU communication but may miss opportunities for solution exchange during evolution. More sophisticated multi-GPU strategies remain to be explored.
2. **Population sizing heuristic:** L2-cache-aware population adaptation prevents catastrophic cache thrashing but may over-allocate on GPUs with ample L2 (A800 tsp225/lin318 gap regression). A more nuanced heuristic considering both cache capacity and convergence requirements is needed.
3. **L2 landscape probing requires further investigation:** The L2 layer would extract landscape features from initial populations to guide strategy selection. Validating this approach requires comprehensive experiments to establish feature-strategy correlations across problem types.
4. **CUDA Graph compatibility:** Large-scale problems ($n \geq 1200$) require disabling CUDA Graph, with 5–10% throughput impact. Identifying and resolving the root cause would improve large-scale performance.
5. **Solution structure size limits:** VRP problems are constrained to Solution sizes below ~ 80 KB, limiting the maximum number of vehicles and customers per route. Alternative memory management strategies (e.g., device-side allocation) may relax this constraint.
6. **Custom operator barrier:** User-defined operators still require CUDA code, posing a barrier for pure-Python users. Future work may explore LLM-assisted operator generation.

10 Conclusion and Future Work

cuGenOpt demonstrates that GPU-accelerated metaheuristics can serve as a practical general-purpose optimization tool, bridging the gap between specialized solvers and general MIP frameworks. The framework achieves this through three key contributions:

(1) **Adaptive architecture:** L1 static priors, L2 landscape probing (design stage), and L3 runtime AOS form a layered adaptation system that balances exploration and exploitation across diverse problem types.

(2) **Memory hierarchy awareness:** Shared memory auto-extension and L2-aware population adaptation automatically tune performance to GPU characteristics, enabling portable performance across T4, V100, and A800.

(3) **User-facing interface:** JIT compilation, LLM-assisted modeling, and user-defined operators provide multiple entry points for users with varying expertise levels, from natural language to CUDA kernels.

Experiments validate these contributions across 12 problem types, with cuGenOpt matching or exceeding general MIP solvers and approaching specialized solver performance on medium-scale instances. Large-scale experiments ($n=1000$) demonstrate framework scalability and multi-GPU effectiveness (up to 3.51% improvement), though technical limitations (CUDA Graph compatibility, Solution size constraints) remain to be addressed.

Future directions include: (1) investigating L2 landscape probing with comprehensive experimental validation; (2) exploring advanced multi-GPU strategies with inter-GPU solution exchange; (3) investigating CUDA Graph compatibility issues at large scales; (4) extending LLM-assisted modeling to complex constraints; and (5) expanding the standard benchmark coverage to include more large-scale instances ($n > 1000$).

cuGenOpt is open-source and available at [URL to be added]. We invite the community to explore, extend, and apply the framework to new problem domains.

References

- [1] Philippe Augerat, José Manuel Belenguer, Enrique Benavent, Ángel Corberán, Denis Naddef, and Giovanni Rinaldi. Computational results with a branch and cut code for the capacitated vehicle routing problem. *Research Report 949-M, IMAG, Grenoble*, 1995.
- [2] John E Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.
- [3] Rainer E Burkard, Stefan E Karisch, and Franz Rendl. QAPLIB—a quadratic assignment problem library. <https://qaplib.mgi.polymtl.ca/>, 1997. Accessed: 2026.
- [4] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.
- [5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [6] Audrey Delevacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel GPU implementation of iterated local search for the travelling salesman problem. *Lecture Notes in Computer Science*, 7744:372–383, 2013.
- [7] Agoston E Eiben and James E Smith. *Introduction to Evolutionary Computing*. Springer, 2nd edition, 2015.
- [8] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. *Evolutionary Computation*, 18(4):579–615, 2010.
- [9] Google. OR-Tools: Google’s operations research tools. <https://developers.google.com/optimization>, 2024. Accessed: 2026-03-05.
- [10] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [11] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.
- [12] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6. ACM, 2015.
- [13] Ke Li, Álvaro Fialho, Sam Kwong, and Qingfu Zhang. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 18(1):114–130, 2014.

- [14] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Gpu computing for parallel local search metaheuristic algorithms. *Computers & Operations Research*, 40(5):1555–1567, 2013.
- [15] Jorge Maturana and Frédéric Saubion. Autonomous operator management for evolutionary algorithms. In *Journal of Heuristics*, volume 17, pages 167–189. Springer, 2011.
- [16] Clair E Miller, Albert W Tucker, and Richard A Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329, 1960.
- [17] NVIDIA Corporation. NVIDIA cuOpt: GPU-accelerated decision optimization. <https://developer.nvidia.com/cuopt-logistics-optimization>, 2025. Accessed: 2025.
- [18] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the 31st Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.
- [19] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.
- [20] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.
- [21] David Pisinger and Stefan Ropke. Large neighborhood search. *Handbook of Metaheuristics*, pages 99–127, 2019.
- [22] Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
- [23] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.
- [24] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.
- [25] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7(1):33–47, 1999.
- [26] Yuren Zhou and Ying Tan. A GPU-accelerated evolutionary computation framework. *Soft Computing*, 22(21):7363–7379, 2018.
- [27] Zuse Institute Berlin. SCIP: Solving constraint integer programs. <https://www.scipopt.org/>, 2024. Accessed: 2026-03-05.