Investigation of Performance and Scalability of a Quantum-Inspired Evolutionary Optimizer (QIEO) on NVIDIA GPU

Aman Mittal
High-Performance Computing Team
BosonQ Psi (BQP) Corporation
New York, USA
aman.mittal@bqpsim.com

Kasturi Venkata Sai Srikanth
High-Performance Computing Team
BosonQ Psi (BQP) Corporation
New York, USA
kasturi.srikanth@bqpsim.com

Ferdin Sagai Don Bosco
High-Performance Computing Team
BosonQ Psi (BQP) Corporation
New York, USA
https://orcid.org/0000-0002-5507-5304

Abhishek Singh
High-Performance Computing Team
BosonQ Psi (BQP) Corporation
New York, USA
abhishek.singh@bqpsim.com

Rut Lineswala
Chief Technical Officer
BosonQ Psi (BQP) Corporation
New York, USA
rutlineswala@bqpsim.com

Abhishek Chopra
Chief Science Officer & Chief Executive Officer
BosonQ Psi (BQP) Corporation
New York, USA
abhishekchopra@bqpsim.com

2025 Supercomputing India (SCI) | 979-8-3315-5758-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/SCI68648.2025.11333835

***Abstract*—Quantum-inspired evolutionary optimization leverages quantum computing principles like superposition, interference, and probabilistic representation to enhance classical evolutionary algorithms with improved exploration and exploitation capabilities. Implemented on NVIDIA Tesla V100 SXM2 GPUs, this study systematically investigates the performance and scalability of a GPU-accelerated Quantum-Inspired Evolutionary Optimizer (QIEO) applied to large-scale 0/1 Knapsack problems. By exploiting CUDA's parallel processing capabilities, particularly through optimized memory management and thread configuration, significant speedups and efficient utilization of GPU resources is demonstrated. The analysis covers various problem sizes, kernel launch configurations, and memory models including constant, shared, global, and pinned memory, alongside extensive scaling studies. The results reveal that careful tuning of memory strategies and kernel configurations is essential for maximizing throughput and efficiency, with constant memory providing superior performance up to hardware limits. Beyond these limits, global memory and strategic tiling become necessary, albeit with some performance trade-offs. The findings highlight both the promise and the practical constraints of applying QIEO on GPUs for complex combinatorial optimization, offering actionable insights for future large-scale metaheuristic implementations.**

***Index Terms*—Metaheuristic Optimization methods, Quantum-Inspired Evolutionary Optimizer (QIEO), NVIDIA GPUs, CUDA, Execution model, Memory model**

I. Introduction

Optimization methods aim to identify best solutions across diverse disciplines [1, 2, 3], but increasing problem size and complexity leads to multimodal, nonconvex, and multidimensional landscapes. While gradient-based approaches risk converging to local optima [4] and gradient-free methods face barren plateaus [5], metaheuristic optimizers [6, 7]—including Genetic Algorithms (GA) [8], Ant Colony Optimization (ACO) [9], and Particle Swarm Optimization (PSO) [10, 11]—are well suited to challenging NP-complete problems [12], though their serial operations limit scalability [13, 14]. Recently, a Quantum-Inspired Evolutionary Optimizer (QIEO), introduced by Eswara Sai et al. [15], has demonstrated major advantages over classical genetic algorithms for complex functions. These algorithms integrate quantum computing concepts such as, qubits (representing solutions as probabilistic superpositions of states) and quantum gates (for updating these superpositions), into classical evolutionary frameworks, which enables it to maintain population diversity more effectively and explore the solution space more broadly, often leading to faster convergence and better solution quality compared to their classical counterparts [16]. However, QIEOs can be computationally intensive for large problems or extensive meta-optimization, making NVIDIA GPUs—with their highly parallel SIMT architecture and high memory bandwidth—an effective platform for accelerating population-wide fitness evaluations.

This research aims to systematically evaluate the performance and scalability of a GPU-accelerated QIEO implemented on the NVIDIA Tesla V100 SXM2 16GB GPU. Specifically, it investigates the benefits of parallelizing key computational components of QIEO, such as fitness evaluation and population updates, on large-scale 0/1 knapsack problems. The study seeks to analyze the impact of different CUDA memory models, thread-block configurations, and population sizes on execution time and efficiency, ultimately providing insights and guidelines for optimizing QIEO performance on modern many-core GPU architectures.

II. Computational Complexity of QIEO

Specific details on the QIEO algorithm can be found in the original investigation [15]. The focus of the present section is

on the computational complexity analysis of the core algorithmic steps in the QIEO method in terms of its population-based evolutionary parameters. These are

- N_G: Number of generations to convergence,
- N_p: Population size (no. of chromosomes per generation)
- N_g: Dimensionality (no. of genes per chromosome),
- C_f: Cost of a single objective function evaluation
- C_c: Cost of a single constraint function evaluation

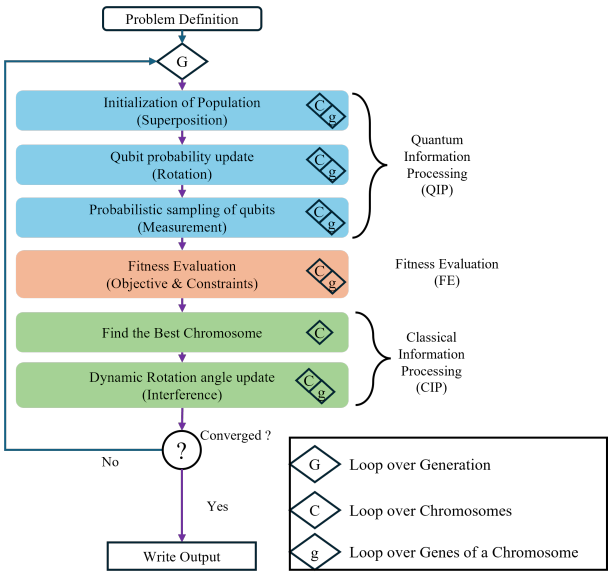

Fig. 1: Modified flowchart of the QIEO algorithm adapted from ref. [15].

The core components of QIEO are depicted in Fig. 1. These include

Quantum Information Processing (QIP)

- initializing qubits to an equal superposition state: $O(N_p \cdot N_g)$
- update qubit probability: $O(N_p \cdot N_g)$
- probabilistic sampling to collapse qubits: $O(N_p \cdot N_g)$

Fitness Evaluation (FE)

- Compute the objective: $O(N_p \cdot C_f)$
- Compute the constraint: $O(N_p \cdot C_c)$

Classical Information Processing (CIP)

- generational best chromosome determination: $O(N_p)$
- dynamic rotation angle adjustment: $O(N_p \cdot N_g)$

The aggregate per-generation complexity is computed as,

- **QIP:** $O(N_p \cdot N_g)$
- **FE :** $O(N_p \cdot (C_f + C_c))$
- **CIP:** $O(N_p + N_p \cdot N_g)$

Considering the Memory (Space) Complexity per generation,

- each chromosome stores a vector of qubits: $O(N_p \cdot N_g)$
- Population fitness and auxiliary data: $O(N_p)$

Thus, in each generation, the time complexity is $O(N_p \cdot (C_f + C_c + N_g))$ and the space complexity is $O(N_p \cdot N_g)$.

III. Problem Statement: 0/1 Knapsack Problem

A. Formal Definition

Given a set of n items, each item i has a weight w_i and a value v_i. The objective is to determine the subset of items to include in a knapsack such that the total value is maximized, while the total weight does not exceed a given capacity W. Each item can be included (1) or excluded (0); fractional selection is not allowed. In the present study, weights and values of the the items are assumed to be integer datatype for simplicity of memory management calculations. The problem can be formally stated as

- **Input:**
  - A set of n items, where each item i has:
$$v_i \in \mathbb{Z}^+, \quad w_i \in \mathbb{Z}^+$$
  - Knapsack capacity:
$$W \in \mathbb{Z}^+$$
- **Objective:**
$$\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$
subject to **Constraint:**
$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$
where
$$x_i \in \{0, 1\} \quad \forall i = 1, \ldots, n$$

Despite its apparent simplicity, the 0/1 knapsack problem is proven to be NP-complete [17, 18], making it computationally intractable for large instances using a naive exhaustive search [19].

This problem is fundamental in fields such as logistics for optimizing cargo loading, in project management for allocating budgets efficiently, and in finance for constructing investment portfolios that balance risk and return. It is also used in network engineering for bandwidth allocation, and in structural engineering for material selection to maximize strength while minimizing weight, making it a versatile tool for solving discrete optimization problems in engineering contexts.

It remains a critical research focus due to the need for efficient, scalable algorithms capable of handling high-dimensional and real-world combinatorial optimization applications. It is a particularly suited use case for metaheuristic optimization [20, 18, 21] that can take advantage of parallel and high-performance computing platforms.

In the context of QIEO algorithm, a chromosome is a potential collection of items that is selected to be placed in the knapsack. The weights and values of the items are then used to evaluate the fitness of that chromosome, which is expressed in eq. 1. The chromosome's fitness value is penalized if the constraint specified in eq. 2 is violated.

B. CUDA Optimization Strategies

The GPU utilized in this study is the NVIDIA Tesla V100 SXM2 16GB, a high-performance accelerator designed for scientific computing and large-scale parallel processing. The appropriate strategy depends on the V100 GPU's micro-architecture and specifications which are specified in the tab. I.

TABLE I: Specification of the hardware utilized in the present study

Category	Specification
CPU Chipset	Intel Core i9
GPU Card	Tesla V100-SXM2-16GB
Compute Capability	7.0
Total Global Memory	16144 MB
Shared Memory per Block	48 KB
Constant Memory	64 KB
Max Threads per Block	1024
Max Block Dimensions	(1024, 1024, 64)
Max Grid Dimensions	(2147483647, 65535, 65535)
Warp Size	32
Warp Granularity	4
No. of Streaming Multiprocessors (SM)	80
Max number of blocks per SM	32

Featuring 80 streaming multiprocessors, a compute capability of 7.0, and 16GB of high-bandwidth memory, the V100 offers ample computational and memory resources for demanding optimization tasks. Its architecture supports up to 1024 threads per block, substantial global and shared memory, and advanced memory management features such as constant and pinned memory, making it particularly well-suited for implementing and accelerating metaheuristic algorithms like the QIEO explored in this research. To maximize performance on this GPU, several CUDA-specific optimization techniques were employed.

Achieving peak performance in a CUDA implementation requires careful optimization of both the execution and memory models.

In CUDA's execution model, increasing threads per block reduces the number of thread blocks that can be concurrently scheduled on each Streaming Multiprocessor (SM). The NVIDIA V100 GPU features 80 SMs, each capable of running up to 32 blocks concurrently, totaling 2,560 concurrent blocks in theory. However, practical concurrency is limited by compute and memory bandwidth constraints imposed by kernel execution. Key parameters such as threads per block and the number of CUDA blocks must be selected to maximize GPU occupancy and enable efficient parallel execution of fitness evaluations and qubit updates.

This work implements chromosome-level parallelism, assigning each chromosome to a single CUDA thread—satisfying the relation $Threads\ per\ block \times CUDA\ blocks = chromosomes$. Consequently, increasing the chromosome count scales the total threads accordingly. Since each thread processes a chromosome through kernels involved in QIEO—Quantum Information Processing (QIP), Fitness Evaluation (FE), and Classical Information Processing (CIP). All of these kernels involve a loop over gene $O(N_g)$. Thus, increasing the number of items proportionally increases the workload per thread.

This chromosome-level strategy reduces the time complexity per generation approximately to the evaluation time of a single chromosome, $O(C_f + C_c)$, assuming full parallelism coverage. The overall runtime corresponds to the longest per-thread evaluation plus overheads like memory access and synchronization. The QIP and CIP steps complexity reduce from $O(N_p \cdot N_g)$ to $O(N_g)$, while space complexity remains at $O(N_p \cdot N_g)$. Effective memory management, however, can minimize overheads in data access and updates.

On the memory front, leveraging the V100's fastest, low-latency memories—constant memory and shared memory—is critical. These memories enable coalesced access patterns within thread blocks, significantly reducing costly host-to-device (HtoD) and device-to-host (DtoH) data transfers by keeping frequent computations on the GPU. When using global memory, pinned host memory optimizes data transfer efficiency compared to paged memory.

Increasing the number of items also increases the memory transfer burden, particularly to the FE kernel, which accesses weight and value data. With integer (4 bytes) weights and values, each thread must retrieve approximately $(4 + 4) \times N$ bytes of information, where N is the number of items. This data transfer is a known bottleneck in GPU programming.

NVIDIA GPUs provide various memory types differing in access speed and scope. Constant memory, ideal for read-only access to weights and values, is extremely fast but limited in size—approximately 64 KB in the V100. Adhering to the recommended 80% utilization accommodates up to about 6,553 items:

$$8 \times N = 64 * 1024 * 0.8 \Rightarrow N = 6553.6 \approx 6553\ items$$

Shared memory, shared only among threads of a single SM, offers larger capacity (48 KB in V100) but cannot be accessed across SMs. With an 80% usage threshold, it supports up to about 4,915 items:

$$8 \times N = 48 * 1024 * 0.8 \Rightarrow N = 4915.2 \approx 4915\ items$$

For item counts exceeding shared memory capacity, tiling techniques partition data into manageable chunks—typically tiles of size 128—to ensure coalesced access and prevent bank conflicts. When problem sizes exceed this limit significantly (e.g., 10240 items), global memory use becomes necessary, incurring expensive HtoD transfers. This overhead can be partially mitigated by employing pinned global memory to improve data transfer rates and memory access patterns.

IV. Performance Metrics

The computational complexity of QIEO operations along with the QIEO process (refer fig.1) indicate that the algorithm is amenable to massive parallel implementation. In such scenarios, scalability is widely used to measure the ability of hardware and solver to deliver computational power when the number of resources increases.

In the strong scaling analysis, efficiency of the solver is calculated by comparing the speedup attained as the number processors are increased for a problem of constant size. This analysis is based on Amdahl's law, which posits diminishing returns as the amount of parallelizable work decreases and the serial footprint is exposed [22, 23]. For a weak scaling analysis, both the number of processors and the problem size increases, maintaining a constant workload per processor. Weak scaling aligns with Gustafson's law [24], where scaled speedup is computed based on the workload of a scaled problem size, as opposed to Amdahl's law, which focuses on a fixed problem size. Efficiency is maintained if the runtime remains unchanged as more processors handle a proportionally larger problem.

However, Amdahl's and Gustafson's laws are predominantly applicable to multi-core hardware, whereas the current study focuses on a many-core application. The extension of these laws to the present hardware and solver needs to be explicitly defined.

The sensitivity of the number of blocks to the execution time for a fixed number of chromosomes and genes is representative of increasing the computational resource for a fixed problem; essentially defining a strong scaling. Thus, strong scaling for the present hardware and solver is defined by increasing the number of threads or threads per block, up to a hardware maximum, to observe the performance scales over multiple Streaming Multiprocessors (SM). On the other hand, increasing the number of chromosomes and the number of blocks represents an increase in the scale (through more chromosomes) and the computational power (through more thread blocks), while keeping the workload per thread constant by maintaining the same number of genes (items). This suffices as a definition for the weak scaling of the QIEO algorithm.

V. Results and Discussion

This comprehensive analysis examines the performance of QIEO algorithm for solving large 0/1 Knapsack problems on an NVIDIA V100 SXM2 16GB GPU, focusing on scaling behavior, memory models, thread-block optimization, and overall hardware utilization. The study encompasses 1023 experimental configurations, namely:

- 3 problem sizes: 4915, 6553, and 10240 genes
- 8 population sizes: 512, 1024, ... to 65536 chromosomes
- 7 memory models:
  - Constant memory
  - Shared memory
  - Tiled (64/128/256) Shared memory
  - Paged Global memory
  - Pinned Global memory
- Multiple thread/block configurations: 1, 2, 4, ... to 1024

A. Strong scaling analysis

Figure 2 illustrates the scaling behavior of time per generation (ms) versus the number of CUDA blocks for four different GPU memory access strategies, using a workload of 4915 genes and 2048 chromosomes. The data reveals two distinct regimes in scaling. In the initial region, increasing the number of CUDA blocks from 16 to approximately 1024 continues to improve performance; all memory models see reduced generation times, and the curves converge below 7ms. This part adheres closely to Amdahl's law, where parallelization effectively speeds-up computation.

Beyond 1024 CUDA blocks, however, performance degrades and the curves diverge, most visibly for Shared Memory, which climbs steeply above 12ms. This marks a transition where overheads from excessive parallelism—such as communication, memory access contention, or kernel launch overhead—overwhelm the benefits from further division of work. Among the memory models, Constant Memory, Global Memory (Paged), and Global Memory (Pinned) show nearly indistinguishable scaling, with Constant Memory just slightly outperforming at the optimal block count. Shared Memory, while competitive at low block counts, witnesses a dramatic loss of efficiency at higher block counts. The inset magnifies the optimal region (128–256 blocks), highlighting how all models reach minimal generation times here, with differences less than 0.5ms.

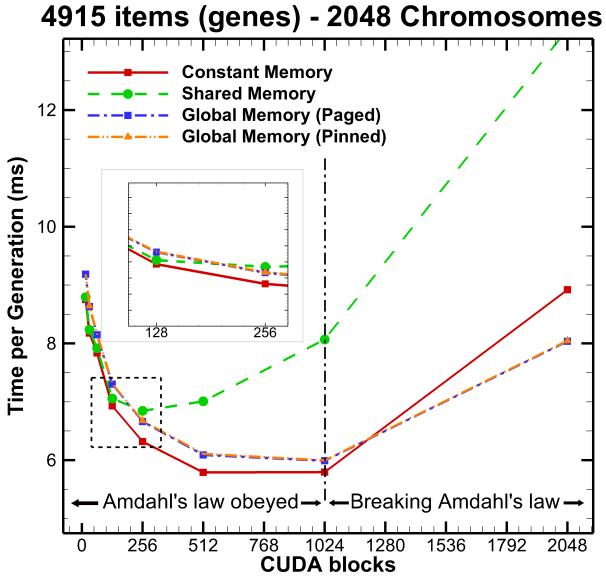

Fig. 2: Strong scaling analysis and relationship with Amdahl's law for 0/1 Knapsack problem with different items with 4915 items.

Overall, this analysis emphasizes that while increasing parallel resources can reduce computational time up to an optimal point, exceeding this leads to diminishing returns or even severe penalties—especially for Shared Memory—demonstrating the practical limits of strong scaling in this evolutionary optimization context.

B. Weak scaling analysis

Figure 3 presents the weak scaling behavior of QIEO on an NVIDIA GPU using global (paged) memory, with 4915 genes and varying chromosome counts. The graph shows execution

time per generation (ms) versus the number of threads per block, with five curves representing different population scales (1024 to 16384 chromosomes).

At low to moderate threads/block (up to 256), the system effectively utilizes parallelism, and Gustafson's law is obeyed: execution times remain relatively stable or rise gently despite increasing population size, reflecting efficient scaling as both workload and resources grow. However, beyond 256 threads/block, the curves for all chromosome counts exhibit a marked, near-linear increase in execution time per generation. This sharp escalation signals a breakdown in weak scaling, where increased parallelism results in diminishing returns and higher runtime.

This is likely caused by hardware limitations such as memory bandwidth saturation, increased inter-thread contention, or overhead from managing large block sizes. The penalties become more pronounced for larger populations, with the 16384 chromosome curve rising steepest, indicating serious efficiency losses at scale.

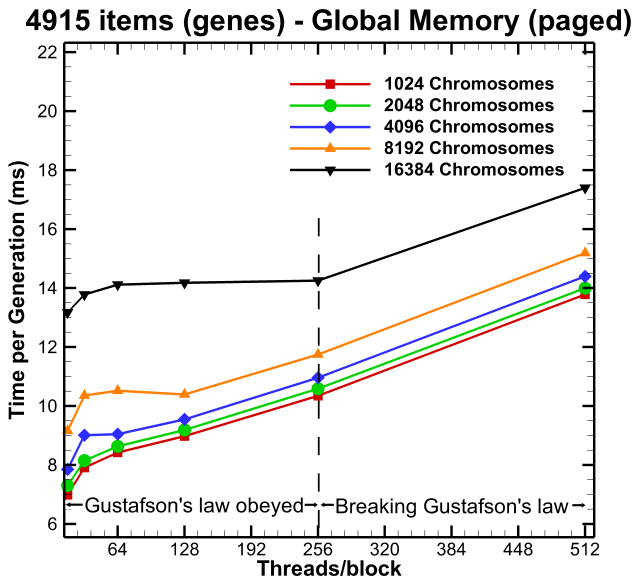

Fig. 3: Weak scaling analysis (Gustafson's law) for 0/1 Knapsack problem with 4915 items.

Overall, the plot underscores that for global memory workloads, optimal performance is attained with carefully chosen thread/block configurations and that pushing parallelism beyond a critical threshold undermines scalability, especially for large evolutionary populations.

C. Population size scaling effect

The graph in fig. 4 demonstrates how time per generation (ms) scales with the number of chromosomes for a fixed problem size (4915 genes) and a constant 32 CUDA blocks, comparing four memory models: Constant Memory, Shared Memory, Global Memory (Paged), and Global Memory (Pinned). The x-axis tracks chromosome count from very small up to 32768, with the y-axis showing the corresponding execution time per generation.

At low chromosome counts, all memory models display polynomial growth, with execution times increasing modestly as population rises, indicating good parallel resource utilization.

As chromosome numbers enter the "Near-Linear" regime (roughly 1000 to 16384 chromosomes), the increase in computation time becomes nearly proportional to population size, reflecting saturation of available parallelism. In this region, Constant Memory maintains a slight edge in efficiency, but all models remain comparable.

When chromosome counts exceed about 16384, the chart shifts into an exponential growth regime: time per generation rises sharply across all models, indicating that hardware resources—such as register file, shared memory, or compute units—are likely overextended, leading to serialization, memory thrashing, or other bottlenecks.

The inset highlights subtleties in the low-to-medium chromosome region, where Shared Memory sometimes lags behind, and Global Memory variants edge out slight advantages at specific points.

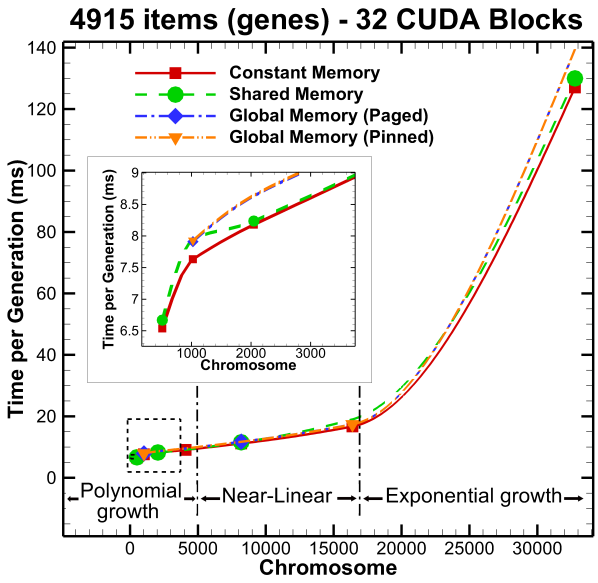

Fig. 4: Scaling of Generation Time with Chromosome Count for 4915-Gene Problem on GPU (32 CUDA Blocks).

Overall, the data reveals that while all GPU memory strategies demonstrate similar scaling up to moderate workloads, their performance diverges and becomes increasingly inefficient under heavy population sizes. Hardware constraints dominate beyond a certain threshold, emphasizing the need for careful population-size management in parallel genetic algorithms to avoid exponential slowdowns, regardless of chosen memory strategy.

D. Memory Model Effect

The bar graph in Fig. 5 presents a comparative analysis of memory model performance for a QIEO running with 4915 genes, as measured by the execution time per generation (ms) across four memory access models—Constant Memory, Shared Memory, Global Memory (Paged), and Global Memory (Pinned)—over increasing population sizes (number of chromosomes: 512, 1024, 2048, 4096). The data demonstrates that Constant Memory consistently delivers the fastest execution times, outperforming the other memory models at every tested population size.

Shared Memory shows moderate efficiency, slightly trailing Constant Memory but maintaining an advantage over the Global Memory models, especially in smaller populations. Both Global Memory (Paged) and Global Memory (Pinned) exhibit higher execution times, with Pinned memory displaying the worst performance, suggesting that the optimization strategy for pinned transfers offers limited benefit in this computation pattern.

Execution times generally increase as the number of chromosomes grows, reflecting higher computational workloads, but the rate of this increase varies by memory strategy: Constant Memory scales most gently, while Global Memory (Pinned) shows the steepest rise.

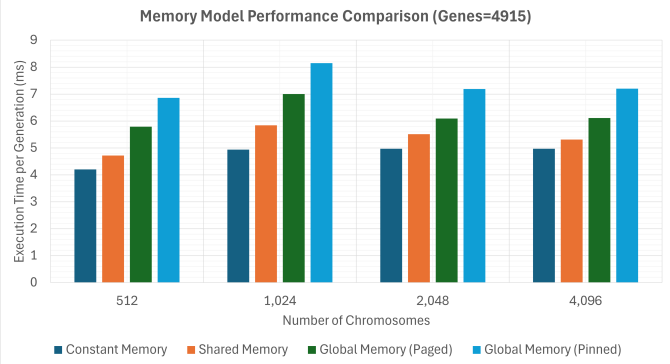

Fig. 5: Comparison of GPU Memory Model Performance Across Chromosome Population Sizes (4915 Genes). Threads per block used is 32.

This visual evidence reinforces industry observations that distilling high-performance out of CUDA genetic workloads depends strongly on memory model selection, with Constant Memory being optimal for both speed and scalability, and also highlights the relative inefficiency of pinned global allocations for these population sizes and problem instances.

E. Item number effect

Figure 6 explores the impact of threads per block on execution time per generation (ms) for QIEO workloads of varying sizes—specifically, 4915, 6553, and 10240 genes. The x-axis categorizes different thread/block configurations (32 to 1024 threads), and annotates the associated chromosome population in parentheses.

At lower threads/block (32, 64, 128), execution times remain modest and increase predictably with the number of genes, demonstrating that workloads scale efficiently in this regime and GPU resources are well utilized.

As the thread count grows (256, 512), there is a gradual rise in execution time, reflecting the higher memory and computational demand of larger populations.

However, a dramatic shift emerges at 1024 threads/block—execution times surge steeply for all gene sizes, with the largest workload (10240 genes) exceeding 280ms, while even the smallest (4915 genes) approaches 140ms.

This exponential increase signals a loss of parallel efficiency, likely due to hardware saturation, register contention, or memory bottlenecks that overwhelm the benefits of large thread blocks. The pattern consistently shows higher gene counts exacerbating the effect.

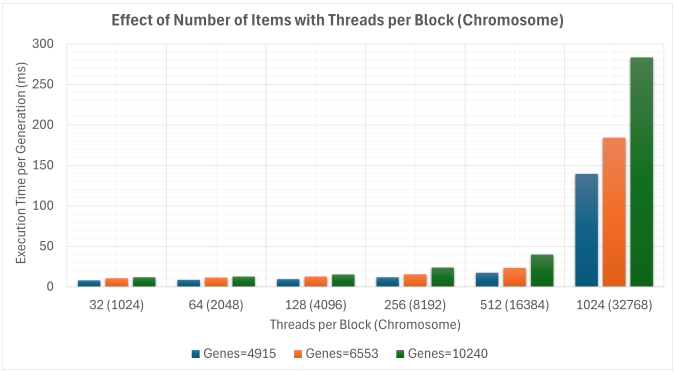

Fig. 6: Impact of Threads per Block and Problem Size on Generation Execution Time for QIEO Algorithm.

Overall, the figure clearly illustrates that while moderate increases in thread/block are beneficial, pushing GPU parallelization too far with large blocks leads to severe performance penalties, and that these penalties grow disproportionately for larger genomic problems.

VI. Key Insights for Large-Scale Implementations

This research provides several quantitative insights into CUDA thread and block configuration when implementing large-scale metaheuristics like QIEO on the NVIDIA V100 SXM2 16GB GPU:

- **Maximum threads per block:**
  NVIDIA V100 supports up to 1024 threads per block. Exceeding moderate thread counts per block (e.g., 256, 512) results in a gradual rise in execution time, while pushing to 1024 threads per block causes a steep increase in generation time due to hardware saturation and parallelization overheads. For instance, with 1024 threads per block, generation execution times can exceed 280ms for 10240 genes, compared to much lower times at smaller thread settings (see fig. 6).
- **Maximum blocks per Streaming Multiprocessor (SM):**

Each SM can run up to 32 blocks concurrently, with 80 SMs available on the V100, giving a theoretical maximum concurrency of 2560 blocks. However, practical scaling is limited by compute and memory bandwidth constraints.

- **Strong scaling behaviour:**
  Increasing CUDA blocks from 16 up to 1024 improves performance and reduces generation times for all memory models, converging to below 7ms for 4915 genes and 2048 chromosomes. Beyond 1024 blocks, however, Shared Memory performance deteriorates steeply (above 12ms), while Constant and Global Memory continue to scale acceptably until hardware limits are reached (refer fig. 2).
- **Chromosome-level parallelism:**
  The relationship Threads per block × CUDA blocks = number of chromosomes is central. Increasing the chromosome count scales the total required threads, mapping each chromosome to a CUDA thread.
- **Population scaling behaviour:**
  For a configuration with 32 blocks and 4915 genes, increasing chromosome counts from below 1000 up to 32768 leads to a scaling regime—from polynomial to near-linear, and then exponential increases in generation time. Once populations exceed 16384 chromosomes, hardware resources begin to saturate, resulting in sharp rises in execution time (see fig. 4).
- **Memory model utilization:**
  Constant Memory (64KB) allows for up to approximately 6553 items (genes), with 80% usage recommended for practical implementations. Shared Memory (48KB per block) supports up to about 4915 items. These limits necessitate strategic transitions to tiling or global/pinned memory for larger problem sizes.

Overall, to maximize scalability and efficiency, it is recommended to utilize up to 256 threads per block or between 512 and 1024 CUDA blocks for strong performance. It is also suggested to avoid pushing configurations to maximal hardware thresholds where overheads increase rapidly.

VII. Conclusion and future work

This comprehensive study presents an in-depth evaluation of a Quantum-Inspired Evolutionary Optimizer (QIEO) tailored for NVIDIA V100 GPUs, focusing on accelerating the solution of high-dimensional 0/1 Knapsack problems using CUDA-based parallelism. The evaluation reveals that GPU acceleration significantly enhances the scalability and speed of QIEO, enabling efficient exploration of large search spaces that would be computationally prohibitive on traditional CPU implementations. Our extensive experiments with varying population sizes, gene counts, and memory configurations demonstrate the critical role of memory hierarchy in achieving optimal performance. Constant memory access consistently delivers the best efficiency for smaller to moderate problem sizes, while global memory with careful tiling handles larger-scale problems effectively. Thread-block configuration and kernel launch parameters are equally influential, with smaller blocks and adequate streaming multiprocessor utilization essential for maximizing parallelism and minimizing execution time.

Despite these advances, the study identifies inherent limitations in strong scaling due to hardware resource saturation and communication overheads at extreme population sizes. These insights emphasize the need for dynamic resource allocation strategies and algorithmic innovations to maintain efficiency on future exa-scale GPU systems. Overall, this work establishes practical guidelines and highlights potential research directions for leveraging GPU-based QIEO frameworks to tackle complex, large-scale combinatorial optimization challenges.

This research contributes to the growing body of knowledge demonstrating how quantum-inspired algorithms, combined with high-performance computing platforms like NVIDIA GPUs, can bridge the gap between theoretical algorithmic advancements and practical, large-scale problem-solving. Future work will focus on exploring more advanced GPU optimization techniques, such as dynamic parallelism and multi-GPU hardware, to push the boundaries of scalability even further. Additionally, applicability of this GPU-accelerated QIEO to specific real-world problems in domains such as hyperparameter optimization for deep learning models, combinatorial optimization for logistics, and drug discovery, to validate its practical impact are also intended investigations.

References

[1] R. B. Statnikov and I. B. Matusov, "Optimization in engineering problems—Analysis and generalizations," *Journal of Machinery Manufacture and Reliability*, vol. 42, no. 4, pp. 289–292, 2013, doi: 10.3103/S1052618813040148.

[2] E. V. Seko, "Optimization and the dimensional analysis in structural engineering problems," *Advanced Materials Research*, vol. 945, pp. 1236–1241, 2014, doi: 10.4028/www.scientific.net/AMR.945.1236.

[3] L. K. Tartibu, *Multi-objective Optimization Techniques in Engineering Applications: Advanced Methods for Solving Complex Engineering Problems*, vol. 1184. Springer Nature, 2025.

[4] S. Asadi, S. Gharibzadeh, H. K. Naeini, M. Reihanifar, M. Rahimi, S. Zangeneh, A. Smerat, and L. Abdullah, "Comparative analysis of gradient-based optimization techniques using multidimensional surface 3D visualizations and initial point sensitivity," arXiv preprint arXiv:2409.04470, 2024.

[5] A. Arrasmith, M. Cerezo, P. Czarnik, L. Cincio, and P. J. Coles, "Effect of barren plateaus on gradient-free optimization," *Quantum*, vol. 5, p. 558, 2021.

[6] S. Balan, "Metaheuristics in optimization: Algorithmic perspective," *OR-MS-Tomorrow (2020)*, 2021.

[7] O. E. Turgut, M. S. Turgut, and E. Kırtepe, "A systematic review of the emerging metaheuristic algorithms on solving complex optimization problems," *Neural Computing and Applications*, vol. 35, no. 19, pp. 14275–14378, 2023.

[8] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 8091–8126, 2021.

[9] C. Blum, "Ant colony optimization: A bibliometric review," *Physics of Life Reviews*, vol. 51, pp. 87–95, 2024.

[10] J. Fang, W. Liu, L. Chen, S. Lauria, A. Miron, and X. Liu, "A survey of algorithms, applications and trends for particle swarm optimization," *International Journal of Network Dynamics and Intelligence*, 2023.

[11] E. H. Houssein, A. G. Gad, K. Hussain, and P. N. Suganthan, "Major advances in particle swarm optimization: theory, analysis, and application," *Swarm and Evolutionary Computation*, vol. 63, p. 100868, 2021.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[13] D. Thierens, "Scalability problems of simple genetic algorithms," in *Proceedings of the 7th International Conference on Genetic Algorithms*, Morgan Kaufmann, 1997, pp. 67–73.

[14] W. Rivera, "Scalable parallel genetic algorithms," *Artificial Intelligence Review*, vol. 16, no. 3, pp. 153–169, 2001, doi: 10.1023/A:101199661.

[15] K. E. S. Kumar, R. Dalvi, A. Mittal, A. Akhtar, F. D. Bosco, R. Lineswala, A. Chopra, et al., "Benchmarking of GPU-optimized quantum-inspired evolutionary optimization algorithm using functional analysis," arXiv preprint arXiv:2412.08992, 2024.

[16] K.-H. Han and J.-H. Kim, "Quantum-inspired evolutionary algorithm for a class of combinatorial optimization," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, 2002.

[17] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. Chichester, UK: John Wiley & Sons Ltd., 1990.

[18] Y. Zhou, Y. Shi, Y. Wei, Q. Luo, and Z. Tang, "Nature-inspired algorithms for 0–1 knapsack problem: A survey," *Neurocomputing*, vol. 554, p. 126630, 2023, doi: 10.1016/j.neucom.2023.126630.

[19] J. Jooken, P. Leyman, and P. De Causmaecker, "A new class of hard problem instances for the 0–1 knapsack problem," *European Journal of Operational Research*, vol. 301, no. 3, pp. 841–854, 2022.

[20] M. Abdel-Basset, R. Mohamed, and O. Kamel, "Evaluating binary metaheuristic algorithms for solving the 0/1 Knapsack Problem," *Applied Soft Computing*, vol. 151, p. 110050, 2024, doi: 10.1016/j.asoc.2023.110050.

[21] F. Althawadi, S. Alotaibi, and Z. Y. Algamal, "A rough set theory enhanced genetic algorithm for the multidimensional 0–1 knapsack problem," *IEEE Access*, vol. 12, pp. 5432–5447, 2024.

[22] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va.,

[23] G. M. Amdahl, "Computer architecture and Amdahl's law," *Computer*, vol. 46, no. 12, pp. 38–46, 2013.

[24] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.