GPU-Accelerated Stochastic Gradient Descent for Scalable Operator Placement in Geo-Distributed Streaming Systems

Tristan Joel Terhaag
TU Berlin
t.terhaag@campus.tu-berlin.de

Xenofon Chatziliadis
BIFOLD, TU Berlin
x.chatziliadis@tu-berlin.de

Eleni Tzirita Zacharatou
Hasso Plattner Institute
University of Potsdam
eleni.tziritazacharatou@hpi.de

Volker Markl
BIFOLD, TU Berlin, DFKI
volker.markl@tu-berlin.de

ABSTRACT

Modern geo-distributed stream processing systems, particularly those supporting Internet of Things (IoT) workloads, rely on efficient operator placement strategies to minimize end-to-end latency and avoid overloading resource-constrained edge nodes. Existing approaches, such as NEMO, address this challenge by modeling latency with Euclidean embeddings of network topologies and solving operator placement using spring relaxation. However, their CPU-bound optimization process limits scalability, particularly in large topologies with millions of nodes.

This paper introduces NEMO-SGD, the first GPU-accelerated, gradient-based optimizer for operator placement in distributed stream processing. NEMO-SGD reformulates the operator placement problem as a differentiable loss function and replaces NEMO's spring relaxation algorithm with a parallelized Stochastic Gradient Descent (SGD) process. Experiments performed on both synthetic and real-world topologies show that NEMO-SGD can optimize placements in under one second for topologies with up to 1 million nodes. This represents a reduction in the optimization time of up to 70% compared to the state-of-the-art NEMO approach. At the same time, NEMO-SGD maintains or even improves the placement quality. Our work shows that gradient-based, GPU-accelerated parallel optimization serves as a practical and scalable foundation for operator placement in next-generation stream processing systems.

VLDB Workshop Reference Format:
Tristan Joel Terhaag, Xenofon Chatziliadis, Eleni Tzirita Zacharatou, and Volker Markl. GPU-Accelerated Stochastic Gradient Descent for Scalable Operator Placement in Geo-Distributed Streaming Systems. VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS25).

1 INTRODUCTION

The widespread implementation of large-scale Internet of Things (IoT) applications places increasing demands on distributed stream processing systems [23, 44, 45]. These systems need to ingest and process continuous data streams from millions of devices spread across vast geographical locations. A core challenge in this context is operator placement, which involves determining where to execute computational tasks within the network to minimize end-to-end latency and prevent overloading nodes with limited resources.

To prevent processing nodes from being overloaded by the high data volumes typical of IoT applications, a common strategy is to parallelize decomposable aggregation functions (DAFs), such as min, max, count, and sum. These functions are widely used in stream processing [39, 43] and can be efficiently replicated and partially aggregated near the data source due to their inherent decomposability [25, 26, 34].

Motivating Example. Imagine a smart city air quality monitoring system deployed across thousands of intersections, roads, and public spaces. Each sensor node continuously transmits detailed pollution data to a stream processing system, which is responsible for detecting any breaches of predefined thresholds and sending alerts in real time. The calculations are performed using decomposable aggregation functions (DAFs), such as calculating the average concentration of nitrogen dioxide (NO_2).

In a typical deployment, raw data from each sensor is sent to a central cloud server for aggregation. However, this approach can result in high communication latency, strained network bandwidth, and delayed response times, which are unacceptable in real-time, latency-sensitive applications. To improve efficiency, operators should be placed close to the data sources, such as on edge devices or fog nodes. However, this presents a complex optimization challenge, as nearby devices vary in computational capacity, network latency, and workload. Overloading a single node can cause performance issues, while poor placement may increase latency and decrease responsiveness.

Challenges. Operator placement (OP) in geo-distributed stream processing systems is an NP-hard optimization problem [33]. Placing DAFs closer to data sources can significantly reduce latency and communication overhead, but also introduces new challenges. IoT environments involve devices that vary significantly in their computational capacity, exhibit non-uniform network latency, and experience frequent topological changes due to factors such as mobility, hardware failures, or fluctuating data rates [31, 35]. Traditional placement strategies for cloud-based systems are not well-suited to these conditions, as they struggle to scale to millions of nodes or

Proceedings of the VLDB Endowment. ISSN 2150-8097.

quickly adapt to rapidly changing environments. Therefore, effective OP solutions must: (i) scale to large, heterogeneous topologies; (ii) prevent the overloading of resource-constrained nodes; and (iii) adapt quickly to topological changes.

State-of-the-Art. NEMO [9] is currently the state-of-the-art approach for latency-aware operator placement and replication of DAFs in resource-constrained, geo-distributed environments. It leverages Euclidean embeddings of network topologies and applies a spring relaxation algorithm to create hierarchical aggregation trees. Although NEMO achieves high-quality placements, its optimization phase is CPU-bound, which can result in performance bottlenecks as the topology size increases.

Our Solution. We present NEMO-SGD, a scalable, GPU-accelerated approach for operator placement in geo-distributed stream processing systems. Building on the NEMO approach, NEMO-SGD replaces the CPU-intensive spring relaxation phase with a GPU-parallelized Stochastic Gradient Descent (SGD) optimizer. The placement objective is reformulated as a differentiable loss function that jointly minimizes latency and penalizes capacity violations. Furthermore, NEMO-SGD executes the entire placement pipeline on the GPU, leveraging parallelism to significantly improve scalability and runtime efficiency, particularly in large-scale topologies.

Results. We compare NEMO-SGD against NEMO as well as heuristics used in SPEs and adaptive aggregation approaches used in Wireless Sensor Networks (WSN). We conduct experiments on both synthetic and real-world datasets, including topologies with up to one million nodes. Our evaluation demonstrates that NEMO-SGD significantly outperforms the baseline methods, including NEMO. It achieves placement runtimes of under one second, making it 70% faster than NEMO. Additionally, NEMO-SGD improves placement quality by reducing the 90th percentile latency by up to 75× compared to NEMO. Importantly, it also avoids all capacity violations across test scenarios, thanks to the integration of soft constraint penalties during the training process.

The remainder of this paper is structured as follows. Section 2 describes the system and resource model of our approach, along with a formal definition of the problem. Our approach is described in Section 3, and its evaluation is presented in Section 4. We then explore related work in Section 5, followed by our concluding remarks in Section 6.

2 PRELIMINARIES

This section introduces the fundamental concepts and definitions of our approach, which we inherit from NEMO [9]. We first define the semantics of a stream processing application in Section 2.1. Next, we describe how we model latency and resource metrics for placement decisions in Section 2.2. Finally, in Section 2.3, we formulate the operator placement and replication problem that we address and parallelize using the GPU.

2.1 Stream Processing Model

SPEs take a user query as input and create a *logical operator plan* that represents the processing pipeline and specifies the order and type of operators and their dependencies. Operators are self-contained units performing specific functions, while streams are unbounded

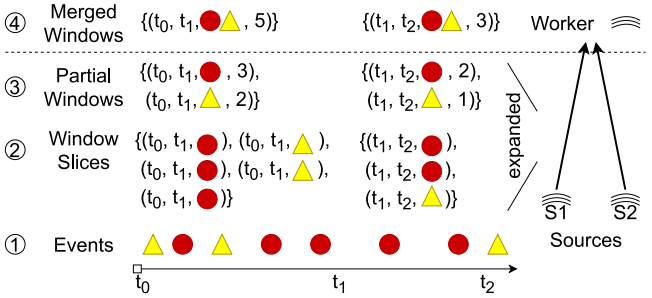


Figure 1: Example of distributed windowing, where partial count-aggregates are computed on remote worker nodes.

sequences of data tuples. This work focuses on Decomposable Aggregation Functions such as sum, average, or count. These operators support partial aggregation, enabling computations to be incrementally performed at intermediate nodes before reaching the final aggregation. Properly placing DAFs near their data sources can significantly reduce bandwidth usage and end-to-end latency.

Figure 1 depicts the operators required for hierarchical aggregation using window merging in a topology with two source nodes and one worker node that also acts as the sink. In ①, source operators emit individual events from their input streams. In ②, these events are grouped into window slices, which are time-partitioned segments of the stream. In ③, nodes compute local intermediate aggregates over their window slices through the partial aggregation operator. Finally, in ④, the worker node receives these partial aggregates and merges them to produce the final results. This hierarchical processing model, known as distributed windowing, enables parallel, slice-level computation and improves scalability [4]. However, it applies only to decomposable window functions; holistic functions do not support partial aggregation and thus limit the applicability of this approach [43].

Logical Plan. We model a query as a directed acyclic graph (DAG) $G = \Omega, S, A, L$, following Rizou et al. [33], where Ω is the set of operators and L the set of links $\overline{\omega_i \omega_j}$ indicating that operator ω_i produces a stream consumed by ω_j. Each operator ω_i has a set of incoming and outgoing links L_{ω_i}. Link weights $w(\overline{\omega_i \omega_j})$ reflect the load imposed on the destination node, and the load $C_u(v_i)$ of a node is computed as the sum of the weights of its incoming links.

Operators in $S \subset \Omega$ (sources) have only outgoing links and produce data, while those in $A \subset \Omega$ (sinks) have only incoming links and consume data. Sources and sinks are pinned, i.e., they have fixed placements, while other operators can be freely assigned to any node in the topology.

Replication. To support parallelism and scalability, we extend the logical plan with operator replication. Each operator $\omega \in \Omega$ is represented as a tuple $\{\omega_{\text{id}}, R_{\text{id}}, v_i, \rho\}$, where ω_{id} is the operator ID, R_{id} the replica ID, v_i the physical node to which the replica is assigned, and ρ the number of replicas. Pinned operators (i.e., sources and sinks) have fixed placements and are not replicated. The resulting *replication plan* $G^* = \{\Omega^*, L^*\}$ augments the original DAG with all operator replicas and their links. The total number of operators $|\Omega^*|$ equals the sum of all replicas.

To avoid duplicate processing and ensure valid partitioning, we adopt a constraint inspired by WSNs [41]: replicas must receive disjoint input streams, i.e., $\forall \omega\,(\omega \in L_j \rightarrow \omega \notin L_k)$ for any two replicas of the same operator. This significantly reduces the number of valid edges in L^*. We denote the superset of all valid replication plans (i.e., with all possible ρ and valid paths) as G'.

2.2 Resource Model

As required by NEMO, we model the topology as a set of connected nodes V in a cost space $\hat{G}_T \in \mathbb{R}^{n \times d}$. Each row of $\hat{G}_T$ represents the coordinates of a node in the cost space, with the ith row corresponding to the coordinates of the ith node $v_i \in V$. We denote the coordinates of node v_i as $\hat{v}_i$ and represent the set of all node coordinates as $\hat{V} = \{\hat{v}_1, \ldots, \hat{v}_n\}$. The cost space $\hat{G_T}$ represents optimization metrics, in our case, latency, in a Euclidean space. Specifically, in our cost space, each node is assigned coordinates in two dimensions, ensuring that the Euclidean distance between any two nodes closely approximates the actual latency between them. Various methods exist for computing such a latency cost space. In this work, we use the Vivaldi algorithm [14] to create the latency cost space. Cost spaces are generally agnostic regarding the types of metrics they represent and can also be used to depict other metrics, as further discussed by Chatziliadis et al. [9].

We model the maximum computational capacity of a node v_i as $C_t(v_i) \in \mathbb{N}$. Nodes with high capacities are typically servers within the cloud, while those with lower capacities are edge or sensor devices. We consider v_i to be overloaded if $C_u(v_i) > C_t(v_i)$, where C_u represents the utilized capacity. Finally, C_r represents the required capacity, and $C_a = C_t - C_u$ the available capacity. The available and utilized capacity of a node depends on the input data rate of the operators placed on it.

2.3 Problem Definition

In a distributed stream processing system, the goal of operator placement is to assign each operator $\omega \in \Omega$ to one or more nodes $v \in V$ such that system constraints are met and a given objective, such as minimizing end-to-end latency, is optimized. The operator placement and replication (OPR) problem extends the classical placement problem by also determining the optimal number of replicas per operator, enabling parallel processing over partitioned data streams. Each replica is assigned to a node such that no two replicas of the same operator receive the same input, preventing redundant computation [41]. The binary decision variable $f_m(\omega, v) \rightarrow 1$ indicates placement of operator ω on node v. As shown by Cardellini et al. [8], OPR generalizes the NP-hard assignment problem [22] and remains computationally NP-hard.

Formally, our objective is to minimize total latency $\mathrm{Lat}(\Omega^*)$ across all source-to-sink paths, subject to node capacity constraints and valid replication semantics, which is formally defined as:

$$\min \mathrm{Lat}(\Omega^*) = \sum_{\forall \omega_x, \omega_y \in \Omega^*} d(\overrightarrow{\omega_x \omega_y}), \forall \Omega^* \in G^*, \forall G^* \in G' \tag{1}$$

subject to the constraint

$$C_u(v_i) \leq C_t(v_i), \forall v_i \in V. \tag{2}$$

3 NEMO SGD

Our approach builds on NEMO [9], a previously proposed optimization approach for latency-aware operator placement in geo-distributed stream processing systems. NEMO performs the placement process in three phases: 1) latency-based clustering of nodes, 2) virtual placement of operators in a continuous coordinate space using spring relaxation, and 3) determination of the number of operator replicas and their mapping to physical nodes with capacity checks. Figure 2 depicts these three phases for an artificial cost space comprising 1000 nodes, where all nodes are sources in the topology that transmit a data aggregate to the sink. For additional details on the three phases of NEMO, we refer the reader to the original paper [9].

Although NEMO is scalable and can be re-optimized, it has two main limitations. First, its placement loop operates solely on the CPU. Second, it manages capacity constraints only during a post-processing step in phase 3. This can lead to an increased degree of replication for intermediate aggregation operators, resulting in higher latency. We address both issues with NEMO-SGD, a GPU-parallel extension that retains NEMO's overall structure but replaces its core placement algorithm with a differentiable, capacity-aware objective solved via stochastic gradient descent (SGD). In addition to SGD calculation, NEMO-SGD performs all computations on the GPU, enabling fast optimization even on large-scale topologies.

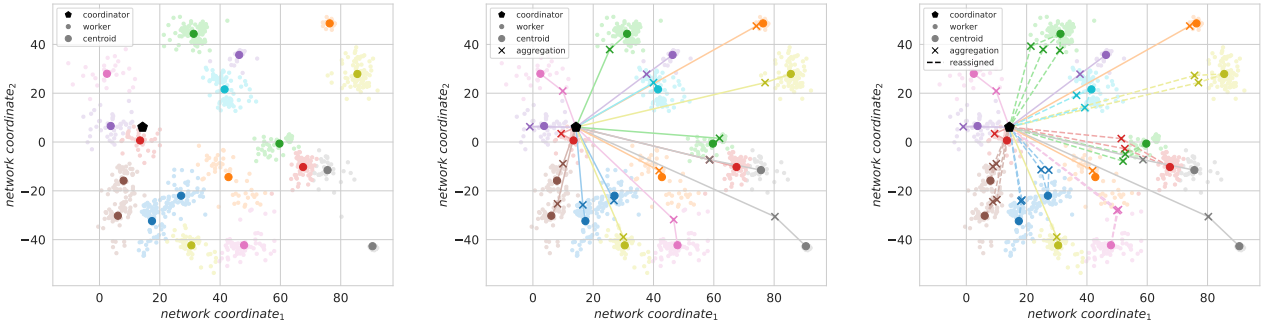
3.1 Unified Latency–Capacity Objective

The initial phase of NEMO serves as a preprocessing step in which nodes within the cost space are grouped into clusters based on minimal latency, as shown in Figure 2(a). Each cluster contains nodes that are situated close to one another in the cost space, resulting in low latency between them. This clustering effectively reduces the search space for the subsequent phases.

In the second phase, NEMO calculates the optimal placement of an operator in the cost space between a given set of upstream nodes and the sink. In this phase, we replace NEMO's spring relaxation method with a differentiable objective that simultaneously minimizes communication latency and avoids overloading physical nodes. Each aggregation operator is modeled as a freely assignable virtual node $\hat{v} \in \mathbb{R}^d$. Initially, the number of aggregation operators to be assigned corresponds to the number of clusters, i.e., we place an aggregation operator for each cluster. Additional intermediate aggregation layers are introduced iteratively until all optimization constraints are satisfied. Given a set of upstream nodes U (initially the sources of the cluster) and a downstream node s (i.e., the sink), the objective function for determining the position of the virtual node $\hat{v}$ is defined as:

$$\min \sum_{u \in U} \alpha \|\hat{v} - u\|_2^2 + \beta \|\hat{v} - s\|_2^2 + \gamma \frac{\max(0, \lambda(\hat{v}) - C(\hat{v}))}{C(\hat{v})}, \tag{3}$$

The first two terms model latency and penalize distance between upstream and downstream nodes, ensuring that the virtual node is placed close to where the data originates and where it is eventually consumed. The third term introduces a soft penalty for exceeding capacity: $\lambda(\hat{v})$ estimates the total load routed through $\hat{v}$, and $C(\hat{v})$



(a) Grouping (b) Virtual Placement (c) Re-assignment

Figure 2: Overview of NEMO SGD's three phases: (1) grouping physical nodes by latency, (2) optimally placing operators on virtual nodes to form an aggregation tree, and (3) mapping operators to physical nodes with replication to prevent overload.

is the combined available capacity of the k-nearest physical nodes. When there is sufficient capacity, the penalty is zero; otherwise, it grows proportionally with the overload. The weighting parameters α, β, and γ balance the trade-off between minimizing latency and avoiding congestion.

We minimize Equation 3 using mini-batch stochastic gradient descent. At each iteration t, a batch $B_t \subseteq U$ of upstream nodes is sampled, and the virtual position is updated as follows:

$$\hat{v}_{t+1} = \hat{v}_t - \eta \left[2\alpha \sum_{u \in B_t} (\hat{v}_t - u) + 2\beta(\hat{v}_t - s) + \gamma \nabla_{\hat{v}} (\text{penalty}) \right],$$

where η is the learning rate, and the penalty gradient becomes active only when the estimated load exceeds capacity. This allows to efficiently determine coordinates in the cost space that are both latency-efficient and resource-feasible for each virtual node. The convexity of the optimization objective in this phase enables the placement of topologies with millions of operators to be optimized independently and in parallel.

The final phase of NEMO involves mapping virtual nodes to physical nodes for deployment. This mapping is crucial, as virtual nodes are abstract operator representations that exist solely within the optimization cost space. The objective is to assign each virtual node to the nearest physical node within its cluster that has enough residual capacity to accommodate the expected load. NEMO-SGD retains this phase unchanged to preserve constraint satisfaction. As in NEMO, it computes the Euclidean distances between each virtual node and all physical nodes in the same cluster. It then sorts the nodes by proximity and assigns the virtual node to the closest physical node with sufficient available capacity.

NEMO-SGD iteratively repeats the above process until all optimization constraints are fulfilled. In each iteration, an intermediate aggregation layer is introduced to perform a new aggregation between the nodes identified in the previous iteration and the sink.

3.2 GPU Implementation

NEMO-SGD executes the complete operator placement pipeline on the GPU. To support efficient parallel computation, node coordinates are stored in a columnar tensor layout, which allows GPU threads to access data in a coalesced manner and improves memory throughput. Cluster memberships are stored in a shared index for parallel thread access during mini-batch updates. The available capacity of each node is tracked using a compact array that resides on the GPU. To speed up proximity queries, which are crucial for making placement decisions, we utilize a grid-based spatial indexing scheme. This indexing is precomputed in the GPU's shared memory at the beginning of each optimization pass.

The entire optimization process, including gradient calculation, capacity evaluation, penalty handling, and coordinate updates, is fused into a single GPU kernel. Instead of launching many small operations separately, we record the full sequence of computations once using GPU graph capture and replay this sequence for each epoch. This drastically reduces kernel launch overhead and improves overall runtime. Initial node clustering is performed on the GPU using an efficient mini-batch K-Means clustering algorithm that runs asynchronously in a separate execution stream. To avoid idle time, we employ a double-buffering strategy: while one epoch is being optimized, the next epoch's clustering runs in parallel, effectively hiding the clustering cost behind ongoing computation.

To synchronize results across threads and thread blocks, we aggregate key metrics such as total loss, gradient norm, and capacity constraints using a single collective operation per epoch, reducing the overhead typically caused by frequent fine-grained synchronizations. Load balancing is handled dynamically: clusters are assigned to thread blocks via a shared work queue, allowing idle threads to steal remaining tasks and redistribute the workload as needed. This ensures high GPU utilization even when cluster sizes or computational demands vary. All numerical operations are performed using single-precision (FP32) arithmetic.

4 EVALUATION

We evaluate our parallelized, gradient-based approach, NEMO-SGD, using the simulation testbed introduced by Chatziliadis et al. [9],

which was also used to evaluate the original NEMO approach. Our evaluation focuses on placement quality and runtime performance, and it is conducted on both real and synthetic datasets that exhibit diverse latency and workload characteristics.

4.1 Experimental Setup

In this section, we detail the experimental setup used to evaluate NEMO-SGD, which includes the hardware environment, datasets, workload and capacity models, and baselines. Our objective is to ensure a fair and reproducible evaluation across various scenarios.

Hardware. All experiments were conducted on a workstation equipped with an AMD Ryzen 7 5800X CPU, 32 GB of RAM, and an NVIDIA RTX 3070Ti GPU with 8 GB of VRAM. The operating system was Ubuntu 22.04 LTS, running under Windows Subsystem for Linux 2 (WSL2).

Implementation. NEMO-SGD is implemented in Python using PyTorch 2.0.1 with CUDA 11.8. For comparison, we also implemented a CPU-parallel version of NEMO-SGD. The number of CPU processes, as well as the number of GPU thread blocks, corresponds to the number of clusters identified in the first phase of NEMO. Parameters were individually tuned for each experiment to ensure optimal performance. All of our experiments also include the time required to load data into GPU memory.

Datasets. We use the Vivaldi algorithm [14] to construct network coordinate systems based on latency measurements collected from various real-world and synthetic topologies. To ensure a fair comparison, we adopt the same hyperparameters as those used in the original NEMO evaluation, including the number of neighbors *mm* (which represents the number of direct latency measurements per node). Network coordinates for all tested topologies are generated and evaluated using the NCSIM simulation tool [11]. We use latency measurements of the following topologies: 1) FIT IoT Lab [1] is an IoT testbed deployed across different regions in France. Our evaluation uses RTTs of 433 geographically distributed nodes comprising different types of microcontrollers and four gateway servers. 2) PlanetLab (PL) [13] measurements represent RTTs from 335 nodes hosted by universities and research institutions across Europe and North America. 3) King [17] contains latency measurements of 1740 Internet DNS servers. 4) We also generate artificial NCSs with varying latency distributions and sizes. The x-axis of the NCSs ranges in $[0, 100]$ and the y-axis in $[-50, 50]$. These ranges represent a combined set of latency distributions found in other topologies. Nodes belong to different Gaussian distributions with uniformly distributed centers across the plane. The artificial NCSs range from 1k to 1M nodes.

Workload and Capacity Models. Consistent with the original NEMO evaluation, our simulations are based on a generic DAF monitoring workload that collects metrics from all devices in the topology and computes window aggregates. This workload models a common scenario in many monitoring systems [3, 5, 10, 16, 27, 34, 36]. The load scales proportionally with the size of the topology, as all nodes serve as data sources, allowing for a comprehensive evaluation of NEMO's scalability and performance. The sink node is randomly selected to avoid bias in the evaluation.

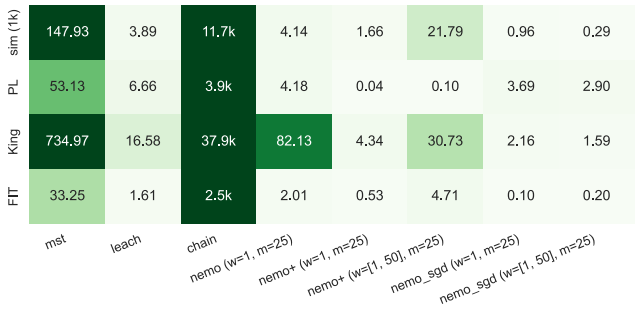


Figure 3: Comparison of the 90th percentile latency deltas against bottom up/top down across different approaches.

Capacities and Weights. As in the original NEMO evaluation, we model node capacities in line with real-world stream processing systems such as Flink, Spark, Storm, and NebulaStream, which typically specify worker capacities through configuration files. To demonstrate NEMO's robustness under diverse deployment conditions, we evaluate it on a range of capacity distributions that reflect different levels of heterogeneity. To ensure consistent comparisons across experiments, we vary individual node capacities while keeping the total capacity approximately constant (minor deviations may arise due to rounding). Capacities are drawn from a log-normal distribution with parameters $\sigma = 1.4$ and $\mu = 7.3$. To evaluate NEMO+ and NEMO-SGD under varying workload intensities, we assign different link weights (w) to sources. Initially, all sources are assigned uniform weights, providing a baseline for comparison with NEMO and alternative aggregation strategies. We then increase load heterogeneity by assigning weights to all sources, drawn from a log-normal distribution within the range of 1 to 50.

Baselines. We compare NEMO-SGD against the following baselines: 1) Optimal: An optimal solution based on Cardellini et al. [8]. 2) LEACH [20]: A cluster-based method commonly used in WSNs, where data is pre-aggregated at randomly selected cluster heads. In our implementation, we use a centralized LEACH version with k-d trees for neighborhood search and set the number of cluster heads to 10% of the nodes, following [20]. 3) MST: A greedy approach based on Prim's algorithm [30], often used for tree-based aggregation in WSNs. It constructs a minimum spanning tree from sources to sink, enabling in-network aggregation at intermediate nodes. 4) Chain: A chain-based method used in WSNs that aggregates data along a sequential chain from sources to sink. Our implementation uses a centralized, probabilistic approach combining stochastic gradient descent and simulated annealing. 5) NEMO: We include both vanilla NEMO and its extension, NEMO+, which supports arbitrary weights. For each topology and weight distribution, we individually tune the hyperparameters.

4.2 Placement Quality

In this section, we evaluate the placement quality achieved by NEMO-SGD, which integrates stochastic gradient descent, GPU-parallel optimization, and a latency-focused cost function to minimize the 90th-percentile end-to-end latency. To compare the performance of NEMO-SGD with the original NEMO approach, we

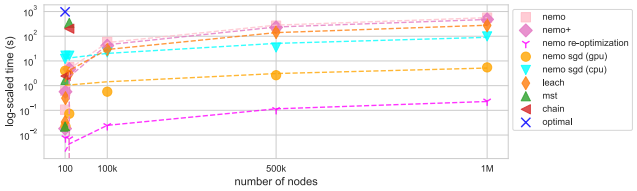


Figure 4: Full-optimization and re-optimization times for the monitoring workload (cf. Section 4.1) that increases in complexity with the size of the topology.

compute the delta in the 90th percentile latency across datasets with respect to the theoretical minimum defined by direct transmission. Figure 3 illustrates the results. In this figure, lighter cells represent lower average latency, while darker cells indicate higher latency. We observed a consistent trend across various levels of heterogeneity, so we will only present the results for the highest level of heterogeneity ($m = 25$) for both uniform and skewed load distributions ($w = 1$ and $w \in [1, 50]$). The optimal approach could not produce results for the tested topologies and is therefore excluded from our analysis.

In the light-load case ($w = 1$), NEMO-SGD consistently narrows the latency gap to the theoretical lower bound across most topologies. On the FIT network, it reduces the 90th-percentile latency delta from 0.53 ms (with NEMO+) to just 0.10 ms, representing a 5.3× improvement. In the King topology, the latency delta is halved from 4.34 ms to 2.16 ms. In the artificial topology with 1000 nodes (Sim-1k), it drops from 1.66 ms to 0.96 ms. PlanetLab is an exception. NEMO+ already achieves near-optimal placement with only a 0.04 ms latency delta, whereas NEMO-SGD has a delta of 3.69 ms, which is still significantly lower than the traditional WSN approaches.

Under heavy-tailed load distributions ($w \in [1, 50]$), the improvements are even more pronounced. In Sim-1k, the latency delta plunges from 21.8 ms to 0.29 ms, a 75× reduction. Similarly, on the King and FIT topologies, the deltas drop from 30.7 ms to 1.59 ms and from 4.71 ms to 0.20 ms, corresponding to 19× and 23× improvements, respectively.

These gains stem from two key properties of NEMO-SGD. First, by extending the cost function to include capacity constraints, it identifies more practical placement locations in terms of the optimal placement in the cost space compared to the original NEMO. Second, the use of GPU parallelism allows the optimizer to evaluate hundreds of candidate steps within each iteration, substantially expanding the search space without increasing wall-clock time. Together, these factors produce placements that avoid redundant intermediate aggregation and strictly adhere to cluster-head capacity constraints, thereby achieving sharp reductions in latency.

In summary, NEMO-SGD retains NEMO's zero-overload guarantee while cutting the 90th-percentile latency by up to two orders of magnitude compared to classical aggregation heuristics, and by up to 75× relative to the best NEMO+ configuration under heavy load. These results highlight the effectiveness of embedding stochastic, GPU-accelerated optimization within the geo-distributed operator placement pipeline.

4.3 Scalability

In this section, we evaluate the scalability of the NEMO-SGD optimization across topologies of different sizes. We also include an evaluation of NEMO's re-optimization performance. The x-axis of Figure 4 shows an increasing number of nodes in the topology, which also linearly increases the total number of required aggregations. The y-axis represents the time taken to compute a full placement of the DAF query in logarithmic scale. We observed a similar trend over all tested capacity and workload distributions and therefore present only the results for $w = 1$ and $m = 50$.

To evaluate the re-optimization of NEMO, we considered four cases: (1) removal of random leaf nodes, (2) removal of random cluster heads, (3) addition of nodes, and (4) computation of coordinates for a node. All re-optimization strategies completed in under one second and are summarized using their average. It is important to note that the re-optimization approach for all NEMO variants is the same, thus yielding the same results.

Our evaluation in Figure 4 shows that NEMO-SGD significantly improves optimization runtime across all tested topologies. Notably, it achieves sub-second optimization times on GPU, reducing the optimization duration by up to 70% compared to the standard NEMO+, making it the fastest approach among all evaluated variants while preserving the same placement quality as NEMO+.

To assess its hardware portability, we also evaluated a parallelized CPU implementation of NEMO-SGD. Although the CPU implementation is slower than the GPU version, it still offers a substantial improvement over prior approaches, with optimization times ranging from 12 to 98 seconds across different topology sizes. This demonstrates that even without access to GPU acceleration, NEMO-SGD remains a practical and scalable choice for large-scale deployments. In contrast, the optimal solution requires over 15 minutes for topologies with fewer than 100 nodes and was manually terminated at 20 minutes in all other cases. Similar timeouts were observed for MST and Chain. Among these, MST could only handle up to 10k nodes within 6 minutes, while Chain reached its limits at 1k nodes with a runtime of 5 minutes.

Although the original NEMO and NEMO+ achieve linear-time complexity, they can take up to approximately 10 minutes for 1 million nodes. LEACH is the only baseline besides NEMO to exhibit linear runtime, thanks to its k-d tree–based nearest neighbor search, requiring 5 minutes for 1 million nodes. We note that although the original NEMO and NEMO+ have a slower optimization phase compared to LEACH due to their more complex computations, such as optimal operator placement in the NCS and overload prevention, their strength lies in dynamic adaptability. NEMO's re-optimization consistently takes under one second. Thanks to advancements in NEMO-SGD, the optimization time for full placements in topologies with millions of nodes is now nearly as fast as the constant re-optimization times of NEMO, allowing both optimization and re-optimization to be completed in under one second on GPU and in just over a minute on CPU.

4.4 Summary

Our evaluation shows that NEMO-SGD significantly outperforms all baseline methods, including the original NEMO and its optimized variant, NEMO+. This improvement is evident in both placement

quality and runtime efficiency. By utilizing GPU-parallel stochastic gradient descent and a capacity-aware cost function, NEMO-SGD reduces the 90th percentile latency by up to 75× compared to NEMO+ and achieves reductions of up to two orders of magnitude over traditional wireless sensor network (WSN) approaches under high-load conditions. Additionally, it provides full placement runtimes that are under one second, surpassing even the lightweight execution of LEACH, while still maintaining NEMO's strict no-overload guarantees. These results establish NEMO-SGD as the most scalable and latency-efficient solution for geo-distributed operator placement in heterogeneous environments.

5 RELATED WORK

We categorize existing work into two main areas: (1) operator placement strategies for distributed stream processing, and (2) GPU-accelerated approaches that enhance performance by leveraging parallel hardware in streaming environments.

Operator Placement in Stream Processing Systems. Operator placement has been extensively studied in cloud environments, typically focusing on minimizing end-to-end latency or inter-node traffic within homogeneous clusters [15, 38]. Subsequent work extended these models to account for device heterogeneity [7], joint placement–replication decisions [8], and more recently, learned cost models for placement in hybrid edge-cloud settings [18]. However, these formulations suffer from prohibitively long solve times on realistic, large-scale topologies and are too slow to adapt to dynamic changes in network conditions. SBON [32] and its extension [33] improve the efficiency of single- and multi-operator placement by reformulating the search as a cost-space optimization problem solved via gradient descent. While faster, these methods ignore resource capacity constraints and do not support replication or re-optimization. Similarly, the latency-spike–aware heuristic in [19] offers incremental placement capabilities but applies only to newly added operators.

Beyond the stream processing domain, operator placement analogues in wireless sensor networks aim to reduce energy consumption through in-network aggregation schemes such as LEACH [20], HEED [42], PEDAP [37], and PEGASIS [24]. These protocols form multi-hop topologies where data is incrementally aggregated en route to the sink. While highly adaptive, they focus on network formation rather than resource-aware operator placement and thus neglect key concerns such as latency and resource constraints. The first approach to explicitly address the distinct challenges of geo-distributed stream processing, while addressing replication, adaptivity, scalability, responsiveness to topology changes, and resource-awareness for DAFs is NEMO [9].

Critically, all prior approaches treat OP as a CPU-bound optimization problem. While techniques like SBON and NEMO offer linear scalability, they still incur substantial solve times on large topologies. None of these methods leverage modern GPU parallelism to accelerate the placement search itself. In contrast, our work introduces a GPU-accelerated solver that achieves near-interactive solve times while maintaining full support for resource constraints, replication, and re-optimization. This addresses a crucial gap in existing solutions.

GPU-accelerated execution. A separate line of work focuses on accelerating the execution of streaming operators by offloading them to GPUs, while leaving placement decisions to conventional, CPU-based mechanisms. G-Storm [12] extends Apache Storm [2] with CUDA-enabled bolts that execute entire operators on discrete GPUs, achieving significant throughput improvements but relying on Storm's original, CPU-bound scheduler for placement. Flink [6] with TornadoVM [40] compiles user-defined functions into GPU kernels, yet uses Flink's standard task manager for operator assignment. FineStream [46] introduces fine-grained co-scheduling of windowed operators across CPU–GPU SoCs to improve throughput and energy efficiency, but the mapping of operators to devices is still determined by a CPU-side heuristic. WindFlow [29] and Springald [28] offload selected operators to GPUs, but the logic that decides when and whether to offload remains on the CPU. dSTREAM [21] enables runtime adaptation between CPU and GPU execution, yet its mapping algorithm also runs entirely on the CPU.

In summary, existing GPU-aware frameworks accelerate the processing of operators but not the decision-making about where and how they are deployed. Placement logic remains serial and CPU-bound, resulting in decision latencies ranging from hundreds of milliseconds to several minutes on large or frequently changing topologies. In contrast, our work is the first to move the placement optimization itself onto the GPU, enabling fast, scalable, and resource-aware scheduling with support for replication and efficient re-optimization.

6 CONCLUSION

Modern stream processing systems require low-latency, resource-efficient computation over large, geo-distributed infrastructures. However, existing operator placement strategies are limited by CPU-bound optimization techniques that do not scale well with increasing topology size. In this paper, we introduced NEMO-SGD, the first GPU-accelerated, gradient-based optimizer for operator placement in stream processing. By replacing spring relaxation with a parallelized Stochastic Gradient Descent algorithm, NEMO-SGD reduces optimization runtime by up to 70% compared to the previous state-of-the-art, while retaining support for resource constraints, operator replication, and efficient re-optimization.

By offloading placement computation to the GPU, NEMO-SGD establishes a new direction for scalable, low-latency optimization in distributed stream processing systems, making it highly suitable for real-time deployment in large-scale, dynamic environments.

REFERENCES

[1] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noël, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2nd IEEE World Forum on Internet of Things*. 459–464.

[2] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive online scheduling in storm. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS*. 207–218.

[3] Tanapat Anusas-Amornkul and Sirasit Sangrat. 2017. Linux Server Monitoring and Self-healing System Using Nagios. In *Mobile Web and Intelligent Information Systems - 14th International Conference, MobiWIS*, Vol. 10486. 290–302.

[4] Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. 2020. Disco: Efficient Distributed Window Aggregation. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*. 423–426.

[5] David Burrell, Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Steffen Zeuch, and Volker Markl. 2023. Workload Prediction for IoT Data Management Systems.

In *Datenbanksysteme fur Business, Technologie und Web, BTW*.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[7] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS*. 69–80.

[8] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurr. Comput. Pract. Exp.* 30, 9 (2018).

[9] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1501–1514.

[10] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Steffen Zeuch, and Volker Markl. 2021. Monitoring of Stream Processing Engines Beyond the Cloud: An Overview. *Open J. Internet Things* 7, 1 (2021), 71–82.

[11] Yang Chen, Xiao Wang, Cong Shi, Eng Keong Lua, Xiaoming Fu, Beixing Deng, and Xing Li. 2011. Phoenix: A Weight-Based Network Coordinate System Using Matrix Factorization. *IEEE Trans. Netw. Serv. Manag.* 8, 4 (2011), 334–347.

[12] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. 2015. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 307–312.

[13] Brent N. Chun, David E. Culler, Timothy Roscoe, Andy C. Bavier, Larry L. Peterson, Mike Wawrzoniak, and Mic Bowman. 2003. PlanetLab: An overlay testbed for broad-coverage services. *Comput. Commun. Rev.* 33, 3 (2003), 3–12.

[14] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Tappan Morris. 2004. Vivaldi: A decentralized network coordinate system. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM*. 15–26.

[15] Raphael Eidenbenz and Thomas Locher. 2016. Task allocation for distributed stream processing. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM*. 1–9.

[16] Stefano Forti, Marco Gaglianese, and Antonio Brogi. 2021. Lightweight self-organising distributed monitoring of Fog infrastructures. *Future Gener. Comput. Syst.* 114 (2021), 605–618.

[17] P. Krishna Gummadi, Stefan Saroiu, and Steven D. Gribble. 2002. King: Estimating latency between arbitrary internet end hosts. *Comput. Commun. Rev.* 32, 3 (2002), 11.

[18] Roman Heinrich, Carsten Binnig, Harald Kornmayer, and Manisha Luthra. 2024. Costream: Learned Cost Models for Operator Placement in Edge-Cloud Environments. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 96–109.

[19] Thomas Heinze, Yuanzhen Ji, Lars Roediger, Valerio Pappalardo, Andreas Meister, Zbigniew Jerzak, and Christof Fetzer. 2015. FUGU: Elastic Data Stream Processing with Latency Constraints. *IEEE Data Eng. Bull.* 38, 4 (2015), 73–81.

[20] Wendi Rabiner Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. 2000. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *33rd Annual Hawaii International Conference on System Sciences, HICSS*. 10–20.

[21] Gyeonghwan Jung, Yeonwoo Jeong, Kyuli Park, Dongjae Lee, Hongsu Byun, Suyeon Lee, and Sungyong Park. 2024. dStream: An Online-based Dynamic Operator-level Query Mapping Scheme on Discrete CPU-GPU Architectures. *IEEE Access* (2024).

[22] Richard M. Karp. 2010. Reducibility Among Combinatorial Problems. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 219–241.

[23] Aljoscha P. Lepping, Hoang Mi Pham, Laura Mons, Balint Rueb, Philipp M. Grulich, Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *Proc. VLDB Endow.* 16, 12 (2023), 3930–3933.

[24] Stephanie Lindsey, Cauligi S. Raghavendra, and Krishna M. Sivalingam. 2002. Data Gathering Algorithms in Sensor Networks Using Energy Metrics. *IEEE Trans. Parallel Distributed Syst.* 13, 9 (2002), 924–935.

[25] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.* 30, 1 (2005), 122–173.

[26] Quazi Mamun. 2012. A Qualitative Comparison of Different Logical Topologies for Wireless Sensor Networks. *Sensors* 12, 11 (2012), 14887–14913.

[27] Matthew L. Massie, Brent N. Chun, and David E. Culler. 2004. The Ganglia distributed monitoring system: Design, Implementation, and Experience. *Parallel Comput.* 30, 5-6 (2004), 817–840.

[28] Gabriele Mencagli, Patrizio Dazzi, and Massimo Coppola. 2024. Springald: GPU-accelerated Window-based Aggregates over Out-of-Order Data Streams. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[29] Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, and Marco Danelutto. 2021. Windflow: High-speed continuous stream processing with parallel building blocks. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2748–2763.

[30] Xia Pan, Xia Zhang, Hongyi Yu, and Chao Zhang. 2009. Study on routing protocol for WSNs based on the improved Prim algorithm. In *2009 International Conference on Wireless Communications & Signal Processing*. 1–4.

[31] Elena Beatriz Ouro Paz, Eleni Tzirita Zacharatou, and Volker Markl. 2021. Towards Resilient Data Management for the Internet of Moving Things. In *Datenbanksysteme für Business, Technologie und Web, BTW*. 279–301.

[32] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE*. 49.

[33] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. 2010. Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks. In *Proceedings of the 19th International Conference on Computer Communications and Networks, ICCCN*. 1–6.

[34] Atul Sandur, ChanHo Park, Stavros Volos, Gul Agha, and Myeongjae Jeon. 2022. Jarvis: Large-scale Server Monitoring with Adaptive Near-data Processing. In *38th IEEE International Conference on Data Engineering, ICDE*. 1408–1422.

[35] Zhitao Shen, Vikram Kumaran, Michael J. Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50.

[36] Nitin Sukhija and Elizabeth Bautista. 2019. Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus. In *IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*. 257–262.

[37] Hüseyin Özgür Tan and Ibrahim Korpeoglu. 2003. Power efficient data gathering and aggregation in wireless sensor networks. *SIGMOD Rec.* 32, 4 (2003), 66–71.

[38] Cory Thoma, Alexandros Labrinidis, and Adam J. Lee. 2014. Automated operator placement in distributed Data Stream Management Systems subject to user constraints. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE*. 310–316.

[39] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 374–389.

[40] Maria Xekalaki, Juan Fumero, and Christos Kotselidis. 2018. Dynamic Acceleration of Big Data Applications on Heterogeneous Hardware Resources. In *proceedings of the 1st International Workshop on Next Generation Clouds for Extreme Data Analytics (Xtreme-Cloud), F.*

[41] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. 2008. Wireless sensor network survey. *Computer networks* 52, 12 (2008), 2292–2330.

[42] Ossama Younis and Sonia Fahmy. 2004. HEED: A Hybrid, Energy-Efficient, Distributed Clustering Approach for Ad Hoc Sensor Networks. *IEEE Trans. Mob. Comput.* 3, 4 (2004), 366–379.

[43] Wang Yue, Lawrence Benson, and Tilmann Rabl. 2023. Desis: Efficient Window Aggregation in Decentralized Networks. In *Proceedings 26th International Conference on Extending Database Technology, EDBT*. 618–631.

[44] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR*.

[45] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M. Grulich, Ariane Ziehn, and Volker Markl. 2020. NebulaStream: Complex Analytics Beyond the Cloud. *Open J. Internet Things* 6, 1 (2020), 66–81.

[46] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. {FineStream}:{Fine-Grained}{Window-Based} stream processing on {CPU-GPU} integrated architectures. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 633–647.