2025 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3315-7844-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPEC67600.2025.11196472

Anonymized Network Sensing using `C++26 std::execution` on GPUs

Michael Mandulak
Rensselaer Polytechnic Institute
Troy, NY, USA
mandum@rpi.edu

Sayan Ghosh
Pacific Northwest National Laboratory
Richland, WA, USA
sayan.ghosh@pnnl.gov

S M Ferdous
Pacific Northwest National Laboratory
Richland, WA, USA
sm.ferdous@pnnl.gov

Mahantesh Halappanavar
Pacific Northwest National Laboratory
Richland, WA, USA
hala@pnnl.gov

George Slota
Rensselaer Polytechnic Institute
Troy, NY, USA
slotag@rpi.edu

***Abstract*—Large-scale network sensing plays a vital role in network traffic analysis and characterization. As network packet data grows increasingly large, parallel methods have become mainstream for network analytics. While effective, GPU-based implementations still face start-up challenges in host-device memory management and porting complex workloads on devices, among others. To mitigate these challenges, composable frameworks have emerged using modern C++ programming language, for efficiently deploying analytics tasks on GPUs. Specifically, the recent C++26 `Senders` model of asynchronous data operation chaining provides a simple interface for *bulk pushing tasks* to varied device execution contexts.**

Considering the prominence of contemporary dense-GPU platforms and vendor-leveraged software libraries, such a programming model consider GPUs as first-class execution resources (compared to traditional host-centric programming models), allowing convenient development of multi-GPU application workloads via expressive and standardized asynchronous semantics. In this paper, we discuss practical aspects of developing the Anonymized Network Sensing Graph Challenge on dense-GPU systems using the recently proposed C++26 `Senders` model. Adopting a generic and productive programming model does not necessarily impact the critical-path performance (as compared to low-level proprietary vendor-based programming models): our commodity library-based implementation achieves up to 55× performance improvements on 8× NVIDIA A100 GPUs as compared to the reference serial GraphBLAS baseline.

***Index Terms*—Network traffic analysis, Multi-GPU, C++26, `std::execution`, Graph Challenge**

I. Introduction

Network sensing at a large-scale [1] plays a critical role in a variety of applications, such as in network traffic characterization, recommender systems, sensor dynamics and network data analytics [2, 3, 4]. This is further emphasized through the recently proposed Anonymized Network Sensing Graph Challenge [5], highlighting contributions to the generation and analysis pipeline of anonymized traffic matrix data from network packet captures (in the form of a standardized file format, namely, PCAP). Focusing on the analysis portion of this Graph Challenge, GPU-based analytics have become increasingly relevant within large-scale network data workloads [6, 7, 8]. Consequently, dense-GPU systems (i.e., 4–16 GPUs on a single node) are commonplace, with terabytes of aggregate memory and proprietary GPU interconnection network to enhance the overall bandwidth. However, designing codes to exploit several GPUs must contend with the challenges associated with effective workload distribution, host-device memory traffic, GPU interconnection performance, and complex algorithms [9], to name a few. These challenges were traditionally mitigated through vendor-based library solutions coupled with implementations explicitly composing the underlying data structures and associated operations. However, to enhance the portability of workloads targeting contemporary device/accelerator environments, driving the development using vendor-agnostic standardized programming models is key. Towards that goal, the C++ programming language has made significant inroads in improving performance and productivity to span across a wide variety of applications and hardware spectrum. C++26 is the "next" generation of the C++ standard (considered a landmark release since C++11 ushering in the modern C++ era, a decade ago), including significant changes

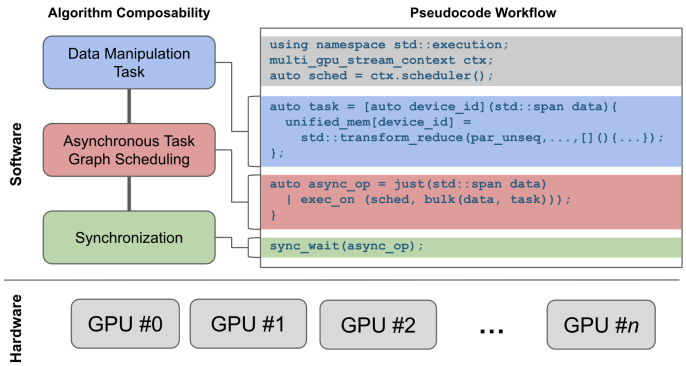

Fig. 1. C++26 asynchronous `Senders` programming model in `std::execution`. Using this standardized model, asynchronous workloads can be composed over diverse execution environments, for e.g., scheduling data manipulation tasks on multiple GPUs, captured here.

to the library itself relative to the core language features[1]. Specifically, C++26 Execution control library (referred as `std::execution` or `Senders` framework) provides composable building blocks for managing asynchronous execution on generic execution environments, as depicted in Fig. 1. This library allows transparent composition of task execution graphs or Directed Acyclic Graphs (i.e., DAGs similar to CUDA graphs[2]) and asynchronous scheduling on queues associated with specific execution contexts. It also introduces explicit syntax for launching the tasks (and invoking subsequent callbacks or "receivers"), thus, providing enhanced flexibility for common "many-tasks" scenarios compared to the existing C++ `std::future` or `std::packaged_task` interfaces.

These developments propose explicit high-level asynchronous semantics for GPU-based paradigms, allowing for the translation of "many-tasks" scenarios to many-threaded device contexts through vendor-optimized standardized library functions such as transform, reduce, scan, etc. Following this, we explore the efficacy of the C++26 `Senders` model on GPUs for the large-scale data analytics workload depicted in the Anonymized Network Sensing Graph Challenge.

This paper is one of the early works to explore C++26 `std::execution` workflows within large matrix workloads on multiple GPUs. Our contributions are as follows:

- Develop Anonymized Network Sensing Graph Challenge using recently proposed C++26 `Senders` model, leveraging data structures built on top of Standard Library containers for platform-agnostic analytics.
- Incorporated generic data batching scheme to subpartition network data between host and GPU (for handling partition sizes exceeding the available GPU memory).
- Achieved performance improvements of up to 55× compared to the serial GraphBLAS-based reference implementation using 8× NVIDIA A100 GPUs.

II. Background

A. Graph Challenge

The Anonymized Network Sensing Graph Challenge [5] highlights contributions towards improved network traffic analytics through multiple facets. These include the improvement of PCAP I/O, packet data extraction methods, IP anonymization, traffic matrix construction, matrix storage, and matrix analytics. This work focuses on improving the final step of network traffic analytics through the application of dense-GPU processing. At a high level, we outline our method in Fig. 2. We begin with the pre-processing steps adapted from the Graph Challenge proposal, extending the analysis portion to the loading of matrix files into generalized data containers. Using these, we follow propositions in the *C++26 Senders Model* workflow, performing data analytics by pushing asynchronous chains of workloads to a multi-GPU setup and retrieving the analytics output. Next, we describe the components of the *C++26 Senders Model*.

[1]https://en.cppreference.com/w/cpp/compiler_support/26
[2]https://developer.nvidia.com/blog/cuda-graphs/

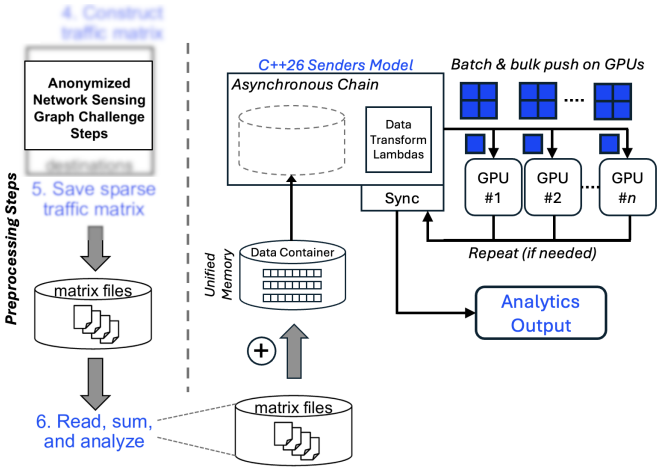

Fig. 2. Graph Challenge workflow overview with our proposed processing method using the *C++26 Senders Model*. The "Preprocessing Steps" graphics are adapted from [5]. Under the C++26 Senders Model, we load and aggregate traffic matrix files, forming an asynchronous workflow, comprised of batching and bulk pushing data operations to multiple GPUs.

B. C++26 Execution Model

Within the P2300 proposal for standard C++ [10], the authors propose an asynchronous task management workflow under the `std::execution` namespace for composable task-based parallel workloads launched on generic execution resources (i.e., an abstraction for a hardware component), handled by a scheduler. Overall, this proposal builds upon the popular asynchronous callback-promise workflow under a "senders-receivers" (`Senders`) model, associating an asynchronous unit of computation to a *sender* with a completed state being processed under a *receiver*. The *execution resource* of this model is expressed via an explicit scheduler abstraction; scheduler represents the place where execution happens (with options to compose senders into task graphs), which could be a thread-pool, single thread or GPU(s).

Vendor-specific implementations of `std::execution` are in active development to extend the `Senders` model to accelerators, such as GPUs. Specifically, NVIDIA is enhancing integration of proprietary CUDA with the C++ Standard Library (STL) functionalities on device through `libcudacxx` [11], backporting several libraries relative to data structures and operations under the `cuda::std` namespace. The integration of STL data structures and basic operations (such as `span`, `mdpsan`, `reduce`, etc.) is extended within the `Senders` model through `nvexec` [12], NVIDIA's device-based schedulers and functionalities for the relevant asynchronous abstractions. There are some generalizations, such as a default allocation of all used device memory to unified memory under `nvexec`. Overall, the `stdexec` library is a reference implementation of the C++26 Senders model, whereas NVIDIA C++17 Standard Parallelism (`stdpar`) allows GPU acceleration of the standard algorithms, both are integrated within the NVIDIA HPC SDK suite.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

III. Methodology

We introduce the key components in implementing network traffic analytics challenge using C++26 *senders*-based programming model for containerized data processing, in §III-A and §III-B. In §III-C, we discuss subpartitioning the input data for concurrent batch processing on GPUs.

A. Programming and Execution Model

From the perspective of our workload, high-level components of the C++26 asynchronous senders model are captured in Fig. 1. We first distinguish between the hardware and software layers of this model, followed by task composition.

Software Model: We first provide an overview of the pseudocode workflow for a generic transformation and reduction operation from Fig. 1, making generalizations for conciseness. The model relies on the usage of a vendor-specific device scheduler built on top of a single or multi-device context (depicted as `sched` in Fig. 1). Using this abstraction, programming on single vs. several devices (i.e., execution resources) are almost indistinguishable; we rely on a multi-GPU context to bulk push operations on a single dense-GPU node.

We consider the data manipulation tasks launched on devices as *lambda* functions (lambda expressions or lambdas are unnamed functions used to define operations where they are invoked), with `std::span` to represent a non-owning view of the contiguous data for processing. This allows for application extensibility, as we can process data of arbitrary contiguous sequences (i.e., trivially copyable)through `span`. Since vendor extensions such as `nvexec` (which provides NVIDIA specific schedulers and runtime support) assumes unified memory (i.e., single shared address space between host and device), we perform operations (in this case, `transform_reduce`) within the `std` namespace and commit results to unified memory relative to the device index (see Fig. 1).

The asynchronous data manipulation tasks are chained and associated with specific data spans. While supporting complex task workflows under `std::execution` for asynchronous chaining, for our purposes we can simply push bulk executions to multiple devices (reusing `sched`) and the set data manipulation lambda accordingly. We then perform a synchronization on this asynchronous chain, retrieving the result at the end of the workflow. This can be further customized with collective operations or other synchronization measures (e.g., multi-node scenarios).

Execution Environment: Traditional device-based applications rely on vendor-specific programming models, with separate multi-node or multi-GPU abstractions. In this model, various options exist to instantiate the underlying execution resource (i.e., `sched` in Fig. 1), comprising of multiple CPU/host threads, single GPU or dense-GPUs (all GPUs in a node), while keeping the asynchronous task descriptions uniform. With appropriate backend support, this model can be expanded to distributed implementations (over network) considering partitioned data.

Tasks Composability: The basis of the programming model relies on the generalization of any simple or complex parallel computation into three basic components: a data manipulation task, asynchronous data operations and bulk synchronizations. In Fig. 1, we provide an example of a generic transform and reduction operation (which transforms each element in the container in place and then accumulates the results), a common data manipulation task in a myriad of data analysis workflows. For our purposes, we demonstrate extensibility to common matrix-based network analysis measures, such as link counts, source counts and max degrees, among others. For more complex algorithms (i.e., which cannot be recast into a combination of containerized scan, reduce and transform operations), GPU integration with `stdpar` allows inclusion of device kernel within the data manipulation lambdas, allowing greater flexibility.

B. Standardized Containers

The usage of `std::span` within the programming model allows expansive data backends for portable analytics. Popular frameworks, such as cuGraph [13] or Gunrock [14], considers variety of input data formats (dedicated data ingestion frameworks such as cuDF [15] can alleviate preprocessing overheads), but often resort to containerized operations internally for implementing workloads. This method espouses the composable and generic aspects of C++ metaprogramming. Using this programming model, we adhere to similar principles, using `std::vector` container for the relevant data structures and passing an `std::span` for device invocations. Complex sparse representations, such as Compressed Sparse Row (CSR), can be managed through a collection of `std::vector` containers (e.g. consider a graph as a vector of a vector of vertex neighborhoods; operations such as triangle counting become a sequence of set intersections).

C. Concurrent Batching

Regarding data distribution across devices using multi-device schedulers, the usage of unified memory simplifies data access and allocation using on-demand page migration [16]. This, however, assumes a simple device-data partitioning occurs to split data evenly among devices. For our purposes, we rely on this even split of matrix data per device, which allows us to further take advantage of data *batching* on device.

We consider *batching* as a technique to subpartition input data into b_n batches, where each batch is processed in parallel one after the other. We detail this workflow in Fig. 3. The batch count b_n is input specified, where we assume $b_n = 1$ is the default case (where the batch is the entire device data partition), and $b_n = k$ splits the input data into k portions per device. This allows us to leverage sequential processing for better scalability and lower page faults through reduced data sizes at processing time. We further discuss the impact on performance in §V.

IV. Implementation Details

In this section, we discuss our implementation alongside the design choices for Graph Challenge analytics, listed in Table I.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

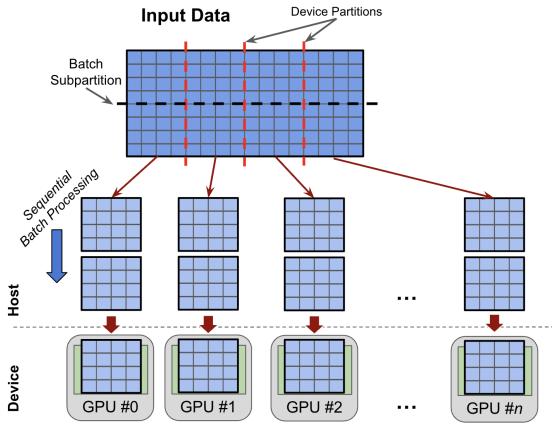

Fig. 3. Demonstrates input data batching from host to GPUs, where individual device partitions are sub- partitioned into fixed-size batches, sequentially moved on GPUs by the host during computation.

TABLE I
GRAPH CHALLENGE PACKET ANALYSIS MEASURES, WITH THE AGGREGATE PROPERTIES AND SUMMATION NOTATIONS ADAPTED FROM [5]. THE RELEVANT PROGRAMMING MODEL DATA OPERATION IS LISTED FOR EACH PROPERTY. A_t REPRESENTS A NETWORK TRAFFIC MATRIX AT TIME t, WITH $A_t(i,j)$ AS THE NUMBER OF PACKETS BETWEEN SOURCE i AND DESTINATION j.

Aggregate Property	**Notation**	**C++ function**
Valid packets N_V	$\sum_i \sum_j A_t(i,j)$	`reduce`($weights$)
Unique links	$\sum_i \sum_j \lvert A_t(i,j)\rvert_0$	`size`($edges$)
Unique sources	$\sum_i \left\lvert \sum_j A_t(i,j)\right\rvert_0$	`size`(row_sums)
Max source fan-out	$\max_i \left\lvert \sum_j A_t(i,j)\right\rvert_0$	`max`($degrees$)
Unique destinations	$\sum_j \left\lvert \sum_i A_t(i,j)\right\rvert_0$	`size`(col_sums)
Max destination fan-in	$\max_j \left\lvert \sum_i A_t(i,j)\right\rvert_0$	`max`($degrees$)

A. Software Dependencies

We utilize a number of vendor-based libraries that integrate C++26 `Senders` model execution workflows on GPUs, specifically NVIDIA CUDA-based implementations of `std::execution` workflows through `nvexec` [12] for the translation of asynchronous workflows and operation lambdas to device code. `nvexec` also provides the multi-GPU scheduler context used to designate the execution resource. Our containerized data structures are STL-based and uses `libcudacxx` or `libcu++` [11] for automatic unified memory management. We further use `libcu++` for device implementations of the STL operations, such as `std::reduce` available in `cuda::std` namespace.

B. Data Representation

In the processing of the network traffic matrix files, we refer to the sequential GraphBLAS-based implementation discussed in [5]. Instead of GraphBLAS data representations, we use generic STL-based containers, requiring intermediate transformation of the CSR into flat containers comprising of the $edges$, $degrees$ and $weight$ data for the respective nonzero entries. Using these source containers, we can build derivative containers for specific data operations, such as row_sums and col_sums for out– and in–degrees, respectively.

Algorithm 1 Pseudocode for *max reduction*

Input: Container: $data$, Multi-GPU Context: ctx, Batch Count: $b_n : n \in \mathbb{R}_{>0}$
Output: Maximum value

```
1:  procedure MAX_LAMBDA(cuda::std::span(data),
    cuda::std::span(result)
2:      d ← device_id
3:      result[d] ← max(result[d],
4:          cuda::std::reduce(data, std::max))
5:  sched ← ctx.GET_SCHEDULER()
6:  cuda::std::span result ← ∅
7:  for each batch b of data do
8:      subspan ← cuda::std::span(b)
9:      sndr ← stdexec::just(subspan, result)
10:     | exec_on(sched, stdexec::bulk(size(b),
11:                 MAX_LAMBDA)
12:     stdexec::sync_wait(std::move(sndr))
13: return result
```

C. Analytics and Operations

The associated analytics include source, destination and total packet counts, maximum fan-in/fan-out and maximum link counts. For all the required measures, a combination of sum reductions and maximum scans suffice, of which we generalize for repeated data processing among the containers. The Graph Challenge properties are listed in Table I, alongside data manipulation operations used to implement the specific functionality. Each operation can be successively invoked on subsequent batches of data, analyzing sub-partitioned portions of the packet data on device (this option is discussed in §III-C). Details of the two primary operations (i.e., scan and reduce) are as follows:

Maximum Scan: A concise example of the maximum reduction implementation is listed in Pseudocode 1, represented as the lambda expression MAX_LAMBDA, which performs maximum reduction on device using `cuda::std::reduce` on a unified memory-backed `span`. The data can be optionally processed in more than one batches, designating an even number of sub-partitions of the data relative to the number of batches in $subspan$. The asynchronous tasks are encapsulated through the *sender* $sndr$, passing the input span and result. Next, we specify the resource context (passing the scheduler and lambda) on which the *sender* will be executed (multiple senders can be successive chained/scheduled, i.e., utilizing overloaded `operator|`). Finally, a blocking synchronization on sender completes the operation.

Sum Reduction: For similar reduction or single buffer type operations, we follow the exact same structure as in Pseudocode 1. We can simply replace the data manipulation operation in Lines 3 and 4 with the relevant operation, and perform the same workflow.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

V. Evaluations

We first describe our software and hardware setup, followed by execution time performance results and an overall analysis. We compare our implementation to the sequential GraphBLAS-based Python implementation provided as part of the Graph Challenge [5].

Platform: Our primary GPU platform in our evaluations is an NVIDIA™ DGX system with 8 GPUs (DGX-A100). The DGX A100 is the third generation server node from NVIDIA™, and consists of 8 "Ampere" A100 GPUs (with 108 SMs) with 40GB HBM2 memory/GPU and two-way 64-core AMD EPYC 7742 CPUs at 2.25GHz, 256MB L3 cache, 8 memory channels, and 1TB DDR4 memory. NVIDIA GPUs either come in the PCIe or proprietary NVLink/NVSwitch based form factors. Proprietary SXM module allows NVIDIA GPUs to directly communicate through NVLink interconnect (DGX-A100 uses SXM4).

Our implementation is built using CUDA version 12.2, GCC version 13.3.0 and version 25.5-0 of the NVIDIA High Performance Computing Software Development Toolkit[3] for `nvc++`. We use `std=c++20`, as C++26 features are experimental using the `--experimental-stdpar` flag. Experimental NVIDIA device support features are available using the `-stdpar=gpu` flag. Our source code is openly available from GitHub: https://github.com/mmandulak1/stdexecANSGC.

Dataset: We use the specific data required for this Graph Challenge [5], which are GraphBLAS matrices derived from randomized network packet data of 2^{30} synthetic packets.

A. Baseline Execution

We separate baseline execution time assessments into two tasks: *analysis time* and *end-to-end time*. Analysis time is the time taken to calculate the analysis measures, excluding preprocessing (e.g., data structure construction and file I/O). End-to-end time is the entire program execution time for the analysis. For each case, we collect the results of our implementation, scaling up from 1–8 GPUs, and compare to the sequential Graph Challenge baseline. The least of 5 test runs is reported in the results. We present the analysis and end-to-end results in Figures 4, 5 and 6, respectively, including batching variation in batch counts of 1, 5 and 10 (see §III-C). We summarize our results as follows, followed by a discussion on the impact of concurrent batching.

Analysis Time: In Figures 4 and 5, we present execution time scaling per GPU count and overall performance improvement relative to the sequential GraphBLAS-based implementation. We observe the lowest execution time using a batch count of 10 and 8 GPUs of 1.17 seconds, compared to the sequential baseline of 64.23 seconds. At its peak, we observe a 55× performance improvement upon the sequential in analysis timing, with a geometric mean performance improvement at 8 GPUs of 47× across all three batch counts. We attribute performance improvements to improved data distribution using higher device counts, leveraging parallel performance.

[3]https://developer.nvidia.com/hpc-sdk

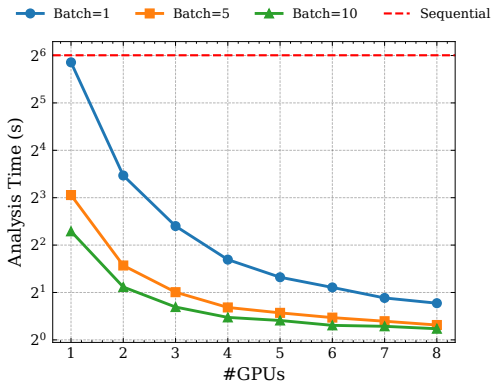

Fig. 4. Scalability (analysis time, *lower* is *better*) on 1–8 GPUs with varying batch counts. Best performance is observed using 8 GPUs and 10 batches at ∼1 seconds compared to ∼64 seconds for sequential baseline.

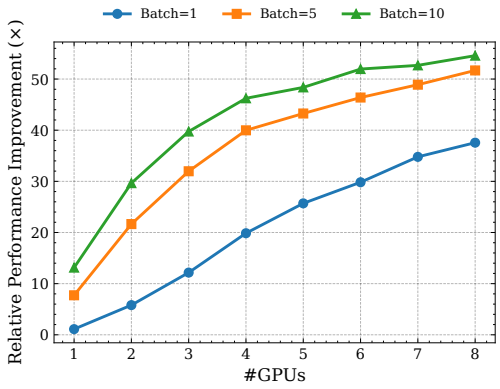

Fig. 5. Relative performance improvement (compared to serial reference implementation, *higher* is *better*) on 1–8 GPUs with varying batch counts. Best performance observed at 8 GPUs using 10 batches: 55×.

Between batch counts, we observe the highest improvement using a batch count of 10, which noticeably outperforms the other batch counts across all GPU counts. On average, the larger batch count yields up to 170% improvement in performance upon a batch count of 1 and up to 20% at a batch count of 5. On average, using batching in this regard yields a 140%improvement over the default case of a batch count of 1. In the single GPU case, the limitations in flexibility of even data distribution on device are very apparent in performance impacts. While the usage of higher device counts alleviates workloads for better distribution, the combination of batching and higher device counts yields the best flexibility for performant data distribution with lower data sizes at a given processing point.

End-to-End Time: We present the end-to-end performance results in Fig. 6, depicting the execution times with varied batch and GPU counts. Aside from the analytics time, we note that the data loading task is relatively expensive, taking approximately 40 seconds. Consequently, host to device data movement costs are nontrivial (about 100 seconds on a single GPU considering a single batch). Still, we demonstrate improvements relative to the sequential baseline by ∼4×, with

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

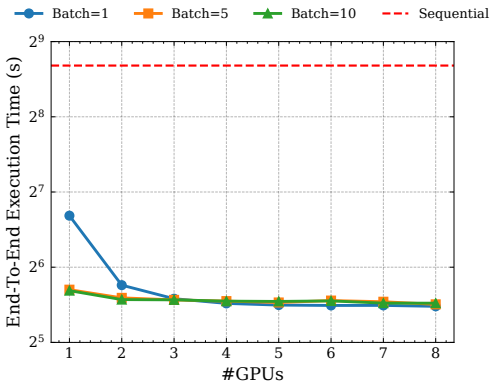

Fig. 6. Scalability (end-to-end time, *lower* is *better*) on 1–8 GPUs with varying batch counts. Best performance observed using 8 GPUs and default batch count of 1, leading to 9× improvement vs. sequential.

the highest improvement using a single batch at 8 GPUs by about 9×. By geometric mean, we observe 8× execution time performance improvement across all data points. We further compare implicitly with the 2024 Graph Challenge champion [17], with approximately 2× improvement in end-to-end execution time relative to their multithreaded implementation.

As shown in Fig. 6, for #GPU >3, a single batch outperforms that of 5 and 10 batch counts, while a batch count of 10 shows the best performance for #GPUs $\leq$ 3. This can be attributed to the relatively expensive data movement and initial container building costs combined with uneven workload distributions owing to the input data. Similarly, we see minimal improvement using 10 batches over 5, with a 1.2× improvement and an average improvement with a batch count >1 of 1.1×.

Batching: Relative to performance, batching serves as a means to balance workload distribution as GPU counts increase. In practice with C++26 `std::execution` device workload scheduling, resources appear to be scheduled efficiently relative to data sizes. Using the `nvexec` scheduler, if a workload fits in the capacity of a single GPU, resources are scheduled accordingly. This does not necessarily balance towards workloads, which is remedied through batching, providing a pre-scheduler workload distribution before bulk pushes to devices. This, alongside the sequential processing loop coupled with batching, results in variable performance improvements relative to tradeoffs in device counts and workload distributions. For our purposes, we express the usefulness of batching in large data instances with low device counts within the non-complex analytics workloads presented.

TABLE II
BEST PACKET RATE (*Higher* IS *Better*) PER BATCH AND #GPUS.

Batch Count	Best Packet Rate (packets/s)	#GPUs
Sequential	2,614,183	-
1	**24,061,441**	8
5	23,626,502	8
10	23,372,598	8

B. Packet Processing

We further summarize our end-to-end execution time performance relative to the sequential in terms of network packets processed per second. This metric is calculated by recording the best end-to-end execution time relative to the number of packets in the randomized network traffic dataset. We highlight the best rate per batch count in Table II, observing approx. 9× improvement as compared to the reference baseline, with analysis times following at approximately 50× over the same.

C. Observations

We discuss the observed productivity and challenges of C++26 `std::execution` model for multi-device execution.

Productivity: Primary benefit of the C++26 execution workflows is flexible device oriented programming through standardized solutions enhanced with vendor optimizations (e.g., device memory management via standardized containers and unified memory). Our implementation is approximately 20 lines of code (LoC) for each of the maximum scan and sum reduction operations, and in total about 100 LoC including the driver functions. We further note that our implementation includes no explicit CUDA code and relies fully on vendor functionalities supplied within the standard. This is beneficial for free performance gains across GPU generations, without the need to adapt entire workflows to be device-aware, as long as the methods conform to a set of standard operations. Complex operations can be included through explicit device code within the lambdas.

Challenges: Notable challenges exist in the conversion of complex algorithms to fit within the C++26 `std::execution` workflows, especially in the domain of graph problems. The straightforward bulk pushing of operations to devices limits user flexibility in controlling data distribution (which led us to consider batching), which is critical for irregular and hierarchical data instances. Furthermore, de facto unified memory requires careful examination of device data accesses for cache efficiency. These considerations, alongside those typical in parallel graph problems (e.g., load imbalance), still require interventions, as the scheduler does not act as an all-encompassing distributor across expansive problem domains.

VI. CONCLUDING REMARKS

We utilize the recently proposed C++26 `std::execution` model on dense-GPU platforms, splitting the network traffic analytics workload into composable set of data manipulation operations as asynchronous task graphs. In addition, we show that explicit control of the data distributions across GPUs (i.e., batching) is still relevant for modern programming abstractions. Despite using high-level abstractions, we achieve up to 55× execution time improvement relative to the sequential linear-algebra baseline and about 2× improvement in implicit comparison of end-to-end execution time to the streaming-based multithreaded implementation by the 2024 Graph Challenge champion.

979-8-3315-7844-2/25/$31.00 ©2025 IEEE

VII. Acknowledgments

This work is in parts supported by the National Science Foundation under Grant No. 2047821, the DOE ASCR End-to-end co-design for performance, energy efficiency, and security in AI-enabled computational science (ENCODE) project at PNNL, and the Laboratory Directed Research and Development program at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DE-AC05-76RL01830. We would also like to thank Dr. Tim Carlson at PNNL and the PNNL Research Computing staff for their continuous support.

References

[1] J. Kepner, M. Jones, P. Dykstra, C. Byun, T. Davis, H. Jananthan, W. Arcand, D. Bestor, W. Bergeron, V. Gadepally, M. Houle, M. Hubbell, A. Klein, L. Milechin, G. Morales, J. Mullen, R. Patel, A. Pentland, S. Pisharody, A. Prout, A. Reuther, A. Rosa, S. Samsi, T. Trigg, C. Yee, and P. Michaleas, "Focusing and calibration of large scale network sensors using graphblas anonymized hypersparse matrices," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2023, p. 1–9. [Online]. Available: http://dx.doi.org/10.1109/HPEC58863.2023.10363471

[2] S. R. Jino Ramson and D. J. Moni, "Applications of wireless sensor networks — a survey," in *2017 International Conference on Innovations in Electrical, Electronics, Instrumentation and Media Technology (ICEEIMT)*, 2017, pp. 325–329.

[3] I. Kawaminami, A. Estrada, Y. Elsakkary, H. Jananthan, A. Buluc, T. Davis, D. Grant, M. Jones, C. Meiners, A. Morris, S. Pisharody, and J. Kepner, "Large scale enrichment and statistical cyber characterization of network traffic," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2022, p. 1–7. [Online]. Available: http://dx.doi.org/10.1109/HPEC55821.2022.9926397

[4] W. Zhou, Y. Zhou, J. Li, and M. H. Memon, "Lsrec: Large-scale social recommendation with online update," *Expert Systems with Applications*, vol. 162, p. 113739, 2020.

[5] H. Jananthan, M. Jones, W. Arcand, D. Bestor, W. Bergeron, D. Burrill, A. Buluc, C. Byun, T. Davis, V. Gadepally, D. Grant, M. Houle, M. Hubbell, P. Luszczek, P. Michaleas, L. Milechin, C. Milner, G. Morales, A. Morris, J. Mullen, R. Patel, A. Pentland, S. Pisharody, A. Prout, A. Reuther, A. Rosa, G. Wachman, C. Yee, and J. Kepner, "Anonymized network sensing graph challenge," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2024, p. 1–8. [Online]. Available: http://dx.doi.org/10.1109/HPEC62836.2024.10938508

[6] W. Wu and P. Demar, "A gpu-accelerated network traffic monitoring and analysis system," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013, pp. 77–78.

[7] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: revitalizing gpu as packet processing accelerator," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 83–96.

[8] W. Zhang and J. P. Lazaro, "A survey on network security traffic analysis and anomaly detection techniques," *International Journal of Emerging Technologies and Advanced Applications*, vol. 1, no. 4, p. 8–16, May 2024. [Online]. Available: https://www.ijetaa.com/article/view/117

[9] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[10] J. Hoberock, E. Niebler, L. Gooch, and B. A. L. Maurer, "P2300R10: std::execution," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html, 2024, iSO C++ Committee Paper, Accessed: 2025-07-07.

[11] N. Corporation, "libcu++: The NVIDIA C++ Standard Library," https://github.com/NVIDIA/libcudacxx, 2024, accessed: 2025-07-07.

[12] ——, "nvexec: Sender/Receiver Framework for GPU-Centric Asynchronous Programming," https://github.com/NVIDIA/nvexec, 2024, accessed: 2025-07-07.

[13] N. Corporation and R. AI, "cuGraph: GPU Accelerated Graph Analytics," https://github.com/rapidsai/cugraph, 2025, accessed: 2025-07-07.

[14] Y. Wang, A. Davidson, Y. Pan, Y. Wu, C. Yang, and J. D. Owens, "Gunrock: GPU Graph Analytics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/SC.2016.55

[15] R. AI and N. Corporation, "cuDF: GPU DataFrame Library for Python," https://github.com/rapidsai/cudf, 2025, accessed: 2025-07-07.

[16] N. Sakharnykh, "Everything you need to know about unified memory," in *GPU Technology Conference (GTC)*, vol. 64, 2018.

[17] K. Zhao, Y. Zhou, H. Pan, Z. Wang, S. Zhong, and C. Tian, "Sans: Streaming anonymized network sensing," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 2024, pp. 1–7.