Processing Event-Based data using MPI and OpenMP

Juan-Pablo Romero-Bermudez

Final Project

FDD3260

1. Introduction

This document presents a high-performance computing (HPC) pipeline for analyzing event-based data, a critical component of neuromorphic computing. The primary objective of this project is to demonstrate the effectiveness of HPC systems, specifically leveraging shared and distributed memory processing, in optimizing the analysis of large-scale event recordings. By utilizing the powerful computational capabilities of HPC systems like Dardel, this project aims to explore how different parallel processing approaches can be employed to efficiently handle the unique challenges posed by event-based data.

The document is structured as follows: §2 provides background on event-based vision HPC, §3 discusses the methodology applied, detailing the pipeline used for processing event-based data and covering how shared and distributed memory parallelism, using OpenMP and MPI, were implemented to handle the datasets and perform the required analyses, §4 presents the results, highlighting the performance improvements achieved through various parallel processing techniques, including the use of GPU offloading and finally §5 provides some conclusions on the suitability of different HPC approaches for event-based data analysis.

2. Background

This document proposes a high-performance computing (HPC) pipeline for analyzing event-based data. The pipeline aims to demonstrate how, and in which specific scenarios, shared and distributed memory processing on HPC systems can optimize the analysis of large-scale event recordings. This section has two main objectives: first, to introduce event-based vision and clarify the inherent challenges it presents (§2.1); and second, to explain why HPC is well-suited for processing event-based data (§2.2).

2.1. Event-based vision

Event-based vision, a key area of neuromorphic computing, utilizes event-based cameras to capture visual information in a fundamentally different way than conventional frame-based cameras, Figure [1]. Instead of recording the entire scene at fixed intervals, event-based cameras asynchronously detect changes in the scene at the pixel level. Each pixel operates independently and generates an event only when a change in light intensity is detected, encoding both the timestamp and location of the event along with the change's polarity. This approach results in

sparse, high-temporal-resolution data that closely mimics biological vision, providing significant advantages in dynamic environments, low-latency applications, and power efficiency. Nonetheless, the unique characteristics of event-based data also pose several computational challenges.

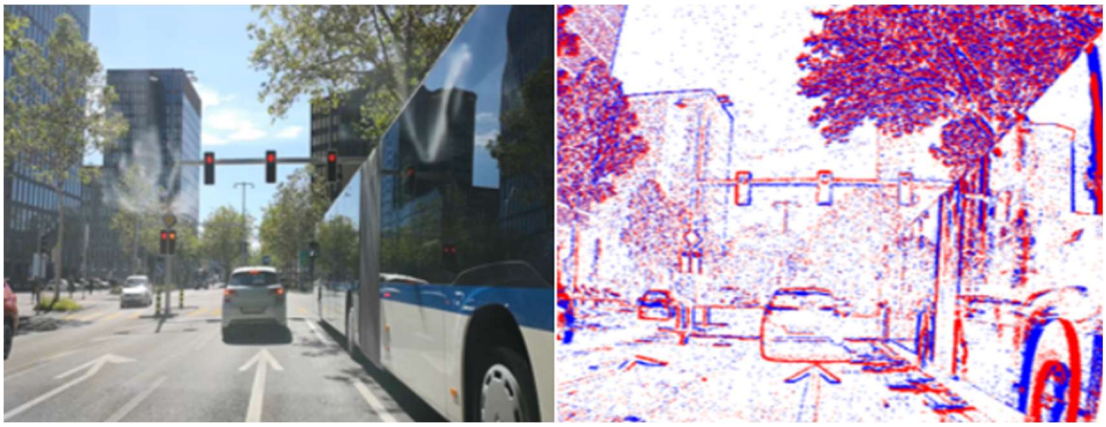


Figure [1] RGB conventional camera output vs visualization of Event-based camera output.

The asynchronous and sparse nature of the data leads to irregular and potentially large-scale datasets that are difficult to process efficiently using traditional computing methods. The datasets, often consisting of millions to billions of events, require specialized algorithms capable of handling high-dimensional data streams. Moreover, processing such data involves temporal and spatial analysis to extract meaningful patterns, which can be computationally expensive. To handle the scale and complexity of these datasets, High-Performance Computing (HPC) resources become essential. HPC environments offer the necessary computational power and parallelism to efficiently manage large event-based datasets. Utilizing multi-core CPUs, GPUs, and specialized accelerators, HPC systems can accelerate the analysis of these data.

2.2. HPC

High-Performance Computing (HPC) refers to the use of supercomputers and parallel processing techniques to solve complex computational problems that are beyond the capabilities of standard computers. HPC systems leverage thousands or even millions of processing cores to perform trillions of calculations per second, making them ideal for tasks that require immense computational power, such as scientific simulations, weather modeling, data analysis, and machine learning. The primary advantage of HPC lies in its ability to process large datasets and perform calculations at high speeds, which significantly reduces the time required to reach solutions. This capability is crucial for advancing research and innovation in fields that rely on intensive computation, enabling scientists, engineers, and analysts to model and analyze complex phenomena, optimize processes, and develop new technologies more efficiently.

2.2.1. Dardel

An example of an HPC system is Dardel, an HPE Cray EX supercomputer which features a CPU partition, suitable for a broad range of computational applications, and a GPU partition, intended for the most computationally demanding applications. The CPU partition consists of 1278

compute nodes. Each node has two AMD EPYC™Zen2 2.25 GHz 64-core processors, which means that each compute node has a total of 128 physical CPU cores or 256 virtual cores. This gives a total of 1278 nodes. As each node has 128 physical cores, that means the CPU partition overall has 163,584 physical cores. The GPU partition comprises 62 GPU nodes, each of which is configured with one AMD EPYC™processor with 64 cores and four AMD Instinct™MI250X GPU chips, each with two GPU devices, that have an impressive performance of up to 95.7 TFLOPS in double precision when using special matrix operations.

Figure [2] A picture of Dardel

2.2.2. Shared/Distributed Memory Parallelism

In Dardel, like in other HPC systems, it is possible to run both shared and distributed memory applications. For shared memory applications, such as those utilizing OpenMP, tasks are parallelized within a single node by dividing them among multiple threads that share the same memory space. This approach simplifies data sharing and synchronization but is limited to the confines of a single node. On the other hand, distributed memory applications, which use MPI, involve multiple nodes communicating over a network, each with its own memory space. This model scales effectively across many nodes and is suitable for handling extensive datasets and complex computations. Additionally, in Dardel, tasks can also be offloaded to GPUs to further enhance performance. GPUs are particularly adept at handling parallel computations and can be integrated with both MPI and OpenMP to accelerate processing.

3. Methodology

As mentioned earlier, event-based data recordings, although typically large[1], are straightforward in structure, with each event represented by 12 bytes containing both temporal and spatial information. The aim of the project documented here was to generate, separately, histograms and heatmaps for data analysis from both temporal and spatial perspectives. A set of K×N files, each representing a 2-second recording of a specific scene, was processed. The approach to parallelization, illustrated in Figure [3], involved distributing the K×N files across N nodes, with

[1] An event-based camera with VGA resolution (640x480 pixels) can easily produce 2 million events per second when looking, for example, at a static scene (i.e. static camera) of a human making hand gestures. This represents 24MB of data stored every second.

each node handling K files sequentially. Each file was divided into M chunks of approximately equal size, allowing for parallel processing using OpenMP threads within each node. MPI was used to manage inter-process communication, while OpenMP facilitated parallel execution of tasks within each MPI process. Upon completion of all parallel tasks, the OpenMP parallel regions and the MPI environment were finalized. The tasks, as previously outlined, were 1) generating histograms and 2) generating heatmaps. That said, while the outputs of the tasks have been described as histograms and heatmaps, they are actually constant-sized binary files[2] representing histograms and heatmaps. To illustrate that heatmap generation was more suitable for GPU processing, these tasks were executed in separate runs. Consequently, four programs, detailed in Table [1], were compiled from a common source with minor variations introduced through compilation switches. Each program was executed six times for the same combination of nodes and CPUs. The node counts tested were {1, 2, 3, 4}, and the CPU counts tested were {2, 4, 8, 16, 32, 48, 64, 96, 128}. In total 432[3] runs, per task, were performed and the results obtained are condensed in §4.

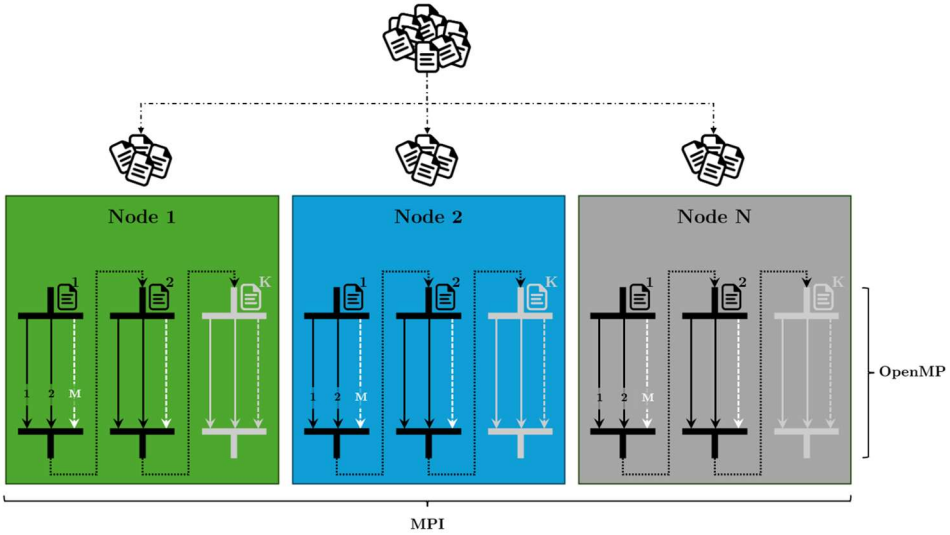


Figure [3] Approach to parallelization: KxN files distributed into N nodes. With the use of MPI primitives, each node handles K files. Within each node, each file is carefully divided into M chunks that are processed in parallel using OpenMP.

Programs	Task	GPU
gpu_mpi_hg_open_mp.exe	Histograms	✓
mpi_hg_open_mp.exe		✗
gpu_mpi_hm_open_mp.exe	Heatmaps	✓
mpi_hm_open_mp.exe		✗

Table [1] List of programs using the pipeline of Figure [3]. The programs are associated with a particular task (histograms or heatmaps) and the use, or not, of GPU for offloading.

[2] Size of histograms: 8000 bytes = [2000 bins of 1ms] x [4 bytes (to count occurrences)]
Size of heatmaps: 1228800 bytes = [640 pixels in x] x [480 pixels in y] x [4 bytes (to count occurrences)]
[3] 432 = [4 node counts] x [9 CPU counts] x [2 GPU counts (with/without)] x [6 runs per combination]

4. Results

As an example of the output produced by the tasks executed using the pipeline described in §3, Figures [4] and [5] show a histogram and a heatmap corresponding to the same recording[4].

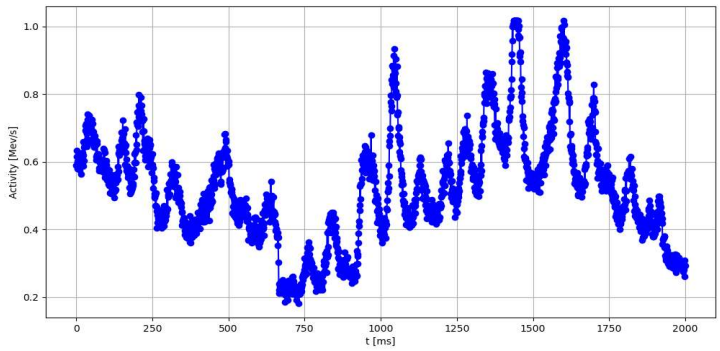


Figure [4] Example of histogram from event data

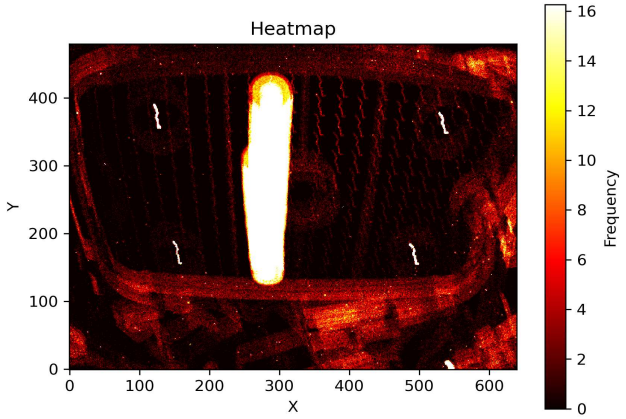


Figure [5] Example of heatmap from event data

The results of 432 runs per task are condensed in Table [2] and Table [3] for histograms and heatmaps respectively. An average elapsed wall-clock time, with the corresponding standard deviation, is shown for each of the 72 combinations (nodes/CPUs/GPU) in each table.

The obvious result is that distributing the tasks into more nodes reduces the time spent as shown in Figures [6] and [7]. This relationship seems to follow the following relationship:

$$T_T(N) = \frac{F \cdot T_F}{N} + T_C(N)$$

Where T_T is the total time, F is the total number of files, T_F the time spent to process a single file, T_C time spent in internode communication and N the number of nodes available.

[4] The plots are created using Python on a CPU based on the binary outputs from the tasks

Time [s]		Without GPU offloading				With GPU offloading			
		Nodes				Nodes			
		1	2	3	4	1	2	3	4
CPUs	2	123.34 ± 0.37	72.04 ± 0.24	56.88 ± 0.24	43.18 ± 0.09	**130.05 ± 0.93**	75.54 ± 0.34	60.03 ± 0.06	44.35 ± 0.46
	4	79.51 ± 0.83	44.95 ± 1.8	37.02 ± 0.11	27.85 ± 0.02	86.87 ± 0.2	50.69 ± 2.13	39.4 ± 0.35	29.76 ± 0.15
	8	43.71 ± 0.48	25.14 ± 0.18	20.5 ± 0.09	15.35 ± 0.28	51.0 ± 0.82	28.8 ± 0.0	23.01 ± 0.09	17.3 ± 0.31
	16	22.55 ± 0.5	13.29 ± 0.14	10.94 ± 0.18	8.12 ± 0.09	29.82 ± 0.05	17.05 ± 0.04	13.34 ± 0.0	10.03 ± 0.22
	32	9.36 ± 0.46	5.53 ± 0.15	4.14 ± 0.03	3.35 ± 0.04	17.4 ± 0.05	9.01 ± 0.04	6.87 ± 0.32	5.19 ± 0.22
	48	6.04 ± 0.15	3.45 ± 0.09	2.85 ± 0.45	2.07 ± 0.06	13.18 ± 0.06	7.01 ± 0.27	5.94 ± 0.76	4.1 ± 0.22
	64	4.75 ± 0.06	2.54 ± 0.01	2.21 ± 0.19	2.48 ± 0.26	12.28 ± 0.01	6.54 ± 0.24	4.95 ± 0.26	3.98 ± 0.84
	96	3.31 ± 0.6	1.91 ± 0.2	1.49 ± 0.27	1.21 ± 0.17	10.78 ± 0.0	5.48 ± 0.19	4.05 ± 0.01	3.29 ± 0.31
	128	2.43 ± 0.15	1.34 ± 0.12	0.99 ± 0.12	**0.9 ± 0.06**	9.79 ± 0.12	5.21 ± 0.02	3.67 ± 0.19	2.97 ± 0.1

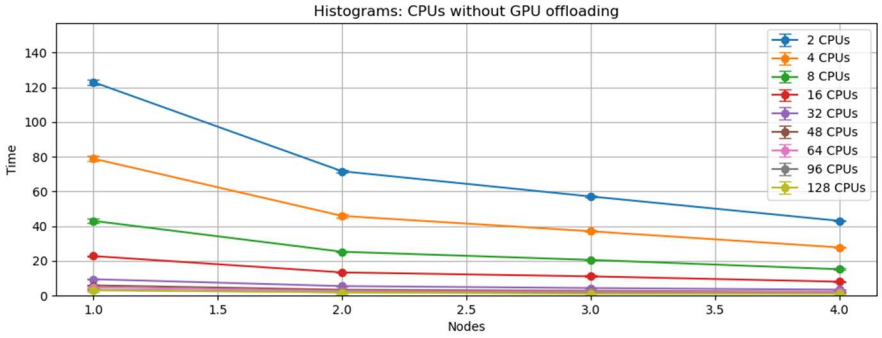
Table [2] Results when running histogram-related programs.

Time [s]		Without GPU offloading				With GPU offloading			
		Nodes				Nodes			
		1	2	3	4	1	2	3	4
CPUs	2	**122.8 ± 2.17**	71.71 ± 0.28	57.65 ± 0.46	43.18 ± 0.04	121.77 ± 2.58	72.61 ± 0.07	55.97 ± 0.75	43.45 ± 0.2
	4	78.97 ± 0.8	44.78 ± 3.04	37.57 ± 0.58	28.11 ± 0.0	77.12 ± 1.83	46.35 ± 0.82	35.4 ± 0.35	28.23 ± 0.02
	8	43.71 ± 0.14	25.35 ± 0.04	20.6 ± 0.02	15.5 ± 0.04	43.1 ± 0.69	25.07 ± 0.78	20.48 ± 0.36	15.59 ± 0.0
	16	23.79 ± 0.05	13.66 ± 0.19	11.16 ± 0.03	8.41 ± 0.19	22.83 ± 0.08	13.41 ± 0.09	10.92 ± 0.05	8.21 ± 0.14
	32	13.25 ± 0.48	7.45 ± 0.2	5.5 ± 0.28	3.85 ± 0.21	10.87 ± 1.16	6.09 ± 0.33	4.93 ± 0.06	3.54 ± 0.1
	48	10.37 ± 0.27	5.58 ± 0.02	4.21 ± 0.11	3.08 ± 0.02	7.4 ± 0.57	3.84 ± 0.18	3.34 ± 0.16	2.45 ± 0.17
	64	11.02 ± 0.79	5.76 ± 0.19	4.29 ± 0.32	3.13 ± 0.04	5.81 ± 0.02	3.46 ± 0.05	2.85 ± 0.35	1.93 ± 0.01
	96	11.71 ± 0.25	6.06 ± 0.24	4.3 ± 0.19	3.45 ± 0.44	4.5 ± 0.05	2.52 ± 0.0	2.08 ± 0.4	1.97 ± 0.28
	128	13.82 ± 0.02	7.05 ± 0.05	4.81 ± 0.02	3.93 ± 0.03	3.95 ± 0.16	2.27 ± 0.04	1.66 ± 0.07	**1.6 ± 0.15**

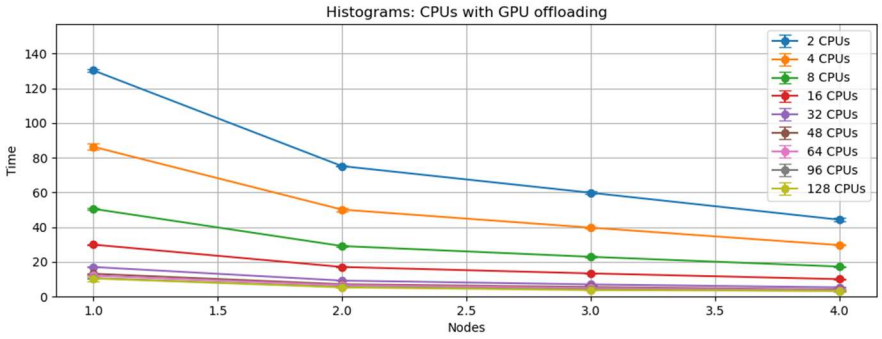
Table [3] Results when running heatmap-related programs

Another predictable result is that, in general, increasing the number of CPUs, M[5], involved in processing each file also contributes to reducing the overall time spent as shown in Figures [8] and [9b]. However, it was noted that this relationship broke for the heatmaps task without GPU offloading, as shown with Figure [9a]. To be specific, it was observed that using a number of and CPUs above 48 was counterproductive for the heatmaps task, see highlighted row in table [3]. This is due to the way the task is implemented: a 3D matrix of depth M is created to count events occurring within the pixel space and then proceeds to squeeze such matrix into a 2D version of it using a for loop. As a solution to this problem, GPU offloading was introduced, and the results show that this decision was highly beneficial for the heatmaps task although it was counterproductive for the histograms task. Overall, the results show that the histograms task was performed faster when using 128 CPUs and 4 nodes without GPU offloading (0.9±0.06 [s]) and slower when using 2 CPUS and 1 node with GPU offloading (130.05±0.93 [s]) which represents a speed up factor of more than x140. As for the heatmaps task, it was performed faster when using 128 CPUs and 4 nodes with GPU offloading (1.6±0.15 [s]) and slower when using 2 CPUS and 1 node without GPU offloading (122.8±2.17 [s]) which represents a speed up factor of almost x80. These results make it clear that the histograms task does not require GPU offloading, and that by offloading it more time is spent mapping memory than time is saved parallelizing the job. Additional figures about the differences between offloading and not offloading tasks on GPU are available in the Supplement.

[5] M: the number of virtual cores and OpenMP threads used. M/2 is the actual number of physical CPUs

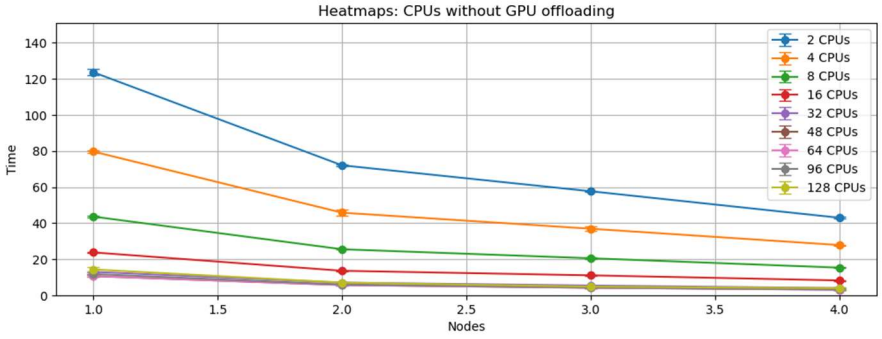


a)

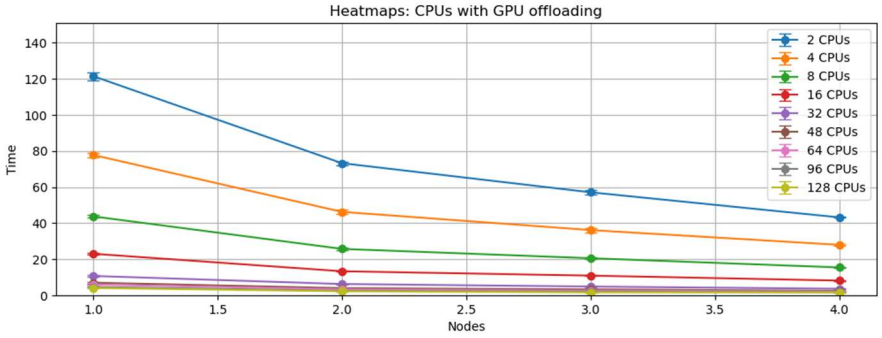


b)

Figure [6] Histograms task: time spent processing vs nodes used

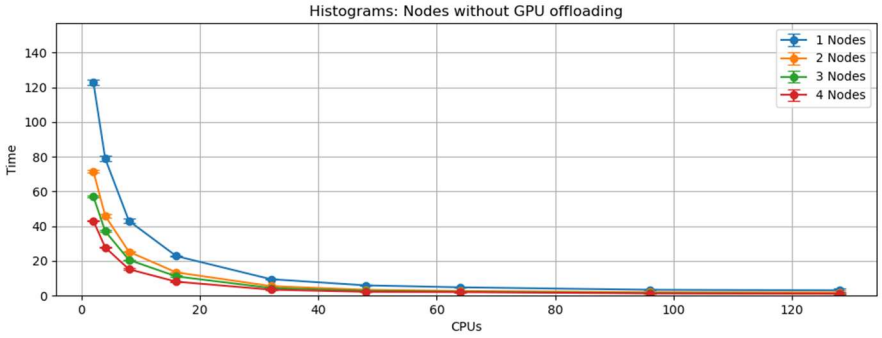


a)

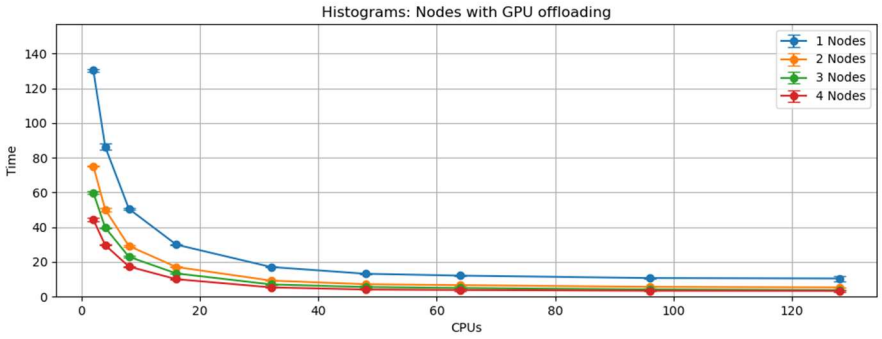


b)

Figure [7] Heatmaps task: time spent processing vs nodes used

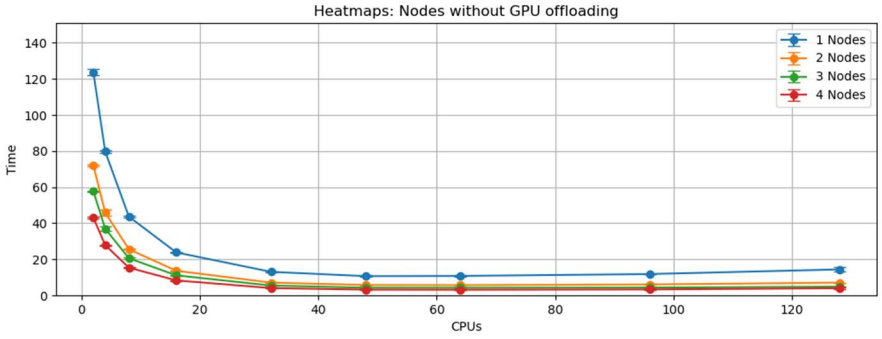


a)

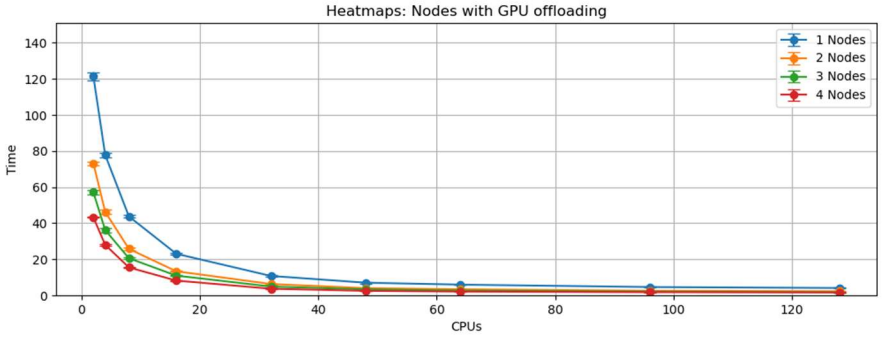


b)

Figure [8] Histograms task: time spent processing vs CPUs used



a)



b)

Figure [9] Heatmaps task: time spent processing vs CPUs used

5. Related work

Most existing approaches to analyzing event-based data rely on standard CPUs, often focusing on small numbers of files or limited datasets due to the relatively low computational demands of these tasks. Typically, researchers process event-data files individually, without resorting to high-performance computing (HPC) environments, because the data volumes are manageable, and the algorithms are optimized for single-threaded or mildly parallel operations. Additionally, event-based data is often processed using neuromorphic hardware, which is designed to mimic the brain's neural networks and is ideally suited for handling the asynchronous and sparse nature of these data streams. Despite this, using HPC resources for large-scale event-data analysis offers an alternative, particularly when dealing with extensive datasets intended for public release and benchmarking of neuromorphic algorithms. HPC systems provide the necessary computational power to efficiently process a vast number of event-data files, enabling the creation of larger datasets that can facilitate further research and development in neuromorphic computing. This approach can be especially valuable for researchers and institutions looking to share comprehensive datasets to support the evaluation of new algorithms in a reproducible and standardized way.

6. Conclusion

This project showed that leveraging HPC resources, specifically the Dardel supercomputer, offers substantial advantages over traditional methods, especially when handling extensive event-based recordings. By utilizing both shared memory parallelism with OpenMP and distributed memory parallelism with MPI, the pipeline effectively managed and processed large volumes of event data, achieving notable performance gains.

The results highlight that distributing tasks across multiple nodes reduces processing time, particularly for extensive datasets. The integration of GPUs further enhances performance, especially for tasks such as generating heatmaps, where GPU offloading proved to be highly beneficial. Conversely, for certain tasks like histograms, GPU acceleration did not provide additional benefits and, in some cases, even led to increased processing times due to memory management overhead. This insight underscores the importance of selecting appropriate computational resources based on the specific characteristics of the task at hand.

In conclusion, this project underscores the value of incorporating HPC resources into the analysis of event-based data. By combining traditional computational techniques with modern HPC capabilities, researchers can achieve more efficient processing and gain deeper insights into large-scale datasets, ultimately contributing to the advancement of neuromorphic computing and related fields.

7. References

GitHub Repository

Supplement

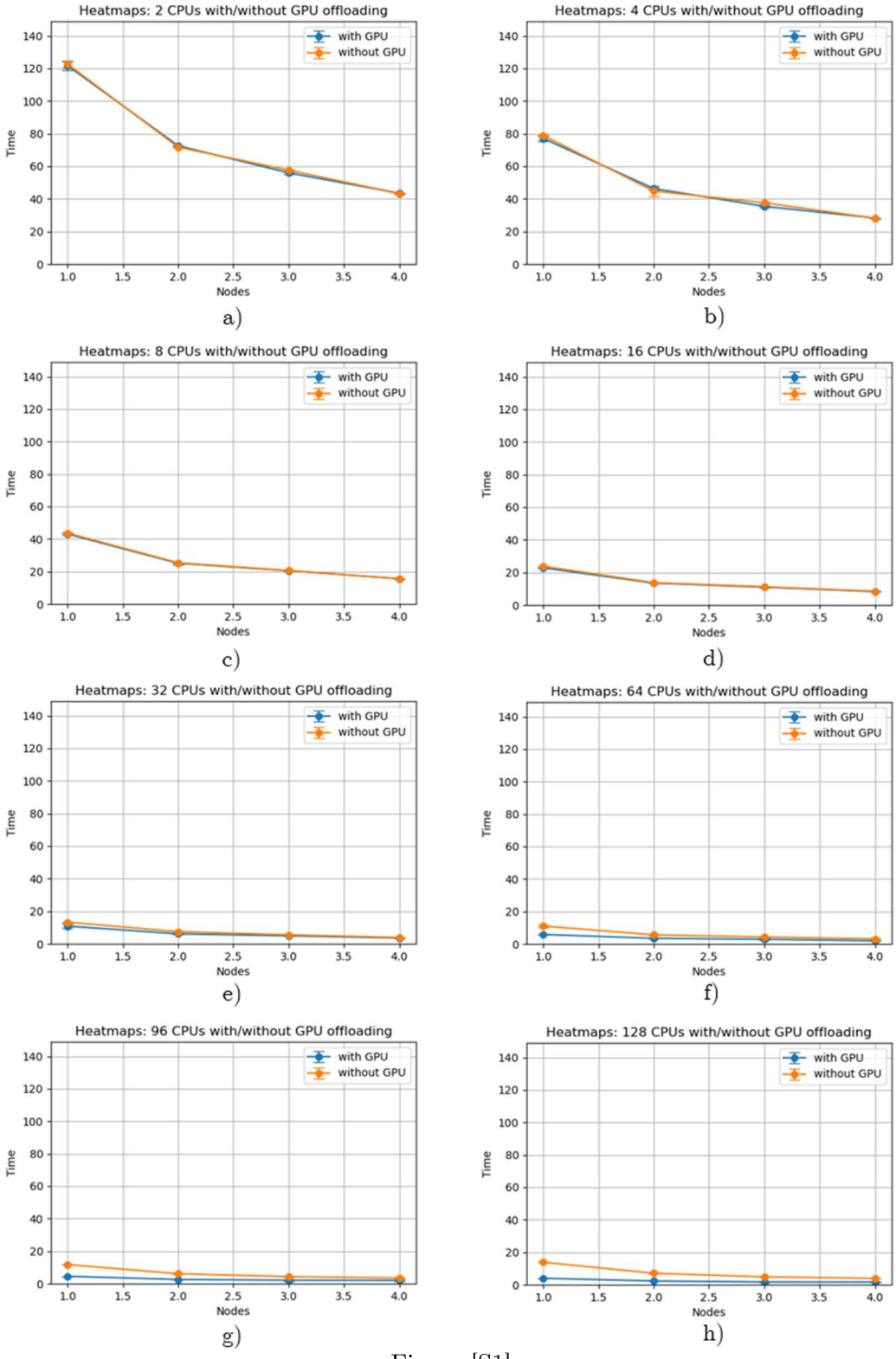


Figure [S1]

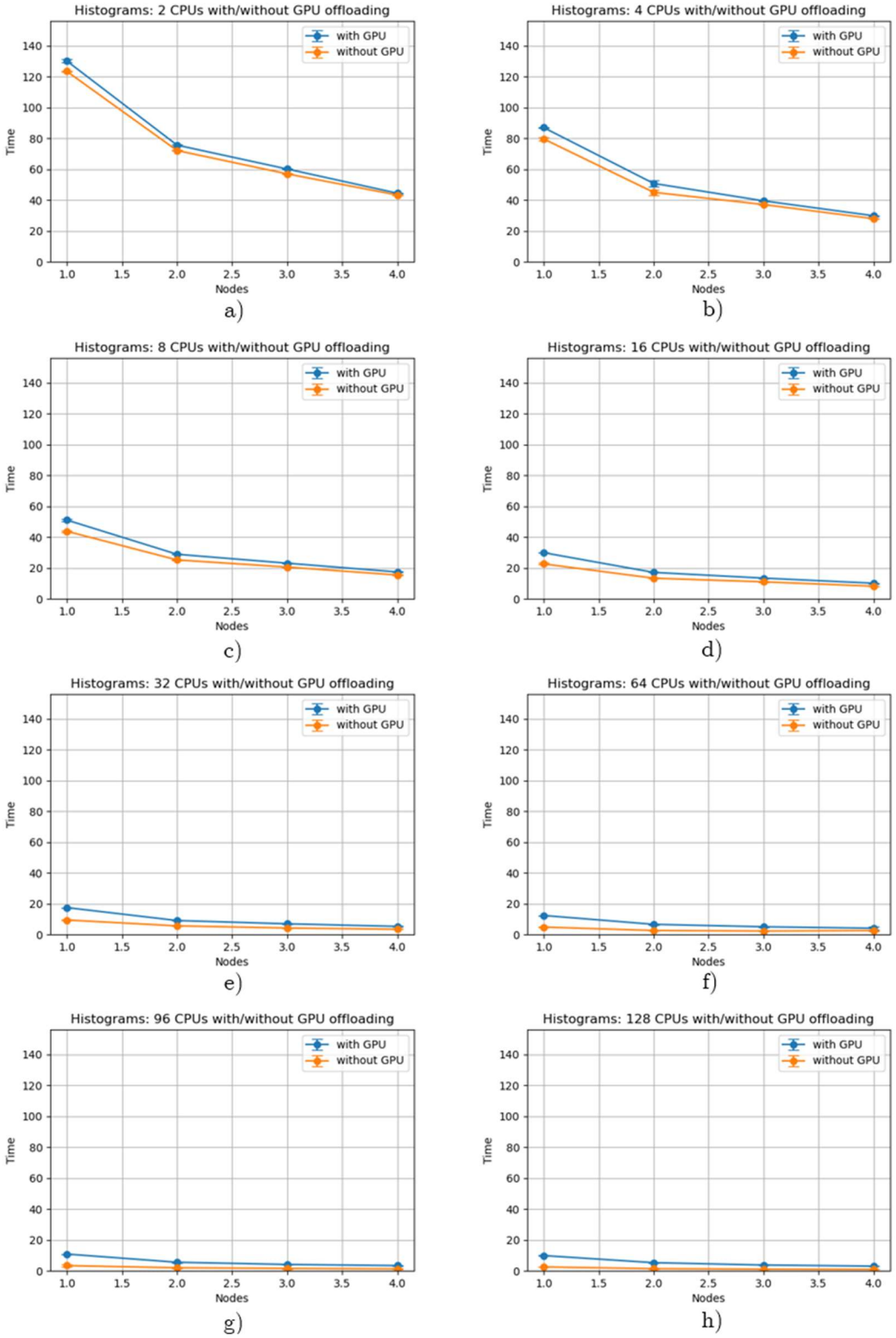

Histograms: 2 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
a)
Histograms: 4 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
b)
Histograms: 8 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
c)
Histograms: 16 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
d)
Histograms: 32 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
e)
Histograms: 64 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
f)
Histograms: 96 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
g)
Histograms: 128 CPUs with/without GPU offloading
with GPU
without GPU
Time
Nodes
h)

Figure [S2]

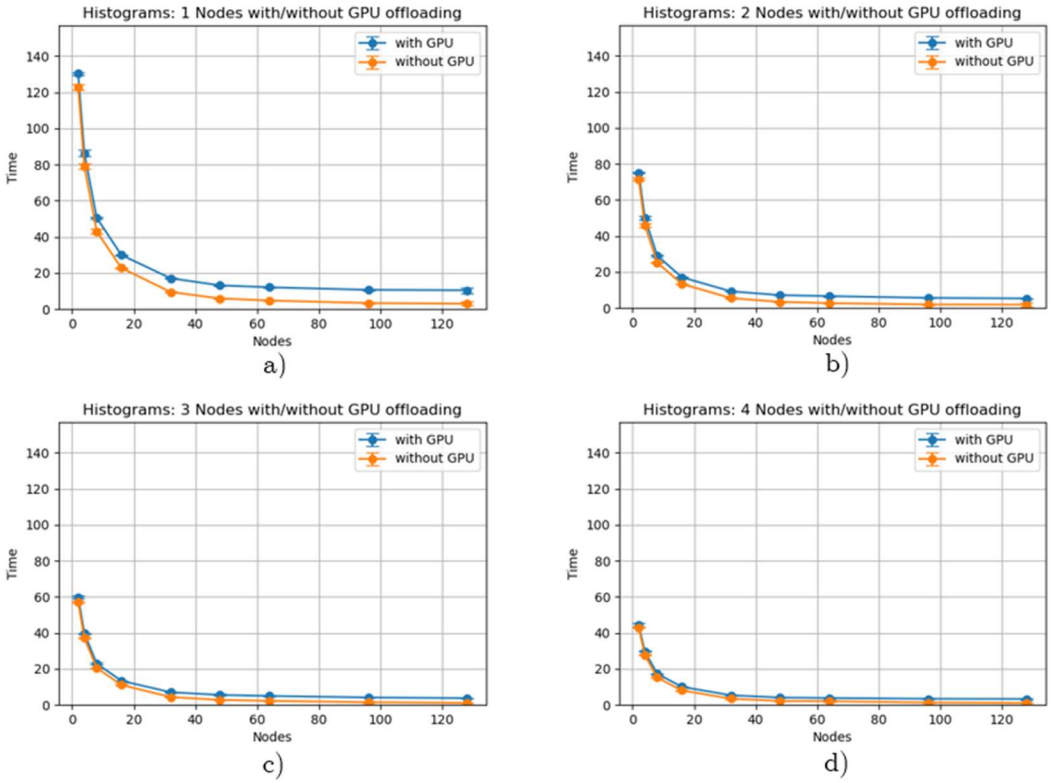


Figure [S3]

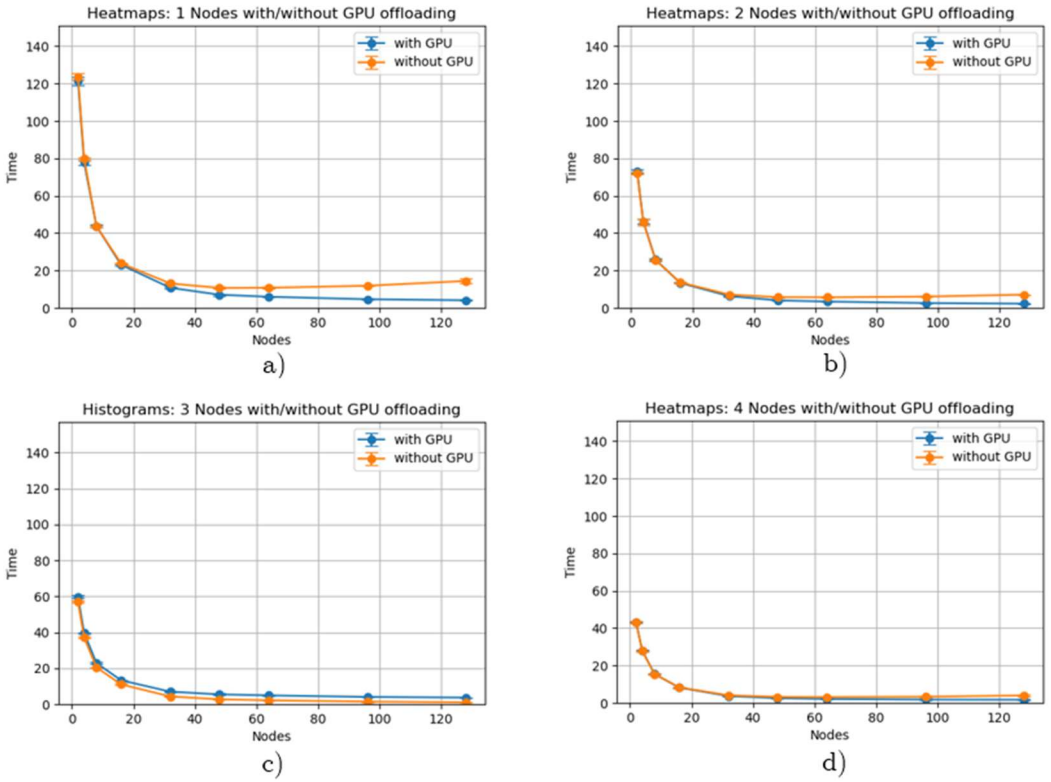


Figure [S4]