# BlueField Assisted Resiliency Framework

AURORA: Accelerated User-space Remote Offload for Resilient Applications

Jacob Chisholm
*Computer Engineering*
*CAESAR Labs, Queen's University*
Kingston, Ontario, Canada
chisholm.jacob@queensu.ca

Dr. Ian Karlin
*Computer Engineering*
*CAESAR Labs, Queen's University*
Kingston, Ontario, Canada
ian.karlin@queensu.ca

Dr. Ryan Grant
*Computer Engineering*
*CAESAR Labs, Queen's University*
Kingston, Ontario, Canada
reg1@queensu.ca

***Abstract*—Efficient checkpoint/restart mechanisms are critical for high-performance computing systems, where long-running applications must quickly recover from failures to maintain productivity and resource efficiency. However, all current checkpoint/restart mechanisms rely on the host CPU, introducing cache contention and operating-system jitter. To address this issue, we present a BlueField-Assisted Resiliency Framework, AURORA: Accelerated User-space Remote Offload for Resilient Applications. Unlike traditional methods that consume significant CPU resources, SmartNIC-assisted checkpoint/restart offloads IO-bound processes to the SmartNIC, allowing truly parallel operation between the host and its recovery mechanism. Through experimental microbenchmarks, we show that AURORA dramatically reduces host overhead, unlocking up to 10% faster runtimes for CPU-bound HPC workloads when compared to traditional host-based mechanisms.**

***Index Terms*—parallel-IO; checkpoint-restart; asynchronous-IO; NVIDIA BlueField; offloaded compute; immutable data**

## I. Introduction

Featuring thousands of compute nodes, systems such as Frontier, El Capitan, and others in the TOP500 list [1] have redefined the boundaries of typical computing. Unfortunately, as High-Performance Computing (HPC) and Artificial Intelligence (AI) transition to Exascale computing, hardware deployment has outpaced the inherent reliability of individual components. As early as 2012, the Titan supercomputer [2] at Oak Ridge National Laboratory reported a Mean Time Between Failure (MTBF) of only 44 hours [3]. More recently, the Llama-3 model experienced 419 unexpected failures during its 54-day training period on 16K GPUs [4]. Roughly, this equates to one failure every 4 hours. Given that unhandled failures can erase days or even weeks of work, system reliability and fault tolerance have been concerns in the HPC industry for many years [5] [6] .

One widely adopted solution to this challenge is Checkpoint-Restart (CR), which involves periodically saving an application's runtime state to stable (persistent) storage. In the event of hardware or software failure, the most recently saved checkpoint can be retrieved from stable storage and restored, eliminating the need for the program to restart from scratch. Early approaches to CR relied on *synchronous checkpointing*, where the application is paused until the entire checkpoint has been saved to the Parallel (persistent) File System (PFS). Because storage writes are IO-intensive, synchronous CR is largely bound by the speed of the stable storage medium, or PFS. To mitigate IO latencies, researchers developed *asynchronous checkpointing*. By performing an intermediate checkpoint to high-speed local storage and utilizing a background process to perform PFS writes, asynchronous CR greatly reduces the application blocking time.

However, when CR mechanisms rely on the host CPU for background processes and PFS writes, they introduce cache contention and Operating System (OS) jitter. As cluster sizes expand, the decrease in MTBF forces applications to checkpoint at significantly higher frequencies, amplifying existing host-side overheads and further degrading application performance. To mitigate this escalating overhead, checkpointing architectures must begin to explore alternative solutions beyond the host CPU, such as SmartNICs.

Smart Network Interface Cards, or SmartNICs, are an emerging technology designed to offload tasks from the host, reducing load while improving efficiency. Built upon traditional Network Interface Cards (NICs), SmartNICs are equipped with onboard processors and dedicated RAM. They run their own Operating System (OS) and can often be logically treated as independent nodes. Because of this distinction, independent workloads may be run on the SmartNIC and host CPU. During standard compute-bound workflows, SmartNIC cores typically sit idle, presenting an interesting opportunity for both failure detection and Checkpoint-Restore.

To alleviate the host-side resource contention in all current checkpoint-restart mechanisms, we present the first SmartNIC-driven checkpoint-restore framework. With the BlueField Assisted Resiliency Framework, we offload checkpointing persistence, reducing application runtime by up to 10% with AURORA: Accelerated User-space Remote Offload for Resilient Applications. AURORA decouples state persistence from host execution by offloading file creation, IO orchestration, and storage flushes to an NVIDIA BlueField-3 SmartNIC DPU. Operating over PCIe via one-sided Remote Direct Memory Access (RDMA) and performing temporary checkpoints in-memory, AURORA reduces the application checkpoint blocking time by an average of 90%.

SC26 Workshops, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-1221-5/26/$31.00 

The key contributions of this paper are summarized as follows:

1) **Reduction of Host Checkpointing Jitter:** By deferring file creation and storage serialization to the SmartNIC, host applications avoid costly POSIX system calls, kernel context switches, and cache pollution.
2) **SmartNIC-Driven Checkpoint Orchestration:** We design an asynchronous notification and memory shadowing model that allows the BlueField DPU to pull staged application memory directly over PCIe without host CPU intervention.
3) **Experimental Evaluation:** Conducted with synthetic and application benchmarks, we explore SmartNIC DPU utilization and demonstrate that offloading the checkpoint process dramatically improves the application runtime of CPU-bound workloads when compared to current State-Of-The-Art (SOTA) mechanisms.

## II. Related Work

Multi-level Checkpoint-Restart solutions have been extensively researched within the HPC industry.

One of the first production-ready implementations of Checkpoint-Restore was Berkeley Labs Linux Checkpoint-Restore, or BLCR [7]. BLCR operates at the kernel level, providing a fully transparent Checkpoint-Restart mechanism. Being fully transparent, BLCR can perform Checkpoint-Restore operations on any arbitrary application without modifications to the source code. Although highly robust for general-purpose workloads, its transparency introduces significant overhead; because BLCR lacks insight into application-level data structures, it must blindly capture all program memory, including temporary or redundant memory not necessary for the program's operation.

More recently, MANA (MPI-Agnostic Network-Agnostic) has addressed many of the challenges involved with transparent checkpointing in a distributed environment [8]. Built as a plugin for the Distributed Multi-Threaded Check Pointing (DMTCP) framework [9] [10], MANA utilizes a split-process approach that decouples the application logic from its underlying libraries, allowing MPI applications to be transparently checkpointed and restarted across different network architectures and MPI implementations. However, like BLCR, MANA remains a host-side solution that incurs additional overhead from limited awareness of application-level data structures.

Unlike the previously mentioned works, SCR was designed to be integrated into applications already using file-based Checkpoint-Restart schemes [5] [11]. By integrating at the application level, SCR eliminates memory overhead present in BLCR and MANA by only saving data crucial to the application's runtime. In this scheme, the SCR library is invoked by the application to perform a distributed, synchronous checkpoint. The SCR checkpoint process uses Parallel File System (PFS) and node-local storage, e.g., RAM disks and SSDs, to achieve aggregate bandwidth that scales nearly linearly with node count. Although this drastically improves performance over system-wide snapshots, synchronous checkpointing requires the application to stall while I/O operations complete. Therefore, while SCR improves checkpointing speed, application blocking time remains constrained by the cumulative I/O bandwidth of the cluster.

To address this issue, the Very Low Overhead Checkpointing (VeLOC) library implements asynchronous checkpointing through a split-process approach [12]. In this approach, the application is decoupled from the PFS by separating temporary from persistent checkpoints. In this model, the application, or "Client," performs temporary checkpoints to local, high-speed storage, typically a RAM disk or node-local SSD. Meanwhile, the split-process, or Active Backend, also referred to as the "Server," handles the data migration to the PFS and the creation of persistent checkpoints. This architecture minimizes the application's checkpointing time by only requiring the application to block during its initial, high-speed checkpoint. Furthermore, VeLOC's implementation extends the SCR Application Programming Interface (API) to support memory-based checkpointing. Rather than requiring application programmers to serialize memory regions into a file, the checkpointing layer manages these regions directly.

However, VeLOC's architecture features several critical limitations in the context of emerging SmartNIC hardware that prevent a simple migration of its backend:

1) **Host Compute & Cache Contention:** Utilizing the host CPU, VeLOC's active backend threads actively contend for CPU time and cache with the primary application. On fully subscribed compute nodes, this contention is significantly detrimental to application performance.
2) **POSIX Reliance & Overhead:** VeLOC relies heavily on traditional POSIX system calls for checkpoint-restart operations. These calls introduce significant operating system overhead and file system jitter, slowing the temporary checkpoint process and increasing application blocking time. Moreover, communication between the VeLOC client and backend relies on traditional, node-local file paths. This dependency makes it ultimately incompatible in an offloaded environment, as attempting to offload its server process to a SmartNIC would require file system synchronization between the SmartNIC and host, inadvertently exacerbating the very problem it seeks to solve.
3) **High-Overhead Host-Bound Communication:** VeLOC relies on Inter-Process Communication (IPC) or TCP for client-server coordination. In both contexts, host kernel intervention is required, generating auxiliary OS overhead.

Beyond the aforementioned works, a vast body of literature addresses specialized Checkpoint-Restart (CR) paradigms across various execution contexts including containers and hardware accelerators such as GPUs [13]. These specialized heterogeneous mechanisms fall outside the scope of this paper, which concentrates strictly on CPU-bound applications. However, extending our framework to interoperate with GPU state-capture utilities remains a promising avenue for future work.

## III. System Design

To address the limitations of host-centric fault tolerance, we propose a BlueField Assisted Resiliency Framework, AURORA: Accelerated User-space Remote Offload for Resilient Applications. The AURORA framework is built on a high-performance Client-Server model designed specifically for RDMA offload. Although optimized for NVIDIA BlueField (BF) SmartNICs, the design is architecture-agnostic, limited only by the capabilities of the UCX transport layer, a consideration that ensures that AURORA can be deployed in any RDMA-capable environment.

While AURORA maintains several conceptual similarities with established SOTA mechanisms to retain API compatibility, it fundamentally re-architects the communication, memory, and storage paradigms to enable true hardware offloading. Rather than relying on host-bound background threads and POSIX system calls, the framework performs all control and data movement via UCX (UCP) RDMA and Active Messages (AM).

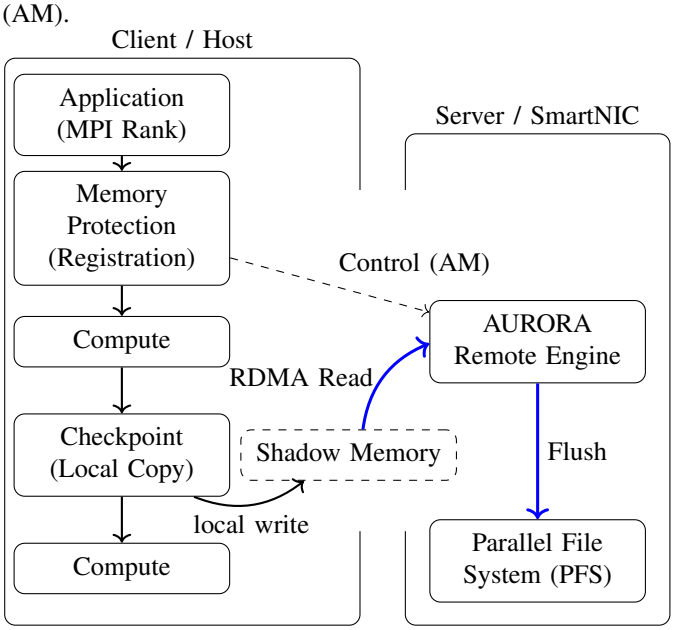


Fig. 1. System Architecture Overview

Shown in Figure 1, this configuration enables an offloaded-copy paradigm: The server pulls data directly from host memory into its own localized buffers before flushing to the PFS. This process occurs without host CPU intervention, utilizing idle SmartNIC cores to manage the complex I/O logic, metadata generation, and filesystem flushes.

In the following sections, we detail the core components:

- **Memory Region Management:** An application-aware shadow memory architecture that enables high-speed, local memory-to-memory copies, deferring file creation, and reducing application blocking time.
- **Offload Communication:** A Client-Server, RDMA-based notification and synchronization mechanism.
- **The Remote Engine:** Manages memory fetching, metadata generation, and PFS writes out-of-band, reducing host CPU contention and application runtime.

### A. Memory Region Management

AURORA utilizes an application-aware, in-memory checkpointing strategy. Unlike system-level tools that snapshot a complete image, AURORA operates on discrete, registered memory segments. This granularity allows the application to protect only the datasets critical to its state.

For every protected (active) region, the framework allocates a corresponding "shadow" region of identical size. Both the active and shadow regions are pinned in host RAM for RDMA operations. While active regions are application-managed, shadow regions are managed entirely within the C/R layer. The application never interacts with shadow regions directly; instead, they serve as a 1:1, version-locked, immutable copy of the active memory.

*1) Registration and Active Message Synchronization:* AURORA maintains a coherent view of these segments across the host-SmartNIC boundary using a Remote-Synchronized Region List. This synchronization is handled through UCP Active Messages (AM), which allow the client to update the server's state without requiring explicit, blocking server interactions. The registration flow follows a specific handshake:

1) **Local Allocation:** The client invokes a protection request, which allocates the shadow region and encapsulates both pointers into a local region structure.
2) **NIC Registration:** The local AURORA Region Manager (ARM) registers the active and shadow memory with the local NIC to generate the necessary RDMA keys (RKeys).
3) **Asynchronous Notification:** The client dispatches an AM to the server containing the region metadata: the unique ID, size, virtual addresses, and RDMA keys.
4) **Remote Integration:** The server receives the AM and registers the remote handles within its own NIC, establishing the one-sided RDMA handshake.

*2) Client-Side Lifecycle:* Because AURORA treats local checkpoints as simple memory-to-memory copies, it could leverage highly optimized transfer mechanisms. Although the current implementation utilizes a standard `memcpy`, future implementations may exploit hardware-accelerated DMA engines and/or dirty-page tracking. Such optimizations would further reduce the time the application is "engaged" in the C/R lifecycle, thus minimizing disturbance to the primary compute task.

**Checkpoint**: A checkpoint is triggered by performing a high-speed local copy from active to shadow memory. By isolating the checkpoint data in this manner, AURORA creates a stable, intermediate buffer that the SmartNIC can pull from at its own pace. Once this internal transfer has completed, host-side work is effectively finished, and the application is released to resume computation.

**Restore**: Conversely, a restore operation involves copying data directly from persistent storage back into the application's active regions. Currently, the client blocks during this phase to ensure a consistent state before resumption. However, AURORA supports future "background-" or "hybrid-" restore

optimizations, allowing the application to continue unrelated work or assist in the restoration process while the SmartNIC repopulates a secondary memory space.

To ensure data integrity, memory de-registration is currently synchronized with the server to prevent the destruction of memory that may be the target of an in-flight RDMA pull. However, the architecture can naturally support fully asynchronous de-registration. By maintaining the shadow region's RKeys until the server completes its current cycle, the application could free its active pointers and destroy local keys immediately, leaving the C/R layer to handle the final teardown of the shadow region once the remote engine signals completion.

### B. SmartNIC Communication & Offloading

To ensure coherence in the global system state, AURORA uses asynchronous completion-based notifications facilitated by RDMA exchanges of a cache-aligned memory structure maintained by both the client and the server. This design is inspired by a Bi-Directional Rx/Tx Memory-Mapped I/O interface, where RX (read-only) and TX (write-only) sets of registers are shared via one-sided RDMA. This allows either side to observe the other's state through monotonic counters, or "ticks," without engaging the remote CPU.

*1) Synchronization Model:* When a participant completes an operation, it increments a specific monotonic counter. Shown in Figure 2, remote states are retrieved via a one-sided RDMA. The resulting difference between the local and remote states represents the "Ahead-Behind" value. The local value exceeding the remote value indicates that the remote is behind the local state. Similarly, the opposite indicates that the remote state is ahead of the local state. Therefore, the difference in states represents the item-specific delta.

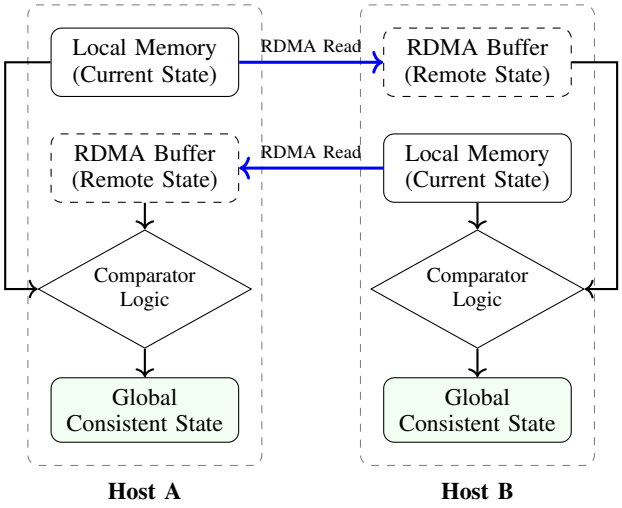


Fig. 2. Asynchronous RDMA Notifications

For checkpoint operations to be initiated from the host, the checkpoint state must first be synchronized due to the 1:1 shadow-region mapping. Once synchronized, the host can increment its checkpointing state, triggering a remote operation. After the persistent flush to the PFS is completed, the remote engine increments its own counter to synchronize the system.

Shown in Figure 1, this model decouples intent to act from action, allowing the host to signal a request and immediately resume computation. Consequently, global MPI synchronization and explicit "checkpoint start" signals are eliminated.

### C. Remote Engine

The Remote Engine serves as the dedicated checkpoint/restart server running on the BF3 SmartNIC. It is responsible for monitoring client states, managing resource allocations, and executing the heavy I/O operations required for persistent storage. By handling these tasks outside the main program's execution environment, the Remote Engine effectively reduces host-side resource contention and OS jitter during the checkpoint lifecycle.

*1) Polling and Resource Management:* To maintain responsiveness while minimizing energy usage, the engine uses a single dispatcher with a polling queue and thread pool. The engine continuously executes a polling loop to identify and act on pending requests:

1) **Status Verification:** The engine pops an instance and performs a one-sided RDMA read of the host's ACN. If the read times out or hits an error threshold, the instance is flagged and requeued.
2) **Task Identification:** Using a *test* method, the engine checks for pending operations. If a delta exists, it attempts to allocate a worker thread and cache buffers; if resources are saturated, the instance is requeued to prevent deadlocks.
3) **Failure Detection:** Instances with high repeated failures are marked as failed and disconnected from the server.

*2) Offloaded I/O Execution:* The engine executes the data movement for the system as follows:

**Checkpointing:** Following the process illustrated in Figure 1, the checkpointing process is initiated by a client completing its local checkpoint. After local completion, the engine performs repeated RDMA reads from the host's shadow regions into DPU cache buffers before flushing to the PFS. This cycle repeats until all regions and metadata have been saved to persistent storage, at which point the engine ticks the client's ACN to signal completion. Backend Checkpoint failures trigger an automatic requeue for retry without signaling the client.

**Restoration:** During restoration, the engine reciprocates this logic by pulling data from the PFS and performing RDMA writes directly into the application's active regions. In the current implementation, the host CPU blocks during the restore to ensure consistency, while the BlueField SmartNIC manages IO bottlenecks and data injection. However, future implementations support the possibility of performing "background-restore" operations, allowing the host to perform unrelated work while the SmartNIC repopulates a secondary memory space.

## IV. Evaluation

This section evaluates the performance and scalability of the AURORA framework. We focus on measuring application-level blocking time and assessing the impact of SmartNIC resource allocation on checkpoint/restart (C/R) efficiency of CPU-bound workloads. In all benchmarks, VeloC is used as the reference implementation due to the major similarities in its architecture.

### A. Experimental Setup

Benchmarks were performed on the HPC Advisory Council ROME Cluster. Each node features a dual-socket AMD EPYC 7742 with 512GB DDR4-2666 RAM. High-performance storage is provided via a Lustre parallel file system. For offloading, each node is equipped with a 16-core NVIDIA BlueField-3 SmartNIC DPU with 32 GB of HBM memory. All tests were run on AURORA `v0.0.3` and VeloC commit `37abb3c` on a single node to isolate differences in framework performance from inter-node network latency and collective PFS contention. All results were collected over an average of 10 trials.

### B. Thread Count Optimization

A critical factor in SmartNIC DPU offloading is the allocation of ARM compute cores. Using a synthetic blocking checkpoint-restore benchmark, we evaluate the baseline performance of AURORA without application interference, utilizing 8, 16, and 32 SmartNIC threads and 128 MPI ranks. We select a total checkpoint size of 16GB (125MB per MPI process) to prevent compounding latency overheads. Additionally, barriers are used before and after checkpointing phases to synchronize all ranks to hit the server simultaneously, thereby maximizing the SmartNIC (backend) load.

*1) Application Checkpoint Latency:* The primary metric for CR frameworks is the application blocking time. Shown in Table I and Figure 3, the results captured in the 16GB test show no differences between 8, 16, and 32 SmartNIC threads. Therefore, we lower the checkpoint size from 16GB (128MB per MPI rank) to 2GB (16MB per MPI rank) to expose hidden latencies, shown in Table II and Figure 4. Given that no differences appear between I and II, we conclude that the application blocking time is not determined by SmartNIC cores, as suggested by our RDMA-based communication model.

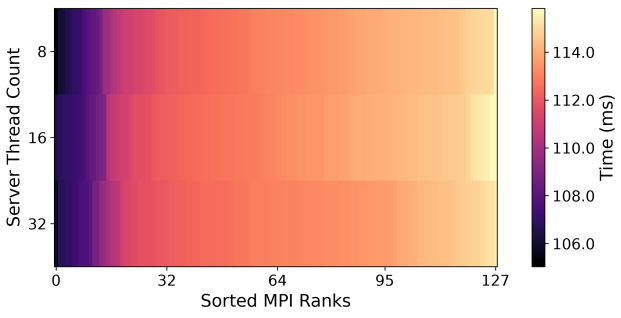


Fig. 3. Application Blocking Time, 16GB

TABLE I
Application Blocking Time, 16GB Checkpoint

| SmartNIC Threads | Application Blocking Time (ms) | | | |
|---|---|---|---|---|
| | ***Min*** | ***Mean*** | ***Median*** | ***Max*** |
| 8 Threads | 99.14 | 112.49 | 113.02 | 118.31 |
| 16 Threads | 100.16 | 112.67 | 113.11 | 118.98 |
| 32 Threads | 100.19 | 112.49 | 112.91 | 118.31 |

TABLE II
Application Blocking Time, 2GB Checkpoint

| SmartNIC Threads | Application Blocking Time (ms) | | | |
|---|---|---|---|---|
| | ***Min*** | ***Mean*** | ***Median*** | ***Max*** |
| 8 Threads | 10.54 | 18.17 | 18.41 | 23.75 |
| 16 Threads | 10.43 | 18.23 | 18.43 | 25.27 |
| 32 Threads | 10.53 | 18.12 | 17.95 | 23.93 |

*2) Checkpoint Execution and Flush:* Beyond initialization, we measure the total duration to commit a complete checkpoint to the PFS. The following metric is captured using the host's wall clock, tracking from checkpoint initialization to receiving the completion notification from the SmartNIC DPU. Due to the shadow region mapping described in Section III-A, this metric also describes the minimum permissible interval between consecutive checkpoint operations.

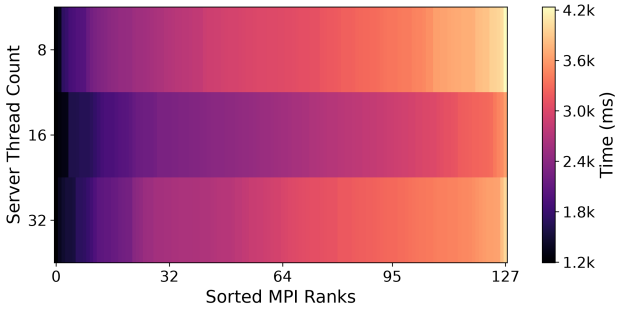


Fig. 4. Total Checkpoint Time, 16GB

Shown in Table III, using 8 threads on the 16-core BlueField yields a wide distribution of checkpoint times with extreme tail latencies. This is likely a result of SmartNIC DPU underutilization combined with MPI rank oversubscription. Conversely, scaling the SmartNIC's thread allocation to 16 and 32 threads establishes a uniform distribution of checkpoint times.

TABLE III
Total Checkpoint Ex. & Flush Time, 16GB Checkpoint

| SmartNIC Threads | Total Checkpoint Time (ms) | | | |
|---|---|---|---|---|
| | ***Min*** | ***Mean*** | ***Median*** | ***Max*** |
| 8 Threads | 292.56 | 3005.42 | 3048.63 | 7712.32 |
| 16 Threads | 340.33 | 2534.23 | 2622.63 | 5439.52 |
| 32 Threads | 392.86 | 2854.24 | 2885.48 | 5059.38 |

*3) Restart Performance:* Finally, we measure the time to complete a restore operation, displayed in Figure 5. Established in Section III-C2, restart operations are completed on the SmartNIC DPU while the host remains blocked. Thus, the following time is measured from the host's wall clock.

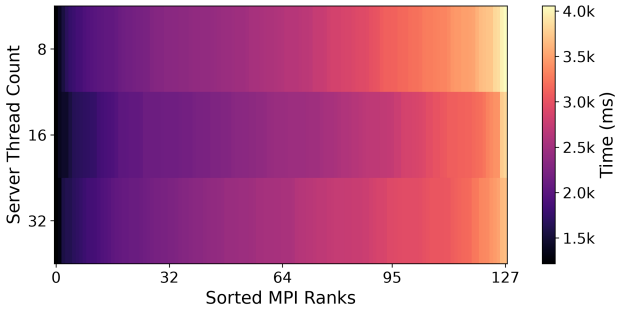


Fig. 5. Total Restart Time, 16GB

Shown in Table IV, restart follows a trend similar to the previous two tests, with the 16-thread configuration displaying slightly lower mean and median runtimes.

TABLE IV
TOTAL RESTART TIME, 16GB CHECKPOINT

| SmartNIC Threads | Total Restart Time (ms) | | | |
|---|---|---|---|---|
| | *Min* | *Mean* | *Median* | *Max* |
| 8 Threads | 60.93 | 2686.92 | 2572.46 | 6304.97 |
| 16 Threads | 69.83 | 2455.27 | 2361.08 | 5512.76 |
| 32 Threads | 75.70 | 2554.43 | 2317.68 | 5359.70 |

In all three evaluations, the 8-thread configuration underutilized the SmartNIC DPU resources, resulting in high mean, median, and tail latencies. Meanwhile, the 16- and 32-thread configurations showed similar results. However, the 32-thread configuration experienced heightened mean and median times across Tables II and III. Therefore, we deduce the optimum SmartNIC DPU thread count to be 16 for further tests.

### C. Heat Distribution Application

Utilizing a Heat Distribution mini-application from the VeLOC test suite, we evaluate AURORA compared to the current state-of-the-art mechanism: VeLOC. Baseline results (runtime without checkpoint-restore) are also collected. All reported benchmark times exclude initial application startup and static memory allocation times, focusing exclusively on the active computation and application blocking phases. To investigate the effects of host resource starvation, we evaluate both frameworks under 64-core and 128-core tests. To ensure identical testing conditions, we use VeLOC's Hybrid-Opt checkpointing strategy, which utilizes on-node cache to accelerate the application blocking checkpoint phase.

*1) Total Runtime:* We first evaluate the execution time after a fixed number of iterations. We perform exactly three checkpoints during each run, triggered at compute iterations 2, 4, and 6.

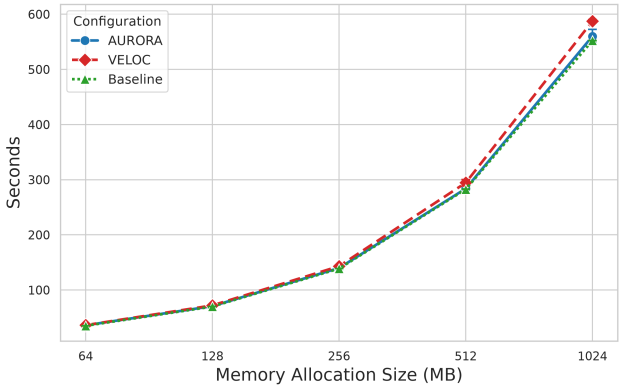


Fig. 6. Total Heat Distribution Run Time - 64 Cores

Shown in Figure 6, both checkpoint-restart mechanisms achieve total runtimes within 3% below 512MB during the 64-core evaluation. This is likely due to the 128-core node providing ample host-side "headroom" for background system tasks, such as the VeLOC backend. By absorbing host-side CPU contention, this test isolates the baseline efficiency of each framework. Consequently, the 4 and 5% performance differences observed at 512 and 1024 MB can be attributed to the differences between traditional POSIX-file-based flushes and AURORA's local memory copy mechanism.

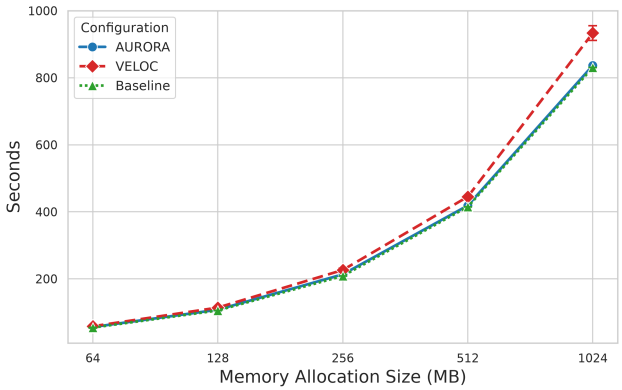


Fig. 7. Total Heat Distribution Run Time - 128 Cores

In the 128-core benchmark, displayed in Figure 7, clear differences in runtime emerge as the memory footprint scales. When the application utilizes all physical cores, no idle host cores remain, and VeLOC's background serialization and compression threads must actively contend with application threads for hardware resources. This resource starvation introduces severe operating system (OS) jitter and degrades cache locality, resulting in 6 and 10% runtime difference between VeLOC and AURORA at 256 and 1024MB.

*2) Checkpointing:* Using the traces captured in Section IV-C1, we record the average application-reported time spent in the checkpointed phase, displayed in Figures 8 and 9.

Shown in Figure 8, AURORA reduced application blocking time by an average of 88% up to 1GB during the 64-core evaluation. As previously discussed, performance differences

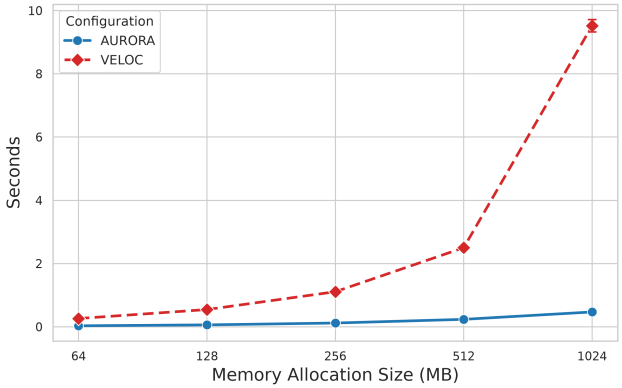


Fig. 8. Total Application Blocking Time - 64 Cores

observed in the 64-core evaluation can be attributed to the differences between traditional POSIX-file-based checkpointing and AURORA's local memory copy. Thus, the increased scaling factor observed in the VeLOC benchmark is likely due to its reliance on the POSIX file-system layer for intermediate checkpoints. Because AURORA offloads all POSIX file-system operations to its server process, it does not experience the heavy metadata lock contention and file-system serialization present in VeLOC.

At 1GB, VeLOC experiences a dramatic increase in runtime, resulting in a 95% performance difference, pointing to heavy file-system serialization overhead inherent to VeLOC's POSIX-based checkpointing layer. Conversely, AURORA displays a linear increase in runtime across all checkpoint sizes. As AURORA's checkpointing phase avoids system calls, it does not incur the same POSIX latency.

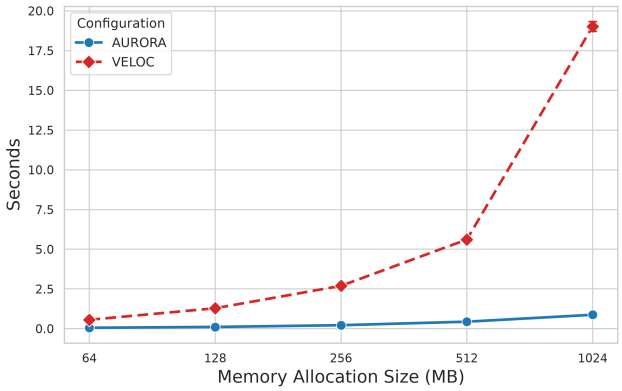


Fig. 9. Total Application Blocking Time - 128 Cores

Figure 9 highlights these further differences at 128 cores, showing an average and peak performance difference of 94 and 96%. When all physical cores are fully utilized, the VeLOC backend process is severely starved of CPU resources. Lacking core headroom, VeLOC's background threads are forced to context-switch with active computation ranks, exacerbating the issues from the 64-core test. Moreover, because all cores are utilized, additional communication latencies are introduced during synchronous operations.

*3) Restart:* After the test described in Section IV-C1, a second run was performed to evaluate the performance of the restart mechanism.

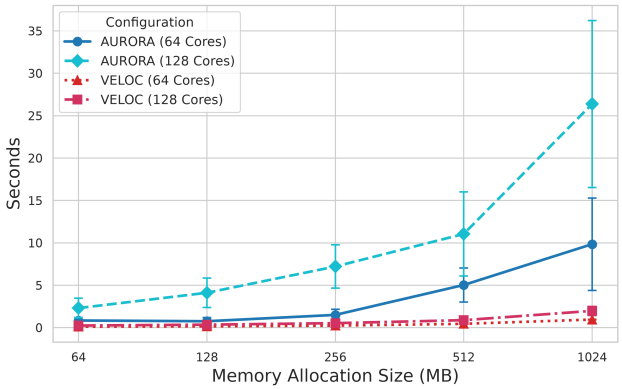


Fig. 10. Total Heat Distribution Restart Run Time

Shown in Figure 10, AURORA exhibits a significantly longer total restart runtime compared to VeLOC. Under VeLOC, the restart operation is executed by the host CPU, allowing the framework to harness the full memory bus of the host node. However, within AURORA, the restart operation is performed on the DPU while the host blocks. Thus, the restart speed is limited by the memory bandwidth of the BlueField-3 DPU. Alternatively, AURORA could take advantage of future "background-" or "hybrid-" restore optimizations, allowing the host CPU to continue unrelated work or assist in the restoration process. Such paradigms are not explored within this paper; however, in this case, the speed of the restore operation is expected to be similar to that demonstrated by VeLOC.

## V. Discussion

Through experimental evaluation, AURORA demonstrates that offloading checkpoint operations to a SmartNIC DPU fundamentally alters the host-side cost structure of fault tolerance. However, while AURORA delivered up to a 10% total application speedup, its performance benefits are strictly domain-specific:

- **Target Workloads**: AURORA is specifically designed for host-CPU-bound applications where compute ranks fully subscribe to physical cores. On systems with spare host CPU headroom, AURORA offers marginal speedups resulting from its RAM-heavy initial checkpoint strategy.
- **Memory Footprint Penalty**: To achieve sub-second application blocking times, AURORA allocates a 1:1 version-locked shadow region for every active application buffer. This effectively doubles the application memory footprint, requiring developers to trade node RAM capacity for compute throughput.
- **SmartNIC DPU/RDMA Reliance**: Although AURORA is designed to be architecture-agnostic, clusters without SmartNIC DPUs may experience degraded application performance from network overhead. Additionally, AURORA relies heavily on RDMA transactions, constraining its use to environments that support RDMA over UCP.

### A. Checkpoint-Restart Performance

While offloading the restart phase to the DPU incurs higher latencies than host CPU mechanisms, the penalty remains a minor concern due to the frequency of restart events. In real-world deployments, checkpoints occur at dozens, or even hundreds, of times the rate of restarts. Thus, trading restart performance for checkpoint speed yields a net benefit for long-running applications as shown in Daly's Model. Furthermore, as previously mentioned, this bottleneck can be overcome by exploring "hybrid" checkpoint-restart mechanisms capable of choosing the lowest-cost restoration path. Such paradigms, while theoretically possible, are not explored in this paper, presenting an interesting opportunity for future work.

### B. Daly Model Comparison

To evaluate real-world impact, we analyze the checkpointing frequency using Daly's optimal interval model, shown in Equation 1 [14]. The MTBF is chosen to be 4 hours (14400 seconds), matching empirical failure rates from the training of Llama-3 [4].

$$T_{\text{ckpt,opt}} = \sqrt{2(t_{\text{MTBF}} + t_{\text{restore}}) \times t_{\text{ckpt}}} \tag{1}$$

With a 1 GB memory footprint and 64-core saturation, VeLOC averages a checkpoint time of approximately 9.5 seconds and a restore time of approximately 0.9 seconds. In this configuration, Daly's model yields an optimal checkpointing frequency of 523 seconds. With a checkpoint time of approximately 470 ms and a restore time of approximately 9.8 seconds, AURORA lowers the interval to 116. Under 128-core saturation, VeLOC's host overhead penalty increases its optimum interval to 740 seconds, whereas AURORA maintains 160 seconds despite a longer restart time. Assuming no restart penalty, AURORA's optimal interval would decrease by approximately 100ms (0.06%), suggesting that improvements to restart runtime would yield minimal overall impact.

Because AURORA decreases application blocking time by up to 95%, applications can theoretically be checkpointed up to 4.5 times faster. Assuming ideal (instant) backend storage flushes, the expected work lost per failure event is roughly half the optimum interval. Therefore, due to AURORA's 4.5x higher checkpointing frequency, it reduces the expected compute lost per failure by 78%.

## VI. Conclusion

In this paper, we demonstrate an asynchronous checkpointing strategy that offloads all POSIX and PFS-related operations from the host CPU to a SmartNIC DPU. Through an evaluation conducted with a synthetic benchmark, we revealed that pinning the server to 16 threads optimizes throughput and minimizes tail latency. We then demonstrated that AURORA successfully offloads checkpoint overhead to NVIDIA BlueField-3 DPUs with VeLOC's heat distribution micro-benchmark, reducing application blocking time by up to 95% and total application runtime by up to 10%. Finally, we discussed the limitations of the framework, such as its memory footprint, and compared its results using Daly's model. In conclusion, this paper demonstrates that there exists a clear path for future architectural refinements in DPU-accelerated resiliency.


## Acknowledgment

This work would not have been possible without their contributions or the support of the HPC-AI Advisory Council, which generously provided access to the NVIDIA BlueField devices used for the evaluation section of this work. Additionally, the authors would like to thank The Oracle for providing expertise, perspective, and inspiration.


## References


[1] "TOP500 Supercomputer Sites," Nov. 2025. [Online]. Available: https://top500.org/lists/top500/2025/11/

[2] "Titan: Oak Ridge National Laboratory." [Online]. Available: https://top500.org/resources/top-systems/titan-oak-ridge-national-laboratory/

[3] D. Tiwari, S. Gupta, J. Rogers *et al.*, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 331–342, iSSN: 2378-203X. [Online]. Available: https://ieeexplore.ieee.org/document/7056044

[4] A. Grattafiori, A. Dubey, A. Pandey *et al.*, "The Llama 3 Herd of Models," Nov. 2024, arXiv:2407.21783 [cs]. [Online]. Available: http://arxiv.org/abs/2407.21783

[5] N. Vaidya, "A case for two-level recovery schemes," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 656–666, Jun. 1998. [Online]. Available: https://ieeexplore.ieee.org/document/689645

[6] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, Sep. 2013. [Online]. Available: https://doi.org/10.1007/s11227-013-0884-0

[7] J. Duell, "The design and implementation of Berkeley Lab's linuxcheckpoint/restart," Tech. Rep. LBNL–54941, 891617, Apr. 2005. [Online]. Available: http://www.osti.gov/servlets/purl/891617-2L2UJc/

[8] R. Garg, G. Price, and G. Cooperman, "MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 49–60. [Online]. Available: https://dl.acm.org/doi/10.1145/3307681.3325962

[9] "DMTCP : Distributed MultiThreaded Checkpointing." [Online]. Available: https://dmtcp.sourceforge.io/

[10] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. Rome: IEEE, May 2009, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/5161063/

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11, iSSN: 2167-4337. [Online]. Available: https://ieeexplore.ieee.org/document/5645453

[12] B. Nicolae, A. Moody, E. Gonsiorowski *et al.*, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 911–920, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/8821049

[13] R. Stoyanov, V. Spišaková, J. Ramos *et al.*, "CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads," Feb. 2025. [Online]. Available: https://arxiv.org/abs/2502.16631v1

[14] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, Feb. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X04002213