Multi-Threading and GPU Acceleration: The Case for Faster Simulators for Maritime Search and Rescue Operations Planning

Adam Boukhari
Department of Computer Science and Software Engineering
Université Laval, QC, Canada
adam.boukhari.1@ulaval.ca

Michael Morin
Department of Operations and Decision Systems
Université Laval, QC, Canada
michael.morin@osd.ulaval.ca

ABSTRACT

In decision support systems (DSSs) for maritime search and rescue (MSAR), simulation can be used to calculate the probability of success of a search operation tasking aircraft and vessels to search patterns. Nonetheless, single-threaded simulators remain a bottleneck in DSSs for MSAR, where thousands of candidate search operations generated by an optimizer have to be evaluated within a few minutes. Slow simulations also influence the design of DSSs when slight manual corrections to an operation require waiting for reevaluation. In response to the need for faster simulators, we redesigned the simulation pipeline and developed two parallel simulators: one based on CPU multi-threading and one based on GPU acceleration. Experiments show that CPU multi-threading provides a 6.7-fold acceleration over the single-threaded baseline, whereas the GPU acceleration yields a 260-fold speedup. Such speedups can render DSSs more responsive and enable optimizers to provide better recommendations, ultimately leading to more effective MSAR operations.

Keywords

Maritime search and rescue, Simulation, Parallelism, Optimization, Decision support system

INTRODUCTION

Although not all distress calls lead to a search operation, every second counts when someone's life is at risk at sea. As a result, the response time of decision support systems (DSSs) used for planning MSAR operations must be as short as possible. This is especially true when the location of potential survivors is unknown. In these cases, SAR mission coordinators establish a search operation tasking each available aircraft or vessel—referred to as search and rescue units (SRUs)—to different search trajectories (also called search patterns) in order to find the search object, e.g., a drifting life raft or person in water.

DSSs for MSAR can assist coordination centers in tasking the available SRUs to search patterns. Some systems employ algorithms to find an allocation of the SRUs with the highest possible probability of success (POS) under operational constraints, e.g., SAR Optimizer (Abi-Zeid et al. 2019). In recent systems such as the Search and Rescue Planning System (SAROPS) (Kratzke et al. 2010) and the new iteration of the Canadian Search and Rescue Planning tool (CANSARP) (Hillier 2008) with SAR Optimizer (Abi-Zeid et al. 2019), the calculation of the POS can be based on search simulations. A search simulator uses the potential trajectories of the drifting object and the SRUs' search patterns to estimate the likelihood of the object being detected. Because search simulators compute the POS of an operation, they can be used as one of the main building blocks of an optimization module responsible for finding the best possible recommendation for tasking SRUs. A heuristic could, for example, enumerate candidate search operations (tasking SRUs to patterns) and retain the best one after evaluating their POS (Abi-Zeid et al. 2019). Or a hybrid solver could use the search simulator as an external function for POS evaluation (Esmaeilpour et al. 2025).

Although examples of DSSs for MSAR using search simulation along with optimization are now operational, we observed that simulations tend to consume much of the optimization time. This is partly due to the fact that the optimization algorithms simulate many candidate operations in order to find one with the highest possible POS.

One possible approach to reduce the total simulation time would be to evaluate fewer candidate operations, but it would likely lead to a recommendation with a lower POS. Another approach could be to use a simpler, but faster, simulator, but this too might impact the quality of the recommendation if the simpler simulator is inaccurate.

As a result, search simulation speed remains one of the main roadblocks to the development of better DSSs for MSAR. In fact, if we are to use search simulation as a component of an optimization module, we need faster simulators that will retain the accuracy and reliability of the current simulators.

We therefore propose multi-threading and GPU acceleration to shorten the search simulation time. We show that, due to the properties of search simulations, multi-threading and GPU acceleration allow for a seemingly instantaneous response to users' input (under 400 milliseconds) when evaluating search plans. While our single-threaded baseline struggles to process more than one SRU within 400 milliseconds, the GPU-accelerated version completes the same task in only a few milliseconds.

Beyond our specific use case, we identify the key conditions necessary to replicate these performance gains in other simulation-based optimization scenarios.

BACKGROUND ON DECISION SUPPORT SYSTEMS FOR MARITIME SEARCH AND RESCUE

SAROPS (Kratzke et al. 2010) and CANSARP (Hillier 2008) are two of the recent operational MSAR DSSs that can provide recommendations to SAR mission coordinators. To simplify the presentation of both systems, we use the term *search operation* (or operation) to designate an assignment of SRUs to search patterns representing their search trajectories. We also assume the main figure of merit of a search operation is the POS.

SAROPS is currently used by the United States Coast Guard to plan MSAR operations. As described by Kratzke et al. 2010, SAROPS has three main components. An *environmental data server* (EDS) that collects data such as wind, weather, and wave heights. A *simulator* that takes as input the EDS data to calculate plausible drift trajectories for the search object and that tracks detections performed by the SRUs along their search pattern. A *planner* that searches for an operation with the highest possible POS.

CANSARP (also called ASPT) is the system of the Canadian Coast Guard (Hillier 2008). CANSARP integrates drift simulation, MSAR case management, and resource allocation management. Its search planning module, called SAR Optimizer (or Search Planner), integrates search simulation and optimization (Abi-Zeid et al. 2019). SAR Optimizer has multiple software components, including a simulator, an evaluator, and an optimizer. The *search simulator* takes as input a drift simulation and simulates the search patterns of the SRUs tasked in a search operation to compute the detections. The *evaluator* computes the POS of the search operation based on the simulated SRUs and object trajectories. The *optimizer* searches for an operation with the highest possible POS.

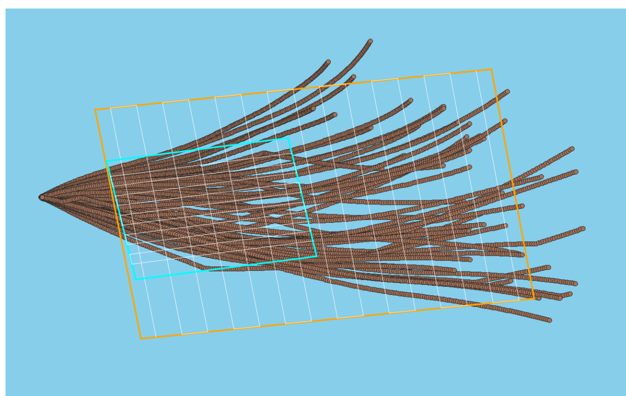

Figure 1. An operation over a drift where two SRUs are tasked with parallel search patterns; this figure is from the authors' system, the visuals from other DSSs for MSAR might differ

Figure 1 shows the particles of a drift and an operation with two SRUs. In SAR Optimizer, the drift is an input to search simulation and optimization. It consists of a set of simulated trajectories for the object. These trajectories are called *particles*. Each particle represents one plausible set of positions given the object's specifications and the environmental conditions, e.g., wind and current. On the figure, the SRUs' search patterns are enclosed by rectangles (blue and orange). During a search simulation, each SRU follows its search pattern. The patterns are divided into (longer) search legs linked by (shorter) cross legs.

As can be seen from our description, both SAROPS and CANSARP/SAR Optimizer share similar software components; they use similar data (e.g., the drift), perform simulation, and can optimize an operation. Given that both systems are operational, we consider their components as standards in the current DSSs for MSAR.

Search Simulation: Calculating the Probability of Success

The goal of the POS is to act as a figure of merit for candidate search operations. In the absence of other criteria, between two candidate operations, one selects the operation with the highest POS (Kratzke et al. 2010; Abi-Zeid et al. 2019; Laperrière-Robillard et al. 2022). POS evaluation uses a Monte Carlo approach that accumulates non-detections over particles in time. We developed our own search simulators based on the descriptions of Abi-Zeid et al. 2019 and Laperrière-Robillard et al. 2022, which can be summarized as follows.

Let U be the set of patterns assigned to SRUs in the evaluated operation, each with a set K of search legs. Let P be the set of particles representing the drifting object. During the simulations, the positions of both the SRU u and the set P are sampled every X seconds (typically, $X = 300$ seconds). The set P is called the *drift*. We call each particle position a *point*. Given the drift, the search simulator first determines the failure probability of SRU u to detect a particle p. Given d_k, the distance of the closest point of particle p to leg k, the probability $P_{\text{fail}}(p, u)$ that SRU u fails to detect p across all legs is:

$$P_{\text{fail}}(p, u) = \prod_{k \in K} (1 - LRC(d_k)) , \tag{1}$$

where $LRC(.)$ is the lateral range curve of the sensor used for searching, e.g., a visual search.

The LRC estimates the probability that SRU u detects the object based on the perpendicular distance at the closest point of approach d_k, assuming both are stationary and an infinite search leg, as illustrated in Figure 2, which shows a fictitious LRC function. In practice, LRCs are based on real-life observational data that allows modeling detection performance under different conditions and are described in search theory (Stone et al. 2016). An LRC shape is determined by parameters, e.g., object type, SRU type, visibility, and wave height.

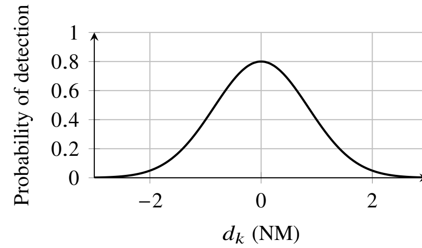

Figure 2. Example of a lateral range curve (LRC) modeling detection probability as a function of distance

Figure 3 illustrates how a particle can be a candidate for evaluation on leg k at a given timestamp. For a distance to be included in the calculations, the closest point of p must be perpendicular to the leg segment corresponding to the current SRU step. Here, only the points at timestamps 3 and 4 are aligned with the pattern segments at steps 3 and 4 on leg k. Because point P4 is the closest to leg k, d_k is set to the distance between this point and leg k. When one SRU step overlaps multiple legs, the step is divided into segments corresponding to each leg. The previous process is applied to each search segment for the same particle point. Each computed distance is then assigned to the corresponding leg.

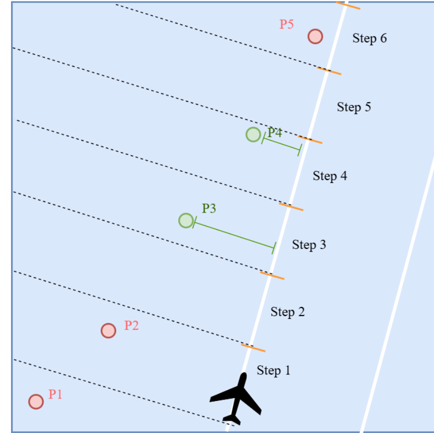

Figure 3. Example of a detection of a particle by an SRU

Using the retained $P_{\text{fail}}(p, u)$, the simulator calculates $POD(p)$, the probability that the object is detected by at least one SRU:

$$POD(p) = 1 - \prod_{u \in U} P_{\text{fail}}(p, u) . \tag{2}$$

Finally, the simulator computes the POS. The weight w_p of a particle p can be used to represent the probability that this trajectory is the object. Using w_p, we have the POS of an operation S:

$$POS(S) = \sum_{p \in P} w_p POD(p) . \tag{3}$$

Without loss of generality, we assume that $w_p = \frac{1}{|P|}$.

Algorithm 1 summarizes our interpretation of the POS computation as reported in (Abi-Zeid et al. 2019), generalized to the case of multiple objects.

Algorithm 1 Calculation of the POS based on (Abi-Zeid et al. 2019)

```
1:  for each object o do
2:      for each particle p of o do
3:          pfail(p) ← 1
4:          for each SRU s do
5:              pfail(p, s) ← 1
6:              for each leg k in the path of s do
7:                  d_k ← ∞
8:                  for each position timestep t during the leg k elapsed time do
9:                      if p_t is laterally aligned to leg_t then
10:                         d ← distance(p_t, leg_t)
11:                         if d < d_k then
12:                             d_k ← d
13:                         end if
14:                     end if
15:                 end for
16:                 pfail(p, s) ← pfail(p, s) · (1 − LRC(d_k))
17:             end for
18:             pfail(p) ← pfail(p) · pfail(p, s)
19:         end for
20:         POD(p) ← 1 − pfail(p)
21:     end for
22:     POS(o) ← ∑_p w_p · POD(p)
23: end for
```

RESEARCH PROBLEM

In order to provide an accurate POS, a search simulator processes thousands of particles, each with hundreds of positions—MSAR drifts consist of 5,000 particles with positions at 1 to 30-minute intervals for SAR Optimizer (Abi-Zeid et al. 2019), with typical drift length varying from 24 to 72 hours (Laperrière-Robillard et al. 2022). Increasing the number of SRUs, or their time on-scene, results in more legs and segments, which increases simulation time.

To our knowledge, currently, the search simulation is performed on a single CPU thread in operational DSSs. The result is that the search simulator is an important performance bottleneck. This leads to at least two issues. First, given that optimizers often need to be stopped early, a slow simulation impacts the quality of the recommendations because fewer candidate operations can be evaluated before early stopping (assuming the simulator is embedded within the optimization algorithm). Second, slow simulations also affect the DSS design because even a single operation cannot be evaluated in real-time. This might lead to less responsive designs where manual modifications to an operation must be followed by a manual call to the simulator to get the POS.

METHODOLOGY

As a response to the need for faster search simulators, we propose to accelerate Algorithm 1. Because the algorithm is based on the Monte Carlo approach, it is highly parallelizable (Rosenthal 2000), and we exploit this property to increase its performance, naturally decomposing it into parallel subtasks. A Monte Carlo method for computing a value $\hat{I}$ given n independent data points takes the following form:

$$\hat{I} = \frac{1}{n} \sum_{i=1}^{n} f(X_i)\,, \tag{4}$$

where f represents computations to be performed on X_i. Because calculating $\hat{I}$ requires repeating a large number of independent computations f, the workload can be distributed across independent computing units, each processing a batch of data points. Given C computing units, indexed by $c = 1, \ldots, C$, the n samples are efficiently distributed among them, with unit c processing n_c samples such that $\sum_{c=1}^{C} n_c = n$.

$$\hat{I}_c = \frac{1}{n_c} \sum_{i=1}^{n_c} f(X_i^c)\,. \tag{5}$$

At the end, the standard approach consists of taking the mean of every $\hat{I}_c$ for all the c computing units:

$$\hat{I} = \frac{1}{C} \sum_{c=1}^{C} \hat{I}_c \,. \tag{6}$$

In our case, the POS evaluation for each particle can be parallelized across multiple computing units. At the time of writing, two parallel programming approaches are widely available on standard computers: multi-threading on CPUs, and multi-threading on GPUs. We adapt Algorithm 1 to both methods in order to explore and compare them for the specific application to search simulation. The next subsections provide the details of our parallel algorithms.

CPU Multi-Threading

Historically, CPU performance improvements relied on increasing clock frequencies. However, due to the fundamental limitation of single-processor performance scaling, manufacturers shifted toward a multi-core architecture, where each core operates as an independent computing unit. This hardware has made parallel computing essential to improve CPUs.

Multi-threading leverages multi-core CPUs by executing multiple threads concurrently within a single process. Threads are independent sequences of instructions scheduled across available CPU cores, sharing the same memory space. Once all threads complete their tasks, their results are then combined by the main program (Birrell 1989).

In our multi-threaded approach, threads serve as the primary computing units. The core strategy involves partitioning equally the set of particles P into T subsets, where T is the number of CPU threads. Each thread t calculates its partial POS for its subset P_t of particles as follows:

$$POS_{\text{local}}(P_t, o) = \sum_{p \in P_t} w_p POD(p) \,. \tag{7}$$

After all threads finish, the main program aggregates their partial results in:

$$POS(S) = \sum_{t=1}^{T} POS_{\text{local}}(P_t, o) \,. \tag{8}$$

Algorithms 2 and 3 detail the interaction between the main function and the concurrent threads.

Algorithm 2 Main Program

1: **for** each object o **do**
2:     Split P into T subsets $\{P_1 \dots P_T\}$
3:     **for** $t = 1$ to T **do**
4:         **launch thread** LocalPOS(P_t, o)
5:     **end for**
6:     **synchronize** all threads ▷ Wait for all to finish
7:     $POS(o) \leftarrow \sum_{t=1}^{T} POS_{\text{local}}(P_t, o)$
8: **end for**

Algorithm 3 Thread $Worker$

1: **function** LocalPOS(P_t, o)
2:     $POS_{\text{local}}(P_t, o) \leftarrow \sum_{p \in P_t} w_p \cdot POD(p)$
3:     **return** $POS_{\text{local}}(P_t, o)$
4: **end function**

GPU Acceleration

Unlike CPUs, whose purpose is to handle complex tasks, GPUs are designed for executing thousands of simple parallel operations simultaneously (Hasnaine et al. 2025). As long as the task to perform involves a large number of independent mathematical calculations, GPU acceleration outperforms CPU multi-threading with a large number of cores (Kirk and Hwu 2010). NVIDIA GPUs are programmable via the CUDA parallel computing framework, whereas AMD GPUs use ROCm. We focus on CUDA because our experiments involve an NVIDIA GPU.

GPU execution relies on thousands of threads running on computing units called the *streaming multiprocessors* (SMs) (Lindholm et al. 2008; Wong et al. 2010; Kirk and Hwu 2010). These threads are organized into equal-sized *blocks*, each assigned to one SM for execution. The SM of a block executes the same instruction at the same time on smaller groups of 32 threads called *warp* (Sanders and Kandrot 2010; Kirk and Hwu 2010). Consequently, this requires all threads of a warp to execute the same instruction path for maximizing efficiency (Fung et al. 2007).

The total sequence of instructions that all threads follow for a task is called a *kernel*. Figure 4 illustrates the grid-block-thread hierarchy of a CUDA kernel structure and the division of blocks into warps of 32 threads.

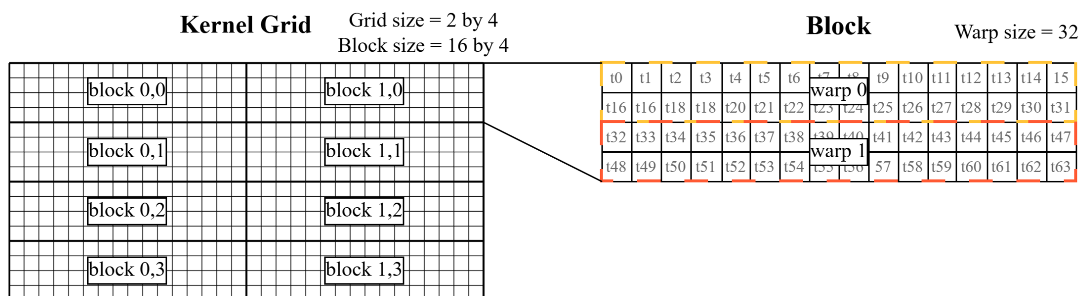

Figure 4. A simple example of a kernel structure showing the grid-block-thread hierarchy and the division of threads in a block into warps

For parallelizing Algorithm 1, two key aspects must be considered: memory coalescing and thread divergence.

Memory coalescing improves bandwidth use by reducing the number of memory transactions (NVIDIA Corporation 2025). GPUs have two types of memory. *Global memory* is a memory accessible by every thread. *Shared memory* is a faster memory space that only threads in the same block can access. When a request is made to the global memory of the GPU VRAM, the GPU fetches a contiguous data segment in the L1 cache—typically 32 bytes (8 single-precision floats). If all threads in a warp access data within nearby contiguous addresses, the hardware fulfills the request with a minimal number of transactions, resulting in up to 128 bytes of contiguous data loaded, after which the data is served to the threads from the L1 cache (Lal et al. 2022). In contrast, when threads access non-contiguous data, it requires multiple transactions and slows down execution. Ensuring aligned memory access within a warp, therefore, significantly accelerates the kernel.

Thread divergence occurs when threads within a warp need to execute different sets of instructions (Fung et al. 2007). Since threads within a warp execute the same instruction simultaneously, any variation in their execution path—such as an `if/else` statement—implies that the warp must traverse each path. This causes some threads to wait, reducing the overall performance. (Han and Abdelrahman 2011) In our case, each SRU detection segment is processed uniformly across particles. Rather than separating the set of particles into subsets for each thread, we simply distribute pairs of SRU segments and particle points to each thread, enabling full utilization of the GPU's massive parallelism.

In GPU programming, input data must be transferred from the RAM to GPU memory as contiguous arrays before kernel execution. To guarantee memory coalescing, we use primitive variable arrays rather than complex data structures. Table 1 summarizes the data arrays we send to the VRAM.

The *Particles Positions X/Y* arrays are central to our algorithm. Figure 5 shows the construction of a small *Particle Positions X/Y* array. For each SRU, the *Particles Positions X/Y* arrays store the points that overlap with the SRU on-scene time. The data is structured into contiguous blocks sorted by SRU, then by objects, and then by particles. Each data block length corresponds to the number of simulation steps where the SRU is on-scene. Points are repeated in the *Particles Positions X/Y* array of each SRU that can detect them. This design removes irrelevant data between the different particles so that no threads perform strides[1] that break memory coalescing.

Because threads are organized into independent blocks and it is hard to synchronize these blocks during execution (Xiao and Feng 2010; Feng and Xiao 2010), we use five distinct kernels. The CPU acts as a global synchronization point and launches the GPU kernels described in the following subsections sequentially.

[1]Jumps from one memory location to another.

Table 1. Description of data structures optimized for GPU memory access

Data Array	Description
Particle Positions X/Y	Longitude and latitude of particle points synchronized with their respective SRU. Stored as a flattened hierarchical vector sorted by: **SRU → Objects → Particles**.
SRU Center X/Y	Longitude and latitude of the center of the rectangle enclosing the SRU's pattern. These coordinates serve as the pivot points for the SRUs.
SRUs Positions X/Y	Local distance/position offsets for all SRUs over time. These offsets represent the physical distance (e.g., nautical miles) from the *SRU Center*. A single step may contain more than two points if it is separated into multiple segments.
SRUs Work Base	Start index in the *Particles Positions* arrays for points synchronized with the corresponding SRU.
Distance Write Base	Start index in the detection output array where results of each SRU detection for all particles are written. Each SRU is assigned to a contiguous memory region.
NB Steps per SRUs	Total number of simulation steps for each SRU.
Nb Particles per Obj	Number of particles per drifting object.
Step Mask	Start index of each step for the corresponding SRU within the *SRU Positions* arrays.
Leg Mask	Start index of each leg for the corresponding SRU within the *SRU Positions* arrays.
Obj Particles Offset	Per-particle index offset for each object in the *Particles Positions* arrays.

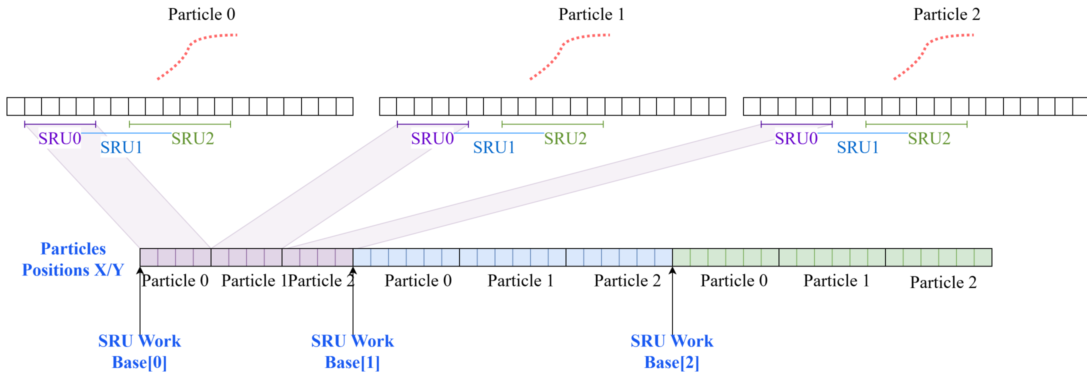

Figure 5. Example of the structure of the *Particles Positions X/Y* arrays

Kernel 1: Evaluation of Detection Distance

Kernel 1 computes the distance of the SRU search segment and its corresponding particle point. Each thread processes one distance detection.

These threads are organized as shown in Figure 6, using a two-dimensional grid of blocks. Each thread is mapped to a unique pair of one search segment and one particle point in the flattened *Particles Position X/Y* arrays. Because of the structure of these arrays presented in the previous subsection, and since many segments may be linked to the same particle point, all threads of a warp access data in the same memory region. This ensures memory coalescing is maintained.

While all particle positions are stored in two flattened arrays, they are segmented using the *SRU Work Base* index array. This allows each thread to identify which part of the *Particles Position X/Y* array to access to retrieve the particle positions synchronized with its own SRU.

The core logic of the first kernel proceeds as follows. It first fetches the input data from the global memory to the shared memory for each thread block. Because accessing data from the global memory takes time and causes overhead (Wong et al. 2010), it is better to first load the data we need from the global memory with coalescing and then access it in the shared memory as much as we need. To take advantage of the memory coalescing, data is loaded by threads in a contiguous manner (i.e., thread i loads data at index i).

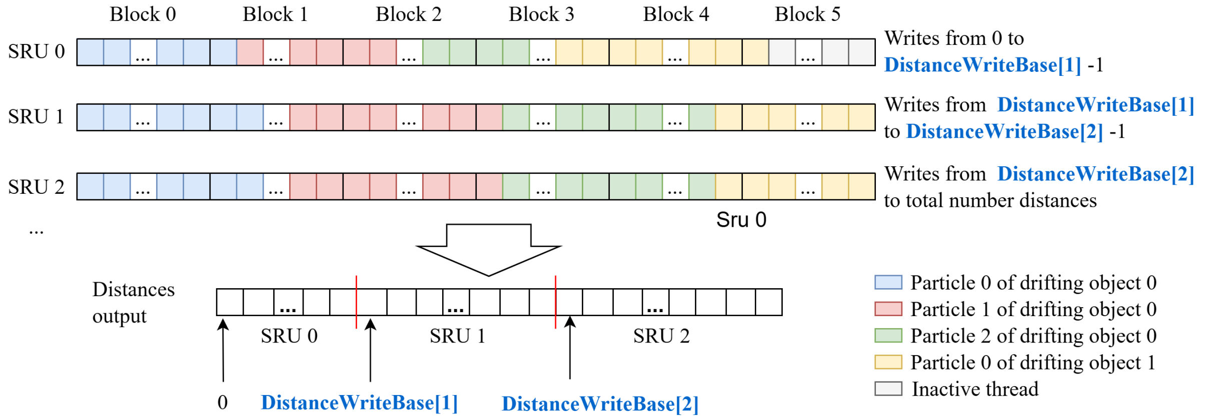

Figure 6. Schema of Kernel 1 threads management and how they write in the detections output array

Subsequently, the threads verify if the particle points are laterally aligned to the search segment, calculate the distances between them, and then write the results in the output distance array. Like in the *Particle Position X/Y* arrays, each SRU has a dedicated section in this array separated at the *Distance Write Base* values. The size of one section is $(NbSruPosition[sru] - 1) \times NbParticle[obj]$.

Kernel 1 minimizes the number of warp divergences because all threads perform the same task.

Kernel 2: Evaluation of the Detection for Each Leg

Kernel 2 performs a sequential reduction on the distances calculated by Kernel 1 to obtain the shortest detection distance between particles and search legs. A sequential reduction is a reduction operation in which elements are processed one by one in a loop, while updating an accumulator at each iteration. The resulting minimum distances are then used to compute the LRC.

Unlike Kernel 1, where each thread handles a pair of segment and particle, Kernel 2 is built so that every single thread calculates the detection probability of an entire search leg for a given particle. In this configuration, each row corresponds to one SRU, where individual threads compute the total leg detection probability $LegDetect(k) = LRC(\min(d_k))$.

To reduce global memory access, the threads of Kernel 2 first collaboratively load data from *distances* into shared memory. More specifically, threads work together when loading the distances assigned to the first thread, then the second, and so on until all the required data is loaded. This process must be partitioned because shared memory is limited. For instance, GPUs based on the NVIDIA Turing architecture are limited to 48 KB of shared memory per block[2] (NVIDIA Corporation 2025).

Figure 7 illustrates how each data is stored in the shared memory in order to be easily accessible for each individual thread. Instead of loading all distances at once, the kernel splits them into tiles (Ryoo et al. 2008; Lam et al. 1991), which are portions of data sized to fit in memory for efficient access. In the first iteration, the kernel loads the first tile of distances for each thread. Each thread then selects the smallest distance and keeps it for subsequent comparisons with the following tiles. The kernel then proceeds to load the second tile of data, and so on. The tile size is fixed at 32, which corresponds to the number of threads in a GPU warp. This ensures that all threads in a warp always read consecutive data and thus maintains memory coalescing. The kernel is limited to being executed with only 256 threads per block. This is because the shared memory size is 49,152 bytes, which has to be partitioned considering the float size (4 bytes) and the tile size (32), which leads to 256 threads per block.

With this encoding, thread divergence occurs when threads within the same warp treat search legs with a different number of steps. However, given that search legs have a small number of steps, we expect this effect to be negligible.

[2]We assume dynamic shared memory is not used.

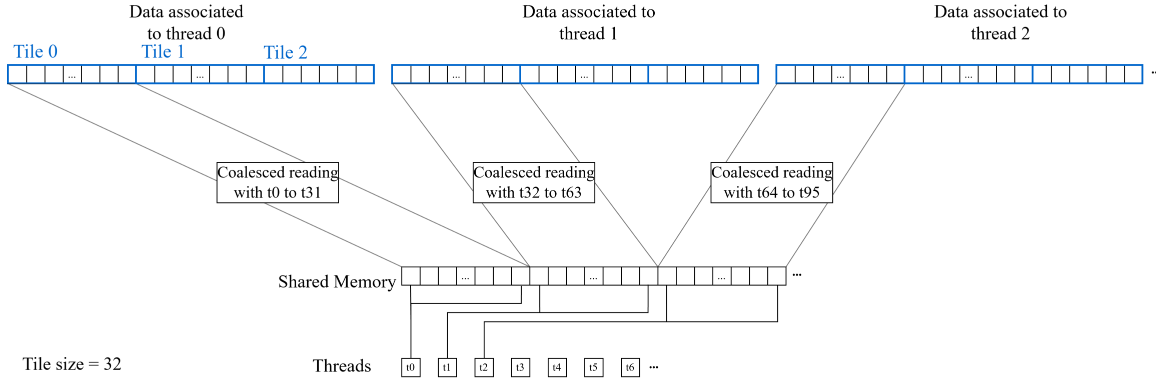

Figure 7. Visual example of the memory reading mechanism for ensuring memory coalescing for Kernel 2

Once all threads found their closest distances associated with their leg, they compute the probability of detection value using the LRC and write them in the output *detections* vector.

Kernel 3: Calculation of $P_{fail}(p, s)$

Now that all legs for all SRUs have a probability of detection for each pair of leg and particle, Kernel 3 performs another sequential reduction for calculating the $P_{\text{fail}}(p, u)$ of each SRU for each particle.

In Kernel 3, each thread is assigned to a pair of SRU and particle and performs a sequential reduction over the probability of detection of all legs of the SRU pattern through cumulative multiplication. Because of the structure of the *detections* input array, Kernel 3 reads the data by tile and stores it in the shared memory, similarly to Kernel 2.

After each tile is loaded, every thread updates its local failure probability by cumulatively multiplying the detection probabilities of the corresponding leg-particle pairs. $P_{\text{fail}}(p, u)$ is initialized to 1 and is progressively updated for each detection within a tile:

$$P_{\text{fail}}(p, u) \leftarrow P_{\text{fail}}(p, u) \times \prod_{d \in \text{tile}} (1 - d)\,. \tag{9}$$

Unlike Kernel 2, Kernel 3 does not introduce thread divergences: Threads within the same warp process the same SRU (because they are all in the same row in the thread grid), but with different particles. Consequently, the threads of a given warp iterate over the same number of leg detections.

Kernel 4: Calculation of $POD(p)$

Kernel 4 calculates the detection probability $POD(p)$ of each particle p by reducing the result of Kernel 3 over all the SRUs. It combines the data from every SRU to determine the total probability of detections for each particle across the entire search operation by performing:

$$POD(p) = 1 - \prod_{u \in U} (P_{\text{fail}}(p, u))\,. \tag{10}$$

Each thread of Kernel 4 processes a single particle. However, instead of having one row of threads per SRU as in Kernels 1 to 3, the grid contains a single row used to reduce the result of all SRUs for each particle.

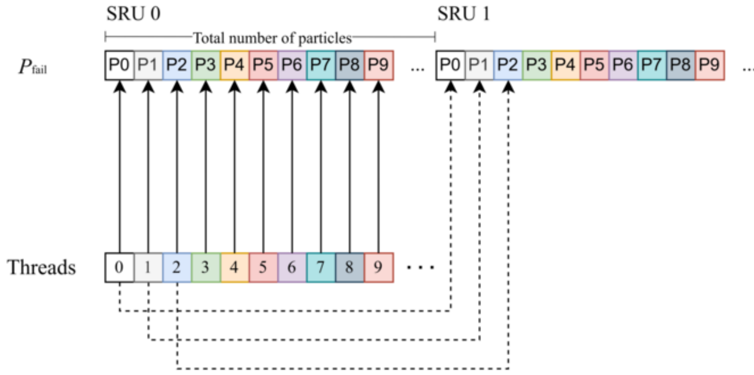

Figure 8. Global memory reading for Kernel 4

Threads are strictly aligned with the data layout to ensure optimal memory throughput. Figure 8 shows that, for each SRU, threads are mapped 1-to-1 with the particles. For instance, thread i reads the P_{fail} of particle i for a given SRU at index i. To access data of the next SRU, threads simply apply a stride equal to the total number of particles (or total number of threads). Because consecutive threads access consecutive memory indices, the memory reads are perfectly coalesced, and there is no need to copy the data from the global to the shared memory. Also, threads do not diverge because they perform the same instructions.

Kernel 5: Calculation of the POS

Once the POD calculation is complete, the final step is to determine the POS for each object. This process requires a large-scale summation of the POD values across all particles for every object.

While a sequential reduction could work as in previous kernels, this would be inefficient here. Indeed, the number of drifting objects to process is small (e.g., in the experiments, we assume one drifting object) compared to the number of particles per object. Consequently, it is a better choice to distribute the workload across a large number of threads, reducing together the same object.

Doing so requires writing to shared memory, followed by synchronization barriers and subsequent read operations, which may cause a bottleneck across all the threads of the same block. To avoid this, Kernel 5 implements the *warp shuffle down reduction* technique (Luitjens 2014), where threads communicate directly through registers by using warp-level primitives. This approach minimizes synchronization and improves performance.

Shuffle down reduction is based on the `shfl_down_sync` instruction. When using the `shfl_down_sync` instruction, registers are transferred from a thread at a higher index in the warp to a thread at a lower index. This allows bypassing the need for using the shared memory and synchronization between threads.

Figure 9 depicts how the warp reductions work. The `shfl_down_sync` instruction can be used to perform a complete warp reduction sum by using all of the 32 threads in a warp to sum 32 values. As shown in the figure, the work is divided into two at each step. The first 16 threads (top) get the value from the thread 16 positions ahead by using `shfl_down_sync(v, 16)` and add it to their own value. Then, the first 8 threads do the same with an offset of 8, followed by offsets of 4, 2, and 1. After five iterations, the total sum of all 32 values is concentrated into the first thread's register. For POS computation, consider a warp where each thread holds a value v representing a particle's POD. When executing `shfl_down_sync(v, 2)`, for example, each thread i directly retrieves the value v from thread $i+2$. In this specific direction, thread 0 receives the data from thread 2, thread 1 from thread 3, and so on.

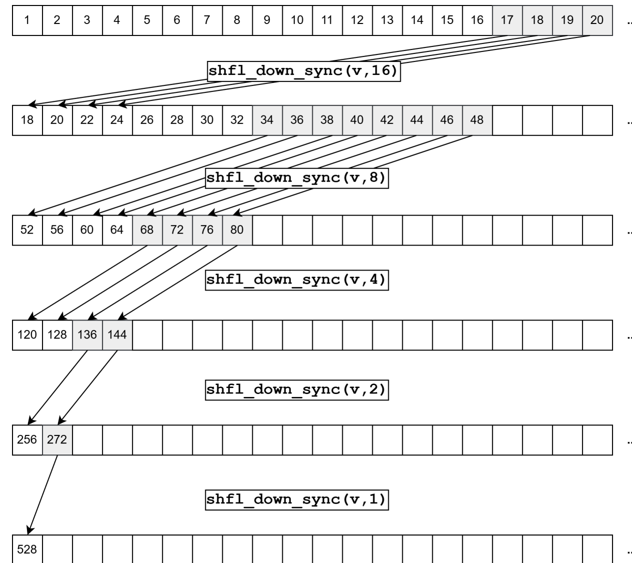

Figure 9. Warp reduction process diagram

With this warp reduction logic established, we can now extend the process to an entire block of 1,024 threads, which consists of 32 warps of 32 threads each. In this process, as shown in Figure 10, Kernel 5 first performs an independent warp reduction for each of the 32 warps, resulting in 32 partial sums stored in the shared memory. These results are then collected by the first warp, which performs one final warp reduction to produce the total sum of the 1,024 values.

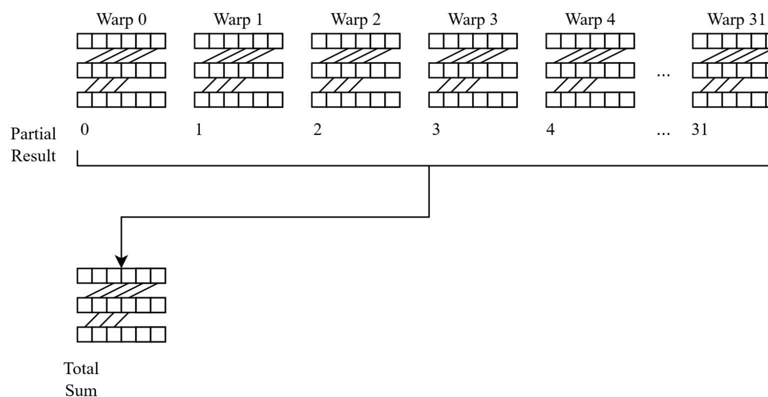

Figure 10. Block reduction process diagram

Kernel 5 implements this hierarchical reduction to compute the final POS for each object. Each CUDA block is assigned to one object. Within a block, each thread first performs a local summation of the POD values. Because the number of particles can exceed the number of threads within a block, threads access their local POD data by starting at their specific index and striding by the block size.

Once each thread in a block has calculated its local sum, Kernel 5 executes a `block_reduce_sum` instruction, then divides the result by the number of particles, resulting in the POS of the current object. After a block completes the calculation for one object, if there are more objects than there are available blocks, the current block performs a stride by the grid size until all objects have received a POS. This ensures that Kernel 5 remains efficient even when the number of hardware resources is limited.

Kernel 5 is designed to be fully coalesced, as consecutive threads access consecutive data points within the POD array. Thread divergence may occur, but its impact is negligible because it is rare. Threads can diverge only once at the end of the summation loop, when the number of particles of an object is not a multiple of 32.

NUMERICAL EXPERIMENTS, RESULTS AND DISCUSSION

We performed a series of experiments comparing the GPU-accelerated, the multi-threaded (with 12 threads), and the single-threaded approaches. The tests were performed on a standard computer equipped with an NVIDIA GeForce RTX 2060 mobile (6 GB GDDR6 VRAM), an Intel Core i7-9750H, and 16GB (2×8 GB) of DDR4 RAM. The code of the search simulator was written in C++ and CUDA 12.6.

We performed a scalability test, where we increased the number of SRUs, and a throughput test, where we calculated the total number of simulations for one SRU performed within 10 seconds. The evaluation metrics we use are the total simulation time and the performance gain (speedup) compared to the single-threaded baseline. For all experiments, the LRC and the drift were fixed. Varying the LRC or the drift could induce a small variation in the search simulation time, which is negligible. We used OpenDrift (Dagestad et al. 2018) to generate a drift with 5,000 particles for a single object sampled every 5 minutes.

Scalability Test

Figure 11 presents the total simulation time (in seconds) as the number of SRUs increases from 1 to 40. All SRUs arrive and depart the scene simultaneously. Each SRU searches for 5 hours and 45 minutes, resulting in 69 evaluation steps per SRU (one every 5 minutes). The first plot (top row) shows the performance of all approaches. The next three plots (second row) show a per approach scaled version.

Even with few SRUs, the benefit of the parallelism is clear. With one SRU, the simulation time decreases from 0.287 seconds with the single-threaded baseline to 0.046 seconds with multi-threading—a 6.2× speedup. As for the GPU approach, it takes 0.0038 seconds to simulate one SRU, which represents a 75× speedup over the single-threaded baseline. The difference between the parallel and the single-threaded approaches increases as the number of SRUs increases, reaching a 6.5× speedup for the multi-threading and a 120× speedup for the GPU at 40 SRUs.

Based on the curves in Figure 11, we estimate that the marginal simulation time cost of adding a single SRU for the single-threaded approach is about 279.4 milliseconds. The multi-threaded approach reduces the marginal cost to about 42.9 milliseconds per SRU, whereas the GPU reduces it to 2.2 milliseconds.

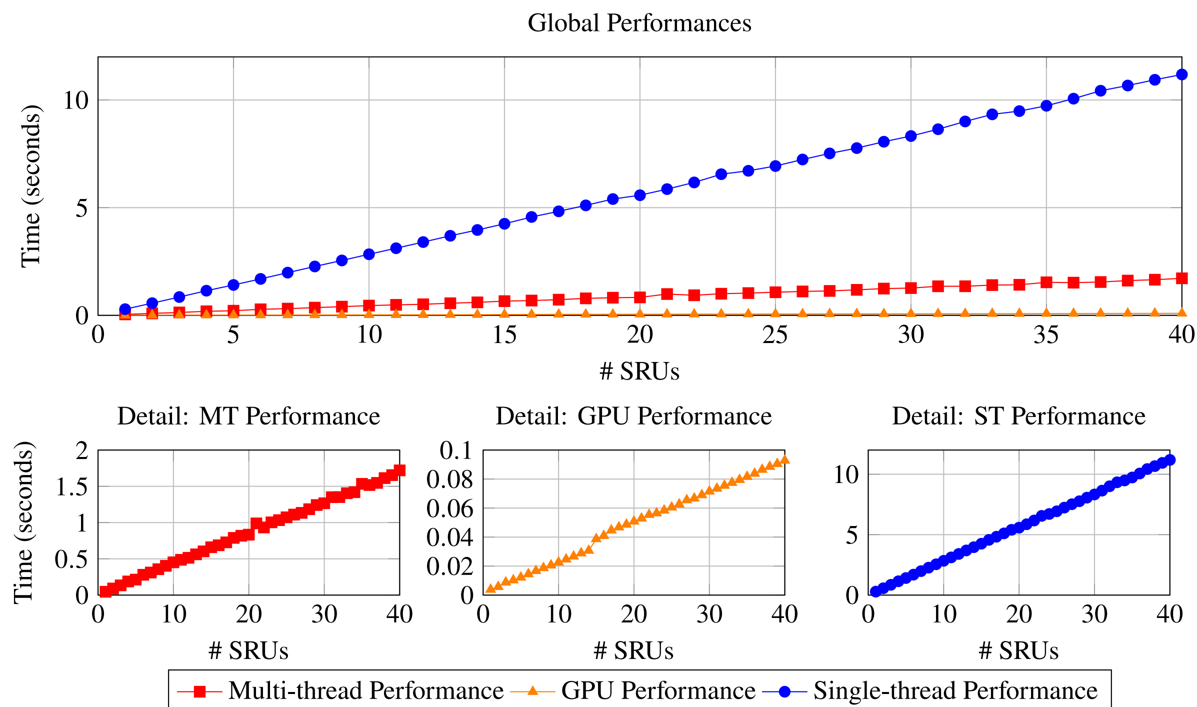

Figure 11. Performance for each approach as the number of SRUs grows

However, the GPU approach comes with a loading overhead because the C++ objects need to be transformed into a one-dimensional array and transferred to the GPU before the launch of Kernel 1. This transformation occurs when data is updated between simulations, but it is kept negligible by performing target transfers. Target transfers work as long as only the dynamic data are updated, e.g., the SRU position arrays. When data changes affect *Particles Positions X/Y* arrays, the complete one-dimensional array is reconstructed and transferred to the GPU. This can considerably reduce the GPU's speedup (Gómez-Luna et al. 2012).

As an example, transforming and loading a scenario with one object of 5,000 particles and one SRU with 69 steps to the GPU takes 0.039 seconds. Transforming and loading only SRUs without modifying the *Particle Positions X/Y* arrays takes 1 millisecond. This means that, depending on the changes being made between two simulations, the simulation time on the GPU can vary. But even with a 0.039 seconds overhead, the total GPU simulation time (0.0038 + 0.039 seconds) is lower than the total multi-threaded simulation time (0.046 seconds).

Throughput Test

To evaluate the effective benefit of the parallelization and measure the simulator's throughput, we developed a Python-based benchmarking framework that interfaces directly with our C++ implementations using PyBind11. We consider one SRU with a 69 time-steps pattern searching for one object. The framework involves a stochastic iterative algorithm that randomly updates and triggers a search simulation each time it generates a new candidate operation. The overhead caused by the algorithm for generating candidates is negligible. In the best case, the algorithm generates a new candidate operation by randomly modifying the SRU pattern position, searching for the best possible candidate operation under a solving time limit. In the worst case, the algorithm modifies the SRU start time, which requires a full reload.

Table 2 shows the number of simulations and the time per iteration (in seconds) for each approach under a 10 seconds solving time limit. For the GPU, Table 2 shows the best case (high-speed loading) and the worst case (fully memory reloading). The GPU outperforms the single-threaded and the multi-threaded approaches in all cases.

By dividing the allowed solving time (10 seconds) by the number of simulations, we observe that the CPU-based approaches yield a time per simulation similar to the simulation time obtained during the scalability test. The single-threaded baseline averages 0.26 seconds (vs 0.28 seconds), and the multi-threaded approach averages 0.049 seconds (vs 0.047 seconds).

The GPU approach time per simulation is 1 millisecond per iteration, compared to the 3.8 milliseconds in the scalability test with one SRU. This is normal because the CUDA context initialization overhead is amortized in the throughput test.

Table 2. Number of simulations completed in 10 seconds for each approach

Method	Number of simulations	Time per iteration (seconds)
Single-Thread	38	0.26
Multi-Thread (12 threads)	202	0.049
GPU (Best case - Only partial loading)	9910	0.0010
GPU (Worst case - Only full reloading)	289	0.0346

Discussions

As shown, CUDA provides the greatest speedup for MSAR search simulation. We argue that it can also benefit simulators used in other crisis contexts, given that some conditions are met.

First, GPUs rely on independent threads working simultaneously. Maximizing speed requires the simulation to be decomposed into a large number of short independent tasks, e.g., a Monte Carlo simulator with many samples. Simulators performing a few sequential and interdependent tasks will only use a few threads and underutilize the GPU SMs. Each task should also execute the same sequence of instructions to minimize thread divergence.

Second, the data are best preloaded in the GPU memory and stored into contiguous arrays. To allow memory coalescing, threads should access memory in a regular and predictable pattern. Simulations requiring irregular data or unpredictable access (e.g., graph based data or random data access) might result in performance loss. The amount of data transferred to the GPU also impacts performance because of the transfer overhead when initializing the simulation. Limiting data transfer and updating only necessary data can significantly reduce the overhead.

Third, when embedding a simulator within an optimization algorithm to calculate an objective function, it is important to consider the time required by the optimizer to generate a candidate solution. If calculating candidate solutions is slow, it can significantly reduce the overall speedup achieved by parallelizing the simulation.

To illustrate this, assume an available time T, a single-threaded simulation time t, a parallel speedup factor s, and an optimization algorithm where an iteration consists of generating a candidate solution and evaluating it with simulation. Given s, the gain in number of iterations provided by parallelism can be expressed as a function G of o, the time required by the optimization algorithm to generate a candidate solution:

$$G(o) = \frac{T}{s^{-1} \cdot t + o} - \frac{T}{t + o} . \quad (11)$$

$G(o)$ is the difference in the number of iterations when using the parallel simulator that runs a fraction s^{-1} of the time t required by the single-threaded simulator.

Figure 12 shows the gain $G(o)$ for specific T, t, and s values. The rate of change for small o shows that a speedup of all components of an optimization algorithm is important, not only the simulator. In the evaluated context for MSAR operations planning, the algorithm overhead o was low, and the single-threaded simulator was identified as the bottleneck. In general, such a context should be observed before developing parallel simulators.

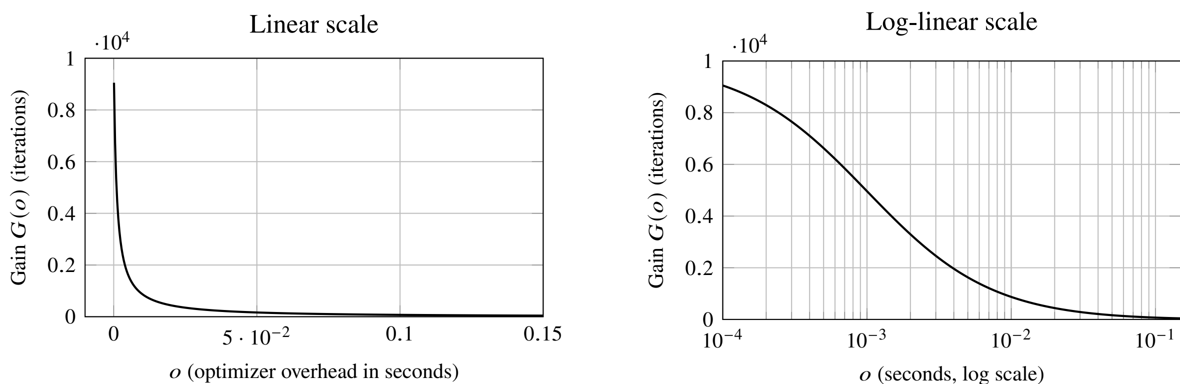

Figure 12. Gain $G(o)$ as a function of optimizer overhead o (left: linear scale, right: logarithmic scale) with $T = 10$, $t = 0.26$, and $s = 260$

CONCLUSION

As a response to the need for faster search simulators in MSAR DSSs, we developed and evaluated two high-performance alternatives to single-threaded search simulators. By doing so, we contributed a methodology to overcome the computational bottleneck of search simulation. In our experiments, CPU multi-threading proved 6.7 times faster than the single-threaded baseline, whereas GPU acceleration proved 75-120 times faster in the scalability test, peaking at 260 times faster once the GPU is warmed up in the throughput test.

We also discussed how GPU performance gains depend on whether a full memory reloading is required or not, but also on general design principles—e.g., parallelizable design, memory-aware layout, and optimization overhead. These findings apply to other crisis management contexts where simulation is used, and can guide the development of algorithms for parallel simulators.

By enabling faster simulations in DSSs for MSAR operations planning, we aim to support decision-making in time-critical situations and shorten the overall response time. Faster simulators can also enable more sophisticated algorithms and more responsive systems, all of which could lead to better recommendations for a variety of systems.

ACKNOWLEDGMENTS

This research was funded by NSERC (nserc-crsng.gc.ca) [grants RGPIN-2021-03495 and DGECR-2021-00189].

REFERENCES

Abi-Zeid, I., Morin, M., and Nilo, O. (2019). "Decision Support for Planning Maritime Search and Rescue Operations in Canada:" en. In: *Proceedings of the 21st International Conference on Enterprise Information Systems*. Heraklion, Crete, Greece: SCITEPRESS - Science and Technology Publications, pp. 328–339.

Birrell, A. (Jan. 1989). *An Introduction to Programming with Threads*. Tech. rep. 35.

Dagestad, K.-F., Röhrs, J., Breivik, Ø., and Ådlandsvik, B. (Apr. 2018). "OpenDrift v1.0: a generic framework for trajectory modelling". en. In: *Geoscientific Model Development* 11.4, pp. 1405–1420.

Esmaeilpour, A., Morin, M., and Abi-Zeid, I. (May 2025). "Toward Blackbox Optimization for Maritime Search and Rescue". In: *Proceedings of the International ISCRAM Conference*.

Feng, W.-c. and Xiao, S. (2010). "To GPU synchronize or not GPU synchronize?" In: *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 3801–3804.

Fung, W. W., Sham, I., Yuan, G., and Aamodt, T. M. (2007). "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 407–420.

Gómez-Luna, J., González-Linares, J. M., Benavides, J. I., and Guil, N. (2012). "Performance models for asynchronous data transfers on consumer Graphics Processing Units". In: *Journal of Parallel and Distributed Computing* 72.9, pp. 1117–1126.

Han, T. D. and Abdelrahman, T. S. (2011). "Reducing branch divergence in GPU programs". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California, USA: Association for Computing Machinery.

Hasnaine, Q. R., Fouda, M. M., and Chiu, S. C. (2025). "GPU vs. CPU: A Comparative Study of Performance, Architecture, and NVIDIA's Innovations in Modern Computing". In: *2025 IEEE 4th International Conference on Computing and Machine Intelligence (ICMI)*, pp. 1–6.

Hillier, L. E. (2008). "Validating and Improving the Canadian Coast Guard Search and Rescue Planning Program (CANSARP) Ocean Drift Theory". Master's thesis. St. John's, NL, Canada: Memorial University of Newfoundland.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA: Morgan Kaufmann.

Kratzke, T. M., Stone, L. D., and Frost, J. R. (2010). "Search and rescue optimal planning system". In: *2010 13th Conference on Information Fusion (FUSION)*. IEEE, pp. 1–8.

Lal, S., Varma, B., and Juurlink, B. (2022). "A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads". In: *International Journal of Parallel Programming* 50, pp. 189–216.

Lam, M. D., Rothberg, E. E., and Wolf, M. E. (1991). "The cache performance and optimizations of blocked algorithms". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, pp. 63–74.

Laperrière-Robillard, T., Morin, M., and Abi-Zeid, I. (Nov. 2022). “Supervised learning for maritime search operations: An artificial intelligence approach to search efficiency evaluation”. en. In: *Expert Systems with Applications* 206, p. 117857.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In: *IEEE Micro* 28.2, pp. 39–55.

Luitjens, J. (2014). *Faster Parallel Reductions on Kepler*. URL: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

NVIDIA Corporation (2025). *CUDA C++ Programming Guide*. NVIDIA Corporation.

Rosenthal, J. S. (2000). “Parallel computing and Monte Carlo algorithms”. In: *Far East Journal of Theoretical Statistics* 4.2, pp. 207–236.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’08. Salt Lake City, UT, USA: Association for Computing Machinery, pp. 73–82.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.

Stone, L. D., Royset, J. O., and Washburn, A. R. (2016). “Search for a Stationary Target”. In: *Optimal Search for Moving Targets*. Cham: Springer International Publishing, pp. 9–48.

Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). “Demystifying GPU microarchitecture through microbenchmarking”. In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246.

Xiao, S. and Feng, W.-c. (May 2010). “Inter-block GPU communication via fast barrier synchronization”. In: pp. 1–12.