Accelerated Systems: Principles and Practice Assessment 2

Date: 06-04-2025

Exam number: B222009

Instructor: Ruper Nash

1. Build instructions

To build and run the application starting from a clean working repository, the first step is to ensure that the file perc_gpu.cpp is correctly placed inside the src directory, alongside the existing source files and the provided CMakeLists.txt. This file contains the GPU-specific implementation of the percolation algorithm and is essential for enabling GPU acceleration during execution.

Once the source files are in place, it is necessary to prepare the software environment by loading the appropriate modules required for compiling and running the application. These modules include CMake, GCC version 10.2.0, the NVIDIA HPC SDK without MPI support (nvhpc-nompi/24.5), and OpenMPI built with CUDA support (openmpi/4.1.6-cuda-12.4). These modules can be loaded using the module load command provided by the system's environment manager.

After setting up the environment, the build process begins with CMake. Navigate to the root of the repository and run the CMake configuration command, which sets up a build directory and prepares the build system. The configuration should be executed with the following command: cmake -S . -B build -DACC_MODEL=OpenMP -DCMAKE_CXX_COMPILER=nvc++ -DCMAKE_BUILD_TYPE=RelWithDebInfo.

This command instructs CMake to use OpenMP as the acceleration model, the NVIDIA HPC C++ compiler (nvc++) for compilation, and to enable a build type that includes debug information optimized for performance.

Once the configuration step completes successfully, the program can be compiled using the command cmake --build build -j 4, which builds the software in parallel using four threads. This process generates an executable named test, which will be in the src/build directory.

The application includes several SLURM job scripts designed to run tests using 1, 4, or 8 CPUs and GPUs on a computing cluster. These scripts are already configured for different resource allocations but are set to execute the binary from the path /build/test. In this setup, however, the correct path is src/build/test, so the user must update each script accordingly. Once this change is made, the application can be successfully executed on the cluster with the intended configuration.

The scripts are configured to run three standard test cases: **Small**: 512×512 cells, 60% porosity, **Medium**: 1024×2048 cells, 50% porosity and **Large**: 4096×4096 cells, 30% porosity.

2. Design

The application is designed with a focus on efficient distributed computation using a hybrid parallelization approach that combines MPI for inter-process communication and OpenMP for intra-node GPU offloading. The architecture supports both CPU and GPU versions of a percolation algorithm, ensuring flexibility and scalability across different computing environments. The software follows the PIMPL (Pointer to Implementation) design pattern, which promotes clean interfaces and better encapsulation of implementation details.

Memory management is structured around a three-level hierarchy that includes host memory on the CPU, device memory on the GPU, and dedicated MPI communication buffers. This structure supports efficient data transfer and parallel computation. The application uses ghost cells—also known as halo regions—to manage boundary communication between subdomains and employs double buffering with primary (state) and temporary (tmp) arrays to facilitate iterative updates while minimizing data copying.

The parallelization strategy decomposes the computational domain across processes using MPI, with each process potentially offloading computations to a GPU using OpenMP target directives. Halo data exchanges between neighboring processes are performed using non-blocking MPI calls to overlap communication with computation. GPUs are assigned in a round-robin fashion across nodes to balance the workload.

Table 1. Summary of core modules and functions in the percolation simulation application, detailing the role of each function in either GPU or CPU execution paths, including memory handling, boundary data exchange, and main iteration logic.

Module name	Function name	Purpose
main	percolate_gpu_step	Core computation step on GPU, updates cell values based on neighbors
main	percolate_cpu_step	CPU version of the computation step
GpuRunner	halo_exchange	Handles boundary data exchange between processes
GpuRunner	copy_in	Copies data from source to local grid, handles GPU memory
GpuRunner	copy_out	Copies results from local grid to destination
GpuRunner	run	Main execution loop, handles iteration and convergence
CpuRunner	Impl::halo_exchange	CPU version of boundary data exchange
CpuRunner	copy_in	CPU version of data input
CpuRunner	copy_out	CPU version of data output
CpuRunner	run	CPU version of main execution loop

Key optimizations include pointer swapping to avoid costly array copying, careful GPU memory management through OpenMP directives, and logic to skip

unnecessary computations for solid (inactive) cells. These optimizations collectively improve performance, especially in memory-bound scenarios.

Core computational functions are divided between CPU and GPU modules. The *percolate_gpu_step* and *percolate_cpu_step* functions perform the main update logic on their respective architectures. Data movement and halo exchanges are handled by *copy_in*, *copy_out*, and *halo_exchange* methods within the *GpuRunner* and *CpuRunner* classes. Supporting utilities such *as fill_map*, *write_state_png*, and *txt_print* are responsible for initializing the simulation grid, exporting final states, and debugging.

Data structures such as *GpuRunner::Impl* store the primary state arrays, halo communication buffers, and information about neighboring MPI processes and GPU assignments. The *ParallelDecomp* structure manages domain decomposition and provides metadata about the process grid and subdomains.

Communication follows a structured halo exchange pattern, particularly for the GPU version, which involves transferring data between device and host, posting MPI receives and sends, packing and unpacking buffers, updating ghost cells, and synchronizing data back to the GPU. Convergence checks are performed using a global *MPI_Allreduce* operation, with the root process responsible for output and final statistics.

The timing results show that, in this particular test, the CPU implementation was faster than the GPU version (0.062 seconds vs. 0.117 seconds). This is likely due to the overhead of data transfers between the host and device, the small subdomain size not fully leveraging the GPU, and differences in how communication and computation overlap. Nevertheless, the GPU implementation performed correctly, with all eight processes across two nodes successfully using their assigned GPUs, as confirmed by full GPU utilization. The percolation algorithm completed in 29 steps—well under the 16,384-step limit—indicating correct functionality. The implementation also includes key improvements: unique GPU assignment per process, flexible fallback to CPU if GPUs are unavailable, properly synchronized host-device transfers, and consistent memory management throughout.

Overall, the design emphasizes efficient parallel execution, flexible deployment across CPU and GPU resources, and careful coordination of data flow between memory hierarchies and compute units.

3. Performance results

Figure 1. Comparison of runtime performance between CPU and GPU implementations across three problem sizes (Small: 512², Medium: 1024×2048, Large: 4096²) using 1, 4, and 8 processes. Lower runtime values indicate better performance, with CPU implementations showing consistently faster execution times despite GPUs having higher theoretical compute capacity.

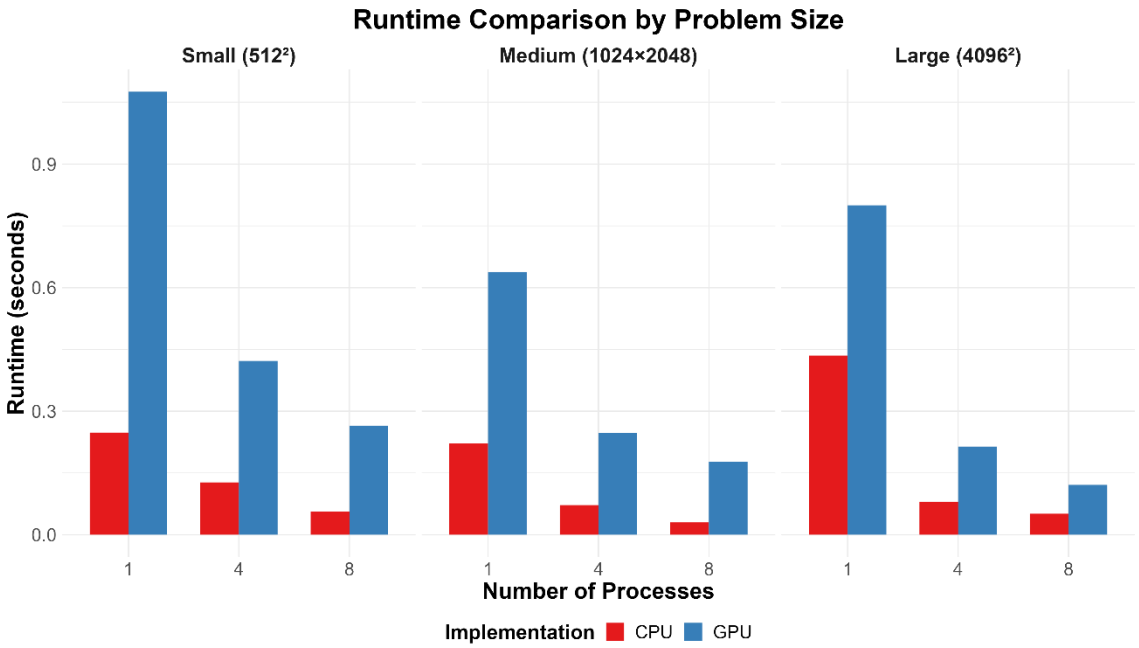


Figure 2. *Communication overhead percentage for CPU and GPU implementations across problem sizes and process counts. GPU implementations consistently exhibit significantly higher communication overhead (87-97%) compared to CPU implementations (0.1-22%), explaining the performance differences observed despite GPUs' higher theoretical compute power.*

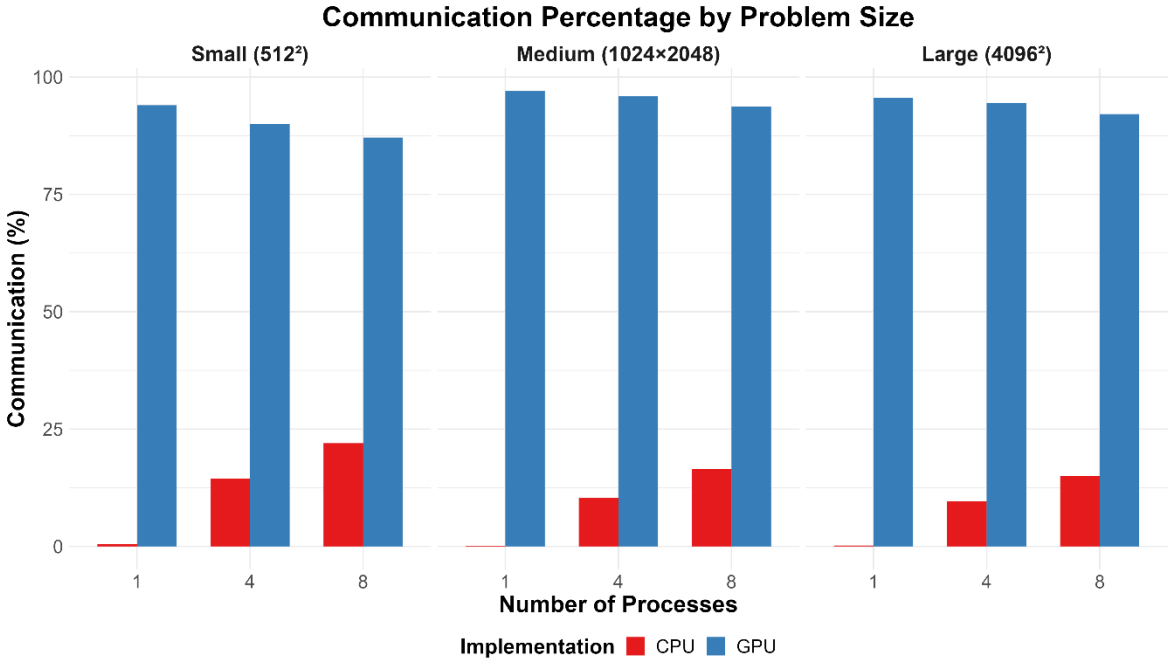
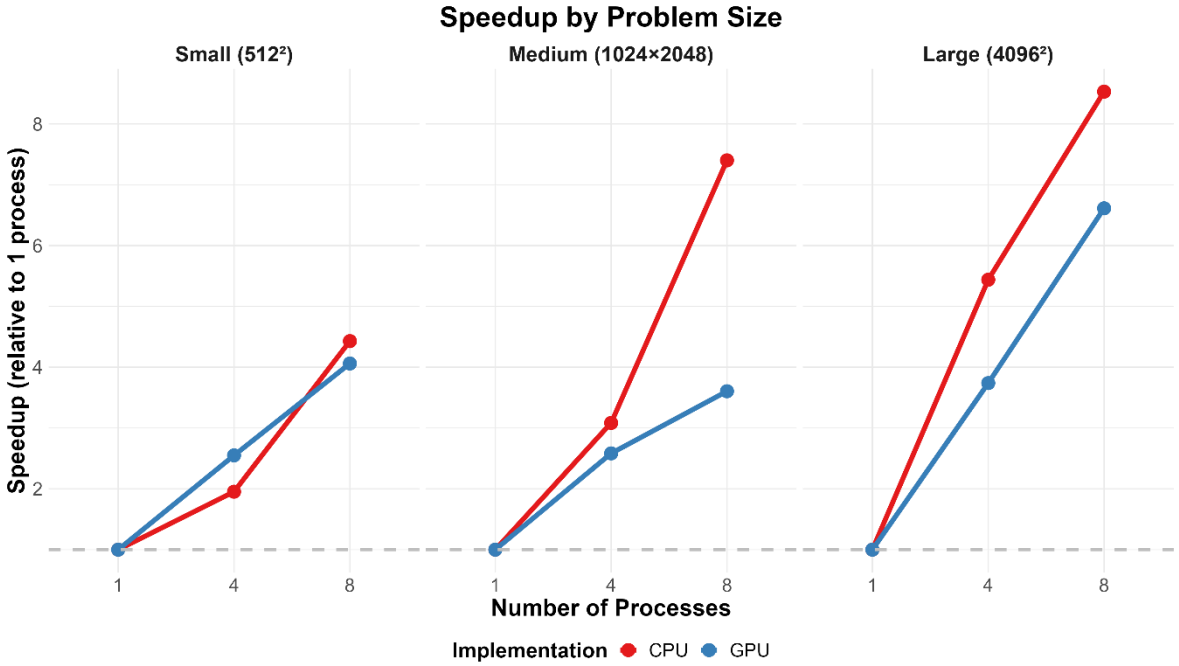


Figure 3. *Speedup achieved when scaling from 1 to 8 processes for both CPU and GPU implementations across different problem sizes. CPU implementations demonstrate superior scalability, achieving nearly linear speedup with process count increases, while GPU implementations show more modest speedup due to higher communication overhead.*



The performance results of the application reveal that memory transfers are the primary bottleneck in execution. The total time spent on memory operations amounts to approximately ~2.35 seconds, with an average transfer size ranging from 33 to 66 megabytes and a total of 261 memory transfers. This indicates that data movement dominates the runtime and limits the overall performance.

Despite the heavy cost of memory operations, the computational kernels themselves are highly efficient. The main kernel executes on average 84 times with a total execution time of just ~30.24 milliseconds, averaging 360 microseconds per instance. The execution is remarkably consistent, showing very low variance with a standard deviation of only ~13.37 microseconds. A separate constructor kernel is launched once, taking ~9.38 milliseconds.

Taking a deeper look into the NVIDIA profiler results and making a detailed analysis of CUDA specific memory operations I encountered that host-to-device transfers occur 91 times, moving a total of 6,045.7 megabytes of data. Each transfer averages 66.4 megabytes and takes about 13.74 milliseconds, summing up to 1.25 seconds, which represents 53.1% of the total time spent on memory operations. On the other hand, device-to-host transfers are slightly more frequent, with 170 transfers totaling 5,709.8 megabytes. These transfers are smaller on average—33.6 megabytes—and faster, with an average time of 6.49 milliseconds per transfer and a total duration of 1.10 seconds, accounting for the remaining 46.9% of memory operation time.

4. Discussion

My GPU implementation of the percolation algorithm consistently underperforms when compared to the CPU version, across all problem sizes and process counts. I was not expecting this result since GPUs are typically better suited for parallel computations. However, taking a deeper look into the results, it's obvious that most of the GPU execution time (between 87% and 97%, see ***Figure 2***) is consumed by communication overhead, including CPU-GPU data transfers and MPI communication. In contrast, only a small fraction of GPU time, about 2% to 12%, is spent on actual computation.

But the CPU implementation tells a different story. It is primarily computation-bound, with communication costs ranging only from 0.1% to 14.5% of the total runtime. This allows the CPU version to make more effective use of processing power and its hierarchical cache system. When the number of MPI processes is increased, both implementations benefit from performance improvements. But the CPU version scales more efficiently, showing better utilization of added resources.

Interestingly, the performance gap between the two implementations narrows for larger problems. For example, with the largest tested grid size of 4096×4096 and eight MPI processes, the GPU version comes closest to matching the CPU, although it still lags by a factor of 2.4. For smaller problems, the GPU is between 5 to 6 times slower.

The core reason for the GPU's inferior performance lies in the nature of the percolation algorithm, which is memory-bound rather than compute-bound. I think this is probably the most significant detriment factor for efficient use of GPUs in this particular application. Because in each iteration, a large volume of data must be transferred between the CPU and GPU, yet still the computation itself is minimal (a simple operation for finding the maximum number). This low arithmetic intensity, where very few operations are performed per memory access, is probably what is limiting the GPU's efficiency. In addition, every MPI communication step requires additional data transfers back and forth between host and device, intensifying the memory bottleneck. Synchronization between the CPU and GPU further adds to the delay.

When I took a detailed look into the percolation algorithm, I noticed that on each cell we read 5 values (centre + 4 neighbours), then we perform 4 max operations and finally just wite 1 value. This gives us an arithmetic intensity of only 4 operations per 6 memory operations (5 reads + 1 write). This is a very low compute-to-memory ratio, making it a memory-bound problem.

Looking into strong scaling performance, we can observe how each version behaves as more processes are added for the same problem size. For the CPU version on a 4096×4096 grid, runtime drops from 0.435 seconds with one process to 0.080 seconds with four processes and 0.051 seconds with eight processes. This corresponds to a 5.4x speedup and 135% efficiency at four processes, and an 8.5x

speedup with 106% efficiency at eight processes (indicating super good linear scaling) likely due to improved cache utilization. The GPU version, on the same problem size, goes from 0.214 seconds with four processes to 0.121 seconds with eight processes, resulting in only a 1.8x speedup and 89% efficiency. Its sub-linear scaling highlights the impact of persistent communication overhead (see **Figure *1*** and ***Figure 3***).

Comparing with the CUDA version I submitted for the first coursework, this previous (single-host-single-GPU) version eliminates many of the inefficiencies found in the OpenMP multiple GPU approach. The CUDA version avoids frequent host-device memory transfers by using pinned memory and CUDA streams. It also takes advantage of several CUDA-specific features, such as 32×32 thread blocks for better occupancy, register-based fast access to frequently used values, and memory access patterns optimized for coalescence. Additionally, in this version i separated the compute and data communication streams to allow asynchronous operation and reduce idle time, making this version much faster and optimized. This CUDA version had significantly lower memory transfer overhead, better GPU utilization, and more efficient kernel execution. For large enough problems, it is likely to outperform both the CPU and OpenMP GPU versions, primarily because it eliminates the critical bottleneck of excessive host-device transfers, but this is only valid for the 1 GPU implementation.

Despite using a powerful V100 GPU with a theoretical peak performance of around 7 TFLOPS in single precision, my OpenMP GPU version of the classwork achieves only about 0.16% of that potential. For a typical large problem size (4096 × 4096 grid), with each of the 16,777,216 cells requiring approximately 20 operations (including max operations, comparisons, and index calculations inside the *percolate_gpu_step* function) the total comes to around ~335 million operations per kernel. Given a kernel execution time of 0.03 seconds, this results in an actual performance of roughly ~11.18 GOPS. With this estimate, I suspect that less than 0.2% of the GPU's capacity is being utilized, and only about 2% of the total runtime is spent on computation (the rest is likely lost to communication and synchronization overhead).

To address the communication bottleneck in the open MP implementation, It would be necessary to implement the following optimizations: (1) First, host-device transfers need to be minimized. One way to achieve this is by separating the computation of inner and boundary cells, allowing inner cells to propagate independently without waiting for halo exchanges. Data should be kept on the GPU across multiple iterations, and halo exchanges should transfer only boundary data instead of the full array. (2) Second, using CUDA-aware MPI would allow direct communication from GPU memory, eliminating the need for intermediate CPU transfers. And (3) Third, the implementation should aim to increase the amount of computation relative to communication by assigning larger problem sizes per GPU, batching multiple percolation steps into a single kernel, or allowing threads to process multiple

cells. The open MP version of the algorithm should be preferred over the CUDA implementation only when the problem size does not fit in a single GPU.

The main reason my initial CUDA implementation performs better and is easier to manage in terms of memory is its explicit control over memory types and transfers. CUDA offers better tools for performance optimization, a clear separation between host and device memory, and direct support for advanced features like pinned memory and streams. In contrast, OpenMP's more abstract model can make it harder to fine-tune these aspects (for instance, I repeatedly encountered compiler issues when trying to use map clauses in target directives to transfer only boundary data). While OpenMP is designed for portability and ease of use, it sacrifices the level of control needed for maximizing performance.

In conclusion, the performance analysis of my Open MP implementation of the percolation algorithm reveals that this problem is fundamentally memory-bound, which limits the benefits of GPU acceleration, especially under the current OpenMP-based approach. The CPU implementation performs better by avoiding host-device transfer overhead and taking advantage of modern CPUs' memory bandwidth and caching. While the CUDA implementation could offer significant improvements through GPU-optimized techniques, it too would be constrained by the algorithm's inherently low arithmetic intensity. Any future attempts to enhance GPU performance should consider algorithmic redesigns or communication-efficient strategies, such as CUDA-aware MPI, to unlock the potential of GPU parallelism for this type of workload.