|epcc|

GPU-Accelerated Percolation Model

Accelerated Systems: Principles and Practice Coursework 2

Exam Number: 185276

April 8, 2025

Contents

1 Introduction

This coursework concerns the adaptation of an existing CPU-only percolation code to exploit both multi-node parallelism and GPU acceleration on the Cirrus HPC system. The primary task involves porting the original MPI-based implementation to run efficiently on multiple nodes, targeting both CPU and GPU partitions.

In the following sections, the chosen design approach is explained, along with the relevant compilation and execution steps on Cirrus. Subsequently, performance results are presented for various problem sizes and node configurations, and a discussion provides insights into optimization choices and potential future enhancements.

2 Build Instructions

2.1 Environment Setup

On Cirrus, the following modules must be loaded to provide the required compilers and libraries:

```
module load cmake
module load gcc/10.2.0
module load nvidia/nvhpc-nompi/24.5
module load openmpi/4.1.6-cuda-12.4

module load oneapi
module load compiler
```

These modules ensure that the GNU toolchain and Intel OneAPI compilers are available, as well as the necessary MPI implementation with CUDA compatibility.

2.2 CMake Configuration

After cloning the repository (and replacing `perc_gpu.cpp` with the submitted version), create a build directory and invoke CMake with the following command:

```
cmake -S src -B build \
    -DACC_MODEL=SYCL \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_CXX_COMPILER=icpx
```

To compile, run:

```
cmake --build build -j 4
```

To submit jobs, run:

```
sbatch run-gpu-8.sh
```

3 Design

3.1 Programming Model Choice

An initial effort employed OpenMP offloading due to its straightforward integration with existing CPU-based code and simple compiler directives (i.e., `#pragma omp`). Various optimizations were attempted, such as overlapping halo exchanges with computation and reducing data transfers. However, the resulting OpenMP version tended to underperform on NVIDIA V100 GPUs compared to the original CPU code.

Consequently, the implementation was revised to use SYCL (`ACC_MODEL=SYCL`), compiled with Intel's DPC++ (`icpx`). Several factors influenced this decision:

1. **Single-Source C++:** SYCL relies on modern C++ templates, permitting simultaneous host-device code in a unified environment.
2. **Buffer-Centric Memory Management:** By controlling regions of interest through `sycl::buffer` and `accessor`, the application can synchronize only the boundaries needed for percolation iterations.
3. **Flexible Kernel Launching:** SYCL allows specifying `nd_range` kernels and offers local memory (`local_accessor`) for block-level reductions, facilitating efficient counting of changed cells.
4. **Improved Performance:** Subsequent tests indicated that the SYCL-based approach consistently outpaced the earlier OpenMP offload version on the target GPU hardware, likely due to more explicit control over data movement and kernel batching.

3.2 Algorithm and Implementation

The percolation procedure begins with a two-dimensional grid, distributed among multiple MPI processes, where each fluid site is assigned a numeric label. At every iteration, each site updates its label to the maximum of its own label and those of its four nearest neighbors, thereby capturing how fluid regions expand or merge. Convergence occurs when no further label changes are detected across the entire grid.

3.2.1 Data Structures

Each MPI rank owns a subdomain of size `(sub_nx + 2)` by `(sub_ny + 2)` (including one layer of halo cells). Two buffers, `d_state` and `d_tmp`, store the local labels on a SYCL device and allow double-buffering during updates. Additional small buffers, `d_halo_send` and `d_halo_recv`, temporarily hold boundary data for communication with neighboring ranks.

3.2.2 Halo Exchange

Before each round of updates, only the outer rows and columns of the subdomain are transferred to and from adjacent processes to ensure that each rank sees the correct

boundary values. This is achieved by:

1. Packing the boundary cells on the device side into `d_halo_send` using the `PackHaloKernel`.
2. Copying that buffer to the host and exchanging boundary elements via `MPI_Irecv` and `MPI_Isend` (or blocking calls), as shown in the `halo_exchange` function.
3. Unpacking the received data back onto the device with `UnpackHaloKernel`.

Because only a small portion of the domain is moved, host-device data transfers are minimized.

3.2.3 Batched Kernel Updates

Rather than performing a single update per kernel launch, the code merges multiple iterations into one kernel call, governed by a `batchSize` parameter. Here, a batch size of 6 has been found to strike the best balance: it lowers kernel launch overhead on larger domains while avoiding discrepancies in small-scale tests. If the batch size becomes too large, the GPU and CPU results may diverge for smaller problems due to delayed halo synchronization. The `MultiStepsKernel` illustrated in the listing iterates over each cell up to `batchSize` times. Within the kernel:

1. Each work item (thread) reads its local cell value and that of its four neighbors.
2. The maximum among these values is assigned to the cell in `d_tmp`.
3. A local counter accumulates how many cells changed, using a shared (`local_accessor`) array for partial reductions.
4. After all steps finish, a final reduction adds this block-level change count to a global atomic variable (`sum_buf`).

This approach reduces kernel launch overhead and often improves occupancy on the GPU.

3.2.4 Convergence Detection

Once the batched kernel completes, an `MPI_Allreduce` sums the local counts of changed cells across all MPI ranks, yielding a global measure of how many updates occurred. If this value is zero, the algorithm halts. Otherwise, the process repeats: a halo exchange is performed to refresh boundary data, and another batched kernel is launched to apply several more iterations.

3.2.5 Overall Flow

The main `run()` function, shown at the end of the listing, encapsulates:

- A loop that continues until convergence or a maximum iteration limit is reached.
- A `halo_exchange` call to synchronize the boundary cells among neighbors.

- A `run_batch_steps_sycl` invocation that performs the percolation steps on the device.
- An `MPI_Allreduce` operation to detect global convergence.

The local buffers `d_state` and `d_tmp` are swapped when necessary to ensure the correct final distribution of labels. This design, combining message-passing for distributed subdomains with batched GPU kernels, allows the program to scale to large grids and exploit device-level parallelism effectively.

3.3 Profiling

This section briefly summarizes Nsight Systems profiling of `build/test`, capturing both the CPU-side and GPU-side behavior when running SYCL with MPI on multiple GPUs. Figure 1 shows a timeline excerpt highlighting kernel launches, halo exchanges, and MPI communication.

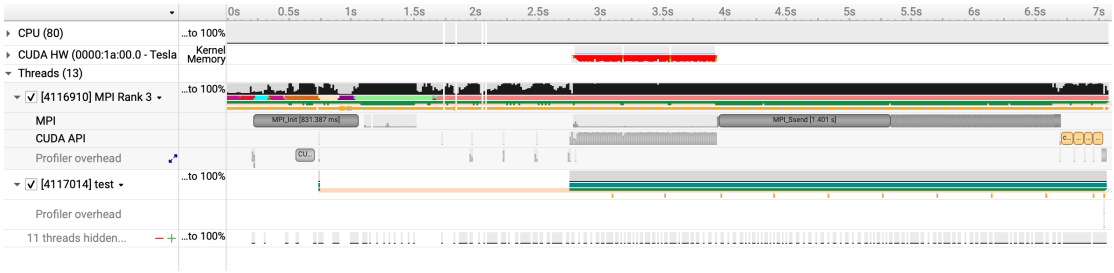

Figure 1: Snapshot of the Nsight Systems timeline.

3.3.1 Profiling Configuration and Summary

Command:

```
nsys profile --trace=mpi,cuda -o output_sycl_large \
   build/test -s 98765 -p 0.3 -M 4096 -N 4096 -P 2 -Q 2 \
   -o large-7371310.png
```

Key Observations:

- **CPU usage:** The main MPI rank thread (ID 4116910) accounts for ∼99.4% CPU utilization, reflecting the time spent managing GPU operations and MPI calls.
- **Module summary:** The OS kernel (`[kernel.kallsyms]`), `libcuda.so`, and UCX/Open MPI libraries (`mca_pml_ucx.so`, `libucp.so`) dominate CPU time, consistent with heavy message-passing and CUDA runtime activity in HPC applications.

3.3.2 Timeline and Main Performance Characteristics

1. **Initialization (0–2.5s):** SYCL GPU context creation, memory allocations, and MPI setup occupy most of the early CPU time.

2. **Kernel Execution and Halo Exchange (2.5–6.5s):**
   - CUDA kernels appear as colored segments, interleaved with `MPI_Send`/`MPI_Recv` calls for halo data.
   - Each batch of iterations (`batchSize=6`) reduces overhead by performing multiple updates per kernel launch.
   - Local memory (`local_accessor`) is used within each block for partial reductions of changed cells.
3. **Finalization (6.5–7.1s):** The application completes MPI synchronization, copies final data back from the GPU, and exits.

3.3.3 CPU-Side Usage

CPU profiling shows:

- **Kernel calls (25.82%) and CUDA runtime (18.36%):** Handling device management, scheduling, and system-level operations.
- **UCX/OMPI libraries ($\sim$30%):** Reflects frequent halo exchanges via non-blocking sends/receives for boundary data.
- **Application code (7.88%):** The percolation logic in `build/test`, with the remainder split across system libraries.

3.3.4 GPU-Side Usage

On the V100 GPU (80 SMs, $\sim$836 GiB/s):

- **Batch Updates:** Each kernel executes up to 6 steps of label updates, reducing launch overhead.
- **Halo Sync Frequency:** After each batch, halo exchange is performed; further overlap with computation may reduce idle time.
- **Thread-Block Configuration:** A tile size of `16x16` is used; more tuning could improve occupancy.

3.3.5 Potential Optimizations and Summary

From the Nsight Systems data, the main bottlenecks are the MPI halo exchanges and repeated kernel launches. The code already implements several optimizations:

1. **Batched Kernel Iterations:** Using `batchSize=6` reduces kernel launch overhead and host-device synchronization.
2. **Local Memory Reductions:** Accumulating changed-cell counts in `local_accessor` helps minimize global atomic contention.

3. **Manual Halo Packing/Unpacking:** Packing just the boundary cells into small buffers (`PackHaloKernel`, `UnpackHaloKernel`) reduces data transfers.

4 Performance Results

4.1 Experimental Setup

All measurements were performed on Cirrus, varying both the number of CPU ranks (1, 2, 4, 8) and GPU ranks (1, 4, 8). Three problem sizes were tested:

- Small: 512×512
- Median: 1024×2048
- Large: 4096×4096

The code was measured in the following scenarios:

1. **Baseline (CPU Only):** Timings collected from CPU runs with 1, 2, 4, and 8 ranks.
2. **Cold-Start (GPU):** The timing for the very first GPU run (`run0`), capturing additional overhead such as device initialization.
3. **Hot-Start (GPU):** The average timing for subsequent GPU runs (`run1` and `run2`), where device setup costs are partially amortized.

4.2 Analysis

Configuration	Small	Median	Large
`run-cpu-1`	2.117496	1.904793	1.558406
`run-cpu-2`	1.188012	1.059928	0.876951
`run-cpu-4`	0.722601	0.545608	0.442346
`run-cpu-8`	0.365303	0.293207	0.233377
`run-gpu-1`	2.091379	1.908047	1.602247
`run-gpu-4`	0.518881	0.469945	0.391592
`run-gpu-8`	0.271738	0.242286	0.199587

Table 1: Baseline timings.

Table 1 shows the baseline timings, where CPU runs (labelled `cpuX`) generally decrease with more ranks. For instance, `cpu8` achieves 0.2334 s for the large grid. Meanwhile, the GPU runs (labelled `gpuX`) also improve with increasing rank counts: `gpu8` records 0.1996 s on the large grid, which is slightly faster than `cpu8`. On the small grid, however, `gpu1` (2.09 s) and single-rank CPU (2.12 s) are close, suggesting GPU initialization overhead is not substantial in this baseline measurement.

Table 2 highlights the cold-start scenario (i.e., each configuration's first run or `run0`). Notably, `gpu1` on the small grid reaches 0.0766 s, beating its baseline time. This improvement may stem from memory allocation differences or the timing of device setup.

Configuration	Small	Median	Large
`run-cpu-1`	0.524074	0.327583	0.481552
`run-cpu-2`	0.486756	0.184010	0.254122
`run-cpu-4`	0.323775	0.138117	0.139196
`run-cpu-8`	0.365303	0.071953	0.078004
`run-gpu-1`	0.076610	0.024435	0.066958
`run-gpu-4`	0.459225	0.135032	0.112417
`run-gpu-8`	0.453679	0.073867	0.061706

Table 2: Cold-start timings.

Configuration	Small	Median	Large
`run-cpu-1`	0.479443	0.310202	0.453884
`run-cpu-2`	0.317016	0.168014	0.230706
`run-cpu-4`	0.248290	0.093910	0.116040
`run-cpu-8`	0.213926	0.061813	0.060513
`run-gpu-1`	0.061917	0.021645	0.057782
`run-gpu-4`	0.389616	0.062464	0.043022
`run-gpu-8`	0.395305	0.057773	0.028223

Table 3: Hot-start timings.

In contrast, multi-rank CPU runs also show substantial time reductions for the median and large grids, dropping to 0.0719 s and 0.0780 s respectively on `cpu8`. Meanwhile, `gpu8` achieves around 0.0617 s on the large grid, demonstrating strong GPU concurrency even at cold startup.

Table 3 reports the hot-start scenario (the second and third runs). As anticipated, GPU times are lower because the device is already active:

- **Small:** `gpu1` improves from 0.0766 s (cold) down to 0.0619 s, while `cpu1` changes from 0.5241 s to 0.4794 s.
- **Median:** `gpu4` and `gpu8` reduce to 0.0625 s and 0.0578 s, respectively, slightly outperforming `cpu8` at 0.0618 s.
- **Large:** `gpu8` reaches 0.0282 s, far exceeding `cpu8` at 0.0605 s.

Taken together, these data suggest several key observations:

- **Scaling on CPU vs. GPU:** Both CPU and GPU versions benefit from increased ranks, but GPUs often excel on larger grids due to higher concurrency.
- **Cold-Start vs. Hot-Start:** GPU initialization introduces some overhead, but once incurred, repeated runs (`hot-start`) provide consistently better performance.
- **Small vs. Large Problems:** For smaller grids, single-rank CPU and GPU are relatively close, as overhead strongly affects overall runtime. Larger problems highlight a more pronounced GPU advantage, especially at higher ranks.

In summary, multi-GPU configurations outperform their CPU counterparts at larger

scales, while single-GPU setups can match or exceed single-CPU performance once the GPU is fully initialized.

5 Discussion

5.1 Reflection on Performance

The code shows strong scaling behavior when additional ranks are introduced, both on CPUs and GPUs. On a single GPU, memory bandwidth and kernel launch overhead can sometimes limit performance, especially for smaller problem sizes. When multiple GPUs operate in parallel, communication overhead grows, but batched updates help to reduce frequent synchronization. The reduced number of kernel launches per iteration increases device occupancy and allows more efficient use of GPU resources. Still, some suboptimal performance may arise from MPI overhead, the cost of repeated halo exchanges, and the inherent latency in host-device transfers.

5.2 Comparison with Theoretical Limits

Modern GPUs like the V100 offer high theoretical memory bandwidth (over 800 GB/s) and substantial compute throughput. As a rough estimate, consider a 4096×4096 grid where each site requires reading and writing 16 bytes (4 floats of 4 bytes each). This amounts to:

$$4096 \times 4096 \times 16 \approx 268{,}435{,}456 bytes \approx 256 MB$$

Under an ideal 800 GB/s bandwidth, the fastest possible time for processing a single iteration of the entire grid would be around:

$$T_{\min} = \frac{256 MB}{800 GB/s} \approx 0.32 ms.$$

In practice, the best observed execution on a V100 for a comparable problem is tens of milliseconds. Much of this discrepancy arises from MPI halo exchanges, atomic updates, and the repeated kernel launches that percolation requires. On CPUs, memory access patterns and cache behavior add their own constraints, so achieving peak theoretical throughput is challenging. Minor tuning of block sizes, overlapping communication and computation, and more efficient data layout could improve performance, but the communication-heavy nature of this problem makes full utilization of hardware bandwidth unlikely.

5.3 Further Improvements

Possible directions for enhancing performance include:

- **Overlapping Communication and Computation:** While the code currently waits for halo exchanges to finish before launching the next kernel, some ranks could proceed with partial interior updates while halo data is still in transit.

- **Specialized CUDA or SYCL Intrinsics:** Using lower-level CUDA atomic operations or certain SYCL built-ins could accelerate the in-kernel reduction and boundary checks if these optimizations fit the SYCL model.
- **Kernel Tuning:** Adapting block sizes or experimenting with different local memory usage could improve occupancy and reduce contention during the reduction of changed cells.