ChatGPT

# Palabos-Hybrid GPU/CPU Strategy Report

## Executive Summary

- **Hybrid Data Containers:** *Palabos-Hybrid* introduces a GPU-optimized data container (`AcceleratedLattice`) alongside the traditional CPU container (`MultiBlockLattice`), preserving the API so existing codes run on either backend [1] [2]. This enables **hybrid CPU/GPU execution**: parts of the domain or algorithms not yet ported to GPU can fall back to CPU, with runtime data exchange between containers [2].
- **Domain Decomposition & MPI Reuse:** The simulation domain is decomposed into regular blocks in both CPU and GPU modes. On CPUs, many smaller blocks are assigned (one per core); on GPUs, **larger blocks** (often one per GPU) are used for efficiency [3]. Despite different granularities, Palabos **reuses the same MPI-based inter-block communication layer** for multi-block coordination on GPU as on CPU [3]. In other words, multi-GPU runs leverage the existing MPI halo-exchange infrastructure, now handling device memory buffers.
- **When (and When Not) to Use MPI:** For a **single-GPU** run, using MPI (with one rank) provides no scaling benefit – MPI can even be disabled entirely to simplify builds (UP-T3). In **multi-GPU single-node** scenarios, MPI is typically employed to launch one rank per GPU and handle boundary swaps, since Palabos does not natively spawn GPU threads across processes. In these cases MPI overhead is modest and outweighed by added GPUs' compute power [4] [5]. In **multi-node** deployments, MPI is of course required for inter-node communication, and Palabos's GPU extension was built to be multi-node capable from the start [6]. The key is to enable MPI only when parallel speedup is expected (e.g. memory or time demands exceed a single GPU's capacity), and avoid it in trivial cases where it would add latency without benefit (e.g. one GPU or one process).
- **GPU Acceleration via Standard C++:** The GPU port is realized with C++17 parallel algorithms (std::execution). This *compiler-driven offload* strategy yields performance on NVIDIA GPUs within a few percent of hand-tuned CUDA code [7], without vendor-specific languages. The maintainers achieved ~20% speedup improvement over initial stdpar attempts, narrowing the gap to native CUDA to just "a few percent" [7]. This approach emphasizes architecture-agnostic optimizations – **structure-of-arrays memory layout**, avoiding virtual function calls, and using static polymorphism – to let the compiler efficiently dispatch GPU kernels [8] [9]. The result is a codebase ~500k lines long that was ported in months, indicating the viability of high-level GPU programming [10].
- **MPI and CUDA-Awareness:** For multi-GPU runs, using a **CUDA-aware MPI** implementation is crucial. Palabos packs/unpacks halo data directly in GPU memory to enable **GPUDirect** transfers between devices [11]. A CUDA-aware MPI library can send device pointers without staging through host memory [12]. If a non-aware MPI is used, performance suffers from extra `cudaMemcpy` overhead (or may even fail if MPI isn't configured for GPU memory). The Palabos team explicitly notes that on systems with direct GPU–GPU communication, all MPI data exchange should occur in GPU memory for best performance [11].
- **Performance Scaling Characteristics:** In benchmarking, the *Palabos-Hybrid* GPU code shows strong scaling efficiency of ~80% with 2 GPUs and ~65% with 4 GPUs on a single node (DGX-A100) [4] [5]. **Weak scaling** is even more impressive: 80–90% of ideal efficiency up to 4 GPUs [13], with minimal loss per GPU even **without overlapping communication and computation** [14]. This indicates that

MPI overhead is reasonably low given the large problem sizes and use of large per-GPU blocks (the communicated-to-computed data ratio stays constant [15]). Nonetheless, overlapping halos with computation is an opportunity for future improvements (current release performs halo exchange in a post-step barrier, leaving GPUs idle during communication) [14] [16].

- **Build Strategy at a High Level:** Conceptually, building Palabos-Hybrid for performance is about **aligning the compiler and MPI choices with your target**. The GPU code requires an HPC-grade C++ compiler (tested with NVIDIA HPC SDK nvc++ 25.x [17] or equivalent) that supports stdpar offloading. The Palabos examples demonstrate two viable approaches: (1) compile Palabos as a standalone library **without MPI** for single-GPU use (relying on on-node parallelism via std::execution) (UP-T3), or (2) incorporate Palabos source into your application's build so that **Palabos and your code are compiled together with the same compiler and MPI**, ensuring ABI consistency [18] [19]. The latter is preferred for multi-node runs. Using consistent compilers and avoiding mixing (e.g. GCC-compiled Palabos with nvc++ application) prevents linker issues due to MPI C++ symbols (see below).

## CPU↔GPU Delegation Map

| Simulation Stage | Default HW | **Rationale** (Why on CPU vs GPU) | Data Movement & Interplay | Performance Optimizations |
|---|---|---|---|---|
| **Domain Partitioning & Setup** | CPU (host) | Initialization tasks (grid/block setup, MPI rank assignment) run on the CPU. This incurs negligible cost relative to the simulation runtime, and the CPU naturally orchestrates process-level setup. The **decomposition strategy differs by hardware**: many small blocks on CPU vs. a few large blocks per GPU for efficiency [3]. | Initial lattice data is allocated and, for GPU containers, copied from CPU to GPU memory at startup. Each MPI rank (process) typically manages one GPU's sub-domain. | **Larger block sizes per GPU** minimize cross-block boundaries (reducing MPI communication frequency) [3]. Domain decomposition can be tuned so that each GPU gets as contiguous a domain as possible to exploit fast on-device memory and NVLink/PCIe bandwidth. |

| **Simulation Stage** | **Default HW** | **Rationale** (Why on CPU vs GPU) | **Data Movement & Interplay** | **Performance Optimizations** |
|---|---|---|---|---|
| **Collision–Streaming Kernel** (LBM core loop) | GPU (device) *(*AcceleratedLattice*)* | The core per-cell update (collision + streaming) is computationally intensive and data-parallel. It is **offloaded to GPUs** to leverage thousands of threads, yielding order-of-magnitude speedups [20]. The GPU-accelerated lattice structure is specifically optimized (SoA memory, no virtual calls) for this stage [8] [9]. (In pure-CPU runs, this executes on CPU cores via MultiBlockLattice.) | During the time-stepping loop, lattice data stays resident in GPU global memory; no CPU-GPU transfer is needed for cells handled on GPU. Only *halo/ghost cells* at block borders are exchanged between GPUs (or between GPU and CPU blocks in hybrid mode) at synchronization points. | Use **structure-of-arrays (SoA)** layout for coalesced memory access [8] and a thread-safe two-population swap scheme to utilize GPU concurrency [9]. The code uses C++17 parallel algorithms (`std::for_each` etc. with execution policy) to let the compiler dispatch GPU kernels efficiently [20]. Ensure high compiler optimization (`-O3`, `-DNDEBUG`) for maximum throughput [21]. |

| **Simulation Stage** | **Default HW** | **Rationale** (Why on CPU vs GPU) | **Data Movement & Interplay** | **Performance Optimizations** |
|---|---|---|---|---|
| **Boundary Halo Exchange** (inter-block communication) | CPU <small>(MPI library on host CPU)</small> – **but GPU memory is used for buffers** | Neighboring sub-domains must share border data each iteration. Palabos retains its **MPI-based halo exchange** design for this purpose [22]. The MPI calls (send/recv) are initiated by the CPU, as MPI libraries execute on host, but the data being sent resides in GPU memory. By using the existing MPI layer, multi-GPU support was achieved with minimal code changes. | Each rank packs its boundary cells into buffers **in GPU memory** and calls MPI to send/receive these to neighbor ranks [23]. With CUDA-aware MPI, this transfer occurs directly GPU↔GPU (via NVLink or InfiniBand RDMA) without extra copies [11] [12]. If the MPI is not CUDA-aware, the data would be implicitly staged through host (incurring additional latency). After MPI transfer, the GPU kernels unpack data back into the local sub-domain. | **CUDA-aware MPI** and GPUDirect features are critical: they eliminate manual host staging, allowing direct device-to-device communication [11] [12]. Use MPI environments optimized for GPUs (e.g. OpenMPI built with `--with-cuda`) so that `MPI_Send/Recv` recognize device pointers. Use large message sizes (send whole-face or chunk of grid per GPU) to amortize latency. (Future optimization: overlap halo exchanges with computation to hide MPI latency, since current Palabos implementation does them sequentially [16].) |

| **Simulation Stage** | **Default HW** | **Rationale** (Why on CPU vs GPU) | **Data Movement & Interplay** | **Performance Optimizations** |
|---|---|---|---|---|
| **Unported Models / CPU-only routines** | CPU (host) *(*MultiBlockLattice*)* | Not all physics models or data processors have GPU versions yet [24]. Examples might include certain complex boundary conditions, custom user dynamics, or file I/O routines. These run on CPU using the original Palabos code. The design permits a **hybrid run** where most of the simulation is on GPU, but specific steps fall back to CPU if needed [6]. | A **bridging mechanism** copies needed data between GPU and CPU containers at runtime [2]. For instance, before a CPU-only routine, the relevant subset of the lattice is transferred from AcceleratedLattice (GPU) to MultiBlockLattice (CPU) memory, computed on the CPU, then results are copied back to GPU to continue the main simulation [2]. Such transfers are explicit in the user code (invoking Palabos API calls for data exchange). | Minimize and isolate CPU-only sections. If a needed model is frequently used, prioritize porting it to GPU to avoid repetitive transfers. When transfers are necessary, use efficient memory copies (pinned host memory if applicable) and consider frequency: e.g., do a heavy CPU analysis every N steps rather than each step. The hybrid model is a **safety net** for completeness, but peak performance comes when the GPU can handle the majority of work. |

| **Simulation Stage** | **Default HW** | **Rationale** (Why on CPU vs GPU) | **Data Movement & Interplay** | **Performance Optimizations** |
|---|---|---|---|---|
| **I/O and Post-Processing** | CPU (host) | Output operations (e.g. writing VTK files, logging, or final reductions) and some post-processing (like computing diagnostics that aren't parallelized) typically execute on the CPU. These tasks rely on libraries (filesystem, HDF5, etc.) that run on the host, and they are not usually performance-critical relative to the main simulation loop. | Data needed for output is gathered from GPUs to the host. For example, each rank might `cudaMemcpy` its lattice or a subset (e.g. a slice or summary) back to host memory and then write to disk. In multi-node runs, a further MPI gather may occur (e.g. collecting results to rank 0) or each rank writes its own files. | Overlap output with computation where possible: e.g., one can dump data asynchronously while the next time-steps compute on GPU (using streams or separate host threads). Use parallel I/O or binary formats (HDF5 parallel) to speed up writes. Also consider writing less frequently or only necessary fields to reduce data volume. Leverage high-throughput interconnects (NVMe, etc.) and **compress data on GPU** before transfer if applicable to reduce transfer size (UP-T2). |

| **Simulation Stage** | **Default HW** | **Rationale** (Why on CPU vs GPU) | **Data Movement & Interplay** | **Performance Optimizations** |
|---|---|---|---|---|
| **MPI Coordination & Overheads** | CPU (host) | Aside from data exchange, general MPI activities (initialization, finalization, barriers, reductions for diagnostics) run on the CPU. Each rank's main thread calls `MPI_Init_thread` at startup [25] and `MPI_Finalize` at the end; these set up the communication world. MPI progress is largely handled by the library internally (which may spawn its own threads if using `MPI_THREAD_FUNNELED` or `MULTIPLE`). | Control messages and synchronization involve minimal data (ints, scalars) and negligible transfer overhead. Intra-node MPI uses shared memory for small messages, while inter-node uses network (InfiniBand, etc.). These calls don't involve GPU memory directly, except that some collective ops might indirectly orchestrate GPU buffers if using `cuda-aware` support. | Ensure the MPI library is configured for multi-threading (`MPI_Init_thread` with appropriate level [25]) if the application uses threads (Palabos GPU uses multi-threading internally via stdpar, but calls into MPI only from the main thread, so `FUNNELED` is usually sufficient). Pin processes to specific CPU cores and align them with GPU affinity (use `numactl`/MPI mapping) to maximize cache locality for any host-side MPI work. Keep an eye on MPI timing in profiles – if barriers or reductions show up significantly, consider tuning their frequency or using non-blocking variants. |

## MPI Decision Tree (When to Use MPI)

```
┌────────────────────────────┐
│   Running on a GPU at all?   │
└────────────────────────────┘
              │
              │
No (CPU-only) │    Yes (GPU backend active)
              │
 (Use MPI for multi-core CPU via domain
                            decomposition or OpenMP; similar to classic Palabos
CPU scaling.)
                                        │
                                        ▼
                        ┌────────────────────────────────────┐
                        │  Will you use more than one GPU?   │
                        └────────────────────────────────────┘
                                      │
                                      │
                      One GPU (single-process run)    More than one GPU
                                      │                   (multi-GPU)
                                      │                      │
                            ┌─────────▼──────────┐            │
                            │  **Single-GPU Case**│           │
                            │ - No need for MPI   │           │
                            │   parallelism;      │           │
                            │   run in one process│           │
                            │   (MPI can be disabled for simplicity).│
│
                            │ - Rely on GPU’s internal parallelism   │
│
                            │   (std::execution or threads) to use   │
│
                            │   all device cores. [26]    │           │
                            └──────────────────────┘           │
                                      ▲                      │
                                      │                      │
              (MPI still works with 1 rank, but adds minor overhead. │
               Recommended to compile without MPI for dev convenience
(UP-T1).)   │
                                      │                      │
                                      │                      ▼
                                      │           ┌───────────────────────┐
                                      │           │ **Multi-GPU Single-**│
                                      │           │ **Node Case**         │
                                      │           │ - Use MPI to launch   │
                                      │           │   one process per GPU, so
each  │
                                      │           │   GPU gets a rank.    │
                                      │           │ - MPI handles halo    │
                                      │           │   exchange between    │
                                      │           │   GPUs via shared mem/
GPUDirect.│
                                      │           │ - Benefit: nearly linear
scaling│
                                      │           │   observed up to 4 GPUs on
one node [14] .│
```

```
                                      │            └─────────┬────────────┘
                                      │                      │
                                      │         (All GPUs in one node
still                          │
                                      │          communicate faster (NVLink/
PCIe) than across network.│
                                      │          MPI’s overhead is small
relative to GPU compute if domain is large.) │
                                      │                      │
                                      │                      ▼
                                      │            ┌──────────────────────┐
                                      │            │ **Multi-Node Cluster**│
                                      │            │ **(MPI Required)**   │
                                      │            │ - Use MPI across nodes,│
                                      │            │   typically with one rank│
                                      │            │   per GPU as well.    │
                                      │            │ - Requires fast
interconnect│
                                      │            │   (InfiniBand) and CUDA-
aware│
                                      │            │   MPI for acceptable
performance.│
                                      │            │ - Necessary when a problem’s
size│
                                      │            │   or required GPUs exceed
one machine’s.│
                                      │            └──────────────────────┘
                                      │
   Note: In multi-node or multi-GPU runs, ensure MPI library supports GPU memory
directly [11] .
         If using one multi-GPU node, you *could* run without MPI by managing
multiple GPUs in one process via threads,
         but Palabos doesn’t natively support that – it expects MPI for multi-
device scaling.
```

*(ASCII decision tree above summarizes when to engage MPI. On a single GPU, MPI-based distribution isn’t needed and can be omitted. On multiple GPUs, MPI is the default strategy: one rank per GPU, even on one node, to reuse the existing halo exchange mechanism. Across nodes, MPI is mandatory. Dashed comments indicate rationale and caveats.)*

## Principles-Based Build & Compile Guidance

- **Use a GPU-Capable Toolchain:** Ensure you compile with a C++17 compiler that supports *stdpar* GPU offloading. The Palabos team used NVIDIA HPC SDK’s **nvc++** (v25.x) with the `-stdpar` flag to enable GPU acceleration [17] . Other compilers (e.g. MSVC, GCC) currently do *not* offload standard `<execution>` algorithms to GPUs, so NVHPC is the proven choice. If using NVHPC, also pass flags to target your GPU architecture (e.g. `-gpu=cc80,cc86,...`) and enable necessary features (see below).

- **Compiler Consistency – No Mixing:** Compile **Palabos and your application together with the same compiler and MPI** to avoid ABI mismatches. Palabos's MPI interface relies on some deprecated C++ binding symbols (e.g. `MPI::Comm::Comm()`), which can cause undefined references if Palabos is compiled with one MPI implementation and your app with another [27]. In practice, this means if you are using *nvc++*, you should compile Palabos with *nvc++* as well. The Palabos GPU examples adopt this strategy by directly adding Palabos source files to the user project's CMake build [18], guaranteeing one unified compilation. This avoids issues like mixing GCC-compiled libraries with NVHPC-compiled code (which can fail due to name-mangling differences for MPI C++ symbols (UP-T3)).
- **MPI or No MPI Build:** Decide early whether you need MPI enabled. **For single-GPU or single-process usage**, it can be beneficial to **build Palabos with MPI disabled** (`-DPALABOS_ENABLE_MPI=OFF`) (UP-T1). This simplifies the build (no MPI libraries needed) and sidesteps the old MPI-C++ bindings problem entirely. It is a valid approach because on one GPU the standard algorithms (with `std::execution::par`) provide parallelism without multi-process distribution [26]. *However*, if you plan to scale out (multiple GPUs or nodes), you should build with MPI enabled and use the same MPI library consistently. A good approach for multi-node readiness is to follow the Palabos GPU example template: include Palabos source in your project and compile with `MPI_CXX_COMPILER` set to your chosen MPI's wrapper [28] – this ensures Palabos is built against the correct MPI.
- **NVHPC-Specific Flags:** When using NVIDIA's compiler, adopt the flags proven in Palabos benchmarks [19]. This includes `-std=c++20` (required C++ standard), `-stdpar` (enable parallel STL on GPU by default), and two important tweaks: `-Msingle` and `-Mfcon`. These flags manage Fortran-compatible math and single-precision consistency – they were used in Palabos tests to ensure numerical consistency across compilers [19]. Define `USE_CUDA_MALLOC` when compiling for GPU [19]; this macro in Palabos triggers usage of GPU-aware memory allocators for lattice data. *(Without it, the code might allocate host memory for certain arrays, which could lead to fallback to slow transfers. By defining it, Palabos will use* `cudaMalloc` *or unified memory for lattice arrays, as indicated in the build scripts.)* Also, if your code uses OpenMP or other threading in addition to stdpar, ensure to include `-fopenmp`/`-mp` as needed – Palabos examples show adding `-fopenmp` even if primarily using stdpar [19], likely for CPU fallbacks or TBB support.
- **MPI Library Configuration:** If MPI is enabled, use an implementation that supports CUDA. OpenMPI 4.x and MVAPICH2 2.x (built with `--with-cuda`) will recognize GPU pointers [12]. Check your MPI by running a simple CUDA-aware test (attempt an `MPI_Send` of a device pointer). If it errors or falls back to host copies, recompile or configure MPI with CUDA support. Notably, **MPI-4.0 removed the C++ bindings** that Palabos's older code expects [27]. OpenMPI still provides them in its `mpi_cxx` library (when built with `--enable-mpi-cxx`), but some vendor MPIs (e.g. NVIDIA's HPC SDK MPI, which is based on MPICH) might omit them. If you encounter linker errors about missing `MPI::` symbols, that's a sign you need to either switch MPI or disable those bindings. The simplest fix is often to use the system OpenMPI (which usually has the deprecated C++ interface available for backward compatibility [29]). In summary: **for multi-GPU, use CUDA-aware OpenMPI with C++ bindings enabled** – or avoid the issue by turning off MPI in single-GPU builds.
- **CMake and Build Process:** As Palabos is a template-heavy library, one strategy (used by the provided GPU examples) is to **build Palabos as a static library within your project** rather than installing it globally [18]. This ensures any template instantiations and compiler-specific optimizations align with your application. If you prefer a pre-built Palabos library, compile it with the exact

compiler and settings you'll use for the application. When using CMake, make sure to add `-DPLB_MPI_PARALLEL` in your compile definitions if MPI is on (the Palabos CMake does this automatically when `ENABLE_MPI` is true [30]). Also verify Palabos's main `CMakeLists.txt` recognizes your compiler – older Palabos versions didn't list NVHPC, but the hybrid pre-release has logic for `PGI`/`NVHPC` [31]. If needed, manually specify `CMAKE_CXX_COMPILER_ID` or add a dummy condition to treat NVHPC similarly to PGI (UP-T2).

- **Optimization and Debugging:** Use `-O3 -DNDEBUG` for performance runs [21], as assertions and debug checks can slow execution. For debugging GPU code, `-g -DPLB_DEBUG` can be enabled (as seen in the CMake files) to include debug symbols and Palabos's internal checks [32]. Be aware that mixing debug mode and GPU offload might expose race conditions that are otherwise masked – always test in release mode for true performance. Finally, **validate your build** by running provided examples (e.g., `examples/gpuExamples/cavity3d`) to ensure that your compiled Palabos is working correctly on GPU. Successful execution and matching output with reference data will confirm that the GPU, MPI (if used), and memory configurations are set up properly.

## Validation & Risks Checklist

To ensure the CPU↔GPU hybrid and MPI strategy is performing as intended, consider the following checks and diagnostics:

- **CUDA-Aware MPI in Action:** When running on multi-GPU or multi-node, verify that MPI is leveraging GPU-direct transfers. **Symptom of a problem:** significantly low GPU utilization or unusual CPU usage during halo exchanges, which could indicate the MPI is staging data through host. Use tools like NVIDIA *Nsight Systems* or *NVPROF* to trace memory transfers – you should see direct device-to-device (or device-to-NIC) communication rather than a series of device→host→device memcopies. OpenMPI's FAQ suggests setting `OMPI_MCA_cuda_use_gpu_mem_hooks=1` (for older versions) or ensuring UCX is used for GPU buffers [12]. If unsure, run a micro-benchmark: have rank 0 `MPI_Send` a GPU buffer to rank 1; on a properly configured setup this should work and scale with GPU PCIe/NVLink bandwidth. If it fails or falls back, reconfigure your MPI (this is a **go/no-go for multi-GPU performance**).
- **MPI Overhead vs. Compute:** Perform a scaling test on a single node: run the same simulation with 1 GPU vs 2 GPUs (half domain each) and compare total runtime. If using two GPUs doesn't yield near 1.8–1.9× speedup (for a large problem), investigate the breakdown. Profile the run with MPI profiling tools or timers around `MPI_Sendrecv` calls. Palabos's paper shows ~80% efficiency at 2 GPUs [4], so significantly lower efficiency might mean your block sizing is suboptimal or the problem size is too small (communication dominates). To mitigate overhead, increase problem size (weak scaling) or use larger blocks per GPU so that surface/volume ratio is lower.
- **Hybrid Correctness (CPU vs GPU sync):** If you utilize the hybrid mode (some operations on CPU MultiBlock, others on GPU), regularly verify that data is consistent after transfers. Palabos provides routines to copy and compare lattices [2] – use these in a debug build to ensure no data corruption when exchanging between CPU and GPU containers. A mismatch in results when toggling a model on/off GPU could reveal a bug in the ported GPU kernel or an overlooked step in the bridging process. It's wise to **compute a known invariant** (like total mass or momentum in the fluid) on both CPU and GPU copies of the lattice periodically; any divergence might signal an issue in data transfer or precision.
- **Check for GPU Utilization and Concurrency:** Use NVIDIA's *nsys* or *nvprof* to ensure that GPU kernels are indeed executing. Especially in a stdpar-based code, it's possible that if the compiler can't offload

a particular loop (due to an unsupported feature), it might fallback to CPU execution silently. Monitor the GPU activity timeline: during the collision-streaming steps, you should see kernels launched (likely with names generated by the compiler, possibly relating to `std::for_each` or Thrust). If the GPU is idle while the CPU is 100% busy, it may indicate the code is not offloading as expected (e.g. if accidentally compiled with `-stdpar=multicore` which keeps execution on CPU threads (UP-T3), or if an environment variable like `NVC_PLUS_PLUS_FORCE_CPU` is set). Ensure the compile flags are correct for GPU, and that no CPU-only fallback is getting triggered.

- **Memory Footprint & NUMA:** On multi-GPU systems, check CPU memory usage as well as GPU memory. The *two-population* scheme doubles memory for lattice populations [33] [34], so ensure each GPU has enough memory for the domain size. If running multi-node, monitor that each rank stays within its GPU's memory – an out-of-memory could trigger paging or Unified Memory thrashing. On the CPU side, MPI buffers and any host-staging (if it happens) should be pinned memory for speed. You can use `nvidia-smi` during runs to see if any GPU memory spikes or if any GPU is copying via host (the *volatile GPU util* dropping while copy engine is active would hint at host involvement). Also, ensure the process affinity is set so that each MPI rank's CPU threads run on the same NUMA node as its GPU (use `--cpu-set` or `--bind-to core` options in `mpirun` to avoid cross-numa traffic).
- **Throughput and Scaling Benchmarks:** Create a **benchmark checklist**: run small vs medium vs large domain cases on 1 GPU and 2 GPUs. The large case should scale well (weak scaling ~80–90% ideally [13]). If small cases scale poorly or even slow down on 2 GPUs, that's expected due to communication overhead dominating; document the crossover point where MPI begins to pay off. For multi-node, if possible, test on 2 nodes (each with 1–2 GPUs) to see network effects. If scaling drops significantly (e.g. <50% efficiency across nodes), investigate if GPU Direct RDMA is active on your network (some MPI require enabling CMA or a specific transport for inter-node GPUDirect). Also, test turning off CUDA-aware features (forcing host staging) as an experiment – performance should plummet, confirming that the faster path is being used when enabled.
- **MPI Synchronization and CPU usage:** Even though GPUs do the heavy lifting, the CPU side shouldn't be neglected. Monitor CPU utilization: Palabos's main thread will orchestrate MPI and launch kernels. If you see one CPU core pegged at 100% (busy-waiting on MPI or constantly launching very small kernels), you might consider batching operations or enabling MPI asynchronous progress (some MPI libraries allow background threads to handle incoming messages, via `MPI_Isend/Irecv` and `MPI_Test`). Currently, Palabos uses blocking sends/ receives for simplicity, which means the CPU waits during communication. If this becomes a bottleneck (e.g., in cases with *many* small messages), an optimization could be to patch Palabos to use non-blocking halo exchange and overlap. This is advanced, but worth noting as a potential improvement – measure and confirm if CPU-side MPI wait time is significant (profilers can attribute time to MPI calls).
- **Numerical Accuracy & Tolerance:** When moving from CPU to GPU, floating-point differences can occur. Validate the physics – e.g., run a known case (like lid-driven cavity or Poiseuille flow) on CPU-only and GPU, and compare results. The Palabos GPU paper reports identical results within machine precision for their tests, but slight differences are possible due to parallel reduction order or math library differences. If using mixed precision or if any part of the code uses single precision on GPU, check that it doesn't impair stability. Use GPU-debugging tools like `cuda-memcheck` to catch any illegal memory accesses or race conditions (especially if you modified/extended the GPU kernels).
- **Logging and Diagnostics:** Enable Palabos's built-in diagnostics when needed (`PLB_DEBUG` flag) – it might report if MPI partitions are misconfigured or if data checksums differ after communications. For MPI, you can run with `MPI_VERBOSE` or equivalent environment to see if CUDA support is being used (some MPI implementations print a message if CUDA buffers are detected, or you can set env

vars to debug the selection of GPU direct protocols). Maintain a **risk log**: e.g., *"Using MVAPICH2 2.3, saw that large messages weren't using RDMA – resolved by setting* `MV2_USE_CUDA=1`*."* This will help future troubleshooting and is good practice in HPC deployments.

## Contradictions vs. User Proposals (UP) – Evidence Check

Several hypotheses were proposed in initial analyses (labeled "UP") – here we contrast them with findings from authoritative sources:

- **UP Claim:** *"Disable MPI for single-GPU builds to avoid complexity."* – **Partially validated:** The Palabos developers don't explicitly say to turn off MPI, but given that MPI adds no benefit on one GPU, it's a reasonable simplification. Evidence from the Palabos GPU example suggests a non-MPI build was used for single-GPU runs (they link with TBB for threading, and MPI is optional) [35] [36]. The key contradiction is cautionary: while disabling MPI avoids the deprecated C++ binding issue (see below) and may simplify development (UP-T3), one must remember to re-enable MPI for multi-node scaling. In short, MPI-off builds are fine for "99% single-machine use cases" (as the user suggested, UP-T1), but they sacrifice multi-rank capability [37]. Always document that limitation so it's not forgotten.
- **UP Claim:** *"Palabos uses deprecated MPI C++ bindings, causing issues with NVHPC's MPI."* – **Confirmed by evidence:** Palabos's code indeed expects the old C++ MPI interface. We saw that NVIDIA's default MPI (based on MPICH) did not include these by default, leading to undefined symbols like `MPI::Comm::Comm()` [27]. The user's analysis noted undefined references to `ompi_mpi_comm_null` and similar (UP-T2), which aligns with this issue. The NVIDIA forum post [27] verifies that enabling the C++ bindings (via `--enable-mpi-cxx` in OpenMPI) resolves the link error. Thus, the hypothesis that "mixing compilers/MPI caused link errors" is **accurate** – our solution above (unify compiler and MPI, or disable MPI) directly addresses this. There is no contradiction; rather, it's a correct diagnosis of a real problem.
- **UP Claim:** *"Use* `-stdpar=multicore` *instead of* `-stdpar=gpu` *to avoid unsupported operations."* – **Contradicted by official guidance:** The Palabos paper and examples indicate **GPU offloading was used** (which implies `-stdpar=gpu` in NVHPC's terminology). In the example CMake, they simply use `-stdpar` without specifying, which in NVHPC defaults to GPU offload when a GPU is present [19]. The suggestion to use the *multicore* backend (which runs code on CPU threads) likely arose from a specific compiler bug or limitation encountered (UP-T3). While that might have temporarily sidestepped an error, it essentially runs the code on CPUs – negating GPU acceleration. The evidence shows Palabos runs correctly with full GPU stdpar, given proper compiler version and flags [17]. Therefore, one should **not** default to `-stdpar=multicore` unless as a last resort for debugging. Instead, update the code or compiler if an unsupported construct is hit. (E.g., Palabos avoids certain C++ features like virtual functions on GPU [38] precisely to work with stdpar.) In summary, the "multicore" switch is a workaround, not a solution: the goal is GPU execution.
- **UP Observation:** *"GPU examples rebuild Palabos from source each time to avoid compiler inconsistencies."* – **Corroborated:** The GPU example CMake confirms they compile Palabos as a static library within the example, using the same NVHPC compiler [18]. This matches the user's note (UP-T2) and is indeed a best practice we've included. There's no conflict with official sources; it's exactly how the Palabos team ensured the GPU examples run smoothly.
- **UP Hypothesis:** *"MPI adds overhead that might negate gains, we should rely on GPU parallelism for single-node."* – **Generally true, with nuance:** The user's intuition that intra-node parallelism (threads on GPU or CPU) can replace MPI for single-node cases is supported by Palabos's design [26]. They explicitly mention running an AcceleratedLattice on a multi-core CPU with threads (using C++ parallel

algorithms + TBB) as an alternative to MPI on that node [39]. This is essentially a threads+MPI hybrid approach. Our findings show MPI overhead is low relative to GPU compute for large problems [14], but for smaller jobs or development, MPI can be seen as unnecessary complexity. There's no direct contradiction; it's a matter of scale. We emphasize measuring actual overhead (as per the checklist) to make informed decisions. In practice, yes – a single GPU job doesn't need MPI, and multi-threading is sufficient to saturate that GPU.

*(In summary, the user-provided (UP) suggestions were largely in line with best practices, focusing on simplifying single-node deployment and avoiding known build pitfalls. The only misalignment was the idea of using a CPU execution mode (*`stdpar=multicore`*) in place of true GPU offload – which we advise against for production, based on the evidence that full GPU offload is both feasible and efficient with Palabos.)*

## Open Questions & Next Steps

Finally, some open questions and future exploration areas to guide ongoing development and tuning of Palabos-Hybrid:

- **Multi-Node Scaling Limits:** How well will Palabos-Hybrid scale beyond a single DGX (4 GPUs)? The current paper demonstrates strong results on 4 GPUs in one node [14], but we need data for, say, 16 GPUs across 4 nodes. Will network latency or bandwidth bottlenecks appear, and can CUDA-aware MPI (with GPUDirect RDMA) keep efficiency high? Running weak/strong scaling tests on a cluster (Infiniband-connected nodes) is an open task. If performance dips, investigate strategies like overlapping comm/compute or using topology-aware decomposition (to minimize off-node boundaries).
- **MPI vs. Threading for Multi-GPU (Single Node):** Palabos currently uses MPI ranks even on a single node for multi-GPU, but could a purely thread-based approach work to avoid MPI overhead entirely on one node? For instance, using OpenMP or std::execution with an executor that spans multiple GPUs (using CUDA MPS or unified memory) is theoretically possible. This would complicate the code significantly, but it's an open research question whether one could manage multiple GPUs within one process more efficiently than multi-process MPI does. For now, MPI is the safe route, but as heterogeneous computing advances (e.g., CUDA MPS, multi-GPU cooperative kernels), this might be worth exploring.
- **GPU-Direct Storage and I/O:** Writing output still involves copying to CPU. With technologies like GPU Direct Storage (GDS), one could potentially have GPUs write data directly to disk or via RDMA to a parallel filesystem. An open question is whether integrating such tech (perhaps via an HDF5 CUDA driver or custom I/O) could significantly speed up large-data outputs. This is especially relevant for multi-node runs where gathering data to one host is a bottleneck – writing per-GPU files in parallel and then merging might be better. We should monitor developments in GDS and see if Palabos could benefit from it in the future.
- **Unported Features Audit:** Which Palabos features remain unported to GPU? The paper hints that many collision models and processors were ported, but not all [24]. We should create a list of frequently used models (e.g. particular boundary conditions, multi-phase models, particle coupling) that currently force a CPU fallback. Those that are performance-critical should be prioritized for GPU porting. Perhaps engage with the Palabos user community to identify top needs. Also, ensure a pathway for users to contribute GPU implementations of their custom models (the paper mentions a formalism for automated migration of extensions [24] – exploring and documenting that would help advanced users).

- **Performance Counters & Diagnostics:** Introduce more built-in diagnostics to easily detect if we're hitting known slow paths. For example, Palabos could log a warning if a data exchange between MultiBlock and AcceleratedLattice happens every time-step (as that would be a performance red flag – maybe the user didn't realize a certain feature wasn't GPU-enabled). Also, implementing a timing breakdown (how much time in compute vs MPI vs transfer) that can be toggled at runtime would greatly help users optimize their runs. Open question: can we integrate something like NVTX ranges around major phases (compute, pack, MPI, unpack) so that in an Nsight Systems profile it's clear where time is spent? This would be invaluable for performance tuning.
- **Compiler and Standard Evolving:** The reliance on C++17 parallel algorithms is cutting-edge. We should watch for improvements in compilers: e.g., will GCC or Clang support stdpar for GPUs (through OneAPI or libcu++/stdpar extensions)? If so, Palabos could become runnable on AMD or Intel GPUs via SYCL or OpenMP target in the future. Ensuring the code remains standard-conforming will ease such ports. Another angle: as NVIDIA updates HPC SDK, do features like virtual function offload or better debugging become available? (The paper noted lack of virtual function support in NVHPC was a limiting factor [40] ). Keeping an eye on MPI standards too – e.g., will MPI-5 bring back a C++ interface or better support for GPU buffers? These external developments could affect Palabos strategy long-term.
- **Benchmark Variety:** So far, performance was measured on cases like Taylor-Green vortex and lid-driven cavity [41] . We should broaden the benchmarks: e.g., a multi-phase flow, or a case with *many* small sub-domains (to stress test MPI), or an irregular domain if possible. How does the hybrid approach handle, say, adaptive mesh refinement or load imbalance? (Palabos supports some adaptive grids on CPU; on GPU, large blocks are better, so perhaps less AMR focus.) Identifying any scenario where the current approach fails or performs poorly is important. This might include very small domains (where overhead is high relative to compute) or extremely memory-intensive models (where doubling memory for two-population scheme might not fit). Gathering these corner cases will guide either documentation (e.g., "for problems smaller than X^3, single GPU without MPI is better") or future code enhancements.
- **Overlapping and Pipelining:** As a future optimization, overlapping communication and computation is mentioned as not yet implemented [16] . Exploring ways to do this within the stdpar paradigm is an open question – since we don't directly control kernels, one approach is to use multiple streams and manual `cudaMemcpyAsync` for halos. Perhaps Palabos could allocate halos in pinned host memory and issue async transfers while computing interior cells on GPU (a form of temporal overlap). This is complex to implement generically, but worth investigating if we aim for scaling beyond 4–8 GPUs. A research question: can the *functional* nature of std::for_each be leveraged to split work (interior vs boundary) easily? If not, maybe a manual CUDA/Kokkos path for advanced users might be introduced. For now, it's an open area for performance improvement once baseline stability is achieved.
- **Community and Support:** Lastly, an open non-technical question: how will the Palabos-Hybrid be maintained and integrated into mainline Palabos? The GPU port was a "pre-release" side project [10] – we should follow its progress (e.g., official Palabos v2.x with GPU support). Ensuring good documentation (perhaps merging papers like the arXiv one into a user manual) and building a user base will be key for its longevity. Questions like *"Will there be support for AMD GPUs (HIP)?"* or *"Will future versions drop the old MPI C++ API?"* are worth tracking. Engaging on Palabos forums or GitLab issues for the hybrid version could provide insights.

Each of these questions points to strategic decisions: balancing portability vs. performance (GPUDirect, overlap), prioritizing development effort (port more models vs. improve comm), and aligning with evolving technology (compilers, MPI, hardware). Continuous benchmarking and user feedback will drive which path yields the best payoff for Palabos-Hybrid's mission: delivering efficient multi-GPU fluid simulations with minimal user code changes. The groundwork is strong – now it's about refining and extending it in a principled way. [14] [11]

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [13] [14] [15] [16] [17] [20] [22] [23] [24] [26] [33] [34] [38] [39] [40] [41]
Palabos_GPU.pdf
file://file-Kg9Xg9ixQ9ZHsNWrAqwjGA

[12] An Introduction to CUDA-Aware MPI | NVIDIA Technical Blog
https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

[18] [19] [21] [30] [31] [32] [35] [36] CMakeLists.txt
https://github.com/gstvbrg/palabos-hybrid-prerelease/blob/ccdf47554feee97c3b8c2c144a73e9c690518ea1/examples/gpuExamples/cavity3d/CMakeLists.txt

[25] mpiManager.cpp
https://github.com/gstvbrg/palabos-hybrid-prerelease/blob/ccdf47554feee97c3b8c2c144a73e9c690518ea1/src/parallelism/mpiManager.cpp

[27] [29] Install openmpi and compilation failed with linking mpi_cxx - GPU-Accelerated Libraries - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/install-openmpi-and-compilation-failed-with-linking-mpi-cxx/177732

[28] GPU_Critical_Learnings_ClaudeCode.md
file://file-825vYyGUuttQ9GjRHfo6Wh

[37] Clauded_Code_GPU_ARCHITECTURE_DECISION.md
file://file-Aa9KY9Eqo6XBLEUMoPE8pw