

1-1-2013

Multi-Level Parallelism for Incompressible Flow Computations on GPU Clusters

Dana A. Jacobsen
Boise State University

Inanc Senocak
Boise State University

Multi-level Parallelism for Incompressible Flow Computations on GPU Clusters

Dana A. Jacobsen^a, Inanc Senocak^{b,*}

^a*Department of Computer Science*

^b*Department of Mechanical and Biomedical Engineering*

^c*Boise State University, Boise, ID 83725*

Abstract

We investigate multi-level parallelism on GPU clusters with MPI-CUDA and hybrid MPI-OpenMP-CUDA parallel implementations, in which all computations are done on the GPU using CUDA. We explore efficiency and scalability of incompressible flow computations using up to 256 GPUs on a problem with approximately 17.2 billion cells. Our work addresses some of the unique issues faced when merging fine-grain parallelism on the GPU using CUDA with coarse-grain parallelism that use either MPI or MPI-OpenMP for communications. We present three different strategies to overlap computations with communications, and systematically assess their impact on parallel performance on two different GPU clusters. Our results for strong and weak scaling analysis of incompressible flow computations demonstrate that GPU clusters offer significant benefits for large data sets, and a dual-level MPI-CUDA implementation with maximum overlapping of computation and communication provides substantial benefits in performance. We also find that our tri-level MPI-OpenMP-CUDA parallel implementation does not offer a significant advantage in performance over the dual-level implementation on GPU clusters with two GPUs per node, but on clusters with higher GPU counts per node or with different domain decomposition strategies a tri-level implementation may exhibit higher efficiency than a dual-level implementation and needs to be investigated further.

Keywords: GPU, hybrid MPI-OpenMP-CUDA, fluid dynamics

*Corresponding author

Email addresses: dana@acm.org (Dana A. Jacobsen), senocak@boisestate.edu (Inanc Senocak)

1. Introduction

Many applications in advanced modeling and simulation require more resources than a single computing unit can provide, whether in the problem size or the required performance. Graphics processing units (GPUs) have enjoyed rapid adoption within the high-performance computing (HPC) community because GPUs enable high levels of fine-grain data parallelism. The latest GPU programming interfaces such as NVIDIA's Compute Unified Device Architecture (CUDA) [1], and more recently Open Computing Language (OpenCL) [2] provide the programmer a flexible model while exposing enough of the hardware for optimization.

Current high-end GPUs can achieve high floating point throughputs by combining highly parallel processing (200-800 scalar processing units per GPU), high memory bandwidth and efficient thread scheduling. GPU clusters, where fast network connected compute-nodes are augmented with latest GPUs, [3] are now being used to solve challenging problems from various domains. Examples include the 384 GPU Lincoln Tesla cluster operated by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign [4] and the 512 GPU Longhorn cluster at the Texas Advanced Computing Center (TACC). Latest supercomputers too allow large numbers of GPUs to be used to solve single problems. Examples include the 7168 GPU Tianhe-1A [5, 6] and the 4640 GPU Dawning Nebulae [7] supercomputers. These new systems are designed for high performance as well as high power efficiency, which is a crucial factor in future exascale computing [8].

2. Related Works

GPU computing has evolved from hardware rendering pipelines that were not amenable to non-rendering tasks, to the modern General Purpose Graphics Processing Unit (GPGPU) paradigm. Owens et al. [9] survey the early history as well as the state of GPGPU computing up to 2007. The use of GPUs for Euler solvers and incompressible Navier-Stokes solvers has also been well documented [10–17].

Modern motherboards can accommodate multiple GPUs in a single workstation with several TeraFLOPS of peak performance, but GPU programming models have to be interleaved with MPI, OpenMP or Pthreads to make

35 use of all the GPUs in computations. In the multi-GPU computing front,
36 Thibault and Senocak [15, 16] developed a single-node multi-GPU 3D incom-
37 compressible Navier-Stokes solver with a Pthreads-CUDA implementation. The
38 GPU kernels from their study forms the internals of the present cluster im-
39 plementation. Thibault and Senocak demonstrated a speedup of $21\times$ for two
40 Tesla C870 GPUs compared to a single core of an Intel Core 2 E8400 3.0 GHz
41 processor, $53\times$ for two GPUs compared to an AMD Opteron 8216 2.4 GHz
42 processor, and $100\times$ for four GPUs compared to the same AMD Opteron
43 processor. Four GPUs were able to sustain $3\times$ speedup compared to a single
44 GPU on a large problem size. The multi-GPU implementation of Thibault
45 and Senocak does not overlap computation with GPU data exchanges. There-
46 fore, three overlapping strategies are systematically introduced and evaluated
47 in the present study.

48 Micikevicius [18] describes both single and multi GPU CUDA implemen-
49 tations of a 3D 8th-order finite difference wave equation computation. The
50 wave equation code is composed of a single kernel with one stencil opera-
51 tion, unlike CFD computations which consist of multiple inter-related kernels.
52 MPI was used for process communication in multi-GPU computing. Micike-
53 vicius uses a two stage computation where the cells to be exchanged are com-
54 puted first, then the inner cells are computed in parallel with asynchronous
55 memory copy operations and MPI exchanges. With efficient overlapping of
56 computations and copy operations, Micikevicius achieves very good scaling
57 on 4 GPUs running on two Infiniband connected nodes with two Tesla 10-
58 series GPUs each, when using a large enough dataset.

59 Goddeke et al. [12] explored coarse and fine grain parallelism in a finite
60 element model for fluids or solid mechanics computations on a GPU cluster.
61 Goddeke et al. [19] described the application of their approach to a large-scale
62 solver toolkit. The Navier-Stokes simulations in particular exhibited limited
63 performance due to memory bandwidth and latency issues. Optimizations
64 were also found to be more complicated than simpler models such as the ones
65 they previously considered. While the small cluster speedup of a single kernel
66 is good, unfortunately acceleration of the entire model is only a modest factor
67 of two. Their model uses a nonuniform grid and multigrid solvers within a
68 finite element framework for relatively low Reynolds numbers.

69 Phillips et al. [20] describe many of the challenges that arise when imple-
70 menting scientific computations on a GPU cluster, including the host/device
71 memory traffic and overlapping execution with computation. A performance
72 visualization tool was used to verify overlapping of CPU, GPU, and commu-

73 nication on an Infiniband connected 64 GPU cluster. Scalability is noticeably
74 worse for the GPU accelerated application than the CPU application as the
75 impact of the GPU acceleration is quickly dominated by the communication
76 time. However, the speedup is still notable. Phillips et al. [21] describe a
77 2D Euler Equation solver running on an 8 node cluster with 32 GPUs. The
78 decomposition is 1D, but GPU kernels are used to gather/scatter from linear
79 memory to non-contiguous memory on the device.

80 While MPI is the API typically used for network communication between
81 compute nodes, it presents a distributed memory model which can poten-
82 tially make it less efficient for processes running on the same shared-memory
83 compute node [22, 23]. For this reason, hybrid programming models com-
84 bining MPI and a threading model such as OpenMP or Pthreads have been
85 proposed with the premise that message passing overhead can be reduced,
86 increasing scalability. With two to four GPUs per compute node, a hybrid
87 MPI-OpenMP-CUDA method warrants further investigation and is studied
88 in this paper along with an MPI-CUDA method to develop a multi-level
89 parallel incompressible flow solver for GPU clusters.

90 Cappello, Olivier, and Etiemble [24–26] were among the first to present
91 the hybrid programming model of using MPI in conjunction with a thread-
92 ing model such as OpenMP. They demonstrated that it is sometimes possible
93 to increase efficiency on some codes by using a mixture of shared memory
94 and message passing models. A number of other papers followed with the
95 same conclusions [27–34]. Many of these papers also point out a number
96 of cases where the applications or computing systems are a poor fit to the
97 hybrid model, and in some cases performance decreases. Lusk and Chan
98 [35] describes using OpenMP and MPI for hybrid programming on three
99 cluster environments, including the effect the different models have on com-
100 munication with the NAS benchmarks. They claim combination of MPI and
101 OpenMP parallel programming is well fitted to modern scalable high perfor-
102 mance systems.

103 Hager, Jost, and Rabenseifner [36] give a recent perspective on the state
104 of the art techniques in hybrid MPI-OpenMP programming. Particular at-
105 tention is given to mapping the model to domain decomposition as well as
106 overlapping methods. Results with hybrid models of the BT-MZ benchmark
107 (part of the NAS Parallel Benchmark suite) on a Cray XT5 using a hybrid
108 approach showed similar performance at 64 and fewer cores, but greatly im-
109 proved results for 128, 256, and 512 cores, where a good combination of
110 OpenMP fine-grain parallelism combined with MPI coarse-grain parallelism

111 can be found that matches well with the hardware. These examples also
112 take advantage of the loop scheduling features in OpenMP. Advantages in
113 fine grain parallelism like this will not be able to be taken advantage of
114 in a model where OpenMP is only used for coarse-grain data transfer and
115 synchronization.

116 Balaji et al. [37] discuss issues arising from using MPI on petascale ma-
117 chines with close to a million processors. A number of irregular MPI collec-
118 tive operations are considered to be nonscalable when applied to a very large
119 number of processes. The tested MPI implementations also allocate some
120 memory which is proportional to the number of processes, limiting scalabil-
121 ity. These as well as other limitations lead Balaji et al. to suggest a hybrid
122 threading / MPI model as one way to mitigate the issue. However, we think,
123 in the case of a typical GPU system the situation is not as bad. Because
124 the CUDA model for fine-grain parallelism manages 256 to 512 processing
125 elements within a single process, and this number will likely increase with
126 future GPUs. Hence a one million processing element GPU cluster using
127 just MPI-CUDA may have fewer than 4000 MPI processes. This suggests
128 that clusters enhanced with GPUs look well suited for petascale and emerg-
129 ing exascale architectures. Therefore, compute-intensive applications need to
130 be evaluated for parallel efficiency and performance on large GPU clusters.
131 Our study is one of few that critically evaluates multi-level parallelism of
132 incompressible flow computations on GPU clusters.

133 Nakajima [38] describes a three-level hybrid method using MPI, OpenMP,
134 and vectorization. This approach uses MPI for inter-node communication,
135 OpenMP for intra-node communication, and parallelism within the node via
136 the vector processor. It closely matches the rationale behind our hybrid MPI-
137 OpenMP-CUDA approach for a GPU cluster implementation. Nakajima's
138 weak scaling measurements showed worse results for 64 and fewer SMP nodes,
139 but improved with 96 or more. GPU clusters with 128 or more compute-
140 nodes (256 or more GPUs) are rare at this time, but trends indicate these
141 machines will become far more common in the high performance computing
142 field [6–8].

143 While these articles show some potential benefits for using the hybrid
144 model on CPU clusters, a question is whether the same benefits will ac-
145 crue to a tri-level CUDA-OpenMP-MPI model, and whether the benefits
146 will outweigh the added software complexity. With high levels of data par-
147 allelism on the GPU, separate memory for each GPU, low device counts per
148 node, and currently small node counts, the GPU cluster model has numer-

149 ous differences from dense-core CPU clusters. In this paper we investigate
150 several methods of distributing computation using a dual (MPI-CUDA) and
151 tri-level (MPI-OpenMP-CUDA) parallel programming approaches along with
152 different strategies to overlap computation and communication on GPU clus-
153 ters. We adopt MPI for coarse-grain inter-node communication, OpenMP
154 for medium-grain intra-node communication in the tri-level approach, and
155 CUDA for fine-grain parallelism within the GPUs. In all of our implementa-
156 tions, computations are entirely done on the GPU using CUDA. We use a 3D
157 incompressible flow Navier-Stokes solver to systematically assess scalability
158 and performance of multi-level parallelism on large GPU clusters.

159 3. Governing Equations and Numerical Approach

160 Navier-Stokes equations for buoyancy driven incompressible fluid flows
161 can be written as follows:

$$\nabla \cdot \mathbf{u} = 0, \quad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla P + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \quad (2)$$

163 where \mathbf{u} is the velocity vector, P is the pressure, ρ is the density, ν is the
164 kinematic viscosity, and \mathbf{f} is the body force. The Boussinesq approximation,
165 which applies to incompressible flows with small temperature variations, is
166 used to model the buoyancy effects in the momentum equations [39]:

$$\mathbf{f} = \mathbf{g} \cdot (1 - \beta(T - T_\infty)), \quad (3)$$

167 where \mathbf{g} is the gravity vector, β is the thermal expansion coefficient, T is the
168 calculated temperature at the location, and T_∞ is the steady state tempera-
169 ture.

170 The temperature equation can be written as [40, 41]

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \alpha \nabla^2 T + \Phi, \quad (4)$$

171 where α is the thermal diffusivity and Φ is the heat source.

172 The buoyancy-driven incompressible form of the Navier-Stokes equations
173 (Eqs. 1–4) do not have an explicit equation for pressure. Therefore, we use
174 the projection algorithm of Chorin [42], where the velocity field is first pre-
175 dicted using the momentum equations without the pressure gradient term.

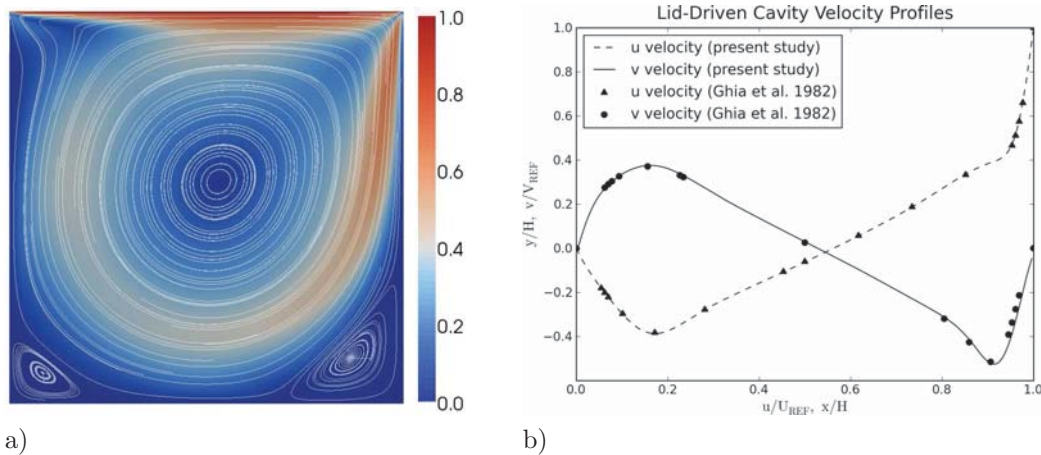


Figure 1. Lid-driven cavity simulation with $Re = 1000$ on a $256 \times 32 \times 256$ grid. 3D computations were used and a 2D center slice is shown. a) Velocity streamlines and velocity magnitude distribution. b) Comparison to the benchmark data from Ghia et al. [44].

176 The resulting predicted velocity field does not satisfy the divergence free condition.
 177 The divergence free condition is then enforced on the velocity field
 178 at time $t + 1$, to derive a pressure Poisson equation from the momentum
 179 equations given in Eq. (2). We solve the discretized versions of the resulting
 180 equations on a uniform Cartesian staggered grid with second order central
 181 difference scheme for spatial derivatives and a second order accurate Adams-
 182 Bashforth scheme for time derivatives. The pressure Poisson equation can
 183 be solved using either a fixed iteration Jacobi solver or a parallel geometric
 184 multigrid solver [43]. Both solvers are available in our code. We do not
 185 activate the geometric multigrid solver in certain computations where we in-
 186 vestigate dual- and tri-level parallelism, because the amalgamated parallel
 187 implementation of the multigrid method complicates the detailed analysis of
 188 scaling and breakdown of communication timings due to the inherent algo-
 189 rithmic complexity in the method.

190 Validation on a number of test cases including the well-known lid-driven
 191 cavity and natural convection in heated cavity problems [44, 45] were used
 192 to compare the overall solutions to known results. Figure 1 presents the
 193 results of a lid-driven cavity simulation with a Reynolds number 1000 on a
 194 $256 \times 32 \times 256$ grid. Figure 1a shows the velocity magnitude distribution

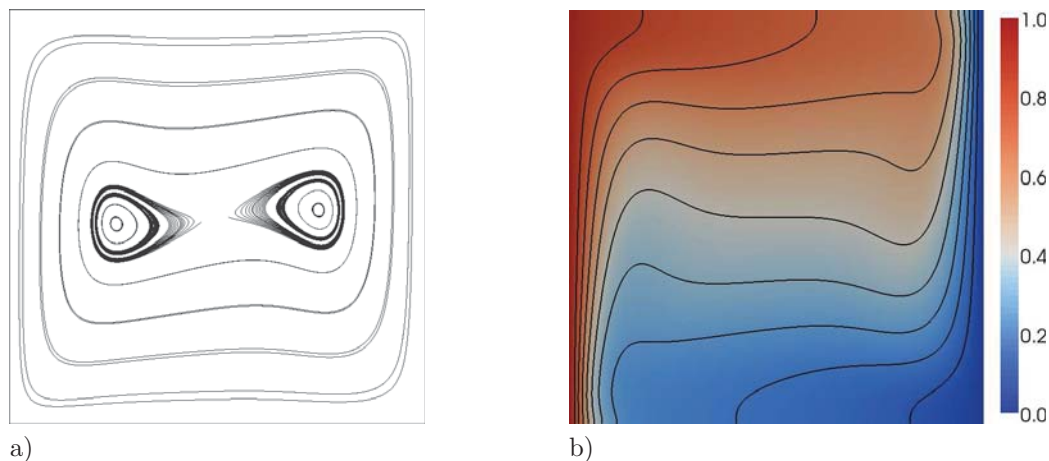


Figure 2. Natural convection in a cavity using a $128 \times 16 \times 128$ grid and Prandtl number 7, with a 2D center slice shown. a) Streamlines for Rayleigh number 200,000. b) Isotherms and temperature distribution for Rayleigh number 200,000.

195 and streamlines at mid-plane. As expected, the computations capture the
196 two corner vortices at steady-state. In Fig. (1b), the horizontal and vertical
197 components of the velocity along the centerlines are compared to the bench-
198 mark data of Ghia et al. [44]. The results agree well with the benchmark
199 data. The numerical results for the tri-level and dual-level parallel versions
200 do not differ.

201 We simulate the natural convection in a heated cavity problem to test our
202 buoyancy-driven incompressible flow computations on a $128 \times 16 \times 128$ grid.
203 Figure 2 presents the natural convection patterns and isotherms for Rayleigh
204 (Ra) numbers of 200,000 and a Prandtl (Pr) number of 7.0. Lateral walls
205 have constant temperature boundary conditions with one of the walls having
206 a higher temperature than the wall on the opposite side. Top and bottom
207 walls are insulated. Fluid inside the cavity is heated on the hot lateral wall
208 and rises due to buoyancy effects, whereas on the cold wall it cools down
209 and sinks, creating a circular convection pattern inside the cavity. Although
210 not shown in the present paper, our results agree well with similar results
211 presented in Griebel et al. [40]. A direct comparison is available in Jacobsen
212 [17]. Figure 3 presents a comparison of the horizontal centerline temperatures
213 for a heated cavity with $Ra=100,000$ and $Pr=7.0$ along with reference data

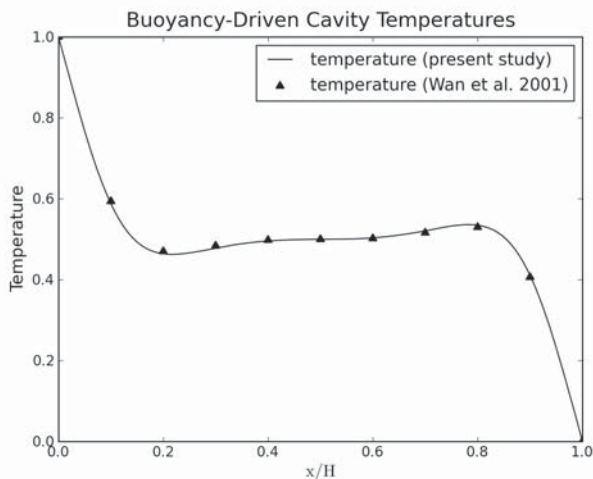


Figure 3. Centerline temperature for natural convection in a cavity with Prandtl number 7 and Rayleigh number 100,000, using a $256 \times 16 \times 256$ grid with a 2D center slice used. Comparison is shown to data from Wan et al. [45].

214 from Wan et al. [45]. Our results are in very good agreement.

215 Aside from these benchmark cases, our CFD solver can compute flow
216 around embedded obstacles such as urban areas and complex terrain can be
217 found in [17, 46, 47]

218 4. Multi-level Parallelism

219 Multiple programming APIs along with a domain decomposition strat-
220 egy for data-parallelism is required to achieve high throughput and scalable
221 results from a CFD model on a multi-GPU platform. For problems that
222 are small enough to run on a single GPU, overhead time is minimized as
223 no GPU/host communication is performed during the computation, and all
224 optimizations are done within the GPU code. When more than one GPU
225 is used, cells at the edges of each GPU's computational space must be com-
226 municated to the GPUs that share the domain boundary so they have the
227 current data necessary for their computations. Data transfers across the
228 neighboring GPUs inject additional latency into the implementation which
229 can restrict scalability if not properly handled. For these reasons we investi-
230 gate multi-level parallelism on GPU clusters with different implementations

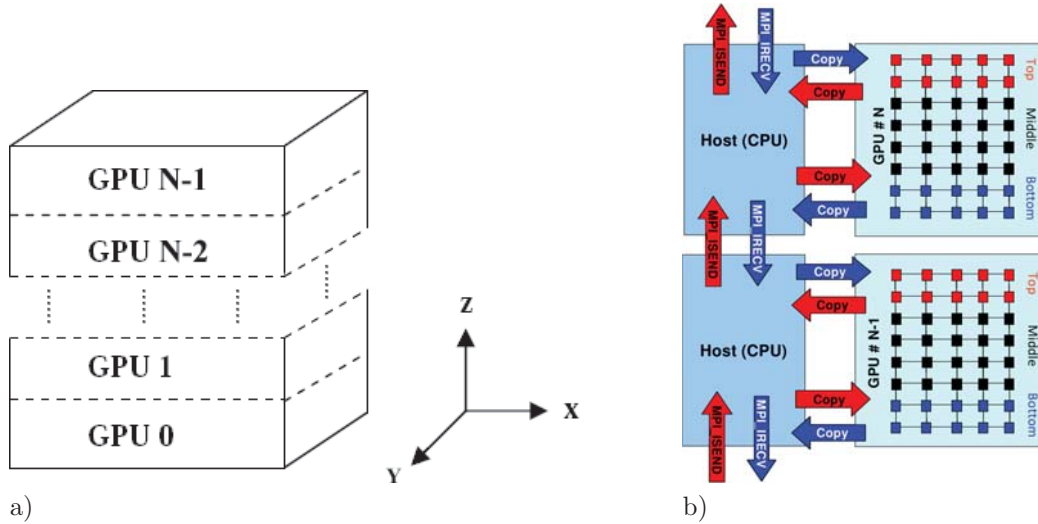


Figure 4. The domain decomposition. a) The decomposition of the full computational domain to the individual GPUs. b) An overview of the communication, GPU memory transfers, and the intra-GPU 1D decomposition used for overlapping.

231 to improve the performance and scalability of our Navier-Stokes solver.

232 4.1. Domain Decomposition

233 A 3D Cartesian volume is decomposed into 1D slices. These slices are
 234 then partitioned among the GPUs on the cluster to form a 1D domain de-
 235 composition. The 1D decomposition is shown in Figure 4a. After each GPU
 236 completes its computation, the edge cells (“ghost cells”) must be exchanged
 237 with neighboring GPUs. Efficiently performing this exchange process is crucial
 238 to cluster scalability as we demonstrate in section 5.

239 While a 1D decomposition leads to more data being transferred as the
 240 number of GPUs increases, there are advantages to the method when using
 241 CUDA. In parallel CPU implementations, host memory access can be
 242 performed on non-contiguous segments with a relatively small performance
 243 loss. The `MPI_CART` routines supplied by MPI allow efficient management of
 244 virtual topologies, making the use of 2D and 3D decompositions easy and
 245 efficient. In contrast, the CUDA API only provides a way to transfer linear
 246 segments of memory between the host and the GPU. Hence, 2D or 3D de-
 247 compositions for GPU implementations must either use nonstandard device

248 memory layouts which may result in poor GPU performance, or run separate
249 kernels to perform gather/scatter operations into a linear buffer suitable for
250 the `cudaMemcpy()` routine. These routines add significant time and hinder
251 overlapping methods. For these reasons, the 1D decomposition was deemed
252 best for moderate size clusters such as the ones used in this study.

253 To accommodate overlapping, a further 1D decomposition is applied
254 within each GPU. Figure 4b indicates how the 1D slices within each GPU
255 are split into a top, bottom, and middle section. When overlapping commu-
256 nication and computation, the GPU executes each separately such that the
257 memory transfers and MPI communication can happen simultaneously with
258 the computation of the middle portion.

259 *4.2. Dual-Level MPI-CUDA Implementations*

260 The work by Thibault and Senocak [15, 16] showed how an incompressible
261 Navier-Stokes solver written for a single GPU can be extended to multiple
262 GPUs by interleaving CUDA with Pthreads. The full 3D domain is decom-
263 posed across threads in one dimension, splitting on the Z axis. The resulting
264 partitions are then solved using one GPU per thread. No effort was made
265 to hide latencies arising from GPU data transfers or Pthreads synchroniza-
266 tion. To solve the restrictions of the shared memory model of Thibault and
267 Senocak, we adopt MPI as the mechanism for communication between GPUs,
268 and introduce three strategies to overlap computations on the GPU with data
269 copying to and from the GPU and MPI communication across the network.

270 In our present implementation, a single MPI process is started per GPU,
271 and each process is responsible for managing its GPU and exchanging data
272 with its neighbor processes. Since we must ensure that each process is as-
273 signed a unique GPU identifier, an initial mapping of hosts to GPUs is per-
274 formed. A master process gathers all the host names, assigns GPU identifiers
275 to each host such that no process on the same host has the same identifier,
276 and scatters the result back. At this point the `cudaSetDevice()` call is made
277 on each process to map one of the GPUs to the process which assures that
278 no other process on the same node will map to the same GPU. All ghost cell
279 exchanges are done via `MPI_Isend` and `MPI_Irecv`. Overlap of computations
280 with inter-node and intra-node data exchanges is accomplished to better uti-
281 lize the cluster resources. All three of the implementations have much in
282 common, with differences in the way data exchanges are implemented. It is
283 shown in section 5 that implementation details in the data exchanges have a
284 large impact on performance.

This is an author-produced, peer-reviewed version of this article. The final, definitive version of this document can be found online at *Parallel Computing*, published by Elsevier. Copyright restrictions may apply. doi: 10.1016/j.parco.2012.10.002

```
for (t=0; t < time_steps; t++)
{
    adjust_timestep();

    for (stage = 0; stage < num_timestep_stages; stage++) {
        temperature <<<grid,block>>> (u,v,w,phiold,phi,phinew);
        ROTATE_POINTERS(phi,phinew);
        temperature_bc <<<grid,block>>> (phi);
        EXCHANGE(phi);

        turbulence <<<grid,block>>> (u,v,w,nu);
        turbulence_bc <<<grid,block>>> (nu);
        EXCHANGE(nu);

        momentum <<<grid,block>>> (phi,uold,u,unew,vold,v,vnew,wold,w,wnew);
        momentum_bc <<<grid,block>>> (unew,vnew,wnew);
        EXCHANGE(unew,vnew,wnew);
    }

    divergence <<<grid,block>>>(unew,vnew,wnew,div);

    // Iterative or multigrid solution
    pressure_solve(div,p,pnew);

    correction <<<grid,block>>> (unew,vnew,wnew,p);
    momentum_bc <<<grid,block>>> (unew,vnew,wnew);
    EXCHANGE(unew,vnew,wnew);
    ROTATE_POINTERS(u,unew); ROTATE_POINTERS(v,vnew); ROTATE_POINTERS(w,wnew);
}
```

Listing 1. Host code for the projection algorithm to solve buoyancy driven incompressible flow equations on multi-GPU platforms. The **EXCHANGE** step updates the ghost cells for each GPU with the contents of the data from the neighboring GPU.

```
// PART 1: Interleave non-blocking MPI calls with device
//          to host memory transfers of the edge layers.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north)

// ... other exchanges may be started here, before finishing in order

// PART 2: Once MPI indicates the ghost layers have been received,
//          perform the host to device memory transfers.

MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)
```

Listing 2. An EXCHANGE operation overlaps GPU memory copy operations with asynchronous MPI calls for communication.

285 The projection algorithm is composed of distinct steps in the solution
286 of the fluid flow equations. Listing 1 shows an outline of the basic imple-
287 mentation using CUDA kernels to perform each step. The steps marked as
288 EXCHANGE are where ghost cells for each GPU are filled in with the calculated
289 contents of their neighboring GPUs. The most basic exchange method is to
290 call `cudaMemcpy()` to copy the edge data to host memory, MPI exchange us-
291 ing `MPI_Send` and `MPI_Recv`, and finally `cudaMemcpy()` to copy the received
292 edge data to device memory. This is straightforward, but all calls are block-
293 ing which greatly hinders performance. Therefore, we have not pursued this
294 basic implementation in the present study.

295 4.2.1. Non-blocking MPI with No Overlapping of Computation

296 The first implementation uses non-blocking MPI calls [50] to offer a sub-
297 stantial benefit over the blocking approach, which we do not pursue. Our
298 first implementation does not overlap computation although it tries to over-

299 lap memory copy operations. The basic **EXCHANGE** operation is shown in
300 Listing 2. In this approach, none of the device/host memory operations nor
301 any MPI communication happens until the computation of the entire domain
302 has completed. The MPI communication is able to overlap with the CUDA
303 memory operations. When multiple arrays need to be exchanged, such as the
304 three momentum components, the components may be interleaved such that
305 the MPI send and receive for one edge of the first component is in progress
306 while the memory copy operations for the later component are proceeding.
307 This is done by starting part 1 for each component in succession, then part
308 2 for each component.

309 *4.2.2. Overlapping Computation with MPI Communications*

310 The second implementation for exchanges aims to overlap the CUDA
311 computation with the CUDA memory copy operations and the MPI com-
312 munication. We split the CUDA kernels into three calls such that the edges
313 can be done separately from the middle. This has a very large impact on
314 the cluster performance as long as the domain is large enough to give each
315 GPU enough work to do. The body of the pressure kernel loop when using
316 this method is shown in Listing 3. Rather than perform the computation
317 on the entire domain before starting the exchange, the kernel is started with
318 just the edges being computed. The first portion of the previously shown
319 non-blocking MPI **EXCHANGE** operation is then started, which does device
320 to host memory copy operations followed by non-blocking MPI communica-
321 tions. The computation on the middle portion of the domain can start as
322 soon as the edge layers have finished transferring to the host, and operates
323 in parallel with the MPI communication. The last part of the non-blocking
324 MPI **EXCHANGE** operation is also identical and is run immediately after the
325 middle computation is started. While this implementation results in signifi-
326 cant overlap, it is possible to improve on it by overlapping the computation
327 of the middle portion with the memory transfer of the edge layers as shown
328 in the final implementation.

329 *4.2.3. Overlapping Computation with MPI Communications and GPU Trans-* 330 *fers*

331 The final implementation is enabled by CUDA streams, and uses asyn-
332 chronous methods to start the computation of the middle portion as soon
333 as possible, thereby overlapping computation, memory operations, and MPI
334 communication. A similar approach is described in Micikevicius [18]. This


```
// The GPU domain is decomposed into three sections:
//   (1) top edge, (2) bottom edge, and (3) middle
// Which of them the kernel should process is indicated
// by a flag given as an argument.

pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

// The cudaMemcpy calls below will not start until
// the previous kernels have completed.
// This is identical to part 1 of the EXCHANGE operation.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north);

pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

// This is identical to part 2 of the EXCHANGE operation.
MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)

pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 3. An example Jacobi pressure loop, showing how the CUDA kernel is split to overlap computation with MPI communication.

```
pressure <<<grid_edge,block, stream[0]>>> (edge_flags, div,p,pnew);
// Ensure the edges have finished before starting the copy
cudaThreadSynchronize();

cudaMemcpyAsync(south edge layer from device to host, stream[0])
cudaMemcpyAsync(north edge layer from device to host, stream[1])

pressure <<<grid_middle,block, stream[2]>>> (middle_flag, div,p,pnew);

MPI_Irecv(new ghost layer from north)
cudaStreamSynchronize(stream[0]);
MPI_Isend(south edge layer to south)

MPI_Irecv(new ghost layer from south)
cudaStreamSynchronize(stream[1]);
MPI_Isend(north edge layer to north);

MPI_Wait(south receive to complete)
cudaMemcpyAsync(new south ghost layer from host to device, stream[0])
MPI_Wait(north receive to complete)
cudaMemcpyAsync(new north ghost layer from host to device, stream[1])

// Ensure all streams are done, including copy operations and computation
cudaThreadSynchronize();
pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 4. CUDA streams are used to fully overlap computation, memory copy operations, and MPI communication in the pressure loop.

335 method has the highest amount over overlapping, and is expected to have
336 the best performance at large scales. The body of the pressure kernel loop
337 when using this method is shown in Listing 4.

338 It is important to note that the computations inside the CUDA kernels
339 need minimal change, and the same kernel can be used for all three imple-
340 mentations. A flag is sent to each kernel to indicate which portions (top,
341 bottom, middle) it is to compute, along with an adjustment of the CUDA
342 grid size so the proper number of GPU threads are created. Since GPU
343 kernels tend to be highly optimized, minimizing additional changes in kernel
344 code is desirable.

345 4.3. Tri-Level MPI-OpenMP-CUDA Implementation

346 GPU cluster nodes are becoming denser with multiple GPUs per node
347 [51]. Therefore we add a threading model to investigate whether additional
348 efficiency can be gained from removing redundant message passing when
349 processes are on the same host and communication and synchronization are
350 handled by a hybrid MPI-OpenMP model. The effectiveness of this solution
351 depends on a number of factors, with some barriers to effectiveness being:

- 352 • *Density of nodes.* With more GPUs per node, the potential effective-
353 ness can be increased. Only clusters with two GPUs per node were
354 available for the present study.
- 355 • *MPI implementation efficiency.* The OpenMPI 1.3.2 software on the
356 NCSA Lincoln Tesla cluster seems reasonably well optimized. Goglin
357 [52] discusses optimizations of MPI implementations to improve intra-
358 node efficiency. A number of optimizations have been performed on
359 MPI implementations since the early hybrid model papers were writ-
360 ten, including a reduction in the number of copies involved. Since the
361 application being studied only uses OpenMP and MPI for coarse-grain
362 parallelism, any benefits in latency for small transactions will not have
363 an impact.
- 364 • *A large number of nodes.* Many of the hybrid model papers note ben-
365 efits occurring only as the number of nodes grows [26, 36, 38]. While
366 the 64-node 128-GPU implementation used in this study is larger than
367 many published cluster results, it may still be too small to see an ap-
368 preciable benefit.

- 369 • *A good match between the hardware, the threading models, and the do-*
370 *main decomposition.* A number of hybrid model papers show applica-
371 tion / hardware combinations that show reduced performance with the
372 hybrid model [26, 28, 30, 35].
- 373 • Interactions between OpenMPI, OpenMP, and CUDA can exist. For
374 instance, the default OpenMPI software on the NCSA Lincoln Tesla
375 cluster is compiled without threading support.

376 There are two popular threading models in use today: POSIX Threads
377 (Pthreads) and OpenMP. We consider OpenMP, because it has become the
378 dominant method for shared memory parallelism in the HPC community. In
379 our implementation the thread level parallelism is on a coarse grain level,
380 since CUDA is handling the fine grain parallelism. We do not consider a
381 more general approach where OpenMP can be used to perform some of the
382 computations on multi-core CPUs in addition to computations on the GPU.

383 MPI defines four levels of thread safety: **SINGLE**, where only one thread
384 is allowed. **FUNNELED** is the next level, where only a single master thread
385 on each process may make MPI calls. The third level, **SERIALIZED**, allows
386 any thread to make MPI calls, but only one at a time is using MPI. Finally,
387 **MULTIPLE** allows complete multithreaded operation, where multiple threads
388 can simultaneously call MPI functions.

389 With many clusters having pre-installed versions of MPI libraries, some-
390 times with custom network infrastructure, it is not always possible to have
391 access to the highest (**MULTIPLE**) threading level. Additionally, this level
392 of threading support typically comes with some performance loss, so lower
393 levels are preferred if they do not otherwise hinder parallelism [53]. Three
394 implementations were created, using the **SERIALIZED**, **FUNNELED**, and **SINGLE**
395 levels. The first implementation used one thread per GPU, with each thread
396 responsible for any possible MPI communications with neighboring nodes.
397 The second used $N + 1$ threads for N GPUs, where a single thread per node
398 handles all MPI communications and the other threads manage the GPU
399 work. This can help alleviate resource contention between MPI and GPU
400 copies, since each activity is on its own thread. Additionally this lets one use
401 the **FUNNELED** level, which increases portability and possibly can increase per-
402 formance. Lastly, the third version uses OpenMP directives to only perform
403 MPI calls inside single-threaded sections.

404 Similar to the dual-level MPI-CUDA testing, simulation runs were per-
405 formed on the NCSA Lincoln Tesla cluster for the tri-level parallel implemen-

```
// COMPUTE EDGES
if (threadid > 0)
    pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

#pragma omp single
{
    MPI_Irecv(new ghost layer from north)
}
if (threadid > 0)
    cudaMemcpy(south edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
    MPI_Isend(south edge layer to south)
    MPI_Irecv(new ghost layer from south)
}
if (threadid > 0)
    cudaMemcpy(north edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
    MPI_Isend(north edge layer to north)
}

// COMPUTE MIDDLE
if (threadid > 0)
    pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

#pragma omp single
{
    MPI_Wait(new ghost layer from north)
    MPI_Wait(new ghost layer from south)
}
// Ensure all threads wait for MPI communication
#pragma omp barrier
if (threadid > 0) {
    cudaMemcpy(new north ghost layer from host to device)
    cudaMemcpy(new south ghost layer from host to device)
}
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
    MPI_Waitall(south and north sends, allowing buffers to be reused)
}

if (threadid > 0)
    pressure_bc <<<grid,block>>> (pnew);

ROTATE_POINTERS(p,pnew);
```

Listing 5. An example Jacobi pressure loop using tri-level MPI-OpenMP-CUDA and simple computational overlapping. This uses the SINGLE threading level.

406 tation. At the time this study was performed, the MPICH2 implementation
407 on NCSA Lincoln had interactions with the CUDA pinned memory support,
408 making it very slow for the CUDA Streams overlapping cases. OpenMPI was
409 used instead. But, unfortunately, the OpenMPI versions available on NCSA
410 Lincoln do not support any threading level other than `SINGLE`, and optimal
411 network performance was not obtainable with custom compiled versions by
412 the first author. Hence only the last implementation was used. An example
413 implementation is shown in Listing 5, where simple computational overlap-
414 ping is performed. CUDA computations are performed on threads $1 - N$,
415 while MPI calls are performed on the single thread 0. With a `FUNNELED` hy-
416 brid implementation, the `omp master` pragma would be used instead, with
417 care taken since it has no implied barrier as `omp single` does.

418 4.4. Parallel Geometric Multigrid Method

419 Solution of complex incompressible flows benefits substantially from an
420 advanced solver for the pressure Poisson equation, such as a multigrid (MG)
421 method. The parallel geometric multigrid method that we implement in this
422 study is built upon the strategies and lessons learned in previous sections.
423 Based on the performance results obtained from parallel computations that
424 adopt the Jacobi solver, we choose to follow the MPI-CUDA implementa-
425 tion described in section 4.2.3 in our MG method implementation. The 3D
426 geometric MG method is composed of the *restriction*, *smoothing*, and *prolon-*
427 *gation* steps. In the restriction step we use a 27-point full weighting scheme
428 to restrict the residual solution from the fine grid to the next coarse grid
429 level. The prolongation operator is the inverse operator of the restriction
430 step. Therefore, we use a trilinear interpolation in the prolongation stage.
431 In the smoothing stage, we use a weighted ($\omega = 0.86$) Jacobi solver with 3
432 to 4 iterations as the *smoother* for 3D computations.

433 Different schemes can be adopted to coarsen the grid in the MG method
434 [56]. In our implementation, we use the V-cycle, which is adequate for the
435 solution of pressure Poisson equation resulting from incompressible flow for-
436 mulations. We develop an amalgamation strategy to overcome the data-
437 starvation issue that arises in a multi-GPU implementation of the MG method.
438 Basically, when the mesh at the finest level is divided and distributed over
439 the GPUs, data-starvation per GPU is inevitable because of the inherent
440 grid coarsening strategy in the MG method. When the coarsest grid per
441 GPU is reached, the overall solution has not reached the deepest level in the
442 V-cycle. We call the implementation that halts the grid coarsening process

443 when the coarsest mesh per GPU is reached as the *truncated* MG method.
444 Depending on the size of the mesh and the number of GPUs deployed in the
445 computations, truncating the MG cycle can substantially degrade the supe-
446 rior convergence rate of the MG method. To avoid this issue, we develop
447 an amalgamation strategy to complete the V-cycle to its full-depth for the
448 whole mesh. Our amalgamation strategy make use of the collective commu-
449 nication in the MPI library. Specifically, we use the MPI `Gather` function to
450 reconstruct the mesh on a single GPU, and continue with the V-cycle down
451 to its full-depth until the coarsest mesh for the overall domain is reached.
452 Once the coarse grid solution is performed on a single GPU, we proceed
453 with the V-cycle on a single GPU and scatter the information to all GPUs
454 with an MPI `Scatter` function at the same MG level where the amalgama-
455 tion to a single GPU took place. The amalgamation strategy enables us to
456 achieve the superior efficiency of the MG method in a parallel multi-GPU
457 implementation.

458 5. Performance Results from NCSA Lincoln and TACC Longhorn 459 Clusters

460 The NCSA Lincoln cluster consists of 192 Dell PowerEdge 1950 III servers
461 connected via InfiniBand SDR (single data rate) [54]. Each compute node
462 has two quad-core 2.33 GHz Intel E5410 processors and 16GB of host mem-
463 ory. The cluster has 96 NVIDIA Tesla S1070 accelerator units each housing
464 four C1060-equivalent Tesla GPUs. An accelerator unit is shared by two
465 servers via PCI-Express $\times 8$ connections. Hence, a compute-node has access
466 to two GPUs. For the present study, performance measurements for 64 of the
467 192 available compute-nodes in the NCSA Lincoln Tesla cluster are shown,
468 with up to 128 GPUs being utilized. The CUDA 3.0 Toolkit was used for
469 compilation and runtime, gcc 4.2.4 was the compiler used, and OpenMPI
470 1.3.2 was used for the MPI library.

471 The TACC Longhorn cluster consists of 240 Dell R610 compute nodes
472 connected via InfiniBand QDR (quad data rate). Each compute node has
473 two quad-core 2.53 GHz Intel E5540 Nehalem processors and 48GB of host
474 memory. The cluster has 128 NVIDIA QuadroPlex S4 accelerator units each
475 housing four FX5800 GPUs. An accelerator unit is shared by two servers via
476 PCI-Express 2.0 $\times 16$ connections. Performance of the GPU units is similar to
477 the Lincoln cluster, however the device/host memory bandwidth is more than
478 $2\times$ higher and the cluster interconnect is $4\times$ faster. For the present study,

This is an author-produced, peer-reviewed version of this article. The final, definitive version of this document can be found online at *Parallel Computing*, published by Elsevier. Copyright restrictions may apply. doi: 10.1016/j.parco.2012.10.002

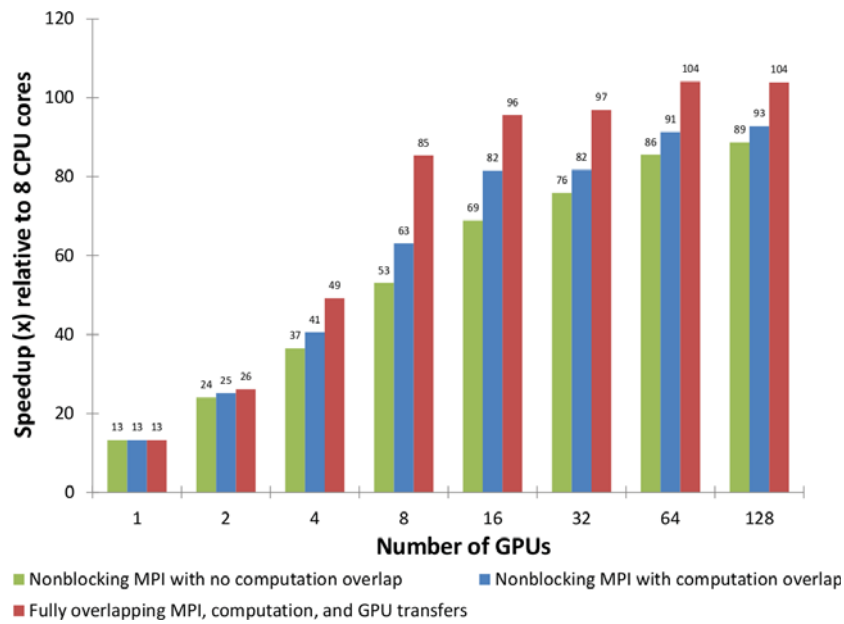


Figure 5. Speedup on the NCSA Lincoln Tesla cluster from the three MPI-CUDA implementations relative to the Pthreads parallel CPU code using all 8 cores on a compute-node. The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

479 performance measurements for 128 of the 240 available compute-nodes in the
480 TACC Longhorn cluster are shown, with up to 256 GPUs being utilized. The
481 CUDA 3.0 Toolkit was used for compilation and runtime, gcc 4.1.2 was the
482 compiler used, and OpenMPI 1.3.3 was used for the MPI library.

483 Single GPU performance has been studied relative to a single CPU proces-
484 sor in many studies. Such performance comparisons are adequate for desktop
485 GPU platforms. On a multi-GPU cluster, a fair comparison should be based
486 on all the available CPU resources in the cluster. To partially address this
487 issue, the CPU version of the CFD code is parallelized with Pthreads to use
488 the eight CPU cores available on a single compute-node of the NCSA Lin-
489 coln cluster [15, 16]. Identical numerical methods are used in the CPU and
490 GPU code for the tests performed. In Thibault and Senocak [16], the per-
491 formance of the CPU version of the code was investigated and the GFLOPS
492 performance was found to be comparable to the NPB benchmark codes.

493 A lid-driven cavity problem at a Reynolds number of 1000 was chosen for
494 performance measurements. Measurements were performed for both *strong*
495 *scaling* where the problem size remains fixed as the number of processing
496 elements increases, and *weak scaling* where the problem size grows in direct
497 proportion to the number of processing elements. Measurements for the CPU
498 application were done using the Pthreads shared-memory parallel implemen-
499 tation using all eight CPU cores on a single compute-node of the NCSA
500 Lincoln cluster. All measurements include the complete time to run the ap-
501 plication including setup and initialization, but do not include I/O time for
502 writing out the results. Single precision was used in all computations.

503 *Strong Scaling Analysis*

504 Figure 5 shows the speedup of the MPI-CUDA implementation of our
505 flow solver relative to the performance of the CPU version of our solver
506 using Pthreads. The computational performance on a single compute-node
507 with 2 GPUs was $26\times$ faster than 8 Intel Xeon cores, and 64 compute-nodes
508 with 128 GPUs performed up to $104\times$ faster than 8 Intel Xeon cores. In
509 all configurations the fully overlapped implementation performed faster than
510 the first implementation that did not perform overlapping. Additionally, the
511 final fully overlapping implementation performs fastest in all configurations
512 with more than one GPU, and shows a significant benefit with more than four
513 GPUs. With the fixed problem size, the amount of work to do on each node
514 quickly drops — on a single GPU a single pressure iteration takes under
515 10ms of compute time. Little gain is seen beyond 16 GPUs on this fixed

516 size problem, which highlights the fact that GPU clusters
517 problems with large data sets.

518 *Weak Scaling Analysis*

519 All three MPI-CUDA implementations presented in section 4.2 were also
520 run with increasing problem sizes such that the memory used per GPU was
521 approximately equal. The analysis is commonly referred to as *weak scalabil-*
522 *ity*. Simulations such as channel or duct flows can lead to extension of the
523 whole domain in one of the three dimensions as the problem size increases. In
524 this case the height and depth of a channel is fixed, while the width increases
525 relative to the number of GPUs. For the 1D network decomposition per-
526 formed in our flow solver, the amount of data transferred between each GPU
527 will be constant, as will the domain dimensions on each GPU. Therefore we
528 expect the scalability to be excellent.

529 Figure 6a indicates how scalability with the fully overlapped implementa-
530 tion performs so well in this one dimensional scaling case, dropping from 94%
531 with 4 GPUs to only 93% with 128 GPUs. Note that four GPUs is the first
532 case where the network is utilized. The results from the TACC Longhorn
533 cluster shows a consistent behavior, with only a 1% drop in efficiency from
534 4 GPUs to 256 GPUs. The fully overlapped MPI-CUDA implementation
535 shows a definite advantage over the other two MPI-CUDA implementations.

536 Figure 6b shows the parallel efficiency when the computational domain
537 grows in two dimensions during a weak scaling analysis. This is a very com-
538 mon scenario seen in such examples as many lid-driven cavity and buoyancy-
539 driven cases, as well as flow in complex terrain, where covering a larger phys-
540 ical area (e.g. more square blocks in an urban simulation) involves growth
541 in the horizontal dimensions, while the number of cells used for height re-
542 mains constant. On the TACC Longhorn cluster, 256 GPUs were utilized
543 on 128 compute-nodes to sustain an 4.9 TeraFLOPS performance. With ap-
544 proximately 400GB of memory used during the computation on 128 GPUs,
545 it is not possible to directly compare this to a single node CPU implemen-
546 tation on traditional machines. Figure 6b also shows the clear advantage
547 of overlapping computation and communication. Parallel efficiency in the
548 two-dimensional growth problem with full overlapping is excellent through
549 64 GPUs, and parallel efficiency drops to 60% beyond 64 GPUs.

550 One obvious feature of Figure 6(b) is that efficiency does not fall in
551 a smooth fashion with increasing GPUs, but steps up and down with an

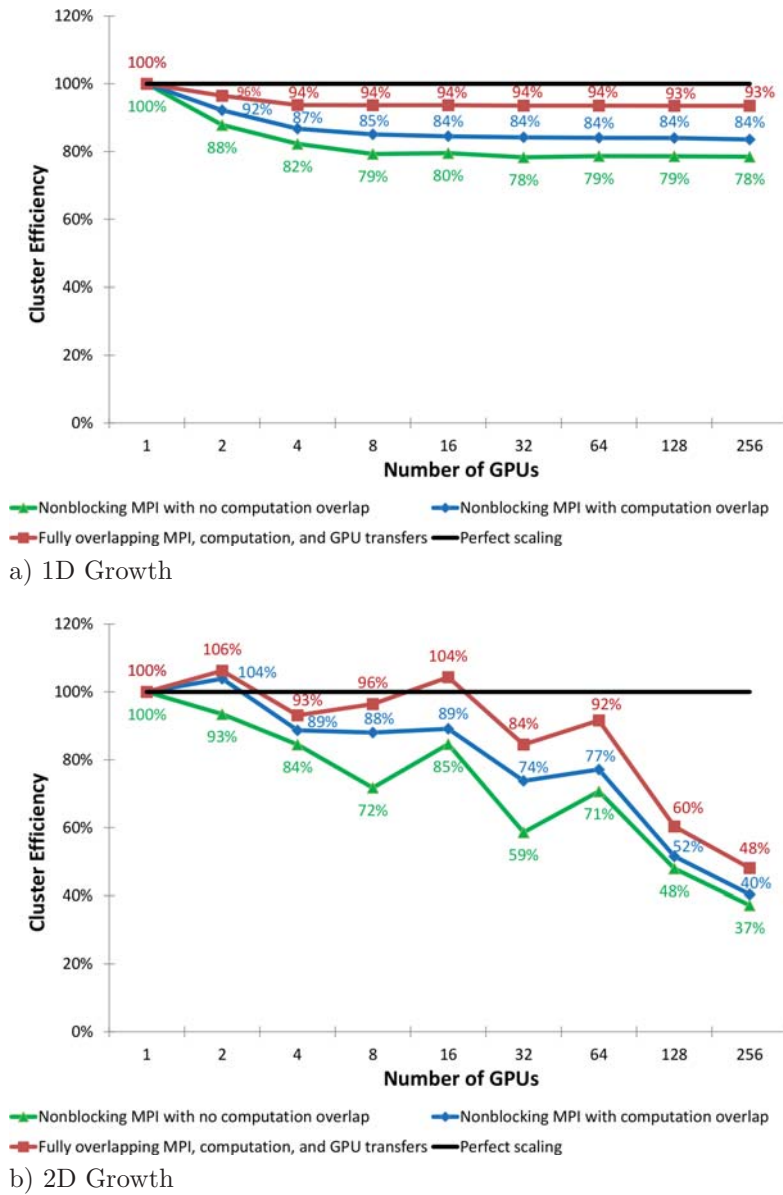


Figure 6. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs on the TACC Longhorn cluster (weak scalability presentation). a) Growth is in one dimension. The size of the computational grid is varied from $512 \times 512 \times 256$ to $512 \times 512 \times 65536$ with increasing number of GPUs. b) Growth is in two dimensions, with the Y dimension fixed. The size of the computational grid is varied from $1024 \times 64 \times 1024$ to $16384 \times 64 \times 16384$ with increasing number of GPUs. Using 256 GPUs, computations sustained 8.5 TeraFLOPS in 1D the growth case and 4.9 TeraFLOPS in the 2D growth case.

552 overall decreasing trend. This is related to an interaction between the two-
553 dimensional problem size growth and the structure of the CUDA kernels.
554 The mechanism of having each thread loop over all the Z planes is very effi-
555 cient, however the CUDA kernel throughput strongly changes as the numbers
556 of threads (the X and Y dimensions) relative to the number of Z planes per
557 GPU is varied. Earlier implementations of the kernels, as seen in Jacobsen
558 et al. [55], show much less variability, but overall performance is lower — for
559 a similar problem the single GPU performance is 33 GFLOPS vs. 41 for the
560 current code, and 2.4 TFLOPS vs. 2.9 TFLOPS with 128 GPUs.

561 Figure 7 presents the weak scaling analysis for a growth in three dimen-
562 sions of the computational domain on the Longhorn cluster. Figure 7(a)
563 indicates how scalability with the fully overlapped implementation trails off
564 sharply at 16 GPUs, and the gap between the overlapping implementations
565 and non-overlapping narrows. The reasons for this behavior are examined
566 in the next section. Figure 7(b) shows the sustained GFLOPS performance
567 on a logarithmic scale. With 256 GPUs, 2.4 TeraFLOPS was sustained with
568 the fully overlapped implementation. Note that for the 1D growth case, 9.5
569 TeraFLOPS was sustained using the same number of GPUs.

570 *Further Remarks on Scalability*

571 NCSA Lincoln cluster was transformed into a GPU cluster from an ex-
572 isting CPU cluster. The connection between the compute-nodes and the
573 Tesla GPUs are through PCI-Express Gen 2 $\times 8$ connections rather than
574 $\times 16$. Measured bandwidth for pinned memory is approximately 1.6 GB/s,
575 which is significantly slower than the 5.6 GB/s measured on a local worksta-
576 tion with PCIe Gen 2 $\times 16$ connections to Tesla C1060s. Kindratenko et al.
577 [54] observed a low host-device bandwidth on Lincoln cluster, and suggested
578 further investigations. This observed low-bandwidth issue with the Lincoln
579 cluster has an impact on our results.

580 We performed bandwidth measurements on the TACC Longhorn clus-
581 ter which uses GPUs with similar performance (Quadroplex 2200 S4 on
582 Longhorn, Tesla S1070 on Lincoln). However, measured device/host memory
583 transfers are over $2\times$ faster on Longhorn, and its Infiniband QDR shows a
584 $4\times$ increase in interconnect bandwidth with simple benchmarks. It should
585 also be pointed out that as the CUDA kernels are optimized and run faster,
586 less time becomes available for overlapping communications, leading to a loss
587 in parallel efficiency.

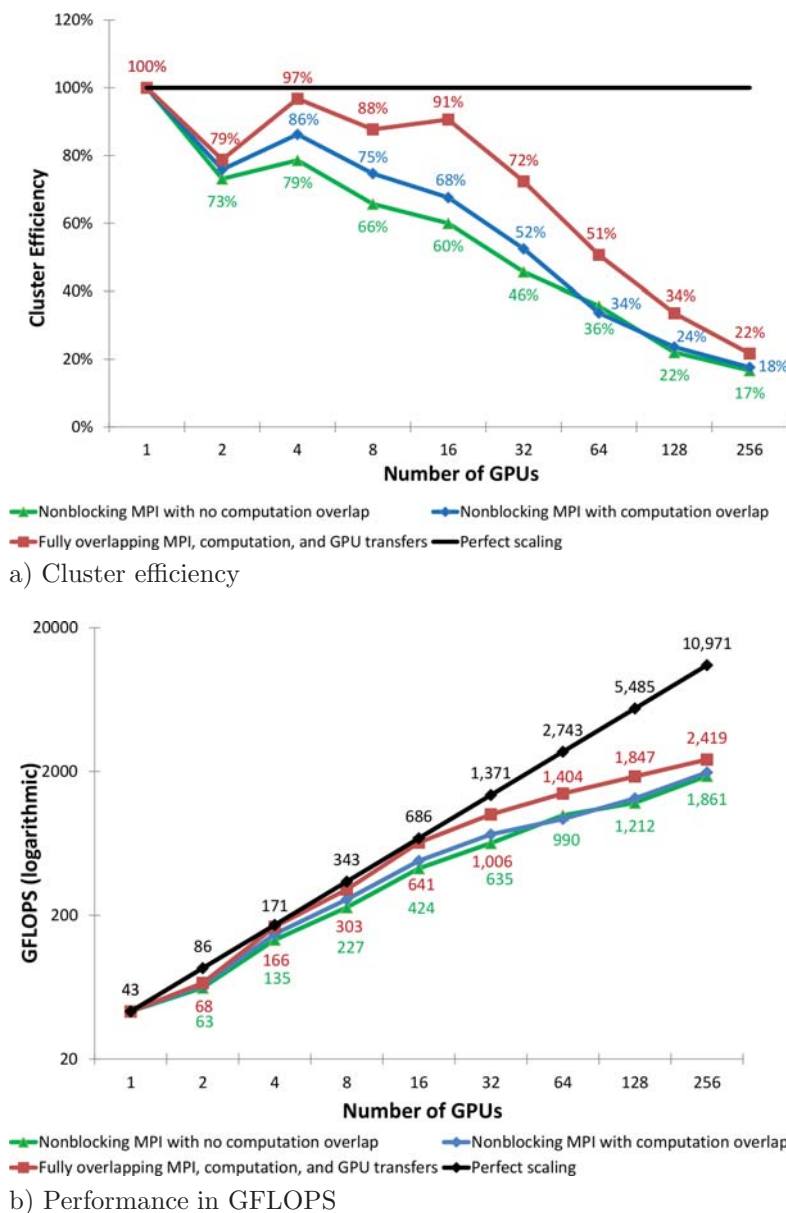


Figure 7. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs on the TACC Longhorn cluster (weak scalability presentation). Growth is in three dimensions. The size of the computational grid is varied from $416 \times 416 \times 416$ to $2688 \times 2688 \times 2560$ with increasing number of GPUs. a) Parallel cluster efficiency, b) Performance in GFLOPS.

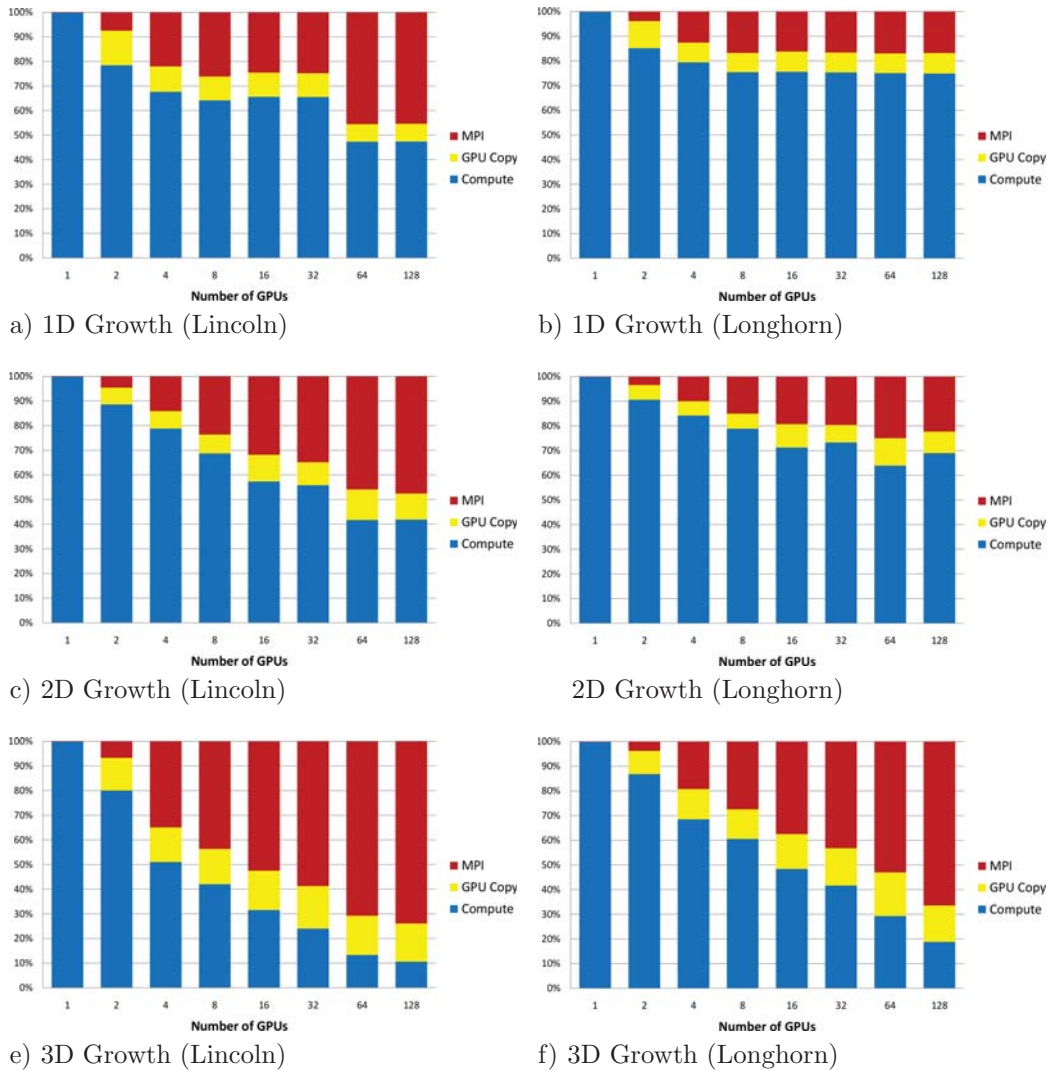


Figure 8. Percent of pressure Poisson solver (30 Jacobi iterations) time spent in computation, host/GPU memory transfer, and MPI calls. No overlapping is used. The problem size grows such that the number of cells per GPU is approximately constant. a–b) 1D growth, c–d) 2D growth, e–f) 3D growth.

588 To further examine the reasons behind the scalability results seen, CUDA
589 event timers were used to get high resolution profiles of time spent in the
590 iterative pressure solver. The timers calculated the time spent doing compu-
591 tation in the pressure and boundary condition kernels, the amount of time
592 spent copying data between the GPU and the host, and the time spent in
593 network communications. This data was collected using the implementation
594 described in section 4.2.1, to show the need for overlapping as well as shed
595 light into the earlier scalability graphs.

596 For the 1D growth case shown in Figure 8a–b, measured compute and
597 GPU copy time was essentially constant for all runs. This is expected, as
598 the per-GPU dimensions of the pressure domain are identical at each size,
599 and the amount of data to be transferred is constant. The amount of data
600 exchanged by each host also remains constant as the number of GPUs in-
601 creases, yet the time spent in MPI calls on the Lincoln cluster increases with
602 more GPUs. While performing the solver iterations, each process only syn-
603 chronizes with its immediate neighbors – no global operations are used. We
604 attribute the observed behavior as a network topology issue with the Lincoln
605 cluster and not with our implementation because it is absent in the results
606 using the Longhorn cluster. On the Longhorn cluster, as shown in Figure 8b,
607 the percent of time spent in copy and MPI is essentially constant once the
608 network is utilized at 4 GPUs, which is what is expected.

609 With the 2D growth case shown in Figure 8c–d, the amount of data to be
610 transferred grows by a factor of \sqrt{N} as the number of GPUs (N) increases.
611 In the 4 GPU case each transferred layer consists of 2048×64 cells, while
612 with 16 GPUs (a $4\times$ increase) each layer has 4096×64 cells — a $2\times$ increase.
613 With 32 or fewer GPUs, it is possible to completely overlap network traffic
614 and GPU copies with computation. However, the particular size used in this
615 simulation for 32 and 128 GPUs leads to slower computation than other cases,
616 as remarked upon earlier to explain the wiggly trend in parallel efficiency in
617 Figure 6b. With 64 and 128 GPUs, complete overlapping of copy, MPI, and
618 computation needs to be done to keep scalability. The data on Longhorn
619 shows a similar pattern, yet scales better as the communication paths are
620 faster.

621 The 3D growth case is shown in Figure 8e–f. The amount of data to be
622 transferred grows by a factor of $N^{2/3}$ with the number of GPUs. In the single
623 GPU case each transferred layer consists of 416×416 cells, while with 64
624 GPUs each layer has 1664×1664 cells — a $16\times$ increase. Both the GPU
625 copy and MPI communication time increase rapidly, with the GPU copy

626 alone taking more time on 64 GPUs than the entire computation time. The
627 picture on the Longhorn cluster is similar, with the faster data copies just
628 moving the saturation point to more GPUs. While large linear transfers are
629 done to achieve maximum copy efficiency, the amount of data is too large
630 in these cases. Calculations are shown below for the 64 GPU case on the
631 Lincoln cluster:

$$\begin{aligned} \text{Copy Bandwidth} &= (\text{layer size} \cdot 4 \cdot \text{iterations} \cdot \text{timesteps}) / \text{time} & (5) \\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200) / 139.62 \text{ seconds} \\ &= 1816 \text{ MB/s} \end{aligned}$$

632

$$\begin{aligned} \text{MPI Bandwidth} &= (\text{layer size} \cdot 4 \cdot \text{iterations} \cdot \text{timesteps}) / \text{time} & (6) \\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200) / 624.5 \text{ seconds} \\ &= 405.9 \text{ MB/s} \end{aligned}$$

633 For each GPU, the two edge layers must be copied from the GPU and then
634 again to the GPU, hence the factor of 4. This simple calculation ignores the
635 effect of the edge nodes. The effective GPU copy bandwidth is similar to that
636 reported with memory benchmarks on this platform, which is 2 to 3 times
637 less than newer hardware. The effective MPI bandwidth is lower than the
638 bidirectional bandwidth measured with MPI benchmarks, suggesting this as
639 a possible point to investigate.

640 A 2D decomposition would greatly reduce the amount of data transferred
641 with these large 2D and 3D simulations. Assuming a domain partition in the
642 growth dimensions, the 2D and 3D simulations would see a $4\times$ reduction in
643 the number of bytes transferred. The ramifications to CUDA are discussed
644 in section 4.1. It is likely that for 3D problems on many GPUs, the extra
645 CUDA work may be worth the per-GPU cost.

646 Figure 9 directly compares the weak scaling efficiency with growth in
647 three dimensions using a fully overlapped version of our flow solver on NCSA
648 Lincoln and TACC Longhorn clusters. While the improved communication
649 bandwidth on the TACC Longhorn cluster greatly helps scalability (at 128
650 GPUs, Lincoln is at 13% while Longhorn achieves 34%), the overall trend
651 in weak scaling is similar. On the NCSA Lincoln Tesla cluster, only 768
652 GFLOPS was sustained with the fully overlapped implementation using 128

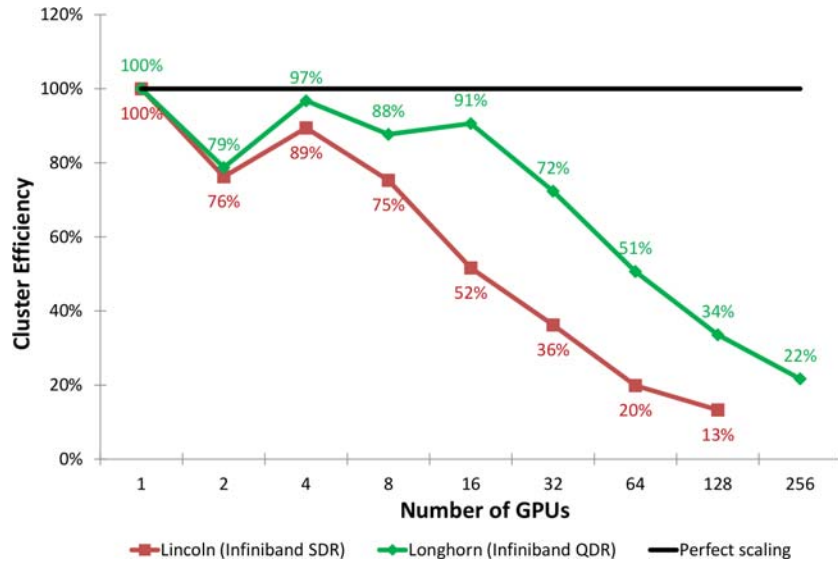


Figure 9. A comparison of weak scaling with the fully overlapped MPI-CUDA implementation on two platforms, with growth in three dimensions. Longhorn has higher bandwidth for both GPU/host and network data transfer than Lincoln.

653 GPUs. On the TACC Longhorn cluster, 2.4 TeraFLOPS was sustained using
 654 256 GPUs.

655 *Performance Analysis of Tri-level MPI-OpenMP-CUDA Implementation*

656 Similar to the dual-level performance results, a lid-driven cavity problem
 657 at a Reynolds number of 1000 was chosen for performance measurements
 658 on the NCSA Lincoln Tesla cluster. As mentioned in section 4.3 earlier,
 659 software issues on the NCSA Lincoln cluster precluded effective testing of
 660 anything but the tri-level implementation to use single threading. The weak
 661 scaling analysis with growth in three dimensions is the most taxing case on
 662 cluster efficiency as compared to growth in one and two dimensions, and shows
 663 the most difference between the parallel methods considered. Therefore we
 664 evaluate the tri-level parallel implementation using weak scaling analysis with
 665 growth in three dimensions, and compare it against the best performing dual-
 666 level parallel implementation.

667 Figure 10 compares the the scaling efficiency of the fully overlapped dual-
 668 level MPI-CUDA and the tri-level MPI-OpenMP-CUDA implementations in

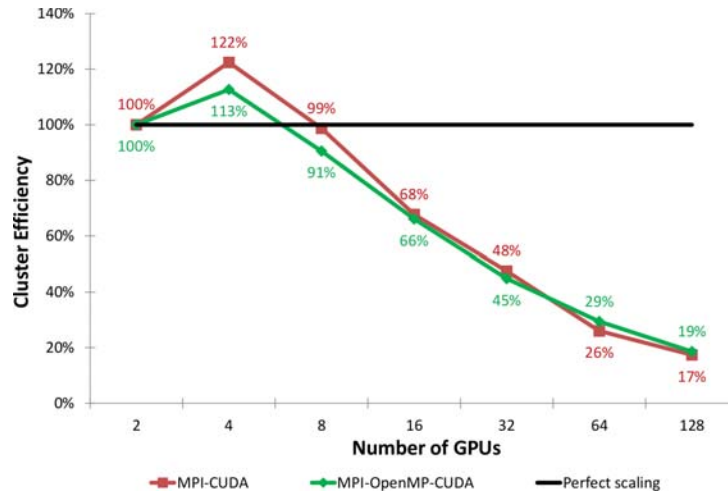


Figure 10. A comparison of weak scaling with the fully overlapped MPI-CUDA and single threaded MPI-OpenMP-CUDA implementations, with growth in three dimensions. Since the tri-level implementation uses all the GPUs of a single node, the base value for parallel scaling is set to a single node of the NCSA Lincoln Tesla cluster containing two GPUs.

669 the 3D growth weak scaling scenario. The MPI-CUDA data matches the fully
670 overlapped data from Figure 7, though 100% is set with two GPUs (a single
671 node) rather than one. We decided to calculate the cluster efficiency relative
672 to the performance of two GPUs in this particular case, because tri-level im-
673 plementation uses all the GPUs of single node with OpenMP addressing the
674 intra-node parallelism and MPI handling the inter-node parallelism. Hence,
675 the super-efficiency observed at 4 GPUs is direct outcome of how we calculate
676 the parallel efficiency in this particular case.

677 With fewer than 4 nodes (8 GPUs), the dual-level MPI-CUDA implemen-
678 tation performs better. With 32 and 64 nodes (64 and 128 GPUs), there is
679 a small benefit with the present MPI-OpenMP-CUDA implementation. At
680 this point the amount of data being transferred may bring any efficiencies
681 of the shared memory model to the forefront, outweighing single-node syn-
682 chronization. Our results are consistent with the hybrid performance results
683 shown by Nakajima [38], where MPI-vector implementation outperformed
684 the hybrid MPI-OpenMP-vector implementation at 64 and fewer nodes, and
685 started showing an increasing benefit at 96 nodes and beyond. We were
686 not able to measure the results beyond 64 nodes (128 GPUs), but we believe

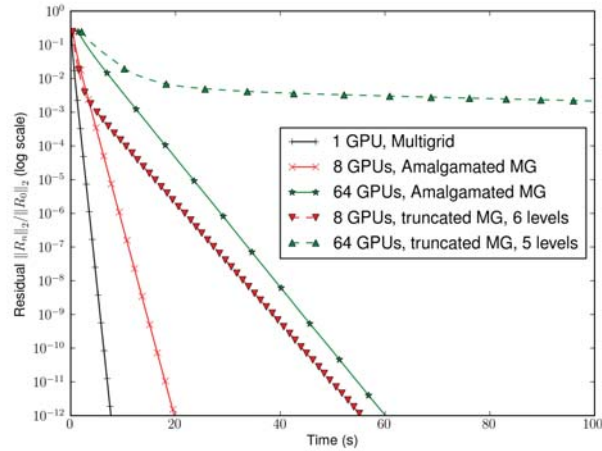


Figure 11. Performance and parallel efficiency of the V-cycle truncated and amalgamated multigrid on 1, 8, and 64 GPUs where the problem size scales with the number of GPUs. Time is plotted against the residual level for a double precision problem using 257^3 on 1 GPU, 513^3 using 8 GPUs, and 1025^3 using 64 GPUs on the NCSA Lincoln Tesla cluster. A marker is shown for each 4 loops of the multigrid cycle.

687 the performance of the tri-level implementation should be further investi-
 688 gated on larger clusters with more than two GPUs per node and also with
 689 different domain decomposition strategies. Unfortunately, such large clusters
 690 with dense GPU nodes were not available or accessible during our study.

691 *Performance of the Parallel Geometric Multigrid Method*

692 A 3D lid-driven cavity problem was started at grid sizes of 257^3 , 513^3
 693 and 1025^3 using 1, 8 and 64 GPUs on the NCSA Lincoln Tesla cluster. We
 694 used double precision in all computations. The actual wall-time taken by the
 695 pressure solver is plotted against the residual level for the initial time step.

696 Figure 11 shows the performance of the multigrid algorithm on the NCSA
 697 Lincoln Tesla cluster for relatively large problems (16M, 128M, and 1024M
 698 cells). In particular the results of the amalgamated full-depth multigrid are
 699 compared to the truncated multigrid, and the single-GPU multigrid imple-
 700 mentation. We note that on a single GPU, issues of amalgamation and
 701 incomplete V-cycles are absent. With 8 GPUs, the coarsest grid is 17^3 , while

702 with 64 GPUs with coarsest grid is 65^3 . On a single GPU a full-depth V-
703 cycle was performed, hence there is no truncation of the V-cycle. Our results
704 show the clear benefit of amalgamation on the convergence rate of a multi-
705 GPU implementation of the MG method. At the larger problem sizes, the
706 convergence rate of the truncated multigrid is unacceptable and the need for
707 amalgamation in the parallel multigrid method becomes obvious.

708 The multigrid level at which amalgamation to a single GPU takes place
709 has an effect on the performance. The current implementation can amalga-
710 mate to a single GPU at the third level in the V-cycle for most problems
711 considered in this study. However, for a grid size of 1025^3 we found that
712 amalgamating at the fourth level or deeper levels produces same performance
713 results, and they are better than performance results obtained when amal-
714 gamating at the third level. We note that the level at which to amalgamate
715 depends on computational problem and device memory sizes.

716 6. Conclusions

717 We have presented both dual-level (MPI-CUDA) and a tri-level (MPI-
718 OpenMP-CUDA) parallel implementations of a Navier-Stokes equations solver
719 to simulate buoyancy-driven incompressible fluid flows on GPU clusters. We
720 adopt NVIDIA's CUDA programming model for fine-grain data-parallel op-
721 erations within each GPU. In the tri-level implementation we use OpenMP
722 for intra-node communications within a compute-node, and MPI for commu-
723 nications across the cluster. In the dual-level implementation, MPI handles
724 all intra- and inter-node communications.

725 We adopted a simple point iterative scheme to solve the pressure Pois-
726 son equation to investigate the interplay of computation, communications,
727 and synchronizations in multi-level parallel implementations on a GPU clus-
728 ter with different strategies to overlap computation with communications.
729 However, many applications, including the present one, require advanced
730 numerical methods and fast solvers such as the multigrid method. There-
731 fore, we extended the best performing multi-level parallel implementation
732 described in this study to a geometric multigrid method, in which we intro-
733 duced an amalgamation strategy to recover the superior convergence rate of
734 the multigrid method on GPU clusters.

735 In all the multi-level implementations we adopted a 1D domain decom-
736 position strategy as the overhead for gathering and scattering the data into
737 linear transfer buffers can exceed the advantages of the smaller transfer sizes

738 that one could get from 2D or 3D domain decompositions. An additional
739 level of 1D domain decomposition is also introduced within the compute-
740 space of each GPU to overlap intra- and inter-node data exchanges with
741 advanced features of MPI and CUDA. We implemented three strategies to
742 overlap computation with communications. With measurements from two
743 different GPU clusters, we showed that performance and efficiency critically
744 depends on the bandwidth of the network, and the strategy that introduces
745 maximum overlapping of computation with communication improves the par-
746 allel performance markedly. Although we have used as many as 256 GPUs on
747 128 nodes of the Longhorn cluster with Infiniband QDR network, the paral-
748 lel efficiency dropped below 50% beyond 64 GPUs on 32 nodes during weak
749 scaling analysis with 3D growth in computational domain sizes, suggesting
750 that multi-GPU computing can benefit substantially from advances in fast
751 networking hardware.

752 Our performance measurements indicate that the dual-level (MPI-CUDA)
753 parallel model with maximum overlapping produces the best performance.
754 We believe the gain from the tri-level MPI-OpenMP-CUDA parallel method
755 is unlikely to offset the additional software complexity that is introduced into
756 the flow solver. Models that share fine-grain parallelism on multi-core CPUs
757 with GPUs, a different domain decomposition strategy than is presented here
758 or have high GPU density per node may see better results and need to be
759 investigated further.

760 A number of issues with obtaining the most benefit from tri-level MPI-
761 OpenMP-CUDA parallel methods have been identified. Compared to early
762 published results, current MPI libraries have much better optimization for
763 multiple processes per node. A number of the benefits ascribed to the hybrid
764 MPI-OpenMP programming model are typically obtained via OpenMP's fine-
765 grain parallelism support, which is not used at all in this study, because all
766 fine-grain parallelism is supplied by CUDA. Other simulation software that
767 can use both CPU and GPU resources for computation may show more
768 advantage from tri-level parallelism. It is also an open question whether a
769 much denser per-node GPU density may be able to take better advantage
770 of the tri-level parallelism. We think having only two GPUs per node on
771 current and planned GPU cluster designs puts a limit on the possible benefit
772 from the mixed API model. At the time of the present study, GPU clusters
773 with denser nodes were not available.

774 Finally, with our best performing implementation using 256 GPUs on the
775 TACC Longhorn cluster, we were able to process 17 billion elements with

776 8.5, 4.9, 2.4 TeraFLOPS of single precision sustained performance in 1D, 2D
777 and 3D growth during weak scaling analysis, respectively. On the NCSA
778 Lincoln cluster, we have shown that 2-GPU performance of our solver is $26\times$
779 faster than the 8-core CPU performance. Our results demonstrate that GPU
780 clusters are powerful computing platforms to solve computationally large
781 problems. With their heterogeneous architectures that can support both
782 CPU and GPU based applications and graphics rendering, we expect a wide
783 adoption of GPU clusters in the industry and academia.

784 **Acknowledgments**

785 This work is partially funded by grants from NASA Idaho Space Grant
786 Consortium and National Science Foundation (NSF) (Award #1043107 and
787 #1056110). We utilized the Lincoln Tesla Cluster at the National Center
788 for Supercomputing Applications and the Longhorn visualization cluster
789 at the Texas Advanced Computing under grant number ASC090054 and
790 ATM100032 from NSF TeraGrid. We thank NVIDIA Corporation for hardware
791 donations and extend our thanks to Marty Lukes of Boise State University
792 for his continuous help with building and maintaining our GPU computing
793 infrastructure.

794 **References**

- 795 [1] NVIDIA, NVIDIA CUDA Programming Guide 3.1.1, 2010.
- 796 [2] Khronos OpenCL Working Group, The OpenCL Specification: Version
797 1.1, 2010.
- 798 [3] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington,
799 W.-M. Hwu, QP: A heterogeneous multi-accelerator cluster,
800 in: Proceedings of the 10th LCD International Conference on High-
801 Performance Clustered Computing, Boulder, Colorado, 2009.
- 802 [4] NCSA, Intel 64 Tesla Linux cluster Lincoln webpage,
803 [http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/
804 Intel64TeslaCluster/](http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/) (2008).
- 805 [5] HPCwire, New Chinese GPGPU super outruns Jaguar,
806 [http://www.hpcwire.com/blogs/New-China-GPGPU-Super-Outruns-
807 Jaguar-105987389.html](http://www.hpcwire.com/blogs/New-China-GPGPU-Super-Outruns-Jaguar-105987389.html) (Oct. 2010).

- 808 [6] HPCwire, NVIDIA Tesla GPUs power world's fastest supercomputer,
809 <http://www.hpcwire.com/offthewire/NVIDIA-Tesla-GPUs-Power>
810 [-Worlds-Fastest-Supercomputer-105983244.html](http://www.hpcwire.com/offthewire/NVIDIA-Tesla-GPUs-Power-Worlds-Fastest-Supercomputer-105983244.html) (Oct. 2010).
- 811 [7] HPCwire, China's new Nebulae supercomputer is
812 no. 2, right on the tail of ORNL's Jaguar
813 [http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-](http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar)
814 [Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar](http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar)
815 [95258669.html](http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar) (May 2010).
- 816 [8] H. Simon, T. Zacharia, R. Stevens, Modeling and simulation at the ex-
817 ascale for energy and the environment, Tech. rep., DOE ASCR Program
818 (2008).
- 819 [9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E.
820 Lefohn, T. J. Purcell, A survey of general-purpose computation on
821 graphics hardware, *Computer Graphics Forum* 26 (1) (2007) 80–113.
- 822 [10] T. Brandvik, G. Pullan, Acceleration of a 3D Euler solver using com-
823 modity graphics hardware, in: 46th AIAA Aerospace Sciences Meeting
824 and Exhibit, 2008.
- 825 [11] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over
826 a hypersonic vehicle using a GPU, *Journal of Computational Physics*
827 227 (24) (2008) 10148–10161.
- 828 [12] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker,
829 C. Becker, S. Turek, Using GPUs to improve multigrid solver perfor-
830 mance on a cluster, *International Journal of Computational Science and*
831 *Engineering* 4 (1) (2008) 36–55.
- 832 [13] J. M. Cohen, M. J. Molemaker, A fast double precision CFD code using
833 CUDA, in: *Proceedings of Parallel CFD*, 2009.
- 834 [14] H. Schive, Y. Tsai, T. Chiueh, GAMER: a GPU-accelerated adaptive
835 mesh refinement code for astrophysics, *Astrophysical Journal Supple-*
836 *ment Series* 186 (2010) 457–484.
- 837 [15] J. C. Thibault, I. Senocak, CUDA implementation of a Navier–Stokes
838 solver on multi-GPU platforms for incompressible flows, in: 47th AIAA
839 *Aerospace Science Meeting*, 2009.

- 840 [16] J. C. Thibault, I. Senocak, Accelerating incompressible flow computa-
841 tions with a Pthreads-CUDA implementation on small-footprint multi-
842 GPU platforms, *The Journal of Supercomputing* 59 (2) (2012) 693–719.
- 843 [17] D. A. Jacobsen, Methods for multilevel parallelism on GPU clusters:
844 Application to a multigrid accelerated Navier-Stokes Solver, Master’s
845 thesis, Boise State University, Boise, Idaho (May 2011).
- 846 [18] P. Micikevicius, 3D finite difference computation on GPUs using CUDA,
847 in: *Proceedings of 2nd Workshop on General Purpose Processing on*
848 *Graphics Processing Units*, 2009, pp. 79–84.
- 849 [19] D. Goddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU acceleration of
850 an unmodified parallel finite element Navier–Stokes solver, in: *Internation-*
851 *al Conference on High Performance Computing & Simulation*, 2009,
852 pp. 12–21.
- 853 [20] J. C. Phillips, J. E. Stone, K. Schulten, Adapting a message-driven
854 parallel application to GPU-accelerated clusters, in: *Proceedings of the*
855 *2008 ACM/IEEE conference on Supercomputing*, 2008.
- 856 [21] E. H. Phillips, Y. Zhang, R. L. Davis, J. D. Owens, Rapid aerodynamic
857 performance prediction on a cluster of graphics processing units, in:
858 *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009.
- 859 [22] G. Krawezik, Performance comparison of MPI and three OpenMP pro-
860 gramming styles on shared memory multiprocessors, in: *Proceedings*
861 *of the Fifteenth Annual ACM Symposium on Parallel Algorithms and*
862 *Architectures*, ACM, San Diego, California, USA, 2003, pp. 118–127.
- 863 [23] M. J. Berger, M. J. Aftosmis, D. D. Marshall, S. M. Murman, Perfor-
864 mance of a new CFD flow solver using a hybrid programming paradigm,
865 *J. Parallel Distrib. Comput.* 65 (4) (2005) 414–423.
- 866 [24] F. Cappello, O. Richard, D. Etiemble, Performance of the NAS bench-
867 marks on a cluster of SMP PCs using a parallelization of the MPI pro-
868 grams with OpenMP, in: *PaCT*, 1999, pp. 339–350.
- 869 [25] F. Cappello, O. Richard, D. Etiemble, Investigating the performance of
870 two programming models for clusters of SMP PCs, in: *HPCA*, 2000, pp.
871 349–359.

- 872 [26] F. Cappello, D. Etiemble, MPI versus MPI+OpenMP on IBM SP for the
873 NAS benchmarks, in: Proceedings of the 2000 ACM/IEEE conference
874 on Supercomputing (CDROM), IEEE Computer Society, Dallas, Texas,
875 United States, 2000, p. 12.
- 876 [27] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, H. A. Gabb,
877 Dual-level parallel analysis of harbor wave response using MPI and
878 OpenMP, *Int. J. High Perform. Comput. Appl.* 14 (1) (2000) 49–64.
- 879 [28] D. S. Henty, Performance of hybrid message-passing and shared-memory
880 parallelism for discrete element modeling, in: Supercomputing '00: Pro-
881 ceedings of the 2000 ACM/IEEE conference on Supercomputing, IEEE
882 Computer Society, Dallas, Texas, United States, 2000, p. 10.
- 883 [29] P. Luong, C. P. Breshears, L. N. Ly, Coastal ocean modeling of
884 the U.S. west coast with multiblock grid and dual-level parallelism,
885 in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE confer-
886 ence on Supercomputing, ACM, New York, NY, USA, 2001, pp. 9–9.
887 doi:<http://doi.acm.org/10.1145/582034.582043>.
- 888 [30] S. Dong, G. E. Karniadakis, Dual-level parallelism for deterministic and
889 stochastic CFD problems, in: Supercomputing '02: Proceedings of the
890 2002 ACM/IEEE conference on Supercomputing, IEEE Computer So-
891 ciety Press, Los Alamitos, CA, USA, 2002, pp. 1–17.
- 892 [31] K. Nakajima, H. Okuda, Parallel iterative solvers for unstructured grids
893 using an OpenMP/MPI hybrid programming model for the GeoFEM
894 platform on SMP cluster architectures, in: Proceedings of the 4th Inter-
895 national Symposium on High Performance Computing, Springer-Verlag,
896 Kansai Science City, Japan, 2002, pp. 437–448.
- 897 [32] R. Rabenseifner, Communication bandwidth of parallel programming
898 models on hybrid architectures, in: Proceedings of the 4th International
899 Symposium on High Performance Computing, Springer-Verlag, Kansai
900 Science City, Japan, 2002, pp. 401–412.
- 901 [33] R. Rabenseifner, Hybrid parallel programming on HPC platforms,
902 in: EWOMP '03: Proceedings of the Fifth European Workshop on
903 OpenMP, Aachen, Germany, 2003, pp. 185–194.

- 904 [34] A. Prabhakar, V. Getov, Performance evaluation of hybrid parallel pro-
905 gramming paradigms, in: Performance analysis and grid computing,
906 Kluwer Academic Publishers, 2004, pp. 57–76.
- 907 [35] E. Lusk, A. Chan, Early experiments with the OpenMP/MPI hybrid
908 programming model, *Lecture Notes in Computer Science* 5004 (2008)
909 36.
- 910 [36] G. Hager, G. Jost, R. Rabenseifner, Communication characteristics and
911 hybrid MPI/OpenMP parallel programming on clusters of multi-core
912 SMP nodes, in: Proceedings of the Cray Users Group Conference, 2009.
- 913 [37] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. L. Lusk,
914 R. Thakur, J. L. Träff, MPI on a million processors, in: Proceedings
915 of the 16th European PVM/MPI Users’ Group Meeting on Recent Ad-
916 vances in Parallel Virtual Machine and Message Passing Interface, 2009,
917 pp. 20–30.
- 918 [38] K. Nakajima, Three-level hybrid vs. flat MPI on the Earth Simulator:
919 Parallel iterative solvers for finite-element method, *Applied Numerical*
920 *Mathematics* 54 (2) (2005) 237–255.
- 921 [39] P. K. Kundu, I. M. Cohen, *Fluid Mechanics*, 4th Edition, Academic
922 Press, 2007.
- 923 [40] M. Griebel, T. Dornseifer, T. Neunhoffer, *Numerical Simulation in*
924 *Fluid Dynamics: A Practical Introduction*, Society for Industrial and
925 Applied Mathematics, Philadelphia, PA, USA, 1997.
- 926 [41] J. C. Tannehill, D. A. Anderson, R. H. Pletcher, *Computational Fluid*
927 *Mechanics and Heat Transfer*, 2nd Edition, Taylor & Francis, 1997.
- 928 [42] A. J. Chorin, Numerical solution of the Navier–Stokes equations, *Math.*
929 *Comput.* 22 (1968) 745–762.
- 930 [43] D. A. Jacobsen, I. Senocak, A full-depth amalgamated parallel 3D ge-
931 ometric multigrid solver for GPU clusters, in: 49th AIAA Aerospace
932 Science Meeting, 2011.
- 933 [44] U. Ghia, K. N. Ghia, C. Shin, High-Re solutions for incompressible flow
934 using the Navier–Stokes equations and a multigrid method, *Journal of*
935 *Computational Physics* 48 (1982) 387–411.

- 936 [45] D. C. Wan, B. S. V. Patnaik, G. W. Wei, A new benchmark quality
937 solution for the buoyancy-driven cavity by discrete singular convolution,
938 *Numerical Heat Transfer, Part B: Fundamentals* 40 (3) (2001) 199–228.
- 939 [46] I. Senocak, J. Thibault, M. Caylor, Rapid-response urban CFD simu-
940 lations using a GPU computing paradigm on desktop supercomputers,
941 in: *Eighth Symposium on the Urban Environment*, Phoenix, Arizona,
942 2009.
- 943 [47] I. Senocak, D. Jacobsen, in: *Fifth International Symposium on Com-
944 putational Wind Engineering, CWE2010*, Chapel Hill, North Carolina,
945 USA, 2010.
- 946 [48] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley
947 Longman Publishing Co., Inc., Boston, MA, USA, 1997.
948 URL <http://portal.acm.org/citation.cfm?id=263953>
- 949 [49] The OpenMP ARB, *OpenMP: A Proposed Industry Standard API for
950 Shared Memory Programming*, 1997.
- 951 [50] W. Gropp, R. Thakur, E. Lusk, *Using MPI-2: Advanced Features of the
952 Message Passing Interface*, MIT Press, 1999.
953 URL <http://portal.acm.org/citation.cfm?id=555150>
- 954 [51] HPCwire, *Microway 9U compact GPU cluster with octopus*,
955 <http://www.microway.com/tesla/clusters.html> (Nov. 2009).
- 956 [52] B. Goglin, High throughput intra-node MPI communication with Open-
957 MX, in: *PDP*, 2009, pp. 173–180.
- 958 [53] W. Gropp, R. Thakur, Thread-safety in an MPI implementation: Re-
959 quirements and analysis, *Parallel Computing* 33 (9) (2007) 595–604.
- 960 [54] V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold,
961 J. E. Stone, J. C. Phillips, W. mei Hwu, GPU clusters for high-
962 performance computing, in: *Proceedings of the IEEE Workshop on Par-
963 allel Programming on Accelerator Clusters*, 2009.
- 964 [55] D. A. Jacobsen, J. C. Thibault, I. Senocak, An MPI-CUDA implemen-
965 tation for massively parallel incompressible flow computations on multi-
966 GPU clusters, in: *48th AIAA Aerospace Science Meeting*, 2010.

⁹⁶⁷ [56] U. Trottenberg, C. Oosterlee, A. Schüller, *Multigrid*, Elsevier, 2001.

A flow solver is parallelized with MPI-CUDA and MPI-OpenMP-CUDA implementations.> Weak and strong scaling analysis performed using up to 256 GPUs> Three strategies to overlap computation and communication are assessed.> MPI-CUDA implementation with maximum overlapping gives the best performance> Tri-level parallelism does not show any advantage for the present application.