A portable and efficient Lagrangian particle capability for idealized atmospheric phenomena

John M. Dennis
National Center for Atmospheric Research
Boulder, Colorado, USA
dennis@ucar.edu

Jian Sun
National Center for Atmospheric Research
Boulder, Colorado, USA
sunjian@ucar.edu

Sheri Voelz
National Center for Atmospheric Research
Boulder, Colorado, USA
mickelso@ucar.edu

George Bryan
National Center for Atmospheric Research
Boulder, Colorado, USA
gbryan@ucar.edu

David Richter
University of Notre Dame
South Bend, Indiana, USA
David.Richter.26@nd.edu

ABSTRACT

The Cloud Model version 1 is an atmospheric model that allows for idealized studies of atmospheric phenomena. A new Lagrangian microphysics capability has been added, enabling a significantly more accurate representation than the traditional bulk or multi-moment approaches frequently found in mesoscale atmospheric models. We have utilized a directive-based approach to enable a single source code to efficiently support execution on both CPU and GPU-based computing platforms. In addition to the use of accelerator directives, changes to the data structures and the message-passing approach used by the Lagrangian particle-based microphysics module were necessary to enable efficient execution for a large number of particles. We focus on a configuration that will be used to investigate the impact of oceanic sea spray on the atmospheric boundary layer within a hurricane. We observe a factor of 5.1× reduction in time to the solution when comparing the execution time for 256 NVIDIA A100 GPUs versus 256 AMD EPYC^{TM} Milan-based compute nodes using 1 billion particles.

CCS CONCEPTS

• **Applied computing** → **Earth and atmospheric sciences**; • **Software and its engineering** → Software design tradeoffs; • **Computing methodologies** → *Massively parallel and high-performance simulations.*

KEYWORDS

GPU, OpenACC, MPI, Roofline, Power efficiency

ACM Reference Format:
John M. Dennis, Jian Sun, Sheri Voelz, George Bryan, and David Richter. 2024. A portable and efficient Lagrangian particle capability for idealized atmospheric phenomena. In *Platform for Advanced Scientific Computing Conference (PASC '24), June 3–5, 2024, Zurich, Switzerland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3659914.3659940

PASC '24, June 3–5, 2024, Zurich, Switzerland

ACM ISBN 979-8-4007-0639-4/24/06.
https://doi.org/10.1145/3659914.3659940

1 INTRODUCTION

Cloud Model 1 (CM1) is a numerical modeling system designed for theoretical studies of atmospheric phenomena. As the name suggests, it was originally developed for studies of clouds and thunderstorms [4], but its capabilities have been expanded to a much broader range of applications, including microscale turbulence (scales less than 1 m) [5], flows induced by mountains/topography [22], and tropical cyclones/hurricanes [27].

Several different equation sets and numerical techniques have been implemented in CM1 to accommodate these broad ranges of applications. The fluid-dynamics solver uses the compressible equations of motion, which are stepped forward in time using a "time-splitting" technique in which terms responsible for the propagation of acoustic waves are integrated over a "small" time step and all other tendency terms are updated only on "large" time steps [26]. The time- and space-discretization methods are documented in [30]. The effects of subgrid turbulence are parameterized using the method of [9], which is widely used for large-eddy simulations of turbulent flows. CM1 also supports a Lagrangian particle capability which has historically been used only for analysis of passive tracers.

Traditionally, clouds are defined and evolved in atmospheric models by tracking Eulerian fields of various cloud properties (number concentration, liquid water content, etc.). However, an Eulerian approach suffers for multiple reasons, including assumptions regarding the droplet size distribution, artificial partitioning of hydrometeor classes, and numerical diffusion [14]. For a wide variety of physical applications, a Lagrangian framework is advantageous, where the governing equations for mass, momentum, and energy conservation are applied in a frame of reference that moves with some elements of the flow. A Lagrangian approach allows for a more straightforward accounting of key processes such as evaporation/condensation, collision with other droplets, activation from the aerosol to droplet state, and rain/drizzle initiation.

For this project, we extend the existing Lagrangian particle capability within CM1 such that the Lagrangian elements or particles become aerosols and cloud droplets. Each computational particle

represents some specified "multiplicity" of actual cloud/aerosol droplets with the same properties — thus a cloud in this kind of representation would be represented by a large number of these Lagrangian "superdroplets" in the same vicinity [25].

While it is well documented that a Lagrangian framework has significant advantages by eliminating artificial diffusion and by increasing the efficacy of the underlying physical description of cloud droplets [14], a major drawback of the method is the need to track potentially huge numbers of droplets through the domain. Shima et al. [25] provided a rule of thumb that to properly capture the droplet size distribution locally, every computational grid point should contain at least $O(100)$ computational droplets. While this scaling with grid size is severe, it is even worse if there are additional droplet properties that must be represented (aerosol composition for example), which would require a larger number of properties per grid point.

To address the significant cost of the Lagrangian droplet approach, we utilize the computational capability of GPU computing. We believe that an efficient GPU-enabled Lagrangian droplet capability could push this numerical method from being a well-acknowledged technique that is impractical to one that becomes the new standard for cloud simulations.

To evaluate the ability of a GPU-enabled Lagrangian droplet capability within CM1 to advance our understanding of atmospheric phenomena, we focus on the boundary layer between the ocean and the high-velocity wind field of tropical cyclones. For this study, additional terms are added to the governing equations of CM1 to account for the rotating-flow environment of tropical cyclones [3]. The Lagrangian droplets in this case correspond to sea spray that is injected from oceanic waves into the turbulent atmospheric boundary layer. We have leveraged the Accelerating Scientific Discovery (ASD) program of the National Center for Atmospheric Research (NCAR) to perform multiple high-resolution simulations to explore the impact of sea spray on the boundary layer. In particular, we performed several simulations with various input parameters including the distance from the center of the cyclone (R); the wind speed at the top of the boundary layer (V); and the radial decay parameter (n) that describes how V changes with radius. Further details, including validation using observations from tropical cyclones, are available in [6].

In the remainder of this paper we describe the necessary code optimizations in Section 2, while in Section 3 we document our experimental configuration. Next in Section 4 we describe the application simulation rates of the resulting code on both CPU and GPU-based nodes. We finish with some concluding remarks in Section 5.

2 OPTIMIZATION

We choose to utilize an OpenACC directive approach to enable the use of GPUs within CM1. Our choice of OpenACC versus OpenMP offload was made based on an assessment at the start of this project that OpenACC was both more robust and performant than OpenMP offload [28]. The choice of a directive-based approach was based on several considerations, including the fact that CM1 is currently written in FORTRAN and is used by a small group of performance-sensitive researchers who do not necessarily have access to GPU-based computing platforms. A directive-based approach also provides the flexibility to enable the code development on GPU to advance incrementally. Incremental code development resolves the challenges of porting a large code volume to GPU at once and facilitates the step-by-step verification of correctness. Note that correctness is determined by comparing several metrics generated by the GPU-enabled code versus the original CPU code compiled with two different but trusted compilers.

Several similar efforts to enable the execution of a single code base on CPUs or GPUs exist. Michalakes et al. [17] had successfully added OpenACC directives to an expensive component of the Weather Research and Forecasting model (WRF) and achieved nearly 10× speedup. Dennis et al. [10] used OpenACC directives and numerical algorithm improvements to achieve a greater than 100× speedup on GPU versus the original CPU version for the Spline Analysis at Mesoscale Utilizing Radar and Aircraft Instrumentation (SAMURAI) code. Gettelman et al. [13] used OpenACC directives to port the cloud microphysics parameterization in the Community Atmosphere Model (CAM) to GPU and obtained 2× to 3× speedup.

Several other Mesoscale and LES models have already been ported to GPU successfully and achieved noticeable speedup, including GALES [24], PALM [16], MicroHH [15], FastEddy [23] and JOULES [2].

2.1 Basic loop offload

A simplified call tree for CM1 is provided in Figure 1. Note that the time stepping loop contains several different basic phases of the calculation including *Advection*, *Turbulence*, *Acoustic Solver*, *Lagrangian droplets*, *Diagnostics*, and *Output IO*. Because the computational cost of CM1 is typically dominated by the time-stepping loop, we focus on the addition of OpenACC directives within the sections of code located within the time-stepping loop. Fortunately, a very large percentage of the execution time was located in sections of the code that were easily ported to the GPU using a trivial application of OpenACC directives to simply nested loop bodies. There were more subtle changes that were necessary to port the *Advection* sections of CM1 to the GPU. We first describe changes to the *Advection* followed by the more involved changes that were necessary to achieve GPU-enablement for the *Diagnostics* and *Lagrangian droplets* sections of the code.

While most of the loop had trivially nested loop bodies, some slightly more complex loops were also present. Figure 2 illustrates a stencil operator that is located within the *Advection* section. While it would be possible to break or fission the outer k-loop and use a simple directive approach on the remaining two loops, such a transformation would, unfortunately, have undesirable side effects. In particular, CPU performance would be reduced by the loop fission because it would eliminate cache reuse for the variable "a". Instead the two inner loops are marked with a *loop worker vector collapse(2)*, while the outer loop is marked with the *loop gang* directive. The use of the multi-level loop directives preserves the original looping structure while still fully exposing the loop-level parallelism necessary for efficient GPU execution.

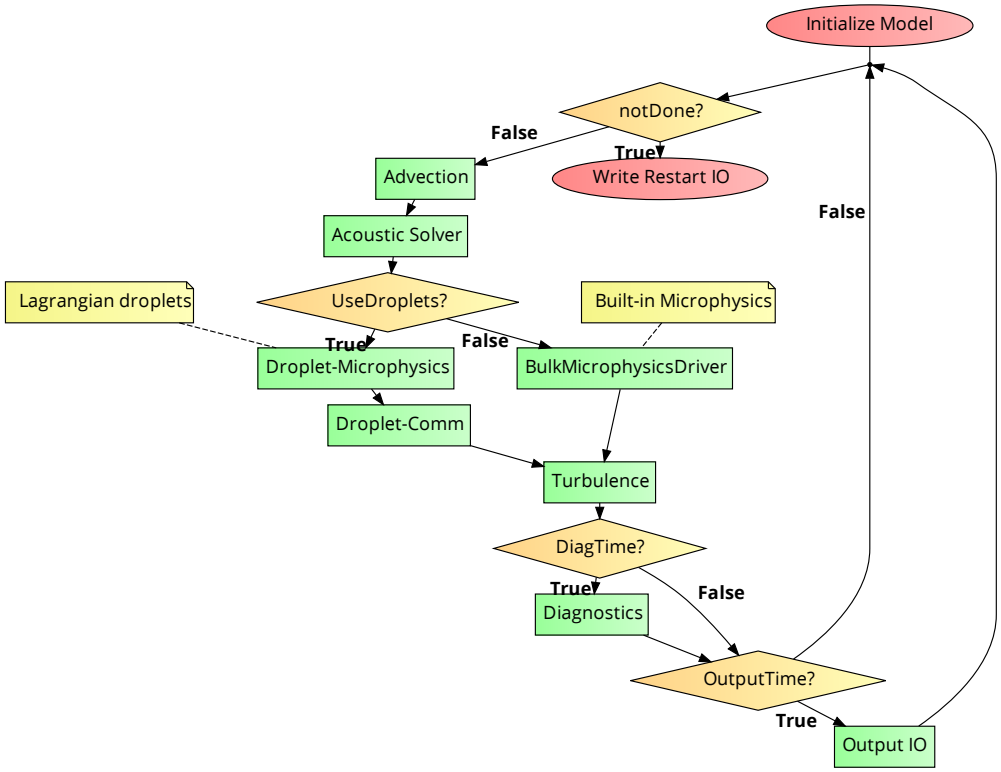

Figure 1: Code flow diagram for CM1 time stepping loop.

```
1  Input: a
2  Output: x, y
3
4  !$acc parallel
5  !$acc loop gang
6  foreach kdx in vertical
7      !$acc loop worker vector
8      foreach idx in horizontal
9          x = flx (stencil-in-x(a))
10     !$acc loop worker vector
11     foreach idx in horizontal
12         y = flx (stencil-in-y(a))
13 !$acc end parallel
```

Figure 2: Example stencil operator used that uses both an 'acc loop gang' and 'acc loop work vector' clause. The multiple acc loops preserve the original code structure and allow cache reuse for the CPU version.

2.2 Diagnostics calculations

While a significant amount of the GPU-enablement work involved the simple addition of OpenACC directives there were a number of loops that required non-trivial transformations. Such loops are frequently located in the *Diagnostics* section of CM1. An example of such a transformation that is necessary to calculate the Courant–Friedrichs–Lewy (CFL) condition within CM1 is provided in Figure 3. While this calculation is equivalent to a combined FORTRAN maxval() and maxloc() operation, its implementation is greatly complicated by the fact that the maxloc() function is not currently supported in OpenACC. While it would be possible to identify the maximum value using the OpenACC reduction clause, its location index can not be easily determined using the current OpenACC standard.

Furthermore, breaking the calculation into two loops that iterate over full 3-dimensional variables, which frequently exceed the size of the last level cache, would potentially hurt CPU performance due to cache misses. The implementation in Figure 3, provided by [20], addresses these challenges by breaking up the calculation of the CFL and its location into two loops, one over the 3-dimensional variables, and a second over a much smaller variable whose horizontal dimension have been collapsed into a single dimension. The first loop which corresponds to lines 8 to 15 in Figure 3 performs a reduction over the vertical dimension by locating the maximum value "wsp" and its indexed location within a single thread. The second loop in lines 20 to 26 calculates the maximum value and its index across threads, by performing the reduction among pairs of threads. The resulting value "fmax" along with its location "locmax" is set on lines 29 and 30 of Figure 3.

In the next section, we describe even larger code transformations that were necessary to enable efficient execution of the *Lagrangian droplets* section of CM1 on both CPU and GPU platforms. Unlike the previously described modifications, which involved loop level

```
Input: velocities   ! velocites
Output: cfl, locmax ! CFL and the location

Initialize cfl = -1.0, loc[idx,kdx] = 0

! reduce the vertical dimension into a
! horizontal array
!$acc parallel loop  ! loop over horizontal
foreach idx := 1 to nhoriz
    !$acc seq loop ! loop over vertical
    foreach kdx in vertical
        calculate wsp :=  sqrt(velocities^2)
        if wsp > cfl(idx)
            update cfl(idx) := wsp
            update loc(idx) := kdx

! Reduce the horizontal dimension to a single value
set pairs := nhoriz/2
set rng  := (nhoriz+1)/2
while (rng is > 1)
   !$acc parallel loop
   foreach i := 1 to pairs
      if cfl(i) <= CFL(rng)
         replace cfl, loc with values from rng+i
   update pairs := rng/2
   update rng := (rng + 1) / 2
! maximum CFL and its location

set fmax := CFL(1)
set locmax := loc(1)
```

Figure 3: Efficient GPU pseudo-code that calculates the Courant–Friedrichs–Lewy (CFL) value and returns its location within the data volume. This calculation is equivalent to a combined FORTRAN maxval() and maxloc() operation.

changes, the necessary alterations for the *Lagrangian droplets* section involved changes to both the data structures, algorithmic and the message-passing approach.

2.3 Lagrangian droplets

At the start of this effort, a basic particle (or, equivalently, droplet) capability existed within CM1. This generic particle capability had several different components that were subsequently modified to support a more sophisticated Lagrangian cloud-droplet capability. The full Lagrangian droplet capability we refer to as *Lagrangian droplets* in Figure 1 and contains both the *Droplet-Microphysics* and *Droplet-Comm*. The *Droplet-Microphysics* performs the actual microphysics calculations for each droplet. The *Droplet-Comm* moves state information from one MPI rank to another. The most problematic aspect of the basic or "naive" droplet capability within CM1 was that it was initially designed to support ≈100,000 droplets, not the very large number that is necessary to enjoy the advantages of the Lagrangian cloud model approach.

In particular, it allocated a single large array "pdata" on each MPI rank with the size *nparcels* × *npvals*, where *nparcels* is the total number of droplets present in the entire simulation and *npvals* the number of physical properties of each droplet. The actual microphysics calculations occur in a large do-loop of length *nparcels*. An if-test is included in this do-loop which determines if a particular MPI rank owns a specific droplet. The communication step present in the naive implementation of *Droplet-Comm* was implemented by setting the value of droplets not owned by a particular MPI rank to a masked value followed by a call to MPI_Allreduce(). While such a naive implementation may be acceptable for a modest number of droplets, the cost of the Allreduce was excessive for greater than one million droplets. A better solution was needed that would both reduce the total memory requirements as well as the communication cost. We next describe our improved solution.

2.3.1 Revised data structures. To address the previously mentioned deficiencies in the naive droplet implementation, we began by eliminating the duplicate droplet-specific data in the "pdata" array. Instead of allocating the array to be of size $nparcels \times npvals$, we now allocate it of size $nparcels_{Local} \times npvals$. Here $nparcels_{Local} = nparcels/p + bufferSize$ where p is the number of MPI ranks, and *bufferSize* is a padding to provide extra storage for additional droplets that may enter the current MPI rank. The modified data structure is illustrated in Figure 4.

For simplicity, we allocate "pdata" at the beginning of the simulation. Because of the characteristics of the geophysical flow, we expect droplet spatial distributions that are approximately homogeneous throughout the simulation. It is for this reason that we believe a fixed length allocated array to be sufficient versus a dynamically allocated link-list-based approach. The value of *bufferSize* is set to be approximately 10% of the total size of $nparcels/p$ which we have found to be generous as early tests indicate that typically only 1% of the total droplets will migrate from one MPI rank to another.

Changing the size of the "pdata" array gives the extra advantage that it is now possible to remove the if-test within the main do loop that was previously necessary to only perform calculations on droplets owned by a particular MPI rank. Instead, all droplets are placed at the beginning of the "pdata" array, and the number of active droplets $nparcels_{Active}$ is used for the extent of the main loop bounds. The elimination of the if-test does however require more detailed bookkeeping routines.

2.3.2 Bookeeping approach. An example of the necessary bookkeeping is illustrated in Figure 4. Panel (a) represents the original form of the "pdata" array. Panel (b) in Figure 4 illustrates the "pdata" array after three droplets depart, leaving three "holes" in the array. Two of these "holes" are filled by incoming droplets illustrated in panel (c). The remaining hole is filled by relocating the last active droplet in the "pdata" array which is illustrated in panel (d). The resulting "pdata" array has the same desirable contiguous storage properties as the initial version of the array.

As will be illustrated in Section 4, the use of the compacted "pdata" variable has a significant positive impact on the execution time of the GPU on the Lagrangian microphysics. Instead of launching a warp of threads where only several threads have active droplets, it is now possible to ensure that virtually all warps will be kept busy calculating active droplets.

2.3.3 Improved MPI implementation. Changes to the "pdata" array also allow the creation of a more sophisticated message-passing implementation for the droplets that minimizes the amount of data volume that is passed through MPI. This is achieved through a two-phase message-passing implementation. In the first phase of the communication, all neighboring MPI ranks exchange a single

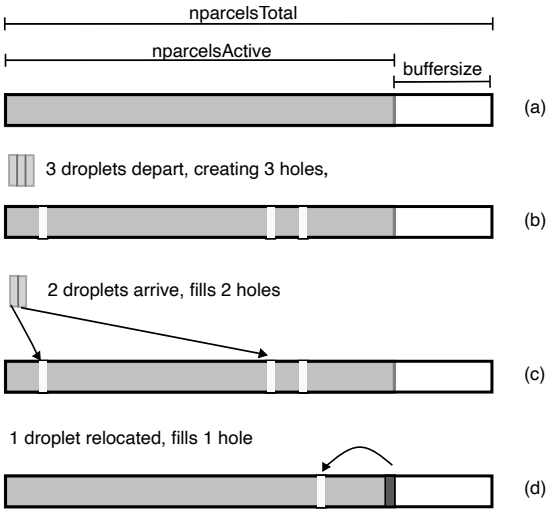

Figure 4: A example of the modified data structure and necessary operations used by the *Lagrangian droplets*.

integer with each of their eight neighbors that indicates the number of droplets that will exit an MPI rank's computational domain. Based on the information from the first phase, the second phase involves the allocation of the necessary buffers, the packing of data into these buffers, and the execution of the necessary MPI send and receive calls. The droplet information is then unpacked into the "pdata" array potentially filling the holes as described previously. While this message-passing approach requires two sets of calls to the MPI interface and the allocation of message-passing buffers, it greatly reduces the data volume versus the naive approach. Message passing volume for improved message passing approach $MsgVol_{Improved}$ is provided in equation 1

$$MsgVol_{improved} = sz_{real} * (nparcels_{Depart} * npvals) + 8 * sz_{int} \quad (1)$$

where $nparcels_{Depart}$ is the number of droplets that are departing a particular MPI rank and sz_{real} and sz_{int} are the size of floating point and integer words respectively. Because the naive implementation uses an MPI_Allreduce call, it is possible to determine the amount of message passing volume using the analytical expression for Rabenseifner's algorithm provided in [29]. The message passing volume for the naive implementation provided in equation 2 is

$$MsgVol_{naive} = 2 * sz_{real} * nparcels * npvals * \frac{(p-1)}{p}. \quad (2)$$

Since $8 << narcels_{Depart} * npvals$ we can estimate that the resulting reduction in message passing volume for the improved versus the naive implementation of *Droplet-Comm* is

$$\frac{MsgVol_{naive}}{MsgVol_{improved}} = \frac{2 * nparcels * \frac{(p-1)}{p}}{(nparcels_{Depart})} \quad (3)$$

2.3.4 Efficient bookkeeping on the GPU. One of the challenges of the more efficient message-passing approach is the need to keep track of exactly which droplets need to be sent. On the CPU this bookkeeping is achieved using a single indirect address array to keep track of the location of the departing droplet in the *pdata* data structure. A trivial loop in pseudo-code to assign this indirect address array is provided in Figure 5.

```
1  input: pd ! input array
2  output: hind ! indirect address array
3
4  initialize: i:= 0
5  foreach droplet
6      if pd(droplet) == undefined
7          update i := i + 1
8          hind(i) = droplet
```

Figure 5: CPU pseudo-code for loop that gathers array indices that satisfy a particular condition.

An efficient implementation on the GPU which requires exposing extensive loop parallelism is non-trivial due to the inherently serial nature of the loop in Figure 5. While it is possible to execute the code in Figure 5 on a single thread, the extent of the loop *nparcelsLocal* is typically quite large (10^6). A better solution is needed which allows the use of all available parallelism on the GPU.

We address this challenge through the use of a built-in CUDA function $count\_prefix()$ [8] which performs a parallel-prefix operation that counts up the number of times a particular condition holds in parallel.

We illustrate a simple example of the use of the $count\_prefix()$ operator in Figure 6. For example, consider locating the indices of an arbitrary array *pd* from Figure 6 that are equal to *U*. Instead of calculating the indirect address *hind* using a simple loop on the GPU, several intermediate steps are necessary. First the bit mask *msk* in Figure 6 is calculated which indicates the location of *U*. Next the $count\_prefix()$ operator calculates the *cindx* array which is the number of times the *msk* variable is true left of the current index. In our example in Figure 6, array elements *cindx(4:5)* are set to one, and element *cindx(6)* is set to two. Note that the *cindx* array says nothing about the presence *U* in the array element *pd(6)*. It is now possible to set the values for the indirect address *hind* array properly.

The GPU pseudo-code that uses the $count\_prefix()$ operator suggested by [21] is provided in Figure 7. Note that the assignment of the *msk* and *cindx* arrays lines 5 to 7 and line 9 in Figure 7 can be executed in parallel. The assignment of all but the last element of the *hind* array lines 11 to 15 can also be executed in parallel. The assignment of the last element of the *hind* array, which corresponds to lines 18 to 22 in Figure 5, must be performed after all other assignments of *hind*.

3 EXPERIMENTAL CONFIGURATION

We use the Derecho system located at the NCAR Wyoming Supercomputing Center (NWSC) [19]. Derecho [12] contains 2488 CPU-based nodes and 82 GPU-based nodes. The CPU-based nodes on Derecho [12] utilize a dual-socket AMD-based EPYC™ Milan 7763 processor that has 128 hardware cores. The GPU-based node on Derecho uses a single socket AMD-based EPYC™ Milan processor

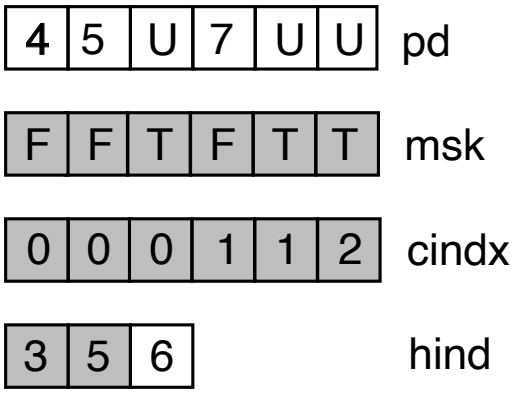

Figure 6: Example of the use of the count_prefix() algorithm for gather arrays indices. The indirect address array "hind" is calculated based on the location of the value "U" in the input array "pd". Two temporary arrays "msk" and "cindx" array are necessary to complete the calculation. Note that the shaded elements of each of the arrays can be assigned in parallel.

```
1
2  Input: pd ! input array
3  Output: hind ! indirect address array
4
5  !$acc parallel loop gang vector
6  foreach droplet := 1, numDroplets
7     set msk :=  array .eq. undefined
8
9  calculate cindx : = count_prefix(mask=msk)
10
11 ! collect the location of the special values
12 !$acc parallel loop gang vector
13 foreach droplet := 1, numDroplets-1
14     if(cindx(droplet) != cindx(droplet+1)
15         set hind(cindx(droplet+1)) := droplet
16
17
18 !Special treatment for the last value in the array
19 !$acc kernels default(present)
20 if mask(numDroplets)
21     set hind(cindx(numDroplets)+1) = numDroplets
22 !$acc end kernels
```

Figure 7: GPU pseudo-code for loop that gathers array indices that satisfy a particular condition.

and four 40GB memory NVIDIA A100 GPUs. Results are provided using Intel OneAPI 22.1, NVHPC 23.5, and GNU 12.1 compilers.

CM1 utilizes a 3-dimensional (3D) rectilinear grid that can be configured to use either cyclic, open, or rigid-wall lateral boundaries. For this work, we utilize cyclic boundaries for the horizontal "x" and "y" dimensions. The top/bottom boundaries in the "z" dimension are rigid walls, with various options for kinematic and thermodynamic boundary conditions (e.g., no-slip, free-slip, semi-slip; no flux, specified flux; etc). We refer to the number of grid points in the "x", "y", and "z" dimensions as "nx", "ny" and "nz" respectively and the size of the total grid using the notation $nx \times ny \times nz$. In this paper, we provide timing results for several different values of grid sizes including $256 \times 256 \times 256$ for a single CPU node results, and for exploring the cost of the Lagrangian droplet capability as a function of the number of droplets. We also provide multi-node and multi-GPU results using a $2048 \times 2048 \times 1024$ using 1 billion droplets. While this configuration falls significantly short of the goal of $O(100)$ droplets per grid point, it nonetheless represents a significant advance in the ability to use Lagrangian particles to understand atmospheric phenomena.

4 RESULTS

We next describe the resulting performance for the modified code that has been GPU enabled on both CPU and GPU platforms.

4.1 Base CPU performance

Recall from Section 2 that a modest number of code changes to loop structure were necessary to support efficient execution of CM1 on the GPU. These types of modifications are necessary to make the application GPU-ready. We illustrate the impact that these modifications have on the performance of the CPU code by measuring the execution time for a $256 \times 256 \times 256$ computational domain on Derecho using the original and modified source code. For this initial comparison, we configure the CM1 fluid-dynamics solver in such a way as to match the intended ASD simulation with the exception that we deactivate any of the droplet capabilities. The execution time for a single minute of simulation is illustrated in Figure 8. It is clear from Figure 8 that modifications necessary to support efficient GPU execution do not significantly impact the execution time on the CPU. We next examine the impact that the more invasive code changes that were necessary to support efficient execution of the Lagrangian droplet capability on both CPU and GPU-based nodes.

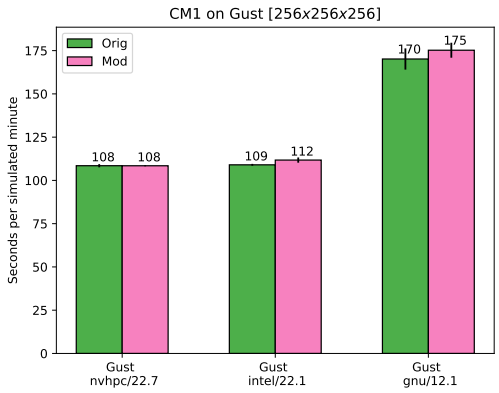

Figure 8: The execution time for CM1 using CPU-based nodes using both the original and modified code base.

4.2 Lagrangian droplets

Unlike the modest source code changes necessary to GPU-enable the *Turbulence*, *Advection*, *Acoustic Solver*, and *Diagnostics* sections of the time step loop, the changes necessary to enable the Lagrangian droplet capability are much more involved. Note that due to the presence of the Lagrangian droplet capability, the built-in "bulk" microphysics schemes performed in *BulkMicrophysicsDriver* were deactivated and will not be discussed. To quantify the impact of our code changes to *Lagrangian droplets* we use the same $256 \times 256 \times 256$ domain used in the previous section. We activate the Lagrangian droplet capability and explore the characteristics of two code implementations for different numbers of droplets.

The two different versions are: the initial version of the Lagrangian droplets which uses the naive MPI message passing implementation (BASE) and an optimized version with an improved data structure and communication operators (OPT). Because the improved Lagrangian droplet implementation, which was described in Section 2.3, involves changes to the message-passing approach, we present all results in this section using multiple nodes. For the CPU-based configurations, we report results on 8 nodes of Derecho that utilize all 128 cores per node. For the GPU-based configurations, we use a total of 8 CPU cores to drive the eight NVIDIA A100 GPUs distributed on 2 nodes of Derecho. We vary the total number of droplets from a low of 100,000 (100K) to a high of 256 million (256M). Figure 9 illustrates the total execution time for the Lagrangian droplet for the BASE, OPT implementations for various numbers of droplets on both the CPU and GPU-base nodes of Derecho. It is clear from Figure 9 that the cost of *Lagrangian droplet* calculation exhibits a linear relation for the BASE-CPU, BASE-GPU, and OPT-CPU configurations. For the OPT-GPU configuration, there appears to be a fixed overhead that is significant for small droplet counts that becomes less important at larger droplet counts. This overhead is a direct result of the use of the $count\_prefix()$ operation to gather indirect address arrays illustrated in Figure 7. Using the BASE implementation, the execution time of eight A100 GPUs is equivalent to the execution time on 1,024 AMD Milan CPU cores (see Figure 9). The GPU does enable the simulation of more droplets on the GPU because the droplet-specific storage is only replicated four times per node, or once per MPI rank, versus the 128 times per node on the CPU. In contrast, by using the improved MPI implementation the same CPU configuration can now simulate four times the number of droplets. The reduced execution time on the CPU for the OPT version is a direct result of an improved message-passing implementation and will be discussed next. On the GPU, the OPT code enables a factor of 109× reduction in execution time compared to the naive MPI implementation for 16 million droplets. This significant reduction from 177.7 seconds to 1.6 seconds results from three distinctive improvements. The first improvement, which was described in Section 2.3 involved the elimination of all host-to-device and device-to-host data movement due to the implementation of a GPU-resident bookkeeping algorithm. The second is the increase in useful work performed by the GPU. The third is a significant reduction in message-passing data volume due to the new message-passing approach. We first describe the increase in useful work performed by the GPU, followed by the reduction in message passing volume.

Recall that for the naive implementation, the extent of the main do-loop is *nparcels* resulting in the launching $nparcels/warpSz$ blocks of work where *warpSz* is the number of threads in a block or a warp. A number of these threads do not represent useful work. Reducing the extent of the main do-loop and eliminating the if-test ensures that each warp issued to the GPU now represents useful work. Furthermore, in the BASE-GPU configuration, the large droplet array is transferred between the host and device while on the OPT-GPU all calculations are GPU resident.

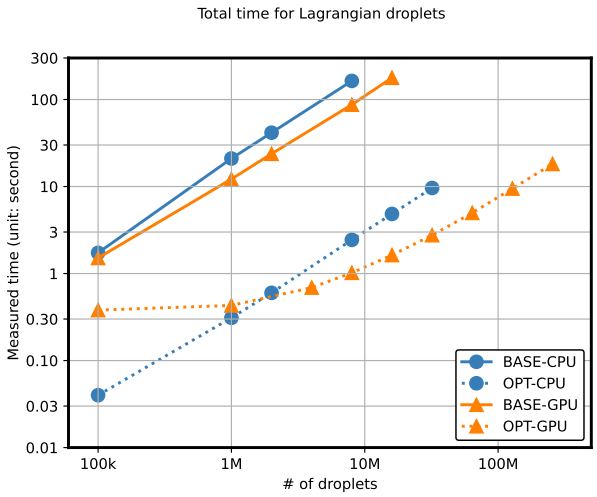

Figure 9: The computational time of *Lagrangian droplets* using the naive and improved message passing code. Results are provided for the BASE-CPU, OPT-CPU, BASE-GPU, and OPT-GPU configurations on the CPU and GPU nodes.

Figure 9 clearly illustrates that the OPT-CPU code enabled an increase in the total number of droplets due to a reduction in memory usage from a maximum of 8 million to 32 million. We next examine the detailed communication cost for the naive and improved MPI implementations on both CPU and GPU.

The communication costs are illustrated in Figure 10. In terms of the communication cost for the BASE version, the Lagrangian droplet capability on the GPU is about 4× to 6× lower than on the CPU. This difference in communication cost is primarily caused by the fact that the GPU uses fewer (8 versus 1024) MPI ranks to perform the "allreduce" operation versus the CPU.

Using the improved MPI implementation present in the OPT versus the BASE versions, we observed 3× to 487× speedup on CPU-based runs and 4× to 272× speedup on the GPU-based runs. This significant reduction in communication time is expected due to a reduction in message volume. Recall that equation 3 describes the expected reduction in message volume for the improved implementation of *Droplets-Comm*. While the value for $nparcels_{Depart}$ is dependent on the wind velocity, we have observed that typically only 1/100 of the local particles depart each MPI domain, so $nparcels_{Depart} = .01 * nparcels/p$. Therefore we expect a theoretical reduction in message volume of approximately 1400× for our particular eight MPI rank case. Unfortunately, the 1400× reduction

in message volume does not directly result in a similar improvement in execution time. Instead, we observe a still impressive 200 to 400× reduction in execution time. This more modest improvement in execution time is a direct result of the significant amount of bookkeeping necessary to support the improved MPI implementation. For example, the two-phased message-passing approach, along with the allocation and deallocation of message-passing buffers consumes time.

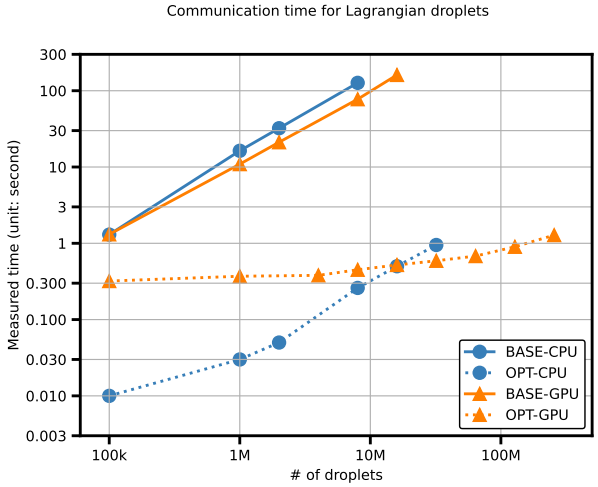

Figure 10: The communication time for *Lagrangian droplets* using the naive and improved message passing code. Results are provided for the BASE-CPU, OPT-CPU, BASE-GPU, and OPT-GPU configurations on CPU and GPU nodes.

4.3 Weak scaling on GPU

We next look at the scaling characteristics of the GPU-enabled CM1. We anticipate that the GPU-enabled CM1 will be run using a fixed physical grid with as many droplets as will fit into the device memory. While it would be possible to provide scaling results for a fixed problem size on a variety of GPUs, the selection of a single number of droplets for the entire simulation would be challenging. In particular, the number of droplets that could be used on a small number of GPUs would not accurately reflect how the code will be used in practice. Instead, to simplify the comparison, we look at a weak-scaled configuration of CM1 that uses a fixed physical grid size and droplet count per GPU. Specifically, we configure CM1 to have a $128 \times 128 \times 1024$ physical grid with 3.9 million droplets per GPU. Such a per GPU configuration matches the target resolution of $2048 \times 2048 \times 1024$ with 1 billion droplets on 256 GPUs used by the ASD project. The timing for the main sections of the CM1 code for GPU counts that range from 4 to 256 is provided in Figure 11.

Figure 11 clearly illustrates that while the total execution time is larger on 256 versus 4 GPUs, in general, CM1 does achieve acceptable weak scaling. The time to simulate a single model minute increases from 219 seconds on 4 GPUs to 319 seconds on 256 GPUs. While both the *Boundary-Exch* and *Droplets-Comm* sections, which include message passing, increase modestly in execution time, the

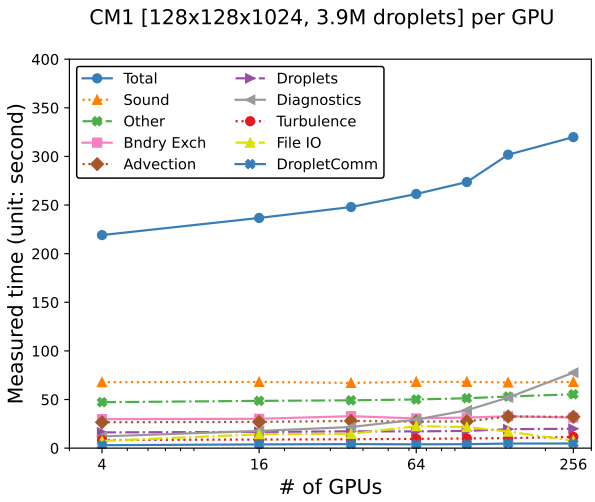

Figure 11: The execution time for one model minute for a weak scaled configuration of CM1. Each GPU has $128 \times 128 \times 1024$ physical grid points with 3.9 million droplets.

rest of the CM1 except for the *Diagnostics* section increases very modestly. The *Diagnostics* section of code, which contains a very large number of MPI reductions, accounts for approximately 65 seconds of the total 100 seconds increase in execution time.

In Figure 11 a single MPI rank uses a single GPU. NVIDIA provides the ability for multiple MPI ranks to share a single GPU. This capability is enabled through the use of the Multi-Process Service (MPS) and can be particularly useful when there is insufficient parallelism to keep a single GPU fully utilized. Using a $1024 \times 1024 \times 1024$ configuration with a total of 250M droplets we tested the impact of the MPS server on execution time. We compare the 64 GPU case from Figure 11 with MPS configurations that utilized 128, 256, and 512 MPI. These MPI rank counts correspond to two, four, and eight MPI ranks sharing a single GPU respectively. A very slight decrease in execution time of 1% is observed for the 128 MPI rank configuration, versus the 64 MPI rank configuration. The 256 and 512 MPI rank configurations are 3% and 9% slower than the 64 MPI rank configuration. Detailed investigation of the various component costs reveals that while the cost of the Acoustic Solver and the *Droplet- Microphysics* decrease with an increase in the number of MPI ranks, the cost of the *Droplet-Comm* and *Bndry Exchange* both increase. Because of the mixed impact that the use of the MPS server has on overall performance, for simplicity, we only use a single MPI rank per GPU for the remainder of the paper.

Next, we look at the overall execution time for an ASD configuration using CPU and GPU-based nodes.

4.4 A large scale configuration

We next compare 256 CPU-based nodes to 64 GPU-based nodes. Recall that each GPU-based node contains 4 A100 GPUs, resulting in a comparison of 256 CPU-based nodes to 256 GPU devices. We use a single problem size configuration with $2048 \times 2048 \times 1024$ physical grid and a total of 1 billion droplets. For this configuration,

each CPU node or GPU has a $128 \times 128 \times 1024$ physical domain with 3.9 million droplets which was described in the previous section.

The execution time in seconds for one simulated minute is provided in Table 1 for both the CPU and GPU configurations.

Table 1: Execution time for 1 model minutes in seconds and speedup for a $2048 \times 2048 \times 1024$ configuration of CM1 with 1 billion droplets using 256 CPU-based nodes and 256 GPU devices.

Section of Code	CPU	GPU	Speedup
AcousticSolver	406.4	68.1	5.97
Turbulence	53.3	11.8	4.50
Advection	156.5	27.8	5.62
File IO	2.8	7.2	0.39
Diagnostics	62.5	68.5	0.91
load-imbalance	251.3	6.1	41.48
Other	184.0	33.4	5.51
Bndry exchange	148.6	31.2	4.76
Lagrangian droplets			
Microphysics	167.4	19.0	8.82
Comm	20.9	4.2	5.02
Diagnostics	8.2	11.6	0.71
Total	1462.1	288.9	5.06

Overall, GPU enablement reduces the execution time by a factor of nearly 5.1× versus the CPU code. The cost breakdown for several sections is also provided in Table 1. Note that in addition to the sections of CM1 previously described, we also provide times for file I/O, MPI boundary exchange operations *Bndry Exchange* and the *Droplet-Comm* operators as well as time spent waiting for computational load imbalance. Note that we time the computational load imbalance by the addition of an MPI_barrier call before each group of MPI calls. Surprisingly, approximately 17% of the total time on the CPU run is spent waiting for load imbalance. To avoid the potential overhead cost of the MPI_barrier call, we utilize the Cray Programming Environment CrayPAT tool [7] which confirms our observation that approximately 20% of the total time is consumed by load-imbalance for the CPU-based runs. Interestingly, the GPU runs only spent 2.1% of the total time in computational load imbalance. Visualizations of droplet densities revealed the presence of compact regions of the computational grid with unusually large droplet counts. The larger MPI domains used by the GPU-based version reduce the overall impact of these small-scale flow inhomogeneities on execution time load imbalance. For simplicity, we indicate all other calculations not otherwise categorized as *Other*.

Excellent speedup is observed for the execution time for the *Acoustic solver*, *Advection*, *Turbulence*, *Other*, and *Lagrangian droplet* which range from 4.5× to 8.8×. Unfortunately, the other sections of CM1 do not achieve a similar speedup. In particular, File I/O and Diagnostics costs are more expensive on the GPU.

Table 1 indicates that the GPU-enabled CM1 achieves a significant reduction in execution time. What is not clear from Table 1 is how efficiently our GPU-enabled application utilizes the GPU. It is possible to evaluate code efficiency by generating a roofline analysis diagram for several of the most expensive kernels using the Nsight compute tool[18]. To simplify data collection, we use a four GPU configuration with a per GPU problem size that is consistent with the weak scaled configuration in Section 4.3. Because we are most interested in the efficiency of the computational components of the GPU-enabled CM1, we disable the rather expensive *Diagnostic* section. The roofline diagram for the five most expensive kernels is provided in Figure 12.

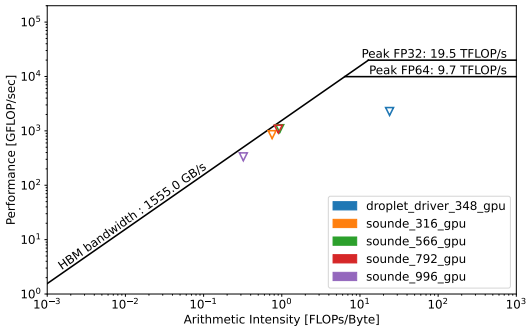

Figure 12: The roofline diagram for the five most expensive kernels.

Note that droplet_driver_348_gpu kernel which represents 8.9% of the total execution time is the main computational kernel for the *Droplet-Microphysics* section of CM1. The four sounde_* kernels, which account for 37.6% of the total execution time, represent the *Acoustic Solver*. It is clear from Figure 12 that the four sounde kernels are quite efficient as their measured floating point operation rate is very near the bandwidth limitation for the High Bandwidth Memory (HBM) used by the GPU. The droplet_driver_348_gpu kernel is significantly less efficient. The reason why the droplet_driver_348_gpu kernel is less efficient is that it is quite large (approximately 2000 lines of code) and uses a large number of registers. The next most important kernel consumes 1.6% of the total time and has characteristics similar to the sounde kernels. The remaining execution time is consumed by either message passing or a very large number of kernels that have similar characteristics to the sounde kernels. While it may be possible to improve the efficiency of the droplet_driver_348_gpu kernel by reducing the size of the kernel, it would only impact approximately 9% of the total execution time of the GPU-enabled version of CM1. We therefore conclude that our current implementation does, in general, make efficient use of the GPU.

4.5 Energy Efficiency

A key advantage of GPU-based computing is that it typically has higher energy efficiency than a CPU-based solution. It is possible to directly measure cumulative energy or instantaneous power through the use of the Cray Power Management counters [1]. The Cray Power Management counters consist of several kernel registers that are accessible through the */sys/cray/pm_counters* directory. While it is possible to access these registers through the use of CrayPAT, we used a simpler approach that captured these registers every several seconds and wrote the output to a log file. The

cumulative energy usage for both the CPU and GPU configuration described previously in Section 4.4 is provided in Table 2.

Table 2: Energy usage in kilowatt-hours (kWh) for one simulated minute of a $2048 \times 2048 \times 1024$ configuration of CM1 with 1 billion droplets using 256 CPU-based nodes and 256 GPU devices.

Component	CPU-based	GPU-based
Accelerators	-	4.39
CPU	33.30	0.49
Memory	16.92	0.57
Other energy	13.13	0.83
Total	63.36	6.27

We collect the energy usage for multiple components of the node including the CPU, memory, and the GPU accelerators if present. In addition to the component-specific registers, there is also an overall energy usage. We indicate the difference between the overall and sum of the component energy usage as *Other energy*. While it is unclear what the exact source of the *Other energy* recorded by the Cray power management counters it does appear to be quite consistent and approximately 13% on GPU-based nodes and 20% on CPU-based nodes. It is clear from Table 2 that the use of the GPU-enabled CM1 reduces energy usage significantly, by a factor of 10×. It should be noted that the accuracy of these measurements has not been independently validated using physical measurements by the authors. However, we do note that the Cray Power Management counters and those provided by the NVIDIA management library are consistent. It should also be noted that there were very small variations in energy measurements between multiple runs except for a small number of anomalous readings. Specifically, two GPU-based nodes indicated instantaneous power measurements that were not physically possible. Based on the fact that these two nodes consistently generated anomalous readings, and all other nodes generated consistent and reproducible results, we believe these nodes to have faulty energy sensors. To prevent these nodes from impacting our energy measurements, we replaced energy log files with data from nodes with healthy sensors.

5 CONCLUDING REMARKS

We have described our efforts to GPU-enable CM1 including the creation of a portable and performant Lagrangian droplet capability. This effort involved the addition of OpenACC directives to CM1 and the redesign and augmentation of the existing particle capability. The creation of an efficient Lagrangian droplet capability involved the changing of the fundamental data structure and the creation of a new message-passing capability that minimizes the amount of data moved through the high-performance network. These efforts have enabled an increase in the number of droplets that can be simulated on a CPU-based system by a factor of 4× versus the initial version of the code. The use of the GPU-enabled version of CM1 further reduces the time to perform scientifically relevant simulations by a factor of 5.1× versus CPU-based nodes and represents a step forward towards leveraging the Lagrangian-based microphysics framework for understanding a wide variety of atmospheric phenomena.

Despite the progress made in reducing the cost of CM1 and its Lagrangian droplet capability, additional work is still necessary. The cost of the *Diagnostics* is disappointingly large and can likely be improved. Additionally, we expect that the efficiency of the Lagrangian microphysics calculations can be improved.

OPEN RESEARCH

The source code, log files, plotting scripts, and example namelist and runscripts used for this paper are available on Zenodo [11]. Instructions on how to build and execute CM1 on both CPU and GPU-based notes as well as descriptions of how data from log files are converted to plots are also provided.

ACKNOWLEDGMENTS

We would like to acknowledge high-performance computing support from Derecho (https://doi.org/10.5065/qx9a-pg09) provided by NSF NCAR's Computational and Information Systems Laboratory, sponsored by the National Science Foundation grant 1852977. We would also like to thank Carl Ponder for sharing his considerable GPU programming experience.

REFERENCES

[1] Bareford, M.: Monitoring the Cray XC30 power management hardware counters. Proceedings of CUG2015, Seattle, WA, USA (2015)
[2] Bieringer, P.E., Piña, A.J., Lorenzetti, D.M., Jonker, H.J.J., Sohn, M.D., Annunzio, A.J., Fry, R.N.: A graphics processing unit (GPU) approach to large eddy simulation (LES) for transport and contaminant dispersion. Atmosphere **12**(7) (2021). https://doi.org/10.3390/atmos12070890
[3] Bryan, G.H., Worsnop, R.P., Lundquist, J.K., Zhang, J.A.: A simple method for simulating wind profiles in the boundary layer of tropical cyclones. Boundary-Layer Meteorology **162**(3), 475–502 (March 2017). https://doi.org/10.1007/s10546-016-0207-0
[4] Bryan, G.H., Wyngaard, J.C., Fritsch, J.M.: Resolution requirements for the simulation of deep moist convection. Monthly Weather Review **131**(10), 2394–2416 (October 2003). https://doi.org/10.1175/1520-0493(2003)131<2394:RRFTSO>2.0.CO;2
[5] Chandrakar, K.K., Morrison, H., Grabowski, W.W., Bryan, G.H., Shaw, R.A.: Supersaturation variability from scalar mixing: Evaluation of a new subgrid-scale model using direct numerical simulations of turbulent Rayleigh–Bénard convection. Journal of the Atmospheric Sciences **79**(4), 1191–1210 (April 2022). https://doi.org/10.1175/JAS-D-21-0250.1
[6] Chen, X., Bryan, G.H., Zhang, J.A., Cione, J.J., Marks, F.D.: A framework for simulating the tropical-cyclone boundary layer using large-eddy simulation and its use in evaluating PBL parameterizations. Journal of the Atmospheric Sciences (September 2021). https://doi.org/10.1175/JAS-D-20-0227.1.
[7] Cray Performance measurement and Analysis Tools. https://hpctools.readthedocs.io/en/latest/perftools.html (2022)
[8] cuTENSOR: A high-performance CUDA library for tensor primitives. https://docs.nvidia.com/cuda/cutensor/latest/index.html (2023)
[9] Deardorff, J.W.: Stratocumulus-capped mixed layers derived from a three-dimensional model. Boundary-Layer Meteorology **18**(4), 495–527 (June 1980). https://doi.org/10.1007/BF00119502
[10] Dennis, J.M., Baker, A.H., Dobbins, B., Bell, M.M., Sun, J., Kim, Y., Cha, T.Y.: Enabling efficient execution of a variational data assimilation application. The International Journal of High Performance Computing Applications (2022). https://doi.org/10.1177/10943420221119801
[11] Dennis, J.M., Sun, J., Voelz, S., Bryan, G., Richter, D.: A portable and efficient lagrangian particle capability for idealized atmospheric phenomena. Dataset on Zenodo, https://10.5281/zenodo.11062325 (2024)
[12] Computer and Information Systems Laboratory, Derecho: HPE Cray EX system (university community computing). https://doi.org/10.5065/qx9a-pg09 (2023)
[13] Gettelman, A., Morrison, H., Eidhammer, T., Thayer-Calder, K., Sun, J., Forbes, R., McGraw, Z., Zhu, J., Storelvmo, T., Dennis, J.: Importance of ice nucleation and precipitation on climate with the parameterization of unified microphysics across scales version 1 (pumasv1). EGUsphere **2022**, 1–28 (2022). https://doi.org/10.5194/egusphere-2022-980
[14] Grabowski, W.W., Morrison, H., Shima, S.I., Abade, G.C., Dziekan, P., Pawlowska, H.: Modeling of cloud microphysics: Can we do better? Bulletin of the American Meteorological Society 100 **100**(4), 655–672 (2019). https://doi.org/10.1175/bams-d-18-0005.1

[15] van Heerwaarden, C.C., van Stratum, B.J.H., Heus, T., Gibbs, J.A., Fedorovich, E., Mellado, J.P.: MicroHH 1.0: a computational fluid dynamics code for direct numerical simulation and large-eddy simulation of atmospheric boundary layer flows. Geoscientific Model Development **10**(8), 3145–3165 (2017). https://doi.org/10.5194/gmd-10-3145-2017

[16] Maronga, B., Gryschka, M., Heinze, R., Hoffmann, F., Kanani-Sühring, F., Keck, M., Ketelsen, K., Letzel, M.O., Sühring, M., Raasch, S.: The parallelized large-eddy simulation model (PALM) version 4.0 for atmospheric and oceanic flows: model formulation, recent developments, and future perspectives. Geoscientific Model Development **8**(8), 2515–2551 (2015). https://doi.org/10.5194/gmd-8-2515-2015

[17] Michalakes, J., Vachharajani, M.: GPU acceleration of numerical weather prediction. Parallel Processing Letters **18**(04), 531–548 (2008). https://doi.org/10.1142/S0129626408003557

[18] Nsight Compute Developer tools documentation. https://docs.nvidia.com/nsight-compute/ (2023)

[19] NCAR-Wyoming Supercomputing Center. https://en.wikipedia.org/wiki/NCAR-Wyoming_Supercomputing_Center (June 2022)

[20] Ponder, C.: Folded maxloc algorithm. Personal Conversation (2022)

[21] Ponder, C.: Cutensor based indirect address gather operation. Personal Conversation (2023)

[22] Rotunno, R., Bryan, G.H.: Numerical simulations of two-layer flow past topography. part II: Lee vortices. Journal of the Atmospheric Sciences **77**(3), 965–980 (March 2020). https://doi.org/10.1175/JAS-D-19-0142.1

[23] Sauer, J.A., Muñoz-Esparza, D.: The FastEddy® resident-GPU accelerated large-eddy simulation framework: Model formulation, dynamical-core validation and performance benchmarks. Journal of Advances in Modeling Earth Systems **12**(11), e2020MS002100 (2020). https://doi.org/https://doi.org/10.1029/2020MS002100

[24] Schalkwijk, J., Griffith, E.J., Post, F.H., Jonker, H.J.J.: High-performance simulations of turbulent clouds on a desktop PC: Exploiting the GPU. Bulletin of the American Meteorological Society **93**(3), 307 – 314 (2012). https://doi.org/10.1175/BAMS-D-11-00059.1

[25] Shima, S., Kusano, K., Kawano, A., Sugiyama, T., Kawahara, S.: The super-droplet method for the numerical simulation of clouds and precipitation: A particle-based and probabilistic microphysics model coupled with a non-hydrostatic model. Quarterly Journal of the Royal Meteorological Society **135**, 1307–1320 (2009). https://doi.org/10.1002/qj.441

[26] Skamarock, W.C., Klemp, J.B.: The stability of time-split numerical methods for the hydrostatic and the nonhydrostatic elastic equations. Monthly Weather Review **120**(9), 2109–2127 (1992)

[27] Stern, D.P., Bryan, G.H., Lee, C., Doyle, J.D.: Estimating the risk of extreme wind gusts in tropical cyclones using idealized large-eddy simulations and a statistical–dynamical model. Monthly Weather Review **149**(12), 4183–4204 (2021). https://doi.org/10.1175/MWR-D-21-0059.1

[28] Sun, J., Dennis, J.M., Mickelson, S.A., Vanderwende, B., Gettelman, A., Thayer-Calder, K.: Acceleration of the parameterization of unified microphysics across scales (PUMAS) on the graphics processing unit (GPU) with directive-based methods. JAMES **15** (2023). https://doi.org/https://doi.org/10.1029/2022MS003515

[29] Thakur, R., Rabenseifner, R., Group, W.: Optimization of collective communication operations in MPICH. The International Journal of High Performance Computing Applications **19** (September 2016). https://doi.org/10.1177/1094342005051521

[30] Wicker, L.J., Skamarock, W.C.: Time-splitting methods for elastic models using forward time schemes. Monthly Weather Review **130**(8), 2088–2097 (August 2002). https://doi.org/10.1175/1520-0493(2002)130<2088:TSMFEM>2.0.CO;2.