GPU Implementation of the OVERFLOW CFD Code

Charles W. Jackson*
NASA Langley Research Center, Hampton, Virginia, 23681

David Appelhans†
NVIDIA, Denver, Colorado, 80302

Joseph M. Derlaga‡ and Pieter G. Buning§
NASA Langley Research Center, Hampton, Virginia, 23681

The high-performance computing (HPC) landscape is quickly changing to systems where most of the performance comes from specialized chips, specifically graphics processing units (GPUs). Such GPU systems are throughput machines, where efficient use of the GPU often requires code refactoring to expose a few orders of magnitude more fine grain parallelism than was previously used on the CPU. Recent modifications to OVERFLOW, an overset, structured grid, computational fluid dynamics flow solver, written in Fortran will be presented. These modifications include both code modernization efforts and algorithmic changes to enable OVERFLOW to efficiently utilize GPUs. Many of these algorithmic changes would likely also be applicable for other structured grid, stencil-based codes wanting to utilize GPUs. The capabilities that have been ported to run on the GPUs are presented, along with the performance gains of the GPU version relative to the CPU version of OVERFLOW.

I. Introduction

OVERFLOW is an overset, structured grid, computational fluid dynamics (CFD) flow solver [1, 2]. OVERFLOW is widely used by NASA, other government agencies, industry, and academia to predict the flow solutions for many different applications, from rockets and capsules to transport aircraft and helicopters to a wide variety of different concept vehicles and research topics. OVERFLOW has been continually developed and optimized to run efficiently on multicore central processing units (CPUs) over the last several decades. This performance is one of the hallmarks of OVERFLOW as a CFD solver and is one of the reasons for its wide use. OVERFLOW's ability to solve problems quickly with a low memory footprint is a key enabler for its many use-cases such as design work where different ideas can be explored quickly, database construction where many conditions can be simulated to help cover the design space, and detailed simulations where large grids are necessary to accurately model the relevant flow physics.

In recent years, the high-performance computing (HPC) community has been moving toward more and more specialized hardware, such as graphics processing units (GPUs). This change is coming at all levels of the computing world, from the largest supercomputer in the world, Frontier [3], to midrange clusters at companies, government labs, and universities, all the way down to individuals with a single GPU in their workstation. GPUs have changed the landscape of computing as GPUs are able to offer much higher performance than CPUs at a lower system clock and power envelope. These accelerators achieve this performance by having tens of thousands of parallel operations in flight, effectively hiding a relatively low system clock (i.e., latency). GPUs typically also have higher memory bandwidth available to them, which is very beneficial to memory-bound codes like OVERFLOW. With this changing landscape in mind, the authors have begun the process of porting the code so that it is able to run on these new systems. While there are a wide variety of capabilities and solver techniques available in OVERFLOW, this paper, and the GPU port so far, is focused on the central-differencing scheme [2] with the diagonalized form of the implicit approximate factorization

*Research Scientist, Computational AeroSciences Branch, AIAA Member
†Senior Developer Technology Engineer
‡Research Scientist, Computational AeroSciences Branch, AIAA Member
§Senior Research Scientist, Computational AeroSciences Branch, AIAA Associate Fellow

Table 1 Compatibility table of different programming paradigm support on different vendor's GPUs. Reproduced with slight modification from Herten [14].

	CUDA		HIP		SYCL		Standard		OpenMP		OpenACC		Kokkos	
GPU Vendor	C	F	C	F	C	F	C	F	C	F	C	F	C	F
NVIDIA	●	●	▼	⊘	▲	⊘	●	●	◆	◆	●	●	▲	⊘
AMD	▼	⊘	●	⊘	▼	⊘	▲	⊘	●	●	▲	▲	▲	⊘
Intel	▼	⊘	⊘	⊘	●	⊘	⊘	◆	●	●	⊘	⊘	▲	⊘

● Full vendor support ▲ Comprehensive support, not from vendor
◆ Partial vendor support ⊘ Very limited or no support
▼ Indirect vendor support C = C++ (sometimes C), F = Fortran

scheme [4]. These options are relatively simple, allowing quick development as we learn how to best utilize this different hardware. With this GPU update, OVERFLOW users can take advantage of this hardware and perform their analysis faster or solve larger problems in the same amount of time.

A. GPU approaches

There are several approaches available to program for GPUs. Each vendor has their own preferred method of programming their hardware, which gives you the most access to the low-level optimizations of these systems. In this work, we will be focused on using NVIDIA GPUs but would like to be able to have a portable solution in the long term. To get the most control over GPU kernel execution on NVIDIA GPUs, one should use CUDA. CUDA [5] is a C++-based programming model designed to give programmers direct access to the hardware to efficiently run on NVIDIA GPUs. While CUDA is proprietary and designed for NVIDIA GPUs, other vendors do have some support to convert CUDA code to their preferred paradigms. The NVIDIA HPC SDK compilers also support CUDA Fortran [6], a collection of libraries and Fortran language extensions that allow writing CUDA directly in Fortran subroutines. Unfortunately, the NVIDIA NVHPC and IBM XLF compiler are the only compilers that support CUDA Fortran, and they only compile for NVIDIA GPUs. AMD provides a CUDA C analogue with their Heterogeneous-compute Interface for Portability (HIP) [7], written as a C++ API to expose capabilities of their GPU hardware. Because of its similarities to CUDA, AMD provides converters that will convert CUDA code into HIP. As a portability layer, HIP allows the code to be compiled to run in parallel on the CPU, AMD GPUs, and NVIDIA GPUs. However, there is no equivalent to CUDA Fortran to easily expose HIP capabilities directly in Fortran code.

A programmer can also use directive-based programming models to execute code on GPUs. The most common of these approaches are OpenACC [8] and OpenMP [9], both of which support C/C++ and Fortran. Both paradigms allow the programmer to expose the parallelism in your program through pragmas or directives, which appear like comments that can be ignored by the compiler unless enabled. Ideally this means that you can have a single code base that the compiler can compile for CPUs, GPUs, or other accelerators if the compiler supports it. In our experience, unfortunately, code or algorithm changes from the CPU-optimized version may be necessary to get the best performance out of the GPU, so a single code base may not be possible.

Another common approach for GPU programming is using a portability programming model such as SYCL [10], OpenCL [11], RAJA [12], Kokkos [13], and many others. These approaches have various benefits, however, nearly all are designed for C++ programs (some do have token support for Fortran).

The final set of approaches that can be used are standard parallelism. The latest versions of both C++ and Fortran support parallel execution in the standard. However, support for these from the different compiler vendors on various hardware is mixed, there are many restrictions on what one is allowed to do within a parallel region, and code modification is likely necessary to get good performance.

All approaches listed above (and others) have various levels of support from the hardware and compiler vendors on the various GPU hardware available. For a good overview of the current state of support (last updated in November 2022), see the helpful chart reproduced in Table 1 from Herten [14]. For more discussion about the classification of each of these levels of support, see the full discussion on their website.

Since OVERFLOW is a Fortran code, and at this stage we are not prepared to rewrite the entire codebase in another language, the choices were mainly limited to Fortran standard parallelism, OpenACC, OpenMP, and CUDA Fortran.

The Fortran standard parallelism, i.e., `DO CONCURRENT`, does not expose any information about data movement nor does it allow asynchronous execution between the host and device, thus it would likely have to be coupled with another technique such as OpenACC or CUDA. While the OpenMP standard has supported executing code on GPUs since version 4.0, which was released in 2013, when we began this work in 2019, there was limited support for this by compilers. There are also several features from OpenACC, such as asynchronous streams, that are not available in OpenMP. CUDA Fortran works well but is tied to NVIDIA GPUs. We have also noticed certain CUDA features and routines were not available directly in CUDA Fortran and were only accessible through handwritten C interfaces, although support for these have improved in recent years. OpenACC is well-supported by the NVIDIA HPC Fortran compiler which also allows for convenient interoperability with CUDA if certain kernels required extra optimization. Thus, OpenACC was selected as the GPU approach to be used to initially port OVERFLOW to GPUs. There are a limited number of routines that benefited from more hand tuning, which has been done using CUDA Fortran.

While we feel like this was the best approach for us now, this may change in the future as software stacks and programming models improve. There are definite benefits of switching to a programming language such a C++ with wider support, more development, and specific features that might be advantageous. Many other codebases have taken this approach, moving away from Fortran and implementing their codes in C++. In our initial testing, re-writing the CPU code in C++ resulted in slowdowns due to a lack of compiler optimizations, but this may change as compilers continue to improve. At this point in time, we did not think the benefits outweighed the cost of converting the entire codebase to C++. Another option that will be considered in the future is moving to OpenMP to gain access to other vendor's GPU hardware if OVERFLOW remains to be written in Fortran.

B. Computing Systems

Several systems were used for the evaluation of the code performance. K4, a midrange computing cluster at the NASA Langley Research Center, was used for the CPU performance metrics. This system has dual-socket nodes with 20 core 2.40 GHz Intel Xeon Gold 6148 Skylake processors, with a 4X EDR InfiniBand interconnect (100 GB/s). K4 has several GPU nodes, some of which have V100 GPUs and some have V100S GPUs. The V100 nodes are dual-socket nodes with 20 core 2.40 GHz Intel Gold 6148 Skylake CPUs with four NVIDIA V100 GPUs connected with NVLink. The V100S nodes are dual-socket nodes with 16 core 2.90 GHz Intel Xeon Gold 622R CPUs with two PCI-e connected NVIDIA V100S GPUs. K5, another midrange computing cluster at NASA Langley, was used for testing with newer A100 40 GB and 80 GB cards. This system has dual-socket nodes with 64 core AMD EPYC 7742 processors with 8 A100 cards connected with NVLink. These nodes are connected with a 4X HDR InfiniBand interconnect (200 GB/s). The H100 results were run using DGXH100 systems which consist of 112 core Intel Sapphire Rapids CPUs and 8 NVLINK 4.0 connected H100-80GB-HBM3 NVIDIA GPUs. All results presented here were run with NVHPC 23.7.

C. OVERFLOW Description

There are several aspects of OVERFLOW that are important for understanding some of the work in this paper. OVERFLOW operates on a grid system which consists of several overlapping grid blocks. Each of these grid blocks are structured grids which use a `(J,K,L)` indexing of the points. The size of these blocks can vary greatly depending on the problem and how the grids were generated. In several sections of this paper, we discuss a "plane" of one of these grid blocks. This can refer to any one of the 2D slices through these grid blocks with a constant `J`, `K`, or `L` value. We also discuss a "pencil" of the grid, which in this paper describes a single grid line, where two of the indices are constant.

When running on multiple MPI ranks, it is necessary to distribute the grid blocks onto the different processes. For good load-balancing, OVERFLOW will split up the larger blocks into smaller blocks and then evenly distribute these grid blocks to the ranks in a round-robin fashion. Because of this splitting, running on different numbers of ranks can slightly change the answer as well as have performance impacts. This splitting process can achieve very good load-balancing for a wide variety of grid sizes and configurations, however, it does not attempt to minimize the amount of communication or minimize the number of grid blocks created. These are areas of potential future work within the code especially as the GPU is changing how many MPI ranks would be used to run these problems.

The communication between the grid blocks (including the split grids) happens using the chimera boundary exchange procedure. Because the grids are overset, each receiver point is matched to a cell in an overlapping grid that contains that point in space. An interpolation stencil is created to interpolate from the donor stencil to the receiver point. In this paper, because the grids are fixed, these interpolation stencils and weights are also fixed. After each time step, the solution is interpolated on each grid and then communicated to the receiver grids. If the grids are on the same MPI rank this is a simple copy, and if they are different MPI ranks this happens through an `MPI_Send` call.

II. GPU Optimizations

During the porting process, several lessons were learned on how to best utilize the GPU that will be presented here. Some of these changes were simple modifications, while others required significant refactoring of the codebase to incorporate. Many of these optimizations were explored in miniapps: small applications that are representative of a portion of the overall code base that are easy to modify, check for correctness, and share with collaborators. These simplified codes enabled quick learning, fostered a lot of experimentation, and were critical to our success porting OVERFLOW to GPUs. The next sections describe many of the modifications made and approaches that were adopted to get good performance on the GPU.

A. Movement of Data

Current GPU-based HPC systems have a CPU (or multiple) in charge of running the main executable. A GPU (or multiple) is attached to this CPU and work is dispatched to it as an accelerator. Both the CPU host and GPU device have their own memory attached to them with discrete memory spaces. Because the memory spaces are physically separate, care must be taken to ensure that memory in the different execution spaces (device or host) is correct and up to date when it is accessed. This can represent a significant burden on the programmer to keep track of pointers to data on the device and on the host and correctly update both versions of the data. To address this issue, NVIDIA has developed Unified Memory [15]. Unified Memory allows the separate address spaces of the GPU and CPU to be "unified" into one, becoming transparent to the user. When the device tries to use a managed pointer on the GPU, if the memory on the GPU is not up to date with the CPU, the attempted data access triggers a page-fault and the memory is copied to the GPU before continuing. The reverse is also true when out-of-date memory is accessed on the CPU. By using Unified Memory, our initial GPU implementation was very easy since we did not have to explicitly implement the data movement. However, this was just a stepping stone because, as programmers, we have better knowledge of what data are needed where and when. Thus, we quickly began using the OpenACC data directives to explicitly move data to and from the GPU. This allows the data to be present on the GPU before they are needed without causing the stall of page-faults to move the data, reducing the overall performance of the kernel. Since we are handling the data movement, we also have explicitly set the `present(data)` clause for all kernels to avoid unnecessary implicit data checks and movements.

OpenACC has several directives to control the movement of data. Since nearly all work is happening on the GPU, our data movement is fairly simple. Before we run any GPU kernels, data that are required to be initialized are initialized on the CPU and copied to the GPU, and variables that will be set by the GPU kernels are allocated on the GPU. If any data are modified on the GPU and later needed on the CPU (for output or post-processing), they can be copied between the CPU and GPU using *`!$acc update`* pragmas. Some examples of these pragmas are illustrated in Listing 1.

There is one major caveat with these data directives when it comes to derived types with allocatable or pointer members. For example, consider the derived type `grid_t` (shown in Listing 2) with an allocatable member variable `xyz`. When a `grid_t` object is copied to the GPU, OpenACC copies the values of `jd`, `kd`, `ld`, and the pointer to the allocated `xyz` array. However, because the `xyz` array was allocated on the CPU, the pointer will still be pointing to the CPU memory space and the GPU will not be able to access the elements in `xyz`. Thus, one needs to first copy in the instance of grid, and then also copy in the allocatable (or pointer) member variables (supported since OpenACC 2.6). For simplicity, we have wrapped this in a copy routine as shown in Listing 3 with a corresponding `copy_grid_from_device` routine.

Listing 1 Example of a OpenACC data region.

```
!$acc enter data async                &
!$       copyin( a,b,c ) create( d )
...
!$acc update self( b )
! Use and update b on the CPU
...
!$acc update device( b ) async
...
!$acc exit data async                 &
!$       copyout( d )                 &
!$       delete( a,b,c )
```

Listing 2 Example of a derived type with an allocatable member variable.

```
type grid_t
   integer :: jd, kd, ld
   real, allocatable,                &
         dimension(:,:,:,:) :: xyz
end type grid_t
```

Listing 3 Example of a copy routine for moving data types with allocatable members.

```
subroutine copy_grid_to_device( grid )
   type(grid_t), dimension(:), intent(inout) :: grid
   integer :: ig
!$acc begin data region async copyin(grid) async
   do ig = 1, size(grid)
!$acc begin data region async copyin(grid(ig)%xyz) async
   end do
end subroutine copy_grid_to_device
```

NVIDIA also has an extension to the OpenACC standard called Deep Copy to address this complication. If the `deepcopy` compiler flag is supplied, their compilers will attempt to copy all member variables for you (like we have done manually). When we began this work, there were several compiler issues with deepcopy that meant it was not able to be used in our application. These issues may have been fixed in more recent versions of the NVHPC compiler, but we have not revisited this approach since a workaround was found and deepcopy is nonstandard.*

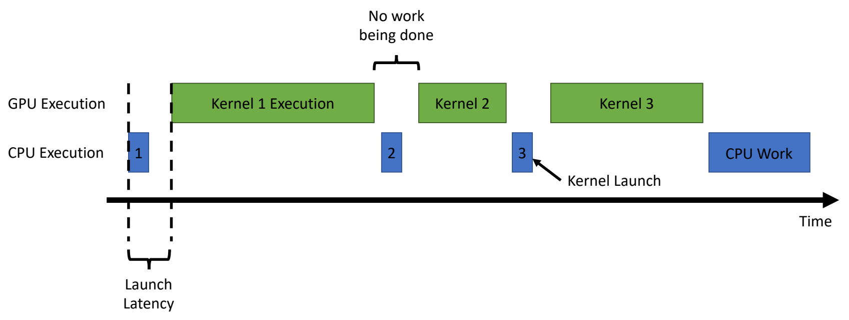
B. Asynchronous Launches and Streams

Even when running code on the GPU, the CPU is still in control of the whole program and assigning work to the GPU. The default behavior for OpenACC is that the CPU will assign work to a GPU (launch a kernel or GPU parallel region) and then wait for that work to be finished before moving on. However, there is some amount of latency between when the CPU launches the kernel and when the GPU can actually start executing the kernel (launch latency). Thus, if the CPU waits for the GPU to finish before starting to issue the next set of work on the GPU, as shown in Figure 1a, it will accumulate these latencies as the GPU waits for the CPU to issue more work as shown in Figure 1a. The latency becomes important when the kernel execution times are very small, which is the case for some kernels in OVERFLOW, especially when the grid blocks are small.

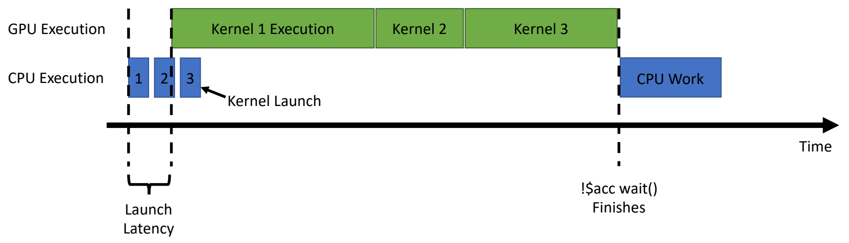
Several approaches were used to hide this launch latency. First, OpenACC allows the CPU to launch kernels (or do memory movement, etc.) asynchronously with the GPU by adding the `async` clause to the OpenACC directives. This allows the CPU to continue running after it has launched a kernel, thus it can launch multiple kernels in a pipeline. Eventually, the CPU will have to wait for some result from the GPU, so OpenACC provides the `!$acc wait` directive. This means that the CPU will wait for all submitted work on the GPU to complete before continuing as seen in Figure 1b. While there is still launch latency for the first kernel, the launch latency for the subsequent kernels is hidden by the execution of earlier kernels.

Asynchronous launches in OpenACC work by placing each of the launched kernels into a CUDA stream on the GPU. A stream is a sequence of GPU actions that will execute on the GPU in the order they are issued. This means that kernel 1 will finish before kernel 2 begins without coordination from the CPU. However, if kernel 2 does not depend on data from kernel 1, it can be launched on a separate stream. Then if a third kernel depends only on kernel 2, it can be put in the same second stream. One possible version of this situation is depicted in Figure 1c. Note that using streams does not force the GPU to do anything specific but is instead providing more information to the GPU scheduler about what is allowed to happen. For instance, kernel 2 could start before kernel 1, or, more likely, the GPU may not overlap any of the kernels. The GPU is unlike to overlap kernels very much if one of the kernels is computationally expensive or utilizes a large portion of the GPU. By using different streams, the programmer is only communicating that operations in different streams can be done independently; the order of execution is up to the GPU scheduler to determine when the kernels in the different streams will actually run. In OpenACC, the stream to place a kernel in is specified by adding the `async(stream_number)` clause to the directive. One can also change the default OpenACC stream using the `acc_set_default_async` function (this is the method most used in OVERFLOW). Another further optimization that was investigated was launching the kernels in parallel on the CPU using OpenMP. In very specific cases, this helps limit the initial latency to get all streams going (note in Figure 1c the second stream must wait until the CPU has launched both the first and second kernels). While helpful in these specific instances, for the most part, the

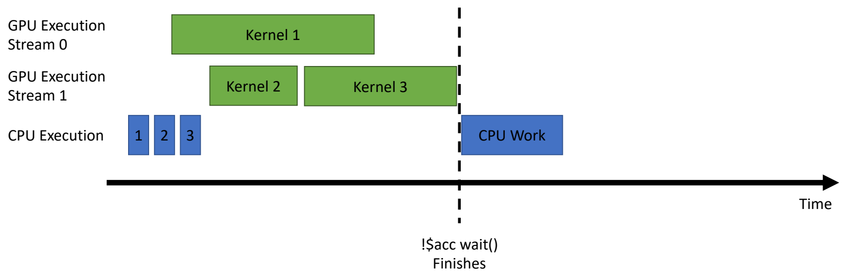
*There are proposals to add this to the standard in the future.



(a) Default blocking behavior. Launch latency prevents the GPU from being busy doing work.



(b) Asynchronous launching of kernels. The launch latency of kernels 2 and 3 is hidden by the execution time of kernel 1.



(c) Asynchronous launching to multiple streams. Kernel 3 will run after kernel 2 finishes but independent of kernel 1.

Fig. 1 Different kernel launch strategies.

other optimizations discussed below minimized the need for launching kernels by parallel CPU threads.

C. Loop Grouping and Grid Batching

Modern GPUs achieve their impressive performance by hiding latency through massive amounts of parallelism. For example, NVIDIA GPUs have about one hundred (depending on the specific GPU) Streaming Multiprocessors (SM) and each SM can support up to 2048 threads if each thread uses a small portion of the available SM resources (32 registers per thread among other things). In practice, CFD kernels such as those in OVERFLOW require 128 or more registers per thread, which means that each SM can only support 512 active threads (25% maximum occupancy). A V100 has 80 SMs, so to fully utilize the GPU when running OVERFLOW on a V100 we need to expose at least 80×512-way parallelism. In practice, this is a minimum and you would like to have a few multiples of this to avoid slowdowns associated with SMs completing work at different times.

Most of the loops in OVERFLOW are like those in Listing 4, where each grid of size `(JD,KD,LD)` is solved completely before moving on to the next grid. The main work is done, not on the whole grid, but rather, on a single plane of data (either a `K`- or `L`-plane for efficiency). However, since typical grids in OVERFLOW do not have `K`- and `L`-planes with 40,000 way parallelism (that many points in a plane), the GPU version of the code needed to move the

Listing 4 Loop structure of OVERFLOW on CPUs.

```
subroutine time_step()
    do ig = 1, ngrids
        call do_everything_on_a_grid()
    end do
end subroutine time_step

subroutine do_everything_on_a_grid()
!$omp loop
    do l = 1, ld
        ! Work on an l-plane
        call sub1(l,...)
        call sub2(l,...)
        call sub3(l,...)
        ...
    end do
!$omp loop
    do k = 1, kd
        ! Work on a k-plane
        call sub1_k(k,...)
        ...
    end do
end subroutine do_everything_on_a_grid

subroutine sub1(l,...)
    do k = 1, kd
    do j = 1, jd ! Vectorized loop
        ! Do only one thing
    end do
    end do
end subroutine sub1
```

Listing 5 First version of the loop structure for OVERFLOW on GPUs. Works well for large grids.

```
subroutine time_step()
    do ig = 1, ngrids
        call do_everything_on_a_grid()
    end do
end subroutine time_step

subroutine do_everything_on_a_grid()
    ! Work on a grid
    call sub1()
    call sub2()
    call sub3()
    ...
end subroutine do_everything_on_a_grid

subroutine sub1()
!$acc parallel loop collapse(3) async
    do l = 1, ld
    do k = 1, kd
    do j = 1, jd
        ! Do only one thing
    end do
    end do
    end do
end subroutine sub1
```

L-loop down a level as shown in Listing 5. This works well when you have large grids with enough points that can be computed in parallel. However, if you have small grids, such as those that are created by OVERFLOW's adaptation process or hand-crafted around small geometric features, a single grid does not have enough parallel work to fill the GPU, and the launch latency becomes a significant portion of the runtime. Even with multiple CPU threads launching kernels simultaneously to fill the GPU, the GPU is so fast that the CPU cannot launch the kernels fast enough, causing an extreme slowdown. To expose more parallelism for these small grid cases, the kernels were modified to work on a batch of grids, as shown in Listing 6. Operating on multiple grids in a single kernel has two main benefits: (1) it exposes more parallelism for better performance on GPUs, and (2) it gives each kernel more work and reduces the number of kernel launches allowing the CPU to launch kernels fast enough to hide their latency. However, there are some additional costs associated with this loop structure such as an increase in the number of registers used. Thus, OVERFLOW uses a mixture of both versions of GPU loops. Some kernels operate on a whole batch of grids while other kernels whose runtime is much greater than the several microseconds launch latency only operate on a single grid.

D. Data Ordering

Another important consideration for good GPU (and CPU) performance is how data are being stored and accessed. For the best performance on GPUs, one wants a set of consecutive threads to access consecutive data locations, up to the size of an L1 cache line. This is known as coalesced data accesses and allows a single cache line load/store operation to serve many threads. For example, in Listing 7, `A` would be accessed in a coalesced manner while the load from `B` would be uncoalesced (strided) since, in Fortran, the first index varies the fastest. Because care was taken in the CPU version of OVERFLOW to ensure good CPU vectorization performance, most of the data structures were already in the

Listing 6 **Second version of the loop structure for OVERFLOW on GPUs. Works well for small grids.**

```
subroutine time_step()
    do ib = 1, nbatch
        call do_everything_on_a_batch(batch(ib),...)
    end do
end subroutine time_step

subroutine do_everything_on_a_batch(batch,...)
    ! Work on a batch of grids
    call sub1(batch,...)
    call sub2(batch,...)
    call sub3(batch,...)
    ...
end subroutine do_everything_on_a_batch

subroutine sub1(batch,...)
!$acc parallel loop gang collapse(2) async    &
!$acc    present(batch,grid)                  &
!$acc    private(ig)
    do iig = 1, batch%ngrids
    do l   = 1, batch%lmax
        ! Gang parallelism over grids in a batch and l-planes
        ig = batch%ig_map(iig)
        if( l <= grid(ig)%ld ) then

!$acc loop vector collapse(2)
            do k = 1, grid(ig)%kd
            do j = 1, grid(ig)%jd
                ! Vector parallelism within an l-plane
                ! Do only one thing
            end do
            end do

        end if
    end do
    end do
end subroutine sub1
```

Listing 7   Example of coalesced and uncoalesced data access.

```
!$acc parallel loop collapse(2) async  &
!$acc    present(A,B)
do k=1,kd
do j=1,jd
    x = A(j,k) ! Coalesced read
    y = B(k,j) ! Uncoalesced read
end do
end do
```

Listing 8   Loop structure for the scalar pentadiagonal solve.

```
! Loop over all J-pencils in parallel
!$acc parallel loop collapse(2) async
do l = 1, ld
do k = 1, kd
    ! Solve this pencil with a single thread serially
    ! Forward Elimination
    do j = 1, jd
       ! want to access the main diagonal coalesced
       c1 = C(k,j,l)
        ...
    end do
    ! Backward Propagation
    do j = jd, 1, -1
        ...
    end do
end do
end do
```

appropriate order. However, there were some modifications that were required for the best performance.

One place where modifications to the data access patterns was made was the pentadiagonal solver. The pentadiagonal solve is directly solving a scalar pentadiagonal linear system for a single grid line, or pencil, through the grid (for instance a `J`-line where `K` and `L` are constant as seen in Listing 8). Note that the implementation presented in Listing 8 is more for demonstration and not the final implementation in OVERFLOW, which is implemented in CUDA Fortran and has many other optimizations. Because we have assigned a single thread to solve a pencil, the right- and left-hand sides should be ordered such that coalesced accesses are possible. Ideally, the solved direction would be the last index, i.e., the first entry on the main diagonal for the first system (`K=1` and `L=1`) should be next to the first entry for the second system (`K=2` and `L=1`). Thus, the ideal ordering within the pentadiagonal solve for the three directions is shown in Table 2. However, there is a cost associated with performing the transpose of these matrices from the order they are computed, `(J,K,L)`. As a compromise, the second index can be the index for the solve direction. This can cause uncoalesced access for some warps if they happen to straddle different `L` values, but this is far better than always being uncoalesced. For the `K` and `L` direction solves, the matrix or right-hand side vectors do not need to be transposed at all, saving significant cost. For the `J` solve, the linear system does need to be transposed so that the `J` index is not first which would result in poor, uncoalesced performance. However, instead of performing the expensive 3D transposition, a cheaper 2D transposition of the first two indices can be performed.

E. Merging Kernels

Once all the above optimizations were implemented, the GPU version was functioning but was not significantly faster than the CPU implementation. Because OVERFLOW is a memory bound code, the global memory usage and memory performance was investigated. The OVERFLOW CPU implementation is designed to easily vectorize on the

Table 2 Ideal and actual index ordering for the pentadiagonal system.

Solve Direction	Ideal Indexing	Used Indexing
`J`	`(K,L,J)`	`(K,J,L)`
`K`	`(J,L,K)`	`(J,K,L)`
`L`	`(J,K,L)`	`(J,K,L)`

CPU and as such, each subroutine does a single thing, such as calculate a pressure or flux, as demonstrated in Listing 4. OVERFLOW then stores the result of these functions in temporary arrays (the size of the slice being worked on) to be used by the next routine. This is very efficient on the CPU, which can typically store these temporary arrays of plane data in local cache and the compilers have no problem optimizing and vectorizing these simple loops. However, on the GPU, the increased parallelism requirements mean we are working not on a plane but multiple grids worth of data, and that these temporary arrays are too large to remain in cache. Instead, the temporary arrays are written to and read from global device memory between each of these kernels. This greatly increases the amount of memory traffic necessary to form the RHS, for example. This problem is made even worse by having separate routines for each of the three computational directions. For example, to calculate the second-order, inviscid central difference scheme's residual, over 24 separate kernels are called, each of which read and write at least the number of points in the batch worth of `real(8)` (often many multiples of this). Thus, when some of these kernels are combined, the memory traffic associated with the temporary arrays would be eliminated and there would be some reuse between the routines. For instance, a single load of the solution state, `Q(J,K,L,:)`, could be reused multiple times instead of being loaded multiple times in different kernels.

F. Shared Memory

GPUs offer a fast, flexible, programmer controlled, scratch pad memory called shared memory. This memory is a partition of the L1 cache available to a thread block where the programmer can control what data should be placed there for fast access. The amount of this memory varies based on the specific card and the desired occupancy of the kernel. All threads in a thread block have access to the same shared memory, so it can be used collaboratively between the threads, and threads can read and write to it uncoalesced without a performance penalty.

Shared memory is used in several instances throughout OVERFLOW. The first use of it is to perform matrix transpose operations, like those described in Section II.D. Using shared memory, one can both read the original matrix from global memory and then write the transposed version of the matrix back to global memory with coalesced memory accesses. Listing 9 shows how this is performed in OpenACC. Because shared memory is fast cache-like memory, uncoalesced accesses to that memory space are not detrimental to the performance of the kernel. The OpenACC parallel region is split into two loops with an implicit synchronization point between the threads in the thread block ensuring that the shared memory tile is fully set before any thread reads from it.

Another use of shared memory is in the RHS routines. In those routines, a single thread calculates the flux at a given point. The threads then atomically write the contribution of that flux to the residual vectors of the points in the stencil. For instance, in a fourth-order central stencil, the Euler flux in `J` at a point `(J,K,L)` contributes to the residual at points from `(J-2,K,L)` to `(J+2,K,L)`. Without shared memory, this would result in 5 atomic writes to the global memory for each entry in the residual vector. However, with shared memory, these contributions can be accumulated in fast cache and then written once. This greatly reduces the memory traffic as each point in the residual vector is only written once.[†] Because the use of shared memory can greatly reduce the memory traffic, the overall performance of these kernels was significantly improved by using shared memory.

G. Multiple GPUs

To scale to larger and larger problems, OVERFLOW needs to be run on multiple GPUs (just like the CPU path needs to scale to multiple CPU nodes). Just like the CPU path, this scaling is achieved primarily using MPI. Instead of assigning a single MPI rank to each CPU core (or set of cores if using OpenMP), the GPU path in OVERFLOW is designed to have a single MPI rank per GPU. One possibility is to have multiple MPI ranks use the same GPU using the NVIDIA Multi-Process Service (MPS) [16]. However, with the above code optimizations, a single MPI rank can

[†]The memory traffic is actually slightly more than this, since there will be contributions to some points between thread blocks multiple times.

Listing 9 Example of efficient transposition using OpenACC.

```
real, dimension(jd,kd,ld) :: A
real, dimension(kd,jd,ld) :: B
real, dimension(tile_side,tile_side) :: shmem
!
!$acc parallel loop gang collapse(3) async       &
!$acc    vector_length(tile_side*tile_side)      &
!$acc    present( A, B )                         &
!$acc    private( shmem, ig )
    ! loop over blocks of J and K
do l      = 1, ld
do kstart = 0, kd-1, tile_side
do jstart = 0, jd-1, tile_side
    !$acc cache(shmem(:,:))
    ! loop within each block of J and K
!$acc loop vector collapse(2) private(j,k)
    do kk = 1,tile_side
    do jj = 1,tile_side
        j = jj + jstart
        k = kk + kstart
        if( j<=jd .and. k<=kd ) then
            ! Do not do this, results in an uncoalesced write
            ! B(k,j,l) = A(j,k,l )
            ! Read coalesced
            shmem(jj,kk) = A(j,k,l)
        end if
    end do
    end do
    ! Implicit sync threads
    ! Switch loop order to change write order
!$acc loop vector collapse(2) private(j,k,nn)
    do jj = 1,tile_side
    do kk = 1,tile_side
        j = jj + jstart
        k = kk + kstart
        if( j<=jd .and. k<=kd ) then
            ! Write coalesced
            B(k,j,l) = shmem(jj,kk)
        end if
    end do
    end do
end do
end do
end do
```

Listing 10 Example of calling a GPU-aware MPI implementation with OpenACC.

```
!$acc host_data use_device(buff) if_present
call MPI_Isend( buff, nbuff, datatype, destination, tag,  &
                comm, request, ierr )
!$acc end host_data
```

saturate a GPU, so the default mode of OVERFLOW is to map one process per GPU.

When running on multiple GPUs, each GPU only handles a subset of the entire problem (using the same grid splitting process as the CPU path) and data need to be exchanged between them at the overset boundaries. This is the same chimera boundary exchange that currently happens on the CPU side where the data on donor grids are interpolated to interpolation points, packed into buffers, sent to the receiver ranks, and unpacked by the receiver grids. During the solve process on GPUs, the updated solution only lives in the GPU memory space, so we currently require a GPU-aware MPI implementation. Most major MPI implementations (Open-MPI, MVAPICH2, MPT, MPC, and others) have GPU-aware capabilities that are sometimes optionally enabled by environment flags. Being "GPU-aware" means that the MPI library can understand and handle buffer pointers that point to data on the GPU. Using OpenACC, you can tell the compiler you want the CPU code to use the device version of a pointer using the `host_data` pragma, as shown in Listing 10. The actual mechanism for transferring these data between GPUs will vary based upon the hardware available, the MPI implementation details, and the configuration of the MPI library. Communication can occur directly from GPU to GPU within the system without involving the CPU process if technology such as NVIDIA's GPUDirect is available and configured properly. If direct communication is not possible (or configured properly) the GPU-aware MPI libraries will copy the buffer to the CPU, perform the communication between CPUs as normal, and then transfer the data from destination CPU to GPU. However, by using the GPU-aware approach, the programmer does not have to manually create these buffers and explicitly copy the data to the host before calling MPI. All MPI communications in the GPU path of OVERFLOW are non-blocking, so the GPU can continue to do work while transferring data, hiding some of the cost of communication.

III. Current Status of OVERFLOW on the GPUs

OVERFLOW has been ported to run on the GPUs using the approaches presented above. This section presents the overall design of the GPU path in OVERFLOW, the options in OVERFLOW that have been ported to run on GPUs so far, and the performance of the GPU path in OVERFLOW compared to the CPU path.

A. Implementation Progress

The GPU path in OVERFLOW is designed as follows:

1) Read namelist and input files
2) Set up normal CPU data structures
3) If running on the GPU:
    1) Check to ensure that the requested options have been ported to the GPU.
    2) Create the GPU data structures and copy the relevant CPU data onto the GPU.
    3) Solve the problem on the GPU, only copying the data from the GPU to CPU when necessary for output.
    4) Copy all the data back to the CPU.
4) Perform all necessary post-processing and output on the CPU.

Unfortunately, our current approach means that there are two paths through the code, one for the normal CPU path and a separate one for the GPU path. Some routines, such as some boundary conditions, are shared between the two paths (a benefit of the directive approach), but most of the GPU path is rewritten as separate routines. This rewrite was necessitated by the significant performance optimizations discussed above to make the GPU version competitive with the optimized CPU path. While many of these changes, such as reducing the memory usage, should also be beneficial for the CPU, our experience is that these changes, when run on a CPU, actually reduce the performance. This is usually because the increased complexity of the loops breaks vectorization and other compiler optimizations. Removing the temporary arrays does not have a significant effect on the CPU version of the code since, with larger cache size per thread, most of the temporary arrays fit in some level of cache.

Note that because there are two paths through the code, not all options have been implemented on the GPU. The central scheme (`IRHS=0`) with the scalar-pentadiagonal implicit scheme (`ILHS=2`) are the only options that have been implemented so far. The GPU version also supports rotating frames of reference, the SA turbulence model, and most of the commonly used boundary conditions. For a full list of ported options see the Appendix. For the options that have been ported there are many checks to ensure that the same answer is obtained in both the CPU and GPU path.[‡]

B. Performance Comparisons

Several performance studies have been performed throughout this process as we optimized and tuned the GPU version of the code. The key takeaways of this work are presented here with a brief discussion.

1. Problem Size

Since the GPU operates differently than CPU, it was important to investigate our recommendation for how large of a problem one should run on a single GPU. We know that the GPU operates best when it has enough work (namely parallelism given by the number of grid points) to take advantage of as much of the GPU's performance as possible. Thus, the performance should increase as the grid size increases until we have enough work to saturate the GPU. OVERFLOW is memory bound so most of the kernels hit memory bandwidth limitations when they have enough work.[§] Once the peak throughput has been achieved it should stay relatively constant until you run out of available memory on the GPU. In OVERFLOW, we see this behavior very clearly in Figure 2 where cubes of various sizes were run on a single GPU (or a single Skylake node). We see that the CPU achieves peak performance at around 10 million points (lining up with our recommendation to run at least 250 thousand points/CPU core). However, as expected, the GPU does not reach peak performance until approximately 3 million points for the V100S and 8 million points for the two A100 cards. We remain at this peak throughput until the GPU runs out of memory. With the ported options, OVERFLOW can fit about 2 million points for each GB of GPU memory so the 32 GB of the V100S can fit a problem with 66 million points. While this study was performed using a single, large grid, the throughput of the GPU does not significantly change for grid systems with many small grids when using the batching strategy discussed in Section II.C. However, some kernels, while operating on a 3D grid, only have parallelism exposed on a 2D plane, so the performance of these kernels can suffer when long, skinny grids result in too little parallelism along one dimension.

2. Weak Scaling

In weak scaling, as the code is run on more GPUs, the size of the problem increases proportionally so that the problem size per each GPU is constant. Thus, ideally the time to perform an iteration should stay constant. However, since there are extra overheads associated with running on multiple devices, such as communication and grid splitting, the actual runtime will increase. To measure the impact of this we ran a problem with a single, 27-million-point grid for

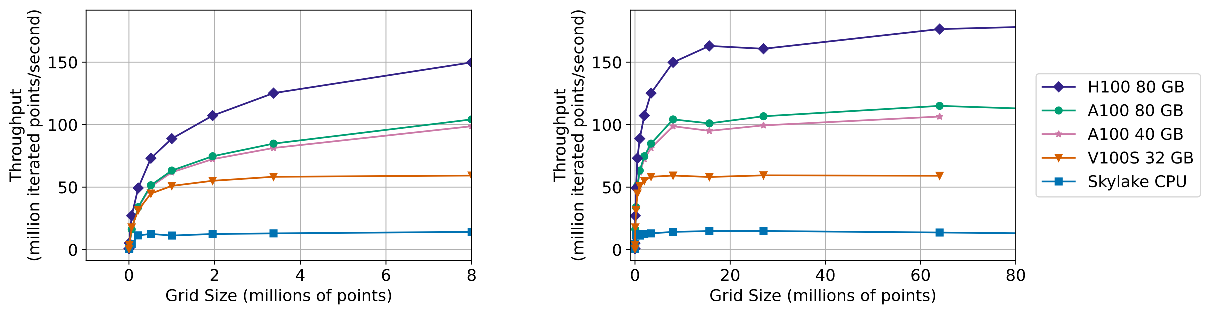


Fig. 2 Throughput for various problem sizes on various hardware. These results are presented for a single GPU or a single CPU node of 40 Skylake Cores. The figure on the left is zoomed in for the small grid sizes.

[‡]There might be a slight difference due to performing some operations in parallel and in a different order than the CPU path, but these differences are typically on the order of round-off error.

[§]Unfortunately, many of the kernels do not achieve the peak theoretical memory bandwidth but roughly 60 – 80% of the peak bandwidth.

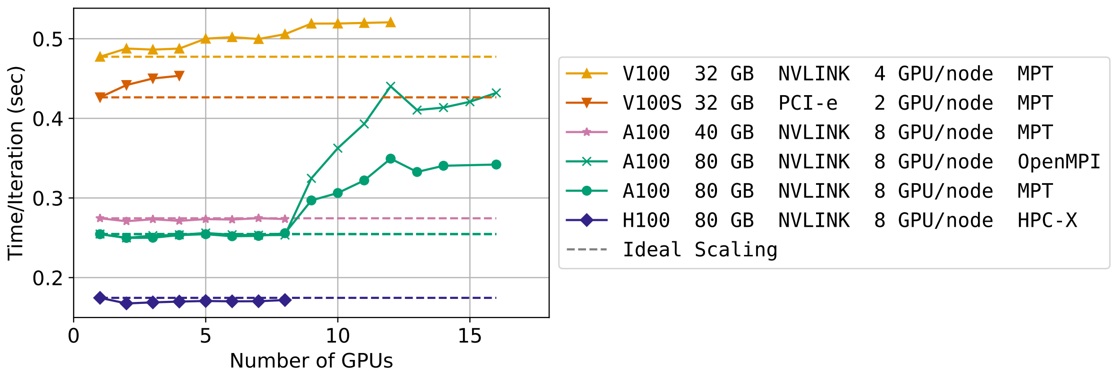


Fig. 3 Time to perform a single iteration with different number of GPUs for problems with 27 million points per GPU.

each GPU.[¶] Figure 3 shows the time it took to perform a single iteration with the different number of GPUs.

The weak scaling performance shown in Figure 3 is fairly good but does highlight some of the quirks of the different systems. The two systems with NVLINK connections for the GPUs (A100 and V100) scale very well on the single node (up to 4 V100 cards or up to 8 A100 cards). This is because with the fast interconnect there is minimal impact for the added communication cost associated with running more ranks. However, the V100S node has the 2 cards connected over PCIe, so the slower connection causes the communication overhead to be higher, impacting the weak scaling performance even on the same node. The impact of higher overhead costs is also seen when running across multiple nodes regardless of the GPU. In this case, the communication is happening over the relatively slower InfiniBand network, meaning the communication overhead is that much higher, hurting the scaling performance. This is dependent on the specific MPI implementation and how it was configured. For instance, on our system, the OpenMPI implementation has similar performance on a single node to MPT, but once you move across multiple nodes the performance is significantly slower using OpenMPI compared to HPE's MPT MPI implementation. All of this, of course, will depend on the specifics of the hardware configuration and the MPI implementation.

3. Drag Prediction Workshop Case

Now that these basic performance characteristics have been examined, the performance can be examined on actual problems of interest. Two cases have been used so far to evaluate the performance gained using GPUs over CPUs. The first case is the Drag Prediction Workshop 6 wing-body configuration (DPW6-WB) [17], using the committee-supplied structured overset grids at the medium grid level. This grid is from the $\alpha = 2.5°$ aeroelastically-deformed case and is shown in Figure 4. The grid system consists of 8 grid blocks with a total of nearly 25 million grid points. The results from the GPU path match the CPU path to within round-off error when the same number of ranks are used (same block-splitting). To compare the performance of this case, the CPU version was run on K4 and strong scaled (same problem size) up to 16 nodes (640 cores). The case was then run on a single V100S on K4, a single 80 GB A100 on K5, and a single 80 GB H100 at NVIDIA. The time per iteration for these cases are presented in Figure 5. For easy comparison, Table 3 shows how much faster a single GPU is compared to 40 Skylake cores, and how many Skylake cores one would need to match the performance of that GPU. These results show that for this problem, the GPU path is significantly faster than the CPU path through the code.

4. Hover Validation and Acoustic Baseline Case

The second case investigated is the Hover Validation and Acoustic Baseline (HVAB) rotor case [18]. To show the impact the grid system can have on the performance of the code (both on the CPU and GPU), two grid systems were created for this case. The first grid system for this case has 14 grid blocks with approximately 56 million points as shown in Figure 6a. The center cylindrical grid does have a singular axis at the origin and is thus referred to as the HVAB-axis case. As will become apparent, this grid topology can reduce the scaling performance of OVERFLOW. The second grid

[¶]The normal grid splitting and distribution algorithm for OVERFLOW was used, so the total number of points in the final grid system is slightly larger than the number of GPUs × 27 million points.

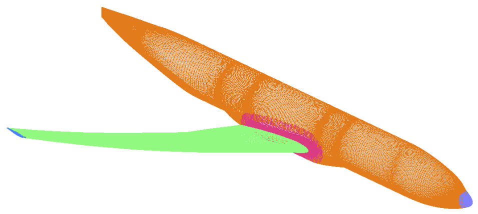


Fig. 4 Surface grids for the DPW6-WB case with 25 million total points.

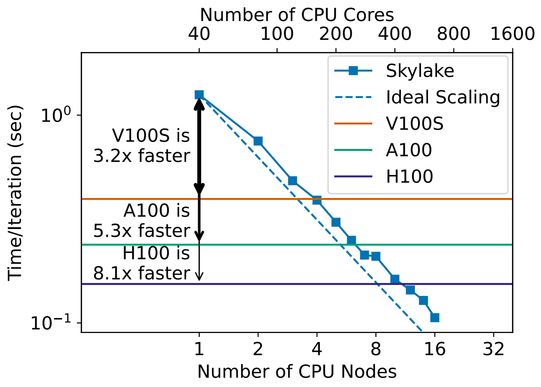


Fig. 5 Comparison between K4 CPUs and a single V100S, A100, and H100 for the DPW6-WB case.

Table 3 Performance Comparison for single GPU for DPW6 case.

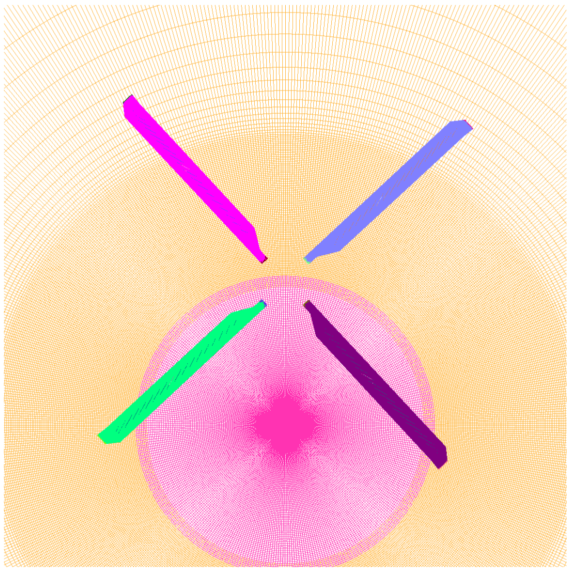
GPU	Speedup Relative to Skylake Node	Equivalent number of Skylake Cores
V100S	3.2	158
A100	5.3	253
H100	8.1	437

system, shown in Figure 6b, removes this axis by placing a core grid over the axis. This “core” grid cuts a hole over the axis boundary to avoid this type of grid topology. This second grid system, which we will call the HVAB-noaxis case, has 15 grid blocks with approximately 53 million points. Because the grid connectivity routines for moving grids have not yet been ported to the GPU, this rotor case is simulated with stationary grids and a rotating frame of reference.

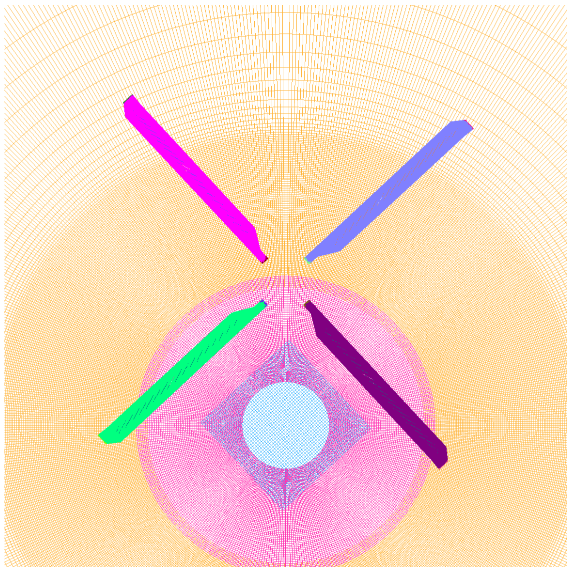
Comparing the strong scaling of the K4 CPU nodes and the GPU systems in Figure 7, the GPUs are significantly faster. Tables 4 and 5 show summaries of the performance of a single GPU relative to the Skylake CPU cores. These tables show that for both grid systems the GPUs are significantly faster than a CPU node, equivalent to many CPU cores. According to our typical guidance this problem would be run on approximately 240 cores but the problem strong scales to this point on both cases, and well past it for the noaxis case.

Taking a closer look at the strong scaling performance, a difference between the HVAB-axis and HVAB-noaxis cases can be seen. The HVAB-axis case does not scale as well as the HVAB-noaxis case on either the CPU or GPU due to the grid topology of this grid system. The axis boundary condition harms strong scaling because the grid-splitting strategy used in OVERFLOW does not split grids azimuthally around the axis boundary condition. This restriction limits how the grids can be split causing non-ideal grid splitting, reducing the parallel efficiency and strong scaling efficiency on the CPU. There is a larger downside to this approach on the GPU since this creates long-skinny grids which do not expose as much parallelism for the penta-diagonal solver. Also, the current GPU implementation of the axis boundary condition only exposes parallelism along the 1D axis, limiting the strong scaling performance of the GPU path. This will be investigated further in the future in an attempt to improve the strong scaling performance of grids with an axis boundary condition on the GPU.

On the CPU, the HVAB-axis case only strongly scales to about 360 Skylake cores, while the noaxis case scales all the way past 16 nodes (640 cores). Similarly, on the GPU the HVAB-axis case only scales to 4 A100 GPUs while the HVAB-noaxis case scales all the way to 8 GPUs with 70% efficiency. At 8 GPUs there are fewer than 7 million points per A100, so we are entering the region where there is not enough work to fill the GPU as discussed in Section III.B.1. However, it is interesting to note that because of the limited strong scalability of both the CPU and GPU, all tested GPUs can achieve a faster time per iteration than possible on CPUs for this problem. This is due to the faster performance of the individual cards and thus the fewer number of ranks (and split grids) reducing the communication overhead required in strong scaling. To match the peak performance possible with the CPU path, one would need approximately four V100S cards, two A100 cards, or only a single H100 card. The strong scaling on A100 GPUs seems to reach a performance limit around 16 cards (2 nodes), where it is 3.6 times faster than the peak performance on CPUs.

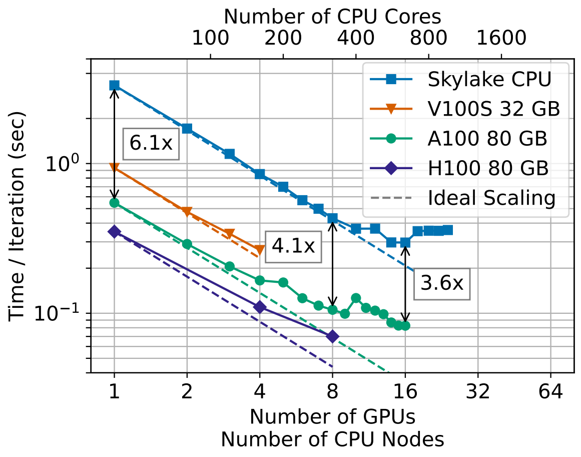


(a) HVAB-axis case with 56 million points. Notice the axis boundary at the center of the grids.

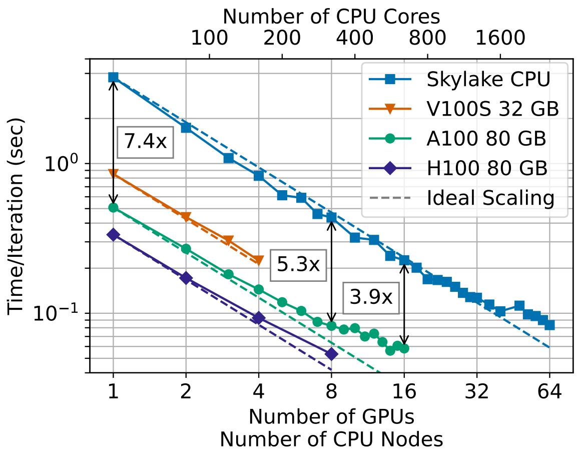


(b) HVAB-noaxis case with 53 million points. Notice the "core" grid replacing the axis boundary.

Fig. 6 Surface grids for the two HVAB grid systems.



(a) HVAB-axis case.



(b) HVAB-noaxis case.

Fig. 7 Strong scaling comparison of HVAB cases between CPU nodes and various GPU nodes.

Table 4 Performance Comparison for single GPU for HVAB-axis case.

GPU	Speedup Relative to Skylake Node	Equivalent number of Skylake Cores
V100S	3.6	149
A100	6.1	253
H100	9.5	717

Table 5 Performance Comparison for single GPU for HVAB-noaxis case.

GPU	Speedup Relative to Skylake Node	Equivalent number of Skylake Cores
V100S	4.4	156
A100	7.4	265
H100	11.3	390

For the better scaling HVAB-noaxis case, there is not as drastic of a drop-off in performance as we scale to more and more CPUs. The GPU also continues to scale well for the noaxis case up to 8 GPUs, which are 5.3 times faster than 8 Skylake nodes. Scaling past that, the strong scaling efficiency begins to drop off as there is not enough work to keep the GPU efficiently utilized, as discussed in Section III.B.1. These strong scaling results highlight the performance benefits of GPUs if there is enough work to keep them well utilized and the impact that grid topology has on that.

IV. Concluding Remarks

This effort is still a work in progress as more options and capabilities of OVERFLOW are being actively ported to the GPU in preparation for the full release of the code. The current status of the GPU implementation was presented above showing significant performance improvements over the CPU version of the code. The approach of using OpenACC, CUDA Fortran, and CUDA C++ has worked well to achieve the goals of running on the latest GPU hardware. This approach will continue to be evaluated as new hardware becomes available in the future so that OVERFLOW will be poised to take advantage of it. Using these programming paradigms, we have been able to implement a path through the code using the central scheme with the scalar-pentadiagonal solver to solve real problems of interest. We presented many of the lessons learned and code optimizations necessary to enable good performance on NVIDIA GPUs compared to the CPU version of the code. These speed-ups allow engineers and scientists to run larger problems faster, enabling a single workstation with a single HPC GPU to take the place of 150 to 500 Skylake CPU cores in a computing cluster. Likewise, it is possible to run certain problem on GPUs faster than would be possible even with unlimited number of Intel Skylake CPUs.

Appendix

Here is an extensive list of ported OVERFLOW options:

IRHS	=	0
ILHS	=	2
NQT	=	0, 102
FSO	=	2,3,4,5 (Only integer values supported)
FSONWT	=	1
IDISS	=	3,4
IBTYPE	=	1, 5, 10, 14–18, 21, 47, 51–53
IGAM	=	0
ISPEC	=	2
NDIRK	=	0
ICC	=	0

Using OpenACC and CUDA Fortran, we have demonstrated significant performance gains solving a variety of problems. The following are not supported on the GPU (but support is planned):

- 2D and axisymmetric grids
- Low-Mach preconditioning
- CFL Ramping
- QCR
- Wall functions
- Limiting for μ_t
- DES
- Rotation Corrections
- Compressibility Correction
- Turbulence regions
- Grid sequencing
- Geometric multi-grid
- Species convection
- CDISC
- C_L driver
- Gravity terms
- Body motion
- Adaptation
- Rotor coupling

The following are not supported on the GPU (with no current plans to support):

- Symmetric grids with reflection planes
- Different `ITERT` values on different grids
- Non-integer values of FSO (for central scheme)

Acknowledgments

The authors would like to thank the NASA Revolutionary Vertical Lift Technology (RVLT) Project of the Advanced Air Vehicles Program (AAVP) for supporting this work. Assistance from the technical staff at NVIDIA, including Jeff Larkin, was greatly appreciated and instrumental in getting us performing well on GPUs. We are also appreciative to the staff at Oak Ridge National Labs and NASA for organizing the several hackathons that we attended as part of this work. Finally, we would like to thank the NASA Langley K-Cluster team for providing the compute-time and resources to perform this work.

References

[1] "NASA OVERFLOW CFD Code," `https://overflow.larc.nasa.gov/`, Nov. 2022. Last Accessed November 7, 2023.

[2] Pulliam, T., "High Order Accurate Finite-Difference Methods: as seen in OVERFLOW," AIAA Paper 2011-3851, 20th AIAA Computational Fluid Dynamics Conference, Honolulu, Hawaii, June 2011. https://doi.org/10.2514/6.2011-3851.

[3] Strohmaier, E., Dongarra, J., Simon, H., Meuer, M., and Meuer, H., "The Top 500 List," http://www.top500.org, 2023. Last Accessed November 7, 2023.

[4] Pulliam, T. H., and Chaussee, D. S., "A diagonal form of an implicit approximate-factorization algorithm," *Journal of Computational Physics*, Vol. 39, No. 2, 1981, pp. 347–363. https://doi.org/10.1016/0021-9991(81)90156-x.

[5] "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Apr. 2023. Last Accessed November 7, 2023.

[6] "NVIDIA CUDA Fortran Programming Guide," https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/index.html, Mar. 2023. Last Accessed November 7, 2023.

[7] "Introduction to HIP Programming Guide," https://docs.amd.com/bundle/HIP-Programming-Guide-v5.4/page/Introduction_to_HIP_Programming_Guide.html, Dec. 2022. Last Accessed November 7, 2023.

[8] OpenACC-Standard.org, "The OpenACC Application Programming Interface Version 3.3," 2022. URL https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf.

[9] Board, O. A. R., "OpenMP Application Programming Interface, Version 5.2," 2021. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[10] The Khronos SYCL Working Group, "SYCL 2020 Specification (revision 7)," 2023. URL https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[11] Khronos OpenCL Working Group, "The OpenCL Specification," 2023. URL https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[12] Beckingsale, D. A., Scogland, T. R., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., and Ryujin, B. S., "RAJA: Portable Performance for Large-Scale Scientific Applications," *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, 2019. https://doi.org/10.1109/p3hpc49587.2019.00012.

[13] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003.

[14] Herten, A., "GPU Vendor/Programming Model Compatibility Table," JSC Accelerating Devices Lab Blog (online), Nov. 2022. https://doi.org/10.34732/xdvblg-r1bvif, last Accessed April 24, 2023.

[15] Harris, M., "Unified Memory for CUDA Beginners," https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, Jun. 2017. Last Accessed November 7, 2023.

[16] "Multi-Process Service," https://docs.nvidia.com/deploy/mps/index.html, Mar. 2023. Last Accessed November 7, 2023.

[17] "6th AIAA CFD Drag Prediction Workshop," https://aiaa-dpw.larc.nasa.gov/Workshop6/workshop6.html, Mar. 2021. Last Accessed November 7, 2023.

[18] Overmeyer, A. D., Copp, P. A., and Schaeffler, N. W., "Hover Validation and Acoustic Baseline Blade Set Definition," NASA/TM 2020-5002153, NASA, 2020.