Early results using Fortran's `do concurrent` standard parallelism on Intel GPUs with the `ifx` compiler

Ronald M. Caplan, Miko M. Stulajter,
Jon A. Linker, and Cooper Downs
Predictive Science Inc.
caplanr@predsci.com

Predictive Science Inc.

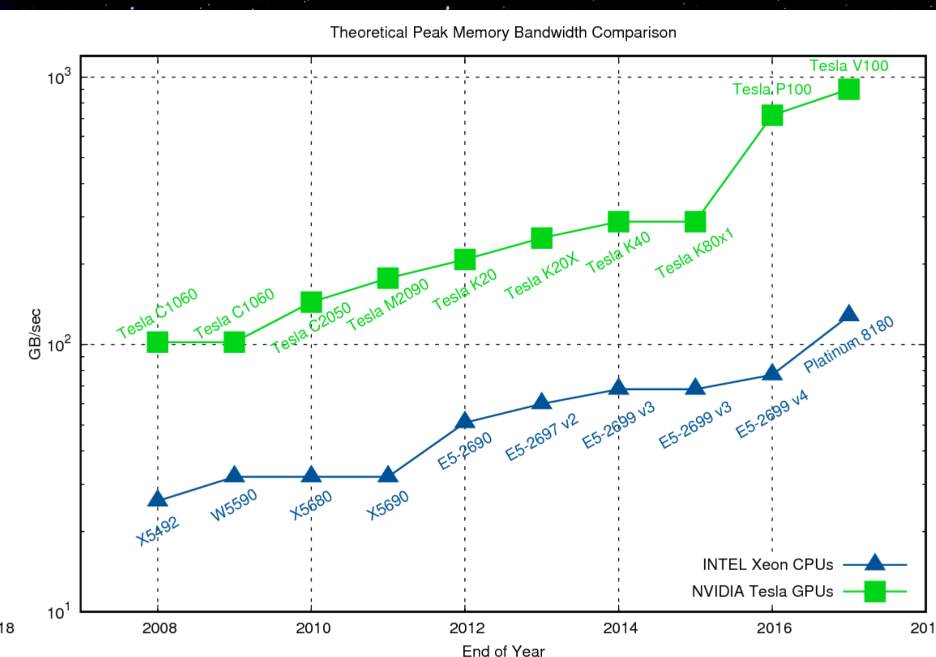
Supported by NSF and NASA

Agenda

- Accelerated computing
- Directives and Fortran standard parallelism
- Previous implementation results on NVIDIA GPUs with `nvfortran`
- Preliminary implementation results on Intel GPUs with `ifx`
- Call to action and future outlook

Accelerated Computing

- *Overall* performance
  - FLOP/s
  - *Memory Bandwidth*
  - Specialized hardware (e.g. ML/DL tensor cores)
- *Compact* performance
  - *In-house workstations*
  - Reduce HPC real estate
- *Efficient* performance
  - Lower energy use
  - Save money



Directives

- Comments that the compiler can use to generate code that the base language does not support (e.g. parallelism, GPU-offload, data movement, etc.)
- Can produce single source code for multiple targets (GPU, CPU, FPGA, etc.)
- Low-risk - can ignore directives and compile as before
- Vendor-independent (subject to implementation)
- Great for rapid development and accelerating legacy codes
- Two major directive APIs for accelerated computing: **OpenACC** and **OpenMP®**

OpenACC

```
!$acc enter data copyin(x) create(y)
!$acc parallel loop
    do i=1,n
      y(i) = a*x(i) + b
    enddo
!$acc exit data delete(x) copyout(y)
```

OpenMP Target

```
!$omp target enter data map(to:x) map(alloc:y)
!$omp target teams distribute parallel do
    do i=1,n
      y(i) = a*x(i) + b
    enddo
!$omp end target teams distribute parallel do
!$omp target exit data map(delete:x) map(from:y)
```


The OpenMP name is a registered trademark of the OpenMP Architecture Review Board

Fortran Standard Parallelism: Do Concurrent (DC)

- ISO Fortran 2008
- Indicates loop can be run **out-of-order**
- Can hint to compiler that loop **may** be parallelizable
- No support for atomics, device selection, async, conditionals, etc.
- Fortran 2023 has added reductions

```
do i=1,N
  do j=1,M
      Computation
  enddo
enddo
```

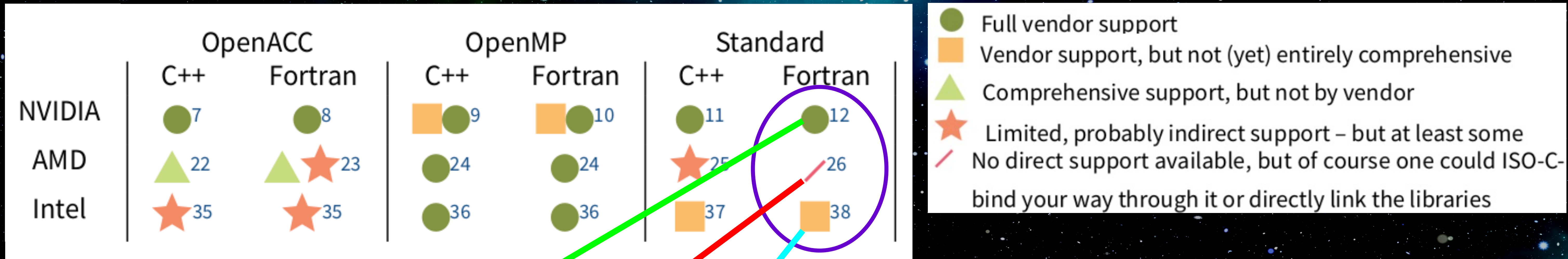
```
do concurrent (i=1:N,j=1:M)
  Computation
enddo
```

```
do concurrent (i=1:N) reduce(+:sum)
  sum = sum + a(i)
enddo
```

Compiler	Version	DO CONCURRENT parallelization support
nvfortran	≥ 20.11	CPU with `-stdpar=cpu` GPU with `-stdpar=gpu`
ifx	≥ 19.1 ≥ 23.0	CPU with `-fopenmp` GPU with `-fopenmp-target-do-concurrent`
gfortran	≥ 9	CPU with `-ftree-parallelize-loops=<#Threads>`

Portability

github.com/AndiH/gpu-lang-compat



- 12: Standard language parallelism of Fortran, mainly `do concurrent`, is supported on NVIDIA GPUs
- 26: Currently, no (known) way to launch Standard-based parallel algorithms on AMD GPUs
- 38: With Intel oneAPI 2022.3, Intel supports DO CONCURRENT with GPU offloading

Directives vs. Standard Parallelism

Why use DC instead of directives?

- Longevity (ISO)
- Lower code footprint
- Less unfamiliar to domain scientists
- For accelerated computing, directives (e.g. OpenMP) are currently more portable

These also apply to codes that already use directives

Original Non-Parallelized Code

```
do k=1,np
  do j=1,nt
    do i=1,nrm1
      br(i,j,k) = (phi(i+1,j,k)-phi(i,j,k))*dr_i(i)
    enddo
  enddo
enddo
```

OpenACC Parallelized Code

```
!$acc enter data copyin(phi,dr_i)
!$acc enter data create(br)
!$acc parallel loop default(present) collapse(3) async(1)
do k=1,np
  do j=1,nt
    do i=1,nrm1
      br(i,j,k) = (phi(i+1,j,k)-phi(i,j,k))*dr_i(i)
    enddo
  enddo
enddo
!$acc wait
!$acc exit data delete(phi,dr_i,br)
```

Fortran's DO CONCURRENT

```
do concurrent (k=1:np,j=1:nt,i=1:nrm1)
  br(i,j,k) = (phi(i+1,j,k)-phi(i,j,k))*dr_i(i)
enddo
```

History of our GPU implementations

OpenACC

Fortran DC

2012-3: Wanted to use GPUs, but not with CUDA due to needing code rewrites, and multiple code bases in multiple languages (Fortran & C)

2014-5: NVIDIA's OpenACC implementation mature enough to start using it for small tools (DIFFUSE)

2016-7: Implemented OpenACC into a larger code that uses MPI for running on multiple GPUs (POT3D)

2018-9: Implemented OpenACC into our production-level MHD code (MAS)

2020: Optimized OpenACC implementations, started using in production runs

2021: Implemented Fortran standard parallelism with `do concurrent' (DC) into DIFFUSE

2022: Implemented DC into POT3D, but retained small amount of OpenACC for performance

2023: Implemented DC into MAS, but retained small amount of OpenACC for performance

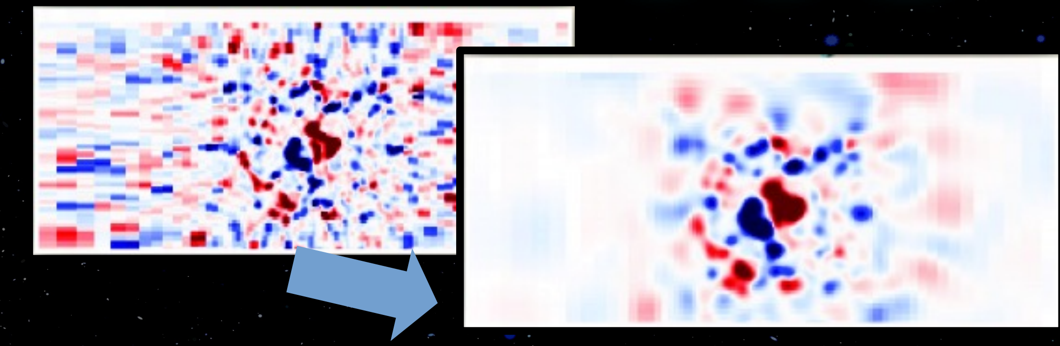
OpenMP Target features and support starts to be competitive to OpenACC, but we already had large amounts of OpenACC implemented, and felt OpenACC was easier/cleaner to code

Decided to pursue standard Fortran (stdpar) due to NVIDIA's support and Intel's announcement of support. Stdpar allows for cleaner code and more portability than OpenACC

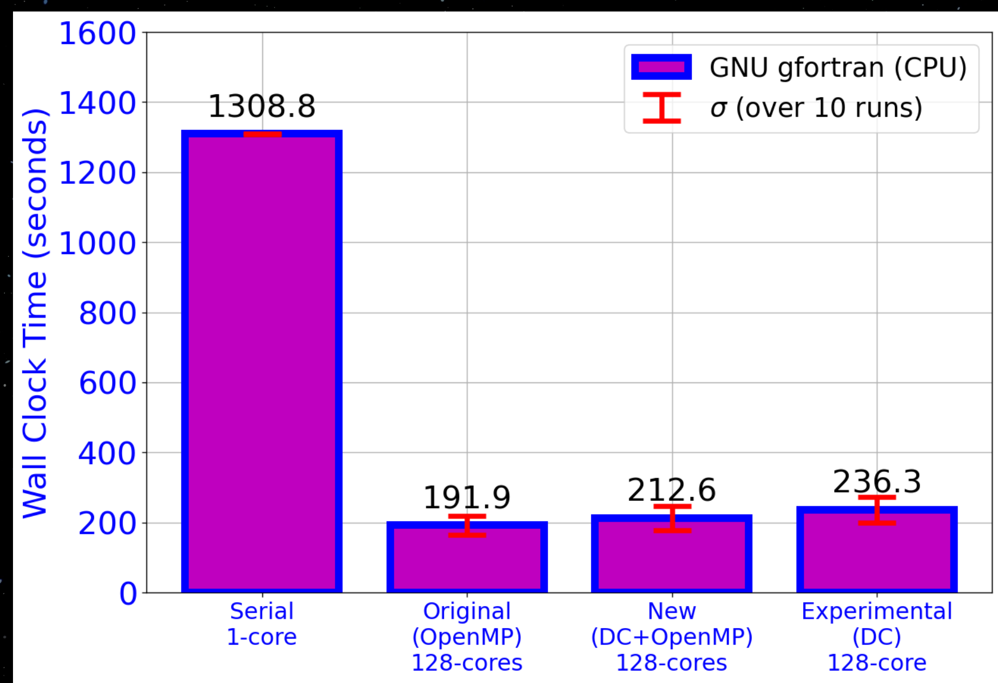
Previous Implementations Results on NVIDIA GPUs: DIFFUSE

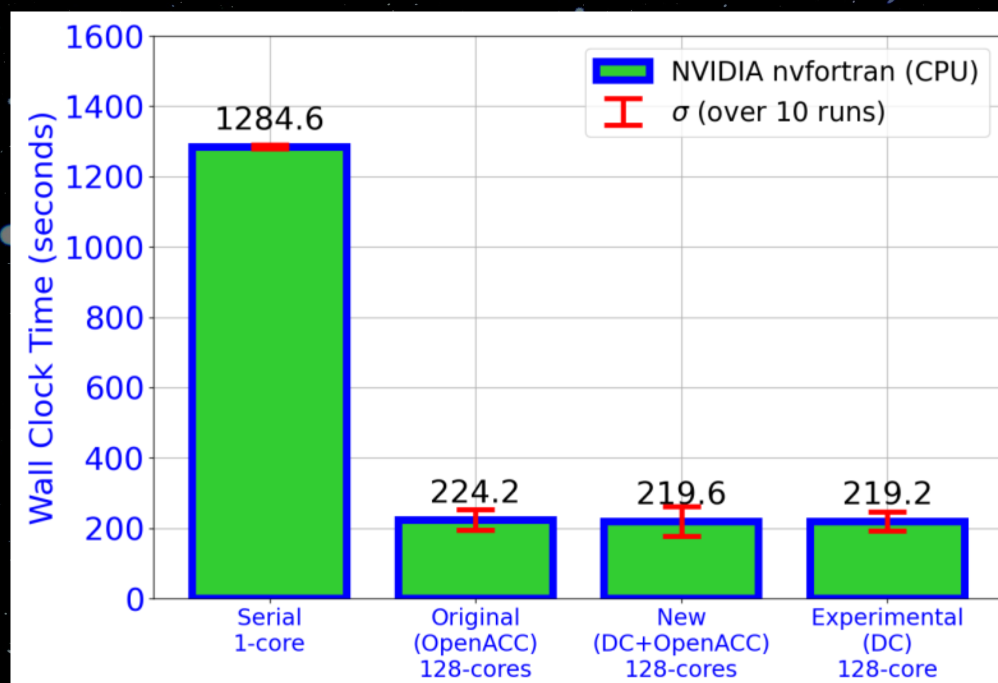
- Small solar surface magnetic field smoothing tool
- Integrates 2D spherical surface Laplacian operator with finite differenceing and super time stepping
- Parallelized for CPUs with OpenMP and for GPUs with OpenACC
- We replaced all directives with DC, and the code retained its performance on multicore CPUs

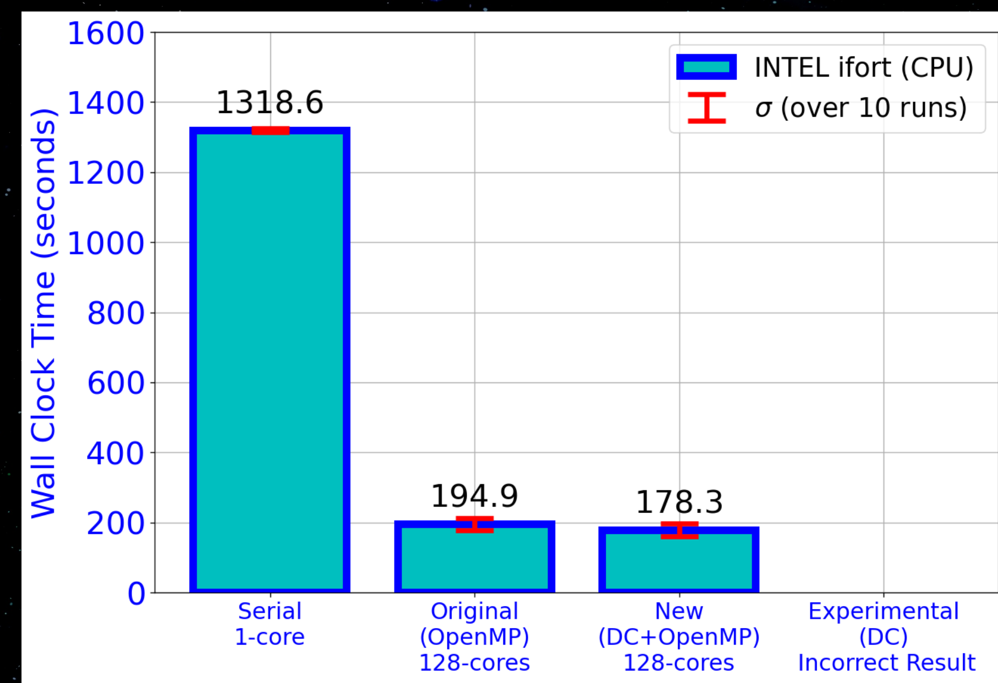
DIFFUSE



Stulajter, et. al. “Can Fortran's `do concurrent' Replace Directives for Accelerated Computing?” Lecture Notes in Computer Science, 13194, 3-21. Springer, Cham. (2021)



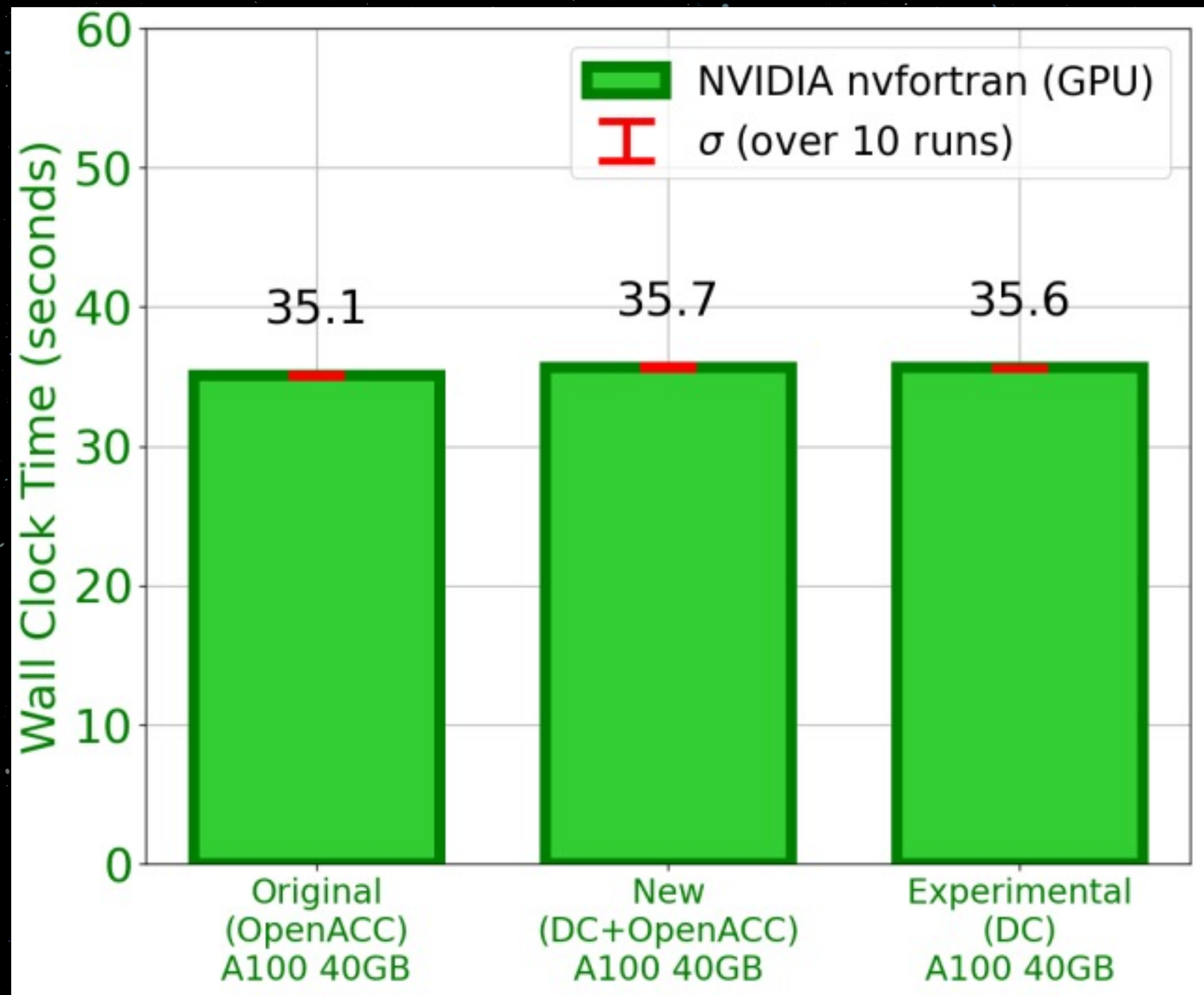


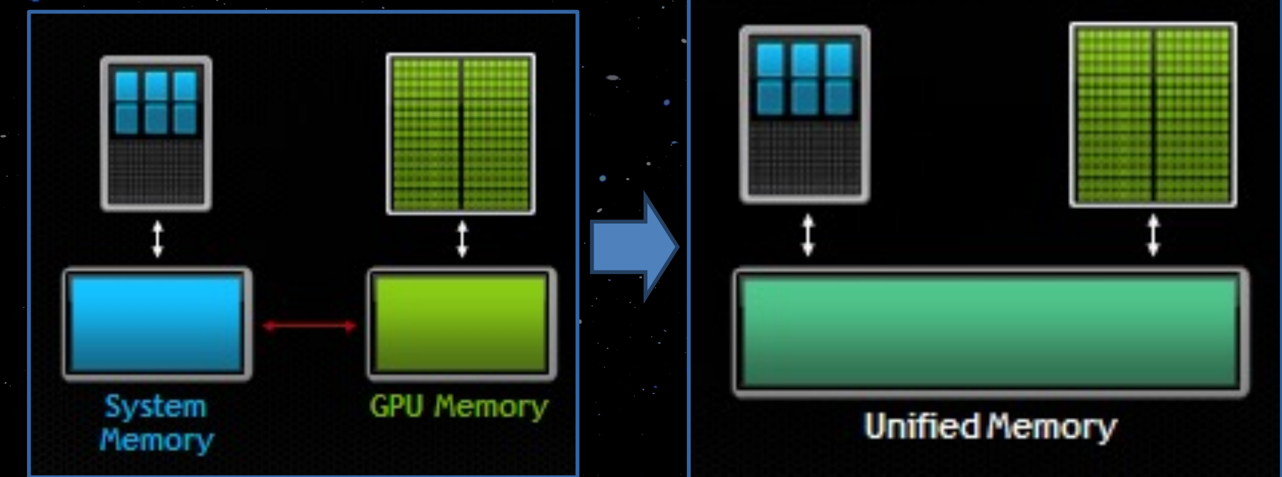


Previous Implementations Results on NVIDIA GPUs: DIFFUSE

DIFFUSE

- We saw similar performance on NVIDIA GPUs using only DC
- This used the default setting of the NVIDIA compiler when using DC, which activates **Unified Managed Memory (UMM)**
- Alternatively, we can turn off **UMM** and manually manage memory with unstructured data movement directives






```
!$acc enter data copyin(a)
!$acc enter data create(b)
...
!$acc update host(a)
...
!$acc host_data use_device(a)
...
!$acc exit data delete(a)
```

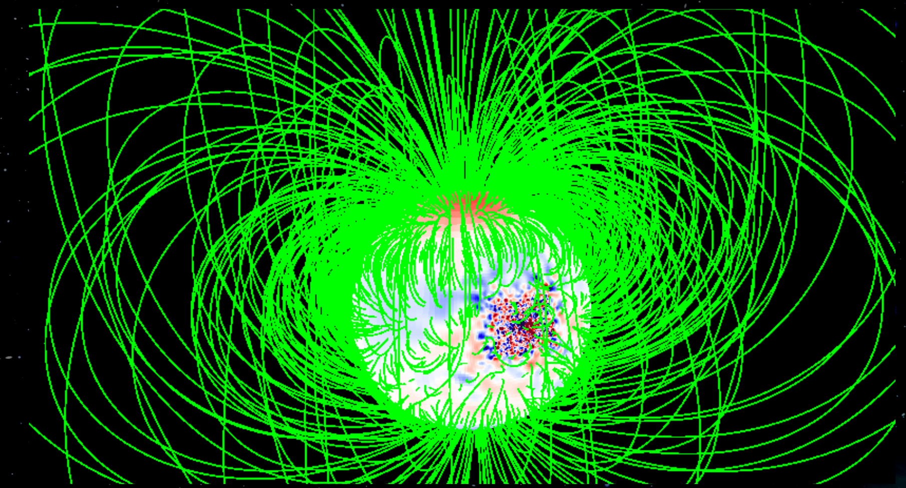
- POT3D computes approximations of the magnetic field of the Sun's lower atmosphere
- It is parallelized for CPUs with MPI and multiple GPUs with MPI+OpenACC
- Part of the SPEChpc(TM) 2021 benchmark suite
- We converted all "do" loops into DC, and the CPU performance did not change. DC also added the ability to run in hybrid MPI+multicore mode (not tested yet)

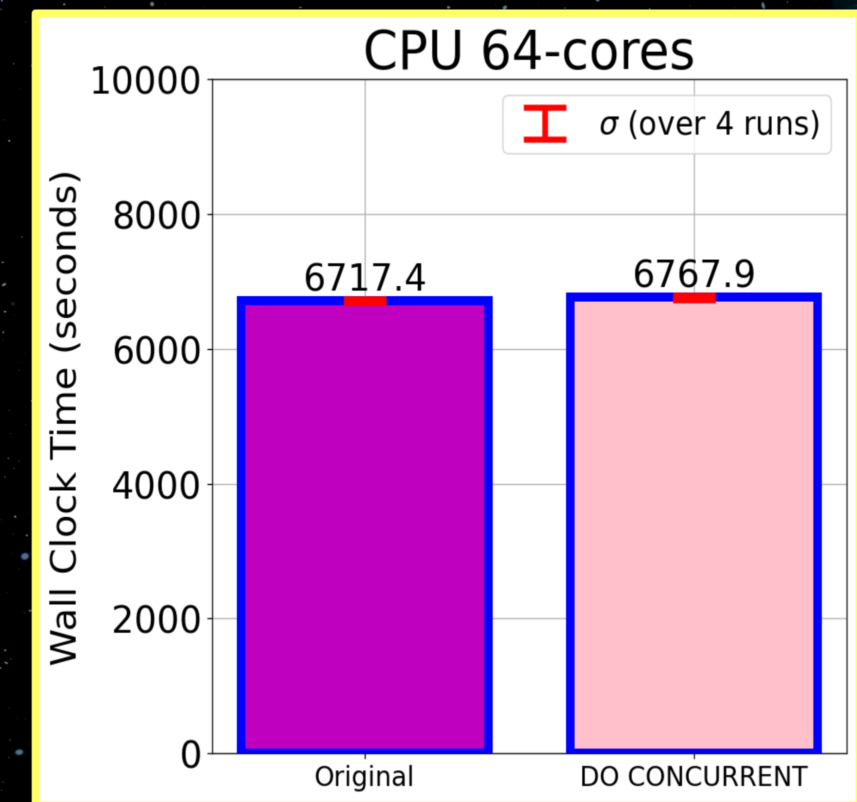
github.com/predsci/POT3D

"Variations in Finite Difference Potential Fields" Caplan, et. al.,Ap.J. 915,1 (2021) 44

https://developer.nvidia.com/blog/using-fortran-standard-parallel-programming-for-gpu-acceleration

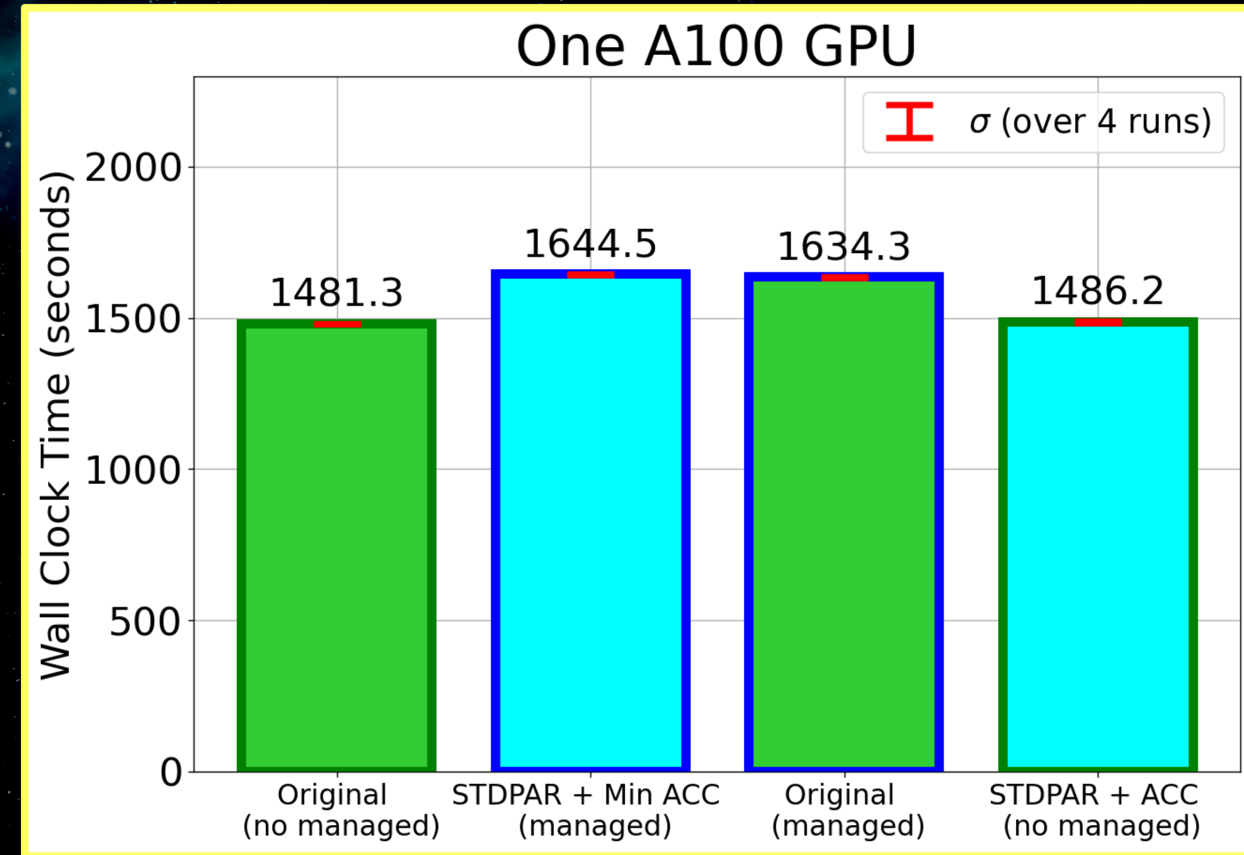
POT3D

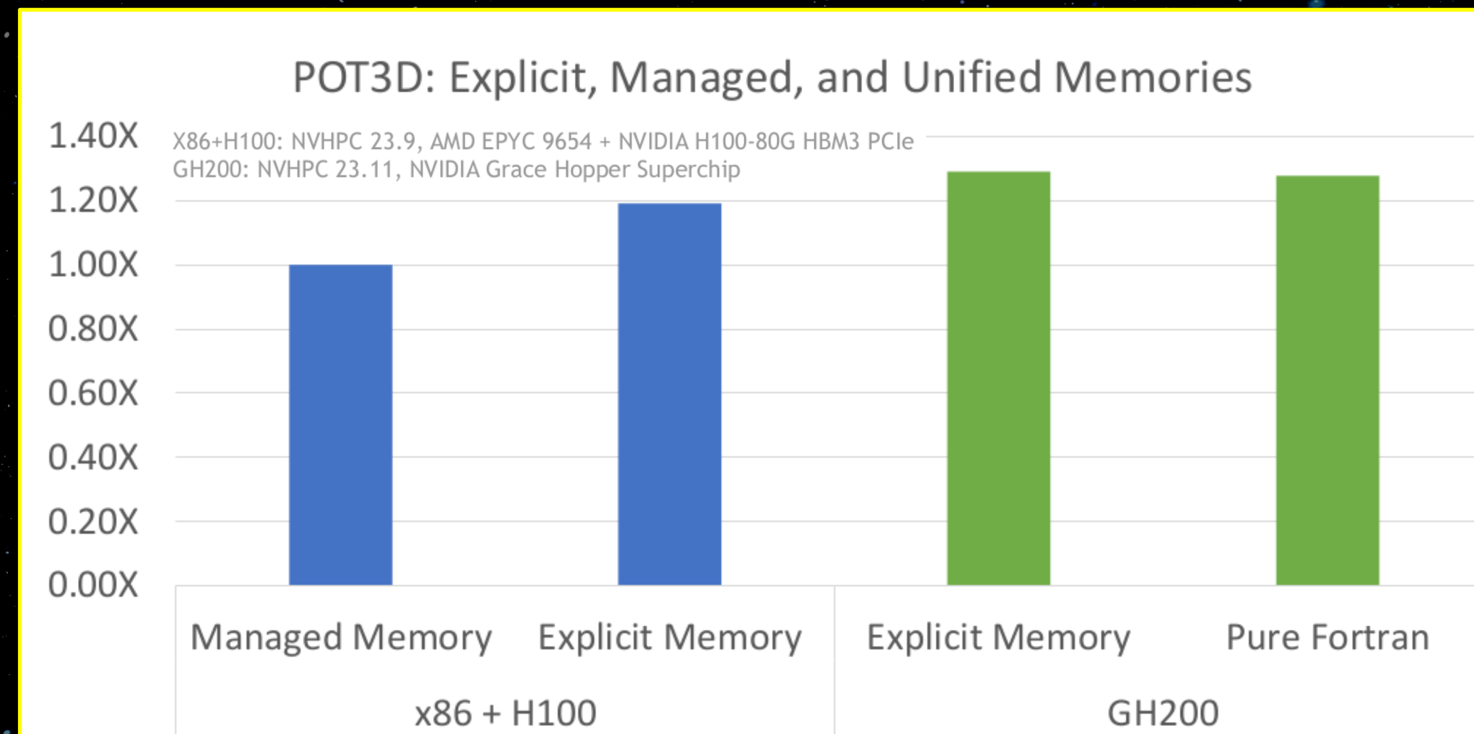




Previous Implementations Results on NVIDIA GPUs: POT3D

- We replaced all directives with DC, letting UMM handle data management
- We saw a ~10% slowdown due to issues with UMM+MPI (not present on Grace-Hopper!)
- Original performance regained by adding back OpenACC data directives
- Hybrid DC+OpenACC still advantageous due to large reduction in number of directives and lines of code, making the code more domain-scientist friendly



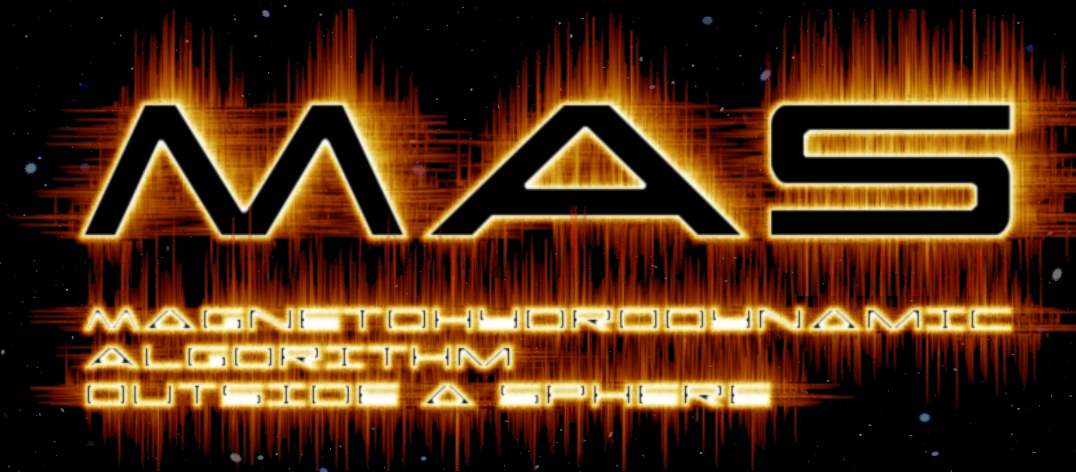


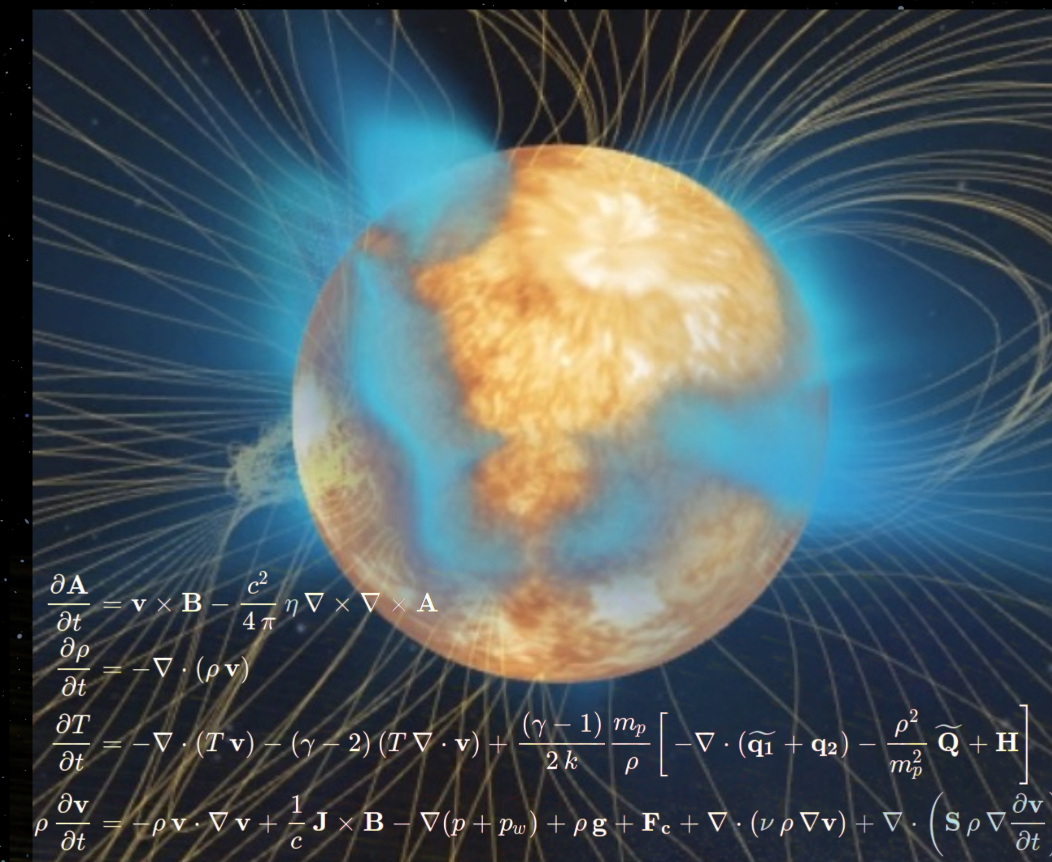
Previous Implementations Results on NVIDIA GPUs: MAS

- Large (~70,000 lines) in-production code for general-purpose simulations of the Sun's atmosphere used in solar physics and space weather research
- Solves spherical 3D thermodynamic MHD equations using implicit & explicit time-stepping with finite-differences and sparse matrix preconditioned iterative solvers
- Parallelized for multiple CPUs with MPI and multiple GPUs with MPI+OpenACC
- We converted "do" loops into DC and the CPU performance did not change. DC again added the ability to run in hybrid MPI+multicore mode (not tested yet)

"GPU Acceleration of an Established Solar MHD Code using OpenACC". Caplan et. al. J. of Phys.: Conf. Series. ASTRONUM 2018. 1225,1 (2019) 012012

"Acceleration of a production Solar MHD code with Fortran standard parallelism: From OpenACC to `do concurrent'" Caplan et. al. IEEE IPDPSW Proceedings., (2023) 582-590.

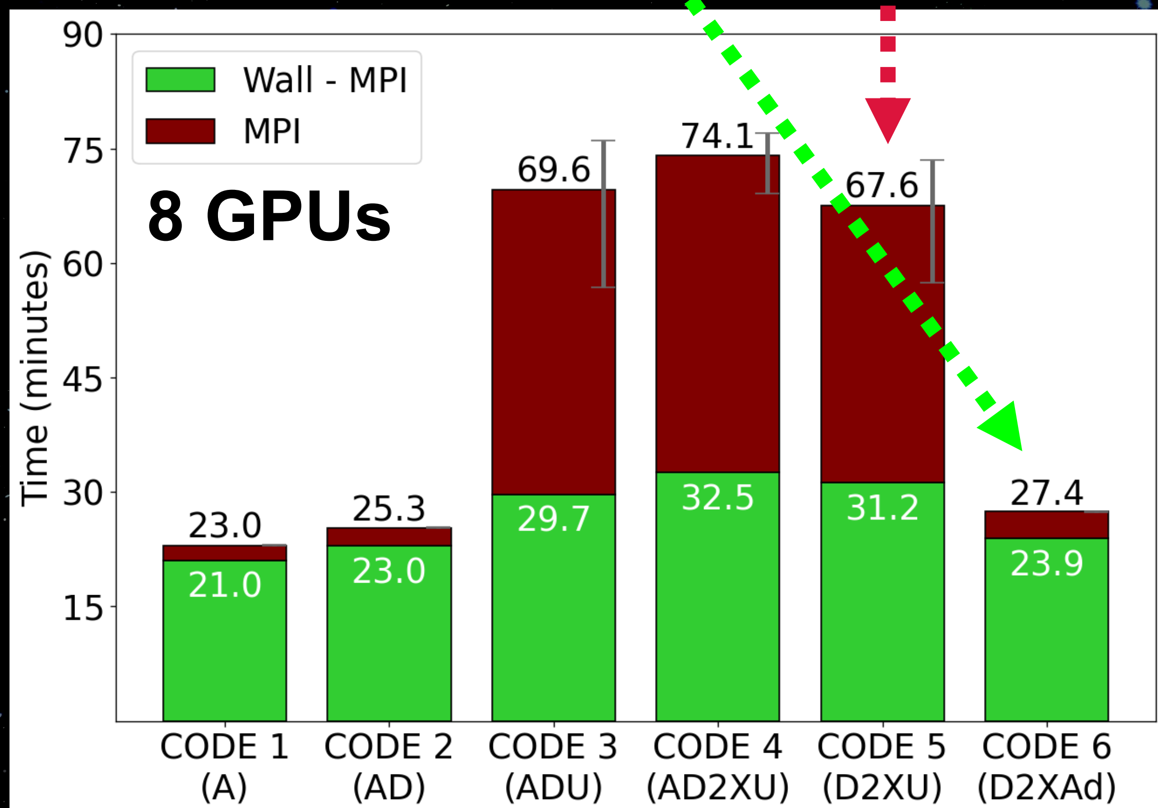




predsci.com/mas

Previous Implementations Results on NVIDIA GPUs: MAS

- We were able to run with pure Fortran using **UMM**, however the issues with **UMM**+MPI severely limited scaling across GPUs
- Adding back OpenACC data directives restored original scaling



Code Version	Total Lines	$acc Lines
0: CPU	69874	∅
1: A	73865	1458
2: AD	71661	540
3: ADU	71269	162
4: AD2XU	70868	55
5: D2XU	68994	∅
6: D2XAd	71623	277

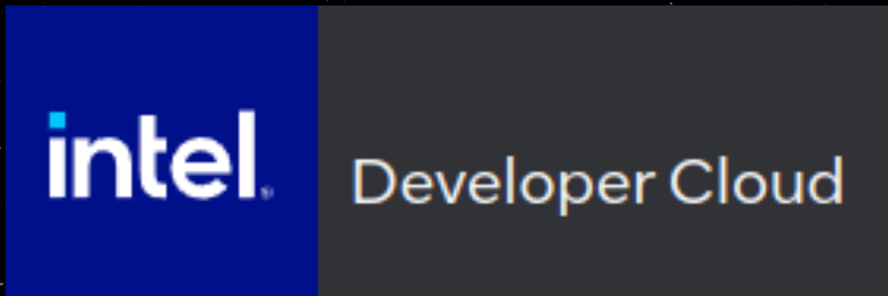
MAS
MAGNETOHYDRODYNAMIC ALGORITHM OUTSIDE A SPHERE

Wall Clock Time (minutes)
240
120
60
30
15
1
2
4
8
A100 (40GB) GPUs
CODE 1 (A)
CODE 2 (AD)
CODE 3 (ADU)
CODE 4 (AD2XU)
CODE 5 (D2XU)
CODE 6 (D2XAd)
Ideal Scaling

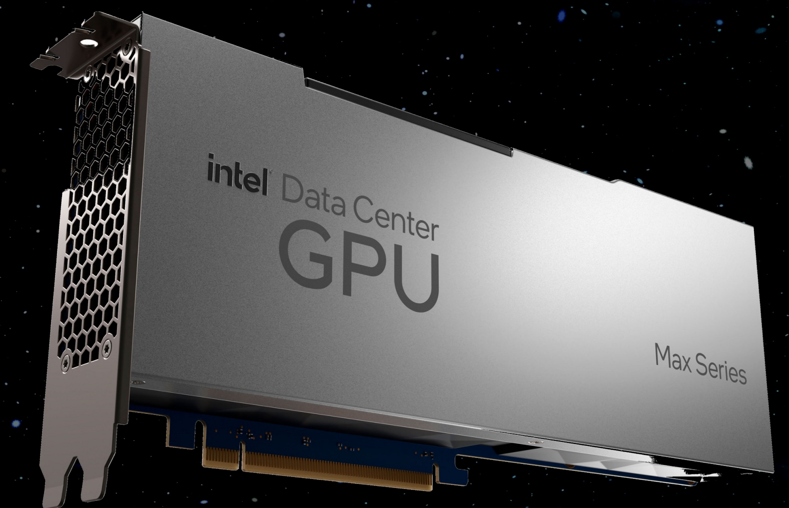
Preliminary Implementation Results on INTEL GPUs

- To test GPU-acceleration on Intel GPUs with DC, we go back to using the DIFFUSE tool
- We start with the pure Fortran version (*zero* directives)
- Test run is the same as in [Stulajter, et. al. (2021)]
- Using the Intel Developer Cloud, we use **`ifx`** compiler v2023.2 on an Intel MAX 1100 Data Center GPU
- DIFFUSE is highly memory-bandwidth bound so performance on the MAX 1100 (1,229 GB/s) expected to be between an NVIDIA V100 (900 Gb/s) and A100 (1,555 GB/s), where the test takes ~35 seconds on an A100
- We first ran the test on a dual-socket Xeon Platinum 8480+ CPU (614 GB/s) and it took a reasonable 95 seconds

DIFFUSE



`console.cloud.intel.com`



- For testing on the MAX GPU, we use the tips shown here:

  www.intel.com/content/www/us/en/developer/videos/offload-fortran-workloads-new-data-center-gpu-max.html

- Compiler flags:

```
-fiopenmp
-fopenmp-target-do-concurrent
-fopenmp-targets=spir64
-Xopenmp-target-backend "-device pvc"
```

- We set the following environment variable to profile the results (adding negligible extra time to the run)

```
export LIBOMPTARGET_PLUGIN_PROFILE=T
```

Preliminary Implementation Results on INTEL GPUs

DIFFUSE

- The test ran extremely slow
- The profile of the run shows that the slow performance is due to excesive amounts of CPU-GPU data transfers

LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Data Center GPU Max 1100, Thread 0

Kernel	Name
Kernel 0	__omp_offloading_10301_bd28a_ax__l1425
Kernel 1	__omp_offloading_10301_bd28a_ax__l1438
Kernel 2	__omp_offloading_10301_bd28a_ax__l1443
Kernel 3	__omp_offloading_10301_bd28a_ax__l1454
Kernel 4	__omp_offloading_10301_bd28a_diffuse_step_sts__l552
Kernel 5	__omp_offloading_10301_bd28a_diffuse_step_sts__l565

	Host Time (msec)				Device Time (msec)				
Name	Total	Average	Min	Max	Total	Average	Min	Max	Count
Compiling	329.88	329.88	329.88	329.88	0.00	0.00	0.00	0.00	1.00
DataAlloc	63045.73	0.03	0.00	3.16	0.00	0.00	0.00	0.00	1.81e+06
DataRead (Device to Host)	2.09e+06	2.16	0.01	15.08	1.44e+06	1.49	0.00	7.40	966000.00
DataWrite (Host to Device)	3.01e+06	1.33	0.00	18.31	1.49e+06	0.66	0.00	7.63	2.25e+06
Kernel 0	45135.81	1.12	0.83	2.29	31140.47	0.77	0.75	0.83	40260.00
Kernel 1	1787.53	0.04	0.04	0.90	1261.04	0.03	0.03	0.06	40260.00
Kernel 2	14269.57	0.35	0.06	2.20	318.11	0.01	0.01	0.04	40260.00
Kernel 3	1470.01	0.04	0.02	0.41	377.62	0.01	0.01	0.04	40260.00
Kernel 4	36.64	0.61	0.60	0.65	32.84	0.55	0.54	0.56	60.00
Kernel 5	30664.22	0.76	0.74	0.95	28215.62	0.70	0.68	0.75	40200.00

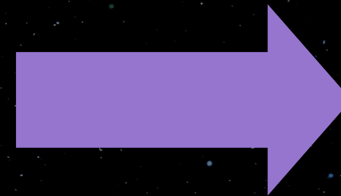
Total wall clock time: 5331.8 seconds

DIFFUSE

- `ifx` does not currently have an equivalent to UMM for DC, so to try to reduce data transfers, we add unstructured data region directives
- OpenACC is (unfortunately) not supported by `ifx`, so we use the OpenMP Target equivalents:

github.com/intel/intel-application-migration-tool-for-openacc-to-openmp

```
!$acc enter data copyin(a)
!$acc enter data create(b)
...
... COMPUTE ...
...
!$acc exit data copyout(a)
!$acc exit data delete(b)
```



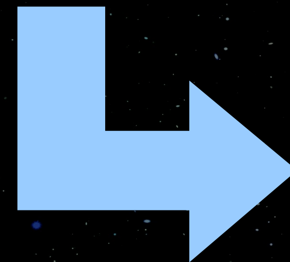
```
!$omp target enter data map(to:a)
!$omp target enter data map(alloc:b)
...
... COMPUTE ...
...
!$omp target exit data map(from:a)
!$omp target exit data map(release:b)
```

OpenACC **OpenMP Target**

DIFFUSE

- The run time improved only 15% (4480 seconds), which is still far too slow due to data transfers
- It turns out that `ifx` currently translates DC into OpenMP Target as:

```
do concurrent (i=1:10)
  x = ...
enddo
```



```
!$omp target teams loop map(always,tofrom:x)
do i = 1,10
  x = ...
enddo
```

- This mapping always performs CPU-GPU transfers on every loop, even if the data is already on the GPU (through OpenMP data regions)
- According to the specification, the behavior of an OpenMP Target loop with no mapping clause is "copy or present" (copy data if needed, but not if the data is already on the GPU)
- Changing (or adding user options) how `ifx` translates DC should be straight forward

Preliminary Implementation Results on INTEL GPUs

DIFFUSE

- To see what performance we can expect with updated mapping, we converted all DC loops into `do` loops with unmapped OpenMP target directives, keeping the unstructured data regions
- The test ran **much** better!

```
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Data Center GPU Max 1100, Thread 0
Kernel 0                  : __omp_offloading_10301_bd3bf_ax__l1454
Kernel 1                  : __omp_offloading_10301_bd3bf_ax__l1470
Kernel 2                  : __omp_offloading_10301_bd3bf_ax__l1476
Kernel 3                  : __omp_offloading_10301_bd3bf_ax__l1488
Kernel 4                  : __omp_offloading_10301_bd3bf_diffuse_step_sts__l565
Kernel 5                  : __omp_offloading_10301_bd3bf_diffuse_step_sts__l581
```

Name	Host Time (msec) Total	Average	Min	Max	Device Time (msec) Total	Average	Min	Max	Count
Compiling	372.01	372.01	372.01	372.01	0.00	0.00	0.00	0.00	1.00
DataAlloc	30.75	0.00	0.00	3.19	0.00	0.00	0.00	0.00	281854.00
DataRead (Device to Host)	548.55	0.01	0.01	2.90	144.88	0.00	0.00	1.43	80521.00
DataWrite (Host to Device)	513.99	0.01	0.00	14.16	16.82	0.00	0.00	7.36	80554.00
Kernel 0	31166.05	0.77	0.74	8.06	30979.28	0.77	0.73	0.84	40260.00
Kernel 1	1389.45	0.03	0.03	0.88	1197.80	0.03	0.03	0.08	40260.00
Kernel 2	275.31	0.01	0.01	0.04	107.58	0.00	0.00	0.03	40260.00
Kernel 3	278.59	0.01	0.01	0.04	116.55	0.00	0.00	0.03	40260.00
Kernel 4	31.62	0.53	0.52	0.55	31.28	0.52	0.51	0.54	60.00
Kernel 5	26509.14	0.66	0.65	1.02	26345.37	0.66	0.64	0.71	40200.00

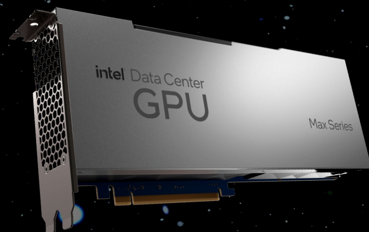
Total wall clock time: 62.1 seconds

Performance Summary for NVIDIA and INTEL GPUs

NVIDIA RTX 3090 Ti
(1.0 TB/s Mem Band)
`nvfortran`

DC (Pure Fortran) (UMM)	55.2 seconds
DC & OpenMP Target Data	54.2 seconds
OpenMP Target Loops & Data	54.6 seconds

INTEL MAX 1100
(1.2 TB/s Mem Band)
`ifx`



DC (Pure Fortran)	5331.8 seconds
DC & OpenMP Target Data	4480.0 seconds
OpenMP Target Loops & Data	62.1 seconds

- The mapping issue in `ifx` is expected to be fixed soon, which should yield efficient results for Intel GPUs DC with OpenMP for data management
- To use DC only, a system similar to NVIDIA's UMM would need to be implemented/activated in `ifx` for DC

Try Fortran's `do concurrent` (DC) to run your legacy (and new) codes on GPUs!

- On the NVIDIA platform, DC codes can be as fast as directive-based codes, with the best performance obtained by adding some Open(ACC|MP) data directives
- The Intel platform's support for DC is just beginning, and can yield decent performance on compute kernels
- With some manual OpenMP data directives and further compiler updates, DC codes are expected to run well on Intel GPUs in the near future
- With the portability of DC GPU-acceleration on NVIDIA and Intel, there is an incentive for an AMD implementation to be developed

EXTRA: Preliminary Results on *Consumer* **INTEL** GPUs

DIFFUSE

- We also tested the run on an Intel Arc A750 Limited Edition GPU (512 GB/s Memory Bandwidth)
- We use the OpenMP Target Loops & Data version of the code
- The Arc GPUs do not have hardware for double precision FLOPs, but can compute them using emulation:

```
export IGC_EnableDPEmulation=1
export SYCL_DEVICE_WHITE_LIST=""
export OverrideDefaultFP64Settings=1
```

- Compiler flags:

```
-fiopenmp -fopenmp-targets=spir64
-Xopenmp-target-backend "-device arc"
```

Total wall clock time:
161.2 seconds

Rocky Linux 9.2, Kernel 6.3.4-1, ifx 2024.0
Resizable bar enabled