Asynchronous 3-D FFTs using OpenMP offload for extreme problem sizes

Kiran Ravikumar[1], P.K. Yeung[1], Stephen Nichols[2],
Oscar Hernandez[3*], John Levesque[4], Dossay Oryspayev[5]

kiran.r@gatech.edu
pk.yeung@ae.gatech.edu

[1]Georgia Institute of Technology, [2]Oak Ridge National Lab., [3]NVIDIA, [4]Cray (HPE), [5]Brookhaven National Lab.
*Work performed while at ORNL and ECP SOLLVE project

OpenMP Users Monthly Teleconferences
May 28, 2021

What is the (3D) Fourier Transform, and why is it important?

Representing complex signals as sums of sines and cosines

- In wavenumber: $f(x) = \sum_k \hat{f}(k) \exp(ikx)$ or in frequency: $g(t) = \sum_\omega \hat{g}(\omega) \exp(i\omega t)$
- Forward transform: obtain set of coefficients from function values
- Inverse transform: obtain function values from the coefficients
- Can be extended to 3D in space: $f(\mathbf{x}) = \sum_{\mathbf{x}} \hat{f}(\mathbf{k}) \exp(i\mathbf{k} \cdot \mathbf{x})$
- Transforming one direction at a time:

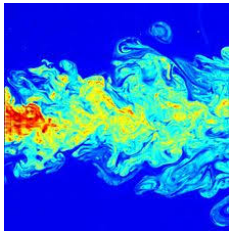
$$f(x,y,z) = \mathcal{F}_x^{-1}\left\{\mathcal{F}_y^{-1}\left\{\mathcal{F}_z^{-1}\left\{\hat{f}(k_x,k_y,k_z)\right\}\right\}\right\}$$

As effective methods of numerical solution of PDEs

- In some cases, equations governing $\hat{f}$ may be more readily solved numerically (which is our prime motivation in this talk)

Science Motivation: The Challenge of Fluid Turbulence

- Disorderly fluctuations over a wide range of scales in 3D space and time
- A physical problem of great complexity, and a critical factor in many disciplines



- Governing equations are known, but mathematically intractable
- Experiments, theory, modeling, computation all useful yet imperfect.
- Better physical understanding is required (e.g. think Covid-19)

Computing Turbulence: Direct Numerical Simulations

- Separate instantaneous velocity field into the sum of an averged state, and departures (fluctuations) from that state
- Form and solve (numerically) equations for the fluctuations
- Simplified geometries: periodic boundary conditions compatible with Fourier decompositions are numerically advantageous and physically useful.
- State-of-the-art around 2000 was 1024^3
- 4096^3 (Kaneda *et al*) on Earth Simulator in Japan, 2003
- In 2019 we reached world-leading $18,432^3$ using CUDA Fortran on Summit

Looking towards even larger problem sizes using OpenMP offload for portability

Navier-Stokes equations and Fourier pseudo-spectral methods

- Numerical solution of PDE governing velocity field $\mathbf{u}(\mathbf{x}, t)$

$$\partial \mathbf{u}/\partial t + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla(p/\rho) + \nu\nabla^2\mathbf{u} + \mathbf{f}$$

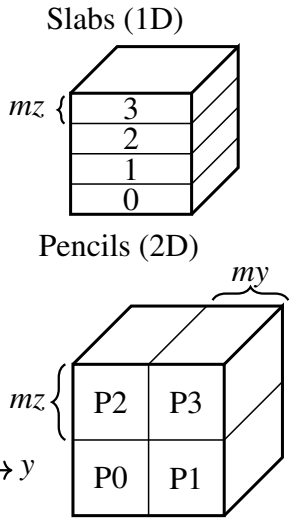
- Fourier decomposition: $\mathbf{u}(x,t) = \sum_{\mathbf{k}} \hat{\mathbf{u}}(\mathbf{k}) \exp(i\mathbf{k}\cdot\mathbf{x})$. In equation for Fourier coefficients nonlinear terms lead to convolution integrals, requiring $\sim N^6$ operations
- "Pseudo-spectral": form products first by multiplication in physical space, before transforming to wavenumber space. Fast Fourier Transform (FFT) $\propto N^3 \ln_2 N$ — but communication is required to make complete lines of data available.
- Aliasing errors in nonlinear terms: use truncation and phase-shifts (Rogallo 1981)
- Cost of simulation per step tied to a number of forward and backward transforms.

Efficient distributed 3D FFT on GPUs forms a key component

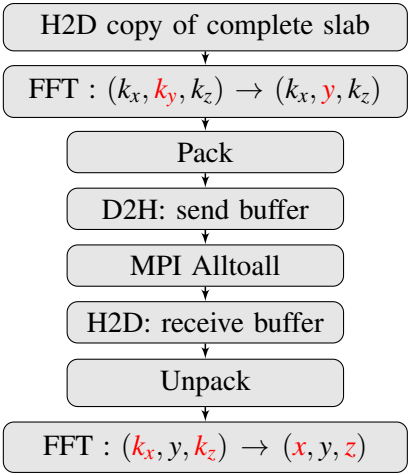
Domain Decomposition: 1D or 2D?

How best to distribute memory among P MPI tasks?

- 1D: Each MPI rank holds a slab
  — one global transpose among all processes ($x-y$ to $x-z$)
- 2D: Each rank holds a pencil
  — two transposes, within row and column communicators
- Pencils used for most large simulations (e.g. we ran 8192^3 using 262,144 MPI tasks on Blue Waters at NCSA)
- Fatter nodes and more GPUs per node: return to slabs?
  — GPU parallelism instead of distributed memory (MPI)
  — fewer nodes (and MPI tasks) in communication
  — associated pack and unpack operations are simplified



A basic (Synchronous) GPU algorithm

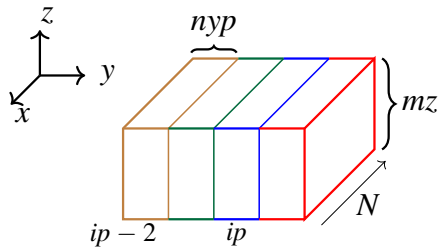


- Copy entire slab from CPU (host (H)) to GPU (device (D)) and back to CPU at end
- 1D FFTs in y, z, x directions using cufft library
- Pack and unpack data on GPU: faster than CPU
- MPI Alltoall among all tasks to transpose $x-y$ to $x-z$ slabs
- D2H and H2D copies of send and receive buffers before and after Alltoall
- Similar operations to transform back to wavenumber space from physical space

Large problem that may not fit on GPU? Any asynchronism possible?

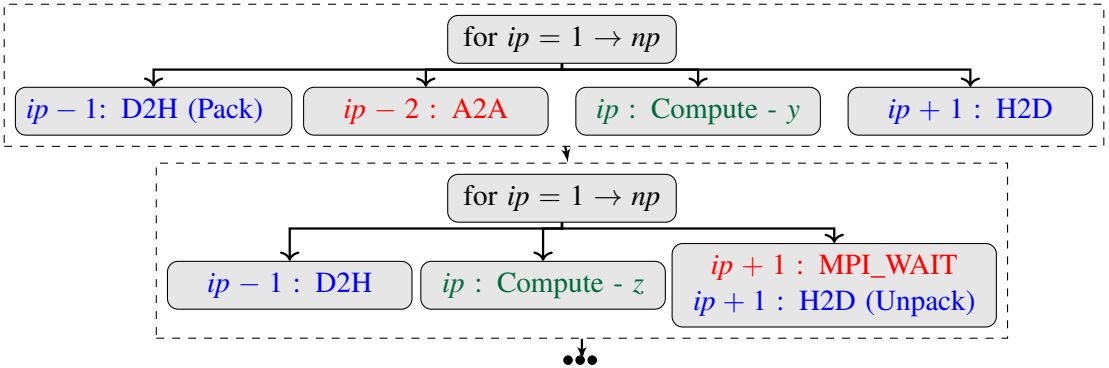
New batched asynchronous algorithm

- Divide slab into *np* pencils and process each pencil separately ($nyp = nxp = N/np$)
- Overlap operations on different pencils to hide some data transfer and compute costs



- Overlap using one stream each (in CUDA Fortran) for data transfer and compute
- Overlap: Compute on *ip*, HtoD on $ip + 1$, DtoH on $ip - 1$ and all-to-all on $ip - 2$
- Non-blocking all-to-all allows overlap, `MPI_WAIT` ensures completion
- GPU-Direct can be used to avoid copies before and after all-to-all
- Repeat until all pencils (*np*) processed on GPU and transposed

Batched asynchronism: Illustrated via operations in y and z



- Operations on same row executed asynchronously but launched from left to right
- Pack and unpack: strided data copy to avoid reordering data before transpose
- Non-blocking all-to-all allows overlap. Call MPI_WAIT before compute

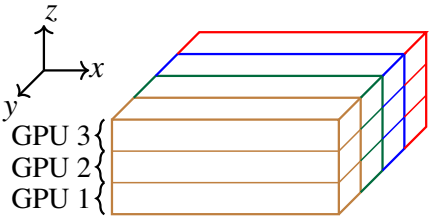
How many tasks per node?

Based on Summit node architecture

- 6 tasks per node: 1 task per GPU
- 2 tasks per node: 3 GPUs per task
  - — OpenMP threads to launch operations to GPUs
  - — 3 times fewer MPI tasks, 3 times larger message size

Number of pencils per all-to-all

- Does it affect the performance?
- 1 pencil at a time
  - — overlap MPI with data movement and compute
- Entire slab (*np* pencils) at a time
  - — no MPI overlap with data movement and compute
  - — *np* times larger message size and fewer MPI calls



Each pencil further divided vertically among multiple GPUs

MPI performance and strided copies

MPI performance occupies a significant fraction of runtime

- Message size between processes in *all-to-all* increases as number of processes decrease: reduce communication overhead and latency
- Transpose multiple pencils together: further increases message size

Many strided copies are needed: compute on part of slab, pack, unpack

- *zero-copy* (Appelhans 2018): GPU initiates many small transfers to/from host pinned memory; uses GPU compute resources for data transfer
- `cudaMemCpy2DAsync`: CUDA library call can handle simple strides without using GPU compute resources

More details on optimization can be found in Ravikumar *et al.* 2019

Fewer MPI tasks; zero-copy & MemCpy2D: optimal strided copies

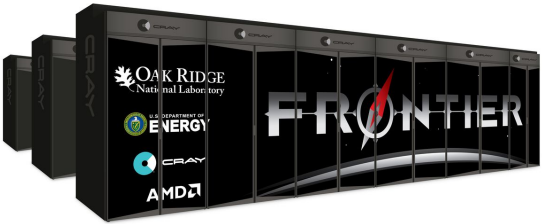
Batched asynchronous code performance (CUDA Fortran)

- Performance data collected on Summit
  — 2nd order Runge Kutta, 3 inverse and 5 forward transforms, 2 substages per timestep

Nodes	Problem Size	Time(s)			
		Sync CPU (Pencils)	Async GPU		
			6 tasks/node	2 tasks/node	
				1 pencil/A2A	1 slab/A2A
16	3072^3	34.38	8.09	6.70	7.50
128	6144^3	40.18	12.17	8.66	8.07
1024	12288^3	47.57	13.63	12.62	10.14
3072	18432^3	41.96	25.44	22.30	14.24

- 2 tasks/node performs better than 6 task/node for all problem sizes tested
- 128 nodes and above: 1 slab/A2A better than 1 pencil/A2A
  — suggests better overall performance without MPI overlapping GPU operations
- $18{,}432^3$: $\sim$ 3X speedup to pencils CPU version; communication bound code

Porting to future exascale architectures



- AMD CPU w/ 4 AMD GPUs per node
- Program GPUs: HIP, OpenMP

- 2 Intel CPUs w/ 6 Intel GPUs per node
- Program GPUs: oneAPI, OpenMP

Support for CUDA Fortran is not likely. Need efficient portable implementation.

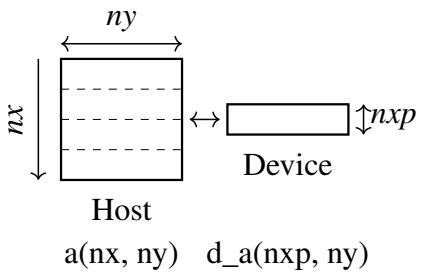
OpenMP is widely accepted standard and a clear favorite for Fortran

Non-contiguous maps and strided copies

- FFTs in *y*: need only *a(1:nxp, 1:ny)* on device
- In CUDA Fortran use: `cudaMemCpy2DAsync`

How to do it in OpenMP?

- `MAP(to:a(1:nxp, 1:ny))`: not 5.0 compliant
- Using `omp_target_memcpy_rect`
  — copy rectangular subvolume from a nD array
  — similar to 2D strided copies in CUDA
  — `TASK` for asynchronism (5.1: async version)
  — need C-FORTRAN interface (5.0 and lower)
- Using *zero-copy* kernels: GPU initiates many small transfers to/from host pinned memory [Appelhans GTC 2018]




```
1 omp_target_memcpy_rect (dst, src, elem_size, &
2     ndims, vol, dst_offset, src_offset, dst_dims, &
3     src_dims, dst_dev, src_dev)
```

OpenMP 4.5+: `omp_target_memcpy_rect`?

- Copy rectangular subvolume from a multi-dimensional array
- Callable from C/C++, use C-Fortran interface
- *ndims*: no. of dimensions in array
- *vol*: no. of elements to copy in each dimension
- *offset*: no. of elements from *base of each dimension*, after which to copy data from/to
  – In 5.0: from *origin of dst (src)*, need clarity
- *dims*: no. of elements in each dimension
- Need to account for C vs. Fortran ordering
  – first dimension along row (*ny*) even though in Fortran it is along column

```
1  ! src on host of shape (nx, ny)
2  ! dst on device of shape (nxp, ny)
3
4  ! copy src(1:nxp, 1:ny) to dst(1:nxp, 1:ny)
5
6  num_dims = 2
7  vol(1) = ny ; vol(2) = nxp
8   dst_offset (1) = 0 ;  dst_offset (2) = 0
9   src_offset (1) = 0 ;  src_offset (2) = 0
10 dst_dims(1) = ny ; dst_dims(2) = nxp
11 src_dims(1) = ny ; src_dims(2) = nx
12
13 omp_target_memcpy_rect(dst, src, elem_size,
    ndims, vol, dst_offset , src_offset ,
    dst_dims, src_dims, dst_dev, src_dev)
```

Zero-copy kernels for complex strided copies

```
1  TARGET ENTER DATA MAP(to:d_buf) &
2  DEPEND(IN:indep) DEPEND(OUT:tdep) NOWAIT
3
4  TARGET TEAMS DISTRIBUTE PARALLEL DO &
5  COLLAPSE(4) IS_DEVICE_PTR(h_buf) &
6  DEPEND(INOUT:tdep) NOWAIT
7  do yg=1,numtasks
8     do z=1,mz
9        do y1=1,my
10          do x=1,nx
11             y = my*(yg-1)+y1
12             d_buf(x,y,z) = h_buf(x,z,y1,yg)
13          end do
14       end do
15    end do
16 end do
17 END TARGET TEAMS DISTRIBUTE PARALLEL DO
18
19 TARGET EXIT DATA MAP(from:d_buf) &
20 DEPEND(IN:tdep) DEPEND(OUT:outdep) NOWAIT
```

- GPU threads copy data to device buffer (`d_buf`) by directly accessing host resident **pinned** memory (`h_buf`)
- `IS_DEVICE_PTR` to make the host buffer accessible to GPU threads
- `h_buf` is dummy argument, separate subroutine with `h_buf` passed into it
- Strided read and write, transpose y and z
- Uses GPU compute resources for copy, slows down other computes
- Best for more complex stride patterns, like unpacking

Interoperability between OpenMP and non-blocking libraries

```
1  TARGET DATA MAP(tofrom: a)
2
3  TASK DEPEND(out:var)                                    Ⓐ
4
5  TARGET DATA USE_DEVICE_PTR(a)
6  FFTExecute (a, forward, stream)
7  FFTExecute (a, inverse , stream)
8  END TARGET DATA
9
10 END TASK
11
12 ! Copy or compute on other data                         Ⓒ
13
14 TARGET TEAMS DISTRIBUTE DEPEND(IN:var) NOWAIT
15 a(:, :, :) = a(:, :, :)/nx                              Ⓑ
16 END TARGET TEAMS DISTRIBUTE
17
18 END TARGET DATA
```

- Ⓐ: launch FFT kernel to GPUs
- Ⓑ waits as dependent on Ⓐ
- Ⓒ executes asynchronously
- Ⓐ finishes prematurely once FFTs launched, does not wait for kernels to finish executing on GPU
- Ⓑ starts to run before FFTs complete on GPU, incorrect results

Timeline: OpenMP and non-blocking library

Asynchronous execution of cudaFFT library and OpenMP TARGET loop

- `TASK DEPEND` used to establish synchronization between FFT & TARGET loop
- Host thread launches FFT & then GPU compute before FFT completes
- `Detach` in OpenMP 5.0: signals event completion for depending tasks to continue

OpenMP 5.0 features critical for asynchronism

DETACH to enforce synchronization

```
1  TARGET DATA MAP(tofrom: a)
2
3  TASK DEPEND(out:var) DETACH(event)
4
5  TARGET DATA USE_DEVICE_PTR(a)  Ⓐ
6  FFTExecute (a, forward, stream)
7  FFTExecute (a, inverse , stream)
8  END TARGET DATA
9
10 cudaStreamAddCallback (stream, ptr_callback, C_LOC(event), 0)
11 END TASK
12
13 ! Copy or compute on other data  Ⓒ
14
15 TARGET TEAMS DISTRIBUTE DEPEND(IN:var) NOWAIT
16 a (:, :, :) = a (:, :, :) /nx                    Ⓑ
17 END TARGET TEAMS DISTRIBUTE
18
19 END TARGET DATA
```

```
1 subroutine callback (stream, status, event)
2   type( c_ptr ) :: event
3   integer(kind=omp_event_handle_kind) :: f_event
4   call C_F_POINTER (event, f_event)
5   call omp_fulfill_event( f_event )
6 end subroutine callback
```

- Ⓐ: launch FFT, add *callback* in stream where FFT will run
- Ⓑ waits as dependent on Ⓐ, Ⓒ executes asynchronously
- Ⓐ finishes after event fulfilled by *callback*

Porting asynchronous CUDA Fortran to OpenMP

```
1  do ip=1,np
2      NEXT = mod(ip+1,3); CURR = mod(ip,3);
3      PREV = mod(ip−1,3); COMM = mod(ip−2,3);
4      cudaStreamWaitEvent (trans_stream, DtoH(NEXT), 0)
5      cudaMemCpy2DAsync (abuf(NEXT),a(ip+1),trans_stream)
6      cudaEventRecord (HtoD(NEXT),trans_stream)
7      cudaStreamWaitEvent (comp_stream, HtoD(CURR), 0)
8      FFTExecute  (abuf(CURR), comp_stream)
9      cudaEventRecord (comp(CURR), comp_stream)
10     cudaStreamWaitEvent (trans_stream, comp(PREV), 0)
11     cudaMemCpy2DAsync (snd(ip-1), abuf(PREV), &
12         trans_stream)
13     cudaEventRecord (DtoH(PREV), trans_stream)
14     cudaEventSynchronize (DtoH(COMM))
15     MPI_IALLTOALL (snd(ip-2))
16 end do
```

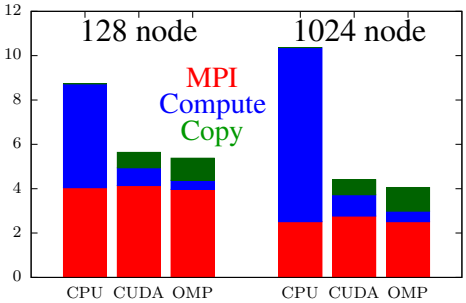
```
1  do ip=1,np
2      NEXT = mod(ip+1,3); CURR = mod(ip,3);
3      PREV = mod(ip−1,3); COMM = mod(ip−2,3);
4      TASK DEPEND (IN:DtoH(NEXT), OUT:HtoD(NEXT))
5      omp_target_memcpy_rect (abuf(NEXT), a(ip+1))
6
7      TASK DEPEND (IN:HtoD(CURR), OUT:comp(CURR))
8          DETACH(event)
9      FFTExecute  (abuf(CURR), comp_stream)
10     TASK DEPEND (IN:comp(PREV), OUT:DtoH(PREV))
11     omp_target_memcpy_rect (snd(ip−1),  abuf(PREV))
12
13
14     TASK DEPEND(IN:DtoH(COMM))
15     MPI_IALLTOALL (snd(ip-2))
16 end do
```

- `DEPEND` clause replaces `cudaEventRecord` & `cudaStreamWaitEvent`
- `omp_target_memcpy_rect` replaces `cudaMemCpy2DAsync`

Performance: Non-Batched synchronous version

Summit (XL compiler) up to 1024 nodes ($\sim 22\%$ of full machine) using 1 task/GPU
Timings for 3 pairs of forward and inverse transforms

# Nodes	Prob. Size	Time (s)		
		CPU	CUDA	OMP
2	1536^3	5.21	2.39	2.41
16	3072^3	6.79	3.30	3.16
128	6144^3	9.10	5.26	5.01
1024	12288^3	10.59	4.30	4.12

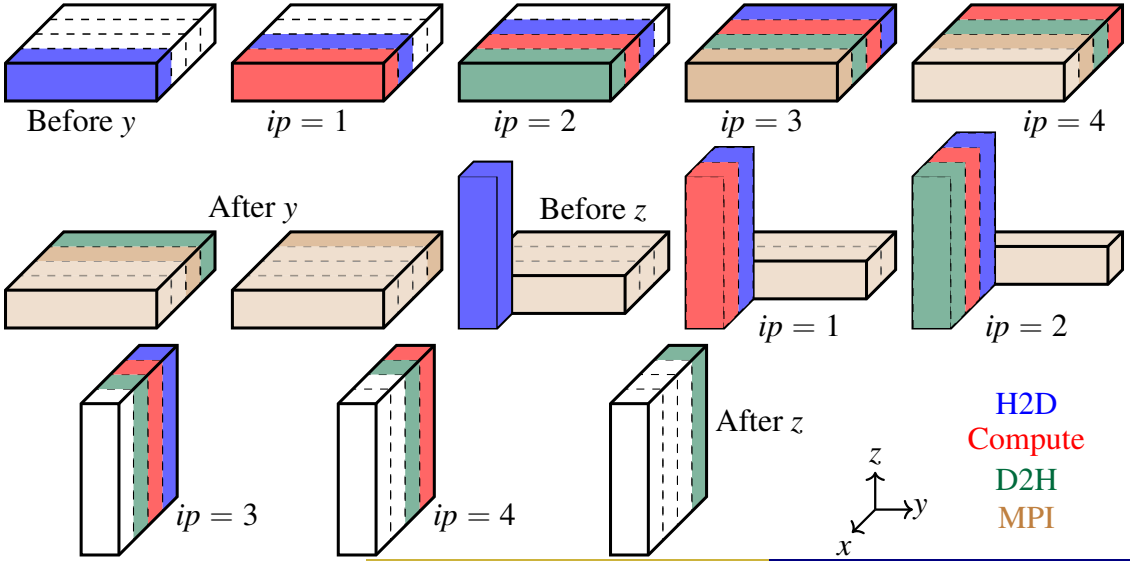


- OpenMP & CUDA show similar performance ($\sim$ 2.6X speedup for $12k^3$)
- GPU: compute negligible but additional cost due to copies, 62% in MPI
- OpenMP data copies slower than in CUDA, but compute faster !
- OpenMP code also works with CCE compiler and AMD GPUs

Summary and Future Work

- Developed algorithm for Summit using CUDA Fortran to run 18432^3 problem size
- Preliminary steps taken towards portability using OpenMP for offload
- Some challenges of portability overcome, some pending full OMP 5.0 availability
- Strided copy b/w small device & larger host arrays: `omp_target_memcpy_rect`
- Synchronizing non-blocking GPU library calls & OpenMP tasks: `DETACH`
- Future work towards 3D FFTs at massive scale, at resolution beyond $18{,}432^3$
  - Batched asynchronism algorithm (using `DETACH`) needed for optimal performance
  - A framework for portable GPU parallelism for communication-intensive applications

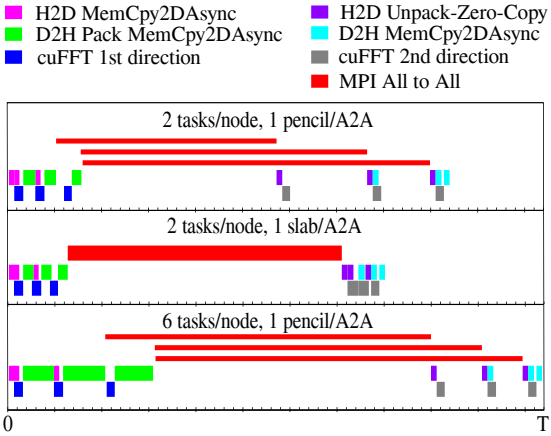
Batched async. algorithm: additional details



Timeline and asynchronous MPI analysis

- Slab/A2A better to pencil/A2A?
  - Faster MPI of data as one large msg
- Why is 2 tasks/node better than 6?
  - Each MPI longer: small P2P, more tasks
  - Slow pack: 3X Cpy2D, high overhead
  - Use ZC: steals GPU resources slowing 2 tasks/node

Normalized timeline of $12,288^3$ on 1024 nodes



MPI dominates runtime; No MPI overlap shows best performance

Performance: Batched version

- OpenMP version: copy on host from large buffer to small buffer before `UPDATE` (workaround)
  – `omp_target_memcpy_rect` slow compared to workaround and `cudaMemCpy2D`
- $6k^3$ OMP is 16.1s slower than CUDA async
  – 12.4s to copy one buffer to another on host
  – 3.7s (or 20%) saving due to asynchronism?
- Work in progress: optimize OpenMP version
  – Fast rectangular copy to avoid host operations
  – DETACH will help enable asynchronism
- Both OMP codes work with CCE & AMD GPUs

6 pencils per slab
Performance on Summit using XL
OpenMP version uses workaround

# Nodes	Prob. Size	Time (s)	
		CUDA async	OMP sync
4	3072^3	10.14	26.20
32	6144^3	13.53	29.64

Production code using CUDA: $18k^3$ on $3k$ nodes, $\sim$ 3X speedup