Fine grained asynchronism for pseudo-spectral codes - with application to turbulence

Kiran Ravikumar[1], David Appelhans[2], P. K. Yeung[1]

kiran.r@gatech.edu

[1]Georgia Institute of Technology, [2]IBM research

3rd OpenPOWER Academia Discussion Group Workshop
Dallas, TX; November 10, 2018

Introduction and Motivation

Petascale computing mainly via massive distributed parallelism

- Extreme core counts, w/ or w/o some shared memory (MPI+OpenMP)
- Scalability often ultimately limited by communication costs

Now in pre-Exascale era, heterogeneous architectures are dominant

- Accelerators provide most of computing power, at cost of non-trivial data movements
- But communication may still be an issue (perhaps even more so)
- On the largest machines, adaptation to hardware (e.g. GPUs) is a must

How do we do it, on Summit? [for turbulence, our domain science]

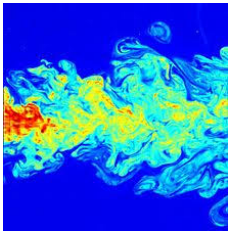
- Multidimensional FFT provides a case study of wide relevance
- Fine-grained asynchronism: while managing memory and data movements

OUTLINE OF THIS TALK

- Turbulence and the need for extreme scale computing
  - pseudo-spectral codes and Fourier transforms
- How to make use of GPUs effectively (specifically, on Summit)
  - while taking full advantage of large CPU memory (fat nodes)
  - optimizing data movements between CPU and GPU
  - asynchronism at a fine-grained level, using CUDA Fortran
- Performance recorded on Summit Phase 1 (to date, up to 1024 nodes)
- Other thoughts: such as optimizing communication

WHY DO WE NEED HUGE TURBULENCE SIMULATIONS?

- Turbulence is found everywhere in nature and engineering



- Disorderly fluctuations over a wide range of scales in 3D space and time
- To advance understanding, obtain numerical solutions of governing equations
- Capture wide range of scales, resolve small scales better than in the past, cover parameter regimes previously relatively unexplored, etc.

Direct Numerical Simulations (DNS)

- Compute all scales, according to (here incompressible) Navier-Stokes equations

$$\nabla \cdot \mathbf{u} = 0$$

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

- Study fundamental behavior of turbulent flows using simplified geometries
  — 3D Fourier decomposition on periodic domain appropriate for the small scales
- Pseudo-spectral in space for FT of nonlinear terms: instead of expensive convolution sums, perform multiplication in physical space; take FFTs back and forth
- Explicit second order Runge-Kutta time stepping scheme
- On Blue Waters at NCSA: production simulation at 8192^3 grid resolution (Yeung *et al.* PNAS 2015), with some short tests at 12288^3 and 16384^3
- INCITE 2019 on Summit: our target is 18432^3: i.e. $O(6)$ trillion grid points

How to compute 3D FFTs?

Transforms between physical (x, y, z) and wavenumber (k_x, k_y, k_z) spaces, any variable ϕ

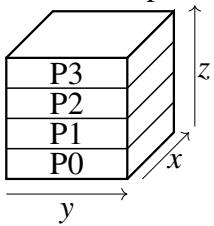
$$\phi(\mathbf{x}) = \sum_{\mathbf{k}} \hat{\phi}(\mathbf{k}) \exp(i\mathbf{k} \cdot \mathbf{x}) \quad ; \quad \hat{\phi}(\mathbf{k}) = \sum_{\mathbf{x}} \phi(\mathbf{x}) \exp(-i\mathbf{k} \cdot \mathbf{x})$$

- 1D FFTs, 1 direction at a time, but complete lines of data must be available
  — use FFTW (on CPUs) or cuFFT (on GPUs)
- Domain decomposition (1D or 2D) among MPI processes
  — up to N vs N^2 MPI tasks allowed
- Collective communication (of "alltoall" type) needed to complete full 3D FFT cycle
  — protocols for faster communication are of broad interest
- Use of strided (non-contiguous) arrays may require packing and unpacking, which are local (serial) in memory but still time consuming

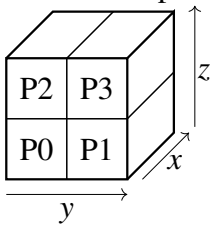
Domain Decomposition: 1D or 2D

- 1D: Each MPI rank holds a slab of size $N \times N \times N/P$, i.e. a collection of N/P planes of data
  — one global transpose (e.g. $x-y$ to $x-z$)
- 2D: Pencils, of size $N \times N/P_1 \times N/P_2$ where $P_1 \times P_2 = P$ define a 2D Cartesian process grid (whose shape matters)
  — two transposes, within row and column communicators
- Massive distributed memory parallelism favors Pencils
- Fatter node architectures : return to 1D decomposition?
  - Fewer MPI and associated pack and unpack operations
  - Fewer nodes (and MPI ranks) involved in communication
  - Large host memory allows for many slabs per node

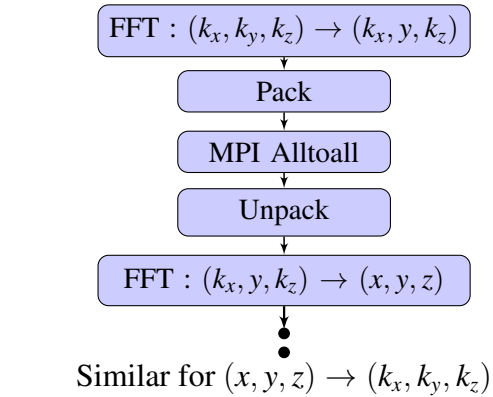
Slabs decomposition

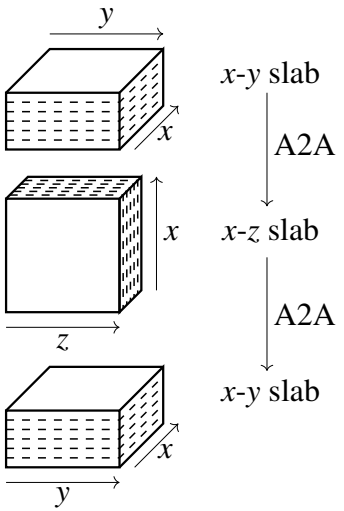


Pencils decomposition



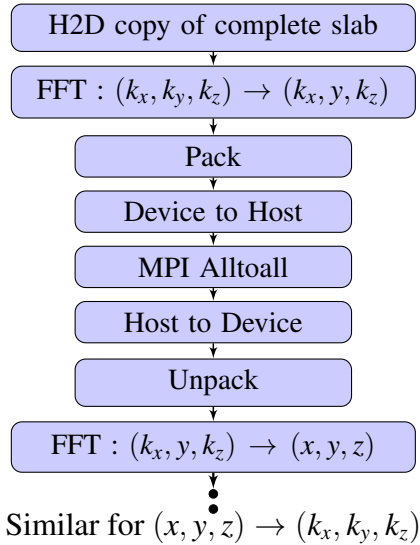
MULTI-THREADED SYNCHRONOUS CPU ALGORITHM





- FFTW library to carry out 1-D transforms
- OpenMP threads to speedup FFT, pack & unpack

SYNCHRONOUS GPU ALGORITHM



- Steps involved in transform from wavenumber to physical space shown
- Entire slab of data is copied from host to device
- Pack and unpack data on GPU
  - Additional buffers occupy limited GPU memory
  - Faster than on CPU
- Similar operations performed to transform to wavenumber from physical space with the last step being a device to host copy of the entire slab
- How to handle problem sizes that do not fit on the GPU?

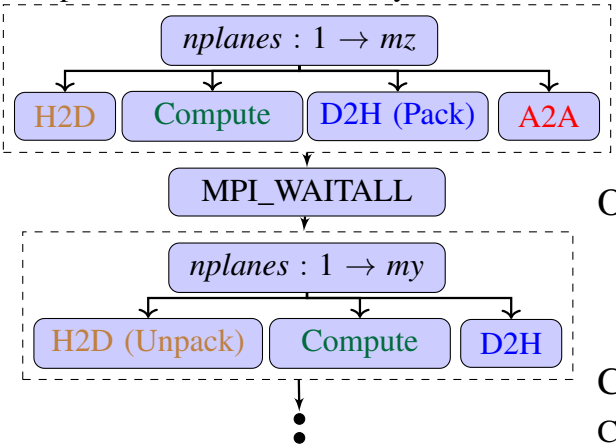
How can summit help?

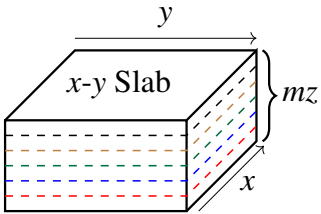
- 200 PF : Fastest supercomputer in the world
- 4608 nodes each with 2 IBM POWER9 CPU's & 6 NVIDIA Volta GPU's
- 512 GB host memory per node & 16 GB GPU memory : large problem size
- 50 GB/s peak CPU-GPU NVLink B/W : faster copies
- 23 GB/s Node injection B/W with Non-blocking fat tree topology : faster communication
- Enables 18432^3 DNS using 3072 nodes

Run massive problem sizes on fewer nodes than other machines currently available!

ASYNCHRONOUS GPU ALGORITHM

Operations in same row : asynchronous





Colors match operations in chart

Only 3 planes on GPU

- Problem size not limited by GPU memory (can also split planes)
- Overlap operation across 3 planes

CUDA streams for async progress

Compute (compute stream) & NVLink (transfer stream) overlapped by MPI

ASYNCHRONOUS EXECUTION USING CUDAFORTRAN

First plane is copied in before entering the following loop.

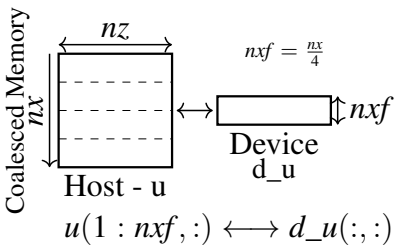
```
1  do zp=1,mz
2      next = mod(zp+1,3); current = mod(zp,3); previous = mod(zp−1,3); comm = mod(zp−2,3);
3      ! if any of above are zero, set to 3, negative cases are not considered
4
5      ! Host−>Device copy of plane zp+1 (NEXT) using cudaMemCpy or similar in transfer stream
6      ierr =cudaEventRecord(HtoD(next),trans_stream)
7
8      ierr =cudaStreamWaitEvent(comp_stream,HtoD(current),0)
9      ! Compute on zp plane (CURRENT) in compute stream
10     ierr =cudaEventRecord(compute(current),comp_stream)
11
12     ierr =cudaStreamWaitEvent(trans_stream,compute(previous),0)
13     ! Device−>Host copy of plane zp−1 (PREVIOUS) using cudaMemCpy or similar in transfer stream
14     ierr =cudaEventRecord(DtoH(previous),trans_stream)
15
16     ierr =cudaEventSynchronize(DtoH(comm))
17     ! MPI Alltoall on plane zp−2 copied out in the previous iteration
18 end do
```

Last plane is copied out after the loop. MPI on last 2 planes are performed.

NON-CONTIGUOUS DATA COPIES

Non-contiguous memory copy b/w host & device

- Pack and transfer
- Many instances of cudaMemCpy, each with small contiguous msg - high overhead



Zero copy (D. Appelhans GTC 2018) - accessing host resident pinned memory directly from GPU without having to copy the data to the device beforehand (i.e. there are zero device copies)

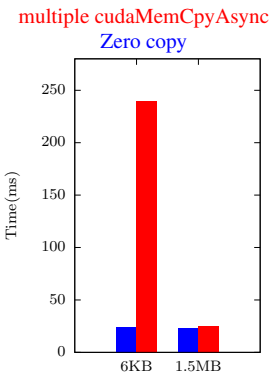
cudaMemCpy2DAsync -

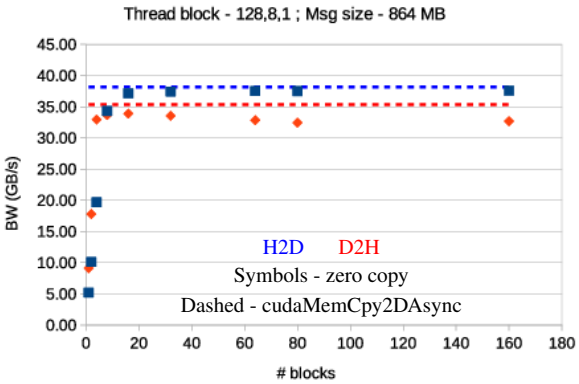
```
1  ! call cudaMemCpy2DAsync (dst,dpitch,src,spitch,&
2  !    width, height, kind, stream)
3  call cudaMemCpy2DAsync (d_u,nxf,u,nx,&
4     nxf, nz, kind, stream)
```

Is there a faster way?

- Zero copy
- cudaMemCpy2DAsync

ZERO COPY AND CUDAMEMCPY2DASYNC

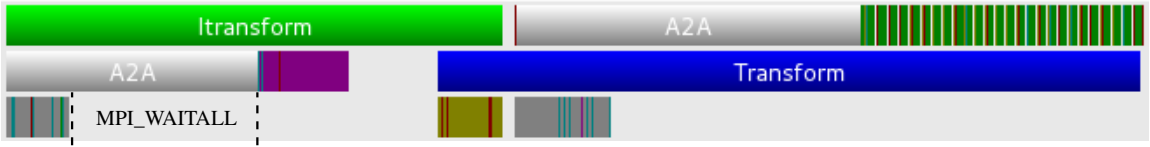




- Zero copy preferred when coalesced message size is small ($<$ 10KB for 18432^3)
- 16 threadblocks sufficient to get maximum NVLink throughput.
- GPU resources can be used for computation instead.
- Ideally use cudaMemCpy2DAsync to avoid using up GPU resources

Is there asynchronous execution?

NVPROF timeline for async. GPU code using zero copy for one Runge-Kutta substep



Compute and NVLINK (other colors). Zero copy kernels (dark green) also incur significant cost.
Further operations have to wait on previous alltoall to complete.

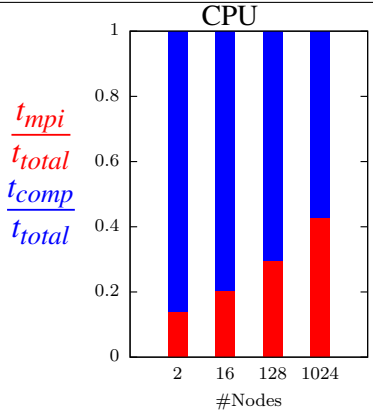
GPU computation (bottom) and NVLINK transfer (top, almost continuous) streams
Computations begin once data transfer on previous buffer is done.

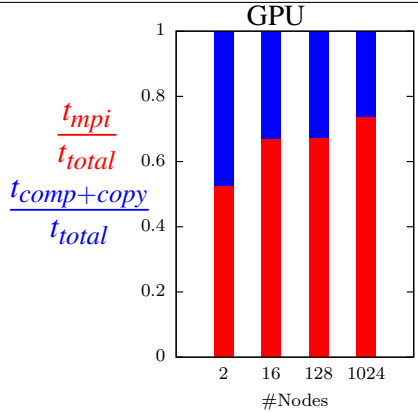
PERF. OF MULTI-THREADED CPU & ASYNC. GPU CODES

Problem size and run configuration

2 nodes - 1536^3 16 nodes - 3072^3 128 nodes - 6044^3 1024 nodes - 12288^3

6 MPI tasks per node & 7 threads per task

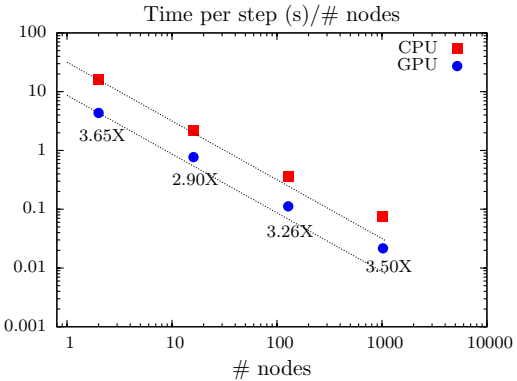




Computations accelerated using GPU's, while % communication increases

SCALING & SPEEDUP OVER MULTI-THREADED CPU CODE

Perf. of GPU async and CPU DNS codes Summit Phase-1 (Early Science Proposal (ESP))

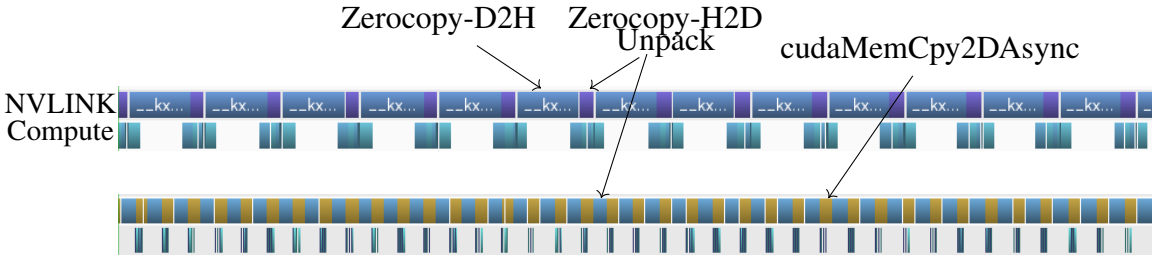


Dashed line denotes ideal weak scaling.
GPU to CPU speedups labeled in the plot.

# nodes	Prob. Size	Time(s)	
		(a)	(b)
2	1536^3	31.70	8.67
16	3072^3	35.69	12.32
128	6144^3	46.68	14.34
1024	12288^3	77.60	22.15

(a) - CPU
(b) - Async GPU using zero copy

DOES CUDAMEMCPY2DASYNC IMPROVE PERFORMANCE?

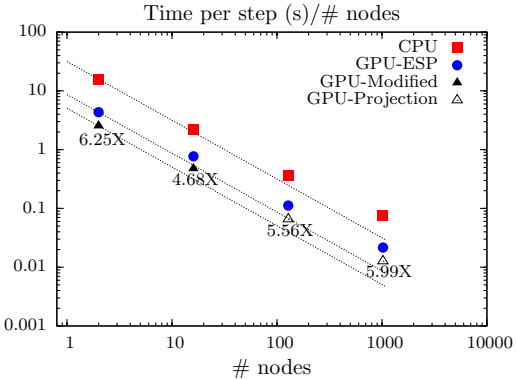


NVPROF timeline for a period of 60 *ms* showing the asynchronous behavior of GPU code. Top - (a) Using zerocopy kernels. Bottom - (b) Using cudaMemCpy2DAsync instead of D2H zerocopy kernel

- (a) : GPU resources shared b/w zerocopy & compute kernels, slowing down both
- (b) : GPU resources available for compute, speeding up data transfers and compute
- (b) : Some zero copy kernels needed for transfers with a more complicated striding pattern, minimal performance effect

PERFORMANCE IMPROVEMENTS & PROJECTIONS

Perf. of async. GPU using cudaMemCpy2DAsync instead of some zerocopy kernels.



# nodes	Prob. Size	Time(s)	
		(a)	(b)
2	1536^3	8.67	5.07
16	3072^3	12.32	7.63
128	6144^3	14.34	8.39
1024	12288^3	22.15	12.96

(a) - using zero copy
(b) - using cudaMemCpy2DAsync

The *projected* performance of the code for 128 and 1024 nodes is estimated assuming the same weak scaling performance as seen in the earlier runs (GPU-ESP).

MPI ALLTOALL COLLECTIVE PERFORMANCE

Environment variables :

- PAMI_ENABLE_STRIPING=0
  - Disable splitting data to network devices
- PAMI_IBV_ADAPTER_AFFINITY=1
  - Use nearest network device
- PAMI_IBV_DEVICE_NAME=”mlx5_0:1”
  - Use this device if on socket 0
- PAMI_IBV_DEVICE_NAME_1=”mlx5_3:1”
  - Use this device if on socket 1

$> 10\%$ perf. improvement compared to default environment

$$Eff.BW = \frac{2 \times size\ of\ sndbuf\ in\ GB \times tpn}{time} = \frac{2 \times N \times \frac{N}{P} \times P \times \frac{N}{P} \times 4 \times tpn}{time \times 1024^3}$$

2 - read+write ; $\frac{4}{1024^3}$ - GB ; *tpn* - tasks per node

```
1  do istep=1,nsteps
2
3      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
4     rtime1=MPI_WTIME()
5     call MPI_ALLTOALL(sndbuf,nx*my*mz,&
6        MPI_REAL,rcvbuf,nx*my*mz,&
7        MPI_REAL,MPI_COMM_WORLD,ierr)
8     rtime2=MPI_WTIME()
9     time_step( istep )=rtime2-rtime1
10
11 end do
```

How to improve alltoall performance?

Tested different methods

- Blocking MPI Alltoall
- Non-blocking MPI Alltoall
- Non-blocking sends and receives
- One-sided MPI
- Hierarchical method

Hierarchical Alltoall (A2A)

- A2A on Node (Row communicator)
- Pack to form contiguous messages
- A2A across Node (Column comm.)

Alltoall using One-sided MPI

```
1  call MPI_WIN_CREATE(sndbuf,size,sizeofreal,&
2    MPI_INFO_NULL,MPI_COMM_WORLD,sndwin,ierr)
3
4  call MPI_WIN_FENCE(0,sndwin,ierr)
5
6  call MPI_GET_ADDRESS(sndbuf(1,1,taskid+1),disp,ierr)
7  disp=MPI_AINT_DIFF(disp,base)
8  disp=disp/ sizeofreal
9
10 do zg=1,numtasks
11    call MPI_GET(rcvbuf(1,1,zg),nx*my,MPI_REAL,&
12       zg-1,disp,nx*my,MPI_REAL,sndwin,ierr)
13 end do
14 call MPI_WIN_FENCE(0,sndwin,ierr)
```

None of these methods seem to perform better than simple MPI alltoall

CONCLUSIONS

- Successful development of a highly scalable GPU-accelerated algorithm for turbulence and 3DFFT exploiting unique features of Summit
  — fine-grained asynchronism allows problem sizes beyond GPU memory
- CUDA Fortran implementation gives GPU speedup > 4 at smaller problem sizes, likely to hold up to largest problem size feasible on Summit (18432^3)
  — Host-to-device and device-to-host data copies optimized using "zero-copy" approach and cudaMemCpy2DAsync, depending on message size
- Communication costs still a major factor
  — bandwidth measured rigorously and various approaches tested
- Working towards OpenMP 4.5 or higher implementation
- Towards extreme-scale simulations w/ INCITE 2019 + Early Science Program.

REFERENCES

[1] P. K. Yeung, X. M. Zhai, and K. R. Sreenivasan. Extreme events in computational turbulence. *Proc. Nat. Acad. Sci.*, 112:12633–12638, 2015.

[2] P. K. Yeung, Sreenivasan. K. R., and S. B. Pope. Effects of finite spatial and temporal resolution on extreme events in direct numerical simulations of incompressible isotropic turbulence. *Phys. Rev. Fluids*, 3:064603, 2018.

[3] M. P. Clay, D. Buaria, P. K. Yeung, and T. Gotoh. GPU acceleration of a petascale application for turbulent mixing at high Schmidt number using OpenMP 4.5. *Comput. Phys. Commun.*, 228:100–114, 2018.

[4] G. Ruetsch and M. Fatica. *CUDA Fortran for Scientists and Engineers*. Morgan Kaufmann Publishers, 2014.

[5] D. Appelhans. Tricks, Tips, and Timings: The Data Movement Strategies You Need to Know. In *GPU Technology Conference*, 2018.

[6] K. Ravikumar, D. Appelhans, P. K. Yeung, and M. P. Clay. An asynchronous algorithm for massive pseudo-spectral simulations of turbulence on Summit. In *Poster presented at the OLCF Users Meeting*, 2018.

STRIDED COPIES

```
real , allocatable , device :: devicebuf (:,:) ,&
    d_hostbuf (:,:)
type(C_DEVPTR) :: d_temp

! get pointer on device that points to host array
ierr =cudaHostGetDevicePointer(d_temp,&
    C_LOC(hostbuf(1,1)),0)
! convert to Fortran pointer
call C_F_POINTER(d_temp,d_hostbuf,[nx,nz])

call zc_h2d<<<grid,tblock,0,trans_stream>>>&
    (devicebuf,d_hostbuf,nx,nxf,nz,ix)
```

```
1  attributes(global) subroutine zc_h2d(devicebuf,&
2      hostbuf,nx,nxf,nz,ix)
3
4  real, device, intent(INOUT) :: devicebuf(nxf,nz),&
5      hostbuf(nx,nz)
6  integer, value, intent(IN) :: nx,nxf,nz,ix
7  integer :: z,x
8
9  do z=blockIdx%x,nz,gridDim%x
10     do x=threadIdx%x,nxf,blockDim%x
11         devicebuf(x,z)=hostbuf(ix+x-1,z)
12     end do
13 end do
14
15 end subroutine zc_h2d
```

SECTIONAL DATA COPIES USING OPENMP

How to use OpenMP to manage memory?

- N/P planes memory on the CPU
- Use only 3 planes of memory on GPU

Simple to process one plane at a time

```
1  do zp=1,mz
2      !$OMP TARGET ENTER DATA MAP(to:u(1:N,1:N,zp))
3
4      !$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
5       ....
6      !$OMP END TARGET TEAMS DISTRIBUTE
           PARALLEL DO
7
8      !$OMP TARGET EXIT DATA MAP(from:u(1:N,1:N,zp))
9  end do
```

3 planes on GPU using structure of pointers

```
1  type  ptr_struc
2      real (kind=4),  pointer  ::  ptr  (:,:)
3  end type  ptr_struc
4  type( ptr_struc )  ::  p_a(3)
5
6  p_a(1)%ptr => a(1:nx,1:ny,1)  ! similar for p_a(2:3)
7
8  !$OMP TARGET DATA MAP(to:p_a(:))
9  do zp=1,mz
10     ! copy in next plane
11     p_a(next)%ptr => a (:,:,  zp+1)
12     !$OMP TARGET ENTER DATA MAP(to:p_a(next)%ptr)
13
14     ! Compute on current plane
15
16     ! copy out previous plane
17     !$OMP TARGET EXIT DATA MAP(from:p_a(prev)%ptr)
18 end do
19 !$OMP END TARGET DATA
```