Achieving portability for a highly optimized GPU code for 3D Fourier Transforms at extreme problem sizes

Kiran Ravikumar[1], Oscar Hernandez[2], John Levesque[3], Stephen Nichols[2], P.K. Yeung[1]

kiran.r@gatech.edu

[1]Georgia Institute of Technology, [2]Oak Ridge National Laboratory, [3]Cray (HPE)

Performance, Portability, and Productivity in HPC Forum
September 1-2, 2020

Introduction & Motivation

Communication intensive codes on exascale heterogeneous machines?

- multidimensional Fourier transforms: fluid dynamics, signal processing, etc
- accelerators provide most of computing power, but need to move data
- communication still an issue (perhaps even more so): some new challenges

Batched asynchronous approach targeting extreme problem sizes

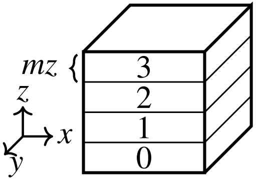
- problem size limited by smaller device memory compared to host
- process data in batches on GPU with entire data residing in CPU memory
- overlap operations on different batches of data, hide compute and transfer costs

CUDA Fortran (Ravikumar *e*t al. SC'19) on Summit (IBM+NVIDIA)

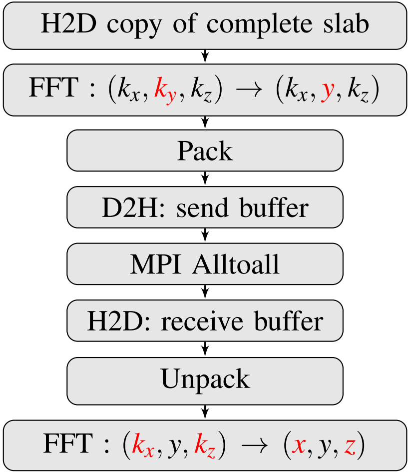
- portability: using OpenMP for offload, up to Version 5.0

Synchronous 3D FFT using GPUs

1D domain decomposition (Slabs)



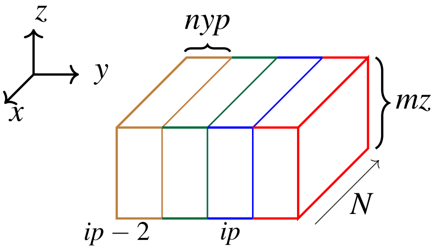
- Slabs: GPU parallelism instead of distributed memory, fewer MPI tasks in communication
- Copy entire slab from CPU (host (H)) to GPU (device (D)) and back to CPU at end
- 1D FFTs using cufft or rocfft library
- MPI Alltoall among all tasks to transpose $x-y$ to $x-z$ slabs



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

Batched approach

- 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

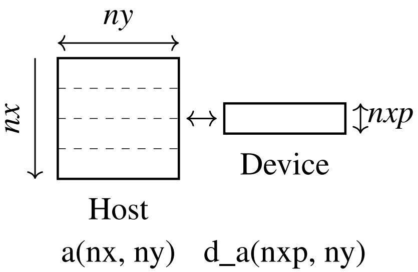
Non-contiguous maps and strided copies

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

```
cudaMemCpy2DAsync (dst, dpitch, src, spitch, &
    width, height, kind, stream)
```

– unique destination (*dst*) & source (*src*) buffers

– *width* (*nxp*) elements out of *spitch* (*nx*) to copy from first dimension, *height* (*ny*) such copies




```
allocate (a(nx, ny), abuf(nxp, ny))
! copy on host : abuf(:, :) = a(1:nxp, 1:ny)
!$OMP TARGET DATA MAP(tofrom: abuf)
! do some work
!$OMP END TARGET DATA MAP
! copy on host : a(1:nxp, 1:ny) = abuf(:, :)
```

How to do it in OpenMP?

- `MAP(to:a(1:nxp, 1:ny))`: not 5.0 compliant
  - — copy *a* to smaller *abuf* on host then MAP
  - — but this adds extra work on the host

Performance penalty due to additional operations on host

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

```
! src on host of shape (nx, ny)
! dst on device of shape (nxp, ny)

! copy src(1:nxp, 1:ny) to dst(1:nxp, 1:ny)

num_dims = 2
vol(1) = ny ; vol(2) = nxp
 dst_offset (1) = 0 ;  dst_offset (2) = 0
 src_offset (1) = 0 ;  src_offset (2) = 0
dst_dims(1) = ny ; dst_dims(2) = nxp
src_dims(1) = ny ; src_dims(2) = nx

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

Interoperability between OpenMP and non-blocking libraries

```
TARGET DATA MAP(tofrom:a)

TASK DEPEND(OUT:var)                      ! (A)
TARGET DATA USE_DEVICE_PTR(a)
FFTExecute (a, forward, stream)
FFTExecute (a, inverse , stream)
END TARGET DATA
END TASK

TARGET TEAMS DISTRIBUTE DEPEND(IN:var) NOWAIT
a(:, :, :) = a(:, :, :)/nx                ! (B)
END TARGET TEAMS DISTRIBUTE

TASKWAIT
END TARGET DATA
```

- Task A: Non-blocking FFT call
- Task B: OpenMP kernel with dependency on previous task
- Expect kernel runs after FFT completes
- But task A is considered complete after call to library, freeing dependency
- B executes before FFT completes
- Returns incorrect results

Task dependency not sufficient to enforce required synchronization

DETACH to enforce synchronization

```
TARGET DATA MAP(tofrom: a)

TASK DEPEND(out:var)  DETACH(event)

TARGET DATA USE_DEVICE_PTR(a)  Ⓐ
FFTExecute (a, forward, stream)
FFTExecute (a, inverse , stream)
END TARGET DATA

 hipStreamAddCallback (stream, ptr_callback, C_LOC(event), 0)
END TASK

! Copy or compute on other  data  Ⓒ

TARGET TEAMS DISTRIBUTE DEPEND(IN:var) NOWAIT
a (:,  :,  :)  = a (:,  :,  :) /nx                    Ⓑ
END TARGET TEAMS DISTRIBUTE

END TARGET DATA
```

```
subroutine  callback  (stream, status, event)
   type( c_ptr )  ::  event
   integer(kind=omp_event_handle_kind) ::  f_event
   call  C_F_POINTER (event, f_event)
   call  omp_fulfill_event( f_event )
end subroutine  callback
```

1. A: launch FFT, add call to *callback* in stream where FFT is running
2. B waits as dependent on A, C executes asynchronously
3. After FFT, function *callback* is called and event fulfilled
4. A completes allowing B to run

Support for `DETACH` not yet available

Porting asynchronous CUDA Fortran to OpenMP

```
do ip=1,np
    NEXT = mod(ip+1,3); CURR = mod(ip,3);
    PREV = mod(ip−1,3); COMM = mod(ip−2,3);
    cudaStreamWaitEvent (trans_stream, DtoH(NEXT), 0)
    cudaMemCpy2DAsync (abuf(NEXT),a(ip+1),trans_stream)
    cudaEventRecord (HtoD(NEXT),trans_stream)
    cudaStreamWaitEvent (comp_stream, HtoD(CURR), 0)
    FFTExecute  (abuf(CURR), comp_stream)
    cudaEventRecord (comp(CURR), comp_stream)
    cudaStreamWaitEvent (trans_stream, comp(PREV), 0)
    cudaMemCpy2DAsync (snd(ip-1), abuf(PREV), &
        trans_stream)
    cudaEventRecord (DtoH(PREV), trans_stream)
    cudaEventSynchronize (DtoH(COMM))
    MPI_IALLTOALL (snd(ip-2))
end do
```

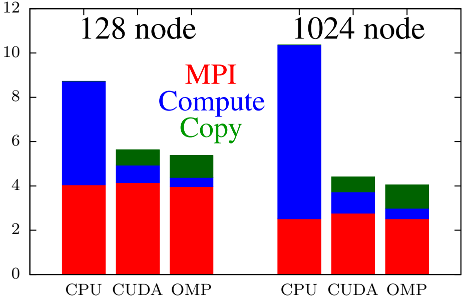
```
do ip=1,np
    NEXT = mod(ip+1,3); CURR = mod(ip,3);
    PREV = mod(ip−1,3); COMM = mod(ip−2,3);
    TASK DEPEND (IN:DtoH(NEXT), OUT:HtoD(NEXT))
    omp_target_memcpy_rect (abuf(NEXT), a(ip+1))

    TASK DEPEND (IN:HtoD(CURR), OUT:comp(CURR))
        DETACH(event)
    FFTExecute  (abuf(CURR), comp_stream)
    TASK DEPEND (IN:comp(PREV), OUT:DtoH(PREV))
    omp_target_memcpy_rect (snd(ip−1), abuf(PREV))


    TASK DEPEND(IN:DtoH(COMM))
    MPI_IALLTOALL (snd(ip-2))
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

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	Time (s) 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

Summary and Future Work

- Progress made on OpenMP offload implementation of 3D Fast Fourier Transforms
  - Porting from a successful CUDA Fortran code (SC'19) on Summit at OLCF
  - At present: batched, synchronous, which enable large problem sizes
- Some challenges of portability overcome, some pending full OMP 5.0 availability:
  - Strided copy b/w smaller 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
  - Portable GPU parallelism for communication-intensive applications

Comment: Need better Fortran support (would like to see OMP 5.0 routines that are only C-callable also callable from Fortran, with good examples)