Pipelining Fast Fourier Transform on the OpenPOWER System

Jun Doi
IBM Research – Tokyo
doichan@jp.ibm.com

7 Mar. 2018

SIAM Conference on Parallel Processing for Scientific Computing 2018
MS2 : State-of-the-Art FFT – Algorithms, Implementations and Applications

Background

- Parallelization of FFT on node with GPUs
  - FFT performance is bounded by the All-to-All bandwidth
  - Faster connection between CPUs and GPUs is demanding
- How to program using multiple GPUs
  - Data management and transfer make it complex and difficult
  - Unified Memory make the code productive
  - NVLink, fast interconnect will help data migration between CPUs and GPUs
  - Productivity + performance

POWER System: NVLink and GPUs

CORAL Supercomputers

OAK RIDGE National Laboratory

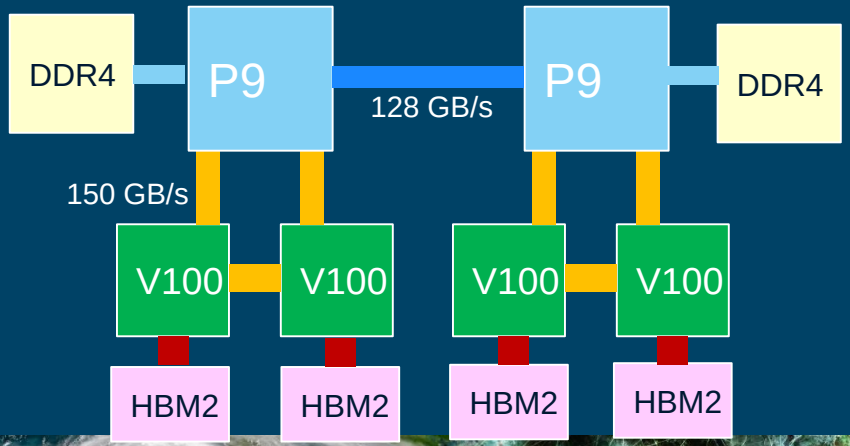
summit

6 GPUs

SIERRA

4 GPUs

4 GPU Newell configuration



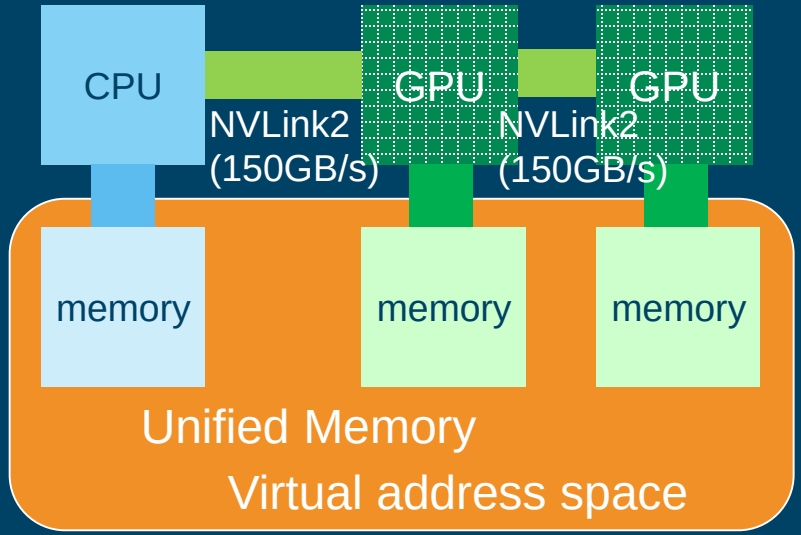
IBM Power System AC922 (Newell)

Our test environment

CPU	IBM POWER9
CPUs per node	2 sockets
CPU memory	DDR4 1024 GB
GPU	NVIDIA Tesla V100
GPUs per node	4
GPU memory	HBM2 16 GB
Inter connection	**NVLink2** (150 GB/s bi-direction)
MPI	IBM Spectrum MPI 10.1.0
CUDA toolkit	9.1
OS	RHEL 7.4

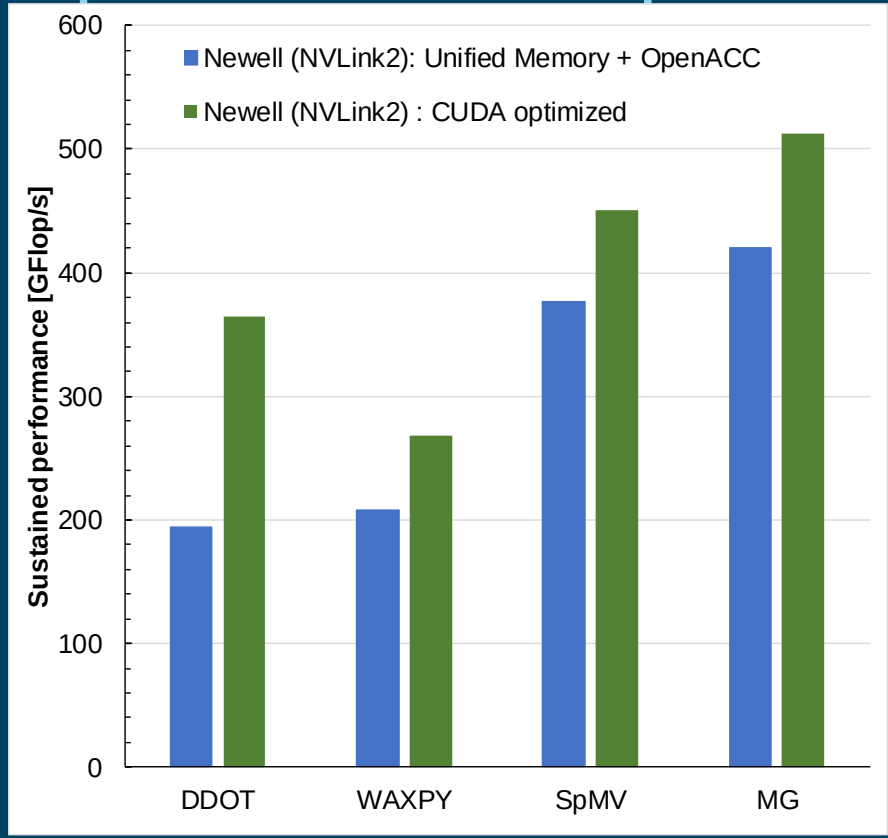
Using Unified Memory to Multi-GPU Programming

- Unified Memory
  - Unified virtual address space across CPU and GPUs
  - Data transfer is automatically done by CUDA runtime
  - Programmers are free from data management and movement
- NVLink + Unified Memory
  - Fast data migration via NVLink
  - Performance + Productivity



HPCG: Unified Memory + OpenACC Example

- OpenACC
  - Directive based programming to offload workload to GPUs
  - With Unified Memory no data transfer directives are needed
- HPCG Benchmark
  - Solves linear equation of sparse problem with CG method
  - Added 25 lines of OpenACC directives to local SMP codes, achieved 80% of CUDA optimized implementation



3D-FFT with Multiple GPUs

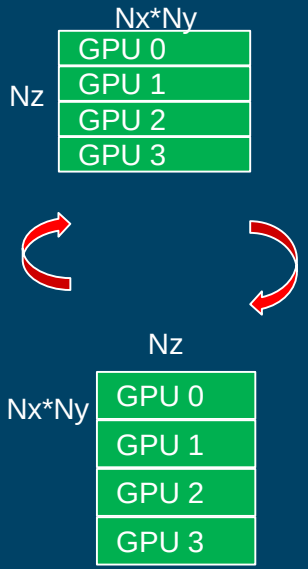
3D-FFT with 4 GPUs

2D-FFT for X and Y

Transpose : (Nx, Ny, Nz/4) to (Nz, Nx, Ny/4)

1D-FFT for Z

Transpose : (Nz, Nx, Ny/4) to (Nx, Ny, Nz/4)



3D array is divided in Z-axis

3D array is now divided in XY-plane

This is optional, most application calculates each array element independently

3D-FFT with cuFFTXt

cuFFTXt the easiest way for multi-GPU FFT

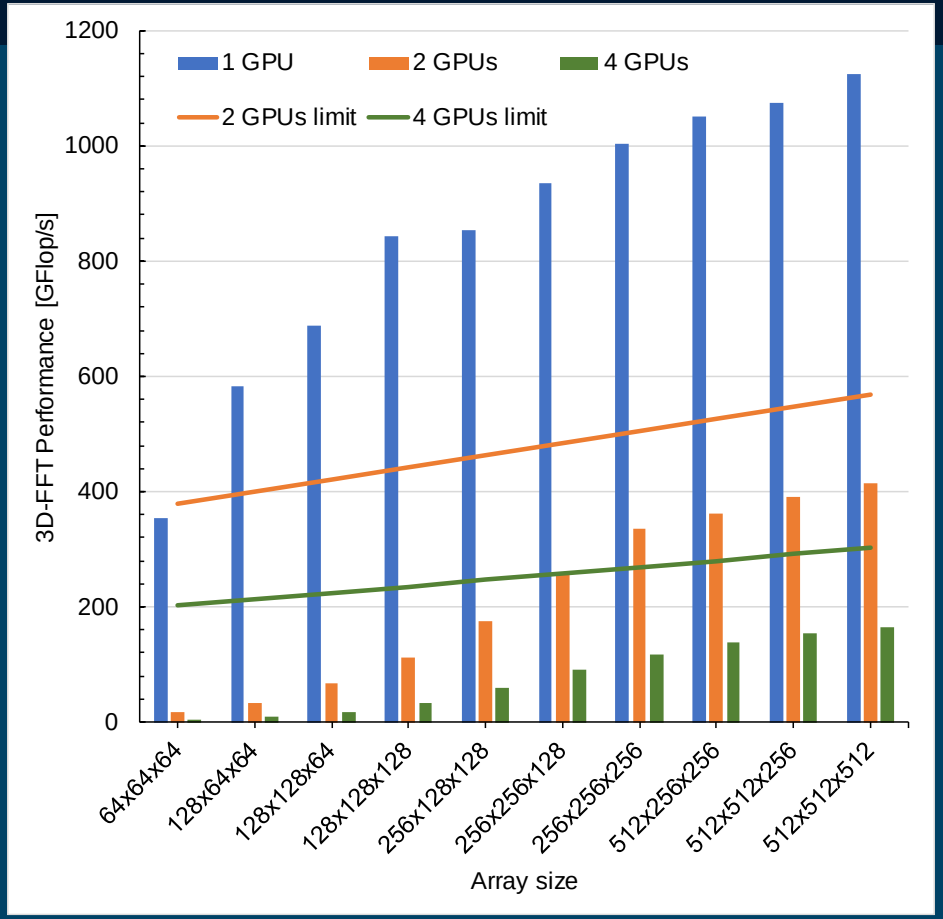
Included in the CUDA toolkit

```
cufftCreate(&hFFT);
cufftXtSetGPUs(hFFT,4,gpus);    //gpus[] = {0,1,2,3}
cufftMakePlan3d(hFFT,nx,ny,nz,CUFFT_Z2Z,worksize);
cufftXtMalloc(hFFT,&pBuf0,CUFFT_XT_FORMAT_INPLACE);
cufftXtMalloc(hFFT,&pBuf1,CUFFT_XT_FORMAT_INPLACE);

cufftXtMemcpy(hFFT,pBuf0,pIn,CUFFT_COPY_HOST_TO_DEVICE);
cufftXtExecDescriptorZ2Z(hFFT,pBuf0,pBuf0, CUFFT_FORWARD);
cufftXtMemcpy(hFFT,pOut,pBuf0,CUFFT_COPY_DEVICE_TO_HOST);
```

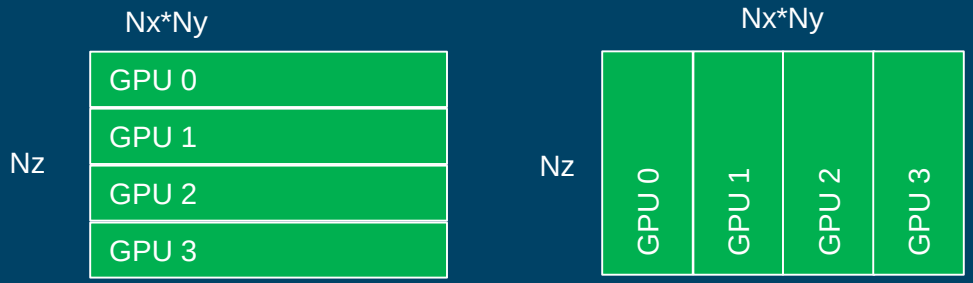
Data is loaded from/ stored to host memory
Programmer do not have to manage data,
however it is not easy to use transformed data on GPU

cuFFTXt transposes twice



3D-FFT measured performance of cuFFTXt on Newell and performance limit calculated by 2*transpose time

3D-FFT by Unified Memory: Naïve Implementation



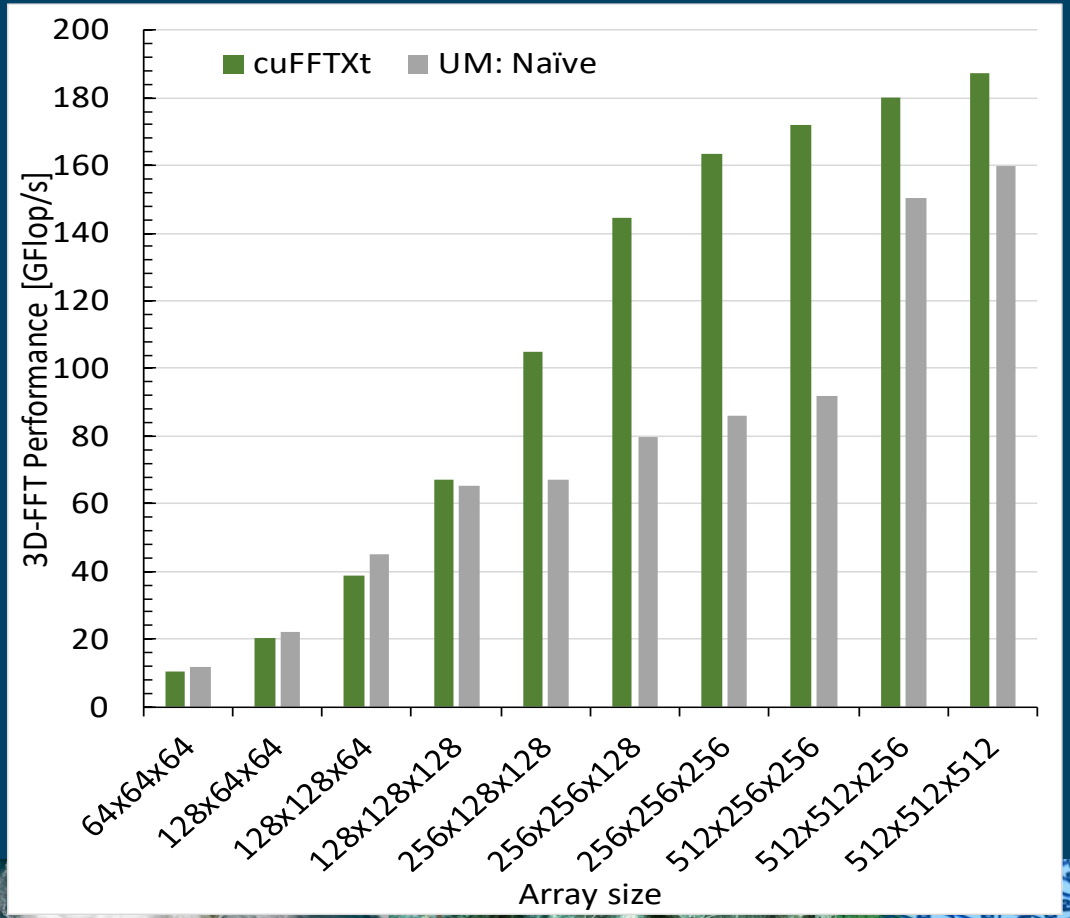
No explicit transpose,
but implicit data transfers between GPUs
are done by CUDA runtime

```
cufftMakePlanMany(fftXY,2,nr,NULL,1,nx*ny,NULL,1,nx*ny,CUFFT_Z2Z,nz/4,workSize);    //nr[]={nx,ny}
cufftMakePlanMany(fftZ,1,nr,NULL,nx*ny,1,NULL,nx*ny,1,CUFFT_Z2Z,nx*ny/4,workSize); //nr[]={nz}

#pragma omp parallel num_threads(4)
{
  int tid = omp_get_thread_num();
  cudaSetDevice(tid);
  cufftExecZ2Z(fftXY,A + nx*ny*nz*tid/4,A + nx*ny*nz*tid/4,CUFFT_FORWARD);
#pragma omp barrier
  cufftExecZ2Z(fftZ,B + nx*ny*tid/4,A + nx*ny*tid/4,CUFFT_FORWARD);
}
```

No data management required, data can be anywhere, programmer can directly use transformed data on GPUs

3D-FFT Comparison, cuFFTXt and Unified Memory Naïve

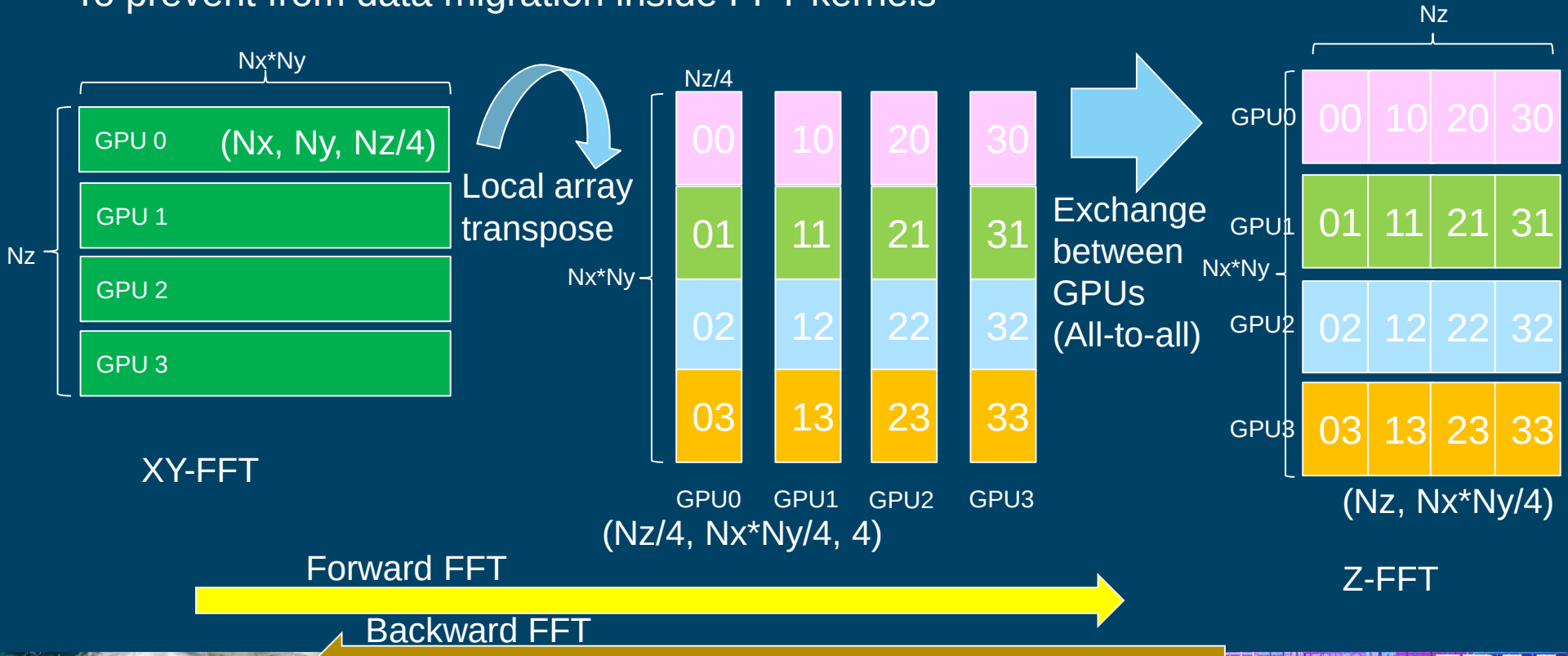


Using 4GPUs on Newell

FFT kernels are very slow if data is not on that GPU
Because of overheads and latencies of implicit data copy (data migration)

GPU Memory Blocking

To prevent from data migration inside FFT kernels

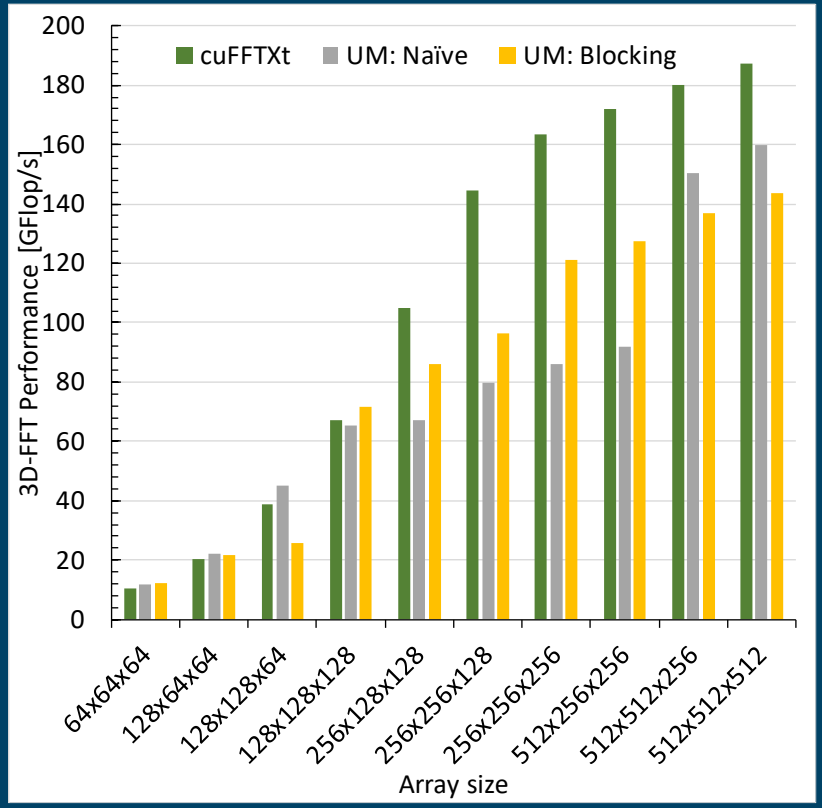


3D-FFT with Memory Blocking

```
cufftMakePlanMany(fftXY,2,nr,NULL,1,nx*ny,NULL,1,nx*ny,CUFFT_Z2Z,n
z/4,workSize);    //nr[]={nx,ny}
cufftMakePlan1d(fftZ,nz,CUFFT_Z2Z,nx*ny,&workSize);  //non-stride FFT

#pragma omp parallel num_threads(4)
{
  int tid = omp_get_thread_num();
  int offset = nx*ny*nz*tid/4;
  cudaSetDevice(tid);
  cufftExecZ2Z(fftXY,A + offset,A + offset,CUFFT_FORWARD);
  LocalTransposeKernel(B + offset,A + offset);
#pragma omp barrier
  AlltoAllKernel(B + offset,A + offset);
  cufftExecZ2Z(fftZ,B + offset,B + offset,CUFFT_FORWARD);
}
```

AlltoallKernel is not implemented by cudaMemcpy
but implemented as CUDA kernel (it's faster than cudaMemcpy)



3D-FFT with Prefetch

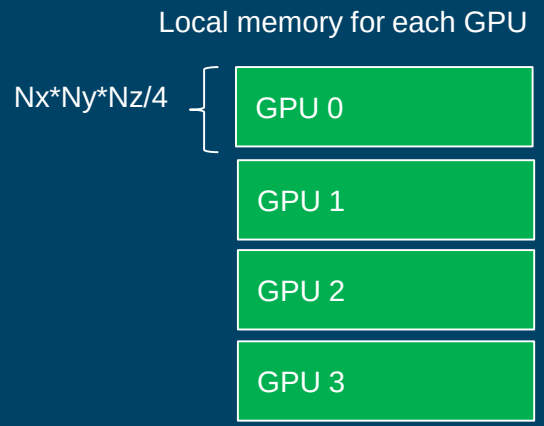
Using helper functions for Unified Memory

cudaMemPrefetchAsync: Explicitly copies data to a device before use

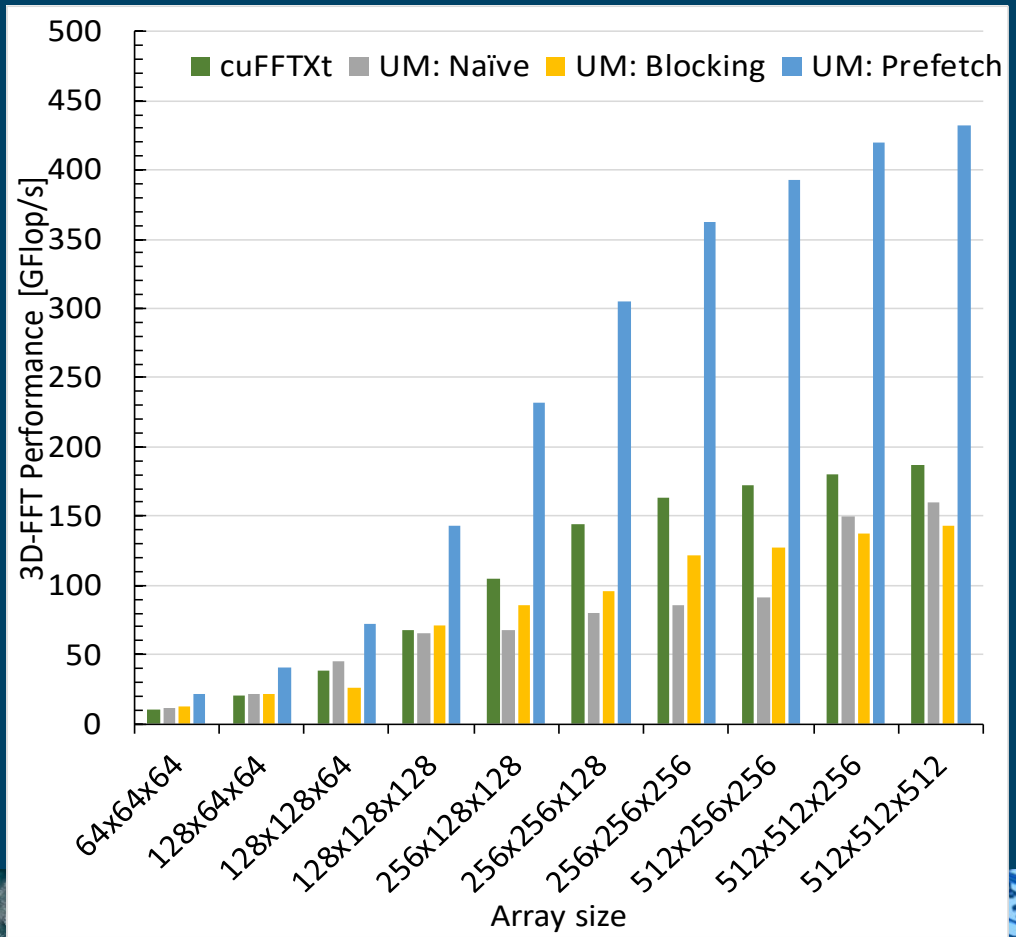
cudaMemAdvise: can tell CUDA runtime where data region will be located

```
cufftMakePlanMany(fftXY,2,nr,NULL,1,nx*ny,NULL,1,nx*ny,CUFFT_Z2Z,nz/4,workSize);    //nr[]={nx,ny}
cufftMakePlan1d(fftZ,nz,CUFFT_Z2Z,nx*ny,&workSize); //non-stride FFT
#pragma omp parallel num_threads(4)
{
int tid = omp_get_thread_num();
int size = nx*ny*nz;
int offset = size*tid/4;
cudaSetDevice(tid);
cudaMemAdvise(A+offset,sizeof(double2)*size/4,cudaMemAdviseSetPreferredLocation,tid);
cudaMemAdvise(B+offset,sizeof(double2)*size/4,cudaMemAdviseSetPreferredLocation,tid);
cudaMemPrefetchAsync(A+offset,sizeof(double2)*size/4,tid,strm);
cudaMemPrefetchAsync(B+offset,sizeof(double2)*size/4,tid,strm);

cufftExecZ2Z(fftXY,A + offset,A + offset,CUFFT_FORWARD);
LocalTransposeKernel(B + offset,A + offset);
#pragma omp barrier
AlltoAllKernel(B + offset,A + offset);
cufftExecZ2Z(fftZ,B + offset,B + offset,CUFFT_FORWARD);
}
```



3D-FFT with Prefetch



Using 4GPUs on Newell

Prefetch version achieves double of the performance of cuFFTXt becuase we only exploit 1 transpose

3D-FFT with Pipelined All-to-All

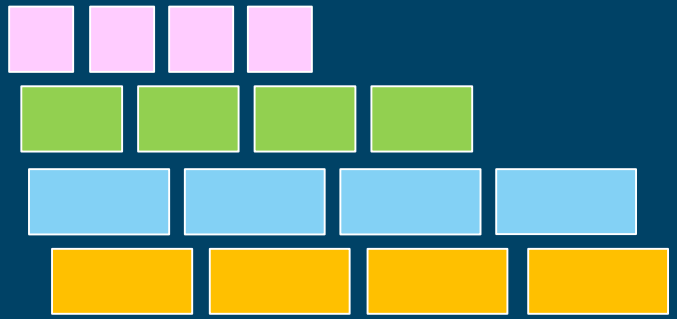
Non-overlap All-to-All

GPU 0: from GPU0 | from GPU1 | from GPU2 | from GPU3

CUDA kernels are not overlapped since number of threads is too large

Pipelined All-to-All

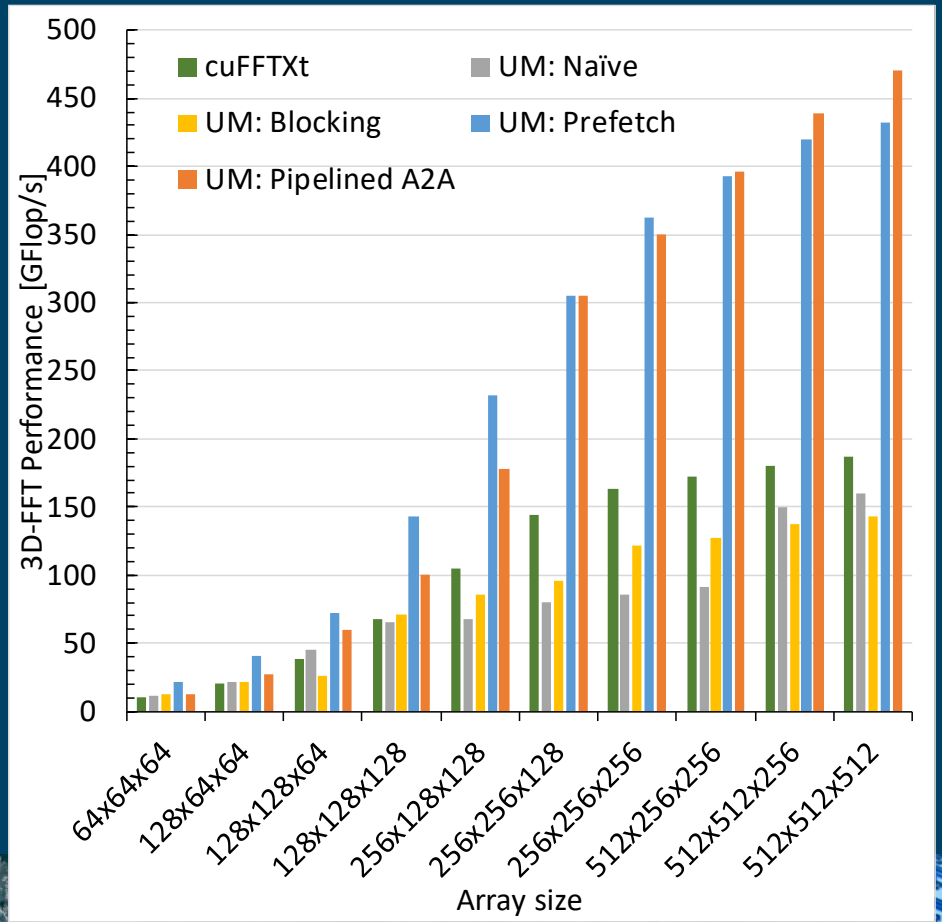
GPU 0:



Dividing each array into smaller arrays to make pipeline

CUDA kernels can be overlapped for smaller number of threads

3D-FFT with Pipelined All-to-All

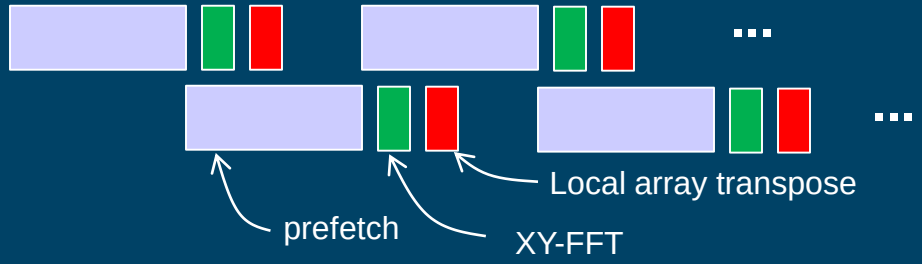


Using 4GPUs on Newell

Calculation time is much smaller than all-to-all time, overlapped time by pipelining is very small

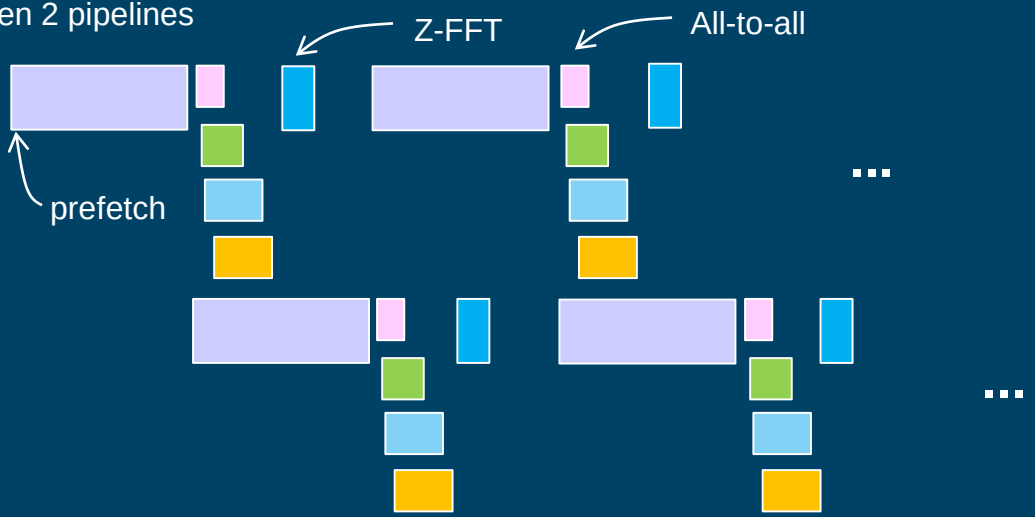
3D-FFT with Full Pipelining

Pipeline for XY-FFT

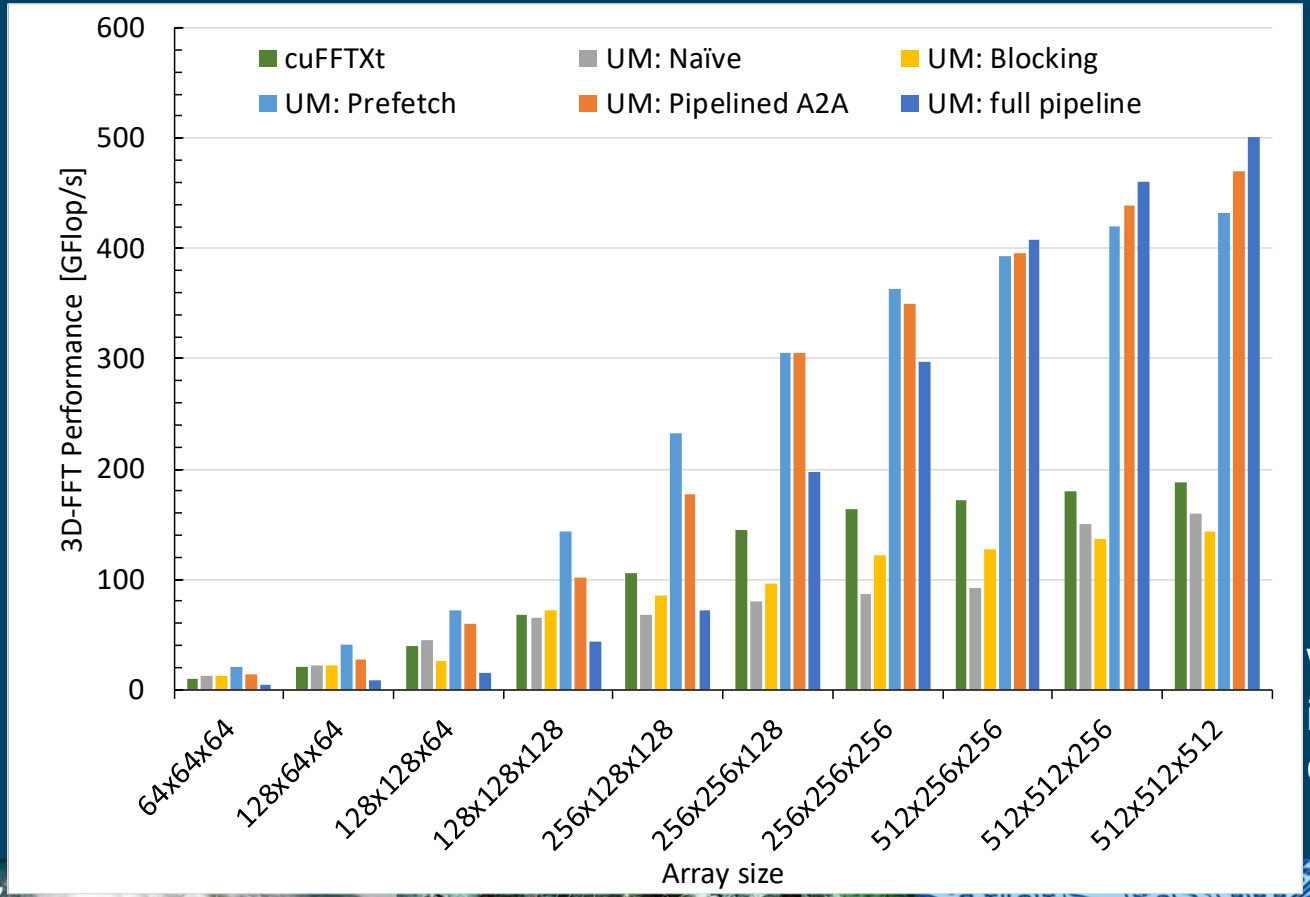


Synchronization between 2 pipelines

Pipeline for Z-FFT



3D-FFT on Newell with 4 Tesla V100

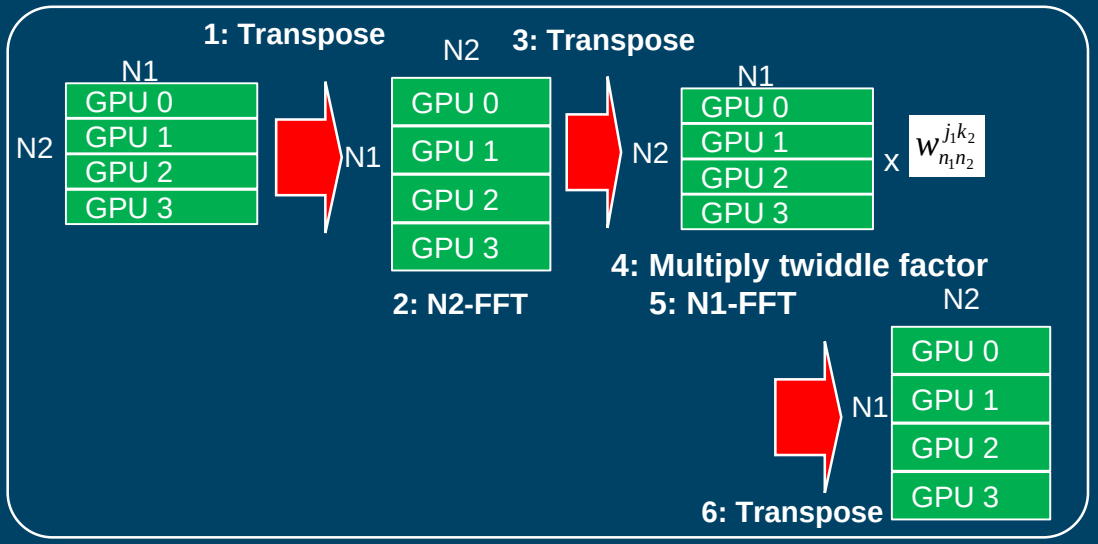


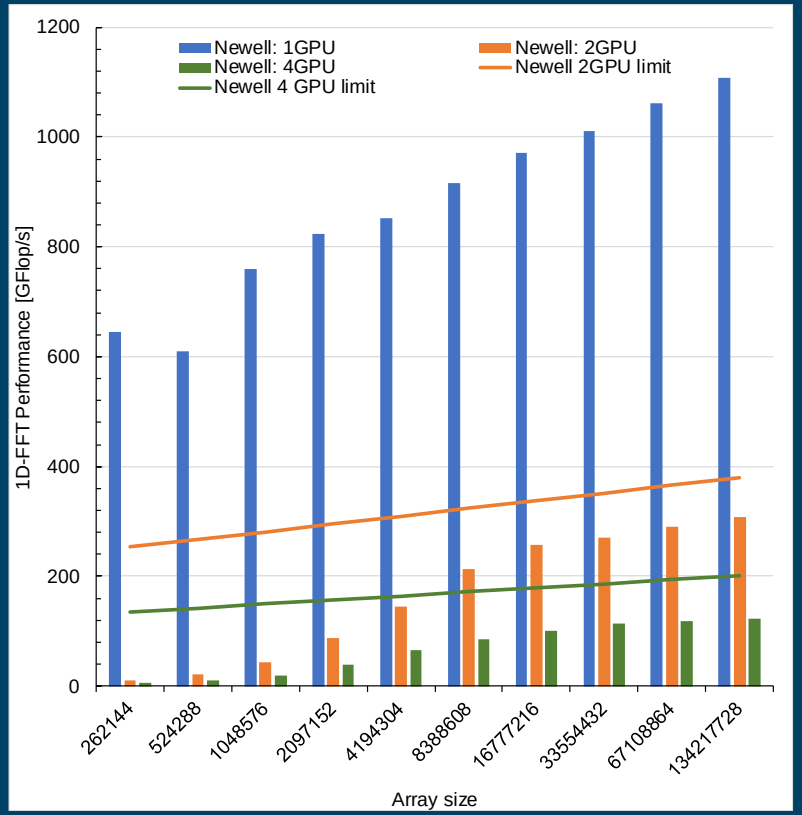
We have to choose the best implementation depending on the array size

1D-FFT with Multiple GPUs

1D DFT: $y_k = \sum_{j=0}^{n-1} x_j w_n^{jk}$ can be rewritten as 2D array

$$y(k_2,k_1) = \sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1} x(j_1,j_2) w_{n_2}^{j_2 k_2} w_{n_1 n_2}^{j_1 k_2} w_{n_1}^{j_1 k_1}$$

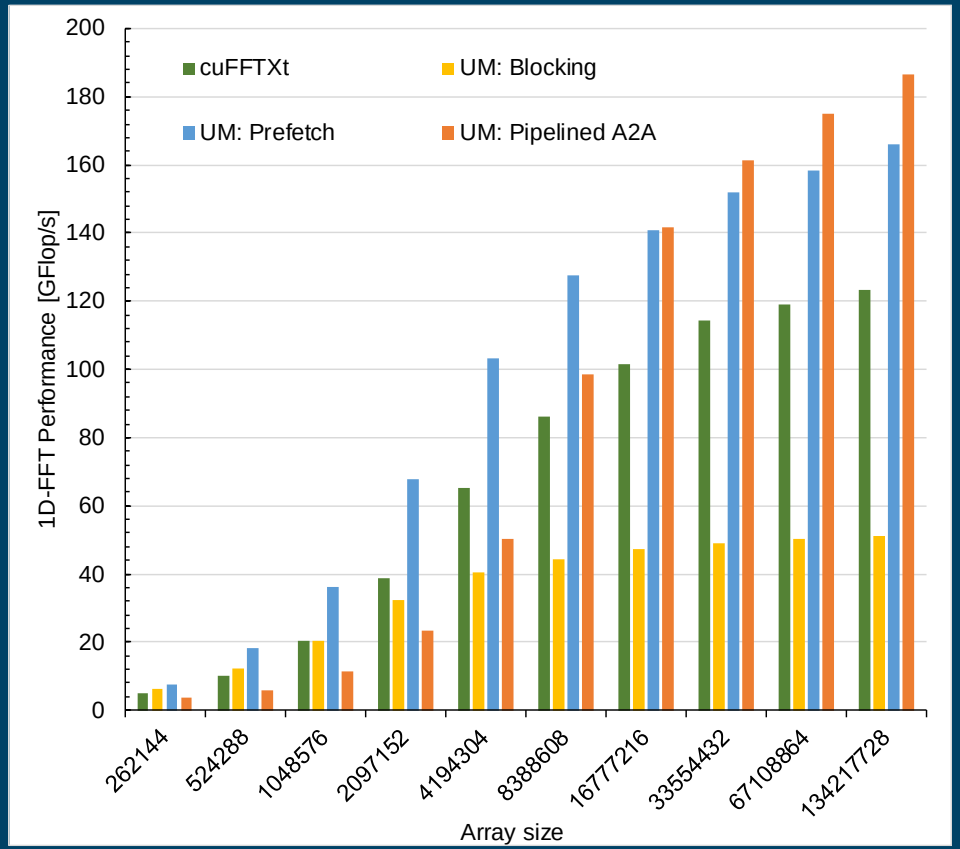




1D-FFT measured performance of cuFFTXt on Newell and performance limit calculated by 3*transpose time

1D-FFT 4 GPUs with Unified Memory

- Similar to 3D-FFT
  - Memory blocking
  - Prefetching
  - Pipelined All-to-all
  - (Full pipeline)
- 3 all-to-alls
  - Can be decreased to 2 all-to-all (not yet)



Summary

- FFT on Multiple GPUs with Unified Memory
  - Easy programming, good performance
  - With prefetch, large performance improvement
  - Pipelining, adds more performance (less productivity)
- Future work
  - Evaluation with real applications
  - Multiple nodes, degrades by lower all-to-all bandwidth