Evaluation of Directive-based GPU Programming Models on a Block Eigensolver with Consideration of Large Sparse Matrices

Fazlay Rabbi[1], Christopher S. Daley[2], Hasan Metin Aktulga[1] and Nicholas J. Wright[2]

[1]Michigan State University, East Lansing MI, USA

[2]Lawrence Berkeley National Laboratory, Berkeley CA, USA

{*rabbimd, hma*}@msu.edu, {*csdaley, njwright*}@lbl.gov

WACCPD @ SC 2019 Denver, CO

Introduction

- Achieving high performance and performance portability on heterogeneous systems is challenging.

- Test Application: *Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG)*
  - *Popular and commonly used, preconditioning can be done!*
  - *Pretty complex and challenging one!*

- Baseline version: OpenMP/OpenACC version for CPUs using LAPACK and BLAS routines.

- Can we port it to GPU using directive based programming model and achieve desired performance?

Motivation

- All DOE's future planned systems will be equipped with GPUs:
  - NERSC - Perlmutter (AMD CPU + NVIDIA GPU)
  - ALCF – Aurora (Intel CPU + Intel XE Accelerator)
  - OLCF – Frontier (AMD CPU+ AMD GPU)
- Portability
  - OpenMP and OpenACC provide pragmas to offload computations to device (i.e. GPUs)
- Efficient use of accelerators is desirable to exploit the full capabilities of these future DOE ASCR systems .

LOBPCG Pseudocode

Algorithm 1: Locally Optimal Block Preconditioned Conjugate Gradient eigensolver.

```
Input: A (sparse matrix), X (dense matrix), neig (# of eigen values),
       num_simulations
Output: lambda (eigen value)
1  for i = 1 to num_simulations do
2      R = X * lambda
3      Calculate residualNorm and activeCols
4      if residualNorm > Tolerence then
5          break
6      end
7      temp = X^T * R(: activeCols)
8      tempR = X * temp
9      R(: activeCols) = tempR − R(: activeCols)
10     gramRBR = R^T(: activeCols) * R(: activeCols)
11     R(: activeCols) = R(: activeCols) * cholesky(gramRBR)
12     actAR = SpMM(A, actR)
13     gramPBP = P^T(: activeCols) * P(: activeCols)
14     gramPBP = inverse(cholesky(gramPBP))
15     P(: activeCols) = P(: activeCols) * gramPBP
16     AP(: activeCols) = AP(: activeCols) * gramPBP
17     gramXAR = X^T * R(: activeCols)
18     gramRAR = actAR^T * R(: activeCols)
19     gramXAP = AX^T * P(: activeCols)
20     gramRAP = actAR^T * P(: activeCols)
21     gramPAP = AP^T(: activeCols) * P(: activeCols)
22     gramXBP = X^T * P(: activeCols)
23     gramRBP = R^T(: activeCols) * P(: activeCols)
24     [GA, GB] = constructMat()
25     [lambda, coordX] = EIGEN(GA, GB)
26     P(: activeCols) = R(: activeCols) * coordX
27     P = P(: activeCols) * coordX
28     AP(: activeCols) = actAR * coordX
29     AP = AP(: activeCols) * coordX
30     newX = X * coordX
31     X = newX + P
32     newAX = AX * coordX
33     AX = newAX + AP
34 end
```

SPMM operation – ~**56%** Total Exe. Time

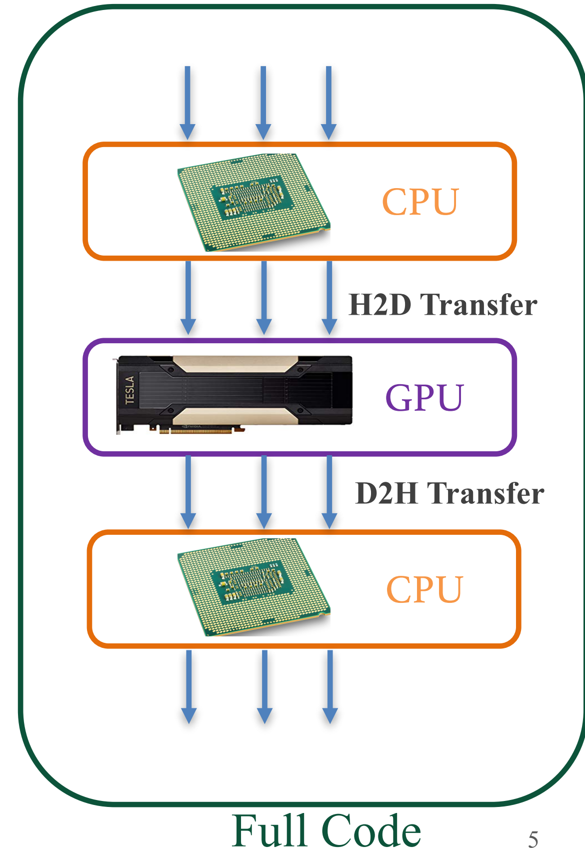
XTY operation - ~**31%** Total Exe. Time

XY operation - ~**6%** Total Exe. Time

Application Kernels - ~**8%** Total Exe. Time

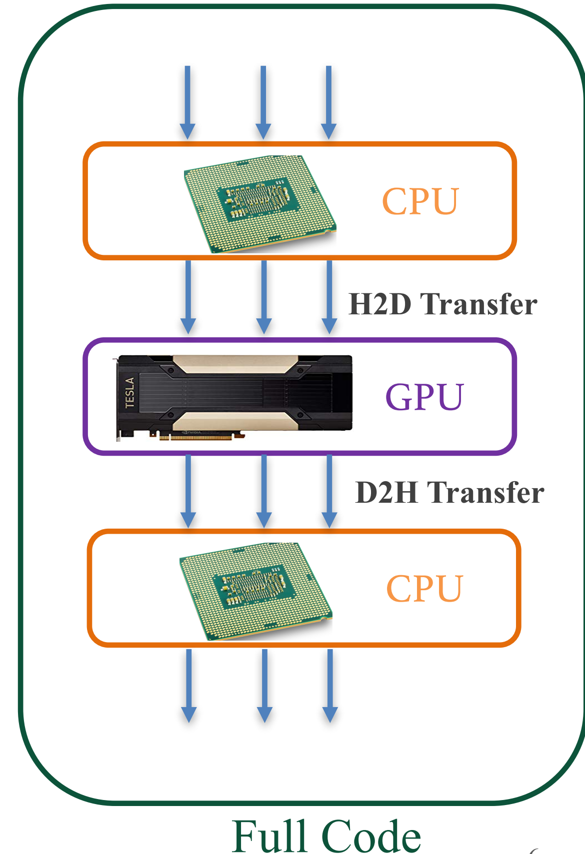
LOBPCG Porting Strategy: Initial Attempt

- Use optimized CUDA library routines for the most expensive kernels.
  - cblas_dgemm → cublasDgemm
  - SpMM → cusparseDcsrmm
- Then started porting other application kernels using directives.
- Creating target data region to copy necessary data to/from GPU
  - #pragma omp target data map(*to: list*) map(*from: list*) map(*tofrom: list*)
  - #pragma acc data copy(..) copyin(..) copyout(..)
- Running slower!!!
  - 1GPU+1CPU was 0.92x slower compared to 1CPU.



LOBPCG Porting Strategy: Device Pointer

- *nvprof* showing huge data movement between CPU and GPU.
  - ~97% of total execution time are spent on data movement
- Need to minimize data movement between CPU & GPU.
- Two useful clauses that allow OpenMP/OpenACC kernels to access data that is already allocated on GPU:
  - *is_device_ptr(list)* → OpenMP
  - *deviceptr(list)* → OpenACC



LOBPCG Porting Strategy: Impact of Device Pointer

Operation:

$R = X * lambda$

$newX = X .* R$

Before using `is_device_ptr`

- cuBLAS call
- D2H transfer
- H2D & D2H transfer
- Launching App. Kernel
- Definition of mat_mult()

```
// d_R and other device arrays allocated with omp_target_alloc
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, b, numrows, b,
  &cudaAlpha, d_lambda, b, d_X, b, &cudaBeta, d_R, b);

// Copy output array d_R to the host array R
omp_target_memcpy(R, d_R, R_size * sizeof(double), 0, 0, h, t);

// Copy host array R to the device in OpenMP target data region
#pragma omp target data map(tofrom: newX[0 : X_size])\
  map(to: X[0 : X_size], R[0 : R_size])
{
  mat_mult(X, R, newX, numrows, b);
}

void mat_mult(double *src1, double *src2, double *dst,
              int row, int col)
{
#pragma omp target teams distribute parallel for collapse(2)
  for(int i = 0; i < row ; i++)
    for(int j = 0 ; j < col ; j++)
      dst[i * col + j] = src1[i * col + j] * src2[i * col + j];
}
```

After using `is_device_ptr`

- cuBLAS call
- Launching App. Kernel
- Definition of mat_mult()

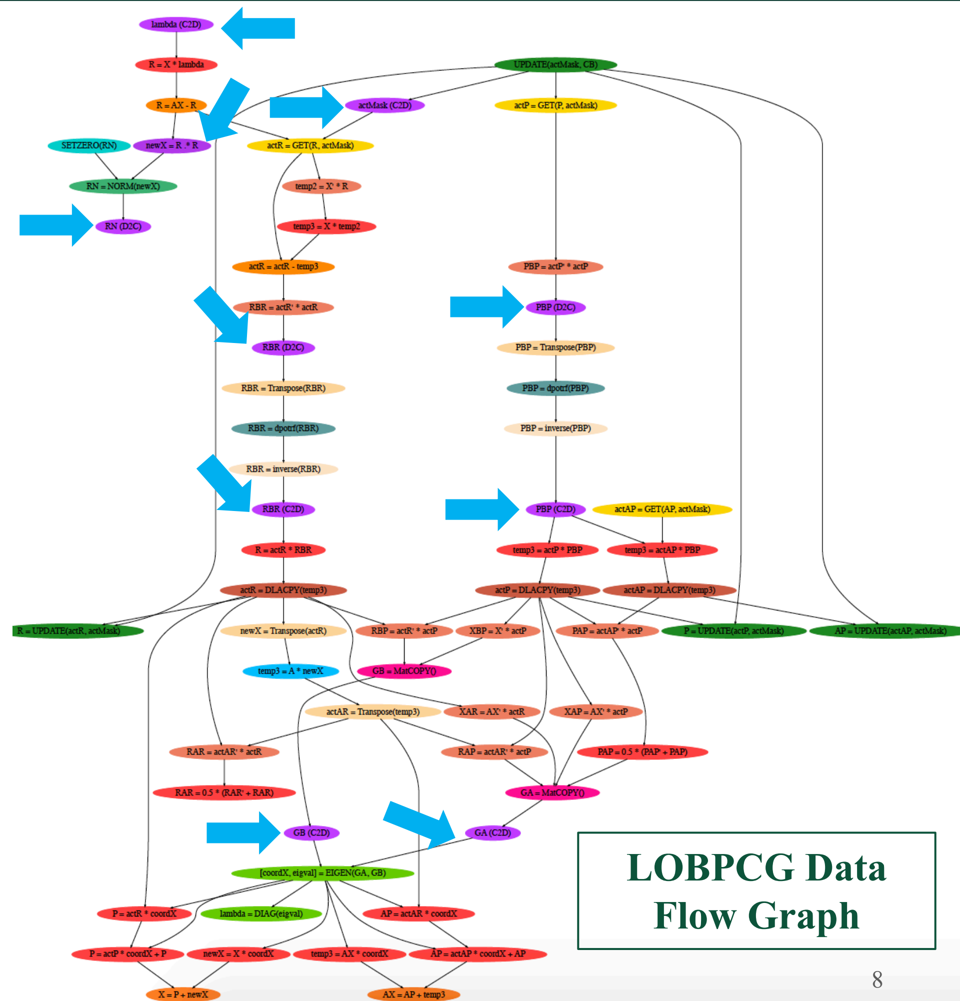
```
// d_R and other device arrays allocated with omp_target_alloc
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, b, numrows, b,
  &cudaAlpha, d_lambda, b, d_X, b, &cudaBeta, d_R, b);

// Pointers to device arrays passed into mat_mult function
mat_mult(d_X, d_R, d_newX, numrows, b);

void mat_mult(double *src1, double *src2, double *dst,
              int row, int col)
{
  // Use is_device_ptr because data is already on the device
#pragma omp target is_device_ptr(src1, src2, dst)
#pragma omp teams distribute parallel for collapse(2)
  for(int i = 0; i < row ; i++)
    for(int j = 0 ; j < col ; j++)
      dst[i * col + j] = src1[i * col + j] * src2[i * col + j];
}
```

LOBPCG Porting Strategy: Non Portable Routines

- We couldn't run completely on GPU because there are host only LAPACK routines:
  - LAPACKE_dpotrf()
  - LAPACKE_dsygv()
  - LAPACKE_dgetrf()
  - LAPACKE_dgetri()
- 10 small matrices are moved between CPU & GPU in every iteration.



Test matrices

- Square Matrix
- Different domains and different sparsity patterns

Matrix	Dimensions	% of Non zeros	Size (GB)	Domain
Queen_4147	4.1M	0.0010	2.02	3D Structural Problem
HV15R	2.0M	0.0070	3.41	Computational Fluid Dynamics
Nm7	5.0M	0.0026	7.79	Nuclear Physics
Nm8	7.6M	0.0010	7.14	Nuclear Physics

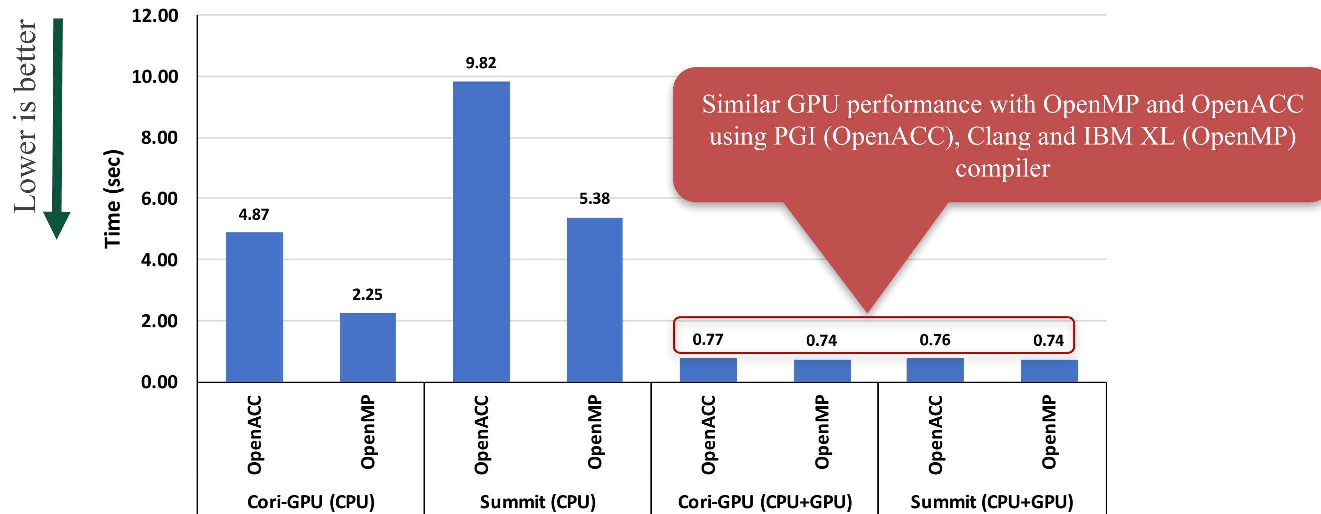
Evaluation Platform

- Cori-GPU and Summit
  - Equipped with NVIDIA Volta V100 GPU

	Cori-GPU	Summit
Processor	Intel Skylake	IBM Power9
CPUs : GPUs	2:8	2:6
CPU-GPU Interconnect	PCIe 3.0, Peak BW 16 GB/s	NVLink2, Peak BW 50 GB/s

- CPU vs GPU Test Configuration:
  - CPU : One Socket, One thread per core.
  - GPU: One Socket (One thread per core) + One GPU.

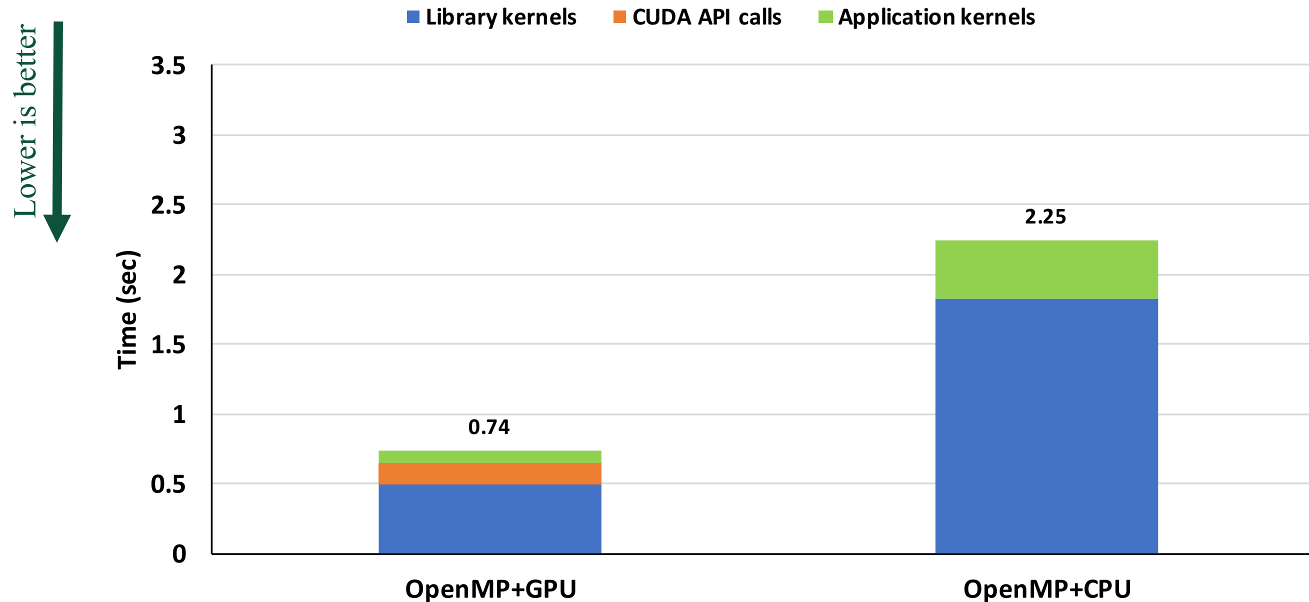
Evaluation – LOBPCG



LOBPCG performance on Cori-GPU and Summit for Nm7 Matrix

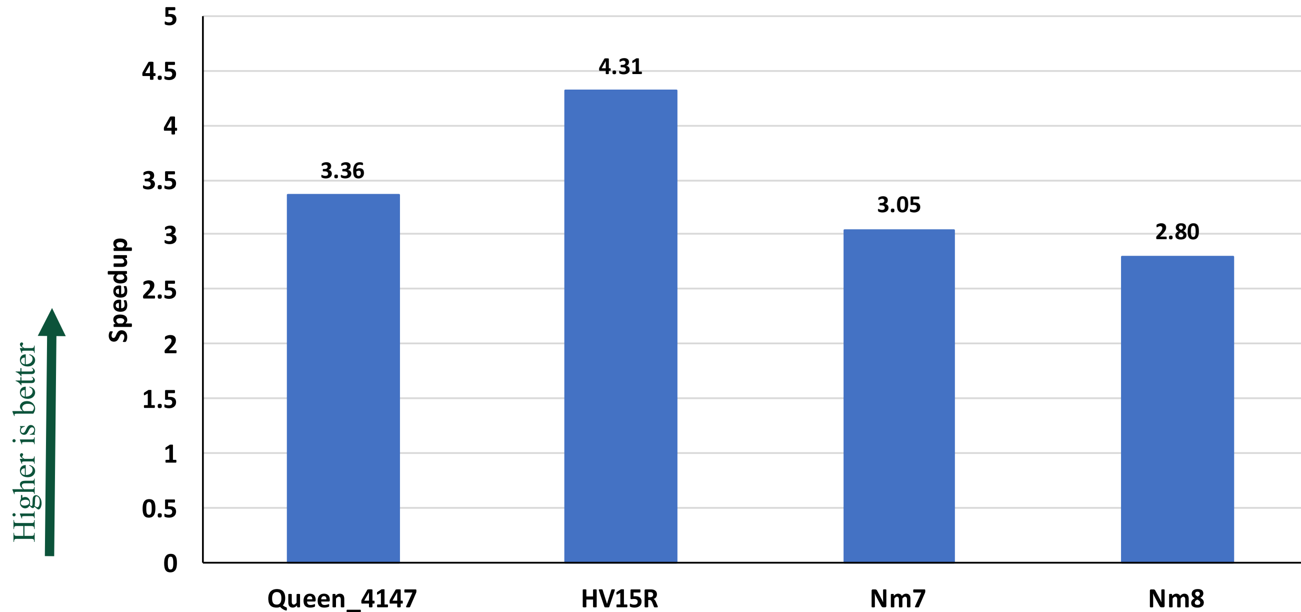
- Variable CPU performance:
  - Custom CSR based SpMM kernel runs **1.5-3X slower** in OpenACC CPU version compared to OpenMP CPU version

Evaluation – CPU vs GPU performance



LOBPCG OpenMP CPU vs OpenMP GPU execution time breakdown on Cori-GPU for Nm7 matrix

Evaluation – Putting All Matrices Together



LOBPCG GPU vs CPU speedup on Cori-GPU

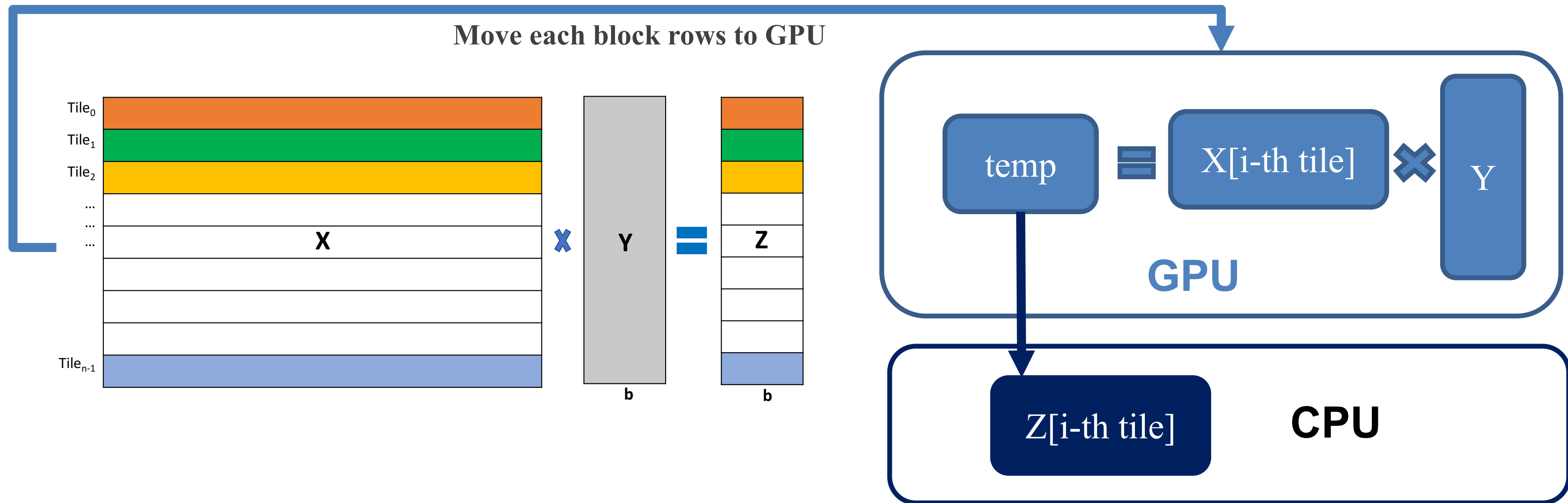
What is next?

- What if the sparse or any other matrix doesn't fit in GPU memory (say sparse matrix > 16GB)?
  - This is a common issue for large-scale scientific computing/data science applications.
- Is it useful to use GPU for large matrices?
- Possible solutions:
  - Matrix tiling
  - Unified memory – Original code with no tiling.

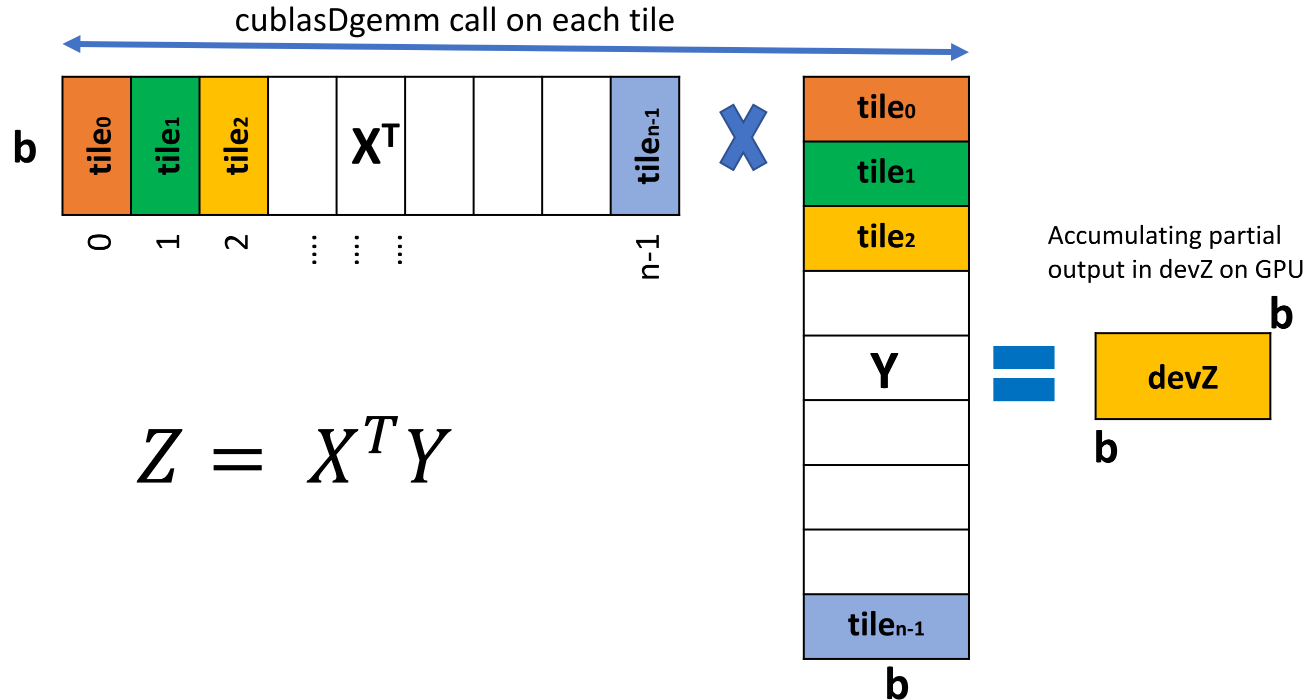
Tiling Matrices

- Utilizing GPU memory properly
- Tiling matrices for each operation
  - Overlapping data movement with computation
- We tested with two dominant kernels in LOBPCG:
  - *cusparseDcsrmm* (SpMM operartion)
  - *cublasDgemm* (Inner product operation)

Tiling SPMM (*cusparseDcsrmm*) Kernel

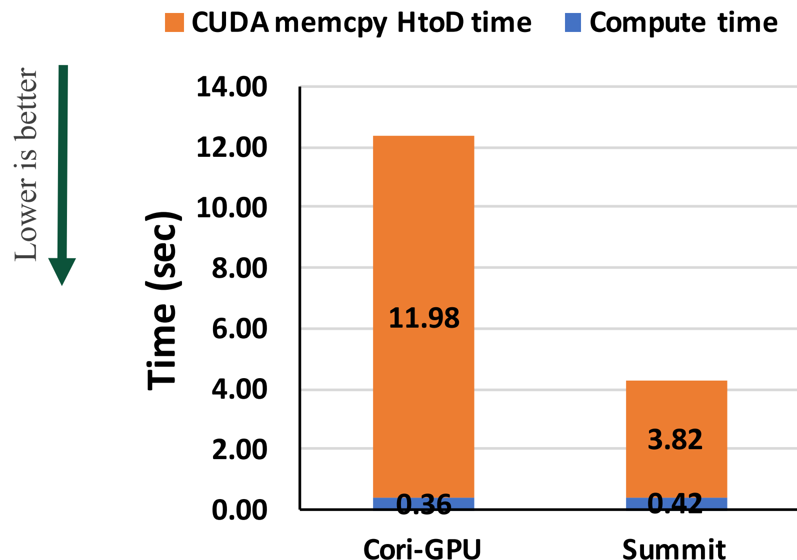


Tiling Inner Product (*cublasDgemm*) Kernel



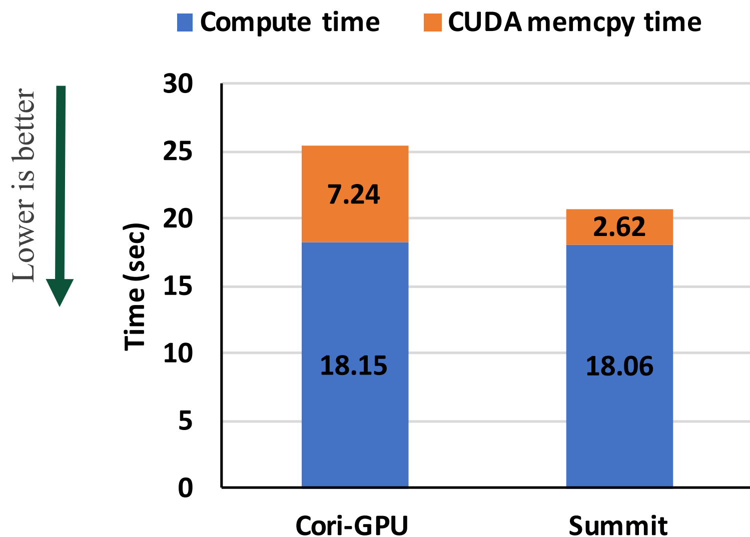
$$Z = X^T Y$$

Evaluation - Inner Product (*cublasDgemm*) Kernel



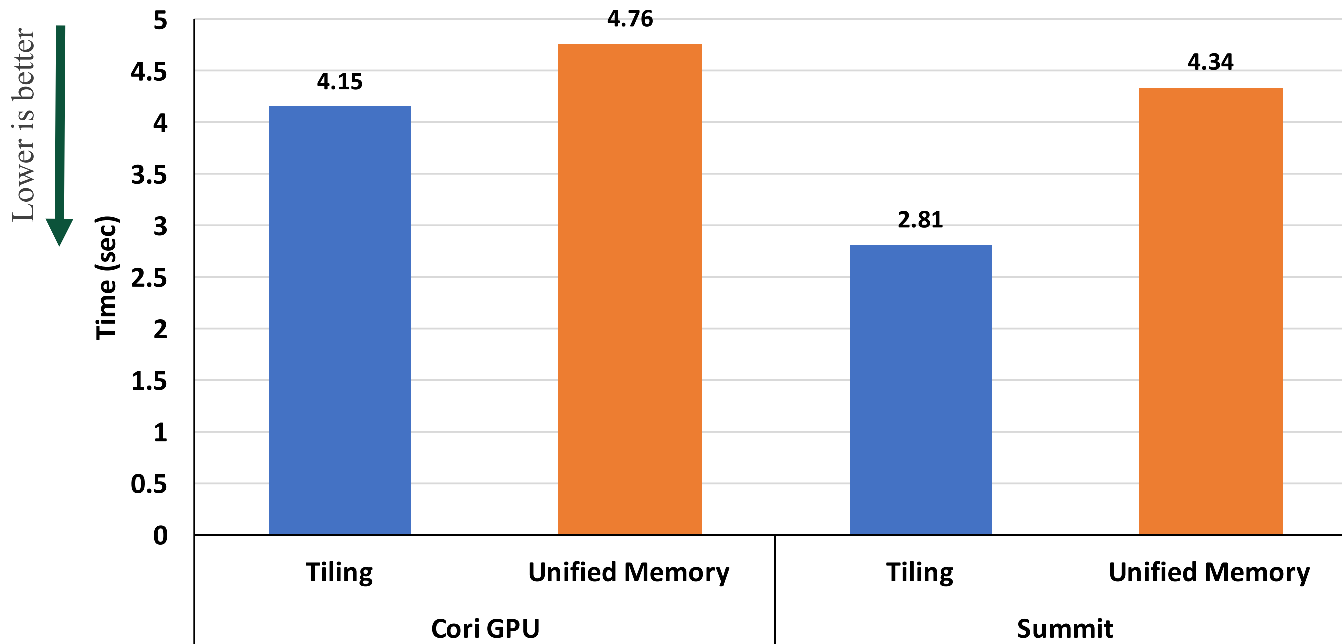
- **Total Memory footprint**: 51.54 GB
- **Main Difference**: Interconnects between CPU and GPU. We measured the following BW in this test application:
  - Cori-GPU – HtoD bandwidth 4 GB/s
  - Summit – HtoD bandwidth 13 GB/s
- Data movement time > Compute time
- We will always perform worse compared to matrices completely resident in GPU memory for this kernel

Evaluation – SPMM Tiling



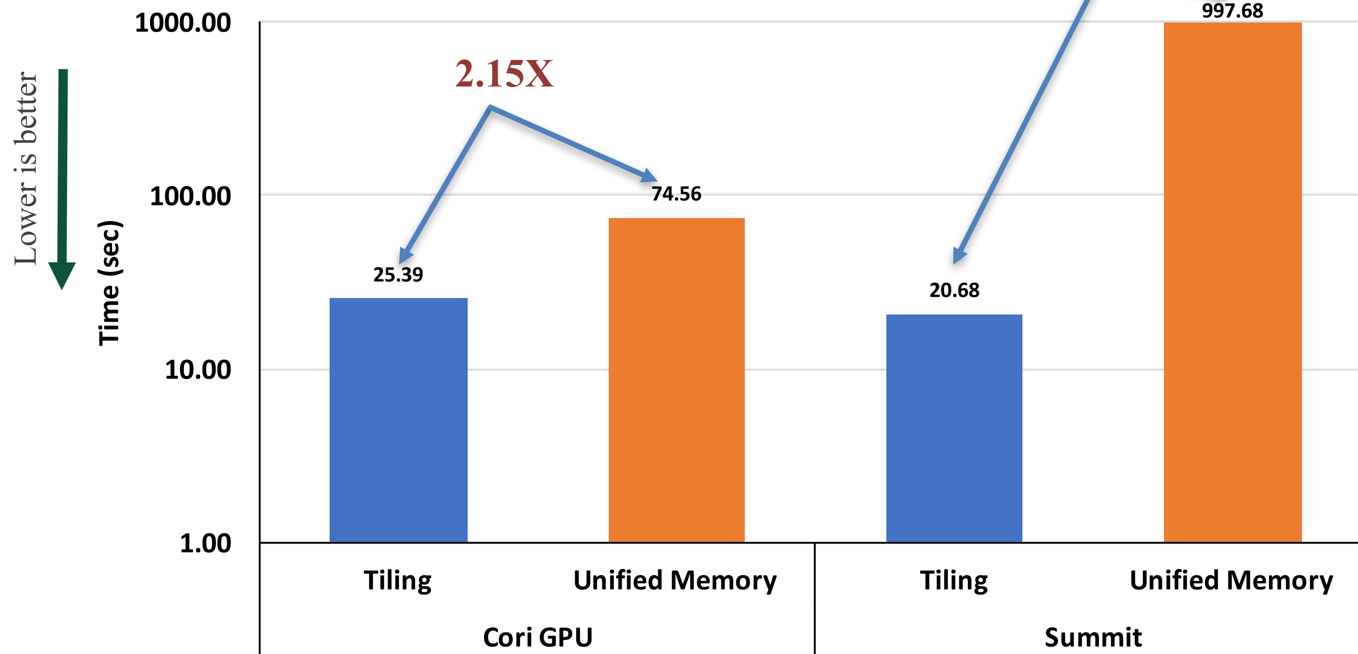
- **Total Memory footprint**: 35.1 GB
- Compute time > Data movement time
- Data movement time could be hidden behind compute time by choosing a clever prefetching scheme such as – using *cudaMemPrefetchAsync*().
- If we could do this prefetching then we would nearly obtain the same throughput compared to matrices resident on GPU.

Evaluation – SPMM (Small Matrix)



- **Total Memory footprint**: 11.7 GB
- Unified memory doesn't hurt the performance.
- This is a productivity win for application programmer: one pointer instead of separate host and device pointer.

Evaluation – SPMM (Big Matrix)

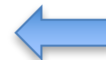


- **Total Memory footprint**: 35.1 GB
- Unified memory gives bad and unpredictable performance.

XY *(cublasDgemm)* Unified Memory – *nvprof* Output

- Summit GPU page faults takes **~30X** longer!!

	Cori-GPU	Summit
GPU page fault group	10.668 sec	313.436 Sec
H2D data transfer	32 GB	32 GB
D2H data transfer	16.64 GB	16.64 GB



- Currently we are in discussion with NVIDIA for understanding the performance on Summit.

Conclusion and Future Work

- We have successfully mixed OpenMP & OpenACC target offloading constructs and CUBLAS library functions.
  - 2.8X – 4.3X speedup over an optimized CPU implementation.
- We have demonstrated that GPUs can accelerate large matrix problems.
  - Tiling is more effective than Unified Memory!
- Future works:
  - Tiling full solver.
  - Finding optimal data movement scheme.

Thank You