Facilitating Electronic Structure Computations on GPU based Exascale Platforms

Jean-Luc Fattebert

Computational Sciences and Engineering Division, Oak Ridge National Laboratory

Team:
Los Alamos National Laboratory: Christian F. A. Negre, Michael E. Wall, Jamal Mohd-Yusof, Joshua Finkelstein, Yu Zhang
Lawrence Livermore National Laboratory: Daniel Osei-Kuffuor
External: Nick Bock (Canonical)

Funded under Co-design Center for Particle Applications (CoPA) project (PI: S. Mniszewski, LANL)

The Exascale Computing Project (ECP)
Project Number: 17-SC-20-SC

www.ExascaleProject.org
Outline

• Motivations

• Computational strategy for Exascale hardware
  – OpenMP offload
  – Vendor libraries (cuSparse, RocSparse, MKL,…) and others (MAGMA,…)

• Solvers
  – Chebyshev dense solver on GPU
  – Distributed

• Some lessons learned
Algorithms and performance portability for electronic structure

• Provide a library to handle the most expensive part of (some) electronic structure codes
  – Computation of single particle Density Matrix – Projector onto subspace associated with lowest eigenvalues of Hamiltonian

• Provide a library that can handle various matrix formats (dense, sparse, distributed) on various hardware (multi-core CPUs, GPUs, multi-nodes)
  – Users can explore algorithms with various matrix formats
  – Users don’t need to worry about implementation
Speeding up electronic structure calculations to enable larger molecular dynamics (MD) simulations

• Time-to-solution is the limiting factor in \textit{ab initio} molecular dynamics
  – How long are we willing to wait for tens of thousands of steps to complete?

• Using the power of GPUs to accelerate these simulations is not an easy task
  – We need enough concurrent operations to use GPU efficiently
  – Larger problems can use GPU resources better, but may lead to time-to-solution that is too long…
Distributing work on several GPUs

• Needs very large problems
  – Each GPU needs enough work to be well utilized

• Time-to-solution in large problems may be too long for MD…

It is difficult to take advantages of multiple-GPUs to speedup Quantum MD
Running MD on exascale platforms

• ECP Application Exascale Atomistic Capability for Accuracy, Length, and Time (EXAALT)
  – Running many MD simulations concurrently

  - ParSplice
    - Launches multiple independent replicas of physical system
  - LAMMPS
    - Molecular Dynamics
  - LATTE
    - Compute atomic forces at quantum level (Tight-Binding)
  - PROGRESS
    - Density Matrix solvers
  - BML
    - Implements matrix operations kernels

Single GPU
Main numerical kernels for electronic structure calculations

• Eigensolver (Dense)
  – Eigenvectors of Hamiltonian corresponding to lowest eigenvalues
  – (For insulators) $\rightarrow$ projector onto space of occupied orbitals

Hamiltonian $H$ (symmetric/Hermitian) $\rightarrow$ Eigenvectors/eigenvalues $(\varepsilon_i, v_i) \rightarrow P = V F V^T$

$$
H v_i = \varepsilon_i v_i
$$

$$
P = V F V^T
$$

$$
f(\varepsilon) = \frac{1}{1 + e^{\beta(\varepsilon - \mu)}}
$$

– $P$ is a symmetric matrix with eigenvalues in $[0, 1]$
– Special case (insulators): $\varepsilon_{HOMO} < \mu < \varepsilon_{LUMO}$
  • $P$ is a projector on subspace spanned by eigenvectors associated with lowest eigenvalues
Underwhelming performance of dense diagonalization on GPU…

- Relative time-to-solution compared to dense matrix-matrix multiplication (dgemm) performance
  - Using Lapack dsyevd on CPU
  - Using MAGMA dsyevd_gpu on GPU

- Similar number of flops but large differences in time-to-solution, specifically for GPUs!

\[ \text{dsyevd: divide \& conquer version of dense diagonalization implemented in Lapack and MAGMA} \]
Developing alternative solvers based on polynomials of matrices

• Iterative solver SP2 for systems with band gap

\[ X_{n+1} = \begin{cases} 2X_n - X_n^2, & \text{Tr}(X_n) < N_e \\ X_n^2, & \text{Tr}(X_n) > N_e \end{cases} \]

with initial guess

\[ x_0 = \frac{\epsilon_{\text{max}} I - H}{\epsilon_{\text{max}} - \epsilon_{\text{min}}} \]


• Chebyshev polynomial expansion of density matrix for metals

Fermi-Dirac function

\[ f(\epsilon) = \frac{1}{1 + e^{\beta(\epsilon - \mu)}} \quad \rightarrow \quad f_H(H) = \left( I + e^{\beta(H - \mu I)} \right)^{-1} \]

\[ \approx \sum_{i=1}^{N} c_i T_i(H) \]

Many of these ideas were introduced to reduce complexity from $O(N^3)$ to $O(N)$

- Full diagonalization in $O(N^3)$
- “Sparse matrix $\times$ sparse matrix” multiplication is $O(N)$
  - $O(N)$ solver provided one can drop off “small” off-diagonal terms that creep in at every iteration
- On GPUs, dense versions of these solvers are competitive with direct diagonalization

SP2 vs. cuSolver on Nvidia V100
[Mniszewski et al., IJHPCA, 2021]

Fastest algorithm on GPU may not be the fastest on CPU
PROGRESS and BML libraries
Implementation divided into two libraries

- **BML**: Basic Matrix Library
  - Linear algebra matrix operations used in solvers
  - https://github.com/lanl/bml

- **PROGRESS**: Parallel, Rapid O(N) and Graph-based Recursive Electronic Structure Solvers
  - Solvers: SP2, Chebyshev, …
  - https://github.com/lanl/qmd-progress
Using OpenMP for GPU offloading

• OpenMP, an implementation of multithreading
  – simple and flexible interface for developing parallel (shared memory) applications

• Usage
  – Add pragmas to C/C++/Fortran loop

• OpenMP 4.5 and beyond
  – Support for offloading to GPU

• Portable
  – Supported by many compilers
  – Turned on with compiler option

```c
#pragma omp target map(from: b) map(to:a)
#pragma omp teams distribute parallel for
for (int i = 0; i < 1000; i++){
    b[i] = 2 * a[i];
}
```
GPU Offload strategy in BML

• Initial plan was to use ‘pure’ OpenMP offload

• Experience
  – Poor performance on critical kernels (sparse-sparse multiply)
  – Do not expect OpenMP to allow fine-grain tuning needed any time soon…

• Current strategy is a hybrid offload programming model
  – OpenMP offload semantics for memory management, data motion
  – Vendor/hardware-specific libraries for critical kernels
  – Some OpenMP native code for non-critical kernels
General implementation strategy

- **CPU**
  - C + OpenMP + MPI + BLAS/Lapack/ScaLapack

- **GPU**
  - OpenMP offload (OpenMP4.5)
  - Rely on MAGMA, ELPA and vendor libraries for performance critical kernels
Computer Science challenges

• Support various architectures
  – GPU: NVIDIA, AMD, Intel
  – no portable library for sparse × sparse matrix multiplication

• Interfaces with various vendor libraries
  – cuSparse, cuSolver, rocSparse, rocSolver, MAGMA, MKL, ScaLapack, ELPA,…
  – We do the work so that users don’t need to understand interfaces to these packages…

• Make OpenMP offload and various libraries coexist
  – Deal with changes in software stack, compiler versions,…
Electronic Structure Software (LATTE, MGmol, DFTB+…)

PROGRESS Solver Library (SP2,…)

Fortran API
- distributed2d

DENSE
- ELLPACK

CSR
- ELLBLOCK

BML Methods

CPU LIBRARIES
- OpenMP
- ScaLAPACK
- BLAS
- LAPACK
- MKL
- ESSL

GPU LIBRARIES
- OpenMP 4.5
- MAGMA
- ELPA
- cuSOLVE
- rocSOLVE
- cuSparse
- rocSparse
- oneMKL
- OpenBLAS
BML: supported (shared memory) matrix formats

- Dense
- ELLPACK
- CSR
- ELLBLOCK

Focus for GPU offload

(a) Matrix

Values

Columns

Non Zero’s
BML: Supporting multiple data types in a C code

• Single precision
• Double precision
• Optional:
  – Single-precision complex
  – Double-precision complex
• Strategy
  – Compile (mostly) same C code several times with different C macros
BML: Fortran interface is important for targeted application codes

- Hand-coded wrapper functions
- Not “automatic,” but low overhead in code writing
- Interface relatively stable
BML: Unit test/Continuous integration

• Over 1,000 unit tests
  – including four different data types and five matrix formats
  – Ctest for developers

• Continuous Integration
  – Pull Requests tested on CPUs with github
  – on GPU using Ascent @ Oak Ridge Leadership Computing Facility (OLCF)
    • Currently testing dense format using MAGMA

• Tracking “issues” on github
Offloading to GPU
Offloading strategy

• Dense format on Nvidia and AMD
  – Rely on MAGMA
  – Use some vendor libraries when better performing (example: dense diagonalization in cuSolver)

• Dense format on Intel and Sparse formats on Nvidia, and AMD
  – OpenMP for memory allocation, CPU-GPU data transfer and various other operations
  – Use vendor libraries for performance critical kernels
GPU offloading with OpenMP

• Matrices are “C struct”
• Mapping to GPU matrix data only, not the whole struct
  – Pointer to datatype
    
```c
REAL_T *A_matrix = (REAL_T*)A->matrix;
#pragma omp target enter data map(alloc:A_matrix[0,sizea])
#pragma omp target update to (A_matrix[0:sizea])
```

  – Full control of data movement between CPU and GPU
Challenges in interfacing with optimized vendor libraries

• Data mapping between BML ELLPACK and vendor sparse formats (CSR) on device

• Some functions in libraries require data to be “ordered” in each row

• cuSparse
  – Workspace required is so large, makes it not practical (CUDA11)
  – implementation of: $C' = \alpha A*B + \beta C$ expects sparsity pattern of $C$ and $A*B$ to be consistent with each other
Using a synthetic Hamiltonian matrix for Performance Benchmarking

- Typical benchmarking requires storing large matrices
- There are no good standard benchmark suite for performance in electronic structure
- We use a synthetic Hamiltonian based on a simple two-orbitals/atom Tight-Binding model
  - Parameters for coupling, onsite energies, distance exponential decay, random noise factor
rocSPARSE performance on Crusher @ OLCF

rocSPARSE DM Build Timings on Crusher

- CPU 8 threads
- Offload GPU
- rocSPARSE initial
- rocSPARSE final

Profiling, fixing implementation issues
Chebyshev expansions for modest matrix sizes (metals)
Chebyshev expansion of Density Matrix

• An alternative to inefficient dense diagonalization on GPU is to use a Chebyshev polynomial expansion

\[ D = (I + e^{\beta(H-\mu I)})^{-1} = f(H) \]

\[ \sum_n c_n T_n(H) \approx f(H) \]

• Problem
  – Expansion can involve over 100 terms and be computationally very costly too
Patterson and Stockmeyer trick

- For $km$ number of terms, $k-1+m-2$ multiplications for $x$ needed

- So, for $K$ terms in an expansion, only need $\sim 2\sqrt{K}$ multiplications of $H$

- Substantial savings when $x$ is a matrix and cost dominated by matrix multiplications!

- Adapted for Chebyshev and DM calculation

\[
a_{km-1}x^{km-1} + \cdots + a_1x + a_0
= \frac{(\cdots(((a_{km-1}x^{k-1} + \cdots + a_{k(m-1)+1}x + a_{k(m-1)})x^k} \sum_{m-1}
+ a_{k(m-1)-1}x^{k-1} + \cdots + a_{k(m-2)+1}x + a_{k(m-2)})x^k} \sum_{m-2}
\vdots
\vdots
+ a_{2k-1}x^{k-1} + \cdots + a_{k+1}x + a_k)x^k} \sum_2
+ a_{k-1}x^{k-1} + \cdots + a_1x + a_0} \sum_1
\]
Chebyshev expansion compared to direct diagonalization

- Time-to-solution on Nvidia V100
  - Baseline is cuSolver
  - Speedup more important for smaller matrices
Exploiting GPU concurrency in calculating Chebyshev terms

• For “small” matrices, a single matrix-matrix multiplication does not fully utilize a GPU

• Several Chebyshev terms can be computed concurrently and use GPU streams for an additional speedup

\[
\begin{align*}
H & \rightarrow H^2 & & & & & & & & \rightarrow H^5 \\
& & & & & & & & & & \rightarrow H^6 \\
& & & & & & & & & \rightarrow H^7 \\
& & & & & & & & & \rightarrow H^8 \\
& & & & & & & & & \cdots
\end{align*}
\]
Distributing computation

1. D&C based on matrix elements
2. Distributed Linear Algebra
Graph-based distributed solver implemented in PROGRESS

• Computations are distributed following a divide and conquer (D&C) approach
• Automatic and adaptive partitioning of matrix using graph-based thresholding
• Sub-systems solved concurrently using single-node solvers developed in project
• $O(N)$ for given threshold/subsystem size

Balancing computational cost and accuracy with matrix thresholding

Graph threshold value vs. Speedup

Graph threshold value vs. Average element error

Legend:
- 1081 Orbitals
- 2172 Orbitals
- 4344 Orbitals
Distributed BML format: “distributed2d”

- 2D matrix decomposition using MPI
  - P x P tasks layout
  - Square submatrices
- Each submatrix is a “shared memory” BML matrix
  - Leverage developments for shared memory formats

```c
struct bml_matrix_distributed2d_t
{
    bml_matrix_t *matrix;
    MPI_Comm comm;
    int ntasks;
    int nprows;
    int npcols;
    int myprow;
    int mypcol;
    int mpitask;
    ...
}
```

Local sub-matrix in dense, ellpack, csr, ellblock format
A non-intrusive implementation

• “Wrapper” calling sub-matrix operations when possible
  – “distributed2d” operations combinations of “shared memory” matrix operations
  – Shared memory code untouched

• Some operations simply need reduction at the end
  – Frobenius norm,…

• Some operations require substantial communications
  – multiplication, transpose,…

• Some operation are more intrusive
  – Bounds on eigenvalues using Gershgorin circles

• Some operations are beyond scope
  – Eigensolver: interface with existing solver (ScaLapack and ELPA)
Distributed BML format: matrix-matrix multiplication

• Implemented Cannon’s algorithm for matrix-matrix multiplication
  – P-length loop over matrix blocks
  – 2 point-to-point communications at each step
    • “shift” blocks to enable computation of local block in product

\[ C01 = A00 \times B01 + A10 \times B11 \]
What about wavefunction-based solver? (Planewaves...)
Numerical Discretization of DFT problem

• Plane-Waves or Finite Differences
  – Large number of degrees of freedom (DOF) / electronic wave function

• Solution
  – M x N “tall-and-skinny” matrix of coefficients
  – Number of DOF/ wave function M \sim 1,000 \times \text{number of wave functions } N

• Hamiltonian very sparse, but very large!
Eigenvalue problem in wavefunction-based solver (Plane Waves, …)

• Project eigenvalue problem into smaller dimension to compute Density Matrix
  – Solve eigenvalue problem

\[ \Psi^T H \Psi V = \Psi^T \Psi V \Lambda \]  \[ \rightarrow AV = SV\Lambda \]

Wavefunctions define smaller dimension space

• Build new trial eigenvectors of \( H \)
  \[ \Psi \rightarrow \Psi V \]

Generalized dense eigenvalue problem

Small problem compared to basis set size!

• Update wave functions using preconditioned gradient
  \[ \Psi \rightarrow \Psi - eKR \]
Proxy-app: Loewdin orthogonalization

• Distributed computation of Gram matrix \( S = \Psi^T \Psi \)

• Accumulate \( S \) on each GPU (communication)

• Compute \( S^{-1/2} \) on each GPU (replicated computation)

• Apply \( S^{-1/2} \) to \( \Psi \)

Diagonalization would involve the same operations!
Parallel scaling/performance on Summit

- Matrix 3,000,000 x 3,000
- Dense iterative solver converges in 7 iterations
- Time-to-solution better than 1 s
- Collective communications using NCCL library

[Dense diagonalization is bottleneck in strong scaling limit!]

[Lupo Pasini, Turcksin, Ge, Fattebert, Parallel Computing (2020)]
Lesson learned: Efficiently using GPUs requires a lot of work!

• OpenMP alone not always sufficient to get “good” performance
• Relying on vendor libraries can help
  – requires understanding well interfaces, requirements,… for each library
• Building a software stack supporting multiple GPUs and third-party libraries is a challenging task
  – there are still a number of Computer Science challenges on exascale architectures…
• More **GPU-friendly algorithms** can provide substantial speedup on GPU accelerators and enable faster time-to-solution in electronic structure calculations
Acknowledgments

This work was performed at Lawrence Livermore National Laboratory under U.S. Government Contract DE-AC52-07NA27344, Oak Ridge National Laboratory under U.S. Government Contract DE-AC05-00OR22725, Los Alamos National Laboratory under U.S. Government Contract 89233218NCA000001.

This research was supported by the Exascale Computing Project (http://www.exascaleproject.org), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.

Project Number: 17-SC-20-SC

www.ExascaleProject.org