Chapter 1

Introduction

1.1 Density Matrix Renormalization Group

The Density Matrix Renormalization Group (DMRG) algorithm, is a numerical approach for low-dimensional interacting many-body quantum systems.[1] The original DMRG algorithm was published in 1992 in a Letter by S.R.White.[2] It was a generalization for the renormalization group process used by K.G. Wilson for the Kondo Hamiltonian.[3]

The DMRG algorithm was made to tackle quantum many-body physics problems. To solve these type of problems, the eigenvalues and eigenvectors of the Hamiltonian needs to be calculated. However the Hilbert space of the system grows exponentially with the system size. Therefore, solving these systems exactly through a full diagonalization would take a large amount of memory and computational time. There are few things that can help with reducing the amount of work needed. Often, the ground state and the first few excited states are enough to study the system considerably, especially at low temperatures. Symmetries and the known quantum numbers of desired state can also be exploited to reduce the size of the Hilbert space.[4]

However, as stated earlier, the expansion of the Hilbert space is exponential with respect to the system size. At large systems, even with the size reductions, the Hilbert space would still be too large to be solved exactly. The DMRG algorithm counters this by starting with a small system that can be solved exactly, and then increasing the system size without increasing the size of the Hilbert space by truncating it back to the original size. This trunctation is done with the dominant eigenpairs conserved.

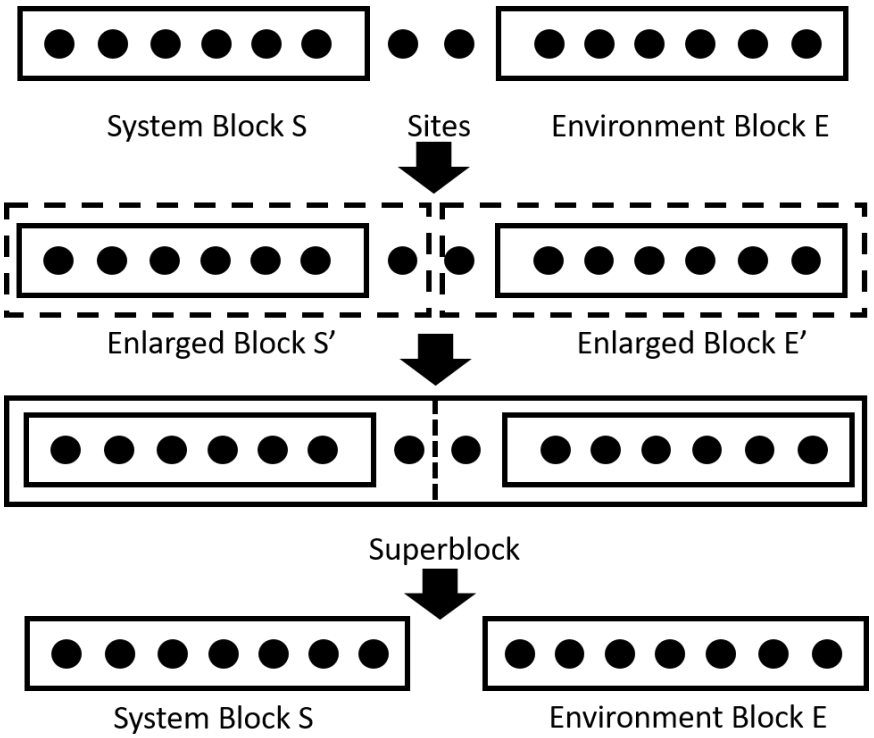


Figure 1.1: Illustration of the Basic DMRG Step.

This process is done in several steps, starting from blocks of small size containing n sites:[4][5]

1. The block is connected to a site, forming an enlarged block.
2. Two of these enlarged blocks are connected forming a superblock, using either open or periodic boundary conditions.
3. The target state (commonly the ground state) of the superblock Hamiltonian is calculated.
4. A reduced density matrix of the system is made from the target state by tracing out the environment.
5. The eigenstates of the matrix is calculated and sorted by their eigenvalues. The top m states are kept while the others are discarded. A sum of the discarded states are kept as a measure of the truncation error.
6. A transformation matrix is created from the m states. Each operator are rotated and truncated according to the transformation matrix.

This process is repeated at each expansion of the system. This way, the Hilbert space of the system can be kept small while the system size is growing. To keep the accuracy of the calculation, the truncation error must be kept minimal (no larger than 10^{-4}) after each expansion and truncation.

There are two implementations of the DMRG algorithm that are involved in this study, the infinite system algorithm and the finite system algorithm.

1.2 Infinite System Algorithm

The infinite system algorithm is the original algorithm formulated back in 1992.[2] The goal was to use the DMRG's advantage to decouple the system size and the size of the Hilbert space and calculate the ground state eneriges of large systems, eventually converging to the thermodynamic limit. In this algorithm, the left enlarged block is mirrored to the right. Thus, two new sites are added to the superblock in each DMRG step.[4] The detail of this is as follows,[6]

1. Consider a small lattice of size ℓ, forming the system block S. S has a Hilbert Space of size M^S with states $\{|M_\ell^S\rangle\}$; the Hamiltonian $\hat{H}_\ell^S$ and the operators acting on the block are assumed to be known in this basis. At initialization, this maybe an exact basis of the block ($N_{site}^\ell \leq M^S$). Form an environment block E of the same size ℓ.

2. Form an enlarged block S' by adding one site to S. S' has a Hilbert space of size $N^S = M^S N_{site}$ with a basis of product states $\{|M_\ell^S \sigma\rangle\} \equiv \{|M_\ell^S\rangle\,|\sigma\rangle\}$. Similarly, a new environment block E' is formed.

3. A superblock of size $2\ell + 2$ is formed from S' and E'. The Hilbert space of this superblock has a size $N^S N_E$, and has a Hamiltonian $\hat{H}_{2\ell+2}$.

4. Find the ground state $|\psi\rangle$ of the superblock Hamiltonian. This is the most time-consuming part of the algorithm.

5. Form a reduced density matrix $\hat{\rho} = \mathrm{Tr}_E\,|\psi\rangle\,\langle\psi|$, where the states of the environment have been traced out, $\langle i|\,\hat{\rho}\,|i'\rangle$ and determine its eigenbasis $|w_\alpha\rangle$, ordered by descending eigenvalues.

6. Get the M^S eigenstates with the largest eigenvalues and form a new reduced basis for S'. In the product basis of S', their matrix elements are $\langle m_\ell^S \sigma | m_{\ell+1}^S\rangle$; taken as column vectors they form a $N^S \times M^S$ rectangular matrix T. Do the same for the environment.

7. Do the reduced basis transformation $\hat{H}_{\ell+1}^{\mathrm{tr}} = T^\dagger \hat{H}_{\ell+1} T$ onto the new M^S-state basis and take $\hat{H}_{\ell+1}^{\mathrm{tr}} \to \hat{H}_{\ell+1}$ for the system. Repeat the transformation for the environment.

8. Restart from step 2 with block size $\ell+1$ until some final desired length is reached.

9. Calculate the desired ground state properties (energies and correlators) from $|\psi\rangle$.

If the Hamiltonian is reflection-symmetric, the system and the environment is considered to be identical. While any state accessible by large sparse matrix diagonalization of the superblock Hamiltonian can be the target state $|\psi\rangle$,[6] this study will be using the ground state only.

The goal of the infinite system algorithm is to grow the system to a be big enough such that the energy and the short range correlations around the center has converged. This is checked by keeping track of the difference ΔE_0 between the ground state energy of the superblocks of two successive states.[4][7].

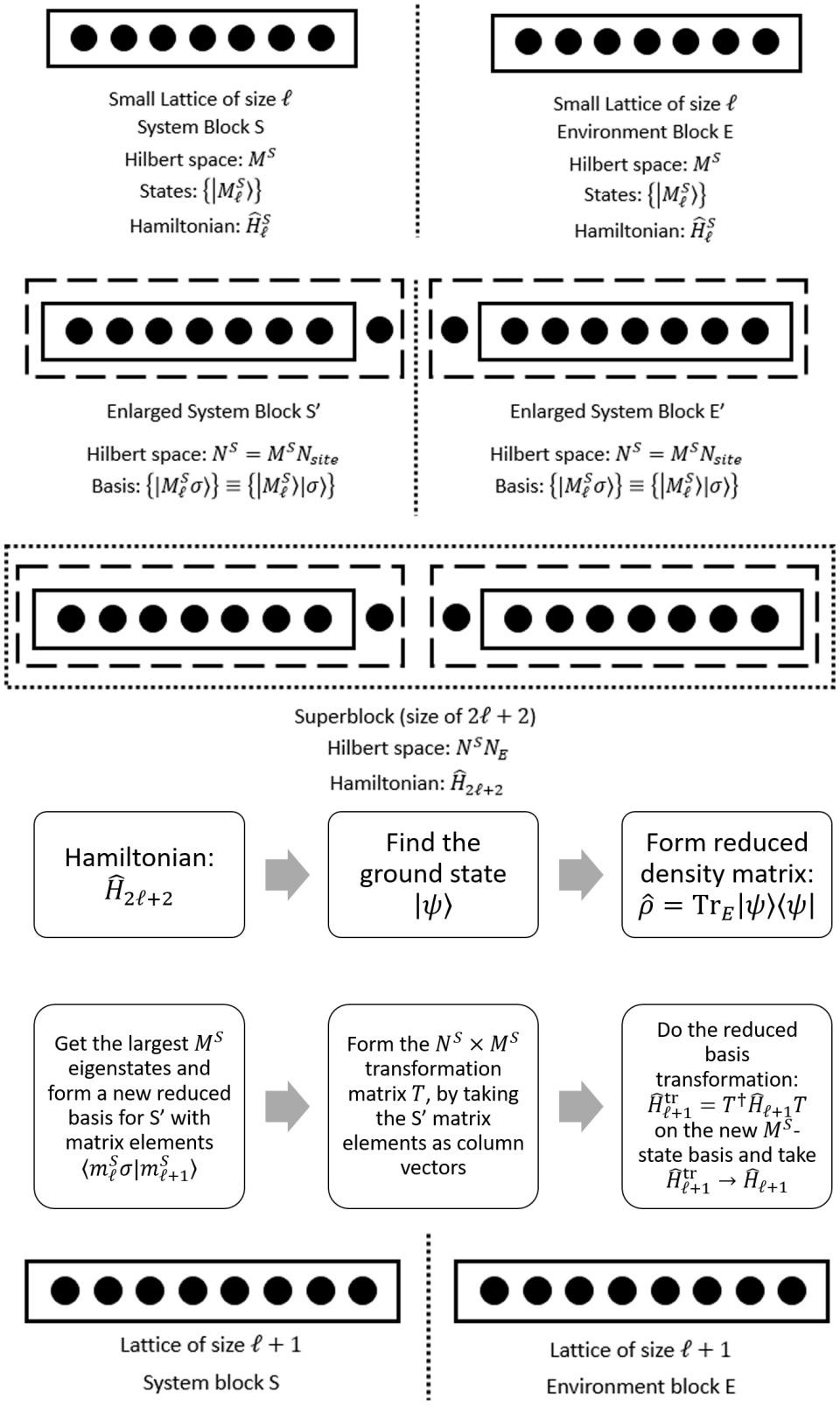


Figure 1.2: Infinite System DMRG Basic Step

1.3 Finite System Algorithm

For a lot of problems, the infinite system DMRG algorithm does not yield satisfactory answers. An example of this are when accounting for the strong effects of impurities or randomness in the Hamiltonian as the total Hamiltonian is not yet known in intermediate steps. Systems with strong magnetic fields or close to a first order transition can be trapped in a metastable state favored for small system sizes by e.g. edge effects.[6]

These issues are largely eliminated by the finite system algorithm. The goal of the finite system algorithm is not to reach the thermodynamic limit like the infinite system algorithm, but to restrict the calculations on a finite system size L. Until the superblock size reaches the intended system size L, the algorithm is identical to the infinite system algorithm. The specifics of algorithm is as follows,[4][6]

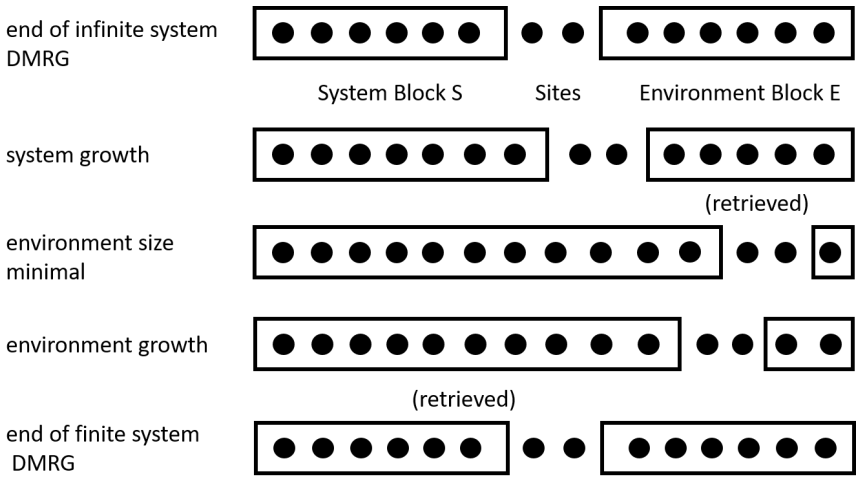


Figure 1.3: Illustration of the finite system DMRG algorithm.

1. The infinite system algorithm is used until the system size reaches the desired system size L. Save the operators after each truncation to disk.

2. Enlarge the system block size $\ell + 1$. Read a block of size $L - \ell - 2$ from the disk. This is the environment block.

3. Enlarge the environment block to the size $L-\ell-1$, and form a superblock from the enlarged system and environment block.

4. Find the ground (or target) state $|\psi\rangle$ of the superblock Hamiltonian.

5. Form a reduced density matrix $\hat{\rho} = \mathrm{Tr}_E \, |\psi\rangle \langle\psi|$, where the states of the environment have been traced out, $\langle i| \, \hat{\rho} \, |i'\rangle$ and determine its eigenbasis $|w_\alpha\rangle$, ordered by descending eigenvalues.

6. Get the M^S eigenstates with the largest eigenvalues and form a new reduced basis for S'. In the product basis of S', their matrix elements are $\left\langle m^S_\ell \sigma | m^S_{\ell+1} \right\rangle$; taken as column vectors they form a $N^S \times M^S$ rectangular matrix T.

7. Do the reduced basis transformation $\hat{H}^{\text{tr}}_{\ell+1} = T^{\dagger}\hat{H}_{\ell+1}T$ onto the new M^S-state basis and take $\hat{H}^{\text{tr}}_{\ell+1} \to \hat{H}_{\ell+1}$ for the system. Save the block with the basis to the disk. The transformed, truncated enlarged system block becomes the system block for the next step.

8. Repeat from step 2 until the environment block becomes a single site.

9. If the environment block is a single site, the growth direction is reversed. This pair of complete shrinkage and growth sequence of both blocks is called a sweep.

10. Repeat the sweeps until convergence between sweeps is reached.

While there is no guarantee that finite system DMRG is not trapped in some metastable state, it usually finds the best approximation to the ground state. The convergence process may take from a few to several dozens of sweeps especially with electronic problems at disproportional fillings and random potential problems. In some cases, a seemingly converged finite system result is suddenly improved after some further sweeps, showing a metastable trapping in the previous answer. Therefore it is recommended to carry out additional sweeps and to test for convergence by doing runs on varying M. One sweep of the finite system algorithm takes two to four times the calculation time of an infinite system DMRG calculation.[6]

1.4 The DMRG Code and Model Used

The code that will be used in this study will be based on the Simple-DMRG python code given at the 2013 summer school on quantum spin liquids in Trieste, Italy.[5] The code presents DMRG in the traditional formulation (not using Matrix Product States). Python, and this code in particular, is chosen to be the basis of this study to maximize the readability for people who are new to programming. The code uses the Heisenberg XXZ chain model for, and although other models can be studied using the code, this study will also only use the XXZ model for convenience.

The Spin 1/2 Heisenberg XXZ model on a periodic spin chain of N sites is described by the Hamiltonian,[1][8]

$$\hat{H} = \frac{J}{2}\sum_{j=1}^{N-1}\left(S^{+}_{j}S^{-}_{j+1} + S^{-}_{j}S^{+}_{j+1}\right) + \Delta\sum_{j=1}^{N-1}S^{z}_{j}S^{z}_{j+1} \tag{1.1}$$

where $J, \Delta > 0$, J is the exchange coupling, Δ is the anisotropy parameter and $S^{\alpha}(\alpha = +, -, z)$ are Spin 1/2 operators. The Spin 1/2 Heisenberg XXZ model simulates magnetic systems with anisotropy and provides a theoretical description of various experimental measurements such as critical points. For this study, both the exchange coupling and the anisotropy parameter are kept at 1. The code used for the DMRG is provided in Appendix **??**.

However, even with the DMRG algorithm, calculations may take days, even weeks to solve large complex systems. Thus, we try to improve the performance of the algorithm using GPU acceleration.

1.5 Calculation Acceleration using GPU

The use of Graphical Processing Units (GPUs) in general computations is introduced during the 2000s, like works on matrix multiplication[9], conjugate gradients and multi-grid solver techniques[10], nonlinear least-square problem[11] and linear algebra operators.[12] Often, this technique leads to a speedup in the computation if used properly. This is due to the difference in architecture of the Central Processing Unit (CPU) and the GPU. While the CPU has few robust cores capable of handling complicated tasks, the GPU has numerous weaker cores that can only handle simple calculation. This makes the GPU fit for solving simple very large data calculations such as matrix operations.

There are two hurdles in using consumer level GPUs for calculation. The first is the accuracy, as most consumer level GPUs has a fast single precision performance but slow double precision performance. The second is the copying overhead. Data must be copied from the CPU to the GPU before the calculation is done. Thus, for a speedup to occur the sum of the copy-time and the run-time of the GPU must be smaller than the run-time of the CPU.

While the original DMRG algorithm was formulated for a single-threaded application, several studies has been made to accelerate the algorithm using parallel computing. One of these is using shared memory CPU multi-processing using the OpenMP API[13]. With regards to GPU parallelization, a study using CUDA C was done by accelerating the calculation of the ground state of the superblock Hamiltonian.[1] This study follows in that track, using the Python libraries available for GPU calculations.

The computer used in this study has the specifications: two Intel Xeon E5-2620 processors operating at 2.0 GHz, 64 GB of memory and an NVIDIA GTX 1070 GPU. The code is run on Python 3.6 and uses CUDA 9.2 for the basis of the GPU libraries.

This rest of the study is divided into the following sections. First, an introduction and benchmark test for the Python GPU library functions are done to determine at what matrix sizes should the GPU be employed. Second following[1], the library functions are introduced to the Jacobi-Davidson algorithm. Third, these algorithms are put into the DMRG algorithm and tested. Fourth, the summary of the study and recommendations for future studies is given.

Chapter 2

Python CUDA Implementation

There are three packages available in Python that would enable GPU calculations, the first is Numba which is a library sponsored by Anaconda Inc.[14], the second is PyOpenCL, which uses OpenCL for all GPU brands[15] and third is PyCUDA, which uses CUDA - NVIDIA's proprietary library for their GPUs.[15] The choice between the three depends on what hardware is being used, but since the computer used in the study uses an NVIDA GPU, and based on a test done comparing Python computational libraries[16], the study used PyCUDA 2018.1.1.

PyCUDA links the NVIDA GPU drivers and maps all of CUDA to the Python API. What it means is that a CUDA C code can be run inside the Python Code under a wrapper with almost zero overhead. PyCUDA also has additional functionality such as a GPU-array class that can automatically handle the copying, retrieving, deleting and block size allocation to the GPU memory for arrays. This takes away the complexity of working with GPU-based computation for people unfamilar to it.

Another package used in conjunction to PyCUDA is the scikit-cuda-0.5.2-egg (skcuda) library.[17] The skcuda library provides additional wrappers for the CUBLAS, CUFFT and CUSOLVER dense libraries. For the CUSPARSE library, an additional experimental package cuda-cffi, which generates library wrappers for CUSPARSE automatically has been used.[18]

As stated in Chapter 1, the most time-consuming part of the DMRG process is solving for the target state of the superblock hamiltonian. Thus, a test for the dense hermitian eigensolver and singular value decomposition(svd) using 100 randomized 5000×5000 symmetric matrices was done. As can be seen in Figure 2.1, the Python CUSOLVER implementations are still slower than NumPy[19]. Thus, there won't be any benefit in using the dense CUSOLVER library. Unfortunately, a test using the sparse CUSOLVER library can't be done as the packages used does not support the library at the current time.

One thing that can be done is to find an eigensolver algorithm that has similar performance to the NumPy and SciPy functions, and try to accelerate it using CUBLAS functions for matrix-matrix and matrix-vector multiplications.

A double precision dense matrix multiplication test on Figure 2.2.a and 2.2.b, and a single precision dense matrix multiplication test on Figure 2.2.c and 2.2.d. These tests are done on 100 matrices with varying sizes. As can be seen, while there is a speed up on both cases, there is more speedup on the single precision matrix test.

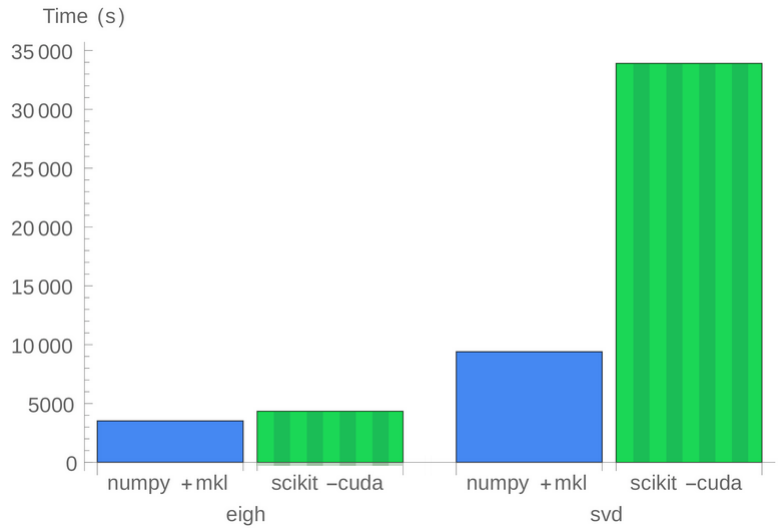


Figure 2.1: Hermitian Eigensolver and SVD comparison.

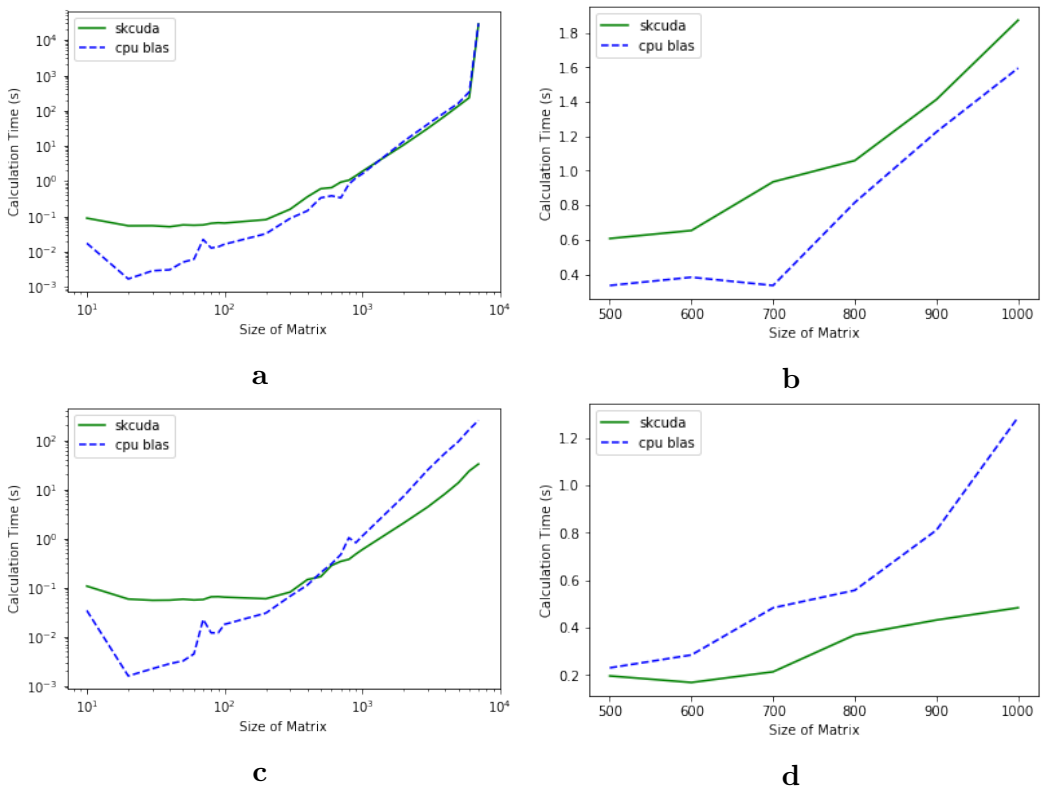


Figure 2.2: Double Precision (a & b) and Single Precision (c & d) Matrix-Matrix product benchmark test.

And as shown on Figures 2.2.b and 2.2.d, at around 500×500 to 1000×1000 a speedup has already occurred with single precision while there is none with double precision. On consumer-level GPUs, this type of performance gap is expected. Therefore, for our purposes, GPU calculations will mostly stick to single precision.

Another test, done by copying a dense matrix to the GPU once and multiplying

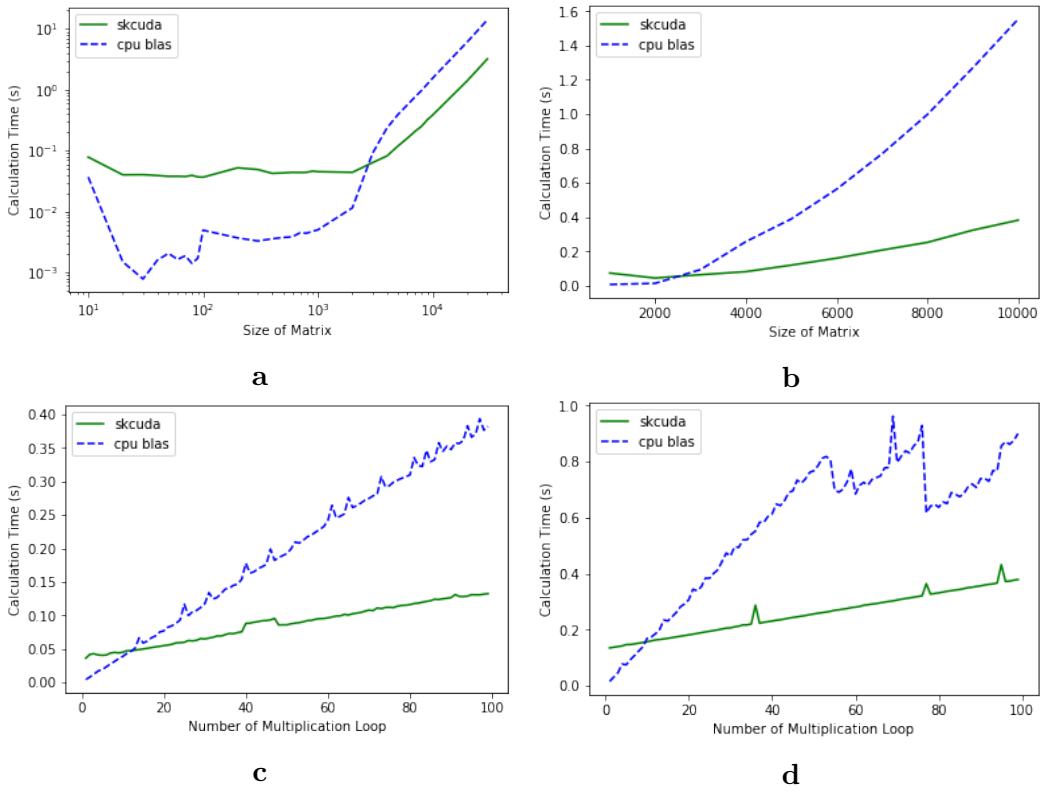
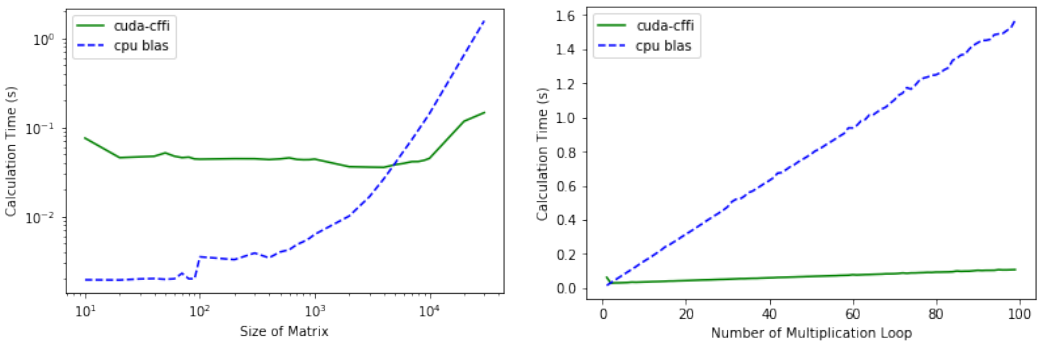


Figure 2.3: Single Precision Dense Matrix-Vector Product for (a & b) Fixed 100 Multiplication Loops with Varying Sizes and (c & d) Fixed Size with Varying Multiplication Loops.

them with a vector 100 times, can be seen on Figure 2.3.a. A better look on where a speedup starts to occur is seen in Figure 2.3.b. Another test, this time with the size fixed to 5000×5000 and 10000×10000 shows how many times the matrix has to be multiplied with a vector for a speedup to occur is shown in Figure 2.3.c and Figure 2.3.d. One of the trade-offs of using GPUs for computation is the data copying overhead. Which is why multiple iterations of matrix-vector multiplications using the same matrix copied once, is needed to achieve a speedup compared to a CPU only operation.

Last, same with the previous matrix-vector tests using the experimental CUSPARSE wrapper package is done for Figure 2.4. The result is similar to the dense library, where a speedup only occurs above the thousands.

Based on the results shown, the Python implementation of CUBLAS and CUSPARSE functions shows a speedup compared to the CPU-only functions of NumPy and SciPy. The eigensolver and svd of the CUSOLVER library didn't show any speedup, and thus a custom eigensolver needs to be devised for the purposes of the study.



a **b**

Figure 2.4: Single Precision Sparse Matrix-Vector Product for (a) Fixed 100 Multiplication Loops with Varying Sizes and (b) 10000×10000 Matrix with Varying Multiplication Loops.

Chapter 3

Jacobi-Davidson Algorithm

There are several options in getting the target state of a large matrix. An attempt at making a stabilized Lanczos algorithm was done before and can be seen in Appendix **??**.[19] An alternative to this is the Jacobi-Davidson algorithm.

The main idea of the the original Davison algorithm is to expand the subspace such that certain eigenpairs would be favored.[20] If the true eigenvector is in the subspace of the current iteration, then the eigenproblem would give the corresponding exact eigenpair. Therefore, a faster convergence can be achieved if the new expansion vector is the component of the error vector orthogonal to the subspace.[21][22] The Davidson algorithm is most suited for diagonally dominant matrices.

The Davidson algorithm solving for the lowest k eigenvectors of a matrix $\mathbf{H}$ can typically split as follows:[22]

1. A guess eigenspace is constructed with dimension $l(l \geq k)$ and an orthonormal basis $\{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, ..., \mathbf{b}_l\}$.

2. Compute the Matrix-Vector Product $\{\mathbf{Hb}_1, \mathbf{Hb}_2, \mathbf{Hb}_3, ..., \mathbf{Hb}_l\}$, and calculate the matrix elements of the subspace $\tilde{\mathbf{H}}_{ij} = \mathbf{b}_i^T \mathbf{Hb}_j$, and solve for the eigenvalues $\{\rho_i\}$ and eigenvectors $\{\mathbf{x}_i\}$ in the subspace as the approximate eigenpairs.

3. Solve the residue vector $\mathbf{r}_i = (\mathbf{H} - \rho_i \mathbf{I})\mathbf{x}_i$. If $\|\mathbf{r}_i\| \leq$ tolerance, then convergence is achieved and the algorithmm ends.

4. Calculate corrections based on the residue and approximate eigenpairs.

5. Orthogonalize the correction vector to the previous subspace using a modified Gram-Schmidt scheme and append to the subspace. Repeat from step 2 until convergence is reached.

There are several methods for calculating the correction vector, but what will be used in this study is the Jacobi Orthogonal Component Correction.[23]

Following the results from Appendix **??**, iterative algorithms like this benefit from GPU acceleration during the basis projection operation, while corrections should be done with the CPU.[19] This helps in keeping some of the accuracy and stability of the CPU based-algorithm while benefiting from the speedup the GPU calculations would provide. The final code for the Jacobi-Davidson algorithm can be perused in Appendix **??**.[24]

function $[\lambda, \mathbf{u}] = \text{JOCC}(\boldsymbol{A})$:

–extract part of the matrix
$n = \text{size}(\boldsymbol{A}, 1)$
$\boldsymbol{F} = \boldsymbol{A}(2:n, 2:n)$
$\boldsymbol{d} = \text{diag}(\boldsymbol{F})$
$\boldsymbol{b} = \boldsymbol{A}(2:n, 1)$
$\alpha = \boldsymbol{A}(1, 1)$
–initialization

$z = \text{zeros}(n-1, 1)$
–iteration
while not converged:
$\lambda = \alpha + \mathbf{b}^T\mathbf{z}$
$z = (\mathbf{d} \odot \mathbf{z} - \boldsymbol{Fz}\text{-}\boldsymbol{b}) \oslash (\mathbf{d} - \lambda)$
end while
$\mathbf{u} = [1; \mathbf{z}]$

Figure 3.1: Jacobi Orthogonal Component Correction

A test, using three different large symmetric sparse diagonally dominant matrices, was conducted to determine the robustness of the code in comparison to the SciPy ARPACK eigensolver. The first matrix, A, was created using a randomized symmetric sparse matrix with the diagonal elements ranging from 0 to 100. The second and third matrices, B and C, are taken from the DMRG at the same max state M but at longer and shorter chain length respectively. Each matrices was solved for 100 times to account for the differences in run-time caused by the randomized starting vector. The results of the Davidson is highly dependent on the dominance of the diagonal. Thus, we define

$$r = \frac{\sum_{diag}|a_{ij}|}{\sum_{non}|a_{ij}|} \tag{3.1}$$

as the measure of the diagonal-dominance of the matrix. The three matrices used has an r of A) 99.63, B) 77.85 and C) 8.28 respectively.

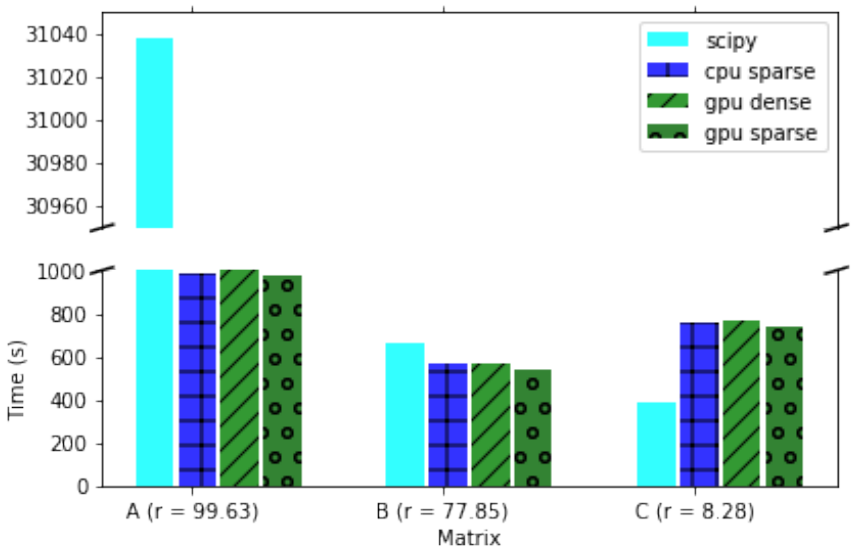


Figure 3.2: Davidson algorithm speed test results

As can be seen in Figure 3.2, the Davidson algorithm is faster than the SciPy function for very large r (A and B) but slower at lower r values (C). This may be due to the specialization of the Davidson algorithm to diagonally dominant matrices. In

terms of the GPU acceleration, we see that the Dense GPU operation performed worse than the Sparse CPU operation, but the experimental implementation of the Sparse GPU is the fastest out of three implentation of the Davidson algorithm. The speed up between the Sparse CPU and GPU implementation ranges from being (0.95% to 4.14%). While the value may seem small, the runs only ran the multiplication around 20-30 times before the answer converged. Thus, consulting Figure 2.4, for matrices that take slower to converge, more speed-up is expected to occur between the two.

Table 3.3: Average eigenvalue accuracy of the Davidson algorithm

	Matrix A (%)	Matrix B (%)	Matrix C (%)
CPU Sparse	3.95×10^{-6}	2.10×10^{-7}	6.10×10^{-7}
GPU Sparse	1.34×10^{-5}	1.54×10^{-6}	2.69×10^{-6}
GPU Dense	8.95×10^{-7}	2.27×10^{-6}	3.18×10^{-6}

Table 3.4: Average eigenvector accuracy of the Davidson algorithm

	Matrix A (%)	Matrix B (%)	Matrix C (%)
CPU Sparse	0.27	0.07	0.07
GPU Sparse	0.43	0.06	0.15
GPU Dense	0.13	0.07	0.20

We then compare the average accuracy for all the matrices compared to the answers SciPy was getting. The GPU implementations show a drop in accuracy compared to the CPU implementation but the difference in error in both the eigenvalue and eigenvectors are small.

Chapter 4

DMRG Results

The test the length of the spin chain was kept at 100 sites, and with the anisotropy parameter $\Delta = 1$. The result for the infinite system algorithm with max states $M = 50$, is as follows,

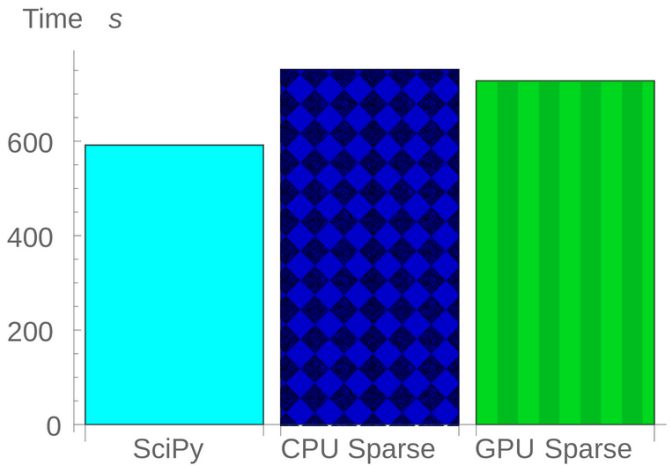


Figure 4.1: Infinite System Algorithm M = 50 and L = 100.

The figure shows that the SciPy implementation is faster than both implementations of the Davidson, with an error in Energy for both at around 10^{-5} %.

The finite system algorithm, using $M = 50$ for warm-up and $M = 50, 60, 70, 80, 80$ for the sweeps yields the results in Figure 4.3. The reason for choosing these M is because the GPU memory can only handle up to $M = 80$ without crashing,

The results are similar to the previous results, but the gap between the CPU and GPU implementation of the Davidson algorithm becomes larger. While still at 10^{-5} % Energy error, the Davidson algorithm loses to the SciPy ARPACK function. This may be because of the measure of the diagonal dominance of the superblock Hamiltonian isn't high enough for the Davidson to beat the SciPy ARPACK. This is alluded with the results of the speed test in Chapter 3.

Longer chain lengths give a higher diagonal dominance ratio r. With how DMRG works, the system starts with a smaller chain that gets longer until it reaches the desired size. Thus it's better to use the SciPy function, until the chain reaches the threshold where r is big enough for the Davidson to be faster.

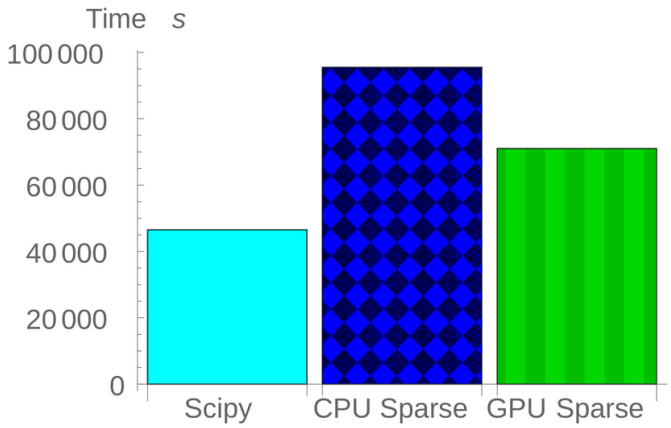


Figure 4.2: Finite System Algorithm M = 50, 60, 70, 80, 80 and L = 100.

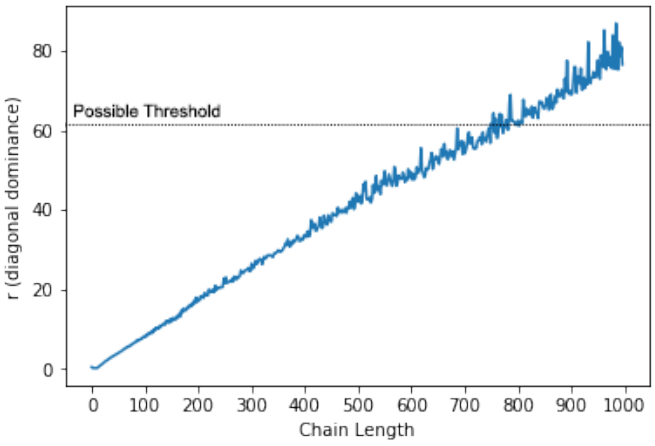


Figure 4.3: Diagonal-Dominance r of the superblock hamiltonian matrix at M = 50 and varying chain length.

However, optimizing for this likely varies from system to system. And for the XXZ system, the threshold for speedup is likely above the chain length of 500 sites and for most cases such size of a system is unnecessary.

Regardless, a speedup is still shown between the CPU and the GPU implementation of the Davidson. And while the gap is small (3.34%) for the infinite case, it widens considerably in the finite case (25.63%) due to the larger superblock hamiltonian matrices at larger M.

Chapter 5

Summary and Conclusions

The python GPU packages PyCUDA, scikit-cuda, cuda-cffi, were tested in comparison to the NumPy and SciPy packages. The CUSOLVER implementation of the general hermetian eigensolver and SVD was slower than the Numpy functions. But the Matrix-Matrix and Matrix-Vector dot products are faster using the GPU than the CPU.

A Davidson algorithm was modified to use the GPU Matrix-Vector Products, and it a speed-up ranging from 0.95% to 4.14% was found. The error for the eigenvalue reached no higher than $10^{-}5$ %, and 0.50 % for the eigenvector. In comparison to the SciPy sparse eigensolver, the Davidson algorithm shows speed-up on matrices with really high diagonal dominance.

Used in the DMRG algorithm, the Davidson code lost to the SciPy sparse eigensolver in speed likely due to the lower diagonal dominance of the Hamiltonian at smaller chain sizes.

For future work, other systems may be investigated to see if the diagonal dominance of the matrix is high enough for the Davidson algorithm to show a speedup. Another idea is to write the whole Davidson algorithm as a CUDA kernel and call it using PyCUDA. A professional-level GPU where double-precision performance is higher may also be used especially for unstable algorithms where exact orthogonality is important.

References

[1] Csaba Nemes, Gergely Barcza, Zoltn Nagy, rs Legeza, and Pter Szolgay. The density matrix renormalization group algorithm on kilo-processor architectures: Implementation and trade-offs. *Computer Physics Communications*, 185(6):1570 – 1581, 2014.

[2] Steven R. White. Density matrix formulation for quantum renormalization groups. *Phys. Rev. Lett.*, 69:2863–2866, Nov 1992.

[3] Kenneth G. Wilson. The renormalization group: Critical phenomena and the kondo problem. *Rev. Mod. Phys.*, 47:773–840, Oct 1975.

[4] André Luiz Malvezzi. An introduction to numerical methods in low-dimensional quantum systems. *Brazilian Journal of Physics*, 33:55 – 72, 03 2003.

[5] James R. Garrison and Ryan V. Mishmash. simple-dmrg/simple-dmrg: Simple dmrg 1.0, November 2017.

[6] U. Schollwöck. The density-matrix renormalization group. *Rev. Mod. Phys.*, 77:259–315, Apr 2005.

[7] P. W. Anderson. New approach to the theory of superexchange interactions. *Phys. Rev.*, 115:2–13, Jul 1959.

[8] M. Kargarian, R. Jafari, and A. Langari. Renormalization of entanglement in the anisotropic heisenberg (xxz) model. *Phys. Rev. A*, 77:032346, Mar 2008.

[9] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[10] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[11] Karl E. Hillesland, Sergey Molinov, and Radek Grzeszczuk. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Trans. Graph.*, 22(3):925–934, July 2003.

[12] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[13] G. Hager, E. Jeckelmann, H. Fehske, and G. Wellein. Parallelization strategies for density matrix renormalization group algorithms on shared-memory systems. *Journal of Computational Physics*, 194(2):795 – 808, 2004.

[14] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[15] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[16] Jean Francois Puget. How to quickly compute the mandelbrot set in python. `https://www.ibm.com/developerworks/community/blogs/jfp/entry/How_To_Compute_Mandelbrodt_Set_Quickly?lang=en`, 2015.

[17] Lev E. Givon, Thomas Unterthiner, N. Benjamin Erichson, David Wei Chiang, Eric Larson, Luke Pfister, Sander Dieleman, Gregory R. Lee, Stefan van der Walt, Bryant Menn, Teodor Mihai Moldovan, Frédéric Bastien, Xing Shi, Jan Schlüter, Brian Thomas, Chris Capdevila, Alex Rubinsteyn, Michael M. Forbes, Jacob Frelinger, Tim Klein, Bruce Merry, Nate Merill, Lars Pastewka, Li Yong Liu, S. Clarkson, Michael Rader, Steve Taylor, Arnaud Bergeron, Nikul H. Ukani, Feng Wang, and Yiyin Zhou. scikit-cuda 0.5.1: a Python interface to GPU-powered libraries, 2015. `http://dx.doi.org/10.5281/zenodo.40565`.

[18] Gregory Lee. Python cuda cffi. `https://github.com/grlee77/python-cuda-cffi.git`, 2018.

[19] John Ian Felismino and Francis N. Paraan. Comparison of python gpu implementations of eigensolvers. volume 36, pages SPP–2018–PB–51, 2018.

[20] Ernest R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics*, 17(1):87 – 94, 1975.

[21] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[22] Bolin Liao. Davidson diagonalization method and its application to electronic structure calculation. 2011.

[23] Roman Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*. PhD thesis, Zurich, ETH, 2002.

[24] Leo GiggleLiu. Jacobi˙davidson. `https://github.com/GiggleLiu/Jacobi_Davidson.git`, 2018.