Efficient Banded Matrix Multiplications

for Quantum Transport Simulation

Semester Project Report v1.0

by Alex Studer

Abstract

The performance of quantum transport simulations using Non-Equilibrium Green's Function (NEGF) formalism depends on using optimized implementations of Basic Linear Algebra Subprograms (BLAS) for matrix multiplications [1]–[3]. Frequently, the matrices involved in such multiplications are banded. Whereas implementations exist for dense and sparse matrix multiplications, e.g., cuBLAS and cuSPARSE, specialized implementations for banded matrix multiplications are lacking. As banded matrices are sparse, using sparse matrix multiplication routines might seem like the optimal choice. However, this may yield suboptimal performance and memory usage on both CPU and GPU architectures. The reason is that the sparsity of banded matrices is structured, versus the assumption of unstructured sparsity used for today's implementations of sparse matrix multiplications. Thus, we propose a BLAS-like algorithm for the multiplication of two banded matrices. This algorithm is based on the algorithm suggested by Benner et al. [4], which assumed only one of the two matrices involved in the operation to be banded. We implement both algorithms for banded-dense and banded-banded matrix multiplications, thus closing our gap of missing implementations for banded matrices. Our implementations use CUDA streams to utilize the GPU of Nvidia efficiently. Our experiments strongly suggest that our optimized implementations can reduce memory usage while maintaining good performance.

1 Introduction

Motivation — Implementations of the Basic Linear Algebra Subprograms (BLAS) specification are widely used to improve the performance of calculations done on computers. There exist vendor-optimized implementations for many hardware platforms, like cuBLAS for NVIDIA Graphical Processing

Units (GPU) or oneMKL for Intel processors (CPU) and non-vendor implementations like OpenBLAS. The original BLAS specification was intended for dense matrices but was expanded with specific routines for *unstructured* sparse matrices in 2002 [5]. The need for optimized operations on sparse matrices can be seen by the many examples of problems in the area of computational fluid dynamics, optimization, structural engineering, and many more [6]. A special case of a sparse matrix is a banded matrix, where all the non-zero entries are aligned around the diagonal. This kind of matrix is particularly common when partial differential equations are approximated for numerical simulations [7], [8] or in model reduction [9]. Especially relevant for this project was the existence of such banded matrices in quantum transport simulations using Non-Equilibrium Green's function (NEGF) formalism [10]. Another indication of the prominence of banded matrices in scientific applications is that the Linear Algebra Package (LAPACK) includes specialized linear system solvers whose system matrix is banded. Even though banded matrices are common, the BLAS specification does not contain a subprogram specifically for banded matrix multiplications. However, using sparse matrix multiplication for banded matrices might yield suboptimal performance and memory usage because it neglects the fact that the sparsity in banded matrices is *structured*.

Related Work — Matam et al. [11] investigated possible implementations of sparse-sparse matrix multiplication (SpGEMM) on modern computer architectures. They focused on banded-banded matrix multiplication as a special case of sparse matrices to investigate how to split the workload between the CPU and GPU. They found that their implementation can increase the performance of sparse matrix multiplication, including banded matrix multiplication, over implementations like cuSPARSE and oneMKL. Multiple works by Benner et al. [4], [9] focused on specifically improving the banded-dense matrix multiplications using GPUs. They showed notable improvements in performance when using GPUs. Another study by Vooturi et al. [12] focused on quasi-band matrix multiplication, a type of matrix that has a large majority of the non-zeros along the diagonals. They found improvements for an SpMM algorithm for quasi-band matrix multiplications in extensive tests using synthetic datasets and real-world datasets from the SuiteSparse Matrix collection [6] (formerly known as the University of Florida Matrix collections). Finally, Hossain et al. [13] investigated how storage schemes based on diagonals and corresponding algorithms for matrix-matrix multiplication could improve such operations involving banded, triangular, and symmetric matrices. We could not find any work specifically looking at the case of banded-banded matrix multiplication and how it can be implemented efficiently.

Objectives — This project aims to address the gap in how banded-banded matrix multiplications can be performance- and memory-efficiently implemented. We implement a BLAS-like routine for banded matrix multiplications with a structure of BLAS Level-3 routine (Equation 1). Our algorithm is based on the algorithm proposed by Benner et al. [4] but modified to account for B being banded instead of dense.

$$C \leftarrow \alpha AB + \beta C \tag{1}$$
$$C \in \mathbb{R}^{m \times n}, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times n}, \quad \alpha, \beta \in \mathbb{R}$$

Report Organization — First, this report describes in Section 2 our algorithm called BdGEMM in detail, showing how it is different from the algorithm it is based on. Section 3 discusses the different implementation options, especially highlighting the motivation to utilize CUDA Streams. We present our experiments and their results in Section 4 and close with our conclusions and suggestions for future work.

2 The BdGEMM Algorithm

This section presents our BLAS-like algorithm for computing banded-banded matrix multiplication. To avoid confusion about which algorithm we refer to throughout the rest of this report, let us first discuss the rationale behind the chosen naming convention. The BLAS Report from 2002 [5] suggested USMM for unstructured sparse times dense matrix multiplication. However, a different naming convention seems to have been adopted for sparse matrix multiplication. cuSPARSE and the Berkeley BLAS library use SpMM for sparse-dense matrix multiplication and SpGEMM for sparse-sparse matrix multiplication. Using a similar naming approach should lead to the least confusion as we consider banded matrix in a context where sparse matrices also arise. The algorithm developed by Benner et al. [4] is intended for banded-dense matrix multiplication, so we refer to it as BdMM. Our algorithm is intended for banded-banded matrix multiplications, so we call it BdGEMM. Throughout this report, we refer to those algorithms only by this naming convention.

BdGEMM is a modified version of BdMM, accounting for the fact that Matrix B is also banded. To understand the BdGEMM and especially what it does differently than BdMM, we first explain the BdMM algorithm and then introduce BdGEMM afterward. For the sake of comparison of what BdGEMM offers over BdMM, in case all the matrices involved are banded, the visualizations of BdMM use only banded matrices.

2.1 Banded-Dense Matrix Multiplication – BdMM

The BdMM algorithm proposed by Benner et al. [4] expresses a BLAS-like kernel of the operation described in Equation 1. It assumes A to be banded, B to be dense, and thus C also to be dense. The algorithm consists of two loops, referred to as the *Outer Loop* and the *Inner Loop*. Those loops are designed to select non-zero blocks, which are then used for multiplication using vendor-optimized kernels.

The outer loop of BdMM groups a fixed number of columns of C and B together to block columns. It then iterates over all these block columns and invokes the inner loop each time with matrix A and the block columns of C and B. Figure 1 illustrates which blocks of A, B, and C are selected in the second iteration for some example matrices and a `block column size` of 3.

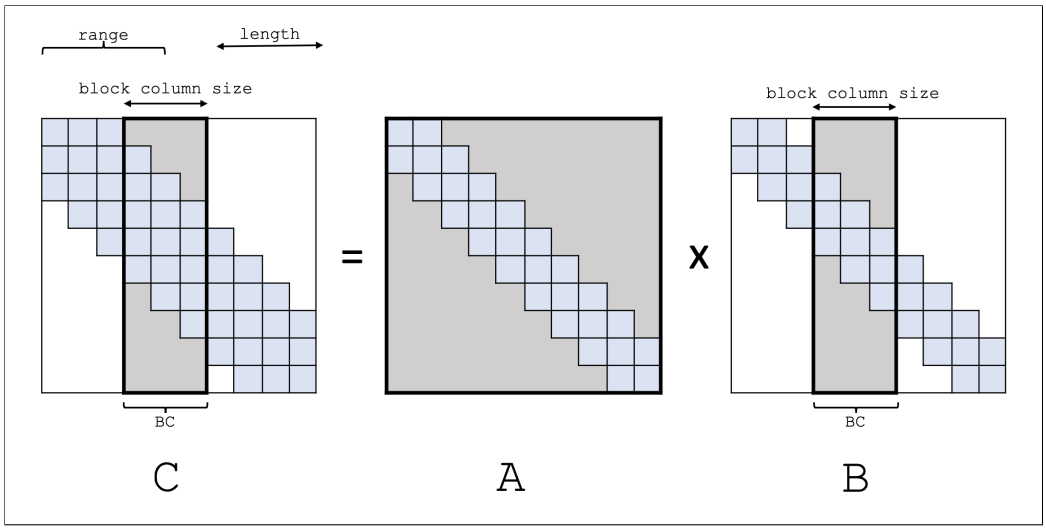

Figure 1: Visualization of the outer loop of the BdMM algorithm

The *Outer Loop*, as described by the pseudocode algorithm 1, starts by setting a `block column size`. It can be tuned for specific hardware or based on the bandwidth of A. After calculating the `number of block columns` based on the `block column size`, the column range `BC` of the first block column is set, and then with every iteration, the column range `BC` is moved by the `block column size`.

The *Inner Loop* (Figure 2, Algorithm 2) iterates over blocks of rows of D. The objective is to reduce the number of zeros involved in the multiplication. To achieve this, the banded structure of A is exploited by further selecting sub-blocks of A. A general banded structure, in combination with a block size smaller than half of the bandwidth, allows for potentially up to three different

```
def BdMM_outer(C,A,ku, kl, B):
    Parameter: C ∈ ℝ^{m×n}, A ∈ ℝ^{m×k}, B ∈ ℝ^{k×n}, ku, kl ∈ ℝ
    Return   : C ∈ ℝ^{m×n}

    /* Determine size of the column block                */
    blk_col_size ← (ku + kl + 1)/2

    /* Determine the number of block columns             */
    num_blocks ← round up(n(B)/blk_col_size)

    /* Determine column range of the first block column  */
    BC ← range(0, blk_col_size)

    /* Iterate over all block columns starting left      */
    for iteration in num_blocks:
        /* Select the subblocks to be invoked in the inner
           loop                                          */
        C[:, BC] = BdMM_inner(C[:, BC], A, ku, kl, B[:, BC])

        /* Adjust block column by shifting the column range
           by blk_col_size                               */
        BC ← range(iter*blk_col_size, (iter+1)*blk_col_size)
```

Algorithm 1: BdMM – Outer Loop

structures in these sub-blocks. As you can see in Figure 3 (c), there can be lower triangular, dense, upper triangular, and zero-only blocks. We assume the selected sub-block in D to be dense (In Figure 2 D is already visualized to be banded for better comparison with BdGEMM later). We do not need to do any operations with the zero-only blocks and can use BLAS kernels for the three other types of blocks. Hence, BdMM translates our banded matrix multiplication into dense and triangular matrix multiplications for which efficient, vendor-optimized BLAS implementations exist. Figure 2 is an example of an inner loop in the second iteration. From the first iteration, C's violet values and red entries already contain some values. The violets already contain the results, whereas the red ones will be added to some product of the sub-blocks of A and B during this second iteration.

The key part of the *Inner Loop* is the selection of the row blocks of A. This process is described in Algorithm 3 with detailed conditions for when a lower triangular, dense, or upper triangular block is available. This calculation is based on knowing which columns are used during each iteration. This positional argument (pos) plus the number of upper (ku) and lower (kl) band diagonals allows us to know exactly the upper and lower limits of the

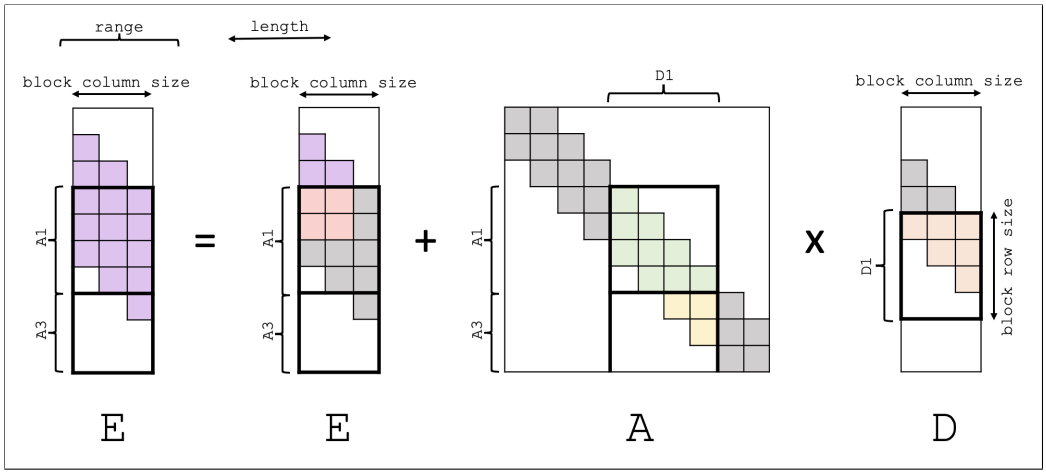

Figure 2: Figure Inner Loop

def `BdMM_inner(`*E,A,ku, kl, D*`):`
Parameter: $E \in \mathbb{R}^{m \times n}, A \in \mathbb{R}^{m \times k}, D \in \mathbb{R}^{k \times n},\ ku, kl \in \mathbb{R}$
Return : $E \in \mathbb{R}^{m \times n}$

`/* Determine size of a row block */`
blk_row_size $\leftarrow (ku + kl + 1)/2$

`/* Determine the number of row blocks columns */`
num_blocks $\leftarrow$ round up$(m(D)/$blk_row_size$)$

`/* Iterate over all block columns starting left */`
for *iter* **in** num_blocks**:**
`/* Determine row ranges */`
D1, A1, A2, A3 $\leftarrow$ `BdMM_slicing(`*iter, ku, kl, m(A), m(D),* blk_row_size)

`/* Select the subblocks to be multiplied */`
E[A1, :] = E[A1, :] + A[A1, D1] $\times$ D[D1, :]
E[A2, :] = E[A2, :] + A[A2, D1] $\times$ D[D1, :]
E[A3, :] = E[A3, :] + A[A1, D1] $\times$ D[D1, :]

Algorithm 2: BdMM – Inner Loop

bands during every iteration. The upper limit is visualized with a red line, and the lower limit is visualized with a violet line in Figure 3 (b). If a range for an upper triangular block exists, the range A1 is calculated based on the upper limit and the row block size. A3 is calculated analogously for lower

triangular blocks. A2 is then the range left between A1 and A3.

```
def BdMM_slicing(iter, ku, kl, m(A), m(D), blk_row_size):
    Return     : D1, A1, A2, A3

    /* Determine position                                    */
    pos ← iter * blk_row_size

    /* Determine band limits                                 */
    band_limit_upper ← max(0, pos - ku)
    band_limit_lower ← min(pos + kl + blk_row_size, m)

    /* D1                                                    */
    D1 = range (pos, min(k, pos+blk_row_size)

    /* If lower triangular matrix                            */
    if pos < ku:
        A1 = range (band_limit_upper, band_limit_upper)
    else:
        A1 = range (band_limit_upper,
         min(band_limit_upper+blk_row_size, m))

    /* If upper triangular matrix                            */
    if (pos + blk_row_size) >= (m − kl − 1):
        A3 = range (band_limit_lower, band_limit_lower)
    else:
        A3 = range (max(band_limit_lower-blk_row_size, A1.stop),
         band_limit_lower)
    A2 = range (A1.stop,A3.start)
```

Algorithm 3: BdMM – Slicing

2.2 Banded-Banded Matrix Multiplication – BdGEMM

To account for the special case of A and B being banded matrices, we propose a modification of the *Outer loop*, shrinking the matrix subsets accessed in every loop iteration. This modification allows the *Inner Loop* to remain as-is. The reduction of block sizes can be easily spotted in a visual comparison of Figures 1 and 4.

The idea behind the shrinking is to reduce the number of zeros involved in computations. In figure 1, the gray area in the selected sub-block of B makes up more than half of the block. This area represents the zero entries. It becomes greater the smaller the bandwidth is compared to the size of the

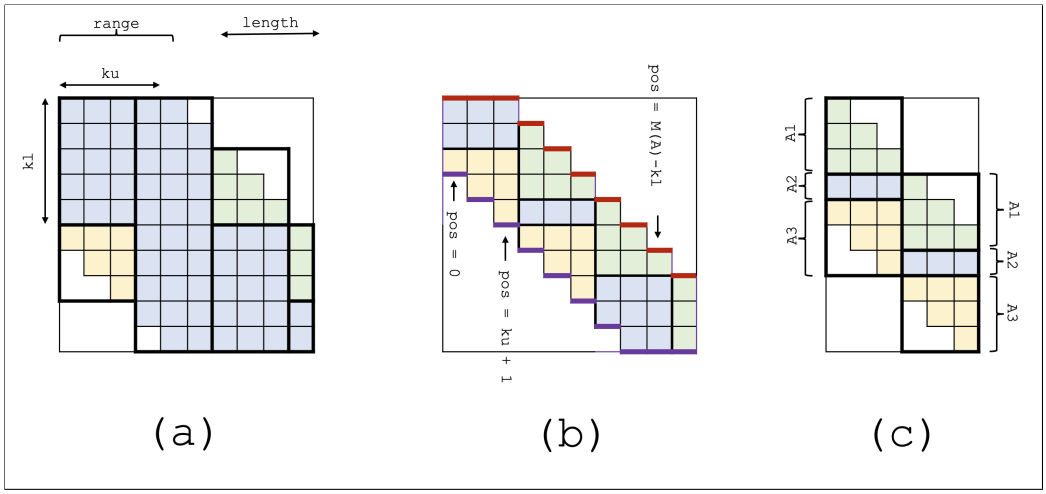

Figure 3: BdMM Slicing Cases

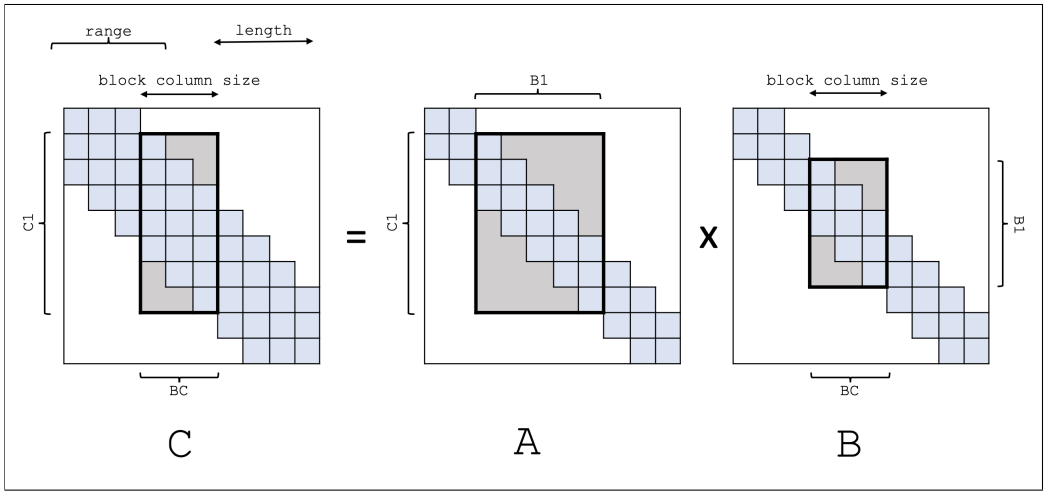

Figure 4: Visualization of the outer loop of the BdGEMM algorithm

matrix B. The unnecessary computation with zero entries can be reduced significantly by simply cutting the top and bottom part of this column block where only zero entries are. The banded matrix B is characterized enough by its dimensions (m, n) and the number of upper diagonals ku and the number of lower diagonals kl to calculate the minimized blocks for each iteration (`iter`) (Equations 2, 3, 4).

$$\texttt{pos} = \texttt{iter} * \texttt{block\_column\_size} \tag{2}$$

$$\texttt{band\_limit\_upper} = \max(0, \texttt{pos} - ku) \tag{3}$$

$$\texttt{band\_limit\_lower} = \min(\texttt{pos} + \texttt{block\_row\_size} + kl, m) \tag{4}$$

Analogously, the column block of C can be shrunk using the same equations 2, 3, 4 and using the characteristics - (m, n), ku, kl - corresponding to C. ku and kl of C are not known *a priori* but can be calculated as the sum of the bands of A and B (Equations 5, 6).

$$ku_C = ku_A + ku_B \tag{5}$$

$$kl_C = kl_A + kl_B \tag{6}$$

Changing the block of A also changes the relative split between ku and kl. For the *Inner Loop* to continue working properly, we additionally need to calculate adjusted ku and kl of the sub-block of A used by the *Inner Loop* (Equations 7, 8)

$$ku = ku_A + (C1.start - B1.start) \tag{7}$$

$$kl = kl_A - (C1.start - B1.start) \tag{8}$$

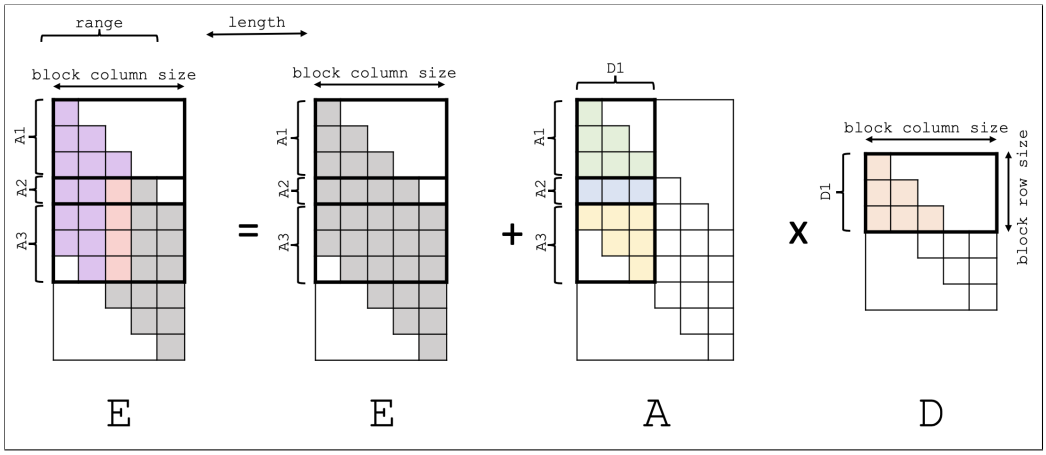

Figure 5: Visualization of a typical inner loop of the BdGEMM algorithm

def `BdGEMM_outer(`*C,A,ku_A,kl_A,B,ku_B,kl_B*`):`
Parameter: $C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$
Return : $C \in \mathbb{R}^{m \times n}$

`/* Determin upper and lower diagonals of C */`
ku_C = ku_A + ku_B kl_C = kl_A + kl_B

`/* Determine size of the column block */`
blk_col_size $\leftarrow (ku + kl + 1)/2$

`/* Determine the number of block columns */`
num_blocks $\leftarrow$ round up$(m(D)/$blk_col_size$)$

`/* Determine column range of the first block column */`
BC $\leftarrow$ range(0, blk_col_size)

`/* Iterate over all block columns starting left */`
for *iteration* **in** num_blocks**:**

`/* Shrink blocks to bandwidth */`
C1 = range(max(0, C1.start - ku_C), min(n(C), C1.stop + kl_C))
B1 = range(max(0, B1.start - ku_B), min(m(B), B1.stop + kl_B))

`/* Adjust number of upper and lower bands matching subblocks */`
ku = ku_A + (C1x.start - B1x.start)
kl = kl_A - (C1x.start - B1x.start)

`/* Select the subblocks to be invoked in the inner loop */`
C[C1, BC] = `BdGEMM_inner(`*C[C1, BC], A[C1, B1], ku, kl, B[B1, BC]*`)`

`/* Adjust block column by shifting the column range by blk_col_size */`
BC $\leftarrow$ range(iter*blk_col_size, (iter+1)*blk_col_size)

Algorithm 4: BdGEMM – Outer Loop

3 Implementations

We present our GPU-optimized implementation of BdGEMM in this section. The expensive computations are all done on an NVIDIA GPU, making heavy use of cuBLAS routines. We wrote our implementation using Python and

utilized CuPy to access CUDA functionality.

3.1 Implementation of BdGEMM$_{\textbf{blocking}}$

Our implementation of BdGEMM for NVIDIA GPUs utilizes CUDA Streams to move data from the host to the GPU device and back asynchronously. NVIDIA supports Host-to-Device (H2D) and Device-to-Host (D2H) operations to be executed parallel to computation on the device for GPUs with Pascal architecture. By calculating the blocks of the next iteration in advance, we can build a queue for the blocks to be computed. This essentially enables us to calculate the products of the blocks in the current iteration while the results of the last iteration are moved back to the host, and the blocks for the next iteration are moved to the device. This should also only store data on the GPU, which is in use for some computation. In theory, this should reduce the utilization of GPU memory. As this version utilizes streams but blocks the next computation to run to mitigate race conditions, it does not make use of the potential parallelism offered by CUDA Streams yet.

3.2 Reference Implementations

We implemented two other algorithms for banded matrix multiplications to assess the performance of BdGEMM$_{blocking}$. First, we implemented BdMM, where all three matrices C, A, and B are copied to the GPU at the beginning. Then, the algorithm computes everything before moving the result back. We will refer to this approach as naive copy. BdMM$_{naiveCopy}$ is a replication of the algorithm presented by Benner et al. [4]. We also implemented a naive copy of BdGEMM, referred to as BdGEMM$_{naiveCopy}$.

4 Experimental Evaluation

This section evaluates the implementation BdGEMM$_{blocking}$ and compares its performance and memory efficiency to the BdMM and BdGEMM naive copy variants. We also compare it to simply using a CSR storage scheme and sparse matrix multiplication libraries for the multiplication of two banded matrices.

4.1 Experimental Platform

We use an NVIDIA Tesla P100 Graphical Processing Unit connected to a host system with two Intel Xeon E5-2860v4 CPUs. The host system has 256 GB RAM available. It runs CentOS Linux version 7 as its operating system. The table below gives more detailed information about the hardware used.

GPU	NVIDIA P100
CUDA Cores	3582
Frequency (MHz)	1190
GPU Memory (GB)	16280 MiB
CUDA Version	11.0
CPU	2x Intel Xeon E5-2860v4
CPU cores	14 (28)
CPU Frequency (GHz)	2.40 (3.30)
L3 Cache (MB)	35
Memory (GB)	256

Table 1: Hardware Details

4.2 Test Cases

The chosen test cases should give a general indication of the efficiency of BdGEMM. They are not designed to benchmark the algorithms' implementation exhaustively.

- **Experiment 1**: We use identical square matrices with symmetrical bands for A and B. The bandwidth makes up 2% of the matrix size, and we chose half of the bandwidth as our block size. The only thing we vary is the size of the matrices. We will measure the median runtime over 20 runs and the peak GPU memory usage during those runs. See Tables 2 and 3.
- **Experiment 2**: We try to evaluate matrix size limits.
- **Experiment 3**: We compare runtime of the naive copy variants if B would be dense.

4.3 Results

A conclusion we can draw from Experiment 1 is that using CSR and cuSPARSE directly for the matrix multiplication yields the fastest runtime but

Matrix Dimensions	2500	5000	10000
CSR	0.002193559	0.007147495	0.065839054
$\text{BdMM}_{\text{naiveCopy}}$	6.707423927	6.447523147	6.44057644
$\text{BdGEMM}_{\text{naiveCopy}}$	0.298732841	0.219333407	0.655851869
$\text{BdGEMM}_{\text{blocking}}$	0.299604353	0.364729015	0.518981477

Table 2: Runtime in seconds for different matrix dimensions

Matrix Dimensions	2500	5000	10000
CSR	303.91 MiB	1.9 GiB	13.49 GiB
$\text{BdMM}_{\text{naiveCopy}}$	151.21 MiB	580.47 MiB	2.24 GiB
$\text{BdGEMM}_{\text{naiveCopy}}$	151.21 MiB	580.47 MiB	2.24 GiB
$\text{BdGEMM}_{\text{blocking}}$	8.57 MiB	9.46 MiB	21.97 MiB

Table 3: Peak GPU Memory Usage for different matrix dimensions

uses enormous amounts of GPU memory. In Experiment 2, we found that by increasing the matrix sizes further, the CSR version will crash when the matrix becomes slightly bigger than 10000. It would need more than the 16 GiB available. We could not directly locate such a size issue for $\text{BdGEMM}_{blocking}$. This makes sense, as its GPU memory usage depends on the block sizes and not the matrices' sizes. The limiting factor of $\text{BdGEMM}_{blocking}$ is thus no longer the GPU memory by the memory available on the host system.

Comparing $\text{BdGEMM}_{naiveCopy}$ and $\text{BdMM}_{naiveCopy}$ in Experiment 1, we find that for the multiplication of two banded matrices, $\text{BdGEMM}_{naiveCopy}$ is significantly faster. This is reasonable, as $textBdMM_{naiveCopy}$ multiplies plenty of extraneous zeros. Both need the same amount of GPU memory as they move the A, B, and C as a whole to the GPU for the computations. During Experiment 3, we found that the runtime of $\text{BdGEMM}_{naiveCopy}$ and $\text{BdMM}_{naiveCopy}$ converged, as now $\text{BdGEMM}_{naiveCopy}$ does not offer any further optimization.

Conclusion and future work

In this work, we design a more efficient implementation of a BLAS-style subroutine for band matrix multiplication compared to implementations optimized for unstructured sparse matrices. We develop BdGEMM for banded-banded matrix multiplication and implement an optimized version using CUDA Streams ($\text{BdGEMM}_{blocking}$). Our Experiments show the expected effects on performance and memory usage. BdGEMM is faster than BdMM

when both A and B are banded. By using CUDA Streams and queuing, we significantly reduce GPU memory utilization. This removes GPU memory as a potential bottleneck compared to sparse matrix multiplications with a CSR storage scheme.

The host memory usage has not been reduced, as all our implementations use a dense matrix storage scheme. As our algorithm works on blocks, we will be able to swap out the underlying block storage scheme in the next step. Using instead a more efficient storage scheme will likely reduce the memory capacities used. As the BdGEMM algorithm was motivated by the need for performance improvements in quantum transport simulations, we plan to integrate it into a quantum transport simulator. This allows us to validate our algorithm's performance and memory usage in a real-world setting.

References

[1] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "Optimizing the data movement in quantum transport simulations via data-centric parallel programming," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, New York, NY, USA: Association for Computing Machinery, Nov. 17, 2019, pp. 1–17, ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356200. [Online]. Available: https://doi.org/10.1145/3295500.3356200 (visited on 01/11/2024).

[2] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, New York, NY, USA: Association for Computing Machinery, Nov. 17, 2019, pp. 1–13, ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3357156. [Online]. Available: https://doi.org/10.1145/3295500.3357156 (visited on 01/11/2024).

[3] M. Calderara, S. Brück, A. Pedersen, M. H. Bani-Hashemian, J. VandeVondele, and M. Luisier, "Pushing back the limit of ab-initio quantum transport simulations on hybrid supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–12, ISBN: 978-1-4503-

3723-6. DOI: 10.1145/2807591.2807673. [Online]. Available: https://doi.org/10.1145/2807591.2807673 (visited on 01/11/2024).

[4] P. Benner, A. Remon, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Orti, "Accelerating the general band matrix multiplication using graphics processors," in *2014 XL Latin American Computing Conference (CLEI)*, Montevideo, Uruguay: IEEE, Sep. 2014, pp. 1–7, ISBN: 978-1-4799-6130-6. DOI: 10.1109/CLEI.2014.6965142. [Online]. Available: http://ieeexplore.ieee.org/document/6965142/ (visited on 12/11/2023).

[5] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 239–267, Jun. 1, 2002, ISSN: 0098-3500. DOI: 10.1145/567806.567810. [Online]. Available: https://dl.acm.org/doi/10.1145/567806.567810 (visited on 01/11/2024).

[6] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 1:1–1:25, Dec. 7, 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663 (visited on 12/31/2023).

[7] A. J. Keeping, "Band Matrices Arising from Finite Difference Approximations to a Third Order Partial Differential Equation," *SIAM Journal on Numerical Analysis*, vol. 7, no. 1, pp. 142–156, 1970, ISSN: 0036-1429. JSTOR: 2949589. [Online]. Available: https://www.jstor.org/stable/2949589 (visited on 12/31/2023).

[8] M. J. Gander, S. Loisel, and D. B. Szyld, "An Optimal Block Iterative Method and Preconditioner for Banded Matrices with Applications to PDEs on Irregular Domains," *SIAM Journal on Matrix Analysis and Applications*, vol. 33, no. 2, pp. 653–680, Jan. 2012, ISSN: 0895-4798, 1095-7162. DOI: 10.1137/100796194. [Online]. Available: http://epubs.siam.org/doi/10.1137/100796194 (visited on 12/31/2023).

[9] P. Benner, E. Dufrechou, P. Ezzatti, P. Igounet, E. S. Quintana-Ortí, and A. Remón, "Accelerating Band Linear Algebra Operations on GPUs with Application in Model Reduction," in *Computational Science and Its Applications – ICCSA 2014*, B. Murgante, S. Misra, A. M. A. C. Rocha, *et al.*, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 386–400, ISBN: 978-3-319-09153-2. DOI: 10.1007/978-3-319-09153-2_29.

[10] L. Deuschle, J. Backman, M. Luisier, and J. Cao, "Ab initio Self-consistent GW Calculations in Non-Equilibrium Devices: Auger Recombination and Electron-Electron Scattering," in *2023 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, Sep. 2023, pp. 297–300. DOI: 10.23919/SISPAD57422.2023.10319647. [Online]. Available: https://ieeexplore.ieee.org/document/10319647 (visited on 01/11/2024).

[11] K. Matam, S. R. Krishna Bharadwaj Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *2012 19th International Conference on High Performance Computing*, Pune, India: IEEE, Dec. 2012, pp. 1–10, ISBN: 978-1-4673-2371-0 978-1-4673-2372-7 978-1-4673-2370-3. DOI: 10.1109/HiPC.2012.6507483. [Online]. Available: http://ieeexplore.ieee.org/document/6507483/ (visited on 12/18/2023).

[12] D. T. Vooturi and K. Kothapalli, "Parallel Algorithm for Quasi-Band Matrix-Matrix Multiplication," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, Eds., vol. 9573, Cham: Springer International Publishing, 2016, pp. 106–115, ISBN: 978-3-319-32148-6 978-3-319-32149-3. DOI: 10.1007/978-3-319-32149-3_11. [Online]. Available: http://link.springer.com/10.1007/978-3-319-32149-3_11 (visited on 12/18/2023).

[13] S. Hossain and M. S. Mahmud, "On Computing with Diagonally Structured Matrices," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA: IEEE, Sep. 2019, pp. 1–6, ISBN: 978-1-72815-020-8. DOI: 10.1109/HPEC.2019.8916325. [Online]. Available: https://ieeexplore.ieee.org/document/8916325/ (visited on 12/18/2023).