2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) | 979-8-3315-3047-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/MCSoC64144.2024.00064

Performance Evaluation of CUDA Parallel Matrix Multiplication using Julia and C++

Robertus Hudi
Departement of Informatics
Universitas Pelita Harapan
Tangerang, Indonesia
robertus.hudi@uph.edu

Mikael Silvano
Departement of Informatics
Universitas Pelita Harapan
Tangerang, Indonesia
01082230019@student.uph.edu

Kennedy Suganto
Departement of Informatics
Universitas Pelita Harapan
Tangerang, Indonesia
01082220002@student.uph.edu

***Abstract*— Compute Unified Device Architecture (CUDA) was developed as a GPU parallel programming platform and API, primarily designed for use with C/C++. Over the years, fundamental linear algebra functionalities on CUDA have reached a mature state, and many of these are now accessible on CUDA's GitHub repository. As other high-level programming languages have begun incorporating CUDA-compatible methods into their libraries, the Julia Programming Language introduced CUDA support in 2021, aiming to offer an abstraction level similar to that of C implementations. However, research has shown that Julia's linear algebra computations—despite leveraging CUDA for parallelization and computational reduction—have yet to match the execution speed achieved by C implementations. This study uses matrix multiplication as a representative linear algebra computation, given its well-optimized CUDA kernel. Outputs of the study include an NSight report file and an SQLite database, which are analysed using NVIDIA Nsight Systems to assess each kernel's runtime and memory usage for performance evaluation. Findings indicate that Julia's CUDA kernel invocation has a high runtime overhead, growing at a rate of $O(n^2)$, which presents a bottleneck when performing high-throughput computations on square binary matrices. This paper suggests that resolving this issue may involve developing a custom CUDA kernel in Julia that employs a more efficient reduction technique to reduce overhead and enhance performance.**

Keywords—CUDA, C/C++, Julia, Performance, Matrix

I. Introduction

General Purpose Graphics Processing Unit (GPGPU) has been developed for many programming languages throughout the year. C/C++ was the first native programming language to be optimized for Compute Unified Device Architecture (CUDA) Library for NVIDIA GPU Computing. CUDA enables programmers to utilize GPU power to compute for example, linear algebra calculation (from basic calculation like vector addition, matrix multiplication, single value decomposition, etc., to case-specific computation of Conjugate Gradient and PHD Algorithm); as it becomes the standard of CUDA programming implementation, more High-Level languages such as Python and MATLAB enable instructions and calculations to run with CUDA. Julia is one of the Higher-Level programming languages that would be used to compute more statistical data. For this research, matrix multiplication implementation on higher-level programming language (Julia) and the performance will be compared to well established programming language for CUDA, namely C++. Although matrix multiplication code is already provided in cuda-samples officially released by NVIDIA, the provided comments on the code file suggests that it should not be used to profile GPU performance. Thus, this research uses basic implementation of matrix multiplication algorithms for CUDA, which is also the case for Julia, where the kernel definition includes naïve multiplication and addition for each element on the resultant matrix. Since the code in Julia does not specify and the indexing is done automatically, monitored variables will be focused on running time for each kernel and memory checking result. This research's purpose is to explore the implementation of baseline matrix multiplication algorithm, using C++ and Julia, then analyze the all the outcome to determine the problem – especially for the implementation in Julia.

II. Methodology

Matrix multiplication is a binary operation that produces a matrix from two matrices. Matrix multiplication denoted by C is a term of multiplication operation that produces a new matrix from two matrices denoted A and B, where i, j, and k are the elements of the matrices [12]. Since it is amongst the basic concepts in linear algebra, it is often used to test performance of General Purpose GPU (GPGPU) The following equation is the equation of matrix multiplication:

$$(C)ij = \sum_{k=1}^{m} A_{ik} B_{kj} \quad (1)$$

C = Product of matrix multiplication

i, j = indices representing the row and column positions in C

k = summation index, iterating over the inner dimension

m = denotation of the same column of matrices

CPU and GPU are connected by a peripheral component interconnect-express bus, in which the GPU is called the host, and the GPU is called device. This system is defined as a heterogeneous system, in such that the CPU is responsible for working logic control and serial computation and the GPU acts as a co-processor that performs parallel computational tasks. CUDA helps GPU by providing a programming environment and organizes threads into three different levels, which are grid, block, and thread and employs a multi-level memory structure [12]. Each thread in CUDA has their own program counter, registers, and global memory as a shared memory address. Threads within the same block also have a shared memory that is more limited in size compared to the global memory and executes the same instruction in parallel, conversely threads execution could also diverge to run the instruction in serial until the divergent section is completed, and afterward it will run in parallel until the next divergence. Block in CUDA has the capability to hold at most 512 or 1024 threads each [5].

2771-3075/24/$31.00 ©2024 IEEE
DOI 10.1109/MCSoC64144.2024.00064

Kernel is the unit of work of a main program that is running on the host computer then offloaded to the GPU of the device. Kernel in CUDA requires the dimensions of the grid, the dimensions of the blocks, and the kernel function [5]. The advantages properties of CUDA make it useful for matrix multiplication, which could handle a large set of data to calculate [12]. – this section will explain the methodology to conduct the evaluation of implementation in Julia and C

A. *CUDA Matrix Multiplication on C/C++*

CUDA Matrix Multiplication takes advantages of the massively parallel architecture of modern GPUs, allowing simultaneous computation of multiple elements of the resulting matrix. In this experiment, the comparison between two language is conducted using the baseline algorithm of Matrix Multiplication on C/C++, thus the optimization is done as an example of how far the code in C++ could stretch in term of manipulating memory in CUDA Architecture.

Our implementation of CUDA matrix multiplication in C/C++ is using baseline algorithm [5]. This naïve algorithm for matrix multiplication only maps each component of the resulting matrix into a thread, thus, the indexing for each thread will be based on first matrix's row and second matrix's column, those components are mapped into `blockIdx` (location of a block within the grid) and `threadIdx` (a structure that gives the location of a thread within its own block) as `blockDim` (the dimensions of the blocks) is set into two-dimensional, according to the matrices dimension. This naïve algorithm implementation can be seen in Figure 1:

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width) return;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
                          B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

Fig. 1. Naïve algorithm of CUDA matrix multiplication in C++ [5]

Kernel functions are specified by declaring `__global__` in the code to launch these functions in the GPU. Kernel functions act as the entry point for GPU computation. Kernel functions launching in the GPU creates a grid of thread blocks in which the blocks are queued to run on the GPU multiprocessors as they become available to be completed. Figure 1 shows that `MatMulKernel global` function is launched and concurrently passing `d_A`, `d_B`, and `d_C` as the parameter [5]. `float Cvalue = 0` declares a register to hold float value of the matrix multiplication. `int row` and `int col` is used to determine the number of row and columns within the matrix. The if statement is implemented to terminates the threads if the row or column of the matrix is outside the bound of the product matrix. The for loop shown in the baseline algorithm computes the entries of row of matrix A and column of matrix B to be accumulated in `Cvalue`. Device's global memory stores the matrix A and B in row major order, in which it is stored as a one-dimensional-array. Therefore, to find the starting index of the *i*th row, we need to compute $i \times$ width of A, and then add *j* to go to the *j*th entry in that particular row. At last, the last line copies the product to the appropriate element of the product matrix C/C++ in the device's global memory [5]. The parallel matrix multiplication shown in Figure 1 takes the matrix sizes as an input argument. The algorithm will divide the matrices dimension into 16 blocks and employ $x \times y$ threads to perform the matrix multiplication simultaneously. Time taken to complete the matrix multiplication is shown after completing the computation [12].

B. *CUDA Matrix Multiplication on Julia*

Implementation of matrix multiplication in Julia has recently been developed by LAPACK for linear algebra functions and SuiteSparse for sparse matrix factorizations call function [17]. Naïve implementation is manually activated by command to represent the matrix multiplication kernel. In this research, Julia implementation could be explained by looking at Figure 2 below.

```
# Simple kernel for matrix multiplication
@kernel function matmul_kernel!(a, b, c)
    i, j = @index(Global, NTuple)

    # creating a temporary sum variable for matrix multiplication
    tmp_sum = zero(eltype(c))
    for k = 1:size(a)[2]
        tmp_sum += a[i,k] * b[k, j]
    end

    c[i,j] = tmp_sum
end
```

Fig. 2. Matrix multiplication in Julia

Furthermore, to uphold the integrity of the research, identical matrix data is used across both the Julia and C/C++ experiments. Where variable `a` and `b` in Figure 2 are utilized to store global float value, extracted and passed from the same csv file to store matrix data used in C/C++ implementation. Thus, variable `c` is used to store the result of multiplication. This approach ensures that any observed differences in performance or outcomes can be attributed solely to the programming languages or implementations, rather than to variations in the input data. Julia owns an automatic parallel programming and has OPENMP-like multithreading as a wrapper for the portable message passing system MPI. Julia also owns main implementation of message passing for distributed-memory systems contained in the `Distributed` module as part of the standard library. `Threads.@threads` macro is used to implement main multithreading in Julia, in which it parallelize a for-loop to run with multiple threads. GPU offloading in Julia utilizes `KernelAbstractions.jl` (KA.jl), which is a package in Julia that supports CUDA, Intel, and AMD. KA.jl is designed purposefully for GPU programming and concentrates in perfomance portability [15].

C. *Profiling with Nsight Systems*

Extensions of the program composed with C++ and Julia are ".cu" and ".jl", thus, generating the report and database information for the code. Nsight systems is a system wide performance analysis tool designed to visualize application algorithms. Nsight Systems 2023.2.1 (GUI Version) is used on this experiment to get necessary information for analysis regarding CUDA Kernel and API on 2 programs, composed with Julia and C++. Following are components in performance information tables generated in a form of nsys-rep file that we utilize in this research:

a. CUDA GPU Kernel Summary
   CUDA GPU Kernel Summary displays information about the CUDA Kernels executed on the GPU. It

includes details such as time, total time, instances, avg, med, min, max, StdDev, and Name.

b. CUDA GPU MemOps Summary (by size)
CUDA GPU MemOps Summary (by size) provides insight such as total, count, avg, med, min, max, StdDev, and operation.

c. CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Summary (Kernels/MemOps) shows the information about the GPU kernel execution and memory operations which is time, total time, instances, avg, med, min, max, StdDev, Category, and Operation.

d. CUDA Summary (API/Kernels/MemOps)
CUDA Summary consolidates the key performance metrics from the API, GPU kernel, and memory operation summary such as time, total time, instances, Avg, Med, Min, Max, StdDev, Category, and Operation.

III. IMPLEMENTATION

A. *Machine Specification and Environment*

Profiling in NVIDIA GPU using CUDA must meet some hardware requirement, thus in this experiment, profiling is conducted using following machine specification:

- CPU: Intel(R) Core™ i5-14600K, 5.3 GHz
- GPU: NVIDIA GeForce RTX 4070 Super GPU (Process Size: 5nm; TDP: 220w; FP32: 35.48 TFLOPS; VRAM: 12 GB)
- CUDA Version: 12.5
- Memory: 32GB DDR5 RAM
- SSD: 1TB (NVMe)
- OS: Ubuntu 22.02 (Standalone)

B. *Generating Nsys-Rep and Performance Collection*

Report files (nsys-rep) and execution information data file (sqlite):

- Step 1: Compose C/C++ and Julia matrix multiplication code in CUDA file format (.cu)
- Step 2: Compile and run the code file using NVCC compiler to get an executable file for profiling.
- Step 3: Generate an integer matrices of a specific element value and size range (simplest method is using square matrices of size $2^n \times 2^n$, with n ranging from 1 until memory limit)
- Step 4: Conduct profiling which will generate `report.nsys-rep` file:

```
nsys profile -o report --stats=true ./
            <program_name>.exe
```

- Step 5: Open the created `nsys-rep` file using NVIDIA Nsight System, which sample iss shown on Figure 3 below.

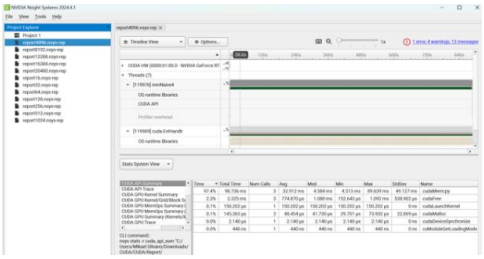

Fig. 3. Nsight System 2023.2.1

- Step 6: Analyse the data of execution time and memory consumption data of each kernel and for the whole application process to be put in a table.

IV. RESULT AND EVALUATION

A. *Results on C/C++*

Based on the generated results from section III.B., these table of running time is obtained:

Table 4.1. C/C++ Memory Consumption (MB)

Size (N)	Memory Consumption (MB)
8	0.001
16	0.003
32	0.012
64	0.049
128	0.197
256	0.786
512	3.146
1024	12.583
2048	50.331
4096	201.327
8192	201.327
12288	201.327
16384	201.327
20480	201.327

Since running test scenarios cover matrix size of 2^N where N is incremented for each iteration of testing, matrix size as input dataset would exponentially. However, since parallel implementation of the algorithm in this experiment uses shared memory on GPU with direct memory allocation, the mapping of the data would follow the CUDA memory hierarchy, thus, 3 matrices would be mapped according to 3 dimensional thread memory. Observation from memory consumption chart shown on Figure 4 below shows that significant increases occur on matrix size $2^N = 4096$, due to this mapping technique. Furthermore, starting from matrix size of 4096, memory consumption has gone stagnant – while result assertion shows a match with the correct solution, due to operations on thread memory mapping with parallel arithmetic instructions occupied maximum amount off thread for each block could be the strongest reason for the stagnancy.

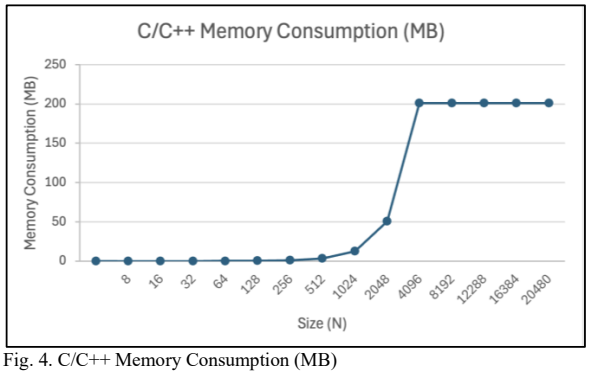

Fig. 4. C/C++ Memory Consumption (MB)

While memory consumption has an unusual increase on matrix with $2^N = 4096$, running time results from Table 4.2. below shows similar result – as there is a big gap between the running time of the kernel-only profiling and total time profiling – which a peak incremental runtime occurs where the on 2^N=4096 as well, reaching over twice the previous running time where 2^N= 2048.

Table 4.2. C/C++ CUDA Performance

Size (N)	Runtime - Kernel (Sec)	Runtime - Total (Sec)
128	0.000007521	0.234264
256	0.000027552	0.266145
512	0.000132641	0.49072
1024	0.000984227	4.753158
2048	0.007611317	32.014082
4096	0.071870155	890.365142
8192	0.071802618	677.5494
12288	0.072057951	630.984
16384	0.071846264	622.6472
20480	0.071820306	603.98

Parallel CUDA matrix multiplication algorithm runs efficiently and opt with big data as the input, thus, slower for smaller data as the result of CUDA creating idles threads (only 100 and 400 threads compute out of 256 and 1024 threads), in which only several threads perform the calculation, and the rest will be idle. The unused created threads create an overhead of controlling them. These idle threads will only be utilized when there is big data to be computed by the GPU and all threads in the blocks run in parallel [12]. Total running time is also considered to find the limit of the algorithm, as it shows similar behavior between kernel running time and memory consumption – where both reach stability from 2^N = 4096 until 2^N = 20480 as both instructions are calculated inside the GPU shared memory.

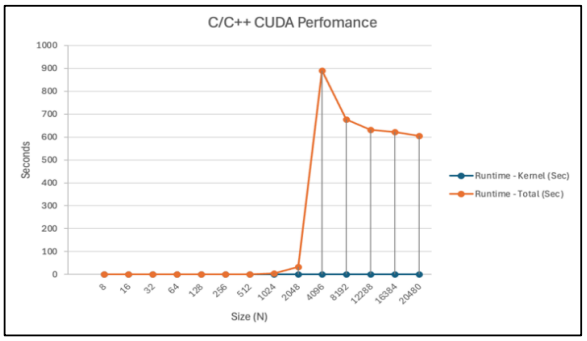

Fig. 5. C/C++ CUDA Performance

The results on C++ program – although using the baseline algorithm with simple row and column reduction technique – shows that it could handle matrix multiplication operation until the size is close to 2^N = 32768. This experiment pushes the matrix dataset to the limit until it could not be handled by the peripherals – as the data transfer from CPU to GPU is also used as consideration of running time – to find the actual capability of the baseline algorithm to handle this matter.

B. Results on Julia

Based on the generated results from section III.B., these table of running time is obtained:

Table 4.3. Julia Memory Consumption (MB)

Size (N)	Memory Consumption (MB)
8	0.003
16	0.008
32	0.033
64	0.131
128	0.524
256	2.097
512	8.388
1024	33.555
2048	134.217
4096	536.871
8192	2147.484
12288	4831.839

CUDA implantation on Julia results show that it could not process due to the memory limitations on mapping variables and resources needed to GPU shared memory, since there is no option for CUDA implementation on Julia to play around with pointer variables, especially on the device side. The impact of this limitation proves that the memory consumption – unlike C/C++ implementation – does not hit stability at any point off the dataset size. The profiling data shows a considerable increase of memory consumption from 2^N = 2048 – slightly earlier than the C/C++ implementation – then increases (approx.) quadratically when the matrix size hits 2^N = 4096.

Result extracted from the memory consumption profiling also provides several insights to the running time performance analysis as for small size of matrix size, Julia total running time is significantly slower than C/C++ implementation. However, it shows more consistency for even until matrix size of 2^N = 2048, while the massive amount of increase is not detected for up until the matrix size reaches 2^N = 2048.

Table 4.4. Julia CUDA Performance

Size (N)	Runtime - Kernel (Sec)	Runtime - Total (Sec)
128	0.000037728	8.965
256	0.000078433	8.905
512	0.00057853	8.786
1024	0.004018382	8.82
2048	0.031514411	8.711
4096	0.257320488	10.05
8192	2.058869684	18.81
12288	30.82358426	55.91

On the other hand, CUDA kernel performance on Julia is proved to be more efficient on the smaller size of matrices. A huge running time result is shown started from 2^N = 8192 and right after that, matrix size of 2^N = 12288 shows more than

10× increment on Kernel running time, 4× on total running time, before it reaches out of memory state.

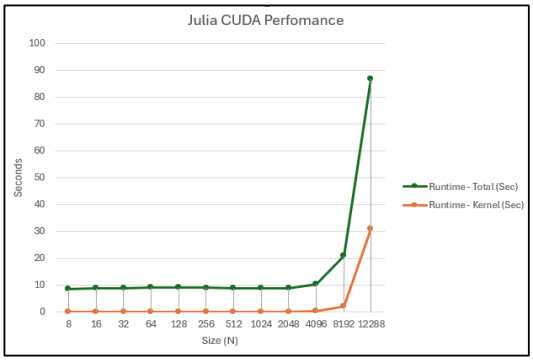

Fig. 6. Julia CUDA Performance

C. *Comparison Analysis*

According to the obtained data from each implementation, a handful of notable results difference could be taken for conclusion. First, for both implementation, peak increment of both memory consumption and running time occurs where matrix size hits $2^N = 4096$; Second, while C/C++ implementation shows stable profiling result on bigger matrix size ($2^N \geq 4096$), implementation on Julia shows otherwise, it holds memory copy transition steadier, most probably because the CUDA kernel that does not offer variety on its memory allocation; Third, implementation on Julia reaches memory limit problem earlier than C/C++ implementation, even though the algorithm is proven more stable on Julia, but du to its kernel nature, it would hit the limitations of data transfer (memory copy) between CPU and GPU.

V. Conclusion

Whilst CUDA is fundamentally optimized for C/C++, development for the Julia platform could be a solution on kernel-level GPGPU usage to higher level programming language. Optimized CUDA libraries on Julia is sure still far from satisfying, as several parts of the profiling result shows excellent stability over small-sized dataset, it is yet to come close to the performance shown on C/C++ implementation. While utilization of dynamic parallelization on C/C++ implementation might solve the sudden peak profiling result on matrix $2^N = 4096$, suggestion to optimize performance on Julia also emerges from this experiment's analysis: the need ability to redesign the kernel.

Acknowledgment

The author would like to thank Universitas Pelita Harapan and the Institute of Research and Community Services (LPPM) UPH for their support towards this research, with research number P-92-SISTech-VII/2023, as well as to all parties involved in the writing of this research.

References

[1] NVIDIA, "NVHPC-Developer-Tools", pp. 1-16. Accessed: Nov. 5, 2024. [Online]. Available: https://developer.nvidia.com/hpc-sdk

[2] N. Yu Ilyasova, V. A. Shikhevich, A. S. Shirokanev, "CUDA parallel programming technology application for analysis of big biomedical data based on computation of effectiveness features" *IOP Publishing Ltd*, pp. 1-8, 2019. doi: 10.1088/1742-6596/1368/5/052006

[3] JULIA, "Julia Programming Language". Accessed: Nov. 5, 2024. [Online]. Available: https://julia.org/

[4] NVIDIA, "Nsight Compute Blogs". Accessed: Nov. 5, 2024. [Online]. Available: https://developer.nvidia.com/tools-overview/nsight-compute/get-started

[5] Robert Hochberg, "Matrix Multiplication with CUDA – A basic introduction to the CUDA programming model" *Semantic Scholar*, pp. 1-44, 2012. Accessed: Nov. 5, 2024. [Online]. Available: https://www.semanticscholar.org/paper/Matrix-Multiplication-with-CUDA-%E2%80%93-A-basic-to-the-Hochberg/8b44ec5cf1785b098cc0ec437ca96c909f231149

[6] Ali Olow Jimale, Fakhitah Ridzuan, Wan Mohd Nazmee Wan Zainon, "Square Matrix Multiplication Using CUDA on GP-GU" *Elsevier*, pp. 1-8, 2020. Accessed: Nov. 5, 2024. [Online]. Available: https://doi.org/10.1016/j.procs.2019.11.138

[7] Nathan Bell, Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA" *NVIDIA*, pp. 1-32, 2009. Accessed: Nov. 5, 2024. [Online]. Available: https://research.nvidia.com/publication/2008-12_efficient-sparse-matrix-vector-multiplication-cuda

[8] Zhibin Huang, Ning Ma, Shaojun Wang, Yu Peng, "GPU computing performance analysis on matrix multiplication" *Semantic Scholar*, pp. 1-6, 2019. Accessed: Nov. 5, 2024. [Online]. Available: https://doi.org/10.1049/joe.2018.9178

[9] Murni, T. Handhika, "Performance Analysis of CUDA by Implementation of Hypergraph Partitioning for Parallelyzing Sparse Matrix-Vector Multiplication Using Quadro K4200" *IOP Publishing Ltd*, 2020. doi: 10.1088/1757-899X/854/1/012057

[10] Alexander Chen, Alan Edelman, Jeremy Kepner, Vijay Gadepally, Dylan Hutchison, "Julia Implementation of the Dynamic Distributed Dimensional Data Model" *IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-7, 2016. doi: 10.1109/HPEC.2016.7761626

[11] C. Ramírez, A. Castelló, H. Martínez, and E. S. Quintana-Ortí, "Performance Analysis of Matrix Multiplication for Deep Learning on the Edge," *Lecture Notes in Computer Science*, pp. 65–76, Mar. 2022. doi: 10.1007/978-3-031-23220-6_5

[12] A. O. Jimale, F. Ridzuan, and W. M. Wan Zainon, "Square Matrix Multiplication Using CUDA on GPGPU," *Procedia Computer Science*, vol. 161, pp. 398–405, 2019. doi: 10.1016/j.procs.2019.11.138

[13] O. Zachariadis, "Heterogeneous Parallel Computing for Image Registration and Linear Algebra Applications," *UCOPress*, pp. 1–123, 2020. Accessed: Mar. 31, 2024. [Online]. Available: https://helvia.uco.es/bitstream/handle/10396/20318/2020000002129.pdf

[14] Z. Gu, "Optimizing Block-Sparse Matrix Multiplications on CUDA with TVM," *CoRR*, Jul. 2020. Accessed: Nov. 5, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2007.13055

[15] A. Kahle, "The Julia Programming Language and Its Suitability for HPC," *Seminar: Newest Trends in High-Performance Data Analytics*, pp. 1–16, Sep. 2022. Accessed: Mar. 31, 2024. [Online]. Available: https://hps.vi4io.org/_media/teaching/summer_term_2022/nthpda_report_julia_by_anna_kahle.pdf

[16] T. Faingnaert, T. Besard and B. De Sutter, "Flexible Performant GEMM Kernels on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2230-2248, 1 Sept. 2022. doi: 10.1109/TPDS.2021.3136457.

[17] Julia, "Linear Algebra," *Linear Algebra - The Julia Language*. Accessed: Apr. 12, 2024. [Online]. Available: https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#man-linalg-abstractq

[18] JuliaGPU, "Flexible and Performant GEMM Kernels in Julia," *GitHub*. Accessed: Apr. 12, 2024. [Online]. Available: https://github.com/JuliaGPU/GemmKernels.jl?tab=readme-ov-file