Efficient Matrix Operations and Parallel Programming for the Conjugate Gradient method on the DelftBlue

Pieter de la Court

July 12, 2025

Supervisor
Prof.dr.ir. M.B. van Gijzen

Faculty EEMCS, Delft University of Technology

TU Delft

Brief summary

This research has focussed on developing efficient matrix operations to utilize in the Conjugate Gradient method by means of parallel programming. The aim was to find a solution to the linear system.

$$Ax = b,$$

The Conjugate Gradient method is an iterative solver for such systems. The main bottleneck of this method is matrix-vector products, so naturally decreasing compute time for this operation was the focus for this project.

The compute time was decreased in two ways. First by storing the matrices used more efficiently by making use of Compressed Diagonal Storage. Secondly by implementing GPU offloading using directive based languages such as CUDA and OpenACC.

Contents

Abstract

This research investigates efficient matrix-vector multiplication and storage techniques for a Fortran implementation of the Conjugate Gradient (CG) method on GPUs. The CG method is a widely used iterative algorithm for solving large, sparse linear systems.

$$Ax = b,$$

Because its performance is heavily influenced by the efficiency of matrix operations it is important to make use of parallel computing to run the computations on a GPU.

Three types of matrices were considered as A, a dense symmetric positive-definite matrix and two sparse matrices, defined by the discretization of a 1D Laplace equation and a 2D Poisson equation respectively. For sparse matrices, the Compressed Diagonal Storage (CDS) format was implemented to reduce memory usage and computational cost. For parallel execution, three implementations were benchmarked: Standard Fortran, OpenACC, and CUDA Fortran.

The results show that CDS significantly improves both memory efficiency and runtime performance for sparse matrices, while CUDA outperforms OpenACC and Standard Fortran in terms of speed, especially for large matrices. Standard Fortran remains competitive for small matrices due to low overhead, but it scales poorly. The convergence behavior of the CG method was also found to be highly dependent on the condition number of the matrix, with the 2D Poisson matrix exhibiting much faster convergence than the 1D Laplace matrix.

This study concludes that efficient storage and GPU-based matrix operations, particularly using CUDA, are critical for scalable performance in solving large linear systems with the CG method. Future work could explore combining the portability and ease of use of Standard Fortran with GPU acceleration to achieve both maintainability and high performance.

1 Introduction

Matrices are a very common tool in many fields of science and engineering. They enable us to represent and manipulate data in a structured way, making them essential for large scale scientific computations. One particular usage of matrices is in solving linear systems, which is a common problem in many fields, such as physics, engineering, and computer science.

In this context, the Conjugate Gradient method [21] is a widely used iterative algorithm for solving large linear systems. This method was discovered in 1952 by Hestenes and Stiefel[10] and implementations of Conjugate gradient have been studied extensively in literature. A handful of these articles are Powell in 1977 [19] Bondarenko in 2014 [2] and Fletcher, Reeves in 1964 [5]. By looking at the dates it is clear that Conjugate Gradient has been around for a while.

The method is particularly effective for sparse matrices, which are common in many applications. The method is based on the idea of minimizing the quadratic form associated with the linear system. In calculating the solution, the method relies heavily on matrix-vector multiplications, which can be computationally expensive, especially for large matrices.

Real world problems often require A in the linear system to be very large, so the necessity for high-performance computing (HPC) arises quickly. On systems like DelftBlue[4] researchers are constantly working with matrices of enormous scale. To effectively utilize the computational power of such systems, it is crucial to leverage the potential of GPUs (Graphics Processing Units) for parallel processing. GPUs are particularly well-suited for matrix operations due to their ability to perform many calculations simultaneously, which can significantly speed up the computation of matrix-vector multiplications [7][1] This was the context which provided the main motivation for this research.

The research has focussed on exploring the most effective ways to store and multiply dense and sparse matrices on the GPU. The language chosen to do this research was Modern Fortran, since it is well suited for numerical computations on a HPC environment and has great support for parallel programming on GPUs [6]. In particular, OpenACC[18], CUDA[16][20] and Standard Fortran were implemented as different means to effectively utilize the GPU. The performance of the Conjugate Gradient method served as a benchmark for these different implementations.

The aim of the research will be to optimize the computation cost of the Conjugate Gradient method. To accomplish this goal efficient matrix multiplication and storage techniques need to be developed, which should generalize well to different problems and improve performance on the GPU. Accordingly, the research question is:

What are the best matrix-vector multiplication and storage techniques for a Fortran implementation of the Conjugate Gradient method for dense and sparse matrices on a GPU?

I will utilize the following structure to examine my research question:

(i) Explanation of Conjugate Gradient method

(ii) Efficient data storage

(iii) Efficient calculation of matrix-vector products

(iv) Test setup

(v) Results

(vi) Conclusion

The first section will be a clarification of the Conjugate Gradient method, it will be vital to understand the method in order to make sense of the results. The second and third will be exploring storage and multiplications of matrices. Finally, the fourth and fifth sections will discuss the results of my research, my conclusion and include my recommendations.

2 Conjugate Gradient method

The Conjugate Gradient (CG)[21] method, originally discovered by Hestenes and Stiefel [10], is an iterative algorithm for solving systems of linear equations of the form:

$$Ax = b,$$

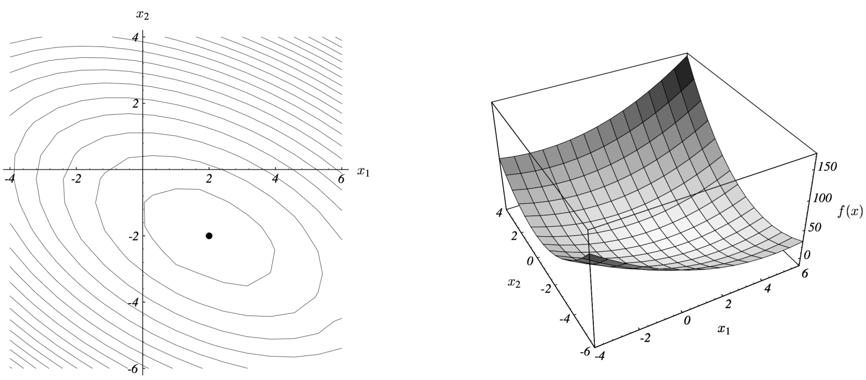
where $A \in \mathbb{R}^{n \times n}$ is symmetric and positive-definite, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ is the unknown solution vector. CG is mainly suited to sparse matrices. For dense matrices, it is usually better to use a direct method instead.

Rather than solving the system directly we reduce the problem to a form solvable by an iterative method. To be able to do this CG makes use of the quadratic function associated with the system.

$$f(x) = \frac{1}{2}x^T Ax - x^T b.$$

We require the matrix A to be positive definite because this ensures that the quadratic form will have a minimum. We require A to be symmetric such that the equality $\nabla f(x) = Ax - b$ holds, where $\nabla f(x)$ is the gradient of $f(x)$, as seen in figure 1a and 1b.

The minimum of this function corresponds to the solution of the linear system, since $\nabla f(x) = Ax - b = 0$. Hence we are looking for the optimal way to find the minimum of the quadratic form.



(a) Contours of the quadratic form.

(b) Gradient of the quadratic form.

Figure 1: The gradient of the quadratic form presented as contours and in a 3D graph. Pictures referenced from [21].

2.1 The method of Steepest Descent

The method of Steepest Descent minimizes $f(x)$ iteratively by moving in the direction of the negative gradient. The negative gradient of f at iteration k is given by:

$$r_k = b - Ax_k,$$

which also represents the residual vector (i.e., A times the difference between the current approximation and the true solution).

Each iteration updates the solution by:

$$x_{k+1} = x_k + \alpha_k r_k,$$

where the step size α_k is chosen to minimize $f(x_{k+1})$ along the direction r_k:

$$\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}.$$

While Steepest Descent is simple to implement, its convergence is often slow, especially when the condition number of A is large. The method can exhibit zigzagging behavior due to the fact that consecutive residuals are orthogonal but not A-orthogonal, this behavior is shown in figure 2.

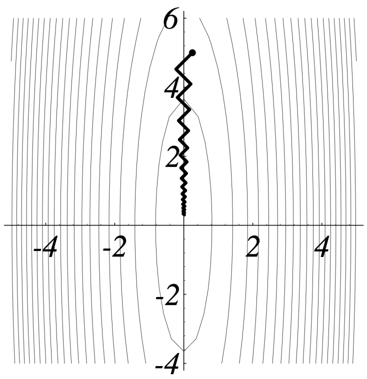


Figure 2: Slow convergence of the steepest descent method for an ill-conditioned matrix. Picture referenced from [21].

2.2 The method of Conjugate Directions

To improve convergence, the method of Conjugate Directions uses a sequence of search directions $p_0, p_1, \ldots, p_{n-1}$ that are A-conjugate (figure 3), i.e.,

$$p_i^T A p_j = 0 \quad \text{for } i \neq j.$$

Each update takes the form:

$$x_{k+1} = x_k + \alpha_k p_k,$$

where

$$\alpha_k = \frac{r_k^T p_k}{p_k^T A p_k}.$$

The residual is updated as:

$$r_{k+1} = r_k - \alpha_k A p_k.$$

With a set of n linearly independent A-conjugate directions, the method will reach the solution in at most n iterations [21]. However, generating and storing all n conjugate directions explicitly is computationally expensive in practice.

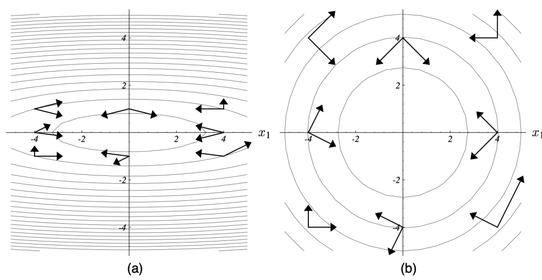


Figure 3: A orthogonality vs Normal orthogonality

2.3 The Conjugate Gradient method

The Conjugate Gradient method combines the efficiency of Steepest Descent with the optimality of Conjugate Directions. Instead of precomputing all directions, CG constructs the A-conjugate directions iteratively from the residuals.

Given an initial guess x_0, the algorithm proceeds as follows:

$$\begin{aligned} r_0 &= b - Ax_0, \\ p_0 &= r_0. \end{aligned}$$

For each iteration $k = 0, 1, 2, \ldots$, compute:

$$\begin{aligned} \alpha_k &= \frac{r_k^T r_k}{p_k^T A p_k}, \\ x_{k+1} &= x_k + \alpha_k p_k, \\ r_{k+1} &= r_k - \alpha_k A p_k, \\ \beta_k &= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}, \\ p_{k+1} &= r_{k+1} + \beta_k p_k. \end{aligned}$$

At each step, the new direction p_{k+1} is constructed to be A-conjugate to the previous directions, and the algorithm avoids the need to store all past directions. CG guarantees convergence in at most n steps in exact arithmetic.

Final notes

The primary computational bottleneck in CG is the matrix-vector product. This makes CG a suitable benchmark for evaluating the efficiency of matrix multiplication techniques, because the computation time of CG will be heavily correlated to the ability of efficient matrix operations.

3 Storing matrices efficiently

As we increase the size of the problems we're trying to solve, the size of the matrices grow as well. To illustrate the problem this could pose, consider a finite difference scheme of size n. The corresponding matrix will be of size $n \times n$ and thus have n^2 entries. If we were to store this matrix in a dense format, it would require $8n^2$ bytes of memory (assuming double precision floating point numbers). For example, for $n = 10^6$, this would require approximately 8 TB of memory, which is impractical for most systems.

For this reason, we need to consider alternative storage formats that are more memory-efficient, where we try to make use of the special properties of the graph to find the best format. In this section, we will explore Compressed Diagonal Storage (CDS)[13] for sparse matrices.

3.1 Sparse matrices

Compressed Diagonal Storage (CDS)

CDS is a memory-efficient format well suited for storing band matrices. These are a special subset of sparse matrices where non-zero elements are clustered around the main diagonal. Instead of storing all entries, CDS stores only the diagonals that contain non-zero values. In theory CDS works well for any matrix with a large amount of diagonals which only contain zeros, but for our purposes we will only consider band matrices. An important note is that CDS will not store all diagonals which fall within the band, regardless if they have non zero entries or not. A diagonal with all entries zero will not be stored, even if it is within the band of the matrix. An example of a matrix where this occurs would be a matrix of the discretized 2D Poisson equation.

Storing the matrix more efficiently does not only have benefits for the storage in memory. Since the original matrix contains many zero entries, when doing a matrix multiplication we are doing lots of unnecessary work. By compressing all the relevant data in a smaller format we can drastically reduce the amount of operations needed for a matrix multiplication.

Example of CDS format

To illustrate the way a matrix is converted to CDS format we consider the following matrix with one lower and one upper diagonal:

$$A = \begin{bmatrix} 2 & 3 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$

This matrix has 3 non-zero diagonals:

- Lower diagonal (offset -1): $[1, 1, 1, 1]$
- Main diagonal (offset 0): $[2, 2, 2, 2, 2]$
- Upper diagonal (offset +1): $[3, 3, 3, 3]$

In CDS, we store the diagonals in a matrix of size n by the number of diagonals. In this case that amounts to 5 by 3. The offsets can be included in the matrix or stored in an additional array. These will help us to reconstruct A in order to perform operations.

- The first column of the matrix corresponds to the first row of A
- Each row of the table represents one diagonal
- Values not present in the matrix (outside the bounds of the diagonal) are padded with 0

-1
0
$+1$

(a) The offsets of Matrix A

0	1	1	1	1
2	2	2	2	2
3	3	3	3	0

(b) Matrix A in CDS format.

3.2 Dense matrices

Unfortunately, dense matrices offer limited opportunities for compression. So for dense matrices, we can use a simple 2D array to store the matrix. This is straightforward and efficient for small to medium-sized matrices. However, as the size of the matrix grows, this approach becomes impractical due to memory constraints. Unfortunately, there is no way to compress a dense matrix in the same way as we can with sparse matrices. The only way to reduce the memory footprint of a dense matrix is to use a lower precision data type.

4 Efficient matrix operations

With the storage formats and algorithm established, we now need to implement the matrix operations in order to solve the linear systems. Conjugate Gradient relies on 2 main operations [21]:

- Matrix-vector multiplication
- Dot products

These operations are the building blocks of the algorithm and need to be implemented efficiently to ensure good performance. There are other operations that are used in the algorithm, such as vector addition and scalar multiplication, but any techniques we develop for the two main operations will be applicable to the other operations as well. The matrix-vector multiplication and dot products are the most computationally expensive operations in the algorithm, and therefore we will focus on optimizing these.

Matrix operations are highly suitable for parallelization as they are made up of many independent operations. This means that we can use multiple threads to perform the operations in parallel, which can significantly speed up the computations. This is where parallel programming and the DelftBlue HPC system become essential. The DelftBlue[4] is a HPC cluster which provides top of the line GPU's and CPU's to perform the computations. The main challenge of this research will be to make optimal use of the available resources of DelftBlue. The best approach to do this is by using parallel programming techniques.

4.1 GPU vs CPU architecture

To start off we must decide whether to offload some of the computation to the GPU or to perform all of them on the CPU. In order to make this decision the main differences between the two architectures will be listed and how they affect the performance of matrix operations.

The main difference between GPU and CPU architecture is the number of cores and the way they are designed to handle parallelism. Have a look at figure 5 for a graphical representation. A CPU is designed to handle a few threads at a time, with each core being optimized for single-threaded performance. A GPU, on the other hand, is designed to handle thousands of threads at a time, with each core being optimized for parallelism [14]. The result is that GPUs are much better suited for tasks that can be highly parallelized, such as matrix operations [12].

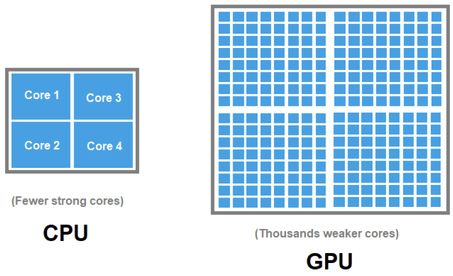


Figure 5: A simplified version of CPU architecture vs GPU architecture. [22]

This highlights limitations in CPU parallelism, but it is important to note that each individual core of a CPU is much more powerful than a single core of a GPU. This means that GPUs are mainly effective at tasks which can be split up in many simple subtasks. So essentially GPUs are a highly specialized form of CPUs, not suitable for all calculations, but extremely well tailored to our problem of matrix operations.

Data transfer

We have now spoken about the CPU and GPU in isolation, but in reality all computations will be performed on the CPU and GPU in tandem. This means that we need to transfer data between the two architectures. The data transfer is a crucial part of the performance of our algorithm, as it can be a bottleneck if not done efficiently[20].

If we inefficiently transfer data between the CPU and GPU, we will create a bottleneck that will slow down our computations. This is because the data transfer is much slower than the actual computations performed on the GPU. This means that we need to minimize the amount of data transfer between the CPU and GPU, and only transfer data when absolutely necessary.

Figure 6 shows an application which makes use of GPU acceleration. The CPU runs the main program, while the GPU performs the tasks which are parallelized. The data transfer is shown by the black line, where the CPU and GPU communicate with each other to transfer data.

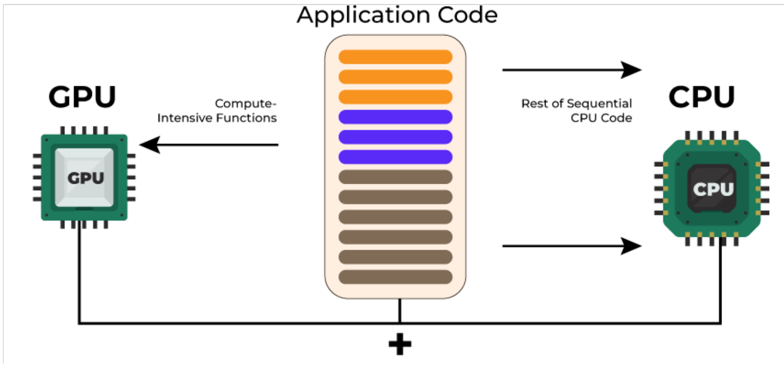


Figure 6: Running a program which makes use of GPU offloading. The CPU runs the main program, while the GPU performs the matrix operations. Picture referenced from [8]

Having seen the difference between CPU and GPU architecture our objective is now clear:

To decrease compute time for CG we need to offload the matrix operations to the GPU while minimizing the time spent transferring data between the CPU and GPU.

4.2 Fortran

To write our code we will use Fortran[6], which is a high-level programming language that is widely used in scientific computing. Fortran has a long history of being used for numerical computations and is well suited to support parallel programming.

Fortran intrinsics provide a set of built-in functions that can be used to perform common operations, such as matrix multiplication (`matmul`) and dot products (`dot_product`) . Additionally, Fortran has a built in construct that allows for easy parallelization of loops (`do concurrent`). However, by default these intrinsics and constructs will only use the CPU.

4.2.1 Compilers

A compiler is a program that translates source code written in a high-level programming language into machine code that can be executed by a computer. There are multiple compilers available for Fortran, but not all of them are suitable for GPU offloading. The most commonly used compilers for Fortran are:

- GNU Fortran [9] (`gfortran`)
- Intel Fortran [11] (`ifort`)
- NVIDIA HPC SDK [17] (`nvfortran`)

We will not go into depth about each compiler, but it is important to note that they all have different capabilities and performance characteristics. The NVIDIA HPC SDK is specifically designed for GPU programming and provides the best support for offloading Fortran code to the GPU.

`Nvfortran` provides us with support for OpenACC and CUDA, which are two programming models that allow us to offload code to the GPU.

4.3 OpenACC

While Fortran intrinsics offer an elegant high-level approach, they provide limited control over hardware-specific optimizations. OpenACC addresses this by allowing explicit control over parallel regions and memory transfers using directives.

OpenACC [18] is a programming model which allows us to write standard Fortran code and then add directives to indicate which parts of the code should be offloaded to the GPU. However, to get the best performance we also have to manually manage the data transfer from the CPU to the GPU. In theory this should allow us to write high-level code that is easy to read and maintain, while giving us more control over the hardware thus increasing performance over standard Fortran.

But as we will see this does not always lead to better performance and can sometimes lead to worse performance than using Fortran intrinsics. The reason being that we manually manage the data transfer, which can lead to suboptimal performance if not done correctly.

4.4 CUDA

CUDA [16] is similar to OpenACC in that it allows us to write code that can be offloaded to the GPU. However, CUDA is a lower-level programming model that requires us to write more code to achieve the same result. This means that we have more control over the hardware, but it also means that the code is harder to read and maintain. The upside of CUDA is that the potential performance benefits are greater [20]. Additionally, CUDA is developed by NVIDIA and is specifically designed for their GPUs, which means that it can take advantage of the latest hardware features.

5 Testing setup

Having established the theoretical background of the Conjugate Gradient method, we now turn to its practical implementation and testing. As we have seen we can use parallel programming to speed up computation.

In section 5.3 we discuss the different ways we implement parallel programming to offload to the GPU. Section 5.2 defines the range of matrices which we will test these implementations on and 5.1 states the system specifications on which the tests were run. Finally, one of the pillars of good research is reproducibility so the section 5.4 will describe the testing procedure used.

5.1 Test environment

The tests were conducted on a GPU node of the DelftBlue with the specifications [3]:

- CPU: Intel XEON E5-6448Y 32C 2.1GHz
- GPU: NVIDIA A100 GPU
- RAM: 512 GB
- OS: Red Hat Enterprise Linux 8

Furthermore, the tests were ran with the following software:

- Modern Fortran
- OpenACC : 2.7 (tied to the NVIDIA HPC SDK)
- NVIDIA HPC SDK : 23.5
- CUDA : 11.6

5.2 Test matrices

The tests were conducted on 3 types of matrices:

- A dense, symmetric and positive definite matrix
- A sparse matrix attained by the finite difference scheme of a 1D laplace equation in CDS format.
- A sparse matrix attained by the finite difference scheme of a 2D poisson equation in CDS format.

The sizes of these matrices we tested on were:

Matrix	**Size Range**	**Storage Format**
Dense matrix	2^4 - 2^{12}	Dense 2D array
1D Laplace	2^4 - 2^{20}	CDS format
2D Poisson	2^4 - 2^{20}	CDS format

Table 1: Test matrices and their sizes, step size of $\cdot 2$ between sizes.

It is evident that the range of size for the matrices is quite large. The dense matrix has the smallest range, because it is not practical to test on larger matrices due to memory constraints. The sparse matrices, however, can be much larger, and we will be testing on matrices with up to 1 million elements. A step size of 2^2 between sizes allows us to see how the performance scales with the size of the matrix.

5.3 Test cases

Having established the matrices and the environment, we can now describe the test cases. A quick reminder of the linear system we are solving:

$$Ax = b,$$

where A is the matrix, b is the right-hand side vector, and x is the solution vector we are trying to find. As mentioned before, we will be using the Conjugate Gradient method to solve this system.

For all test cases, we will be using the same right-hand side vector b which is a vector of ones. The initial guess for the solution vector x will be a vector of zeros.

The test cases will be the 3 matrices described above, and we will be testing the following implementations of CG on each of these matrices.

- Standard Fortran, run on the CPU.
- OpenACC, parallelized and offloaded to the GPU using OpenACC directives.
- CUDA, parallelized and offloaded to the GPU using CUDA directives.

In each case we will implement the Compressed Data Storage (CDS) format for the sparse matrix, where applicable, and compare the performance of the different implementations. The goal is to see how much speedup we can achieve by using parallel programming and efficient storage formats.

5.4 Test procedure

It is now clear what our test cases are and how we will implement them. The next step is to describe the procedure we will follow to conduct the tests.

1. Set epsilon to 1.0d-8 and max iterations to a suitable number for the matrix size
2. Compile each case with `nvfortran` and the `-Ofast` flag, along with any other required flag for the specific implementation.
3. Run the program 100 times for each test case to ensure statistical significance.
4. Measure the time taken for each run using the `cpu_time()` function.
5. Calculate the average time taken for each test case and implementation.

6 Results & Discussion

The previous sections were suggestively named Efficient matrix operations and Efficient storage, but the performance of the implementations were never shown. In this section, the results of the tests conducted for the different implementations of matrix operations and storage methods will be shown.

6.1 Storage Comparison

We will start off by focussing on the storage methods. Because storing dense matrices rapidly becomes infeasible, we will keep the size of the matrices small, but the results will still be representative of larger matrices. We will consider the 3 cases mentioned in section 5.3 and compare a naive implementation, which stores the matrix in a 2D array, with the Compressed Data Storage (CDS) format.

The table below shows the memory usage of the different storage methods for the 3 matrices. The memory usage is given in bytes, and the size of the matrix is given in number of elements.

Matrix	Naive Storage	CDS Storage	Size matrix
Dense matrix	50 Mb	50 Mb	2500×2500
1D Laplace	50 Mb	0.06 Mb	2500×2500
2D Poisson	50 Mb	0.1 Mb	2500×2500

Table 2: Memory usage of different storage methods for matrices

As we can see, the memory usage is the same for both storage methods for the dense matrix, as it is a dense matrix and the CDS format does not provide any benefits. However, for the 1D Laplace and 2D Poisson matrices, the CDS format provides a significant reduction in memory usage. The reason for this reduction is due to the fact that the CDS format only stores the diagonals with non-zero elements of the matrix, which is a significant reduction in size for sparse matrices.

Now for the important part, how does this affect the performance of the Conjugate Gradient (CG) method? The table below shows the average time taken to solve the linear system $Ax = b$ for the different storage methods and matrices. For this comparison standard Fortran was used without any offloading to the gpu. The time is given in seconds, and the size of the matrix is given in number of elements.

Matrix	Naive Storage (s)	CDS Storage (s)	Size matrix
Dense matrix	102.9	-	4096×4096
1D Laplace	22.03	0.015	4096×4096
2D Poisson	1.326	0.001	4096×4096

Table 3: Conjugate Gradient performance comparison of naive storage vs CDS storage for matrices

As we can see, the CDS format provides a significant speedup for the 1D Laplace and 2D Poisson matrices, while the dense matrix does not show a significant difference. This is because the CDS format reduces the number of operations required to solve the linear system, as it only operates on the non-zero elements of the matrix.

Due to this conclusion, in the next section we will only consider the CDS format for the 1D Laplace and 2D Poisson matrices, as in practice one would never store a sparse matrix in a dense format. This will also allow us to run our tests on larger, more realistic matrices, as the memory usage will be significantly lower.

6.2 Parallel computing comparison

In this section, we will compare the performance of the different parallel computing methods for the test cases defined in section 5.3. We will compare the performance of Standard Fortran without any optimizations, OpenACC and CUDA. Both the 1D Laplace and 2D Poisson matrices will be in CDS format, while the dense matrix will be stored in a 2D array. The performance will be measured in seconds.

Tables and Graphs

The results of the performance tests are presented in the form of tables and accompanying graphs. The tables will show the time taken to solve the linear system $Ax = b$ for the different storage methods and parallel computing methods. The graphs will show the performance of the different methods in a log-log plot, which allows us to see the performance scaling with increasing matrix size.

6.2.1 Laplacian 1D

Size	Standard Time (s)	OpenACC Time (s)	CUDA Time (s)	Iterations
2^4	4.268e-7	6.600e-4	5.388e-4	8
2^6	3.419e-6	2.519e-3	1.564e-3	32
2^8	4.293e-5	1.009e-2	6.800e-3	128
2^{10}	6.901e-4	3.535e-2	2.788e-2	512
2^{12}	1.463e-2	1.556e-1	1.1109e-1	2048
2^{14}	2.281e-1	6.384e-1	4.526e-1	8192
2^{16}	4.572	2.928	1.827	32768
2^{18}	175.7	20.23	7.712	131072
2^{20}	-	221.6	72.84	524288

Table 4: Comparison of Laplacian 1D results for Standard, OpenACC and CUDA implementations. The last two entries of Standard and OpenACC were omitted, because they took too long to compute.

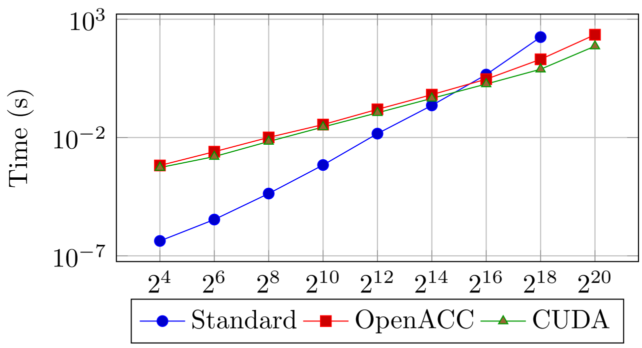


Figure 7: Log-log plot of Laplacian 1D times for Standard, OpenACC, and CUDA.

6.2.2 Poisson 2D

Size	Standard Time (s)	OpenACC Time (s)	CUDA Time (s)	Iterations
2^4	1.979e-7	2.859e-4	2.055e-4	3
2^6	1.416e-6	7.914e-4	5.358e-4	10
2^8	1.331e-5	2.018e-3	1.561e-3	28
2^{10}	1.146e-4	4.176e-3	3.313e-3	59
2^{12}	1.198e-3	9.128e-3	6.459e-3	119
2^{14}	8.874e-3	1.904e-2	1.351e-2	239
2^{16}	8.781e-2	4.070e-2	2.751e-2	470
2^{18}	1.650	0.1496	6.022e-2	941
2^{20}	18.11	0.8389	0.3000	1898

Table 5: Comparison of Poisson 2D results for Standard, OpenACC and CUDA implementations.

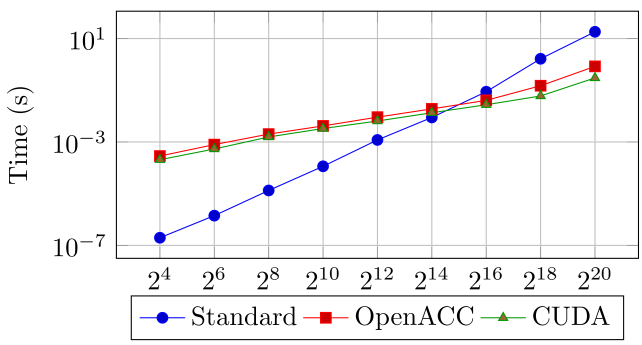


Figure 8: Log-log plot of Poisson 2D times for Standard, OpenACC, and CUDA.

6.2.3 Dense SPD

Size	Standard Time (s)	OpenACC Time (s)	CUDA Time (s)	Iterations
2^4	1.457e-6	1.273e-3	8.985e-4	18
2^6	9.882e-5	7.494e-3	5.011e-3	104
2^8	6.057e-3	3.657e-2	3.126e-2	524
2^{10}	6.989e-1	2.037e-1	2.047e-1	2448
2^{12}	103.0	2.710	2.534	9759

Table 6: Comparison of Dense SPD results for Standard, OpenACC, and CUDA implementations.

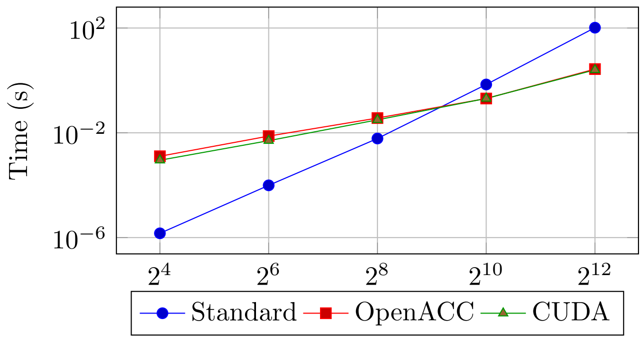


Figure 9: Log-log plot of Dense SPD times for Standard, OpenACC, and CUDA.

6.3 Discussion

The results of the performance tests show that the OpenACC and CUDA implementations provide a significant speedup compared to the Standard Fortran implementation when large matrix-vector multiplications are involved. The speedup is especially significant for large dense matrices, where the performance difference can be several orders of magnitude. To structure the discussion the following aspects will be discussed:

- Performance of Standard Fortran vs Parallel implementations
- The impact of matrix choice on performance.
- The scalability of the different implementations.

6.3.1 Standard Fortran vs Parallel Implementations

The Standard Fortran implementation is significantly slower than the OpenACC and CUDA implementations for large matrices. The performance difference is especially pronounced for the dense matrix, where the OpenACC and CUDA implementations are several orders of magnitude faster. This is due to the fact that the OpenACC and CUDA implementations are able to take advantage of the parallelism offered by the GPU, while the Standard Fortran implementation is limited to the CPU's capabilities.

It is evident that for small matrices, the performance difference is not as pronounced, and the Standard Fortran implementation can still be competitive. This is because the overhead of transferring data to and from the GPU can outweigh the performance benefits for small matrices. The effects are especially pronounced for the 1D Laplace and 2D Poisson matrices. The cause for this effect presumably is the conversion to CDS format. This means that the number of operation required to solve the linear system is significantly reduced. Thus the OpenACC and CUDA implementations aren't able to take advantage of the parallelism offered by the GPU, as the number of operations is too small. Hence the performance difference becomes pronounced only when the matrix size increases to significantly larger matrices.

The dense matrix, on the other hand, does not benefit from the CDS format and the OpenACC and CUDA implementations are able to take advantage of the parallelism offered by the GPU. This is why the performance difference becomes apparent at smaller matrix sizes for dense matrices.

6.3.2 Impact of Matrix Choice on Performance

So far we only made a distinction between a sparse and a dense matrix, but this does not tell the entire story. The performance of the different implementations is also affected by other aspects of the matrix. The 1D Laplace and 2D Poisson matrices are both sparse matrices, but they have different sparsity patterns. The 1D Laplace matrix has a tridiagonal structure, while the 2D Poisson matrix has a block structure. When looking at their convergence, the 2D Poisson matrix converges in significantly less iterations.

To understand why this happens it is important to have a solid grasp of Conjugate Gradient (CG) method. The number of iterations in which CG converges is mainly driven by the condition number of the matrix. A brief reminder: the condition number of a matrix is the ratio of the largest eigenvalue to the smallest eigenvalue.

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

A matrix with a low condition number will converge in fewer iterations than a matrix with a high condition number.

The 1D Laplace matrix has a condition number which scales with $\kappa(A) \sim \mathcal{O}(N^2)$ [23]. While the 2D Poisson matrix has a condition number which scales with $\kappa(A) \sim \mathcal{O}(N)$ [23]. This means that the 2D Poisson matrix will converge in significantly fewer iterations than the 1D Laplace matrix, as the condition number is lower. This means that even though each iteration is computationally more expensive for the 2D Poisson matrix since it has 2 extra non zero diagonals, the number of iterations is significantly lower, resulting in a lower overall computational cost.

The difference between these 2 matrices illustrates the importance of the choice of matrix in the performance of the CG method. The condition number of the matrix is a key factor in determining the number of iterations required for convergence, and thus the overall performance of the CG method.

6.3.3 Scalability of the Implementations

The previous section have briefly touched on the scalability of the different implementations, but it is important to discuss this aspect in more detail. Real world use cases often involve matrices which are even larger than the ones tested, so it is important to understand how the performance of the different implementations scales with increasing matrix size.

The results are somewhat surprising, as particularly the OpenACC implementation does not scale as well as one might expect. The performance of the OpenACC implementation is significantly slower than the CUDA implementation for large matrices, and the performance difference becomes more pronounced as the matrix size increases.

We identified two possible causes for this performance difference. The first is that the OpenACC implementation is not able to take advantage of the full parallelism offered by the GPU, possibly due to the way the OpenACC directives are compiled. The second is that within the implementation of the OpenACC directives, the data transfer between the CPU and GPU is not optimized, resulting in a significant overhead for large matrices.

The CUDA implementation, on the other hand, scales well with increasing matrix size. This was expected as CUDA is a lower level programming model which allows for more fine-grained control over the GPU. Comparing CUDA to Standard Fortran, the performance difference becomes quite large when the matrix size increases. The CUDA implementation is able to take advantage of the parallelism offered by the GPU, and it is clear that for large matrices the CUDA implementation is the best choice.

6.4 Limitations

6.4.1 Standard Fortran with GPU offloading

One of the other implementations that was attempted is Standard Fortran with GPU offloading. This implementation showed promise due to combining the ease of use and portability of Standard Fortran with GPU offloading. However, the implementation could not be finalized within the project timeline. This might seem contradictory since it was just mentioned that Standard Fortran is easier to use, but the problem arose in compilation, not in writing the actual code. For future research this implementation could definitely be an interesting area to explore.

6.4.2 Profiling the code

While this research focused on algorithmic and architectural optimization, it did not involve formal profiling of execution time or memory usage. Tools such as NVIDIA Nsight Systems or Nsight Compute could have provided deeper insights into kernel efficiency, memory transfer bottlenecks, and warp utilization. Without such profiling, performance interpretations—particularly for OpenACC and CUDA—rely on indirect evidence. Future work should include systematic profiling to support optimization decisions and reveal hidden inefficiencies.

6.4.3 Coarrays and MPI

A reader familiar with HPC and parallel computing might have expected a section on MPI or on coarrays, the Standard Fortran alternative to MPI. While MPI and coarrays are both powerful tools to distribute work over multiple nodes, they are not included in this thesis. There were multiple reasons as to this decision was made.

The first reason is that the scale of the problems we were working with was not large enough to warrant the use of MPI or coarrays. The matrices were still relatively small, and the performance gains from using MPI or coarrays would not have been significant enough to justify the added complexity.

The second and main reason was that we chose to focus on developing the fastest solution for a GPU on a single node. The addition of MPI or coarrays would have added unnecessary complexity and taken away from this focus.

7 Conclusion

Let's start off by restating the research question of this project.

What are the best matrix-vector multiplication and storage techniques for a Fortran implementation of the Conjugate Gradient method for dense and sparse matrices on a GPU?

The key findings of this research can be split into two parts: the storage of matrices and the multiplication of matrices and vectors.

For the storage of matrices, it became clear that the choice of format has a significant impact on performance. The Compressed Diagonal Storage (CDS) format was found to be the most efficient for sparse matrices, while for dense matrices, no improvement was found over the standard Fortran array storage. The CDS format allowed for efficient access to non-zero elements, which is crucial for the performance of the Conjugate Gradient method.

The main part of the research focused on the multiplication of matrices and vectors. While OpenACC was promising, the results showed us that the CUDA implementation outperformed OpenACC in terms of speed. The CUDA implementation was able to leverage the parallel processing capabilities of the GPU more effectively, resulting in faster matrix-vector multiplications. Standard Fortran was faster for small matrices as expected, but it did not scale well for larger matrices. We conclude that CUDA is the preferred choice for high-performance computing applications.

The scope of this research was limited, which meant that not all possible matrix configurations and multiplication techniques could be explored. However, the findings provide a solid foundation for future work in this area. Future work could focus on implementing Standard Fortran combined with GPU acceleration. This was an area which showed promise, but which was not realized in this research. Other potential research could focus on using coarrays or profiling tools such as Nsight Compute to further optimize memory transfers and kernel launch overheads. Incorporating MPI or multi-node extensions would also be essential for scaling this approach beyond a single GPU node on DelftBlue.

In conclusion, this research has demonstrated the importance of efficient matrix storage and multiplication techniques using parallelism for the Conjugate Gradient method on a GPU environment. While the allure of easy to implement GPU acceleration is tempting, as of right now CUDA Fortran still reigns supreme in terms of performance. However, the potential for future work in this area is promising, and further research could combine the portability of Standard Fortran with the performance benefits CUDA provides [15]. The findings of this research hopefully serve as a basis for future work, and it is hoped that they will contribute to the DelftBlue project and the broader field of high-performance computing.

References

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

[2] E Bondarenko. Modern parallel programming tools for solving system of the linear equations by conjugate gradient methods. *System technologies*, 2(2):108–115, 2014.

[3] Delft High Performance Computing Centre. System specifications — delftblue supercomputer. `https://www.tudelft.nl/dhpc/system`, 2023.

[4] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2`, 2024.

[5] Reeves Fletcher and Colin M Reeves. Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154, 1964.

[6] fortran-lang community. Fortran programming language general information. `https://fortran-lang.org`, 2023.

[7] Noriyuki Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[8] GeeksforGeeks. What is gpu acceleration? `https://www.geeksforgeeks.org/what-is-gpu-acceleration/`, 2022.

[9] GNU Project. *GNU Fortran Compiler Manual*, 2023.

[10] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[11] Intel Corporation. *Intel Fortran Compiler Classic and Intel Fortran Compiler Developer Guide and Reference*, 2023.

[12] Feng Li, Yunming Ye, Zhaoyang Tian, and Xiaofeng Zhang. Cpu versus gpu: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms. *Neural Computing and Applications*, 31:4353–4365, 2019.

[13] Netlib. Compressed diagonal storage format. `https://www.netlib.org/`, n.d.

[14] NVIDIA Corporation. Whats the difference between a cpu and a gpu? `https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/`, 2009.

[15] NVIDIA Corporation. Bringing tensor cores to standard fortran, 2022.

[16] NVIDIA Corporation. *CUDA Fortran Programming Guide and Reference*, 2023.

[17] NVIDIA Corporation. *NVIDIA HPC SDK Documentation*, 2023.

[18] OpenACC. The openacc application programming interface, version 2.7. Technical report, OpenACC-Standard.org, 2022.

[19] Michael James David Powell. Restart procedures for the conjugate gradient method. *Mathematical programming*, 12:241–254, 1977.

[20] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming.* Morgan Kaufmann, 2013.

[21] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Technical Report, Carnegie Mellon University*, 1994, 1994.

[22] TecAdmin. Cpu vs gpu: Key differences, 2022.

[23] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* SIAM, 1997.

A Appendix: CDS code

```
  module cds_matrix
  use iso_fortran_env , only: real64
  implicit none
  public

  !CDS stands for Compressed Diagonal Storage , which is a format for storing sp
  type :: cds_type
    integer :: n              ! matrix size (n x n)
    integer :: ndiag          ! number of diagonals
    real(real64), pointer :: data(:,:)  ! diagonal data (n x ndiag)
    integer, pointer :: offsets(:) ! diagonal offsets
    end type cds_type

contains

! Helper subroutine to count the number of diagonals in a 2D array and store th
subroutine count_diagonals(array, num_diagonals, diagonal_offsets)
    real(real64), intent(in) :: array(:,:)     ! Input 2D array
    integer, intent(out) :: num_diagonals  ! Number of diagonals
    integer, allocatable, intent(out) :: diagonal_offsets(:)
! Valid diagonal offsets

    integer :: m, n, min_dim, max_dim, d, i
    logical :: offset_exists
    integer :: count
    real(real64), parameter :: tolerance = 1.0e-10_real64
! Add tolerance for floating-point comparison

    m = size(array, 1)  ! Number of rows
    n = size(array, 2)  ! Number of columns

    ! Calculate possible diagonal range
    min_dim = 1 - n       ! Minimum possible diagonal offset
    max_dim = m - 1       ! Maximum possible diagonal offset

    ! Initialize counter
    num_diagonals = 0

    ! First pass: Count the number of valid diagonals
    do d = min_dim, max_dim
      offset_exists = .false.

      ! Check if this diagonal has any non-zero elements
      do i = max(1, 1-d), min(m, n-d)
        if (abs(array(i, i+d)) > tolerance) then  ! Compare using tolerance
          offset_exists = .true.
          exit
```

```fortran
        end if
      end do

      ! If diagonal exists, increment the count
      if (offset_exists) then
        num_diagonals = num_diagonals + 1
      end if
    end do

    ! Allocate diagonal_offsets to the exact size of valid diagonals
    allocate(diagonal_offsets(num_diagonals))

    ! Second pass: Store the valid diagonal offsets
    count = 0
    do d = min_dim, max_dim
      offset_exists = .false.

      ! Check if this diagonal has any non-zero elements
      do i = max(1, 1 - d), min(m, n - d)
        if (array(i, i + d) /= 0.0) then
          offset_exists = .true.
          exit
        end if
      end do

      ! If diagonal exists, store the offset
      if (offset_exists) then
        count = count + 1
        diagonal_offsets(count) = d
      end if
    end do

  end subroutine count_diagonals

! Initialize a CDS matrix with size n, ndiag number of diagonals and [] as the
subroutine cds_init(cds_matrix, n, ndiag, offsets)
    type(cds_type), intent(out) :: cds_matrix
    integer, intent(in) :: n, ndiag
    integer, intent(in) :: offsets(ndiag)

    cds_matrix%n = n
    cds_matrix%ndiag = ndiag
    allocate(cds_matrix%data(n, ndiag))
    allocate(cds_matrix%offsets(ndiag))
    cds_matrix%offsets = offsets
    cds_matrix%data = 0.0_real64   ! Initialize to zero
  end subroutine cds_init

!Convert regular finite difference matrix to CDS format
subroutine cds_convert(fdm_matrix, cds_matrix)
```

```
    real(real64), intent(in) :: fdm_matrix(:,:)
    type(cds_type), intent(out) :: cds_matrix
    integer :: n_diags
    integer, allocatable :: diag_offsets(:)
    integer :: i,j

    call count_diagonals(fdm_matrix, n_diags, diag_offsets)
    call cds_init(cds_matrix, size(fdm_matrix, 1), n_diags, diag_offsets)

    do i = 1, cds_matrix%ndiag
      do j = 1, cds_matrix%n
        ! Check if the column index is within bounds
        if (j + cds_matrix%offsets(i) >= 1 .and. j + cds_matrix%offsets(i) <= c
          cds_matrix%data(j, i) = fdm_matrix(j, j + cds_matrix%offsets(i))
        else
          cds_matrix%data(j, i) = 0.0_real64   ! Set to zero if the entry doesn
        end if
      end do
    end do

  end subroutine cds_convert
!Multiply a vector with a Matrix in CDS format
subroutine cds_matvec(cds_matrix, vector, result)
    type(cds_type), intent(in) :: cds_matrix
    real(real64), intent(in) :: vector(cds_matrix%n)
    real(real64), intent(out) :: result(cds_matrix%n)
    integer :: i,j
    integer :: n,ndiag, offset

    n = cds_matrix%n
    ndiag = cds_matrix%ndiag

    result = 0.0_real64

    do j = 1, ndiag
       offset = cds_matrix%offsets(j)
       if ( offset < 0 ) then
          do i = 1-offset,n
             result(i) = result(i) + cds_matrix%data(i,j)*vector(i+offset)
          end do
       else
          do i = 1,n-offset
            result(i) = result(i) + cds_matrix%data(i,j)*vector(i+offset)
          end do
        end if
    end do

  end subroutine cds_matvec
!Multiply a vector with a Matrix in CDS format using regular do concurrent par
subroutine cds_matvec_parallel(cds_matrix, vector, result)
```

```fortran
    type(cds_type), intent(in) :: cds_matrix
    real(real64), intent(in) :: vector(cds_matrix%n)
    real(real64), intent(out) :: result(cds_matrix%n)
    integer :: i,j
    integer :: n,ndiag, offset

    n = cds_matrix%n
    ndiag = cds_matrix%ndiag

    result = 0.0_real64

    do j = 1, ndiag
       offset = cds_matrix%offsets(j)
       if ( offset < 0 ) then
          do concurrent(i=1-offset:n)
             result(i) = result(i) + cds_matrix%data(i,j)*vector(i+offset)
          end do
       else
          do concurrent (i=1:n-offset)
            result(i) = result(i) + cds_matrix%data(i,j)*vector(i+offset)
          end do
        end if
    end do

  end subroutine cds_matvec_parallel

!Multiply a vector with a Matrix in CDS format using OpenACC parallelization
subroutine cds_matvec_OpenACC(cds_matrix, vector, result)
    type(cds_type), intent(in)  :: cds_matrix
    real(real64), intent(in)     :: vector(cds_matrix%n)
    real(real64), intent(out)    :: result(cds_matrix%n)
    real(real64), allocatable :: entries(:,:)
    integer, allocatable :: offsets(:)
    integer :: i, j
    integer :: n, ndiag, offset
    real(real64) :: sum

    n = cds_matrix%n
    ndiag = cds_matrix%ndiag
    result = 0.0_real64

    ! Allocate and initialize local arrays
    allocate(entries(n, ndiag), source=cds_matrix%data)
    allocate(offsets(ndiag), source=cds_matrix%offsets)

    !$acc data present_or_copyin(entries, offsets, vector) copyout(result)
    !$acc parallel loop gang vector_length(128) private(j,offset,sum) present(e
    do i = 1, n
        sum = 0.0_real64
        do j = 1, ndiag
```

```fortran
                offset = offsets(j)
                if (offset < 0) then
                    if (i + offset >= 1) sum = sum + entries(i, j) * vector(i + off
                else
                    if (i + offset <= n) sum = sum + entries(i, j) * vector(i + off
                end if
            end do
            result(i) = sum
        end do
        !$acc end parallel loop
        !$acc end data

        deallocate(entries, offsets)

    end subroutine cds_matvec_OpenACC
    !Print CDS matrix in correct formatting
    subroutine cds_print(cds_matrix)
        type(cds_type), intent(in) :: cds_matrix
        integer :: i, j

        print *, "CDS-Matrix:"
        do i = 1, cds_matrix%n
          write(*, '(10F8.2)', advance='no') (cds_matrix%data(i, j), j = 1, cds_ma
          print *
        end do
      end subroutine cds_print

    !Free memory after allocating memory to CSD matrix
    subroutine cds_destroy(cds_matrix)
        type(cds_type), intent(inout) :: cds_matrix
        if (associated(cds_matrix%data)) deallocate(cds_matrix%data)
        if (associated(cds_matrix%offsets)) deallocate(cds_matrix%offsets)
      end subroutine cds_destroy
    ! Build a 1D Laplacian matrix directly in CDS format
    subroutine build_laplacian_1d_cds(cds_matrix, n)
        use iso_fortran_env, only: real64
        implicit none
        type(cds_type), intent(out) :: cds_matrix
        integer, intent(in) :: n
        integer, parameter :: ndiag = 3
        integer :: i, d
        integer, dimension(ndiag) :: offsets

        offsets = (/ -1, 0, 1 /)
        call cds_init(cds_matrix, n, ndiag, offsets)

        ! Fill diagonals
        do d = 1, ndiag
            select case(offsets(d))
            case(-1)
```

```fortran
                do i = 2, n
                    cds_matrix%data(i, d) = -1.0d0
                end do
            case(0)
                do i = 1, n
                    cds_matrix%data(i, d) = 2.0d0
                end do
            case(1)
                do i = 1, n-1
                    cds_matrix%data(i, d) = -1.0d0
                end do
            end select
        end do
    end subroutine build_laplacian_1d_cds
    ! Build a 2D Poisson matrix directly in CDS format
    subroutine build_poisson_2d_cds(cds_matrix, N)
        use iso_fortran_env, only: real64
        implicit none
        type(cds_type), intent(out) :: cds_matrix
        integer, intent(in) :: N
        integer :: total_size, i, j, idx, d
        integer, parameter :: ndiag = 5
        integer, dimension(ndiag) :: offsets

        total_size = N*N
        offsets = (/ -N, -1, 0, 1, N /)
        call cds_init(cds_matrix, total_size, ndiag, offsets)

        ! Loop over all grid points
        do j = 1, N
            do i = 1, N
                idx = (j-1)*N + i
                do d = 1, ndiag
                    if (offsets(d) == -N) then
                        if (j > 1) cds_matrix%data(idx, d) = -1.0d0
                    else if (offsets(d) == -1) then
                        if (i > 1) cds_matrix%data(idx, d) = -1.0d0
                    else if (offsets(d) == 0) then
                        cds_matrix%data(idx, d) = 4.0d0
                    else if (offsets(d) == 1) then
                        if (i < N) cds_matrix%data(idx, d) = -1.0d0
                    else if (offsets(d) == N) then
                        if (j < N) cds_matrix%data(idx, d) = -1.0d0
                    end if
                end do
            end do
        end do
    end subroutine build_poisson_2d_cds

end module cds_matrix
```

B Appendix: Standard Fortran code

```
program cg_solver
 use iso_fortran_env, only: real64
 use cds_matrix

 ! Problem sizes
 integer, parameter :: size = 2**4
 integer, parameter :: s_P = 2**2
 integer, parameter :: size_large = 2**20
 integer, parameter :: s_P_large = 2**10
 real(real64), parameter :: eps = 1.0d-8
 integer, parameter :: max_iteration = 600000
 integer :: iter_test
 integer :: num_runs = 100

 ! Host arrays
!real(real64) :: matrix_full(size,size)

type(cds_type) :: laplace_1d_cds, laplace_1d_cds_large
!real(real64) :: laplace_1d(size,size)

type(cds_type) :: poisson_2d_cds, poisson_2d_cds_large
!real(real64) :: poisson_2d(s_P*s_P, s_P*s_P)

real(real64) :: output(size), input(size)
real(real64) :: output_large(size_large), input_large(size_large)

real(real64) :: t_start, t_end
real(real64) :: sum_time_lap1d = 0.0d0, sum_time_lap1d_cds = 0.0d0, sum_time
real(real64) :: sum_time_poisson2d = 0.0d0, sum_time_poisson2d_cds = 0.0d0, s
real(real64) :: sum_time_randspd = 0.0d0
integer :: sum_iters_lap1d = 0, sum_iters_lap1d_cds = 0, sum_iters_lap1d_cds_
integer :: sum_iters_poisson2d = 0, sum_iters_poisson2d_cds = 0, sum_iters_po
integer :: sum_iters_randspd = 0
integer :: it_lap1d, it_lap1d_cds, it_lap1d_cds_large, it_poisson2d, it_poisso


 ! Initialize Laplacian matrix
!call build_laplacian_1d(laplace_1d, size)
call build_laplacian_1d_cds(laplace_1d_cds, size)
call build_laplacian_1d_cds(laplace_1d_cds_large, size_large)

! Initialize a 2D Poisson matrix
!call build_poisson_2d(poisson_2d, s_P)
call build_poisson_2d_cds(poisson_2d_cds, s_P)
call build_poisson_2d_cds(poisson_2d_cds_large, s_P_large)

! Initialize a random symmetric positive definite matrix
```

```
!call build_symmetric_pd(matrix_full, size)

 input = 1.0d0
 input_large = 1.0d0

 do iter_test = 1, num_runs

     ! Laplacian 1D
     !call cpu_time(t_start)
     !call cg_solve(laplace_1d, input, output, size, eps, max_iteration, it_la
     !call cpu_time(t_end)
     !sum_time_lap1d = sum_time_lap1d + (t_end - t_start)
     !sum_iters_lap1d = sum_iters_lap1d + it_lap1d

     ! Laplacian 1D CDS
     !call cpu_time(t_start)
     !call cg_solve_cds(laplace_1d_cds, input, output, size, eps, max_iteratio
     !call cpu_time(t_end)
     !sum_time_lap1d_cds = sum_time_lap1d_cds + (t_end - t_start)
     !sum_iters_lap1d_cds = sum_iters_lap1d_cds + it_lap1d_cds

     ! Laplacian matrix 1D converted to CDS format LARGE
     !call cpu_time(t_start)
     !call cg_solve_cds(laplace_1d_cds_large, input_large, output_large, size_
     !call cpu_time(t_end)
     !sum_time_lap1d_cds_large = sum_time_lap1d_cds_large + (t_end - t_start)
     !sum_iters_lap1d_cds_large = sum_iters_lap1d_cds_large + it_lap1d_cds_lar

     ! Poisson 2D
     !call cpu_time(t_start)
     !call cg_solve(poisson_2d, input, output, s_P*s_P, eps, max_iteration, it
     !call cpu_time(t_end)
     !sum_time_poisson2d = sum_time_poisson2d + (t_end - t_start)
     !sum_iters_poisson2d = sum_iters_poisson2d + it_poisson2d

     ! Poisson 2D CDS
     call cpu_time(t_start)
     call cg_solve_cds(poisson_2d_cds, input, output, s_P*s_P, eps, max_iterat
     call cpu_time(t_end)
     sum_time_poisson2d_cds = sum_time_poisson2d_cds + (t_end - t_start)
     sum_iters_poisson2d_cds = sum_iters_poisson2d_cds + it_poisson2d_cds

     ! Poisson matrix 2D converted to CDS format LARGE
     call cpu_time(t_start)
     call cg_solve_cds(poisson_2d_cds_large, input_large, output_large, s_P_la
     call cpu_time(t_end)
     sum_time_poisson2d_cds_large = sum_time_poisson2d_cds_large + (t_end - t_
     sum_iters_poisson2d_cds_large = sum_iters_poisson2d_cds_large + it_poisso

     ! Random SPD
```

```
        !call cpu_time(t_start)
        !call cg_solve(matrix_full, input, output, size, eps, max_iteration, it_r
        !call cpu_time(t_end)
        !sum_time_randspd = sum_time_randspd + (t_end - t_start)
        !sum_iters_randspd = sum_iters_randspd + it_randspd

    end do

    call cds_destroy(laplace_1d_cds)
    call cds_destroy(laplace_1d_cds_large)
    call cds_destroy(poisson_2d_cds)
    call cds_destroy(poisson_2d_cds_large)

    print *, '=============================='
    print *, 'Averages over', num_runs, 'iterations:'
    print *, '=============================='
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Laplacian 1D avg time (s):'
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Laplacian 1D CDS avg time (s
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size_large, ' Laplacian 1D CDS LARGE
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Poisson 2D avg time (s):',
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Poisson 2D CDS avg time (s):
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size_large, ' Poisson 2D CDS LARGE a
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Random SPD avg time (s):',
    contains

!Subroutine for Conjugate Gradient solver
      subroutine cg_solve(M, b, x, n, epsilon, imax, iter_count)
      integer, intent(out) :: iter_count
      integer, intent(in) :: n, imax
      real(real64), intent(in) :: M(n,n), b(n), epsilon
      real(real64), intent(out) :: x(n)
      real(real64) :: r(n), d(n), q(n)
      real(real64) :: delta_new, delta_old, delta_0
      real(real64) :: alpha, beta
      real(real64) :: dq_sum
      integer :: iter

      x = 0.0d0

      r = b                    ! residual
      d = r                    ! search direction
      delta_new = dot_product(r, r)
      delta_0   = sqrt(delta_new)

      iter = 0
      do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
         ! q = M * d using intrinsic matmul
         q = matmul(M, d)

         ! d*q using intrinsic dot_product
```

```
      dq_sum = dot_product(d, q)

      alpha = delta_new / dq_sum

      x = x + alpha * d    ! whole-array updates
      r = r - alpha * q

      delta_old = delta_new
      delta_new = dot_product(r, r)

      beta = delta_new / delta_old

      d = r + beta * d
      iter = iter + 1
   end do

   if (iter >= imax-1) then
      print *, '  WARNING: Maximum number of iterations reached before conve
   end if
   iter_count = iter

   !print *, '  Converged in', iter, 'iterations.'
   !print *, '  Residual norm =', sqrt(delta_new)
   !print *, '  Solution vector x (first 10 values):'
   !print '(10F12.4)', x(1:min(10,n))
   end subroutine cg_solve

!Subroutine for CG solve which implements CDS file format
   subroutine cg_solve_cds(M, b, x, n, epsilon, imax, iter_count)
   integer, intent(out) :: iter_count
   integer, intent(in) :: n, imax
   real(real64), intent(in) :: b(n), epsilon
   type(cds_type), intent(in) :: M
   real(real64), intent(out) :: x(n)
   real(real64) :: r(n), d(n), q(n)
   real(real64) :: delta_new, delta_old, delta_0
   real(real64) :: alpha, beta
   real(real64) :: dq_sum
   integer :: iter

   x = 0.0d0

   r = b                ! residual
   d = r                ! search direction
   delta_new = dot_product(r, r)
   delta_0  = sqrt(delta_new)

   iter = 0

   do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
```

```fortran
        ! q = M * d using cds_matvec

        call cds_matvec_parallel(M, d, q)

        ! d*q using intrinsic dot_product
        dq_sum = dot_product(d, q)

        alpha = delta_new / dq_sum

        x = x + alpha * d    ! whole-array updates
        r = r - alpha * q

        delta_old = delta_new
        delta_new = dot_product(r, r)

        beta = delta_new / delta_old

        d = r + beta * d
        iter = iter + 1
    end do

    if (iter >= imax-1) then
        print *, '  WARNING: Maximum number of iterations reached before conve
    end if
    iter_count = iter

    !print *, '  Converged in', iter, 'iterations.'
    !print *, '  Residual norm =', sqrt(delta_new)
    !print *, '  Solution vector x (first 10 values):'
    !print '(10F12.4)', x(1:min(10,n))

    end subroutine cg_solve_cds

! Include matrix building subroutines
include 'matrix_builders.f90'

end program cg_solver
```

C Appendix: OpenACC code

```fortran
program cg_solver
 use iso_fortran_env, only: real64
 use cds_matrix

 ! Problem sizes
 integer, parameter :: size = 2**4
 integer, parameter :: s_P = 2**2
 integer, parameter :: size_large = 2**20
 integer, parameter :: s_P_large = 2**10
 real(real64), parameter :: eps = 1.0d-8
 integer, parameter :: max_iteration = 600000
 integer :: iter_test
 integer :: num_runs = 100

 ! Host arrays
!real(real64) :: matrix_full(size,size)

type(cds_type) :: laplace_1d_cds, laplace_1d_cds_large
!real(real64) :: laplace_1d(size,size)

type(cds_type) :: poisson_2d_cds, poisson_2d_cds_large
!real(real64) :: poisson_2d(s_P*s_P, s_P*s_P)

real(real64) :: output(size), input(size)
real(real64) :: output_large(size_large), input_large(size_large)

real(real64) :: t_start, t_end
real(real64) :: sum_time_lap1d = 0.0d0, sum_time_lap1d_cds = 0.0d0, sum_time
real(real64) :: sum_time_poisson2d = 0.0d0, sum_time_poisson2d_cds = 0.0d0, s
real(real64) :: sum_time_randspd = 0.0d0
integer :: sum_iters_lap1d = 0, sum_iters_lap1d_cds = 0, sum_iters_lap1d_cds_
integer :: sum_iters_poisson2d = 0, sum_iters_poisson2d_cds = 0, sum_iters_po
integer :: sum_iters_randspd = 0
integer :: it_lap1d, it_lap1d_cds, it_lap1d_cds_large, it_poisson2d, it_poiss


 ! Initialize Laplacian matrix
!call build_laplacian_1d(laplace_1d, size)
call build_laplacian_1d_cds(laplace_1d_cds, size)
call build_laplacian_1d_cds(laplace_1d_cds_large, size_large)

! Initialize a 2D Poisson matrix
!call build_poisson_2d(poisson_2d, s_P)
call build_poisson_2d_cds(poisson_2d_cds, s_P)
call build_poisson_2d_cds(poisson_2d_cds_large, s_P_large)

! Initialize a random symmetric positive definite matrix
```

```
!call build_symmetric_pd(matrix_full, size)

 input = 1.0d0
 input_large = 1.0d0

 do iter_test = 1, num_runs

    ! Laplacian 1D
    !call cpu_time(t_start)
    !call cg_solve(laplace_1d, input, output, size, eps, max_iteration, it_la
    !call cpu_time(t_end)
    !sum_time_lap1d = sum_time_lap1d + (t_end - t_start)
    !sum_iters_lap1d = sum_iters_lap1d + it_lap1d

    ! Laplacian 1D CDS
    !call cpu_time(t_start)
    !call cg_solve_cds(laplace_1d_cds, input, output, size, eps, max_iteratio
    !call cpu_time(t_end)
    !sum_time_lap1d_cds = sum_time_lap1d_cds + (t_end - t_start)
    !sum_iters_lap1d_cds = sum_iters_lap1d_cds + it_lap1d_cds

    ! Laplacian matrix 1D converted to CDS format LARGE
    !call cpu_time(t_start)
    !call cg_solve_cds(laplace_1d_cds_large, input_large, output_large, size_
    !call cpu_time(t_end)
    !sum_time_lap1d_cds_large = sum_time_lap1d_cds_large + (t_end - t_start)
    !sum_iters_lap1d_cds_large = sum_iters_lap1d_cds_large + it_lap1d_cds_lar

    ! Poisson 2D
    !call cpu_time(t_start)
    !call cg_solve(poisson_2d, input, output, s_P*s_P, eps, max_iteration, it
    !call cpu_time(t_end)
    !sum_time_poisson2d = sum_time_poisson2d + (t_end - t_start)
    !sum_iters_poisson2d = sum_iters_poisson2d + it_poisson2d

    ! Poisson 2D CDS
    call cpu_time(t_start)
    call cg_solve_cds(poisson_2d_cds, input, output, s_P*s_P, eps, max_iterat
    call cpu_time(t_end)
    sum_time_poisson2d_cds = sum_time_poisson2d_cds + (t_end - t_start)
    sum_iters_poisson2d_cds = sum_iters_poisson2d_cds + it_poisson2d_cds

    ! Poisson matrix 2D converted to CDS format LARGE
    call cpu_time(t_start)
    call cg_solve_cds(poisson_2d_cds_large, input_large, output_large, s_P_la
    call cpu_time(t_end)
    sum_time_poisson2d_cds_large = sum_time_poisson2d_cds_large + (t_end - t_
    sum_iters_poisson2d_cds_large = sum_iters_poisson2d_cds_large + it_poisso

    ! Random SPD
```

```
        !call cpu_time(t_start)
        !call cg_solve(matrix_full, input, output, size, eps, max_iteration, it_r
        !call cpu_time(t_end)
        !sum_time_randspd = sum_time_randspd + (t_end - t_start)
        !sum_iters_randspd = sum_iters_randspd + it_randspd

    end do

    call cds_destroy(laplace_1d_cds)
    call cds_destroy(laplace_1d_cds_large)
    call cds_destroy(poisson_2d_cds)
    call cds_destroy(poisson_2d_cds_large)

    print *, '=========================='
    print *, 'Averages over', num_runs, 'iterations:'
    print *, '=========================='
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Laplacian 1D avg time (s):'
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Laplacian 1D CDS avg time (s
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size_large, ' Laplacian 1D CDS LARGE
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Poisson 2D avg time (s):',
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Poisson 2D CDS avg time (s):
    print '(A,I0,A,F15.10,A,I0)', 'Size: ', size_large, ' Poisson 2D CDS LARGE a
    !print '(A,I0,A,F15.10,A,I0)', 'Size: ', size, ' Random SPD avg time (s):',
    contains

!Subroutine for Conjugate Gradient solver
        subroutine cg_solve(M, b, x, n, epsilon, imax, iter_count)
        integer, intent(out) :: iter_count
        integer, intent(in) :: n, imax
        real(real64), intent(in) :: M(n,n), b(n), epsilon
        real(real64), intent(out) :: x(n)
        real(real64) :: r(n), d(n), q(n)
        real(real64) :: delta_new, delta_old, delta_0
        real(real64) :: alpha, beta
        real(real64) :: dq_sum
        integer :: iter

        x = 0.0d0

        r = b                   ! residual
        d = r                   ! search direction
        delta_new = dot_product(r, r)
        delta_0   = sqrt(delta_new)

        iter = 0
        do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
            ! q = M * d using intrinsic matmul
            q = matmul(M, d)

            ! d*q using intrinsic dot_product
```

```fortran
      dq_sum = dot_product(d, q)

      alpha = delta_new / dq_sum

      x = x + alpha * d    ! whole-array updates
      r = r - alpha * q

      delta_old = delta_new
      delta_new = dot_product(r, r)

      beta = delta_new / delta_old

      d = r + beta * d
      iter = iter + 1
   end do

   if (iter >= imax-1) then
      print *, '  WARNING: Maximum number of iterations reached before conve
   end if
   iter_count = iter

   !print *, '  Converged in', iter, 'iterations.'
   !print *, '  Residual norm =', sqrt(delta_new)
   !print *, '  Solution vector x (first 10 values):'
   !print '(10F12.4)', x(1:min(10,n))
   end subroutine cg_solve

!Subroutine for CG solve which implements CDS file format
   subroutine cg_solve_cds(M, b, x, n, epsilon, imax, iter_count)
   integer, intent(out) :: iter_count
   integer, intent(in) :: n, imax
   real(real64), intent(in) :: b(n), epsilon
   type(cds_type), intent(in) :: M
   real(real64), intent(out) :: x(n)
   real(real64) :: r(n), d(n), q(n)
   real(real64) :: delta_new, delta_old, delta_0
   real(real64) :: alpha, beta
   real(real64) :: dq_sum
   integer :: iter

   x = 0.0d0

   r = b                  ! residual
   d = r                  ! search direction
   delta_new = dot_product(r, r)
   delta_0  = sqrt(delta_new)

   iter = 0

   do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
```

```
      ! q = M * d using cds_matvec

      call cds_matvec_parallel(M, d, q)

      ! d*q using intrinsic dot_product
      dq_sum = dot_product(d, q)

      alpha = delta_new / dq_sum

      x = x + alpha * d    ! whole-array updates
      r = r - alpha * q

      delta_old = delta_new
      delta_new = dot_product(r, r)

      beta = delta_new / delta_old

      d = r + beta * d
      iter = iter + 1
   end do

   if (iter >= imax-1) then
      print *, '  WARNING: Maximum number of iterations reached before conve
   end if
   iter_count = iter

   !print *, '  Converged in', iter, 'iterations.'
   !print *, '  Residual norm =', sqrt(delta_new)
   !print *, '  Solution vector x (first 10 values):'
   !print '(10F12.4)', x(1:min(10,n))

   end subroutine cg_solve_cds

! Include matrix building subroutines
include 'matrix_builders.f90'

end program cg_solver
```

D Appendix: CUDA code

```
program cg_solver
use cudafor
use cds_matrix
use iso_fortran_env, only: real64
implicit none

! Add device memory attributes for arrays used in CUDA kernels
real(real64), device, allocatable, dimension(:) :: d_d, d_r, d_q, d_x, d_b, c
real(real64), device, allocatable, dimension(:,:) :: d_M

integer, parameter :: size = 2**4
integer, parameter :: s_P = 2**2
integer, parameter :: size_large = 2**20
integer, parameter :: s_P_large = 2**10
real(real64), parameter :: eps = 1.0d-8
integer, parameter :: max_iteration = 600000
integer :: iter_test
integer :: num_runs = 100

!real(real64), allocatable :: laplace_1d(:,:), matrix_full(:,:), poisson_2d(:
type(cds_type) :: laplace_1d_cds, laplace_1d_cds_large
type(cds_type) :: poisson_2d_cds, poisson_2d_cds_large
real(real64), allocatable :: output(:), input(:), output_large(:), input_larg

real(real64) :: t_start, t_end

real(real64) :: sum_time_lap1d = 0.0d0, sum_time_lap1d_cds = 0.0d0, sum_time
real(real64) :: sum_time_poisson2d = 0.0d0, sum_time_poisson2d_cds = 0.0d0, s
real(real64) :: sum_time_randspd = 0.0d0

integer :: sum_iters_lap1d = 0, sum_iters_lap1d_cds = 0, sum_iters_lap1d_cds_
integer :: sum_iters_poisson2d = 0, sum_iters_poisson2d_cds = 0, sum_iters_po
integer :: sum_iters_randspd = 0
integer :: it_lap1d, it_lap1d_cds, it_lap1d_cds_large, it_poisson2d, it_poiss

!allocate(laplace_1d(size,size), matrix_full(size,size), poisson_2d(s_P*s_P,s
allocate(input(size), output(size), input_large(size_large), output_large(siz

 ! Initialize Laplacian matrix
!call build_laplacian_1d(laplace_1d, size)
call build_laplacian_1d_cds(laplace_1d_cds, size)
call build_laplacian_1d_cds(laplace_1d_cds_large, size_large)

! Initialize a 2D Poisson matrix
!call build_poisson_2d(poisson_2d, s_P)
call build_poisson_2d_cds(poisson_2d_cds, s_P)
call build_poisson_2d_cds(poisson_2d_cds_large, s_P_large)
```

```fortran
! Initialize a random symmetric positive definite matrix
!call build_symmetric_pd(matrix_full, size)

 input = 1.0d0
 input_large = 1.0d0

 do iter_test = 1, num_runs
     ! Laplacian 1D
     !call cpu_time(t_start)
     !call cg_solve(laplace_1d, input, output, size, eps, max_iteration, it_la
     !call cpu_time(t_end)
     !sum_time_lap1d = sum_time_lap1d + (t_end - t_start)
     !sum_iters_lap1d = sum_iters_lap1d + it_lap1d

     ! Laplacian 1D CDS
     !call cpu_time(t_start)
     call cg_solve_cds(laplace_1d_cds, input, output, size, eps, max_iteration
     call cpu_time(t_end)
     sum_time_lap1d_cds = sum_time_lap1d_cds + (t_end - t_start)
     sum_iters_lap1d_cds = sum_iters_lap1d_cds + it_lap1d_cds

     ! Laplacian matrix 1D converted to CDS format LARGE
     call cpu_time(t_start)
     call cg_solve_cds(laplace_1d_cds_large, input_large, output_large, size_l
     call cpu_time(t_end)
     sum_time_lap1d_cds_large = sum_time_lap1d_cds_large + (t_end - t_start)
     sum_iters_lap1d_cds_large = sum_iters_lap1d_cds_large + it_lap1d_cds_larg

     ! Poisson 2D
     !call cpu_time(t_start)
     !call cg_solve(poisson_2d, input, output, s_P*s_P, eps, max_iteration, it
     !call cpu_time(t_end)
     !sum_time_poisson2d = sum_time_poisson2d + (t_end - t_start)
     !sum_iters_poisson2d = sum_iters_poisson2d + it_poisson2d

     ! Poisson 2D CDS
     call cpu_time(t_start)
     call cg_solve_cds(poisson_2d_cds, input, output, s_P*s_P, eps, max_iterat
     call cpu_time(t_end)
     sum_time_poisson2d_cds = sum_time_poisson2d_cds + (t_end - t_start)
     sum_iters_poisson2d_cds = sum_iters_poisson2d_cds + it_poisson2d_cds

     ! Poisson matrix 2D converted to CDS format LARGE
     call cpu_time(t_start)
     call cg_solve_cds(poisson_2d_cds_large, input_large, output_large, s_P_la
     call cpu_time(t_end)
     sum_time_poisson2d_cds_large = sum_time_poisson2d_cds_large + (t_end - t_
     sum_iters_poisson2d_cds_large = sum_iters_poisson2d_cds_large + it_poisso
```

```
        ! Random SPD
        !call cpu_time(t_start)
        !call cg_solve(matrix_full, input, output, size, eps, max_iteration, it_r
        !call cpu_time(t_end)
        !sum_time_randspd = sum_time_randspd + (t_end - t_start)
        !sum_iters_randspd = sum_iters_randspd + it_randspd
    end do

    call cds_destroy(laplace_1d_cds)
    call cds_destroy(laplace_1d_cds_large)
    call cds_destroy(poisson_2d_cds)
    call cds_destroy(poisson_2d_cds_large)

    print *, '=========================='
    print *, 'Averages over', num_runs, 'iterations:'
    print *, '=========================='
    !print '(A,I0,A,F15.10,A,I0)', 'Size:', size, ' Laplacian 1D avg time (s):',
    print '(A,I0,A,F15.10,A,I0)', 'Size:', size, ' Laplacian 1D CDS avg time (s)
    print '(A,I0,A,F15.10,A,I0)', 'Size:', size_large, ' Laplacian 1D CDS LARGE
    !print '(A,I0,A,F15.10,A,I0)', 'Size:', size, ' Poisson 2D avg time (s):', s
    print '(A,I0,A,F15.10,A,I0)', 'Size:', size, ' Poisson 2D CDS avg time (s):'
    print '(A,I0,A,F15.10,A,I0)', 'Size:', size_large, ' Poisson 2D CDS LARGE av
    !print '(A,I0,A,F15.10,A,I0)', 'Size:', size, ' Random SPD avg time (s):', s
contains

  subroutine cg_solve_cds(M_host, b_host, x_host, n, epsilon, imax, iter_count)
    integer, intent(out) :: iter_count
    integer, intent(in) :: n, imax
    type(cds_type), intent(in) :: M_host
    real(real64), intent(in) :: b_host(n), epsilon
    real(real64), intent(out) :: x_host(n)
    ! Device variables
    real(real64), device, allocatable, target :: d_data(:,:), d_b(:), d_x(:), c
    integer, device, allocatable, target :: d_offsets(:)
    real(real64), allocatable :: r(:), d(:), q(:)
    real(real64) :: delta_new, delta_old, delta_0, alpha, beta, dq_sum
    integer :: iter, istat, ndiag

    ! Allocate and copy CDS matrix components to device
    ndiag = M_host%ndiag
    allocate(d_data(n, ndiag))
    allocate(d_offsets(ndiag))
    d_data = M_host%data
    d_offsets = M_host%offsets

    ! Allocate device vectors
    allocate(d_b(n), d_x(n), d_r(n), d_d(n), d_q(n))
    d_b = b_host
    d_x = 0.0d0
    d_r = d_b
```

```fortran
d_d = d_r

! Allocate host workspace for final result
allocate(r(n), d(n), q(n))

! Initial residual norm
delta_new = 0.0d0
!$cuf kernel do(1) <<<*,128>>>, reduction(+:delta_new)
do i = 1, n
  delta_new = delta_new + d_r(i) * d_r(i)
end do
delta_0 = sqrt(max(delta_new, tiny(1.0d0)))
delta_old = delta_new
iter = 0

do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
   ! q = M*d (CDS matvec)
   call cds_matvec(M_host%n, M_host%ndiag, d_data, d_offsets, d_d, d_q)

   ! dq_sum = d^T q
   dq_sum = 0.0d0
   !$cuf kernel do(1) <<<*,128>>>, reduction(+:dq_sum)
   do i = 1, n
     dq_sum = dq_sum + d_d(i) * d_q(i)
   end do

   alpha = delta_new / max(dq_sum, tiny(1.0d0))
   !$cuf kernel do(1) <<<*,128>>>
   do i = 1, n
     d_x(i) = d_x(i) + alpha * d_d(i)
     d_r(i) = d_r(i) - alpha * d_q(i)
   end do

   delta_old = delta_new
   delta_new = 0.0d0
   !$cuf kernel do(1) <<<*,128>>>, reduction(+:delta_new)
   do i = 1, n
     delta_new = delta_new + d_r(i) * d_r(i)
   end do

   beta = delta_new / max(delta_old, tiny(1.0d0))
   !$cuf kernel do(1) <<<*,128>>>
   do i = 1, n
     d_d(i) = d_r(i) + beta * d_d(i)
   end do

   iter = iter + 1
end do

! Copy result back to host
```

```
  x_host = d_x

  if (iter >= imax-1) then
     print *, '  WARNING: Maximum number of iterations reached before converg
  end if
  !print *, '  Converged in', iter, 'iterations.'
  !print *, '  Residual norm =', sqrt(delta_new)
  iter_count = iter

  ! Deallocate device arrays
  if (allocated(d_data)) deallocate(d_data)
  if (allocated(d_offsets)) deallocate(d_offsets)
  if (allocated(d_b)) deallocate(d_b)
  if (allocated(d_x)) deallocate(d_x)
  if (allocated(d_r)) deallocate(d_r)
  if (allocated(d_d)) deallocate(d_d)
  if (allocated(d_q)) deallocate(d_q)
  if (allocated(r)) deallocate(r)
  if (allocated(d)) deallocate(d)
  if (allocated(q)) deallocate(q)
end subroutine cg_solve_cds


subroutine cg_solve(M, b, x, n, epsilon, imax, iter_count)
  integer, intent(out) :: iter_count
  integer, intent(in) :: n, imax
  real(real64), intent(in) :: M(n,n), b(n), epsilon
  real(real64), intent(out) :: x(n)
  real(real64) :: r(n), d(n), q(n)
  real(real64) :: delta_new, delta_old, delta_0
  real(real64) :: alpha, beta, dq_sum
  integer :: iter, istat

  ! Allocate device memory
  allocate(d_d(n), d_r(n), d_q(n), d_x(n), d_b(n), d_M(n,n), stat=istat)
  if (istat /= 0) stop 'Failed to allocate device memory'

  ! Initialize host arrays
  x = 0.0d0
  r = b
  d = r

  ! Copy initial data to device
  d_b = b
  d_M = M
  d_x = x
  d_r = r
  d_d = d

  ! Compute initial residual norm squared on device
```

```
delta_new = 0.0d0
!$cuf kernel do(1) <<<*,128>>>, reduction(+:delta_new)
do i = 1, n
  delta_new = delta_new + d_r(i) * d_r(i)
end do

delta_0 = sqrt(max(delta_new, tiny(1.0d0)))  ! Ensure non-zero
delta_old = delta_new
iter = 0

do while (iter < imax .and. sqrt(delta_new) > (epsilon * delta_0))
   ! Compute q = A*d on device
   call matvec(d_d, d_q, n)

   ! Compute dq_sum = d^T * q and update x, r in a single kernel
   dq_sum = 0.0d0
   delta_old = delta_new
   delta_new = 0.0d0

   ! First, compute alpha = delta_old / (d^T * q)
   dq_sum = 0.0d0
   !$cuf kernel do(1) <<<*,128>>>, reduction(+:dq_sum)
   do i = 1, n
     dq_sum = dq_sum + d_d(i) * d_q(i)
   end do

   ! Update x and r
   alpha = delta_old / max(dq_sum, tiny(1.0d0))  ! Avoid division by zero
   !$cuf kernel do(1) <<<*,128>>>
   do i = 1, n
     d_x(i) = d_x(i) + alpha * d_d(i)
     d_r(i) = d_r(i) - alpha * d_q(i)
   end do

   ! Compute new residual norm
   delta_new = 0.0d0
   !$cuf kernel do(1) <<<*,128>>>, reduction(+:delta_new)
   do i = 1, n
     delta_new = delta_new + d_r(i) * d_r(i)
   end do

   ! Update search direction
   beta = delta_new / max(delta_old, tiny(1.0d0))  ! Avoid division by zero
   !$cuf kernel do(1) <<<*,128>>>
   do i = 1, n
     d_d(i) = d_r(i) + beta * d_d(i)
   end do

   iter = iter + 1
end do
```

```
  ! Copy final result back to host
  x = d_x

  if (iter >= imax-1) then
     print *, '  WARNING: Maximum number of iterations reached before converg
  end if

  !print *, '  Converged in', iter, 'iterations.'
  !print *, '  Residual norm =', sqrt(delta_new)
  !print *, '  Solution vector x (first 10 values):'
  !print '(10F12.4)', x(1:min(10,n))
  ! Copy result back to host
  x = d_x
  iter_count = iter

  ! Free device memory (if allocated)
  if (allocated(d_d)) deallocate(d_d)
  if (allocated(d_r)) deallocate(d_r)
  if (allocated(d_q)) deallocate(d_q)
  if (allocated(d_x)) deallocate(d_x)
  if (allocated(d_b)) deallocate(d_b)
  if (allocated(d_M)) deallocate(d_M)
end subroutine cg_solve

!Multiply a vector with a Matrix in full format using CUDA parallelization
subroutine matvec(d_in, d_out, nn)
  integer, intent(in) :: nn
  real(real64), device, intent(in) :: d_in(nn)
  real(real64), device, intent(out) :: d_out(nn)
  integer :: i, jj
  real*8 :: sum_val

  ! Perform matrix-vector multiplication on device
  !$cuf kernel do(1) <<<*,128>>>
  do i = 1, nn
    sum_val = 0.0d0
    do jj = 1, nn
      sum_val = sum_val + d_M(i,jj) * d_in(jj)
    end do
    d_out(i) = sum_val
  end do
end subroutine matvec

!Multiply a vector with a Matrix in CDS format using CUDA parallelization
subroutine cds_matvec(n, ndiag, data, offsets, vector, result)
  integer, value :: n, ndiag
  real(real64), device :: data(n, ndiag)
  integer, device :: offsets(ndiag)
  real(real64), device :: vector(n)
```

```
    real(real64), device :: result(n)
    integer :: i, j, offset
    real(real64) :: sum_val

    !$cuf kernel do(1) <<<*,128>>>
    do i = 1, n
      sum_val = 0.0_real64
      do j = 1, ndiag
          offset = offsets(j)
          if (offset < 0) then
              if (i + offset >= 1) sum_val = sum_val + data(i, j) * vector(i +
          else
              if (i + offset <= n) sum_val = sum_val + data(i, j) * vector(i +
          end if
      end do
      result(i) = sum_val
    end do
  end subroutine cds_matvec


  include 'matrix_builders.f90'

end program cg_solver
```

E Appendix: Matrix building code

```
! Creates a 1D Laplacian matrix (tridiagonal with 2 on diagonal,
! -1 on off-diagonals)
subroutine build_laplacian_1d(A, n)
  use iso_fortran_env, only: real64
  implicit none
  integer, intent(in) :: n
  real(real64), intent(out) :: A(n,n)
  integer :: i

  A = 0.0d0
  do i = 1, n
    A(i,i) = 2.0d0
    if (i > 1) A(i, i-1) = -1.0d0
    if (i < n) A(i, i+1) = -1.0d0
  end do
end subroutine build_laplacian_1d

! Creates a random symmetric positive definite matrix
subroutine build_symmetric_pd(A, n, shift)
  use iso_fortran_env, only: real64
  implicit none
  integer, intent(in) :: n
  real(real64), intent(out) :: A(n,n)
  real(real64), optional, intent(in) :: shift
  real(real64) :: C(n,n)
  real(real64) :: diag_shift
  integer :: i

  ! Default shift to make matrix positive definite
  diag_shift = 1.0d-3
  if (present(shift)) diag_shift = shift

  ! Create random matrix
  call random_seed()
  call random_number(C)

  ! Make it symmetric positive definite: A = C^T * C + shift*I
  A = matmul(transpose(C), C)
  do i = 1, n
    A(i,i) = A(i,i) + diag_shift
  end do
end subroutine build_symmetric_pd

! Creates a 2D Poisson matrix (5-point stencil for 2D Laplacian)
subroutine build_poisson_2d(A, N)
  use iso_fortran_env, only: real64
  implicit none
```

```fortran
  integer, intent(in) :: N
  integer :: total_size
  real(real64), intent(out) :: A(N*N, N*N)
  integer :: i, j, idx, neighbor

  total_size = N*N
  A = 0.0d0

  do j = 1, N
    do i = 1, N
      idx = (j-1)*N + i   ! Flattened index
      A(idx, idx) = 4.0d0

      ! Left neighbor
      if (i > 1) then
        neighbor = idx - 1
        A(idx, neighbor) = -1.0d0
      end if

      ! Right neighbor
      if (i < N) then
        neighbor = idx + 1
        A(idx, neighbor) = -1.0d0
      end if

      ! Bottom neighbor
      if (j > 1) then
        neighbor = idx - N
        A(idx, neighbor) = -1.0d0
      end if

      ! Top neighbor
      if (j < N) then
        neighbor = idx + N
        A(idx, neighbor) = -1.0d0
      end if
    end do
  end do
end subroutine build_poisson_2d
```