Sparse matrix-vector products using Numba.

September 26, 2023

1 Contents

1. Task 1.
2. Sparse Matrix Formats: Converting from CSR to Ellpack-R using a Numba routine.
3. CPU and GPU accelerated SpMV
4. Benchmarking CPU and GPU SpMV

1.1 Task 1

Currently, there is a wide variety of ways in which sparse matrices can be stored, all of which store the non-zero elements of the matrix. This is an especially useful task when conducting sparse matrix-vector products (SpMV), which are widely used in image processing, simulation and other areas of scientific computing and engineering. To compute SpMVs we only need to consider contributions from non-zero elements, which helps optimise this computation. However, how these non-zero elements are stored can have massive impacts on the performance of SpMV computations. In this assignment, we will be focusing on Compressed Sparse Row (CSR) and Ellpack-R (ELL-R) sparse matrix formats, exploring their effects on the computation time of SpMVs, their integration into GPGPU computing on NVIDIA GPUs and discussing alternative storage formats that stem from CSR and ELL-R, providing further optimisation.

The most intuitive format for storing sparse matrices is the coordinate (COO) format, where each data entry is logged alongside its column and row indices in, three separate one-dimensional vectors. Regarding SpMV based on the COO format, there is no implicit information retained in the coordinate ordering. Similarly, for a multi-threaded SpMV, atomic data access is used (meaning the lowest level kind of data is accessed.) when outputting elements of the final vector.
The development from the COO format is the CSR format, which holds arrays for the non-zero data, the corresponding column indices, and a final array containing pointers indicating which elements in the data and column index arrays mark the beginning of one row and the end of the previous one. However, for SpMV the CSR format has some severe drawbacks when considering a linear system of the form $A \cdot v = u$ Firstly, the use of pointers to indicate row locations in the CSR format means that accessing elements of the multiplication vector v, isn't accessed locally since we're indirectly accessing elements via a pointer. Thus since the elements of v aren't directly accessed from the memory, we have slower execution times for accessing the memory. Secondly, fine-grained parallelism isn't fully exploited due to different numbers of row elements having to be iterated over compared to the vast number of column indices being iterated over.
From this, the ELLPACK format has been formed which consist of 2D arrays which convert the sparse matrix into a dense matrix with fewer columns. These arrays are then stored in column-major ordering which allows the utilisation of coalesced global memory access, where consecutive threads access data points that are contiguous in global memory. Similarly, ELLPACK improves

on CSR because thread blocks do not have to be executed synchronously, meaning each thread block will work independently of each other without the need of having to wait for the others.
ELLPACK-R (ELL-R) develops on its predecessor ELLPACK by defining the number of non-zero elements in each row of the sparse matrix, which allows us to reduce the number of iterations required per row to calculate the i^{th} element of the output vector. This storage format further improves both ELLPACK and CSR by maximising the usage of warps in the GPU architecture. Zeroes are used to pad the rows of the dense matrix to be multiples of 16 (half-warp). Within thread warps themselves, individual threads execute the same instructions simultaneously and stop executing once their iteration finishes. Finally, the occupancy of the thread warps is maximised, meaning we minimise the number of idle threads in a warp.
Further developing on ELL-R is PELLR [1], which introduces the permutation of rows in the dense matrices created by ELL-R, sorting them by the number of non-zero elements in each row. In ELL-R there is the possibility for a large difference between the maximum number of non-zero elements in a row of the matrix and the number of non-zero elements in a given row. When rows are grouped into warps there arises the possibility of one row having many non-zero elements and another row having few, which will result in more computational work and more time with threads being idle. Once rows are sorted in the PELLR format, more saturated rows share the same warps, and less saturated rows share the same warps. This minimises unnecessary calculations by reducing the number of iterations required to calculate an element while optimising storage format.

1.2 Sparse Matrix Formats: Converting from CSR to Ellpack-R using a Numba routine.

1.2.1 CPU and GPU accelerated SpMV

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import numba
from numba import cuda
import math
import scipy
from scipy import sparse
import timeit
import time
```

[2]:
```
from scipy.sparse.linalg import LinearOperator

class EllpackMatrix(LinearOperator):

  def __init__(self, csr, GPU=False):

    csr_data = csr.data.astype(np.float32)
    csr_indices = csr.indices.astype(np.int32)
    pointers = csr.indptr.astype(np.int32)

    # Initialising ELLPACK-R arrays.

    self.gpu = GPU
    self.shape = csr.get_shape()
    self.rows = int(len(csr.indptr) - 1) # defining number of rows in the␣
↪matrix/vector
    self.max_nzr = int(np.amax(pointers[1:] - pointers[:-1])) # initialising␣
↪the maximum number of non-zero elements

    self.data = np.zeros((self.rows, self.max_nzr), dtype=np.float32)
    self.col_ind = np.zeros((self.rows, self.max_nzr), dtype=np.int32)
    self.rl = pointers[1:] - pointers[:-1]

    self.to_ellpack(csr_data, csr_indices, pointers, self.data, self.col_ind) #␣
↪converting the csr matrix into ellpack-r form.

    # Initialising the required variables, reshaping data and uploading arrays␣
↪to GPU for the CUDA kernel.

    if self.gpu == True:
      self.initialise_gpu()


 @staticmethod
 @numba.njit(parallel=True)
 def to_ellpack(csr_data, csr_indices, csr_pointers, data, col_ind):
    """
    CPU implementation converting a CSR matrix format into an Ellpack-R matrix␣
↪format.
    ==============================================
    Inputs:

    csr_data[:] - type: float - array: contains the data of the CSR format␣
↪sparse matrix.
    csr_indices[:] - type: int - array: contains the column indices of the␣
↪corresponding csr_data entries of the sparse matrix.
    csr_pointers[:] - type: int - array: contains pointers to which elements in␣
↪csr_data[:] and csr_indices[:] mark the beginning of one row and the end of␣
↪the last.
    data[:,:] - type: float - array: contains the sparse matrix data in the␣
↪Ellpack-R format (padded with zeros)
    col_ind[:,:] - type: int - array: contains the sparse matrix column indices␣
↪for the non-zero elements in each row (padded with zeros).
    ==============================================

    Updates data, col_ind and rl which are arrays for the Ellpack-R format.
    """
```

```
        for pointer in numba.prange(len(csr_pointers) - 1): # storing the csr␣
↪matrix in ellpackr format.
            data[pointer, :(csr_pointers[pointer + 1] - csr_pointers[pointer])] =␣
↪csr_data[csr_pointers[pointer] : csr_pointers[pointer + 1]]
            col_ind[pointer, :(csr_pointers[pointer + 1] - csr_pointers[pointer])] =␣
↪csr_indices[csr_pointers[pointer] : csr_pointers[pointer + 1]]


    def initialise_gpu(self):
        """
        Generating parameters and uploading arrays to the device relevant for GPU␣
↪computation.
        """

        global tpb, bpg, data_device, col_ind_device, rl_device, u_device

        # tpb = math.ceil(self.rows/32) * 32 # defining block size and grid size.
        # bpg = self.max_nzr

        # data = np.zeros((self.max_nzr, tpb))
        # col_ind = np.zeros((self.max_nzr, tpb))

        # for i in range(self.rows): # padding the rows of my ellpack-r form␣
↪matrices so that each row calculation is acted on by a block of threads.
        #   data[i, :self.rl[i]] = self.data[i, :self.rl[i]]
        #   col_ind[i, :self.rl[i]] = self.col_ind[i, :self.rl[i]]

        tpb = 128 # defining block size and grid size.
        bpg = math.ceil(self.rows/tpb)

        data_device = cuda.to_device(self.data.flatten(order='F')) # uploading to␣
↪gpu device.
        col_ind_device = cuda.to_device(self.col_ind.flatten(order='F'))
        rl_device = cuda.to_device(self.rl)

        u = np.zeros((self.rows))
        u_device = cuda.to_device(u)



    def _matvec(self, vec):
        """
        Overwriting the matvec '@' operator to execute my CPU/GPU parallelised␣
↪matvec function.
        """
        vec = np.asanyarray(vec)
```

```
  if vec.shape != (self.shape[0],) and vec.shape != (self.shape[1],1):
      raise ValueError('dimension mismatch')


  if self.gpu == True: # GPU parallelisation

    vec_device = cuda.to_device(vec)
    # start = time.time()
    self.gpu_matvec[bpg, tpb](vec, data_device, col_ind_device, rl_device,␣
↪u_device)
    # interval = time.time() - start
    # print(interval)
    u = u_device.copy_to_host()

    return u

  else: # CPU parallelisation

    return self.cpu_matvec(vec, self.rows, self.data, self.col_ind, self.rl)

@staticmethod
@numba.njit(parallel=True)
def cpu_matvec(vec, rows, data, col_ind, rl):
   """
   CPU parallelised matrix-vector product using a matrix stored in ELLPACK-R␣
↪format.
   ==============================================
   Inputs:

   vec[:] - type: float - array: the vector being multiplied with the matrix.
   rows - type: int - scalar: number of rows in the matrix/vector.
   data[:,:] - type: float -  array: contains the float elements within the␣
↪matrix.
   col_ind[:,:] - type: int - array: contains the column indices for␣
↪corresponding to each data point for a givne row.
   rl[:] - type: int - array: contains the number of non-zero elements in each␣
↪row of the matrix.
   ==============================================
   Outputs:

   result[:] - type: float - array: contains the final resulting array after␣
↪matvec multiplication.
   ==============================================
   """
   # Initialising arrays.
```

```
    result = np.zeros((rows))

    for row in numba.prange(rows):
      for column in range(int(rl[row])):
        result[row] += data[row, column] * vec[int(col_ind[row, column])] #␣
↪computation of matvec accelerated using cpu.

    return result

  @staticmethod
  @cuda.jit()
  def gpu_matvec(vec, data, col_ind, rl, u):
    """
    GPU parallelised matrix-vector product for vectors stored in the Ellpack-R␣
↪sparse matrix format.
    ==============================================
    Inputs:

    vec[:] - type: float - array: contains the vector elements of the vector␣
↪being multiplied by.
    data[:,:] - type: float - array: row-wise flattened vector containing the␣
↪matrix elements of the sparse matrix.
    col_ind[:,:] - type: float - array: row-wise flattened vector containing␣
↪the column indices corresponding to each row in the sparse matrix.
    rl[:] - type: int - array: contains the number of non-zero elements in each␣
↪row of the matrix.
    u[:] - type: float - array: contains the final matrix-vector product result.
    ==============================================

    Updates the elements of the array u to contain the corresponding solutions␣
↪to the matrix-vector product.
    """

    # shared_data = cuda.shared.array(shape=(tpb), dtype=numba.float32)
    # shared_col_ind = cuda.shared.array(shape=(tpb), dtype=numba.int32)

    gx = cuda.grid(1) # loading relevant gpu parameters
    tx = cuda.threadIdx.x
    bdx = cuda.blockDim.x
    bid = cuda.blockIdx.x

    # Below is a row-major method for CUDA accelerated matvec - currently␣
↪slower than Numba CPU implementation
    # Using column order but padding rows up to half warps in size (or the␣
↪nearest warp size that includes max_nzr), this will hopefully improve␣
↪coalescence?
```

```
    # Create a shared array for the result calculated by a half-warp.

    # if bid < bpg:
    #   result = 0
    #   if tx < bid*bdx + rl[bid]:
    #     shared_data[tx] = data[(bid*bdx) + tx] # loading data into shared␣
 ↪arrays
    #     shared_col_ind[tx] = col_ind[(bid*bdx) + tx]

    #     cuda.syncthreads()

    #     for tx in range(rl[bid]):
    #       result +=  shared_data[tx] * vec[shared_col_ind[tx]] # summing over␣
 ↪the shared result

    #       cuda.syncthreads()

    # u[bid] = result

    num_threads = vec.shape[0]

    if gx < num_threads:
      result = 0
      for i in range(rl[gx]):
        result += data[i*num_threads + gx] * vec[col_ind[i*num_threads + gx]]

      cuda.syncthreads()

      u[gx] = result


# I'm not iterating over all of the blocks?
# Stores the relevant vec elements and data elements for only a single row.
# Storing multiplication of a single row in the entire final result.
# Currently for this matrix (or small matrices in general) my thread block size␣
 ↪is too large and encompassing the entire matrix in a single thread.
# This means when I specify the global thread to run for the first block (which␣
 ↪is the only block here).
# Solution could be to pad the rows of the flattened array with zeros so they␣
 ↪have length tpb (?).
# Use LU decomposition to decompose the matrix.
# Takes a dense matrix, and makes it into two triangular matrix (one upper and␣
 ↪one lower) - makes it easier.
```

1.2.2 Benchmarking CPU and GPU SpMV

Now that a class converting from the CSR format to the Ellpack-R format has been defined we can begin to benchmark and check the accuracy of the results obtained for SpMV operations.
Firstly, the following cell compares the precision of GPU/CPU accelerated SpMV to SciPy's CSR SpMV, and we aim to have our results' errors be in the order of machine precision when looking at the relative distances between our implementation and SciPy's.

[3]:
```
# Testing for accuracy within machine precision using a randomly generated␣
 ↪sparse matrix and 3 random vectors.
# Generate a table for recording the relative distance (for GPU and CPU␣
 ↪computations) between the Ellpack-R matvec and CSR matvec.

from tabulate import tabulate

A = sparse.csr_matrix(sparse.random(1000,1000, density=0.05, dtype=np.float32))␣
 ↪# Initialising matrix and arrays for testing.
vec1 = np.random.rand(A.shape[0]).astype(np.float32)
vec2 = np.random.rand(A.shape[0]).astype(np.float32)
vec3 = np.random.rand(A.shape[0]).astype(np.float32)

arr = np.array([vec1, vec2, vec3])

cpu = EllpackMatrix(A) # creating instances for gpu and cpu accelerated matvec␣
 ↪product.
gpu = EllpackMatrix(A, GPU=True)

gpu_err = ['GPU']
cpu_err = ['CPU']

for vec in arr: # calculating relative distance for different random vectors
  gpu_rel = np.linalg.norm(A @ vec - gpu @ vec) / np.linalg.norm(A @ vec)
  gpu_err.append(gpu_rel)

  cpu_rel = np.linalg.norm(A @ vec - cpu @ vec) / np.linalg.norm(A @ vec)
  cpu_err.append(cpu_rel)

print('For float32 numbers, machine precision is', np.finfo(np.float32).eps)
print('=================================================================')
print("Calculating the relative error between calculating SpMVs using CPU/GPU␣
 ↪acceleration and those using SciPy's Linear Operator")
print('=================================================================')
print(tabulate([gpu_err, cpu_err], headers=['Parallelisation', 'Vector 1',␣
 ↪'Vector 2', 'Vector 3']))
```

```
/usr/local/lib/python3.7/dist-packages/numba/np/ufunc/parallel.py:363:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 9107. The TBB
```

```
threading layer is disabled.
  warnings.warn(problem)

For float32 numbers, machine precision is 1.1920929e-07
=======================================================================
Calculating the relative error between calculating SpMVs using CPU/GPU
acceleration and those using SciPy's Linear Operator
=======================================================================
Parallelisation       Vector 1     Vector 2     Vector 3
-----------------  -----------  -----------  -----------
GPU                1.01293e-07  1.07811e-07  1.04362e-07
CPU                1.01293e-07  1.07811e-07  1.04362e-07
```

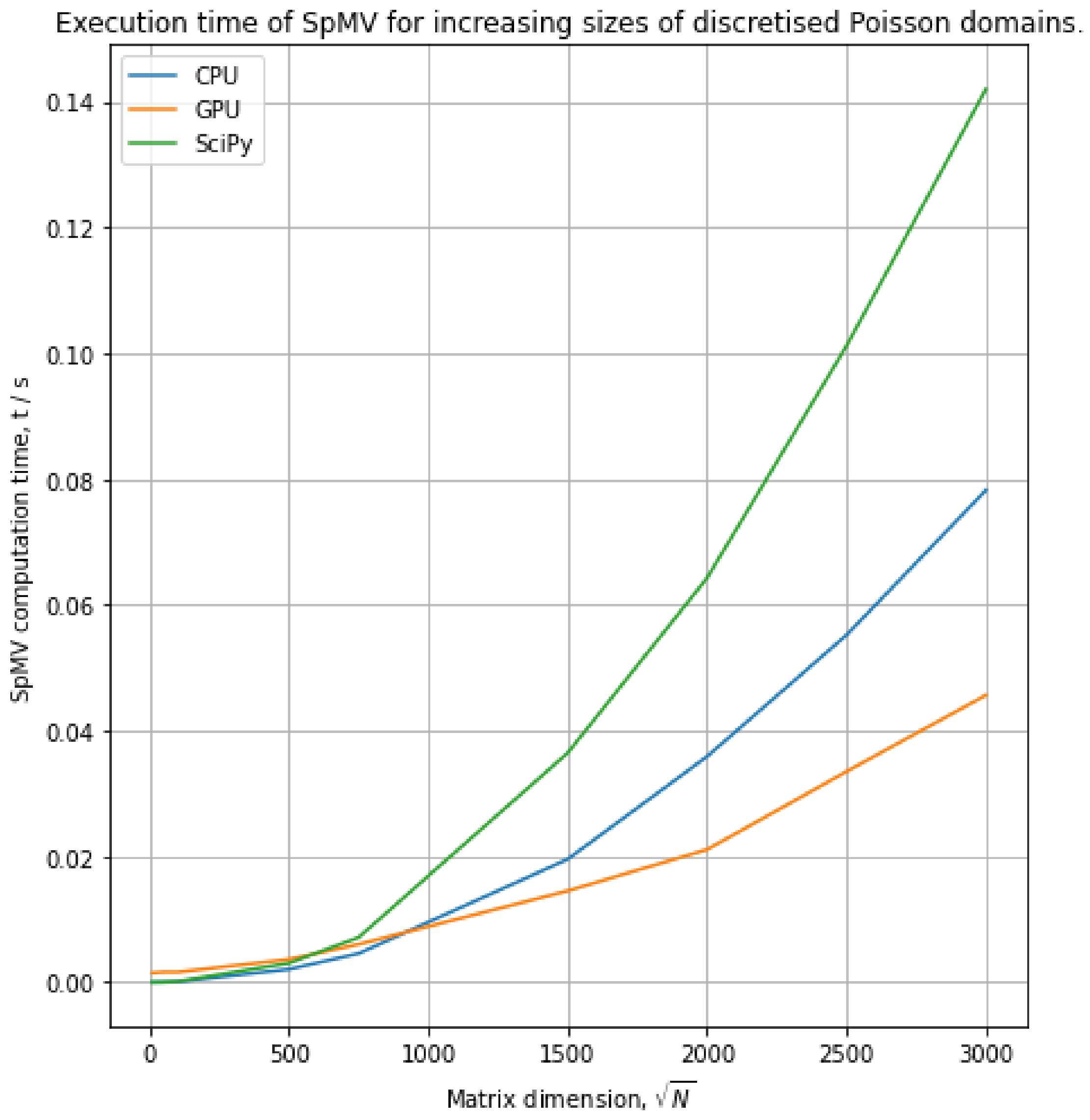
Since I'm using a GPU architecture, the maximum precision that can be used is 32-bit hence why I'm making comparisons to the machine precision of decimal32 numbers.
Our results using CPU/GPU implementations are within machine precision.

[4]:
```python
from scipy.sparse import coo_matrix

def discretise_poisson(N):
    """Generate the matrix and rhs associated with the discrete Poisson␣
 ↪operator."""

    nelements = 5 * N**2 - 16 * N + 16

    row_ind = np.empty(nelements, dtype=np.float32)
    col_ind = np.empty(nelements, dtype=np.float32)
    data = np.empty(nelements, dtype=np.float32)

    f = np.empty(N * N, dtype=np.float32)

    count = 0
    for j in range(N):
        for i in range(N):
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = col_ind[count] = j * N + i
                data[count] =  1
                f[j * N + i] = 0
                count += 1

            else:
                row_ind[count : count + 5] = j * N + i
                col_ind[count] = j * N + i
                col_ind[count + 1] = j * N + i + 1
                col_ind[count + 2] = j * N + i - 1
                col_ind[count + 3] = (j + 1) * N + i
                col_ind[count + 4] = (j - 1) * N + i
                data[count] = 4 * (N - 1)**2
                data[count + 1 : count + 5] = - (N - 1)**2
                f[j * N + i] = 1

                count += 5

    return coo_matrix((data, (row_ind, col_ind)), shape=(N**2, N**2)).tocsr(), f
```

[ ]:
```
# Generating a matrices (increasing in time) using discretise_poisson and␣
 ↪benchmarking its computation time for a single matvec.

N = np.array([10, 50, 100, 500, 750, 1500, 2000, 2500, 3000])
A = []
x = []

for n in N:
  matrix, b = discretise_poisson(n)
  A.append(matrix)
  x.append(np.random.randn(matrix.shape[0]))

A = np.array(A)
x = np.array(x)

# Benchmarking
cpu_times = []
gpu_times = []
csr_times = []
density = []

for i in range(len(N)):
  # GPU timings

  gpu_mat = EllpackMatrix(A[i], GPU=True)
  gpu = %timeit -o -q -n 10 gpu_mat @ x[i]
  gpu_times.append(gpu.best)

  # CPU timings

  cpu_mat = EllpackMatrix(A[i])
  cpu = %timeit -o -q -n 10 cpu_mat @ x[i]
  cpu_times.append(cpu.best)

  csr = %timeit -o -q -n 10 A[i] @ x[i]
  csr_times.append(csr.best)
```

[6]:
```
# PLotting computation times for GPU and CPU matvecs.
```

```
fig = plt.figure(figsize=(16,8))
ax = fig.add_subplot(1,2,1)

ax.plot(N, cpu_times, label='CPU')
ax.plot(N, gpu_times, label='GPU')
ax.plot(N, csr_times, label='SciPy')
ax.set_xlabel(r'Matrix dimension, $\sqrt{N}$')
ax.set_ylabel('SpMV computation time, t / s')
ax.set_title('Execution time of SpMV for increasing sizes of discretised␣
 ↪Poisson domains.')
ax.legend()
ax.grid()
```



From the above plot we can see that as matrix dimensions (N^2) increase our GPU and CPU implementations of SpMV far exceed SciPy's CSR matvec. For low matrix dimensions, there are too few non-zero elements in the sparse matrix and the thread allocation of the GPU architecture isn't fully utilised as we cannot utilise more threads during computation (simply because there isn't enough computation to be done!).
However, as we reach matrices with dimensions 9,000,000 we see that the GPU implementation far exceeds the execution time of both the CPU and SciPy implementations.

[14]:
```
# Benchmarking using larger matrices from the matrix market.

from scipy.io import mmread

# uploading matrices from the matrix market and converting them from COO format␣
 ↪to CSR format.

matrix_1 = mmread("fidap011").tocsr() # non-symmetric sparse matrix
matrix_2 = mmread("s3dkq4m2.mtx").tocsr() # symmetric dense sparse matrix.

# timing comparisons for the larger matrix market matrices.

# iterating and timing

arr = np.array([matrix_1, matrix_2])

v1 = np.random.randn(matrix_1.shape[0])

GPU_times = ['GPU']
CPU_times = ['CPU']
scipy_times = ['SciPy']
count = 0
for mat in arr:
  count += 1
  a1 = EllpackMatrix(mat, GPU=True)
  v = np.random.randn(a1.shape[0])
  t1 = %timeit -o -q -n 10 a1 @ v
  GPU_times.append(t1.best)

  a2 = EllpackMatrix(mat)
  t2 = %timeit -o -q -n 10 a2 @ v
  CPU_times.append(t2.best)

  t3 = %timeit -o -q -n 10 mat @ v
  scipy_times.append(t3.best)

  rho = np.sum(a2.rl) / (a2.shape[0] * a2.shape[1])
  print('The percentage density of matrix {} is'.format(count), rho*100)
print("==========================================================================================
print("SpMV execution times (s) for CPU, GPU and SciPy on a symmetric and␣
 ↪non-symmetric matrix from the matrix market")
print("==========================================================================================
print(tabulate([GPU_times, CPU_times, scipy_times], headers=['SpMV',␣
 ↪'Non-symmetric', 'Symmetric']))
```

```
The percentage density of matrix 1 is 0.39538520634558766
The percentage density of matrix 2 is 0.058927738308510544
================================================================================
=====================
SpMV execution times (s) for CPU, GPU and SciPy on a symmetric and non-symmetric
matrix from the matrix market
================================================================================
=====================
SpMV      Non-symmetric    Symmetric
------  ---------------  -----------
GPU          0.00177297   0.00321461
CPU          0.00170399   0.00802565
SciPy        0.00133498   0.00727283
```

I have chosen these matrices from the matrix market to demonstrate the utility of the ELLPACK-R format because they display many different factors that play into the computation time of SpMV when accelerated using processors.

For matrix 1, we have a non-symmetric matrix of relatively small dimensions (~17000) compared to those generated using the *discretise_poisson* function. Comparing the CPU and GPU execution times for this matrix, there is a time difference of the order 10^{-5}, with the CPU being slightly faster. The non-symmetric nature of the matrix implies that the GPU implementation will be less efficient during SpMV because of the optimisations discussed in the PELLR storage format. Due to the non-symmetric nature of the matrix, there will be large variation between the between the number of non-zero elements in thread in a warp, leading to increased wait times for threads during computation in a warp when this difference is large. Despite being the denser matrix of the two we see that GPU execution time is slower for this matrix

On the other hand, matrix 2 is extremely large with dimensions over $90,000$, and is almost a factor of 10 less dense than matrix 1. However, the faster GPU execution time for this matrix (which is faster than that of matrix 1 and the CPU and SciPy SpMV execution times for matrix 2.), reinforces the idea that for GPU implementation, the number of rows rather than the density of the matrix, increases computation time as it allows for more parallelisation across threads, which can be further optimised within threads blocks and warps.

To further imrpove the performance of these algorithms, the implementation of the PELLR format could be introduced, in conjucation or alternatively to the Sliced Ellpack (S-ELL) which spreads thread calculations over rows, leading to multiple threads calculating a contribution of a single final vector element and then being summed together.

[1] Wang, Z. and Gu, T., 2020. PELLR: A Permutated ELLPACK-R Format for SpMV on GPUs. Journal of Computer and Communications, 8(4), pp.44-58.

[7]:

```

```