Assignment3

Contents

Task1

1.1 Difficulties of the CSR format on GPUs

There are several drawbacks of the matvecs(y=Av) computation based on CSR format.

1.The locality of memory access of vector v[ ] cannot be guaranteed.

Locality is a kind of predictable behavior in computer system, the program's performance will be imporved with a strong locality reference. Common locality includes Temporal locality and Spatial locality. Former one means that if a location is referenced at a certain time, it will be accessed again not long afterwards, while the latter refers to the fact that if a memory location is referenced at a certain time, memory near that location will be accessed shortly after **[1]**.Here we analyze matvec operations based on the CRS format mainly in terms of the maintenance of spatial locality. The code is as follows **[2]**:

```
import numba
@numba.jit(nopython=True, parallel=True)
def csr_matvec(data, indices, indptr, shape, v):
        """Evaluates the matrix-vector product with a CSR matrix."""
        # Get the rows and columns
        m, n = shape
        y = np.zeros(m, dtype=np.float64)
        for row_index in numba.prange(m):
                col_start = indptr[row_index]
                col_end = indptr[row_index + 1]
                for col_index in range(col_start, col_end):
                        y[row_index] += data[col_index] * v[indices[col_index]]
        return y
```

We can see that in the loop we access the elements of v by v[indices[col_index]], but the addresses of indices[col_index] are not contiguous in memory. It can be seen that the spatial locality of accessing elements in v is not maintained. Indirect addressing of elements in v consumes program time.

2.Not take advantage of fine-grained parallelism

From the above code, we can see that the number of iterations of the inner loop is small and variable. This makes it difficult to do the usual fine-grained parallel optimization methods such as loop chunking or expansion, and there is limited room for parallel optimization using SIMD.

1.2 How Ellpack and Ellpack-R solve these difficultues

When we use a thread whose index.x=i to compute an element i of u vector, and store the arrays with column-major order,we can improve the SpMV performance based on ELLPACK format.The reasons for improvement are as following:

- With the column-major order, we can coalesce global memory access, making accesses of threads to A[ ], J[ ] and v[ ] are on the continuous memory address. This operation maintain the spatial locality.
- It could save the time of the communication between blocks because every threads complete computation without synchronization with other blocks.

However,the ELLPACK format is not faultless. A high percentage of zeros and a quite amount of padding zeros in the data structure decrease its efficiency. Because we have to repeat to excute conditional branches to reduce memory access to zero elements in the loops. This kind of operation really decreases the performance.

Despite this, the ELLPACK-R format help solve ELLPACK's problem. We maintain the N*Max_nzr dimension arrays: A[ ],J[ ] in ELLPACK, and add an integer array rl[ ] (N demension), which records number of non-zero elements in every rows of A[ ].This operation make us not need to excute conditional branches everytime to avoid redundant accesses. And at the same time, some implementation of ELLPACK-R like ELLR-T can enhance the performance further more. In this implementation, we use T threads to compute the specific i element in u vector.With the assistance of shared memory utilization, padding and alignment, etc, we can make the best use of our GPU resource. In this experiment, we implement ELLR-T, so more analysis of this will be mentioned afterwards.

1.3 Overview on GPU based matvecs

Matvecs operation is a significant part in the computation related to sparse matrix.At the same time, GPU based matvecs can really improve the performance of SpMV.In this part, we give an overview on modern GPU based matvecs.

In 2013, Xing Liu et al designs an efficient SpMV kernel on Intel R Xeon PhiTM 1 Coprocessor, codenamed Knights Corner(KNC), which is a microprocessors with extremely high memory

bandwidth. They design the ELLPACK Sparse Block(ESB), which is a novel sparse matrix format tuned for KNC. In this format, they manage to improve the SIMD efficiency of the kernel with finite-window sorting,encoding of non-zero positions to reduce bandwidth requirements and improve the spatial locality of memory access by column locality **[3]**.

In 2015, Liu and Vinter proposes a brand new Sparse storage format—Compressed Sparse Row5(CSR5) and designed a SpMV algorithms based on it. This format is extended from CSR format, fixing one of three arrays of CSR format, and tile-transposing other two other arrays. Meanwhile, two groups of extra auxiliary information are added. Such optimizatoin saves the cost of extra space and are very SIMD-friendly **[4]**.

In 2016, Duane Merrill and Michael Garland discussed the tactics of CsvMV parallelization, including row splitting and nonzero splitting. Row splitting allocates the regularized portion of a long line to multiple processes in orter to limit the number of digital data items allocated to each processor.The partial sum from the relevant sub rows can be summarized in the subsequent process.While nonzero splitting is to allocate equal split of nonzero elements into different processor, which is a better way to deal with imbalance of nonzeros in format**[5]**.

Recently, researchers induced deep learning method into SpMV. Cao Zhongxiao et al uses CNN based on dual_channel sparse matrix feature fusion and other constructor features to predict SpMV computation performance and auto-tune the SpMV algrithom, making the algorithms compatible with different sparse matrixes**[6]**.

Task2

```
import numpy as np
from math import ceil
import numba
from numba import cuda
from scipy.sparse import coo_matrix,csr_matrix
from scipy.sparse.linalg import LinearOperator
```

2.1 Implement a new class EllpackMatrix

We create a subclass EllpackMatrix derived from scipy.sparse.linalg.LinearOperator. By calling EllpackMatrix, we can convert Scipy CSR matrix format to Ellpack-R format. In the term of matvec operation, we implement EllR-T**[9]** based on numba.cuda, codes are shown below. .

```
@cuda.jit
def ELLRT(A,adjusted_A,adjusted_J,rl,x,u):#We suppose use 32 threads to compute in every
    bs=128
    T=32
    #bs=np.float32(4*T)
    sharing=cuda.shared.array((bs,),numba.float32)#Allocate share memory space
```

```
    N=A.shape[0]#A  matrix's  row  number  in  ELLPACKR  format
    gap=ceil(N*T/16)*16
    idb=cuda.threadIdx.x
    idx=cuda.grid(1)
    idp=idb%T
    i=idx//T#refer  to  ith  row
    if  (i<N):
        svalue=0.0
        max=ceil(rl[i]/T)
        for  k  in  range(max):
            position=int(k*gap+i*T+idp)
            value=adjusted_A[position]
            col=adjusted_J[position]
            col_pos=int(col)
            svalue=svalue+value*x[col_pos]
        sharing[idb]=svalue
        cuda.syncthreads()
        if  idp<16:
            sharing[idb]=sharing[idb]+sharing[idb+16]
            cuda.syncthreads()
        if  idp<8:
            sharing[idb]=sharing[idb]+sharing[idb+8]
            cuda.syncthreads()
        if  idp<4:
            sharing[idb]=sharing[idb]+sharing[idb+4]
            cuda.syncthreads()
        if  idp<2:
            sharing[idb]=sharing[idb]+sharing[idb+2]
            cuda.syncthreads()
        if  idp==0:
            u[i]=sharing[idb]+sharing[idb+1]
            cuda.syncthreads()
```

We store the A[ ] and J[ ] into 1-D arrays in column-major order, and also we pad zeros to make sure every length of row as multiple of 16. This operation help coalesce and align memory segments to reduce the frequency of memory accesses.

```
def  PaddingAndAlign(A,T=32):
    N=A.shape[0]
    max_nzr=A.shape[1]
    m=int(ceil(max_nzr/T))
    ad_colNum=m*T
    row_padding=np.zeros((N,ad_colNum-max_nzr),np.float32)
    adjusted_A=np.hstack((A,row_padding))
    #-------------------------------------------------------------------#
    num_per_group=N*TypeError
    #print(num_per_group)

    #-------------------------------------------------------------------#
    if  isinstance((num_per_group/16),int):
        result=np.zeros((m*num_per_group,),np.float32)
```

```
        for  k  in  range(m):
            result[k*num_per_group:(k+1)*num_per_group]=adjusted_A[:,k*T:(k+1)*T].flatten()

    else:
        ad_num_per_group=int(ceil(num_per_group/16)*16)
        #print(ad_num_per_group)
        result=np.zeros((m*ad_num_per_group,),np.float32)
        for  k  in  range(m):
            result[k*ad_num_per_group:k*ad_num_per_group+num_per_group]=adjusted_A[:,k*T:(k+1)*T].f
    #print(result.shape)
    return  result



class  EllpackMatrix(LinearOperator):
    def  __init__(self,csr):
        self.shape=csr.shape
        self.dtype=csr.dtype
        super().__init__(dtype=self.dtype,  shape=self.shape)
        self.N=len(csr.indptr)-1
        self.rl=[int(len(csr.data[csr.indptr[i]:csr.indptr[i+1]]))  for  i  in  range(self.N)]
        self.max_nzr=max(self.rl)
        self.A=np.zeros((self.N,self.max_nzr),np.float32)
        self.J=np.zeros((self.N,self.max_nzr),np.float32)

        for  i  in  range(self.N):
            for  j  in  range(self.rl[i]):
                self.A[i,j]=csr.data[csr.indptr[i]:csr.indptr[i+1]][j]
                self.J[i,j]=csr.indices[csr.indptr[i]:csr.indptr[i+1]][j]
        T=32
        self.adjusted_A=PaddingAndAlign(self.A,T)
        self.adjusted_J=PaddingAndAlign(self.J,T)
        self.device_A=cuda.to_device(self.A)
        self.device_adA=cuda.to_device(self.adjusted_A)
        self.device_adJ=cuda.to_device(self.adjusted_J)
        self.device_rl=cuda.to_device(self.rl)
        self.device_u=cuda.device_array((self.shape[0],),dtype=np.float32)
        self.BPG=int(ceil(self.shape[0]/4))+1
        self.TPB=int(4*T)
        self.result=np.zeros((self.shape[0],),np.float32)
    def  _matvec(self,x):
        #self.device_x=cuda.to_device(x)
        #ELLRT[self.BPG,self.TPB](self.device_A,self.device_adA,self.device_adJ,self.device_rl,self
        ELLRT[self.BPG,self.TPB](self.device_A,self.device_adA,self.device_adJ,self.device_rl,x,sel
        self.device_u.copy_to_host(self.result)
        return  self.result

    def  __matmul__(self,x):
        return  self._matvec(x)
```

2.2 Test relative distance between Scipy csr matvec and Ellpack-R

We create a 1000*1000 sparse matrix with density=0.5, and three random vectors v1,v2,v3.

```
import  scipy  as  spy
n=1000
m=1000
density=0.5
matrixformat='csr'
S1=spy.sparse.rand(m,n,density=density,format=matrixformat,dtype=np.float32,random_state=42)
np.random.seed(1234)
v1=np.random.randn(m)
np.random.seed(1235)
v2=np.random.randn(m)
np.random.seed(1236)
v3=np.random.randn(m)
```

```
ELLR=EllpackMatrix(S1)
```

Test on v1

```
ELL_result=ELLR@v1
CSR_result=S1@v1
rel_error  =  np.linalg.norm(ELL_result  -  CSR_result,  np.inf)  /  np.linalg.norm(CSR_result,  np
print(rel_error)
```

```
7.689158252693797e-08
```

Test on v2

```
ELL_result2=ELLR@v2
CSR_result2=S1@v2
rel_error  =  np.linalg.norm(ELL_result2  -  CSR_result2,  np.inf)  /  np.linalg.norm(CSR_result2,
print(rel_error)
```

```
1.0017166557357096e-07
```

Test on v3

```
ELL_result3=ELLR@v3
CSR_result3=S1@v3
rel_error  =  np.linalg.norm(ELL_result3  -  CSR_result3,  np.inf)  /  np.linalg.norm(CSR_result3,
print(rel_error)
```

```
9.056612197434149e-08
```

From test results, we find that the relative distance is around 1.2×10^{-7},which is the order of

2.3 Compare times of Ellpack-R matvec to the CSR matvec

We implement this comparison on the basis of the coefficient matrix of discretise possion problem.

```
import numpy as np
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.float64)
        col_ind = np.empty(nelements, dtype=np.float64)
        data = np.empty(nelements, dtype=np.float64)
        f = np.empty(N * N, dtype=np.float64)
        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
```

```
from time import time
csr_time=[]
ell_time=[]
for N in [100*i for i in range(1,16)]:
    possion_coefficiences_matrix,_=discretise_poisson(N)
    ell=EllpackMatrix(possion_coefficiences_matrix)
    test_vector=np.random.randn(N*N)
    #csr_begin_t=time()
    ct=%timeit -o -q -n5 csr_result=possion_coefficiences_matrix@test_vector
    #csr_end_t=time()
```

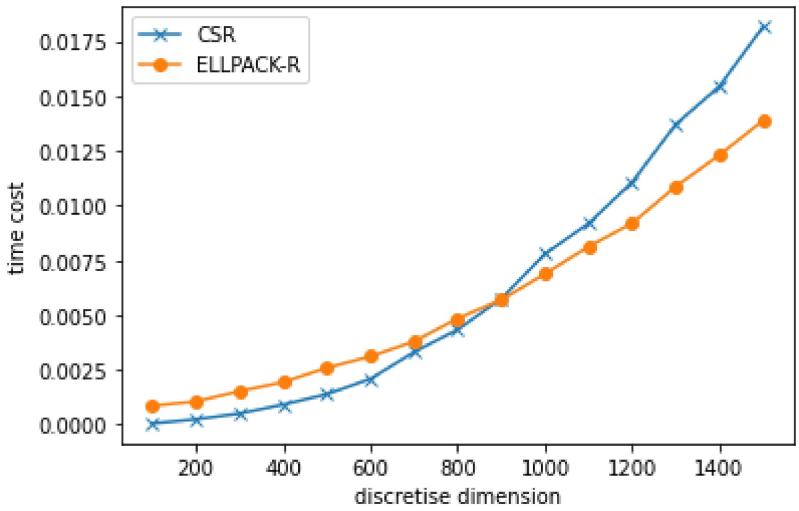
```
    csr_time.append(ct.best)
    #ell_begin_t=time()
    et=%timeit  -o  -q  -n5  ell_result=ell@test_vector
    #ell_end_t=time()
    ell_time.append(et.best)
```

```
print(csr_time)
print(ell_time)
```

```
[5.228019999776734e-05, 0.00023911240004963473, 0.0005029089999879944, 0.0009111977999964437,
[0.0008575476000260096, 0.0010537278000811057, 0.0015270208000401908, 0.0019366284000170708,
```

```
import  matplotlib.pyplot  as  plt
N=[100*i  for  i  in  range(1,16)]
l1,=plt.plot(N,csr_time,'x-',label='CSR')
l2,=plt.plot(N,ell_time,'o-',label="ELLPACK-R")
plt.legend(handles=[l1,l2],labels=['CSR','ELLPACK-R'])
plt.xlabel('discretise  dimension')
plt.ylabel('time  cost')
```

```
Text(0, 0.5, 'time cost')
```



We benchmark the consuming time of matvec both on csr format and ELLPACK-R format, we found that after N=800(matrix dimension=640000),ELLPACK-R's time cost is exceeded by CSR format.So we can conclude that the ELLPACK-R's high efficiency is shown when matrix size is large enough. Because in ELLR-T algorithm, we use T threads to compute one row, which means, when the size of the matrix is not large enough, some threads are idle, which causes that the parallelism of allocated threads are not fully used, and the acceleration potential are not exploited.

2.4 Compare times of Ellpack-R matvec to the CSR matvec based on other sparse matrices

```
from  scipy  import  io
```

From matrix market, we buy two matrixes

- dwt_1007:

```
1007  x  1007,  8575  entries
Average  nonzeros  per  row  :  8.5
Standard  deviation  :  1.2
```

- fidapm11:

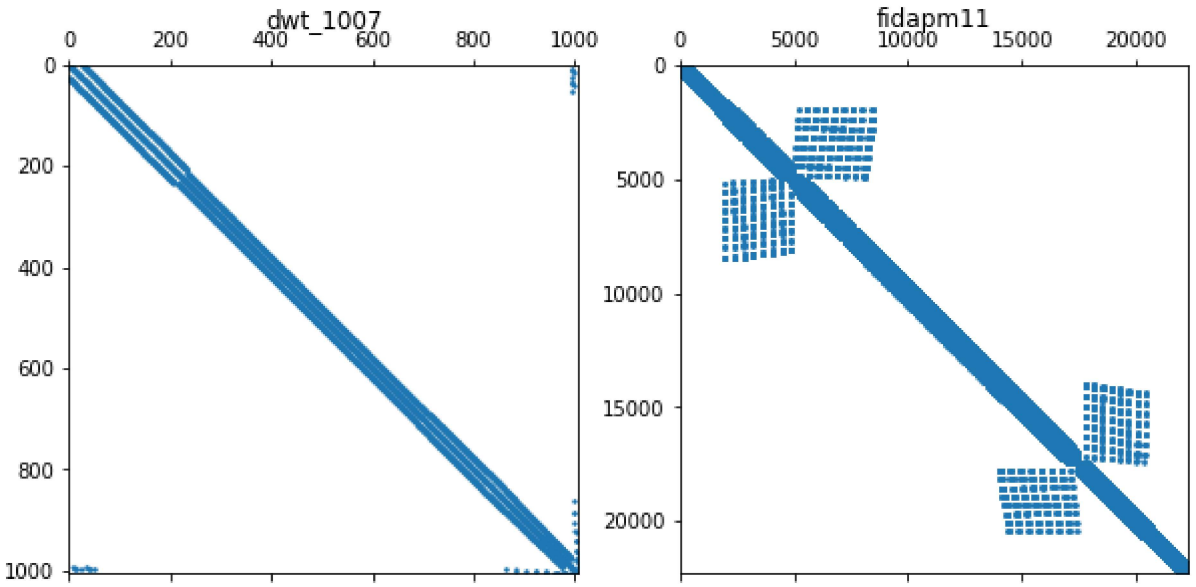
```
22294  x  22294,  623554  entries
Average  nonzeros  per  row  :  28
Standard  deviation  :  7.1
```

```
dwt_1007=io.mmread('/content/dwt_1007.mtx.gz')
fida=io.mmread('/content/fidapm11.mtx.gz')
#lshp=scipy.io.mmread('/content/lshp2233.mtx.gz')
```

Matrix visualisation

```
fig,ax=plt.subplots(1,2,figsize=(10,  10))
ax[0].spy(dwt_1007,markersize=1)
ax[0].set_title('dwt_1007')
ax[1].spy(fida,markersize=1)
ax[1].set_title('fidapm11')
```

```
Text(0.5,  1.05,  'fidapm11')
```



Benchmark dwt_1007

```
csr_dwt=csr_matrix(dwt_1007)
ell_dwt=EllpackMatrix(csr_dwt)
np.random.seed(1234)
test_vector=np.random.randn(ell_dwt.shape[1])
ct=%timeit -o csr_result=csr_dwt@test_vector
test_vector_cuda=cuda.to_device(test_vector)
et=%timeit  -o  ell_result=ell_dwt@test_vector_cuda
print('csr_time:  ',ct.best)
print('ell_time:  ',et.best)
```

```
The slowest run took 4.21 times longer than the fastest. This could mean that an intermediate
100000 loops, best of 5: 14.1 µs per loop
1000 loops, best of 5: 409 µs per loop
csr_time:  1.4123299339998994e-05
ell_time:  0.0004094577729993034
```

Benchmark fidapm11

```
csr_fida=csr_matrix(fida)
ell_fida=EllpackMatrix(csr_fida)
np.random.seed(1234)
test_vector=np.random.randn(ell_fida.shape[1])
ct=%timeit  -o  csr_result=csr_fida@test_vector
test_vector_cuda=cuda.to_device(test_vector)
et=%timeit  -o  ell_result=ell_fida@test_vector_cuda
print('csr_time:  ',ct.best)
print('ell_time:  ',et.best)
```

```
1000 loops, best of 5: 661 µs per loop
1000 loops, best of 5: 435 µs per loop
csr_time:  0.0006610030090005239
ell_time:  0.00043547985199984395
```

From benchmark result, we find that on the fidamp11 matrix, ELLPACK-R format has a better performance. We can explore the backgroud reason from matrix's parameters, we can see that the size of matrix fidapm11 is larger and the matrix contains more average nozeros per row. In this case, we can let threads compute homogeneously in warps, taking an advantage of high parallelism**[8]**. Also, we find standard deviation of row nonzeros is big, which means the number difference of nonzeros of different rows is larger. For this case, the rl[ ] in ELLPACK-R comes in handy. With the nonzeros' numbers recorded in rl[ ], we reduce the unbalance of the threads of one warp and we do not need to excute too much time-consuming useless iterations and the control of loops**[8]**, which really saves the time.

Reference

[1] https://en.wikipedia.org/wiki/Locality_of_reference

[2] https://tbetcke.github.io/hpc_lecture_notes/sparse_data_structures.html

[3] Liu X, Smelyanskiy M, Chow E, Dubey P. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors[C]. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS'13, 2013, 273-282.

[4] Liu W, Vinter B. CSR5:An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication[C]. In Proceedings of the 29th ACM International Conference on Supercomputing, ICS'15, 2015, 339-350.

[5] Liu W, Wang H, Vinter B. Parallel Segmented Merge and Its Applications to Two Sparse Matrix Kernels[C]. In 2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, 2018.

[6] 曹中潇,冯仰德,王珏,闵维潇,姚铁锤,高岳,王丽华,高付海.基于深度学习的稀疏矩阵向量乘运算性能模型[J/OL].计算机工程:1-8[2021-12-01].

[7] F. Vázquez, G. Ortega, J. J. Fernández and E. M. Garzón, "Improving the Performance of the Sparse Matrix Vector Product with GPUs," 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1146-1151, doi: 10.1109/CIT.2010.208.

[8] F. V'zquez, G. Ortega, J. J. Fern'ndez, I. García and E. M. Garzón, "Fast Sparse Matrix Matrix Product Based on ELLR-T and GPU Computing," 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, 2012, pp. 669-674, doi: 10.1109/ISPA.2012.99.