COL380 Assignment 4
GPU Matrix Multiplication

Nischay Diwan
2020CS50433

April 2023

1 Approach 1

The initial approach of the algorithm was using the compressed sparse row matrix representation of Matrix A. Then the algorithm iterates over all the blocks of the resultant matrix initialized to zero and computes this block. Here for each block of the resultant matrix, we iterate over only non-zero blocks of Matrix A and then find the corresponding block if it exists in Matrix B. Then, if the blocks are non-zero, it is sent to the device for parallel multiplication for this block. So, only 1 block is computed simultaneously but by m^2 parallel threads. If the block is non-zero, the output is written. This algorithm performs equivalently or slower than using a CPU.

2 Approach 2

First, the cuda memory is allocated repeatedly and transfers only 2 blocks of data. This is changed to allocating and sending a complete matrix(non-zero blocks only) once at the start. This finally brings some speedup into the algorithm. But the main change in approach 2 is to allow parallel computation of all the blocks. So each block in n^2/m^2 output is computed by each block of GPU using m^2 threads. This brings about 25-30x speedup. It also allows scalability on large sizes and dense matrices.

3 Approach 3

Final cuda memory allocation and transfer code:

```
void matMul(vector<array<int,3>> &mp1, vector<uint> &blksA, vector<array<int,3>> &mp2,
    vector<uint> &blksB,  long long n, long long m, vector<uint> &blksC){
  long long nm = n/m;
  // sending data to GPU
  int streamSize = 2;
  cudaError_t err;
  cudaStream_t stream[streamSize];
  for(int i = 0;i<streamSize ;i++){
    cudaStreamCreate(&stream[i]);
  }
  size_t size = sizeof(uint);
  size_t size2 = sizeof(int);
  size_t size3 = sizeof(uint) * n * n;
  uint *a = &blksA[0], *b = &blksB[0], *da, *db;
  uint *c = &blksC[0], *dc;
  cudaMalloc(&da,size*blksA.size());
  cudaMalloc(&db,size*blksB.size());
  cudaMalloc(&dc,size3);
  cudaDeviceSynchronize();
  cudaMemset(dc,0,size3);
  cudaMemcpyAsync(da,a,(size_t)size*(size_t)blksA.size(),cudaMemcpyHostToDevice,stream
    [0]);
  cudaMemcpyAsync(db,b,(size_t)size*(size_t)blksB.size(),cudaMemcpyHostToDevice,stream
    [1]);
```

Initially, 2 cuda streams are generated, and both matrices in non-zero blocks are transferred, as shown in the above code. Here initial memory allocation is done. 2 streams help in parallel data transfers.

```
// converting to CSR
vector<int> valV;
vector<int> colV;
vector<int> rofV;
int offset = 0;
int rowno = 0;
rofV.push_back(offset);
for(int i = 0; i <mp1.size() ; i++){
  colV.push_back(mp1[i][1]);
  valV.push_back(mp1[i][2]);
  if(mp1[i][0] > rowno){
    for(int cc = 0;cc<(mp1[i][0]-rowno);cc++){
      rofV.push_back(offset);
    }
    rowno = mp1[i][0];
  }
  offset+=1;
}
for(int j = rowno;j<nm;j++){
  rofV.push_back(blksA.size()/m/m);
}
vector<int> valV2;
vector<int> cofV;
vector<int> rowV;
offset = 0;
int colno = 0;
cofV.push_back(offset);
for(int i = 0; i <mp2.size() ; i++){
  rowV.push_back(mp2[i][0]);
  valV2.push_back(mp2[i][2]);
  if(mp2[i][1] > colno){
    for(int cc = 0;cc<(mp2[i][1]-colno);cc++){
      cofV.push_back(offset);
    }
    colno = mp2[i][1];
  }
  offset+=1;
}
for(int j = colno;j<nm;j++){
  cofV.push_back(blksB.size()/m/m);
}
std::cout << "CSR converted\n";
```

In the above part of the code, the non-zero blocks are iterated, and matrices A and B are converted to Compressed Sparse Matrix Representations. The difference here is that both are converted to sparse form, and later, binary search is replaced with 2 pointer approach(this is a major speedup: about 10-15x). This also reduces the data to be transferred and also faster.

```
int *ka;
cudaMalloc(&ka,(size_t)size2*(size_t)mp1.size());
cudaMemcpyAsync(ka,valV.data(),(size_t)size2*(size_t)mp1.size(),cudaMemcpyHostToDevice,
  stream[0]);
int *kb;
cudaMalloc(&kb,(size_t)size2*(size_t)mp2.size());
cudaMemcpyAsync(kb,valV2.data(),(size_t)size2*(size_t)mp2.size(),cudaMemcpyHostToDevice
  ,stream[1]);
int *rof, *col, *cof, *row;
cudaMalloc(&rof,(size_t)rofV.size()*size2);
cudaMalloc(&col,(size_t)colV.size()*size2);
cudaMalloc(&cof,(size_t)cofV.size()*size2);
cudaMalloc(&row,(size_t)rowV.size()*size2);
cudaMemcpyAsync(rof,&rofV[0],(size_t)rofV.size()*size2,cudaMemcpyHostToDevice,stream
  [0]);
cudaMemcpyAsync(col,&colV[0],(size_t)colV.size()*size2,cudaMemcpyHostToDevice,stream
  [1]);
cudaMemcpyAsync(cof,&cofV[0],(size_t)cofV.size()*size2,cudaMemcpyHostToDevice,stream
```

```
    [0]);
  cudaMemcpyAsync(row,&rowV[0],(size_t)rowV.size()*size2,cudaMemcpyHostToDevice,stream
    [1]);
```

After this remaining CSMR data is sent asynchronously, the device kernel is invoked with $2 * m^2 * sizeof(uint)$ of shared memory for each block, n^2/m^2 blocks and m^2 threads per block. After this, the cudamemory is freed.

```
  int stride = (int)(nm*nm);
  cudaDeviceSynchronize();
  matMulGPU<<<stride,m*m,2*size*m*m,0>>>(da,db,dc,ka,kb,rof,col,row,cof,m,n,mp1.size(),
    mp2.size()); // i X k
  cudaMemcpy((void *)c,(void *)dc,size3,cudaMemcpyDeviceToHost);
  // free the memory
  cudaFree(da);
  cudaFree(db);
  cudaFree(ka);
  cudaFree(kb);
  cudaFree(rof);
  cudaFree(col);
  cudaFree(dc);
  for(int i = 0;i<streamSize ;i++){
    cudaStreamDestroy(stream[i]);
  }
}
```

Final device kernel code:

```
__global__
void matMulGPU(uint *a, uint *b, uint *c, int *ka, int *kb, int *rof, int *col, int m,
   int n, int k1, int k2){
  extern __shared__ uint dab[];
  int bid = blockIdx.x;
  int tid = threadIdx.x;
  int nm = n/m;
  int i = bid / nm;
  int k = bid % nm;
  uint64_t temp = 0;
  for (int j = rof[i]; j < rof[i+1]; j++){
    int id1 = ka[j];
    int cl = col[j];
    int id2 = binASearch(kb,k2,cl,k);
    if(!(id2 == -1)){
      dab[tid] = (uint)a[tid + id1*m*m];
      dab[tid + m*m] = (uint)b[tid + id2*m*m];
      __syncthreads();
      int ii = tid/m;
      int jj = tid%m;
      for (int kk = 0; kk < m; ++kk)
      {
        temp = temp + (uint64_t)(dab[ii*m + kk] * dab[kk*m + jj + m*m]);
      }
      __syncthreads();
    }
    __syncthreads();
  }
  __syncthreads();
  c[tid + i*m*n + k*m*m] = min(temp,MAX_VAL);
}
```

Here simply, each block bid in n^2/m^2 blocks multiplies the corresponding blocks of A and B to make the resultant block. The m^2 threads individually calculate each block element within each block multiplication.

4 Notes and Other Optimizations

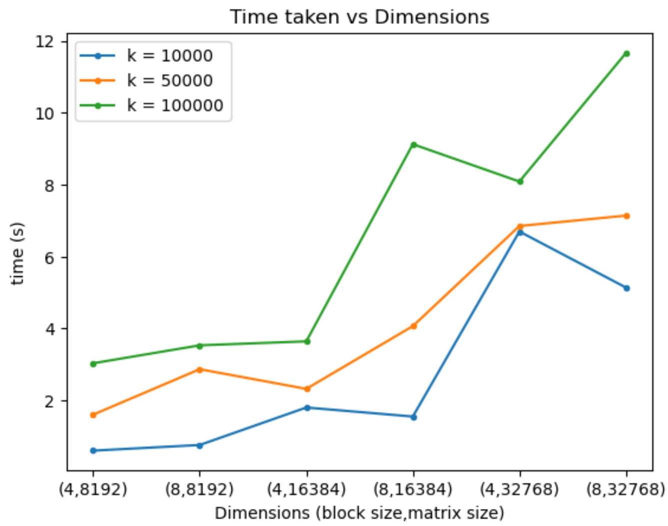
- Earlier stl maps were used to read and store the input but are now replaced with vectors of int3 and unit that store the index of the form block's row, column and array index and a vector that stores $k1 * m^2$ non zero block values.

- Experimented with grid dimensions and block sizes; however, that did not bring any significant speedup. So, the GPU kernel uses $2 * m^2 * sizeof(uint)$ shared memory for each block; it uses n^2/m^2 blocks and m^2 threads per block.

5 Analysis

5.1 Time for each test-cases

(Nodes,Threads)	(4,8192)	(8,8192)	(4,16384)	(8,16384)	(4,32768)	(8,32768)
Test1	0.605	0.762	1.804	1.553	6.697	5.139
Test2	1.606	2.869	2.322	4.067	6.850	7.139
Test3	3.034	3.533	3.643	9.125	8.086	11.659



5.2 Large Tests

```
Input reading done, n: 32768, m: 8, k-values: 8800000 and 8800000
Sorted input: 50.2979
==1170== NVPROF is profiling process 1170, command: ./exec input1 input2 outFile.bin
CSR converted
Time taken for matrix multiplication: 73.9428
Writing output
Number of output non-zero blocks: 16777216
Writing done
==1170== Profiling application: ./exec input1 input2 outFile.bin
==1170== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.38%  71.2514s         1  71.2514s  71.2514s  71.2514s  matMulGPU(unsigned int*, unsigned int*,
                                                unsigned int*, int*, int*, int*, int*, int*, int, int, int, int)
                    1.35%  986.72ms         8  123.34ms  3.4560us  479.70ms  [CUDA memcpy HtoD]
                    1.27%  926.46ms         1  926.46ms  926.46ms  926.46ms  [CUDA memcpy DtoH]
                    0.01%  7.3874ms         1  7.3874ms  7.3874ms  7.3874ms  [CUDA memset]
      API calls:   98.18%  72.1783s         1  72.1783s  72.1783s  72.1783s  cudaMemcpy
                    1.35%  995.15ms         8  124.39ms  20.320us  485.51ms  cudaMemcpyAsync
                    0.41%  298.98ms         2  149.49ms  3.3510us  298.97ms  cudaStreamCreate
                    0.04%  31.351ms         7  4.4788ms  7.3040us  10.615ms  cudaFree
                    0.01%  10.379ms         9  1.1533ms  4.2550us  4.3703ms  cudaMalloc
                    0.00%  381.67us         1  381.67us  381.67us  381.67us  cuDeviceTotalMem
                    0.00%  164.09us       101  1.6240us     137ns  69.008us  cuDeviceGetAttribute
                    0.00%  56.581us         2  28.290us  8.6290us  47.952us  cudaDeviceSynchronize
                    0.00%  42.429us         1  42.429us  42.429us  42.429us  cudaLaunchKernel
                    0.00%  38.825us         1  38.825us  38.825us  38.825us  cudaMemset
                    0.00%  29.269us         1  29.269us  29.269us  29.269us  cuDeviceGetName
                    0.00%  23.936us         2  11.968us  3.4970us  20.439us  cudaStreamDestroy
                    0.00%  9.3970us         1  9.3970us  9.3970us  9.3970us  cuDeviceGetPCIBusId
                    0.00%  1.6910us         3     563ns     274ns  1.1100us  cuDeviceGetCount
                    0.00%  1.3800us         2     690ns     226ns  1.1540us  cuDeviceGet
                    0.00%     275ns         1     275ns     275ns     275ns  cuDeviceGetUuid
```

```
real 2m50.428s
user 2m31.280s
sys 0m15.662s


Input reading done, n: 32768, m: 4, k-values: 1000000 and 1000000
Sorted input: 2.02364
==386== NVPROF is profiling process 386, command: ./exec input1 input2 outFile.bin
CSR converted
Time taken for matrix multiplication: 6.78835
Writing output
Number of output non-zero blocks: 56225712
Writing done
==386== Profiling application: ./exec input1 input2 outFile.bin
==386== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   84.08%  5.33634s         1  5.33634s  5.33634s  5.33634s  matMulGPU(unsigned int*, unsigned int*,
                                                  unsigned int*, int*, int*, int*, int*, int*, int*, int, int, int, int)
                   15.16%  961.98ms         1  961.98ms  961.98ms  961.98ms  [CUDA memcpy DtoH]
                    0.47%  29.659ms         8  3.7073ms  4.9920us  13.585ms  [CUDA memcpy HtoD]
                    0.29%  18.442ms         1  18.442ms  18.442ms  18.442ms  [CUDA memset]
      API calls:   95.86%  6.29870s         1  6.29870s  6.29870s  6.29870s  cudaMemcpy
                    3.15%  207.14ms         2  103.57ms  3.6860us  207.13ms  cudaStreamCreate
                    0.74%  48.943ms         8  6.1179ms  21.123us  31.733ms  cudaMemcpyAsync
                    0.12%  8.1196ms         7  1.1599ms  6.3680us  4.5655ms  cudaFree
                    0.10%  6.4531ms         9  717.01us  6.8990us  4.9123ms  cudaMalloc
                    0.01%  711.94us         2  355.97us  351.97us  359.98us  cuDeviceTotalMem
                    0.01%  601.26us       202  2.9760us     129ns  133.62us  cuDeviceGetAttribute
                    0.00%  83.458us         2  41.729us  9.1930us  74.265us  cudaDeviceSynchronize
                    0.00%  72.175us         2  36.087us  24.986us  47.189us  cuDeviceGetName
                    0.00%  30.094us         1  30.094us  30.094us  30.094us  cudaMemset
                    0.00%  28.433us         1  28.433us  28.433us  28.433us  cudaLaunchKernel
                    0.00%  15.683us         2  7.8410us  3.3400us  12.343us  cudaStreamDestroy
                    0.00%  8.1140us         2  4.0570us  1.3880us  6.7260us  cuDeviceGetPCIBusId
                    0.00%  2.2730us         4     568ns     165ns  1.4900us  cuDeviceGet
                    0.00%  2.1840us         3     728ns     301ns  1.5370us  cuDeviceGetCount
                    0.00%     493ns         2     246ns     229ns     264ns  cuDeviceGetUuid

real 0m49.730s
user 0m41.582s
sys 0m7.030s

Input reading done, n: 32768, m: 4, k-values: 10000000 and 10000000
Sorted input: 21.1806
==222== NVPROF is profiling process 222, command: ./exec input1 input2 outFile.bin
CSR converted
Time taken for matrix multiplication: 66.5818
Writing output
Number of output non-zero blocks: 67108864
Writing done
==222== Profiling application: ./exec input1 input2 outFile.bin
==222== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.10%  64.4583s         1  64.4583s  64.4583s  64.4583s  matMulGPU(unsigned int*, unsigned int*,
                                                  unsigned int*, int*, int*, int*, int*, int*, int*, int, int, int, int)
                    1.40%  921.59ms         1  921.59ms  921.59ms  921.59ms  [CUDA memcpy DtoH]
                    0.47%  307.30ms         8  38.413ms  5.7910us  138.32ms  [CUDA memcpy HtoD]
                    0.03%  18.419ms         1  18.419ms  18.419ms  18.419ms  [CUDA memset]
      API calls:   98.89%  65.3804s         1  65.3804s  65.3804s  65.3804s  cudaMemcpy
                    0.58%  386.23ms         2  193.11ms  6.6290us  386.22ms  cudaStreamCreate
                    0.49%  326.70ms         8  40.837ms  20.126us  156.72ms  cudaMemcpyAsync
                    0.02%  13.080ms         7  1.8686ms  25.425us  6.7470ms  cudaFree
                    0.01%  7.6059ms         9  845.10us  5.7630us  4.4642ms  cudaMalloc
                    0.00%  740.06us         2  370.03us  1.3170us  738.74us  cuDeviceGetPCIBusId
                    0.00%  723.05us         2  361.52us  344.31us  378.74us  cuDeviceTotalMem
                    0.00%  550.85us       202  2.7260us     126ns  123.41us  cuDeviceGetAttribute
                    0.00%  67.440us         2  33.720us  8.1370us  59.303us  cudaDeviceSynchronize
                    0.00%  59.749us         2  29.874us  23.872us  35.877us  cuDeviceGetName
                    0.00%  41.699us         1  41.699us  41.699us  41.699us  cudaLaunchKernel
                    0.00%  33.383us         1  33.383us  33.383us  33.383us  cudaMemset
                    0.00%  22.023us         2  11.011us  4.0410us  17.982us  cudaStreamDestroy
                    0.00%  1.8620us         4     465ns     135ns  1.0720us  cuDeviceGet
                    0.00%  1.1170us         3     372ns     182ns     557ns  cuDeviceGetCount
                    0.00%     496ns         2     248ns     223ns     273ns  cuDeviceGetUuid

real 2m20.456s
user 2m7.119s
sys 0m11.027s
```

```
Input reading done, n: 32768, m: 4, k-values: 33000000 and 33000000
Sorted input: 63.7081
==286== NVPROF is profiling process 286, command: ./exec input1 input2 outFile.bin
CSR converted
Time taken for matrix multiplication: 310.647
Writing output
Number of output non-zero blocks: 67108864
Writing done
==286== Profiling application: ./exec input1 input2 outFile.bin
==286== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.36%  306.862s         1  306.862s  306.862s  306.862s  matMulGPU(unsigned int*, unsigned int*,
                                                 unsigned int*, int*, int*, int*, int*, int*, int*, int, int, int, int)
                    0.33%  1.02598s         8  128.25ms  10.687us  466.49ms  [CUDA memcpy HtoD]
                    0.30%  931.87ms         1  931.87ms  931.87ms  931.87ms  [CUDA memcpy DtoH]
                    0.01%  18.447ms         1  18.447ms  18.447ms  18.447ms  [CUDA memset]
      API calls:   99.53%  307.794s         1  307.794s  307.794s  307.794s  cudaMemcpy
                    0.34%  1.04535s         8  130.67ms  24.050us  484.88ms  cudaMemcpyAsync
                    0.13%  392.57ms         2  196.28ms  3.5380us  392.57ms  cudaStreamCreate
                    0.01%  16.332ms         7  2.3331ms  6.3520us  6.3657ms  cudaFree
                    0.00%  11.147ms         9  1.2385ms  7.4930us  4.4978ms  cudaMalloc
                    0.00%  1.0613ms         2  530.63us  1.6300us  1.0596ms  cuDeviceGetPCIBusId
                    0.00%  814.36us         1  814.36us  814.36us  814.36us  cudaMemset
                    0.00%  719.74us         2  359.87us  348.62us  371.12us  cuDeviceTotalMem
                    0.00%  581.48us       202  2.8780us     128ns  130.50us  cuDeviceGetAttribute
                    0.00%  86.239us         2  43.119us  11.294us  74.945us  cudaDeviceSynchronize
                    0.00%  62.307us         2  31.153us  24.757us  37.550us  cuDeviceGetName
                    0.00%  37.457us         2  18.728us  4.0820us  33.375us  cudaStreamDestroy
                    0.00%  33.162us         1  33.162us  33.162us  33.162us  cudaLaunchKernel
                    0.00%  2.1860us         4     546ns     150ns  1.2830us  cuDeviceGet
                    0.00%  1.0410us         3     347ns     172ns     565ns  cuDeviceGetCount
                    0.00%     498ns         2     249ns     207ns     291ns  cuDeviceGetUuid

real 7m7.282s
user 6m46.856s
sys 0m15.976s
```

5.3 Observation

Based on the above large outputs we can say that there is a massive speedup in GPU parallelism. The edge cases take 2min50secs(74 secs) and 7min7secs(310 secs) for m= 8 and 4 respectively. Also, the sparse matrix representation greatly improves the algorithm. Also based on different n, m, and k values as in the graph, the scaling is almost proportional to the problem size. Though m as 4 or 8 does not have major effects when n is small.