High Performance Sparse Matrix Vector Multiply

Project Report
APMA E4302: Methods in Computational Science

Yan Cheng (UNI: yc3855)

December 27, 2020

1 Introduction

During this semester, we have looked at some fundamental yet very illuminating concepts and techniques in modern scientific computation. These include performance analysis of algorithms, shared and distributed parallelism, and etc. In this project, I want to use what we learned to study a concrete problem, which is the sparse matrix-vector multiplication.

A sparse matrix is a matrix with zero in most of its entries. It arises from many problems in science and engineering such as computational fluid dynamics, structural analysis, and so on. As an example, in class, we have seen that the finite difference discretization for a boundary value problem yields a tridiagonal matrix which is sparse. In fact, the resulting tridiagonal matrices has a special “band structure" that one can take advantage of. However, in general, we will not have the luxury of picking what kind of sparse matrix we have. Therefore, in this projct, we want to consider the general sparse matrix, and find out what we can say about it.

A common approach for solving linear systems would be the iterative methods such as the Jacobi method we used in class. A kernel of many iterative methods is the sparse matrix-vector multiplication. The amount of time for iterations is heavily dependent on how we fast the multiplication is, so one natural wants to optimize the multiplication procedure. Meanwhile, since in many practical situations, one would be dealing with large linear systems, it would be strategic to try to take advantage of the sparsity to optimize the matrix-vector multiplication.

In this project, we want to explore some known techniques for optimizing the sparse matrix-vector multiplication. We will start with the basic problem of how to store a sparse matrix efficiently in the memory, and introduce some common data structures for this purpose; the special feature of sparse matrices makes it suitable for us to consider different storage formats. For each format, we will discuss how it affects the matrix-vector multiplication. These are the general techniques that one can use; one can also say about how hardware features can be exploited to optimize the performance. Assuming only a single processor is at present, we will both vectorize the code, and try to formulate a scheme to run it on GPU. After that, we will consider parallelizing the multiplication algorithm. This part is similar to our discussion on distributed memory parallelism from class. We will be able to analyze the performance of the parallel algorithm by recalling the idea of arithmetic intensity and scalability. Lastly, we will see how one can combine vectorization and parallelism to create a parellel fix scheme for sparsee matrix-vector multiplication.

2 General Matrix Vector Multiplication

For an $m \times n$ matrix with floating point entries, the conventional storage will take $O(mn)$ amount of space in the memory. However, with some clever manipulation, we can take advantage of the sparsity of the matrix to reduce the waste of memory space. In this section, we will introduce three common storage formats for sparse matrices. For each format, we will also present how the matrix-vector multiplication is done.

Compressed Sparse Row

The most common storage scheme for sparse matrices is the *compressed sparse row* (CSR) or the *compressed row storage* (CRS). It stores the nonzero entries of a sparse matrix in terms of three one-dimensional arrays, thereby reducing the amount of necessary sotrage in memory. Specifically, the information encoded in the matrix is extracted by the representation ($\texttt{value}, \texttt{col\_ind}, \texttt{row\_ptr}$). The array `value` stores nonzero floating point numbers, and `col_ind` is the column index of each element in `value`. The size of these two arrays will both be the number of nonzero entries `nnz`. The third array is `row_ptr`, and it is the range of pointers to where the new rows start. Therefore, ($\texttt{col\_ind}, \texttt{row\_ptr}$) helps us recover which entry each nonzero value is at. For instance, consider

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 0 \end{pmatrix}.$$

If we use the zero-based index set[1], then the nonzero entries are $M_{0,1} = 1, M_{1,2} = 2, M_{1,3} = 3, M_{2,2} = 4$ and $M_{3,0} = 5$. The CSR format of M is given by

$$\begin{array}{rcccccc} \texttt{value} & = & 1 & 2 & 3 & 4 & 5 \\ \texttt{col\_ind} & = & 1 & 2 & 3 & 2 & 0. \\ \texttt{row\_ptr} & = & 0 & 1 & 3 & 4 & 5 \end{array}$$

In this case, `row_ptr` has the same size with the other two, but in general, it needs not have. Since $M_{0,1} = 1$ and $M_{1,2} = 2$ are both the first nonzero element of their rows but $M_{2,4} = 3$ is not, no pointer corresponding to 3 is there. By convention, we add an extra last element to `row_ptr`, and this is always equal to `nnz`.

It is clear that we can assign each sparse matrix with a CSR representation, but one may be curious if this mapping is invertible, *i.e.* we know how to go back from a CSR format to the original matrix. Upon contemplating, one may notice that we cannot have all-zero rows. A solution provided by Intel MKL [2] is to repeat the last row pointer if there is an all-zero rows. But for simplicity, we will assume that we are dealing with sparse matrices without all-zero rows.

Now let us see how the sparse matrix-vector multiplication is done in the CSR format. Without loss of generality, suppose we have converted an $n \times n$ matrix into the CRS format, which we denote

[1]Sometimes people start the index with 1 but this is probably not very important.

by A. Then the following pseudocode is an implementation of the matrix vector product $y = Ax$.

Algorithm 1: Sparse matrix-vector product in CSR format

```
Data: Sparse matrix $A$ in the CSR format and a vector $x$
Result: Compute $y = Ax$ when $A$ is in CSR format.
for (row = 0; row < n; row++) do
    s = 0;
    for (icol = row_ptr[ row ]; icol < row_ptr[ row + 1 ]; icol++) do
        col = ind[icol];
        $s+ = A[icol] * x[col]$;
    end
    y[row] = s;
end
```

One can notice that the inner loop is not using the column number as index; instead, it uses the location (pointer) in the memory. This is known as the *indirect addressing*. It first stores the data in an inter array, so it requires an additional data access. However, one advantage is that by addressing the location instead of the actual data, the same set of instructions can be used multiple times.

Our discussion on the roofline model in class suggests that the performance of an algorithm can be constrained by the bandwidth and data reuse. At a glance, this sparse matrix-vector product has the two problems:

- At the indirect addressing step, we have to load the elements of an integer vector from the memory. So more memory traffic is anticipated.
- The elements of A are not necessarily loaded sequentially from the memory; in fact, they can be loaded in completely random order. This means the cacheline may contain elements that is not utilized in computation.

However, with a single processor, these are not the dominant issues. A major problem is that we cannot fully take advantage of the vectorization when the number of nonzero entries for each row is different.

Ellpack/Itpack

The Ellpack/Itpack (*ELL*) forces all rows to have the same length as the number of nonzero elements of the row with the largest number of nonzero elements. This means that if a row has less number of nonzero elements, we pad more zeros to fill the length. Evidently, the zeros do not contribute to the multiplication, but this is useful in exploting vectorization (more precisely, to repeat the instructions). The multiplication algorithm is as follows.

Algorithm 2: Sparse matrix-vector product in ELLPACK format

```
Data: Sparse matrix $A$ in the ELL format and a vector $x$
Result: Compute $y = Ax$ when $A$ is in ELL format.
y(1:n)= 0.0;
for (j = 0; j < NZ; j++) do
    y(1:n) = y(1:n) + A(1:n, j) * x(col_ind (1:n, j)) ;
end
```

This method yields good results when the number of nonzero elements in each row are very close. However, in the case when the largest number of nonzero elements in a row is much larger than the average number, storing the extra zeros presents a serious issue.

Jagged Diagonal Format

The Jagged Diagonal (*JAD*) format also aims at exploiting the vectorizability. It first sorts the rows into rows of increasing order in terms of the number of nonzero elements. Then it stores the leftmost element in each row to a vector, then the second leftmost elements to a second vector (possibly shorter than the first), and so on. The matrix-vector multiplication is done on this columns instead. Therefore, the vectorization takes place down along the rows. The pseudocode for JAD is given as follows.

Algorithm 3: Sparse matrix-vector product in jagged diagonal format

```
Data: Sparse matrix $A$ in the JAD format and a vector $x$
Result: Compute $y = Ax$ when $A$ is in JAD format.
y(1:n)= 0.0;
IP = 1;
for (j = 0; j < NZ; j++) do
    i = ind[j];
    m = n - i + 1;
    y(1:n) = y(1:n) + A(IP:(IP + m - 1)) * x(IP:(IP + m - 1)) ;
    IP += m;
end
```

The JAD format is very efficient on vector machines. However, it requires data rearangement which takes additional efforts which presents additionall difficulty. Plus, in general, it is more preferable if we can use CSR instead. It would bee troublesome if we have to convert CSR to JAD, particularly when we are to solve a large linear system.

3 Hardware-dependent Sparse Matrix-Vector Multiplication

With only a single processor, fetching data from the memory would not typically cause significant delay. Therefore, one can hope to optimize the algorithm by some other strategies, such as vectorization or running the algorithm on a GPU. For vectorization, the problem is that, since the the number of nonzero elements in a sparse matrix row is low, the vector length is low. For running on GPU, the problem is that without the zero elements, the rows may be of different length, so we cannot hope to find a large number of identical instructions. In this section, we want to discuss what can we do differently so that we can still take the full advantage of vectorization and the power of GPU.

Vectorized Sparse Matrix-Vector Multiply for CSR Format

In this section, we want to cover the research article by D'Azevedo, Fahey and Mills [3]. Suppose we have a vector machine (the Cray series, for instance). To achieve better preformance, we would prefer long regular vector operations over scalar ones.

As mentioned in the previous section, on a vector machine, the potential for vectorization is limited by the number of of nonzero elements in the sparse matrix. A popular solution, as in both Ellpack and JAD, is to first sort the rows by the number of nonzero elements, then vectorizes the code. However, if a matrix is already in CSR format, we would need memory space to temporarily store and convert the JAD format into CSR. Therefore, it would be preferable if we can vectorize the CSR directly.

The idea of the algorithm is to apply a permutation so that the rows with the same number of nozeros are groups together. This is similar to the JAD format, but the difference is that there is no data rearrangement, and are accessed indirectly by the permutation vector. Consider the following algorithm.

Algorithm 4: Vectorized sparse matrix-vector product

```
Data: Sparse matrix $A$ in the CSR format and a vector $x$
Result: Compute $y = Ax$ when $A$ is in the CRS format.
for (group = 0; group < NumGroup; group++) do
    Start = xGroup (group);
    End = xGroup (group + 1) - 1;
    NZ = NZGroup(group);
    /* rows(Perm(Start:End)) all have same NZ nonzero elements per row      */
    for (row = Start; row < End; row+=NB) do
        rowEnd = min(End, row + NB - 1);
        m = rowEnd - row + 1;
        Ind_Perm (1:m) = row_ptr (Perm(row:rowEnd));
        y_perm(1:m) = 0.0;
        /* Consider y_perm(:)  and Ind_Perm(:)  as vector registers         */
        for col = 0; jcol< NZ; j++ do
            y_perm(1:m) = y_perm(1:m) +A(Ind_Perm(1,m)) * x(col_ind(1:m));
            Ind_Perm(1:m) = Ind_Perm(1:m) + 1;
        end
    end
    y(Perm(row:rowEnd)) = y_perm(1:m);
end
```

The `IPerm` is the premutation vector. It can be constructed by sorting algorithms when `row_ptr` is known. `xGroup` points ot the beginning indices of groups of `IPerm`. As the line Ind_Perm (1:m) = row_ptr (Perm(row:rowEnd)) suggests, the memory location is only acceseed indirectly. A consequence is that then the acceesses of A and `col_ind` look “irregular" (this is directtly pointed out by the authors).

Sparse Matrix-Vector Multiply on GPU

In another paper [4], a similar is used in GPU computing. The authors are interested in implementing the sparse matrix conjugate gradient solver on a GPU (the authors used NVIDIA's GeForce FX). In this paper, implementing a fundamental algorithm on GPU is considred as a mapping from classical algorithms to a new computing paradigm. This new paradigm is viewed as an instance of stream processing: “the processor executes the same kernel to produce each element of an output stream," which then “is saved and used as input for downstream krenels" ([4] Page.918). It follows from this interpretation that we can “map" the traditional sparse matrix-vector to this computing paradigm by finding a suitable sparse matrix data structure and an associated fragment program to execute the multiplication.

In the terms of GPU computing, we “render" groups of rows with an equal amount of nonzero entries, and for each group we associate a “fragment program". This fragment program accesses the nonzero entries, and finds the corresponding entry in the unknown array x. The x values are stored in the texture memory, which we denote as $\mathcal{X}^x$.

Now we describe the data structure for sotring a sparse matrix A. It is stored in two textures. For the first one, we have a texture $\mathcal{A}_i^x$ that stores the diagonal entries A_{ii} with tthe same laid out as $\mathcal{X}^x$. The off-diagonal, nonzero entries are stored in another texture $\mathcal{A}_j^a$ in a consecutive order as segments (like in the CSR format). The reason for separating $\mathcal{A}_i^x$ and $\mathcal{A}_j^a$ is to keep an edge for when we have diagonal preconditioning.

Each address of a new row (texture address of each segment's starting entry) is stored (as pointers) in an indirection texture $\mathcal{R}^x$. Also, let $\mathcal{C}^a$ be the texture keeping the correspondence beween $\mathcal{A}^a$ and $\mathcal{X}^x$: each a_{ij} in $\mathcal{A}_j^a$ matches x_j in $\mathcal{X}^x$. Then, the inner product between a row of A and x can be formulated as two lines:

$$j = \mathcal{R}^x[i]$$

$$\mathcal{Y}^x[i] = \mathcal{A}_i^x[i] * \mathcal{X}^x[i] + \sum_{c=0}^{k_i-1} \mathcal{A}_j^a[j+c] * \mathcal{X}^x[\mathcal{C}^a[j+c]]$$

where from our previous notation, it is clear that $\mathcal{Y}^x$ is the texture for y. The authors showed that on a GeForce FX processor, rows with up to 200 nonzero entries can be processed in a single clock cycle.

4 Parallel Sparse Matrix-Vector Product

In this section, we will discuss how to optimize the sparse matrix-vector multiplication by parallelism. First, assume a sparse matrix A is partitioned into blocks of rows (like strips). Each processor p then owns the number of rows indexed by some index set I_p (so this is actually the index for the columns).

We will first handle the communication between processes. For each y_i, conventionally, we will need the i-th row of A and all of x. However, the multiplication only needs those elements that are nonzero. So for every $i \in I_p$, define

$$S_{p,i} = \{j : j \notin I_p, A_{ij} \neq 0\}.$$

Then $y = Ax$ is given by

$$y_i += A_{ij}x_j, \quad j \in S_{p,i}.$$

This means for each processor p, to find y_i, we will need to communicate the off-processor values x_j where $x_j \in S_{p,i}$. By sparsity, the space for each local buffer (dependent on $i \in I_p$) would not cost so much as in the dense matrix case. Therefore, to reduce cost for communication, we combine them into a single message per processor. This can be accomplished by defining

$$S_p = \bigcup_{i \in I_p} S_{p,i}.$$

Then we perform the matrix-vector by collecting all x_j with $j \in S_p$ into one buffer.

In the case of dense matrix-vector multiplication, partitioning by blocks of rows does not lead to a scalable algorithm. This is partly due to the communication each processor needs. For sparse matrices, the situation is different. Suppose we have P processors, and we want to solve a 2D boundary value problem on a square. If we partition the domain to strips, then each processor has to communicate with at most 2 of its neighbors. Then let w be th amount of time each processor takes to complete the operations, and let c be the time for communication with one neighbors. Then with a single proccessor, the amount of time for the multiplication is $T_1 = Pw$. With P processors, the time is about $T_P = w + 2c$. So the speed up is

$$S_P = \frac{T_1}{T_P} = \frac{Pw}{w+2c} = P\frac{w}{w+2c}.$$

Therefore, this parallel algorithm is weakly scalable. This is saying that when P is large, the parallelism displays an apparent advantage.

If we can partition the domain into a $\sqrt{P} \times \sqrt{P}$ grid instead, then each processor communicates with at most 4 neighbors. With P processors, the time is about $T_P = w + 4c$. The speed up is, hence,

$$S_P = \frac{T_1}{T_P} = P\frac{w}{w+4c}.$$

This parallel scheme is also weakly scalable.

One might be interested to increase data reuse in our scheme to increase the arithtmetic intensity. However, the same problem in the single processor case also appears here: each row may have different numbers of nonzero elements. Getting around it is a tricky problem so we will not discuss it here.

5 Segmented Scans for Sparse Matrix-Vector Multiply

There is also a way to utilize both vectorization and parallel processors at the same time. In [5], the authors introduced a *segmented sum* that can be vectorized within each processor and parallelized across processors. It is based on a method known as the *segmented scan operation*. Here the *scan operation*, also called the *parallel prefix* by some other authors, means the following. Given an associative binary operator $\oplus$ and an array $[a_1, \ldots, a_n]$, define

$$x_i = \begin{cases} a_1, & i = 1, \\ x_{i-1} \oplus a_i, & 1 < i \leq n. \end{cases}$$

The key is that partial reductions, *i.e.* $(x_1 \oplus x_1) \oplus (x_3 \oplus x_4) \oplus \cdots$ can be computed in parallel. Then an array of size n can be reduced in $\lceil \log_2 n \rceil$ steps.

The segmented scan operations takes an additional array that specifies how this array is partitioned into segments. One way to do this is use flags; for instance, suppose we are to sum a group of integers $[5, 1, 3, 4, 3, 9]$. Then the segmented scan is looks like

$$\begin{array}{lcrrrrrr} \texttt{VALUE} & = & 5 & 1 & 3 & 4 & 3 & 9 \\ \texttt{FLAG} & = & T & F & T & F & F & F. \\ \texttt{SUM} & = & 5 & 6 & 3 & 7 & 10 & 19 \end{array}$$

In other words, the flag $T = True$ indicates the start a new summation. As is commented in [1], the flags act as the indicator

$$s_{ij} = \begin{cases} 1, & \text{if } j \text{ is the first nonzero index in row } i \\ 0, & \text{otherwise.} \end{cases}$$

Without it, the parallel prefix sum can only produce y_1, $y_1 + y_2$, and so on, which is not what we want.

One would notice that this similar in some sense to the CSR format with the flags used in the same fashion as the row pointers. Indeed, this is a motivation to represent sparse matrix as a segmented vector where each row as a segment. Using the row pointer array in CSR format, we can generate a flag array: first we set all flags to `False`, then for each element in the pointer array, we reset it to `True`. Then the sparse matrix-vector multiplication can be written as follows

$$\begin{aligned} \texttt{PRODUCTS} &= \texttt{VALUE} * \texttt{X(Col\_Ind)} \\ \texttt{SUM} &= \texttt{SUM\_SUFFIX}(\texttt{PRODUCTS}, \texttt{SEGMENT=PARITY\_PREFIX(FLAG)}) \\ \texttt{Y} &= \texttt{SUM}(\texttt{PTR}) \end{aligned}$$

The first line calculates the coordinate-wise product by indirectly addressing the `X(Col_Ind)`; each entry of `X(Col_Ind)` is `X(Col_Ind(k))`. The second line performs the suffix sum [2] on the array `PRODUCTS`, while the second argument tells the function how to break the array into segments. For instance, we have

$$\begin{array}{rcllllll} \texttt{PRODUCTS} & = & 5 & 1 & 3 & 4 & 3 & 9 \\ \texttt{FLAG} & = & T & F & T & F & F & F. \\ \texttt{SUM\_SUFFIX}(\texttt{PRODUCTS}, \texttt{SEGMENT=PARITY\_PREFIX(FLAG)}) & = & \mathbf{6} & 1 & \mathbf{19} & 16 & 12 & 9 \end{array}$$

The advantage of this setup is evident when we are working on a parallel machines with vector multi-processors. Although the actual implementatiton is slower than the JAD method, the segmented scan method allows us to keep the sparse matrix as CSR format, and we do not have to permute it. What we want do is to modify it so that we can both keep these advantages and achieve better performance.

The authors of [5] noted that the reason why this scan-based method than JAD is slower in practice is because the former generates all the partial sums in each segment, but only the final sum is needed. Both the parallel and the vectorized scan algorithms then have to communicate more data than are required. To get around with this problem, the authors introduced two new functions `SETUP_SEG(PNTR,M,N)` and `SEG_SUM(VAL,PNTR,SEG)`. The former, `SETUP_SEG(PNTR,M,N)`, takes in a pointer array of length M, and a total number of nonzero elements N, and returns an array of integers which is a “segment descriptor". So when we call

$$\texttt{SEGDES} = \texttt{SEG\_SUM(PNTR,M,N)},$$

the array `SEGDES` describes how the segmentation is given. The other function to be defined, `SEG_SUM(VAL,PNTR,SEG)`, sums each segment of the array `VAL`, returns an array of length M, and stores one sum per segment into contiguous locations. To implement the sparse matrix-vector multiplication, one then runs

$$\begin{aligned} \texttt{SEGDES} &= \texttt{SETUP\_SEG(PNTR,M,N)} \\ \texttt{PRODUCTS} &= \texttt{VAL} * \texttt{X(INDX)} \\ \texttt{Y} &= \texttt{SEG\_SUM(PRODUCTS,PNTR,SEGDES)} \end{aligned}$$

The two aforementioned functions allow us to make less data communication. However, the implementation is very involved. Being quite complicated is just one drawback; as is pointed out in [3], it requires assembly code level implementation to achieve optimal efficiency. That said, using the parallel prefix sum is really a fundamentally different approach from what we have seen before.

[2]The SUM_SUFFIX function is available from High Performance Fortran (HPF).

Conclusion

I was originally interested reading the part of [1] on sparse matrix-vector multiplications. It then lead me to [3],[4] and [5]. In conclusion, I think this project presents a good opportunity to apply what we have seen this semester in class. There are multiple perspectives to assess the performance of algorithms, and as is emphasized by this project.

References

[1] Victor Eijkhout. *Introduction to high performance scientific computing.* The online version which is updated regularly.

[2] Intel Math Kernel Library. *Sparse Matrix Storage Formats.* `https://www.smcm.iqfr.csic.es/docs/intel/mkl/mkl_manual/appendices/mkl_appA_SMSF.htm`

[3] Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. *Vectorized sparse matrix multiply for compressed row storage format.* Lecture Notes in Computer Science, Computational Science - ICCS 2005, pages 99–106, 2005.

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

[5] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. *Segmented operations for sparse matrix computation on vector multiprocessors.* Technical Report CMU-CS-93-173, CMU, 1993.