Sparse Matrix-Vector Multiplication on a GPU

*Design and Analysis of Parallel Algorithms*

Due: 9:00, May 24, 2011 (presentation + code)
1 Introduction

The matrix–vector multiplication operation (GEMV in level 2 BLAS)

\[ y \leftarrow Ax + y, \]

where \( A \in \mathbb{R}^{n \times n} \) is a square sparse\(^1\) matrix and \( x, y \in \mathbb{R}^n \) are \( n \)-vectors is a fundamental and often performance-critical computational kernel in many applications. A natural question is whether or not the high bandwidth and high peak performance of a GPU can be utilized to speed up the computation. It turns out that the answer to this question depends on the sparsity pattern of the matrix. To exploit the sparsity, one first needs a compressed storage format. If no assumptions are made on the sparsity pattern of \( A \), then a general storage format is required. However, if the sparsity pattern of \( A \) has some special structure, then it can often be stored and operated on more efficiently if we use a special-purpose storage format. In this assignment, you will implement a sparse matrix–vector multiplication kernel on a GPU. You will use both a general-purpose and a special-purpose storage format and compare their effectiveness on a particular class of sparsity patterns. In the following, we let \( \text{nnz}(A) \) denote the number of non-zero elements in \( A \).

2 The Compressed Sparse Row (CSR) storage format

The Compressed Sparse Row (CSR) storage format is a general-purpose format that stores the non-zeros of each row contiguously. It is especially suited for a parallel block row decomposition of the matrix–vector product. The CSR format consists of three arrays:

- \( \text{val} \): A floating point array of length \( \text{nnz}(A) \) that stores the non-zero elements of \( A \) in row-major order.
- \( \text{rowptr} \): An integer array of length \( n + 1 \) that stores the index in \( \text{val} \) of the first non-zero element of each row. By convention, the last element of the array holds \( \text{nnz}(A) + 1 \).
- \( \text{colind} \): An integer array of length \( \text{nnz}(A) \) that stores the column index of the corresponding element of \( \text{val} \).

Performing a matrix–vector multiplication with \( A \) stored in CSR is fairly straight-forward. A pseudo-code algorithm is given below.

\[
\text{CSRMatVec}(n, \text{val}, \text{rowptr}, \text{colind}, x, y) \\
\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
\quad \text{for } j \text{ from } \text{rowptr}[i] \text{ to } \text{rowptr}[i+1]-1 \text{ do} \\
\quad\quad y[i] = y[i] + \text{val}[j] \times x[\text{colind}[j]]
\]

Of particular interest when analyzing the performance of this algorithm is the memory access pattern. The array \( y \) is accessed with unit stride, and it is accessed only once. The same applies to the arrays \( \text{val} \) and \( \text{colind} \), although that this is the case is slightly less obvious than for the \( y \) array. On the other hand, there are \( \text{nnz}(A) \) element accesses to the array \( x \), and the indices, and hence the memory access pattern, is entirely dependent upon the sparsity pattern of \( A \) via the array \( \text{colind} \). Therefore, we can construct sparse matrices for which CSR leads to a very good memory access pattern, and we can construct matrices for which the opposite is true.

\(^1\)Most elements in a sparse matrix are zero.
3 The Diagonal storage format

The **main diagonal** of an \( n \times n \) matrix \( A \) consists of the elements \( A(i, i) \) for \( i = 1, 2, \ldots, n \). Besides the main diagonal, there are also \( n - 1 \) sub-diagonals (below the main diagonal) and \( n - 1 \) super-diagonals (above the main diagonal). In general, diagonal \( d \in \{-n+1, \ldots, n-1\} \) consists of the elements on the diagonal that is \( d \) steps above the main diagonal. In particular, the main diagonal corresponds to \( d = 0 \), the sub-diagonals to \( d < 0 \), and the super-diagonals to \( d > 0 \).

The Diagonal storage format is well suited for matrices in which the non-zeros form (more or less) dense diagonals. Let \( k \) denote the number of diagonals that contain at least one non-zero element. The Diagonal storage format consists of a 2D floating point array \( \text{values} \) of size \( n \times k \) and an integer array \( \text{distance} \) of length \( k \). Each diagonal is stored contiguously as a “column” in the \( \text{values} \) array. Using the Diagonal storage format, matrix–vector multiplication can be arranged as a sum in which each term is essentially a diagonal matrix times a vector, which is an operation that can be very efficiently implemented. The second component of the Diagonal storage format is the \( \text{distance} \) array, which stores the distance from the main diagonal, i.e., \( d \), of the corresponding “column” of the \( \text{values} \) array.

As an example of the Diagonal storage format, consider the matrix

\[
A = \begin{bmatrix}
a_1 & 0 & b_1 & 0 & 0 \\
c_1 & a_2 & 0 & b_2 & 0 \\
0 & c_2 & a_3 & 0 & b_3 \\
0 & 0 & c_3 & a_4 & 0 \\
0 & 0 & 0 & c_4 & a_5 \\
\end{bmatrix}
\]

and its representation in the Diagonal storage format

\[
\text{values} = \begin{bmatrix}
\star & a_1 & b_1 \\
c_1 & a_2 & b_2 \\
c_2 & a_3 & b_3 \\
c_3 & a_4 & \star \\
c_4 & a_5 & \star \\
\end{bmatrix}, \quad \text{distance} = [-1 \ 0 \ 2],
\]

where a star, \( \star \), denotes padding. This padding is such that the rows of \( A \) correspond to the “rows” of the \( \text{values} \) array.

A matrix–vector multiplication algorithm for \( A \) stored in the Diagonal storage format can be expressed in pseudo-code as follows:

\[
\text{DiagonalMatVec}(n, k, \text{values}, \text{distance}, x, y)
\]

\begin{algorithmic}
\FOR {i from 1 to k}
\STATE d = \text{distance}[i]
\STATE r0 = \max(1, 1-d)
\STATE r1 = \min(n, n-d)
\STATE c0 = \max(1, 1+d)
\FOR {r from r0 to r1}
\STATE c = r - r0 + c0
\STATE y[r] = y[r] + \text{values}[r,i] * x[c]
\ENDFOR
\ENDFOR
\end{algorithmic}

The memory access pattern of this algorithm is quite good. Consider the inner loop. The arrays \( y \), \( \text{values} \), and \( x \) are all accessed with unit stride, which is optimal. The location of the
first entry depends on the elements of the distance array, which should therefore be sorted in increasing (or decreasing) order to enable reuse of portions of y and x.

4 Instructions

Your task is to design and analyze two sparse matrix–vector multiplication kernels on a GPU: One kernel using the CSR format and one kernel using the Diagonal storage format. You should use the float (single precision) data type for your floating point data.

One of the most straightforward (but probably not the most efficient) ways of parallelizing the kernels is to assign one GPU thread to each row of the matrix A. Another possibility is to assign an entire warp instead of just one thread to each row of A.

For testing purposes, you might want to consider the following classes of matrices:

- Let the size of A be $n \times n$. Set the elements of the diagonals $d \in \{-1, 0, 1\}$ to random numbers within the range $(-1, 1)$.

- Let the size of A be $n^2 \times n^2$. Set the elements of the diagonals $d \in \{-n, -1, 0, 1, n\}$ to random numbers within the range $(-1, 1)$.

- Let the size of A be $n^3 \times n^3$. Set the elements of the diagonals $d \in \{-n^2, -n, -1, 0, 1, n, n^2\}$ to random numbers within the range $(-1, 1)$.

- Let the size of A be $n \times n$. Set the elements of the main diagonal and the first $k \geq 0$ sub- and super-diagonals to random numbers within the range $(-1, 1)$.

Let the order, $n$, of the matrix range from a few thousand up to a point where the data stored on the GPU requires a large fraction of the GPU memory, e.g., hundreds of MB.

Instead of writing a report, you should prepare a 15 min presentation to be delivered on May 24. Begin by describing your two kernel implementations. Continue by presenting the results of your performance evaluation. End by discussing possible improvements.