Sparse Matrix Multiplication with Two Algorithms on GPU and Comparing with CPU Implementation

Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in scientific computing and is used in a variety of applications such as graph analytics, scientific simulations, and machine learning. It involves multiplying a sparse matrix with a dense vector, resulting in a sparse output vector. Due to the sparsity of the matrix, most of the matrix elements are zero, and thus, SpMV is computationally expensive, and parallelization is required to achieve high performance.

SpMV has been extensively studied in the past few decades, and various algorithms have been proposed to parallelize it on GPUs. One of the earliest and most popular algorithms is the compressed sparse row (CSR) format, which stores the non-zero elements of the matrix in three arrays: row indices, column indices, and values. CSR has been shown to perform well on GPUs due to its memory coalescing properties and efficient use of shared memory [1].

Another popular algorithm is the compressed sparse column (CSC) format, which stores the non-zero elements of the matrix in three arrays: column indices, row indices, and values. CSC has been shown to perform well on GPUs due to its efficient use of global memory and parallel reduction [2].

The hybrid ELLPACK-R format (HYB) combines the advantages of CSR and CSC by storing the matrix in two formats: ELLPACK-R for dense rows and CSR for sparse rows. HYB has been shown to perform well on GPUs for matrices with varying densities [3].

More recently, the compressed sparse blocks (CSB) format has been proposed, which stores the non-zero elements of the matrix in a block format, where each block is a dense submatrix. CSB has been shown to perform well on GPUs for matrices with a large number of non-zero elements per row [4].

Each of these algorithms has its advantages and disadvantages. CSR and CSC are well-suited for matrices with a low and high number of non-zero

elements per row, respectively, but may suffer from poor performance for matrices with varying densities. HYB is more flexible and can handle matrices with varying densities but requires additional storage overhead. CSB is well-suited for matrices with a large number of non-zero elements per row, but may suffer from poor performance for matrices with a low number of non-zero elements per row.

CSR Adaptive is a sparse matrix vector multiplication ***(SpMV)*** algorithm that is used in scientific computing. It is known for its ability to handle matrices with varying row sizes efficiently. In this project CSR Adaptive is used a one of the methods because it has been shown to outperform other algorithms in certain situations. The algorithm involves a CPU procedure and a GPU kernel that work together to divide the matrix into row-blocks and allocate shared memory space to each block [5].

LightSpMV is a sparse matrix-vector multiplication (SpMV) algorithm that dynamically distributes rows to subsections of warps called vector lanes, with the number of lanes determined by the host code based on the average number of nonzero elements per row of the input matrix. This method has several advantages, including efficient use of memory, fast performance due to the use of shuffle instructions instead of shared memory, and dynamic row retrieval and broadcasting over vectors through a global row management data structure. Additionally, LightSpMV can be easily parallelized and scaled for larger matrices, making it a promising approach for high-performance computing applications [6].

SpMV is an important operation in scientific computing, and parallelization is required to achieve high performance. Various algorithms have been proposed to parallelize SpMV on GPUs, each with its advantages and disadvantages. The choice of algorithm depends on the characteristics of the matrix being multiplied and the target GPU architecture. The runtime, GFLOPS, and BBs were evaluated to compare the efficiency of CSR adaptive and LightSpMV with OpenMP SpMV using mixed precision. The bottlenecks and bandwidth utilization were measured to identify the most effective method. A sensitivity analysis was performed to determine the optimal block size and achieve higher bandwidth utilization.

Methodology

A. CSR Adaptive

The CSR Adaptive method [5] is an algorithm used for sparse matrix vector multiplication (SpMV) in scientific computing. The algorithm involves a CPU procedure and a GPU kernel. The CPU procedure divides the matrix into row-blocks by creating an array of pointers that point to the starting and ending row numbers of all consecutive rows whose elements are less than or equal to a certain block size. If a row is too large to fit into the desired block size, it is allotted one full block. The GPU kernel allocates a shared memory space exactly equal to the block dimensions. Each thread loads one element of the input matrix and input vector, multiplies them, and stores the product into shared memory. Finally, the products for each row are summed up by one thread per row. The number of blocks launched by the kernel is equal to the number of row blocks as calculated by the CPU procedure. If the size of a row is larger than the size of the shared memory allocated, the algorithm uses CSR Vector to process that row block. In the CSR Vector implementation used for CSR Adaptive, all threads of the block are used to process the large row instead of only one warp.

This resulted in a significant improvement in performance for matrices with very large rows, making CSR Adaptive the best performing algorithm for such matrices [5]. The advantage of CSR Adaptive is that it can handle matrices with varying row sizes efficiently. The code can be found on ‘spmvCSRadaptiveGPU.h‘

B. LightSpMV

LightSpMV is a method for sparse matrix-vector multiplication that dynamically distributes rows to subsections of warps called vector lanes, with the number of vector lanes for each warp being determined by the host code based on the average number of nonzero elements per row of the input matrix. This approach relies on a global row management (GRM) data structure that manages the distribution of rows by accepting new requests for row indices from vectors and responding with the corresponding values. The use of shuffle instructions allows for the exchange of variables between threads within a warp without the use of shared memory, resulting in faster execution. The number of blocks launched by the kernel can be variable in this algorithm, but it is set to the number of rows divided by the block dimensions. The advantages of LightSpMV include improved efficiency due to dynamic row retrieval and broadcasting over vectors, as well as faster execution due to the

use of shuffle instructions instead of shared memory [6 and 7]. The code is provided by 'spmvLightGPU.h'.

C. OpenMP for SpMV on CPU

The provided code is a template function for computing Sparse Matrix-Vector Multiplication (SpMV) using Compressed Sparse Row (CSR) format. The template function takes as input the matrix dimensions (m), pointers to the values, row pointers, and column indices arrays, the input vector x, and output vector y. The function computes the SpMV product y = Ax iteratively for ITER number of iterations using OpenMP parallelization. The outer loop runs ITER number of times, and the inner loop calculates the dot product of the ith row with the input vector x. The dot product is performed using mixed precision with PREC representing the precision type of the matrix values, and YT representing the precision type of the output vector y. To improve the performance of SpMV on CPU, other methods, including Thread Affinity, Thread Pinning, and Loop Unrolling, were implemented. Except Loop Unrolling, the other two methods couldn't enhance the runtime. However, Loop Unrolling was causing errors on single precision. The code is written in 'spmv.h'.

The tests were run on CUDA 12 on a single device NVIDIA GeForce GTX 1050 Ti with following characteristics:

Compute capability of device: 6.1
Shared memory per block of device: 49152 bytes
Registers per block of device: 65536
Warp size of device: 32 threads
Max threads per block of device: 1024 threads
Max thread dimensions of device: 1024 x 1024 x 64
Total constant memory of device: 65536 bytes
Multiprocessor count of device: 6
Memory bus width of device: 128 bits
L2 cache size of device: 1048576 bytes
Max threads per SM of device 0: 2048

To obtain the mentioned information, please refer to 'info.cu' and compile the code with 'nvcc'.

The data from three datasets are extracted in CSR format from the Ma-

trix Market Exchange format using the GSL library on the CPU. The extracted data is then transferred to the GPU memory using CUDA. The code can be find in 'readcsr.h'. The matrices utilized in this study comprise a diverse range of data types and sizes, including those employed in the research papers showcasing the aforementioned algorithms. These matrices were obtained from the University of Florida Sparse Matrix Collection and are outlined below:

Table 1: Input Matrix characteristics

Name	Dimension	NNZ	NNZ/row
af_shell10	1,508,065 x 1,508,065	27,090,195	17.96
nlpkkt80	1,062,400 x 1,062,400	14,883,536	14.01
StocF-1465	1,465,137 x 1,465,137	11,235,263	7.67

The grid dimensions for each algorithm are configured to meet their specific requirements. In cases where there are no specific requirements, the dimensions are optimized based on performance across various data sizes. Global memory is used to store all input and output data arrays in all algorithms, with analysis conducted on both single and double precision versions of the input. To ensure the accuracy of the results, a comparison is made between both methods and SpMV on CPU, which is the same algorithm used in a previous project for this course.

The code calculates the following performance metrics:

1. Computational throughput in GFLOP/s: This metric is based on the fact that there are two floating point operations required for each element in the input matrix. The calculation for GFLOP/s is given by:

$$GFLOP/s = \frac{2 \times N}{t \times 10^9}$$

Here, N represents the number of elements in the input matrix and t is the time taken by the kernel in seconds.

2. Effective memory bandwidth in GB/s: It represents the rate at which data can be transferred between the memory and the processor, and is a commonly used metric to evaluate the memory performance of a system. It is represented as follows:

$$GB/s = \frac{\text{number of bytes processed}}{t} \times \frac{1}{10^9}$$

Results and Analysis

The total number of threads launched is equal to the product of the number of threads per block and the number of blocks in the grid. The number of threads per block is limited by the maximum number of threads per block supported by GPU's compute capability. The number of blocks in the grid depends on the problem size and the number of threads per block. A good rule of thumb is to choose a grid size that is large enough to fully utilize the GPU, but not so large that it exceeds the maximum number of blocks allowed by GPU's compute capability.

Based on the compute capability reported by device's GPU (6.1), the maximum number of threads per block supported by the GPU is 1024. It's important to note that the maximum number of blocks in the grid may be further limited by available device memory or other factors, so it's important to carefully choose the number of blocks to use in the code based on the specific requirements of of the algorithm. To tackle this issue a sensitivity analysis is need to find the optimal block size.

Based on Figure 1 and Figure 2, for both single precision and double precision computations, LightSpMV consistently outperforms CSR-Adaptive in terms of GB/s. The performance of both methods decreases as the block size increases from 512 to 1024. This is likely due to the increase in memory usage and cache misses as the block size increases. In single precision, CSR-Adaptive has a lower peak performance than LightSpMV. However, the difference is less pronounced than in double precision. In double precision, LightSpMV has a significantly higher peak performance than CSR-Adaptive. This is likely due to the more efficient use of memory bandwidth by LightSpMV.

The same sensitivity analysis were conducted for other matrices, and it was seen the optimal value of block size (the peak) was different for others. Seemingly, the optimal value depends on matrix specifications as well as GPU capability. In all the following results, 512 block size is chosen.

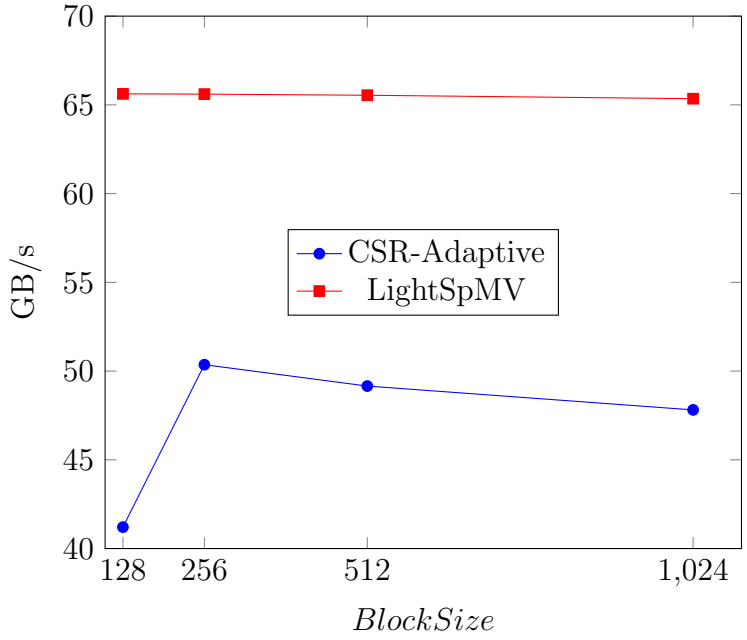

Figure 1: GB/s in Single precision for nlpkkt80.

Table 2 shows the runtime for Adaptive-CSR (ACSR) and LightSpMV (LS) compared with computation on CPU using 8 threads (CPU8). The comparison shows that LightSpMV outperforms CSR-Adaptive in terms of GB/s for both single and double precision computations, in all matrices. However, LS may produce different results compared to the real result computed by CPU8, with a maximum difference of 0.2 in LS for af_shell10 and infinity for StocF-1465 in single precision computations. For double precision computations, LS produces the exact same results as CPU8 for all three sparse matrices tested.

The runtime comparison for three sparse matrices using CPU8, ACSR, and LS shows that LS is faster than CPU8 and ACSR for both single and double precision computations, with CPU8/LS ratio ranging from 7.0 to 15.2 and CPU8/ACSR ratio ranging from 4.0 to 11.4.

Figure 3 shows the performance of two methods, CSR-Adaptive and LightSpMV, in terms of GB/s as a function of NNZ/row (the number of non-zeros per row) for double-precision. In the context of sparse matrix

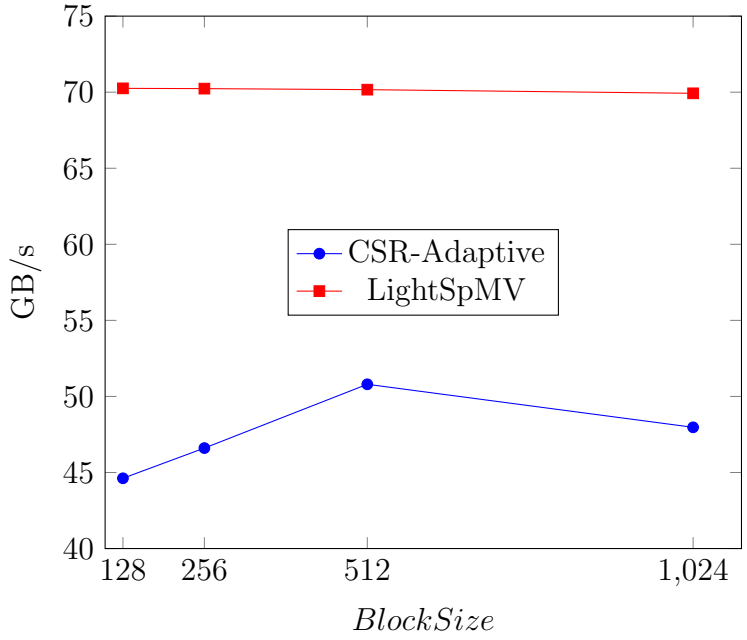

Figure 2: Runtime with Double precision for nlpkkt80.

Table 2: Runtime

Precision	Name	CPU8 (s)	CPU8/ACSR	CPU8/LS	max dif
Single	af_shell10	0.027826	5.8	9.3	0.2 in LS
	nlpkkt80	0.030623	11.4	15.2	0
	StocF-1465	0.013622	6.7	7.0	inf in LS
Double	af_shell10	0.028644	4.0	7.3	0
	nlpkkt80	0.027181	6.9	9.5	0
	StocF-1465	0.015093	5.0	6.1	0

computations, GB/s is a measure of the speed of the matrix-vector multiplication operation, which involves reading the sparse matrix and dense vector data from memory, performing the computation, and writing the result back to memory. The higher the GB/s, the faster the operation can be performed.

For CSR-Adaptive, the performance in GB/s decreases as NNZ/row increases, which means that this method is less efficient when the sparsity

of the matrix decreases. On the other hand, for LightSpMV, the performance in GB/s increases as NNZ/row increases, indicating that this method is more efficient for matrices with lower sparsity. At low NNZ/row values, CSR-Adaptive outperforms LightSpMV. However, as NNZ/row increases, LightSpMV becomes more efficient and eventually surpasses CSR-Adaptive in terms of performance. At around NNZ/row=14, LightSpMV reaches its peak performance, while CSR-Adaptive continues to decline.

Overall, this graph suggests that the choice between CSR-Adaptive and LightSpMV depends on the sparsity of the matrix. For matrices with low sparsity, CSR-Adaptive may be the better choice, while for matrices with high sparsity, LightSpMV may provide better performance.

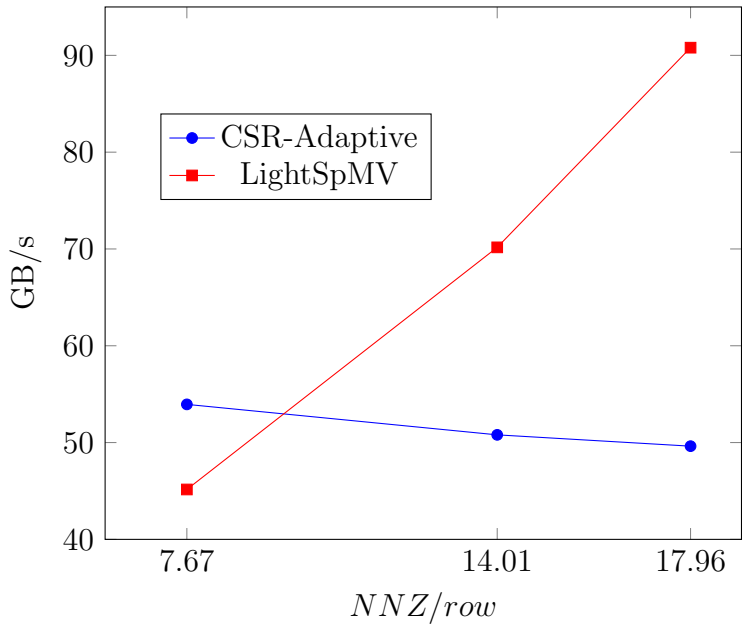

Figure 3: GB/s vs. NNZ/row (Double precision).

Figure 4 represents the performance of two methods, CSR-Adaptive and LightSpMV, for doing A*x operation on a sparse matrix in CSR format and a dense matrix x. The performance is measured in GFLOPS (Giga Floating Point Operations Per Second) and is plotted against the NNZ/row (number of non-zeros per row) of the sparse matrix.

From the graph, we can observe that for all three values of NNZ/row,

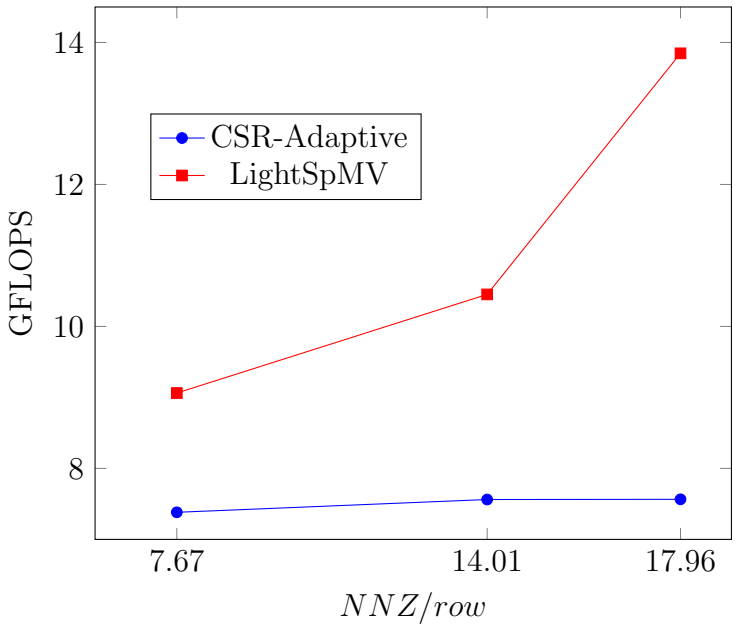

Figure 4: GFLOPS vs. NNZ/row (Double precision).

the LightSpMV method performs better than the CSR-Adaptive method in terms of GFLOPS. As the NNZ/row increases, the GFLOPS of both methods decreases, but the decrease is relatively slower for LightSpMV. At NNZ/row=17.96, LightSpMV achieves a peak performance of 13.85 GFLOPS, while CSR-Adaptive achieves only 7.56 GFLOPS.

The performance of both methods is highly dependent on the sparsity structure of the matrix, which is reflected by the NNZ/row metric. Therefore, for different matrices with different sparsity structures, the relative performance of these methods may vary. However, in this case, we can conclude that for the given matrices and precision, the LightSpMV method outperforms the CSR-Adaptive method in terms of GFLOPS.

The previous methods are designed to take advantage of the highly parallel architecture of CUDA-enabled devices, while the CPU implementation using OpenMP attempts to parallelize the code for a CPU architecture.

Comparing the data, we can see that the CPU implementation is sig-

nificantly slower in terms of both GFLOP/s and GB/s compared to the CUDA-enabled device. For example, when NNZ/row is 17.96, the GFLOP/s and GB/s are 1.891488 and 3.371469, respectively, for the CPU implementation, while the same values for the Adaptive-CSR and LightSpMV on the CUDA-enabled device are 49.628232 and 90.792728 for GB/s, and 7.564493 and 13.847355 for GFLOP/s, respectively. This indicates that the CUDA-enabled device performs significantly better than the CPU implementation in terms of both GFLOP/s and GB/s, especially when dealing with large sparse matrices.

One reason why these methods are better is that they take advantage of the highly parallel architecture of CUDA-enabled devices, which can perform many calculations simultaneously, resulting in a significant increase in performance compared to CPU implementations. Additionally, these methods use specific data structures and algorithms that are optimized for sparse matrices, which further improves their performance.

In summary, the data presented here shows that the Adaptive-CSR and LightSpMV methods on a CUDA-enabled device outperform a CPU implementation using OpenMP in terms of both GFLOP/s and GB/s. This highlights the importance of using specialized methods and architectures that are optimized for specific tasks, such as sparse matrix multiplication.

Conclusion

The performance of sparse matrix multiplication using different methods depends heavily on the characteristics of the matrix being used. In general, the performance improves as the number of non-zeros per row increases, but this trend may not hold for all matrices.

The CSR-Adaptive and LightSpMV methods were evaluated on NVIDIA CUDA and were found to have higher GFLOPS and GB/s compared to CPU-based OpenMP parallelization. These methods are better suited for sparse matrix multiplication on large matrices.

GB/s can be used to evaluate the effective memory bandwidth of the system. However, it is important to consider other factors such as the memory hierarchy and cache utilization when analyzing the performance of sparse matrix multiplication.

Overall, selecting the appropriate method for sparse matrix multiplication

requires careful consideration of the matrix properties, system architecture, and performance metrics.

References

[1] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1-11.

[2] J. D. Owens, M. Garland, D. Luebke, W. R. Mark, J. C. McMillan, and T. J. Purcell, "GPU parallel computing for mixed-size matrix multiplication," in Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2005, pp. 154-162.

[3] T. Pham, M. Girkar, and P. Sadayappan, "Optimizing sparse matrix-vector multiplication on GPUs," in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, 2009, pp. 1-12.

[4] Y. Zhang, Y. Yang, X. Sun, and X. Chen, "Efficient sparse matrix-vector multiplication on GPUs using compressed sparse blocks," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 6

[5] Greathouse, J. L., Daga, M. "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format." SC14: Inter- national Conference for High Performance Computing, Networking, Storage and Analysis, 2014, doi:10.1109/sc.2014.68.

[6] CUDA C Programming Guide. NVIDIA Developer Documentation. (n.d.). Retrieved December 18, 2018,from https://docs.nvidia.com/cuda/cuda-c-programming- guide/index.htmlwarp-shuffle functions.

[7] CUDA Pro Tip: Do The Kepler Shuffle. (2016, January 11). Retrieved December 18, 2018, from https://devblogs.nvidia.com/cuda-pro-tip- kepler-shuffle/