Faster Compressed Sparse Row (CSR)-based Sparse Matrix-Vector Multiplication using CUDA

Yongchao Liu, Jorge González-Domínguez, Bertil Schmidt
Institute of Computer Science, University of Mainz, Germany
Emails: {liuy, j.gonzalez, bertil.schmidt}@uni-mainz.de

Abstract

LightSpMV [1] is a novel CUDA-compatible sparse matrix-vector multiplication (SpMV) algorithm using the standard compressed sparse row (CSR) storage format. It achieves high speed by benefiting from the fine-grained dynamic distribution of matrix rows over vectors, where a warp is virtualized as a single instruction multiple data (SIMD) vector and can be further split into a set of equal-sized smaller vectors for finer-grained processing.

In LightSpMV, we have investigated two dynamic row distribution approaches at the vector and warp levels with atomic operations and warp shuffle functions as the fundamental building blocks. We have evaluated LightSpMV using various sparse matrices and further compared it to the CSR-based SpMV subprograms in the state-of-the-art CUSP [2] and cuSPARSE [3] libraries. Performance evaluation reveals that on a single Tesla K40c GPU, LightSpMV is superior to both CUSP and cuSPARSE, with a speedup of up to 2.60 and 2.63 over CUSP, and up to 1.93 and 1.79 over cuSPARSE for single and double precision, respectively. The source code of LightSpMV is available at http://lightspmv.sourceforge.net.

Compressed Sparse Row (CSR) Format

- A frequently used format for sparse matrix storage in CPU-centric software
- Efficient compression of structured and un-structured sparse matrices
- Good amenability to efficient algorithms designed for CPUs
- Enables good SpMV performance on CPUs, but shows a relatively low performance on GPUs
- Uses three separate vectors: *row_offsets*, *column_indices*, and *values* to represent a matrix

$$A = \begin{bmatrix} 0.1 & 0.7 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 \\ 0.5 & 0 & 0.3 & 0.9 \\ 0 & 0.6 & 0 & 0.4 \end{bmatrix}$$

row_offsets =	0	2	4	7	9				
column_indices =	0	1	1	2	0	2	3	1	3
values =	0.1	0.7	0.2	0.8	0.5	0.3	0.9	0.6	0.4

CSR representation of an example sparse matrix

Sparse Matrix-Vector Multiplication

General SpMV equation:

$$y = \alpha A x + \beta y$$

- A is a sparse matrix of size $R \times C$ with NNZ non-zeros
- x is the source vector of size C
- y is the destination vector of size R
- α and β are scalars

```
procedure sequentialCSRSpMV()
                                        ▷ iterate each row
    for (i = 0; i < R; ++i) do
                                        ▷ compute the dot product of two vectors
        sum = 0;
        for (j = row_offsets[i]; j < row_offsets[i + 1]; ++j) do
            sum += values[j] * x[column_indices[j]];
        end for
                                        ▷ finalize and save the multiplication result
        y[i] = α * sum + β * y[i];
    end for
end procedure
```

Pseudocode of the sequential SpMV using CSR

Host-side SpMV Driver Routine

- Dynamic determination of vector size based on average row length
- Do not need any host-side pre-processing of the CSR data structure
- Launch only a single kernel to perform the SpMV operation.
- CUDA kernels are implemented as CUDA C++ template functions

```
procedure spmvHostDriver(cudaDevicePropt& prop, ...)
                                        ▷ set thread block and kernel grid configuration
    T = prop.maxThreadsPerBlock;
    B = prop.multiProcessorCount
            * prop.maxThreadsPerMultiProcessor / numThreadsPerBlock;
                                        ▷ reset row_counter to zero
    cudaMemset(row_counter, 0, sizeof(int));
                                        ▷ calculate the average row length
    mean = rint(N_nz / R);
                                        ▷ launch the kernel
    if (mean ≤ 2) then                  ▷ set the vector size to 2
        spmvCudaKernel <<< B, T >>> (2, ...);
    else if (mean ≤ 4) then             ▷ set the vector size to 4
        spmvCudaKernel <<< B, T >>> (4, ...);
    else if (mean ≤ 64) then            ▷ set the vector size to 8
        spmvCudaKernel <<< B, T >>> (8, ...);
    else                                ▷ set the vector size to 32
        spmvCudaKernel <<< B, T >>> (32, ...);
    end if
end procedure
```

Pseudocode of the host-side driver for SpMV kernel invocation

Vector-Level Dynamic Row Distribution

- Initially, each vector obtains a row index i from a global row management (GRM) data structure, and computes $y[i]$.
- GSR contains an integer-type variable *row_counter*, which is stored in global memory and represents the lowest row index among all unprocessed rows.
- When a vector has completed its current row, it will retrieve a new row from GRM by incrementing *row_counter* through an atomic addition operation.
- The first thread of each vector takes charge of the new row retrieval and broadcasts the new row index to all of the other threads in the vector.
- Warp shuffle functions are used for row index broadcasting and intra-vector reduction for vector dot product.

```
function getRowIndexVector()
                                        ▷ compute the lane ID of each thread
    laneId = threadIdx.x % V;
                                        ▷ get the row index
    if (laneId == 0) then
        row = atomicAdd(row_counter, 1);
    end if
                    ▷ broadcast the row index to all other threads within the vector
    return (row = __shfl(row, 0, V));
end function
```

Pseudocode for vector-level row distribution

```
procedure spmvCudaKernel()
                                ▷ get the lane ID and vector ID for the thread
    laneId = threadIdx.x % V;
    vectorId = threadIdx.x / V;
    __shared__ volatile int space[NUM_VECTORS_PER_BLOCK][2];
                                ▷ get a row index
    row = getRowIndexVector();
    while (row < R) do
                                ▷ get the starting and end offsets for the row
        if (laneId < 2) then
            space[vectorId][laneId] = row_offsets[row + laneId];
        end if
        row_start = space[vectorId][0];
        row_end = space[vectorId][1];
                                ▷ compute the dot product of the vector
        sum = 0;
        if (V == warpSize) then
            i = row_start - (row_start & (warpSize - 1)) + laneId;
            if (i ≥ row_start && i < row_end) then
                sum += values[i] * x[column_indices[i]];
            end if
            for (i += V; i < row_end; i += V) do
                sum += values[i] * x[column_indices[i]];
            end for
        else
            for (i = row_start + laneId; i < row_end; i += V) do
                sum += values[i] * x[column_indices[i]];
            end for
        end if
                                ▷ intra-vector reduction
        sum *= α;
        for (i = V >> 1 ; i > 0; i >>= 1) do
            sum += __shfl_down(sum, i, V);
        end for
                                ▷ save the result
        if (laneId == 0) then
            y[row] = sum + β * y[row];
        end if
                                ▷ get a new row
        row = getRowIndexVector();
    end while
end procedure
```

Pseudocode for the vector-level CUDA kernel

Warp-Level Dynamic Row Distribution

- Only one atomic operation is issued for a warp
- Distributes $warpSize \,/\, V$ rows to a single warp at a time
- Obtains the warp-level CUDA kernel by replacing the function *getRowIndexVector* with the function *getRowIndexWarp*.

```
function getRowIndexWarp()
            ▷ compute the lane ID and vector ID of each thread within the warp
    warpLaneId = threadIdx.x & (warpSize - 1);
    warpVectorId = warpLaneId / V;
                                        ▷ get the row index
    if (warpLaneId == 0) then
        row = atomicAdd(row_counter, warpSize / V);
    end if
                ▷ broadcast the row index to all other threads within the vector
    return (row = __shfl(row, 0, warpSize) + warpVectorId);
end function
```

Double Precision Support

Intra-vector reduction for double precision.

- Overloads the *__shfl_down* function for double
- Uses the *reinterpret_cast* compiler directive
- Uses integer *__shfl_down* to exchange data

```
function __shfl_down(value, delta, vectorSize)
    int2 tmp = *reinterpret_cast<int2*>(&value);
    tmp.x = __shfl_down(tmp.x, delta, vectorSize);
    tmp.y = __shfl_down(tmp.y, delta, vectorSize);
    return *reinterpret_cast<double*>(&tmp);
end function
```

Texture fetch for double precision

- Uses texture object API to reinterpret a double-type value to an int2-type value
- Uses the *__hiloint2double* function to recover the double-type value

```
function texFetch(x, i)
    int2 tmp = tex1Dfetch<int2>(x, i);
    return __hiloint2double(tmp.y, tmp.x);
end function
```

Benchmark Sparse Matrices

- 14 sparse matrices are used for performance evaluation
- One half are from NVIDIA Research [4]
- The other half are from the University of Florida sparse matrix collection [5]
- Average row lengths range from 3 up to 2,633 with standard deviations varying from 0 up to 4,210

Name	Rows / Cols	N_{nz}	μ / σ	Src
webbase-1M	1,000,005	3,105,536	3 / 25	N
dblp-2010	326,186	1,615,400	5 / 8	F
in-2004	1,382,908	16,917,053	12 / 37	F
uk-2002	18,520,486	298,113,762	16 / 28	F
cop20k_A	121,192	2,624,331	22 / 14	N
eu-2005	862,664	19,235,140	22 / 29	F
indochina-2004	7,414,866	194,109,311	26 / 216	F
nlpkkt120	3,542,400	96,845,792	27 / 3	F
qcd5_4	49,152	1,916,928	39 / 0	N
rma10	46,835	237,4001	51 / 28	N
pwtk	217,918	11,634,424	53 / 5	N
shipsec1	140,874	7,813,404	55 / 11	N
kron_g500-logn21	2,097,152	182,082,942	87 / 756	F
rail4284	4,284 / 1,092,610	11,279,748	2,633 / 4,210	N

Performance Evaluation

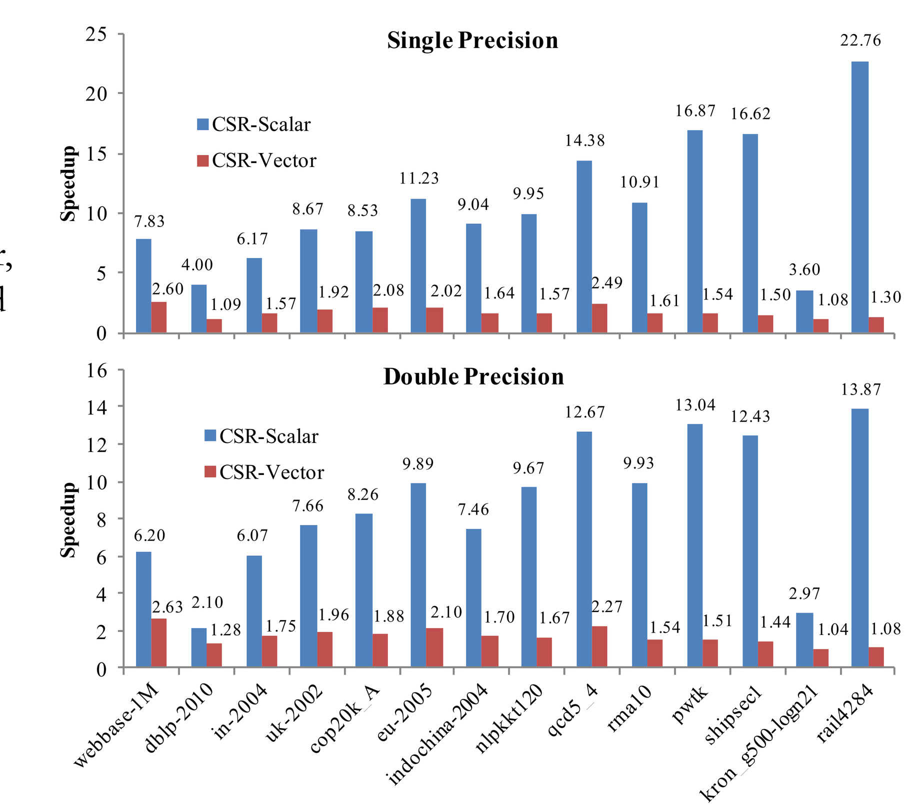
- A Kepler-based Tesla K40c GPU and CUDA 6.5 toolkit
- The vector-level kernel produces an average performance of 14.8 GFLOPS with the maximum performance of 27.0 GFLOPS for single precision, and an average performance of 12.2 GFLOPS with the maximum performance of 20.9 GFLOPS for double precision
- The warp-level kernel yields an average performance of 21.7 GFLOPS with the maximum performance of 32.0 GFLOPS for singe precision, and an average performance of 16.6 GCUPS with the maximum performance of 23.8 GFLOPS for double precision

Matrix	Warp / Vector (GFLOPS)		Speedup	
	Single	Double	Single	Double
webbase-1M	14.7 / 3.6	13.0 / 3.5	4.15	3.71
dblp-2010	11.4 / 5.1	9.6 / 4.8	2.25	1.98
in-2004	19.3 / 10.4	15.6 / 9.4	1.85	1.66
uk-2002	22.0 / 13.0	17.7 / 11.5	1.70	1.54
cop20k_A	22.6 / 13.4	16.2 / 11.6	1.69	1.40
eu-2005	24.1 / 15.5	18.9 / 13.4	1.55	1.41
indochina-2004	22.5 / 15.8	17.4 / 13.1	1.42	1.34
nlpkkt120	25.3 / 15.1	19.3 / 12.7	1.68	1.52
qcd5_4	31.9 / 21.4	23.8 / 17.8	1.49	1.34
rma10	28.0 / 22.5	21.4 / 18.0	1.24	1.19
pwtk	31.0 / 27.0	23.0 / 20.9	1.15	1.10
shipsec1	32.0 / 26.3	23.3 / 20.4	1.22	1.14
kron_g500-logn21	4.8 / 4.8	4.0 / 4.0	1.00	1.00
rail4284	13.5 / 13.5	9.3 / 9.3	1.00	1.00

`Warp` and `Vector` denote the warp-level and vector-level kernel, respectively; `Single` and `Double` denote single and double precision, respectively.

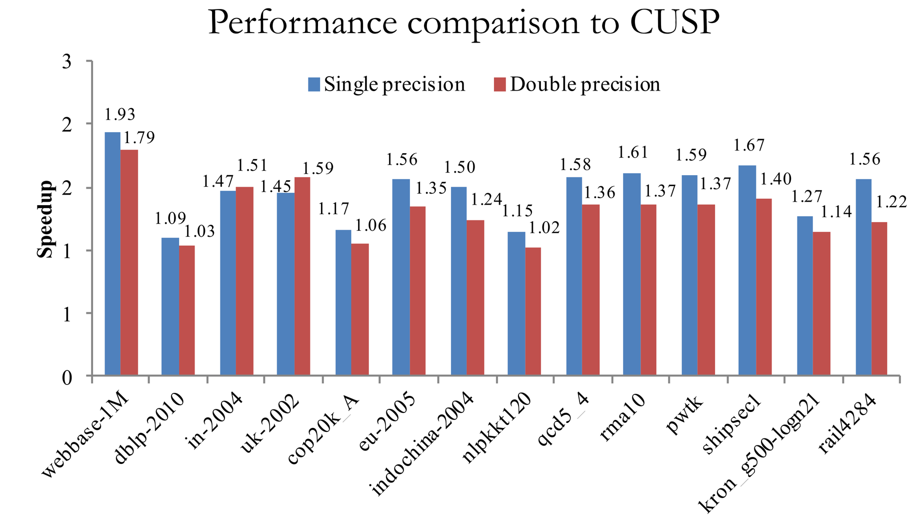
Performance of the vector-level and warp-level kernels

- Two CSR-based SpMV subprograms in CUSP: *spmv_csr_scalar_tex* (CSR-Scalar) and *spmv_csr_vector_tex* (CSR-Vector)
- LightSpMV is far superior to CSR-Scalar, achieving average speedups of 10.76 and 8.73 with maximum speedups of 22.76 and 13.87 for single and double precision, respectively
- Compared to CSR-Vector, the average speedups of LightSpMV are 1.72 and 1.70, and the maximum speedups are 2.60 and 2.63 for single and double precision, respectively



Performance comparison to CUSP

- Two CSR-based SpMV subprograms in cuSPARSE: *cusparseScsrmv* and *cusparseDcsrmv* for single and double precision, respectively
- LightSpMV outperforms cuSPARSE for each case, with the average speedup of 1.47 and the maximum speedup of 1.93 for single precision, and an average speedup of 1.32 with the maximum speedup of 1.79 for double precision



Performance comparison to cuSPARSE

References

1. Y. Liu and B. Schmidt: **LightSpMV: Faster CSR-based Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs.** *26th IEEE International Conference on Application-specific Systems, Architectures and Processors,* 2015, ready to submit.
2. N. Bell and M. Garland: **CUSP : Generic Parallel Algorithms for Sparse Matrix and Graph Computations (v0.4).** *http://cusplibrary.github.io*, 2014
3. NVIDIA: **The NVIDIA CUDA Sparse Matrix Library (cuSPARSE),** In CUDA 6.5 toolkit, 2014
4. N. Bell and M. Garland: **Implementing Sparse Matrix-Vector Multiplication on Throughput-oriented Processors.** *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis,* 2009
5. T. A. Davis and Y. Hu: **The University of Florida Sparse Matrix Collection.** *ACM Transactions on Mathematical Software,* 38 (1), 2011