Adapting an Existing Multilevel Compressed Sparse Row Format for General Purpose GPU Programming

Recap of CSR-k

- CSR-k is a multilevel CSR representation
- Uses several tiers of super-rows
- Originally for CPU SpTRSV and SpMV
  - Aided in cache access for older CPUs with poor caching algorithms
- Do we see similar performance gains on a GPU?

Our Goal: SpMV

- SpMV is a very simple kernel
- Iterate over the rows of the sparse matrix
- Multiply each row by the column vector
- Store the result into the output vector
- Repeat until iterated through all rows of the sparse matrix
- $O(n)$ where n is the number of nonzeros

CSR-k on GPUs

- On a GPU, instead of mapping the layers of CSR-k to cache levels, they instead map to block/grid dimensions
- This provides a very explicit hierarchy of parallelism, as we can parallelize across the 3D blocks
- The innermost loops map to the block's *x* and *y* axes, and the outermost ones map to a block's *z* axis and the grid dimensions

SpMV CSR-3 Algorithm

- There are 2 SpMV kernels, one with the innermost loop parallelized, and one with the innermost loop running in serial
- This is because certain very sparse matrices may see no speedup from running the innermost loop in parallel, and parallelizing the innermost loop can hurt purformance
- Other, denser matrices may see a significant speedup

SpMV CSR-3 Serial Algorithm

```
__global__ function cuSpMV_3(sup_sup_row_ptr[], sup_row_ptr[]
                             row_ptr[], col_idx[], vals[], x[], y[]) {
    let block = blockIdx.x
    let outer_start = sup_sup_row_ptr[block],
        outer_end = sup_sup_row_ptr[block + 1]

    for i = outer_start to outer_end parallelize across blockDim.y {
        let inner_start = sup_row_ptr[i], inner_end = sup_row_ptr[i + 1]

        for j = inner_start to inner_end parallelize across blockDim.x {
            let nnz_start = row_ptr[j], nnz_end = row_ptr[j + 1]
            let temp = 0.0

            for k = nnz_start to nnz_end no parallelization {
                temp += vals[k] * x[col_idx[k]]
            }

            y[j] = temp
        }
    }
}
```

Parallelization Techniques

- The grid is large enough that the outermost loop (iterating across super-super-rows) is unnecessary, since one block corresponds to one super-super-row
- The next two innermost loops are parallelized across the *x* and *y* axes of a block, being careful to preserve locality
- Locality is preserved by ensuring we parallelize in order from the *z* axis, to the *y* axis, and finally to the *x* axis, since threads along an *x* axis are spawned on the same warp, spilling over to *y* and *z* in order to capture 32 threads/warp

SpMV CSR-3 Parallel Algorithm

```
__global__ function cuSpMV_3(sup_sup_row_ptr[], sup_row_ptr[]
                             row_ptr[], col_idx[], vals[], x[], y[]) {
   let block = blockIdx.x
   let outer_start = sup_sup_row_ptr[block],
       outer_end = sup_sup_row_ptr[block + 1]

   for i = outer_start to outer_end parallelize across blockDim.z {
      let inner_start = sup_row_ptr[i], inner_end = sup_row_ptr[i + 1]

      for j = inner_start to inner_end parallelize across blockDim.y {
         let nnz_start = row_ptr[j], nnz_end = row_ptr[j + 1]
         let temp[blockDim.x] = fill(0.0)

         for k = nnz_start to nnz_end parallelize across blockDim.x {
            temp[threadIdx.x] += vals[k] * x[col_idx[k]]
         }

         y[j] = parallel_reduction(temp)
      }
   }
}
```

Parallelization Techniques

- Again, the grid is big enough that the outermost loop is unnecessary
- The next two loops are parallelized over the *z* and *y* dimensions of a block, also paying attention to locality
- In the CUDA implementation, we allocate a grid-local shared memory space to perform fast parallel reductions
- The innermost loop is parallelized across the *x* axis, which corresponds to threads in a warp

Finding Optimal Block Size

- Through empirical testing we found that a grid large enough to correspond one block to one super-super-row yields excellent performance
- We also found that the optimal block size was 4x8x12 (`dim3(4, 8, 12)`)

Finding Optimal Super-Row Sizes

- Finding optimal super-row and super-super-row sizes is time consuming, and usually requires a brute force search through several combinations to find an optimal representation of the sparse matrix
- This is highly dependent on the 4x8x12 block size, and optimal super-row/super-super-row sizes drastically changes with any change in block size

There is a Better Solution

- This was the state of the research a month ago, until we found a way to yield excellent performance by dynamically adjusting block and super-row sizes based on metadata about the sparse matrix, which is all done at runtime
- The next slide presents the algorithm used to determine optimal super-row size, super-super-row size, and grid dimensions

Heuristic Algorithm for Runtime Parameters

```
let d = nnz / m

let sup_sup_row_size = round(3.333 + 20 / (d * ln(d)))
let sup_row_size = round(0.667 * sup_sup_row_size + 2.667)

let blockDim.x = 8, blockDim.y = 12, blockDim.z = 1
let parallelize_inner_loop = false

if d > 8 {
    parallelize_inner_loop = true
    super_super_row_size *= 2
    super_row_size += 2
    blockDim.x = 4
    blockDim.y = 8
    blockDim.z = 12
}
if d > 16 {
    super_row_size += 1
    blockDim.x = 8
    blockDim.z = 8
}
if d > 32 {
    super_super_row_size += 1
    super_row_size -= 1
    blockDim.x = 16
    blockDim.z = 4
}
if d > 64 {
    super_row_size += 1
    blockDim.x = 32
    blockDim.y = 2
}
```

Explanation

- We assume a 2-dimensional block (no innermost loop parallelism) and adapt from there
- The initial block dimensions are set to 8x12
- The super-row and super-super-row sizes are computed using the following formula where *d* is the matrix density:

```
let sup_sup_row_size = round(3.333 + 20 / (d * ln(d)))
let sup_row_size = 0.667 * sup_sup_row_size + 2.667
```

About the Formula

- The formulas originally represented a logarithmic curve mapped to a normal distribution, but both of them have gone through several iterations of minor changes and simplification
- At their current state it's best to view them as black box formulas

The Rest of the Algorithm

- This provides good results for relatively sparse matrices, but falls short on denser matrices
- From there we begin to add parallelism to the innermost loop and adapt the block size
- This is done incrementally in order of increasing row density
- The denser the rows, the more parallelism on the inner loop
- The super-row and super-super-row sizes are adjusted accordingly through an empirically-derived heuristic

Test Matrices

- The following matrices are used in testing:

Matrix	NNZ	Dimension	NNZ Density
G3_circuit.mtx.rcm.csr	7660826	1585478	4.83187152392
ecology1.mtx.rcm.csr	4996000	1000000	4.996
Emilia_923.mtx.rcm.csr	40373538	923136	43.7352004472
bmwcra_1.mtx.rcm.csr	10641602	148770	71.5305639578
hugetric-00000.mtx.rcm.csr	17467046	5824554	2.99886411904
hugebubbles-00000.mtx.rcm.csr	54940162	18318143	2.99922115468
wave.mtx.rcm.csr	2118662	156317	13.5536250056
thermal2.mtx.rcm.csr	8580313	1228045	6.98696953288
delaunay_n20.mtx.rcm.csr	6291372	1048576	5.99991989136
brack2.mtx.rcm.csr	733118	62631	11.7053535789
packing-500x100x100-b050.mtx.rcm.csr	34976486	2145852	16.2995798405
hugetrace-00000.mtx.rcm.csr	13758266	4588484	2.9984339054
cont-300.mtx.rcm.csr	988195	180895	5.46280991736
fl2010.mtx.rcm.csr	2346294	484481	4.84290199203
wi2010.mtx.rcm.csr	1209404	253096	4.7784398015
roadNet-TX.mtx.rcm.csr	3843320	1393383	2.75826531542

Test Setup

- All tests are run on the Alabama Supercomputer Authority's Dense Memory Cluster
- There are two test beds used, described on the following slides

Test Bed 1

- This test bed used has two Xeon E5-2650v4 CPUs with 12 cores each, though the system is virtualized to only expose one core (since this is a GPU test)
- The GPU used is an Nvidia Tesla V100 with 32GB of VRAM

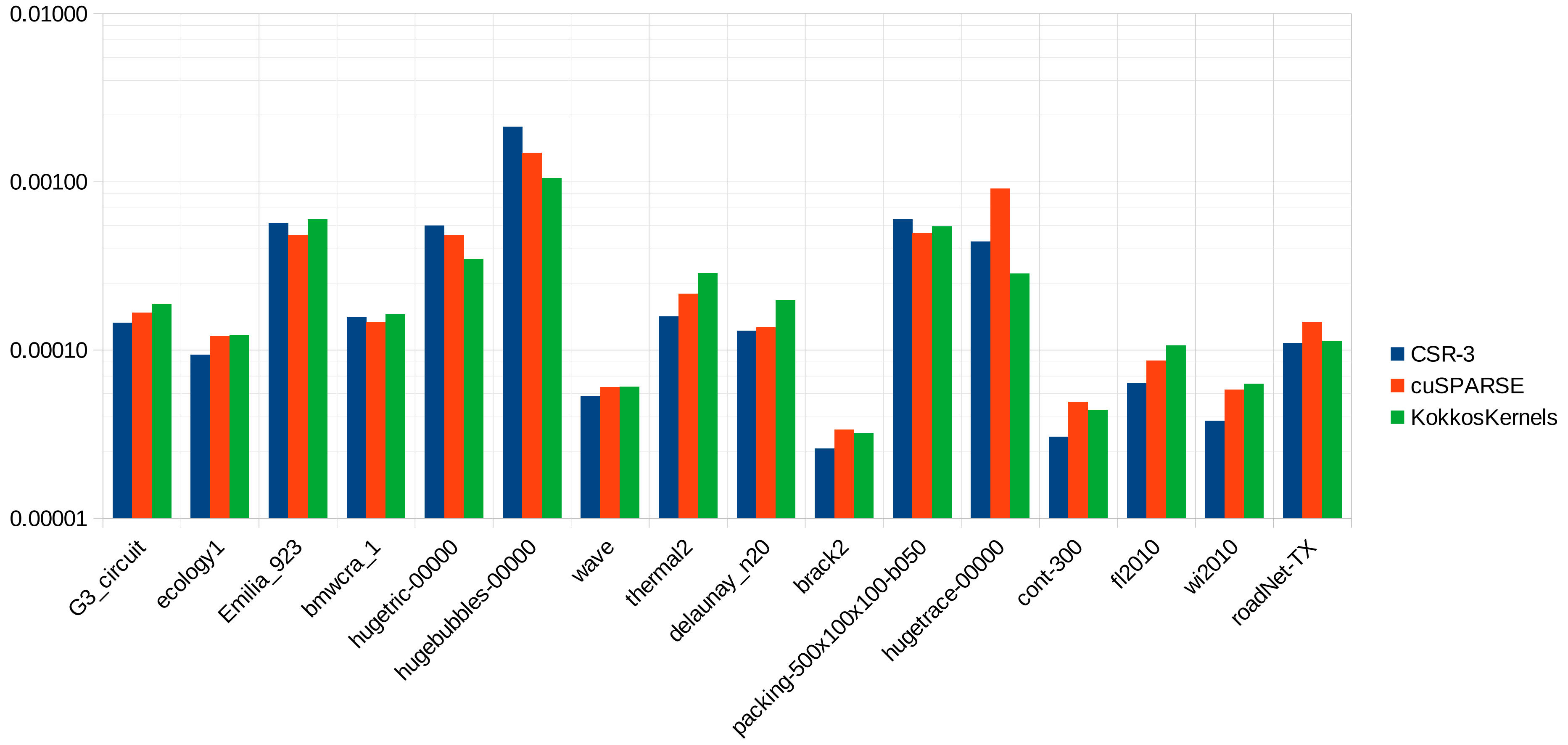
Test Bed 2

- This test bed used has two Epyc 7713 CPUs with 64 cores each, though the system is virtualized to only expose one core (since this is a GPU test)
- The GPU used is an Nvidia Ampere A100 with 80GB of VRAM

Test Methodology

- Our SpMV implementation using CSR-3 is tested against Nvidia’s cuSPARSE and Sandia’s KokkosKernels
- Our algorithm is fed with matrices in natural ordering and reordered to RCM using a multilevel banding algorithm
- cuSPARSE and KokkosKernels are fed with RCM-ordered matrices as reordered by Octave’s `symrcm` algorithm
- Each matrix is multiplied 20 times and the results averaged
- Timing only includes the kernel, and does not include copying to the device or reordering costs
- KokkosKernels is not tested on bed 2 because the current version of KokkosKernels does not support compilation for the Ampere architecture (compute capability 8.0)

Test Results (Bed 1, Lower is Better)



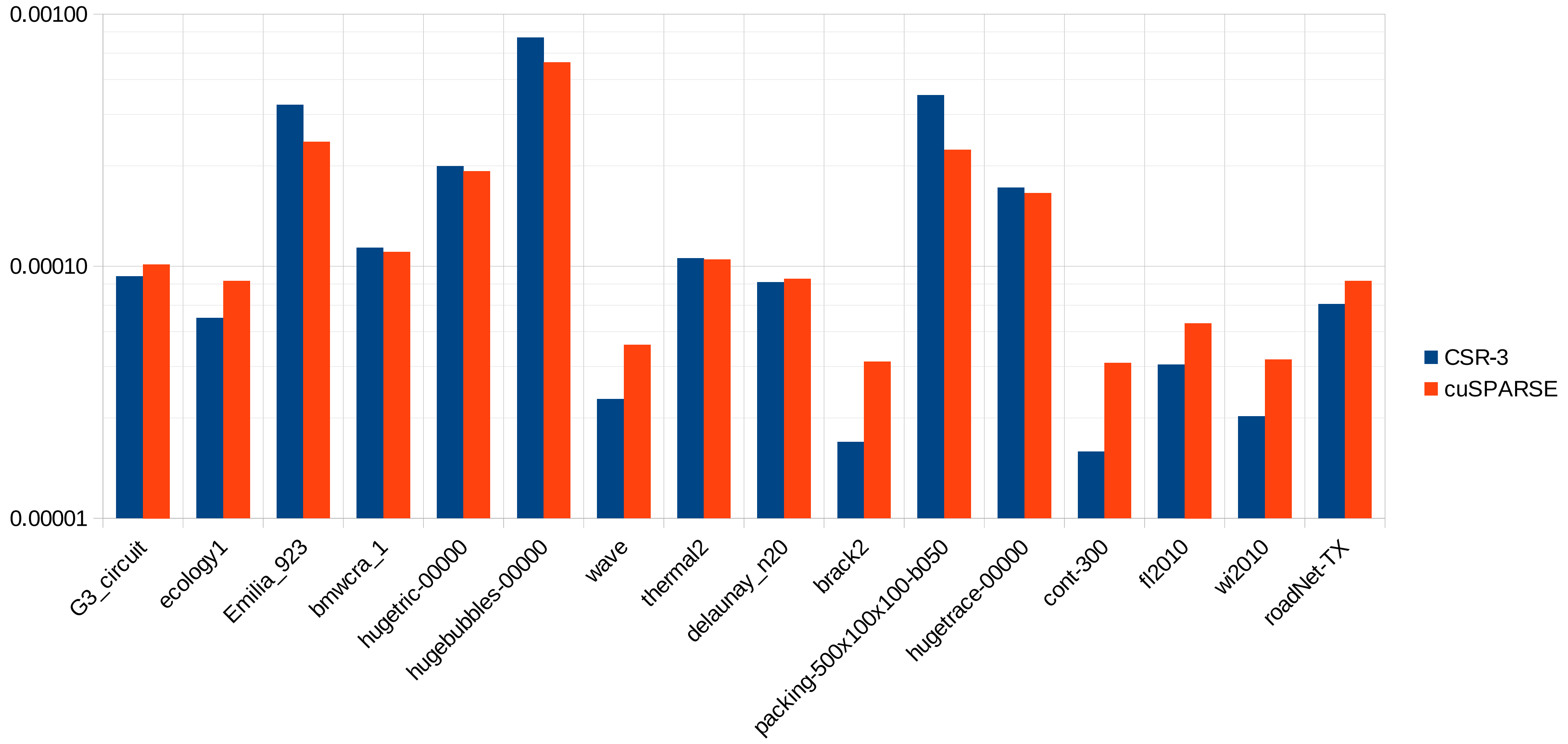
Analysis of Results (Bed 1)

- CSR-3 shared the majority of wins across the input set, performing better than both cuSPARSE and KokkosKernels in 10/16 tests
- KokkosKernels has only 3/16 matrices performing faster than CSR-3 or cuSPARSE
- cuSPARSE also has only 3/16 of the matrices favoring this algorithm

Average Speedup

- Speedup of CSR-3 is compared against both cuSPARSE and KokkosKernels by taking the geometric mean of all speedups, including when CSR-3 performed well and when it didn’t
- CSR-3 performed, on average, 16.6% faster than cuSPARSE, and 11.1% faster than KokkosKernels

Test Results (Bed 2, Lower is Better)



Analysis of Results

- CSR-3 again shared the majority of wins across the input set, performing better than cuSPARSE in 9/16 tests
- cuSPARSE came in second with 7/16 of the matrices favoring this algorithm

Average Speedup

- Speedup of CSR-3 is compared against cuSPARSE by taking the geometric mean of all speedups, including when CSR-3 performed well and when it didn’t
- CSR-3 performed, on average, 16% faster than cuSPARSE

Conclusion

- Our SpMV implementation comes with a Band-3 reordering algorithm and two GPU kernels that are selectively chosen based on matrix metadata
- Even for a very, very simple kernel, it performs very well, even when competing algorithms are fed matrices in RCM ordering
- The heuristic algorithm used to calculate optimal block dimensions and super-row sizes is a one-time operation that can be calculated very cheaply, since it only needs to know matrix dimensions and the number of nonzeros

Acknowledgement

- This work was made possible in part by a grant of high performance computing resources and technical support from the Alabama Supercomputer Authority.

- Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1 (December 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663
- H. Carter Edwards and Christian R. Trott and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing 74, 12, pages 3202 - 3216. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003