Optimizing Sparse Matrix-Vector Multiplication via Dynamic Register Blocking and Reordered Cache Blocking

Nithin Tumma, Neel Patel
CS265, Harvard University, Fall 2014

Abstract
The Sparse Matrix-Vector multiplication (SpMV) kernel is of central importance in scientific computing, with applications ranging from PDE solvers to social network analysis. Due to random accesses into the source vector, naive implementations of SpMV based on the compressed sparse row (CSR) format have very high memory cost (cache misses, TLB misses, etc.), and in many cases the CPU is only utilized at 10% of its peak operating performance. Prior work has focused on exploiting dense substructures within a sparse matrix via blocking the matrix into non-overlapping r x c subblocks. However, because the block size is static, this method wastes a lot of space/computation. We propose a new storage format and SpMV algorithm that supports dynamic block sizes. Then, we develop a pre-processing technique to choose the proper block size (based on local density) for each non-zero in the matrix. To further improve cache locality, we also develop a pre-processing step to optimize the order in which the blocks-vector products are computed (i.e. reordering a cache block). We compare our approach to an implementation of Sparsity, a leading SpMV optimization framework using static register block sizes and cache blocking. Our dynamic blocking approach is shown to significantly reduce storage space (2.7x on average across test matrices) compared to static blocking. Furthermore, compared to Sparsity, our SpMV compute time is decreased by roughly the same magnitude. Memory profiling shows our methods significantly reduce cache and TLB misses. Moreover, our pre-processing time is within 5x the pre-processing time of Sparsity's, easily amortizable over repeated multiplications.

Introduction and Background
Sparse matrices are amenable to optimizations to the representation and storage of data, but traditional implementations of sparse matrix-vector multiplication have been relatively poor [Williams].

We also see many cases of problems where an offline matrix needs to be multiplied by many vectors throughout the day (e.g. reordering a cache block). We compare our approach to an implementation of Sparsity, a leading SpMV optimization framework using static register block sizes and cache blocking. Our dynamic blocking approach is shown to significantly reduce storage space (2.7x on average across test matrices) compared to static blocking. Furthermore, compared to Sparsity, our SpMV compute time is decreased by roughly the same magnitude. Memory profiling shows our methods significantly reduce cache and TLB misses. Moreover, our pre-processing time is within 5x the pre-processing time of Sparsity's, easily amortizable over repeated multiplications.

Storage
A common storage form for sparse matrices is compressed sparse row (CSR) format.

Reordering Cache Blocks
We cache block in the same manner as Sparsity, but within a cache block, we order the blocks to force successive reads/writes to the same cache lines (if possible).

Dynamic Register Blocking
Based on the size of a system’s register, we can determine r_max and c_max, the maximum dimensions of a block that can fit in register (along with space for the source vector and for writing output). Then, our dynamic register blocking scheme is based on sampling the region around each non-zero, and choosing the largest block size with sufficient density that covers the original non-zero.

- We allowed 1x1, 2x2, 3x3, and 4x4 register blocks and achieved significant compression compared to static blocking.

SpMV Compute Performance
- Average speedup of 2.7x compared to Sparsity
- Reductions in cache misses and TLB misses follow similar patterns to speedup, suggesting that more efficient memory access is causing our performance gains.

Conclusions and Future Work
Future work included demonstrating the compression and speedup by our methods are robust over a large set of sparse matrices and identifying the classes for which they are the most beneficial. Furthermore, the pre-processing heuristics can be improved and further evaluated. Lastly, we have not considered optimizations via parallelism, which these methods are very amenable to.

Bibliography