Memory Hierarchy Optimizations for Sparse Matrix Powers

June 10, 2002

Background

The goal of this project is to apply memory hierarchy performance optimizations to the sparse matrix kernel $A^k x$, where $A$ is a sparse matrix and $x$ is a dense vector. This kernel arises in many important applications in scientific computing (e.g., iterative methods for solving linear systems and computing eigenvalues), information retrieval (e.g., latent semantic indexing), and web searching (e.g., the Google search engine); however, this kernel has not been optimized for general $A$ on modern cache-based architectures.

A na"ıve implementation of $A^k x$ computes the sequence $y_0 = x$, $y_j = Ay_{j-1}$ for all $1 \leq j \leq k$. This computation requires bringing every element of the matrix $A$ through the memory hierarchy $k$ times. Recently, Strout, et al., proposed a cache-optimization known as serial sparse tiling which in principle brings $A$ through the memory hierarchy only once for small $k$. However, they consider what we may view as a particular instance (or class of instances) of $A$.

Their technique can be generalized to all sparse $A$ as shown in the following example.

Consider the computation of $y = A^2 x$ for a $7 \times 7$, tridiagonal matrix. This calculation can be represented symbolically as the graph shown in Figure 1. Each node in the left-most and right-most columns correspond to elements of the $x$ and $y$ vectors, respectively. Nodes in the center column represent the intermediate result $t_1 = Ax$. An edge $u \rightarrow v$ represents multiplication by an element of $A$, and indicates that the result $v$ depends on the result of node $u$.

Observe that elements of $A$ (i.e., edges) are each used twice. The idea behind serial sparse tiling is to partition this symbolic computation into regions such that the same edges fall into a partition. Figure 1 depicts one such partitioning, where members of the same partition are shown in the same color. Notice that both uses of the $(1,1)$, $(1,2)$, $(2,1)$, $(2,2)$, and $(2,3)$ edges are in the same partition. Thus, if all the red edges fit approximately into some level of cache, then they will remain there if we compute the red partition first. Note that partitions must be executed such that they respect the dependencies in order to arrive at the same result as the na"ıve implementation.

1In particular, they consider the $A$ that arises in the iterative method known as Gauss-Seidel for solving linear systems.
Figure 1: Illustration of serial sparse tiling. The figure shows the symbolic computation of \( y = A^2 x \), where \( A \) is a \( 7 \times 7 \) tridiagonal matrix. The top-most row of round nodes corresponds to the elements of the vector \( x \) (i.e., \( x_1, x_2, \ldots, x_7 \)). The center set of round nodes corresponds to a temporary vector \( t = Ax \). The final, bottom-most round nodes correspond to elements of \( y = At = A^2 x \). Arrows indicate multiplication by elements of \( A \). Partitioning this computational dependency structure into groups marked by the same color as shown are tiles, and many elements of \( A \) within a tile are re-used.
Although we expect this technique to yield performance improvements, the performance of sparse matrix kernels, in general, is also strongly affected by the choice of data structure. There are a wide variety of formats for storing sparse matrices (see Saad for a survey [7]), all of which incur some storage and processing overhead in order to know which non-zeros are being stored. In prior work on optimizing sparse matrix-vector multiply, $y = Ax$, Im and Yelick proposed a register-level optimization called register blocking, which identifies and stores naturally occurring dense submatrices more compactly, yielding significant performance improvements [1]: a later analysis showed, by examining hardware counters and applying a model of execution time, that these optimizations yield execution rates [2] that are close an estimated upper-bound [3]. Therefore, we expect that the combination of register blocking and serial sparse tiling could yield significant performance improvements over the conventional implementation.

This work, among other projects, is being conducted by the Berkeley Benchmarking and OPtimization (BeBOP) group.

Task Summary

A very basic implementation in C of serial sparse tiling for general $A$ has already been completed. The remaining tasks are as follows:

1. A prototype of the analysis phase (symbolic computation and partitioning) has been implemented in Python. Due to interpreter and other processing overheads, this implementation cannot perform the symbolic analysis of the larger matrices in our test suite. The first task is to reimplement the analysis phase in C. (For the partitioner, we currently use the METIS package [6].) (Estimated time: approximately 3 weeks.)

2. The code should be instrumented with PAPI [2], a library for reading hardware performance counters, to confirm that we do reduce the number of cache misses as expected, and to determine the performance benefits (if any) of this optimization on a wide class of matrices. (Estimated time: approximately 2 weeks.)

3. Serial sparse tiling is a cache optimization. Combine this method with register blocking and do some basic measurements. (Estimated time: approximately 3 weeks.)

Recommended Reading

Understanding the following papers should be sufficient to get started: Strout, et al. [8], Im and Yelick [7], and the first 10 pages of Saad’s survey [4].

---

2 Measured in millions of floating point operations per second, or Mflop/s.
3 See www.cs.berkeley.edu/~ richie/bebop
References


