Sparse Matrix Computations on Manycore GPU's

Michael Garland
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
mgarland@nvidia.com

ABSTRACT

Modern microprocessors are becoming increasingly parallel devices, and GPUs are at the leading edge of this trend. Designing parallel algorithms for manycore chips like the GPU can present interesting challenges, particularly for computations on sparse data structures. One particularly common example is the collection of sparse matrix solvers and combinatorial graph algorithms that form the core of many physical simulation techniques. Although seemingly irregular, these operations can often be implemented with data parallel operations that map very well to massively parallel processors.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Algorithms

Keywords

parallel programming, data-parallel algorithms, GPU computing, sparse matrix-vector multiplication, shortest path algorithms.

1. INTRODUCTION

Modern microprocessors are increasingly parallel devices. Current commodity CPUs typically provide 2–4 processor cores augmented by vector units. The transition to such multi-core processors has been the dominant trend in processor design in recent years, and has been the main engine driving processor performance. As this trend continues, parallelization of core computations is becoming the only reliable route for substantially improving application performance.

Because of the abundance of parallelism available in graphics workloads, GPUs have been at the leading edge of expanding chip-level parallelism for some time. Current NVIDIA GPUs range in scale from 16-core chips at the low end to 128-core chips at the high end. These GPUs are specifically designed to provide high throughput on parallel computations, with a peak processing rate of approximately 500 GFLOPS and peak bandwidth of just under 80 GB/s.

The processing power of the GPU makes it an attractive platform for expensive parallel tasks. Physical simulations, such as those involving numerical solution of partial differential equations, are important examples of computations in this category. Its high degree of parallelism also makes the GPU an attractive platform for research and development of parallel algorithms, a problem becoming increasingly important as CPUs also become increasingly parallel.

Designing parallel algorithms for manycore chips like the GPU can present interesting challenges, particularly for computations on sparse data structures. Interestingly, while these computations are seemingly irregular, they can often be reformulated in terms of data parallel primitives—operations such as `map`, `scan`, and `reduce`—which have a regular mapping onto massively parallel machines. This paper explores the application of these techniques to sparse matrix and graph operations, with a specific focus on sparse matrix-vector multiplication and shortest path computations in graphs. As we shall see, both can be directly mapped onto regular data parallel algorithms, and thus can be mapped to a manycore processor like the GPU in a natural way.

2. GPU COMPUTING WITH CUDA

A decade ago, GPUs were largely fixed-function devices focused on accelerating the rendering pipelines specified by the OpenGL and DirectX interfaces. In contrast, NVIDIA's current GPUs, based on the Tesla unified graphics and computing architecture [8], feature a fully programmable parallel processor array. In graphics applications, these processors execute the shader programs that have become the main building blocks for most rendering algorithms. With NVIDIA's CUDA software environment, developers now also have the means to program these parallel processors directly.

The CUDA programming model [10, 11] extends standard C/C++ with a minimalist set of parallel programming abstractions, namely a hierarchy of threads, shared memories, and barrier synchronization. A CUDA program is organized into a sequential *host* program and one or more parallel *kernels* that can be executed by the host program on a parallel *device*. Typically, the host program executes on the CPU and the parallel kernels execute on the GPU, although CUDA kernels may also be compiled for efficient execution on multi-core CPUs [13].

A kernel executes a scalar sequential program across a set of parallel threads. The programmer organizes these threads into *thread blocks*; a kernel thus consists of a *grid* of one or more blocks. A thread block is a group of concurrent threads that can cooper-

DAC 2008, June 8–13, 2008, Anaheim, California, USA

```
// Serial loop for computing y ← αx+y
void saxpy(uint n, float a, float *x, float *y)
{
    for(uint i = 0; i<n; ++i)
        y[i] = a*x[i] + y[i];
}


void serial_sample()
{
    // Call serial SAXPY function
    saxpy(n, 2.0, x, y);
}
```

```
// Parallel kernel for computing y ← αx+y
__global__
void saxpy(uint n, float a, float *x, float *y)
{
    uint i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i<n )  y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
    // Launch parallel SAXPY kernel using
    // ⌈n/256⌉ blocks of 256 threads each
    saxpy<<<ceil(n/256), 256>>>(n, 2.0, x, y);
}
```

Figure 1: Simple example of parallelizing $y \leftarrow \alpha x + y$ across vectors of length n using CUDA, assigning 1 output element per thread.

ate amongst themselves through barrier synchronization and a per-block shared memory space private to that block. Each thread is given a unique integer index within its block—via the `threadIdx` identifier—and each block is given a unique integer index—via `blockIdx`. When launching a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when lauching the kernel. These dimensions are available to the kernel via the identifiers `gridDim` and `blockDim`, respectively. For convenience, thread blocks and grids may have 1, 2, or 3 dimensions. The program is free to choose the dimensions that best fit the data being processed and the algorithm being used, for every kernel that is launched.

Each CUDA thread has access to three levels of a memory space hierarchy. An individual thread has its own per-thread local memory, for storing its local variables. On the GPU, these variables typically reside in live registers. Threads may also declare variables in a per-block shared memory using the `__shared__` type qualifier. Finally, the external GPU DRAM is visible to all threads of a kernel; variables are placed in this memory space with the `__device__` qualifier.

It is important that kernels scale well across chip families with significantly different levels of physical parallelism. To achieve transparent kernel scalability, the CUDA programming model requires that separate blocks of a kernel be *independent*, meaning that the program should be valid under any possible interleaving of block executions. In particular, it should be possible to execute blocks in any order, without pre-emption, either in parallel or sequentially, without effecting the correctness of the kernel.

Figure 1 provides a very simple example of parallel programming in CUDA: a straightforward implementation of the SAXPY routine defined by the BLAS linear algebra library. Given vectors x and y containing n floating point numbers, it performs the update $y \leftarrow \alpha x + y$. The serial implementation is a simple loop over each element of y in turn. The parallel kernel assigns a separate thread to each element of y. The `__global__` declaration specifier indicates that the procedure is a kernel entry point, and the extended function call syntax `saxpy<<<B, T>>>(...)` is used to launch the kernel `saxpy` in parallel across `B` blocks of `T` threads each. This example demonstrates a common parallelization pattern, where a serial loop with independent iterations can be executed in parallel across many threads.

2.1 GPU Computing Architecture

The Tesla architecture [8] is built around an array of SM multithreaded multiprocessors. A single SM is capable of executing up to 768 simultaneous concurrent threads. Thread management is performed entirely in hardware, providing zero scheduling overhead and negligible creation overhead. Synchronization amongst threads in a block is also extremely lightweight; the CUDA barrier intrinsic maps to a single instruction.

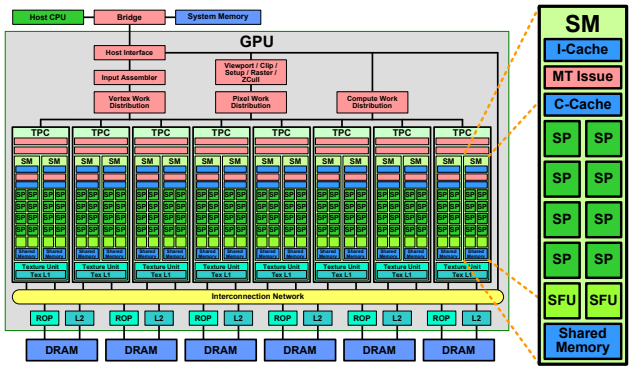
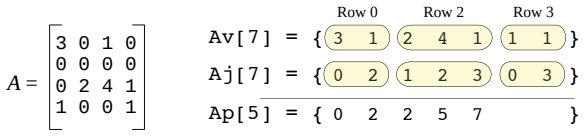


Figure 2: Diagram of a Tesla GPU with 128 SP cores arranged in 16 SM's interconnected with 6 DRAM memory partitions.

To efficiently manage its large thread population, the Tesla SM employs a SIMT (single instruction, multiple thread) architecture [8, 10]. Threads are executed in groups of 32 called *warps*. The threads of a warp are executed on separate scalar (SP) processors which share a single instruction unit. The SM transparently manages any divergence in the execution of threads in a warp. This SIMT architecture allows the hardware to achieve substantial efficiencies while executing non-divergent data parallel codes, which is the common case in many massively parallel computations. At the same time, the SIMT execution of threads is relatively transparent to the programmer. Much like cache line sizes on traditional CPUs, it can be ignored when designing for correctness, but must be carefully considered when designing for peak performance.

Each SM is equipped with a 16KB on-chip scratchpad memory that provides the CUDA per-block shared memory space. This shared memory has very low access latency and high bandwidth, similar to an L1 cache. Along with the SM's lightweight barriers, this memory is an essential ingredient for efficient cooperation and communication amongst threads in a block.

In contrast to earlier generations of GPUs, programs executing on the SM have access to the full range of general purpose instruc-



(a) Sample matrix A (b) CSR representation of matrix

Figure 3: Example of Compressed Sparse Row representation.

tions one would expect in a microprocessor. Memory is accessed through load/store instructions supporting arbitrary address arithmetic. Both floating point and integer data types are fully supported. Threads may even branch at will, although there will be a performance penalty if threads in a warp branch divergently.

3. SPARSE MATRIX-VECTOR PRODUCT

A wide variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. Sparse matrix-vector multiplication is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss below, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient method [6] straightforward.

A sparse $n \times n$ matrix is one in which the number of non-zero entries m is only a small fraction of the total. Sparse matrix representations seek to store only the non-zero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only $m = O(n)$ non-zero elements, this represents a substantial savings in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the Compressed Sparse Row (CSR) representation. The m non-zero elements of the matrix A are stored in row-major order in an array `Av`. A second array `Aj` records the corresponding column index for each entry of `Av`. Finally, an array `Ap` of $n+1$ elements records the extent of each row in the previous arrays; the entries for row i in `Aj` and `Av` extend from index `Ap[i]` up to, but not including, index `Ap[i+1]`. This implies that `Ap[0]` will always be 0 and `Ap[n]` will always be m, the number of non-zero elements in the matrix. Figure 3 shows an example of the CSR representation of a simple matrix.

Given a matrix A in CSR form—represented by the arrays `Av`, `Aj`, and `Ap`—we want to compute the product $y = Ax$, where x and y are both dense column vectors of length n. A simple procedure for performing this multiplication is shown in Figure 4. The `multiply_row()` procedure is used to compute a single row y_i of the output vector. Computing the full product Ax is then simply a matter of looping over all rows, applying `multiply_row()` to each in turn, as done in `csrmul_serial()`.

3.1 Simple Loop Parallelization

The simple serial algorithm shown in Figure 4 can be translated into a parallel CUDA kernel quite easily. Since each iteration of the loop over `row` in `csrmul_serial` is independent of all the others, we can simply spread the loop over many parallel threads.

Figure 5 shows the resulting CUDA kernel. Notice that it looks nearly identical to the serial procedure in Figure 4. This is a fairly common pattern. The serial implementations of many naturally parallel tasks are structured around loops with independent iterations. Breaking up such loops and spreading them across parallel

```
float multiply_row(uint rowsize, const uint *Aj,
                   const float *Av, const float *x)
{
    float sum = 0;

    for(uint column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}

void csrmul_serial(const uint *Ap, const uint *Aj,
                   const float *Av,uint num_rows,
                   const float *x, float *y)
{
    for(uint row=0; row<num_rows; ++row)
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        // Compute the row for this thread
        y[row] = multiply_row(row_end-row_begin,
                              Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

Figure 4: Basic serial code for computing $y = Ax$.

threads is a common approach in many parallel programming systems, including CUDA, OpenMP, and Intel's Threading Building Blocks.

Both the serial and parallel procedures we have seen so far are very simple implementations. There are numerous possible avenues for optimizing their performance. Williams *et al.* [14] explore a number of such optimizations, using an autotuning framework to uncover the best mix of optimizations for various multicore architectures.

On the GPU, two kinds of optimizations are particularly worthwhile. First, the kernel in Figure 5 can be augmented to use the GPU's on-chip shared memory as a cache for elements of x. Second, the shared memory can also be used to allow the threads of a block to load data from memory in contiguous chunks, subsequently changing their access pattern when reading from shared memory. These two optimizations alone improve the running time

```
__global__
void csrmul_kernel(const uint *Ap, const uint *Aj,
                   const float *Av,uint num_rows,
                   const float *x, float *y)
{
    uint row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin,
                              Aj+row_begin,
                              Av+row_begin, x);
    }
}
```

Figure 5: Parallelizing $y = Ax$ with 1 thread per row.

of the kernel by roughly a factor of 2.

Such optimizations aside, the primary weakness of the parallel kernel shown above is that it assigns one complete row to each thread. This is fairly reasonable as long as each row of the matrix has a fairly similar number of non-zeros. In many important applications, we do expect this to be true. For instance, Laplacian matrices derived from the connectivity of triangulated meshes will have an average of 7 non-zeros per row and very low variance from the mean. There are other classes of matrices, however, for which this static assignment of 1 row per thread is a disastrous choice. Power-law graphs [2, 9], whose vertex degree distributions follow a power law, are likely to have at least a handful of rows with $O(n)$ non-zeros while having many that will have $O(1)$ non-zeros. Processing matrices with such an irregular vertex degree distribution with a kernel like that shown in Figure 5 will lead to very poor performance due to a gross imbalance of work between threads. Fortunately, by taking a fairly different view of the problem, we can remove the possibility for load imbalance entirely.

3.2 Data Parallel Approach

Sparse matrix-vector multiplication can also be implemented in terms of a *segmented scan* operation. Scan operations are one of the central data parallel primitives [7, 3]. The key advantage of this approach to sparse matrix-vector multiplication is that the running time of the multiplication will be *independent* of the distribution of non-zeros across rows.

Given a sequence a of length n and a binary associative operator $\oplus$, the *scan* operator produces the output sequence:

$$\text{scan}(a,\oplus) = \left[a_0, \quad a_0 \oplus a_1, \quad \ldots, \quad a_0 \oplus \cdots \oplus a_{n-1}\right]$$

This is an *inclusive* scan operator, in the sense that element i of the output sequence includes the input value a_i. As a concrete example, here is the effect of the scan operator applied to a sequence of integers and taking the operator $\oplus$ to be integer addition:

$$a = \begin{bmatrix}3 & 1 & 2 & 4 & 1 & 1 & 1\end{bmatrix}$$
$$\text{scan}(a,+) = \begin{bmatrix}3 & 4 & 6 & 10 & 11 & 12 & 13\end{bmatrix}$$

In this case of choosing an addition operator, the scan operation is equivalent to a prefix sum over the input sequence.

More generally, we would like to allow the input sequence to be composed of multiple sub-sequences. Performing a *segmented scan* on such a sequence would be equivalent to performing a scan on each subsequence in isolation. For example:

$$a = \begin{bmatrix}[3 & 1] & [2 & 4 & 1] & [1 & 1]\end{bmatrix}$$
$$\text{segscan}(a,+) = \begin{bmatrix}[3 & 4] & [2 & 6 & 7] & [1 & 2]\end{bmatrix}$$

To implement an efficient parallel segmented scan, we would like to represent a composite sequence such as the one above as the combination of (1) a flat sequence of values and (2) a segment descriptor that indicates where the subsequence boundaries are. There are several possible representations of the segment descriptor. One would be to create a sequence storing the first index of all subsequences in order, which would be equivalent to the CSR matrix representation. Another representation, and one which is generally a bit more convenient for massively parallel machines, is a so-called *head flags* array which records a 1 for elements that begin a subsequence and a 0 for all other elements. The following is the head-flags representation of the segmented sequence shown above:

$$a.\text{values} = \begin{bmatrix}3 & 1 & 2 & 4 & 1 & 1 & 1\end{bmatrix}$$
$$a.\text{flags} = \begin{bmatrix}1 & 0 & 1 & 0 & 0 & 1 & 0\end{bmatrix}$$

This head flag sequence would typically be packed to use only 1 bit per element. For sparse matrix representations, it may be represented implicitly. If we extend the CSR representation to record the row index array `Ai` for all non-zeroes, segment boundaries can be determined by comparing row indices of adjacent elements.

The scan-based algorithm for sparse matrix-vector multiplication will look like the following:

```
// Gather appropriate x_j for each non-zero element.
xs = gather(x, Aj);

// Compute product A_ij x_j for each non-zero element.
products = map<multiply>(Av, xs);

// Perform segmented scan, using totals() to keep
// the last (total) value in each subsequence.
y = totals(segscan<plus>(products, Ai));
```

One could implement this directly, using one CUDA kernel for each of the data-parallel operations `gather`, `map`, and `totals`. Implementing `segscan` actually requires two kernels, but is still reasonably straightforward. It is more efficient, however, to fuse these operations together into as few kernels as possible, avoiding unnecessary traffic to memory. The GPU's on-chip shared memory is particularly helpful in accomplishing this. For instance, the `gather`, `map`, and the initial phase of the `segscan` operations can all be fused into a single kernel. Sengupta *et al.* [12] provide the details of implementing segmented scan efficiently in CUDA and explore its application to sparse matrix-vector multiplication. Efficient scan procedures are publicly available in the CUDPP [5] library of data parallel primitives.

Implementing sparse matrix-vector multiplication using segmented scan is attractive because the running time does not depend on the distribution of non-zeros across the rows. It is, therefore, the best choice for the most general case where nothing is known about the structure of the matrix being used. As always with sparse routines, applications that have specific knowledge about the structure of the matrices they will encounter will generally benefit from using kernels customized to that structure. For example, if the matrices being used are known to be tridiagonal, a tridiagonal-specific multiplication kernel should obviously be used. For matrices with very small and regular rows, optimized forms of the kernels shown in Section 3.1 may provide the best performance.

4. SHORTEST PATHS ON GRAPHS

Aside from forming the core of most numerical solvers, sparse matrix operations can also provide the substrate on which to build efficient graph algorithms. Building graph algorithms from efficient sparse matrix operations, rather than developing entirely separate implementations, is an attractive design approach. Many of the core computations are fundamentally the same. There is also much to be gained by focusing on parallelizing and optimizing core operations, like `scan`, from which higher-level algorithms can be built, rather than trying to parallelize and optimize each new algorithm in turn.

As one example, consider finding shortest paths in a graph. Suppose that we are given a graph $G = (V,E)$ whose edges are labelled by distances e_{ij}, giving the length of the edge $j \to i$. Given a set of one or more source vertices $S \subset V$, we want to compute the distance d_i for each vertex $i \in V$ to the nearest source vertex in S. This is a generalization of the typical single-source shortest path problem, where S would contain precisely one source vertex. However, generalizing to multiple sources requires only a trivial change to the algorithm, and is more useful for certain applications.

Dijkstra's algorithm [4] is perhaps the most common solution

Figure 6: Triangle mesh partitioned in CUDA using Lloyd relaxation. Colored patches show the partition of the vertex set.

to this problem. However, it is rather inconvenient in a massively parallel regime, as it relies on a global priority queue to visit edges in a specific order. A global queue imposes a single serialization point across all threads, which we would very much like to avoid.

The Bellman-Ford algorithm [4], in contrast, is quite easy to parallelize. It begins by initializing the distance vector d as follows:

$$d_i = \begin{cases} 0 & \text{if } i \in S, \\ \infty & \text{otherwise.} \end{cases} \quad (1)$$

The algorithm then iteratively *relaxes* each vertex by the rule

$$d_i \leftarrow \min_{j \in N_i}(d_j + e_{ij})$$

until no value of d_i is changed by further relaxation. This will require, at worst, $O(n)$ iterations over the vertex set, although in practice the number of iterations will be far lower, especially for graphs that can be embedded on a 2-manifold.

Notice that this relaxation step is structurally identical to matrix multiplication, but with the operators $(+,\times)$ replaced by $(\min,+)$. The Bellman-Ford algorithm is thus equivalent to iterated multiplication $d \leftarrow Ed$, where E is the matrix of edge lengths e_{ij}. Consequently, we can build an efficient Bellman-Ford implementation from an efficient sparse matrix-vector kernel simply by templatizing the matrix-vector kernel to accept operations other than the usual $(+,\times)$.

Figure 6 shows an example of using this technique in practice. The application is designed to partition a triangulated surface mesh using Lloyd relaxation [1]. The colored patches indicate the partitioning of the vertex set. This iterative algorithm attempts to minimize the sum of squared distances from all vertices to the centroid of their respective patches. It is thus a form of k-means clustering. Lloyd relaxation on a graph largely consists of repeatedly running shortest path computations where the source set S contains one vertex per patch, representing the patch centroids. The parallel partitioning code used to generate the result in Figure 6 is built directly on a sparse matrix-vector style Bellman-Ford implementation.

5. CONCLUSION

Designing algorithms for massively parallel computers presents a number of interesting challenges, and today's manycore GPUs provide an attractive platform for designing and testing algorithmic solutions. Moreover, the tremendous peak throughput of these chips make them a powerful deployment platform for parallel computations.

This paper has explored some of the issues involved with designing parallel algorithms for sparse matrix-vector multiplication and, by extension, shortest path computations on combinatorial graphs. Data parallel primitives, specifically `scan`, allow us to convert this seemingly irregular computation into a regular one that maps well onto massively parallel hardware. Furthermore, supporting generic operators in such kernels allows us to accomplish other tasks, such as shortest-path calculations, with the same kernels.

More generally, programming massively parallel processors becomes substantially easier if we can identify a few core building blocks (e.g., `scan`) from which we can synthesize higher level procedures (e.g., sparse matrix-vector multiplication). Identifying such building blocks and understanding how to implement them most efficiently on various architectures will be a key research challenge over the coming years.

6. REFERENCES

[1] P. Alliez, G. Ucelli, C. Gotsman, and M. Attene. Recent advances in remeshing of surfaces. In L. D. Floriani and M. Spagnulo, editors, *Shape Analysis and Structuring*, pages 53–82. Springer, 2007.

[2] A.-L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, 288:60–69, 2003.

[3] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Second edition, Sept. 2001.

[5] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`.

[6] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, Third edition, 1996.

[7] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), Mar/Apr 2008.

[9] M. E. J. Newman. The structure and function of complex networks. In *SIAM Review*, volume 45, pages 167–256, 2003.

[10] J. Nickolls, I. Buck, K. Skadron, and M. Garland. CUDA: Scalable parallel programming. *ACM Queue*, Mar/Apr 2008.

[11] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, Nov. 2007. Version 1.1.

[12] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.

[13] J. A. Stratton, S. S. Stone, and W. W. Hwu. M-CUDA: An efficient implementation of CUDA kernels on multi-cores. IMPACT Technical Report IMPACT-08-01, UIUC, Feb. 2008.

[14] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. Supercomputing 2007*, Nov. 2007.