# Performance Evaluation of the NVIDIA GeForce 8800 GTX GPU for Machine Learning

Ahmed El Zein[1], Eric McCreath[1], Alistair Rendell[1], and Alex Smola[2]

[1] Dept. of Computer Science, Australian National University, Canberra, Australia
[2] Statistical Machine Learning Program, NICTA, Canberra, Australia
{Ahmed.ElZein,Eric.McCreath,Alistair.Rendell,Alex.Smola}@anu.edu.au

**Abstract.** NVIDIA have released a new platform (CUDA) for general purpose computing on their graphical processing units (GPU). This paper evaluates use of this platform for statistical machine learning applications. The transfer rates to and from the GPU are measured, as is the performance of matrix vector operations on the GPU. An implementation of a sparse matrix vector product on the GPU is outlined and evaluated. Performance comparisons are made with the host processor.

## 1 Introduction

The GeForce 8800 GPU is the first GPU from NVIDIA to implement a unified architecture where the pixels and vertices are processed by the same hardware. This provides a higher degree of programmability than for previous GPUs and is much better suited to general purpose computing. In recognition of this, NVIDIA have released a general purpose programming interface called CUDA (see section 2.3 for details) and have packaged the same basic hardware as a dedicated co-processor for use by high performance computing applications (the Tesla product range). Moreover, NVIDIA have also announced [1] that future generations of their hardware will provide support for IEEE double precision arithmetic; a move that will arguably remove the one remaining major bottleneck to the widespread use of GPUs in scientific computations.

While the CUDA programming interface significantly eases use of the NVIDIA GPUs for general purpose programming, the programming model provided by CUDA is very different to that available on a traditional CPU. For instance CUDA has the concepts of shared, constant, texture, and global memories that all have slightly different properties, and determining how best to use each memory type for a given application is non-trivial. Also, it must be remembered that when using any coprocessor the observed performance will depend heavily on what fraction of the application can be run on the coprocessor, and whether the overheads introduced in order to move data to and from the coprocessor are small compared to the computational times involved.

In this paper we outline our initial efforts to migrate a Statistical Machine Learning (ML) application to the GeForce 8800 GPU. The kernel of this application involves an iterative solver that performs repeated matrix vector products.

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 466–475, 2008.

For a number of reasons this application would appear to be well suited to use of a GPU. First, matrix operations are generally well suited to vector or stream processors such as the GeForce 8800. Second, matrix vector products scale as $O(N \cdot d)$, where $N$ is the number of data points and $d$ is the inherent dimensionality of the problem, whereas other steps of a ML application typically scale as $O(N)$. Consequently for high-dimensional problems, migrating this part of the application to the GPU is potentially beneficial. Third, the matrix does not change between iterations, so it can be copied once to memory on the GPU and reused during each iteration. Finally, for many ML problems single precision arithmetic is sufficient, so the porting effort required is of immediate benefit even before double precision GPUs become available.

On closer inspection the situation is not quite as simple. In particular, although matrix operations can be easily vectorized, the amount of data may exceed what a single GPU card can hold (there is 768 MB on the GPU used). Consequently the resulting performance depends heavily on the bandwidth of the bus connecting the CPU and the GPU. Secondly, for matrix-vector multiplications, the limiting factor is the memory bandwidth rather than the raw floating point performance (the latter exceeds the former on both CPU and GPU). This ratio is generally less favourable for GPUs than the ratio between GPU and CPU peak floating point ratios. Finally, many ML problems involve sparse matrices, so the use of a sparse matrix vector product may be preferable to use of the dense equivalent. Sparse matrix algorithms are, however, considerably harder to adapt to stream processors.

With the goal of migrating the complete ML application to CUDA this paper addresses three issues: i) What transfer rates can be achieved between host and GPU memory and vice versa, ii) What performance is achieved when using the CUDA supplied BLAS library to perform a variety of dense matrix vector products of sizes similar to those required by ML applications, iii) How does the performance of the latter compare with what we can obtain by hand coding sparse matrix vector products in CUDA.

The following section gives background information about the ML application, the NVIDIA 8800 GPU hardware, its CUDA programming model, and methods for sparse matrix vector products. Section 3 details our experimental setup, while Section 4 contains detailed performance results. Section 5 uses the performance data gather here to discusses how a full ML application is likely to perform on the GeForce 8800 and outlines plans for our future work.

## 2 Background

### 2.1 The ML Application

One of the key objectives in ML is, given some patterns $x_i$, such as pictures of apples and oranges, and corresponding labels $y_i$, such as the information whether $x_i$ *is* an apple or an orange, to find some function $f$ which allows us to estimate $y$ from $x$ automatically. See e.g. [2] for an introduction. In this quest, convex optimization is a key enabling technology for many problems. For

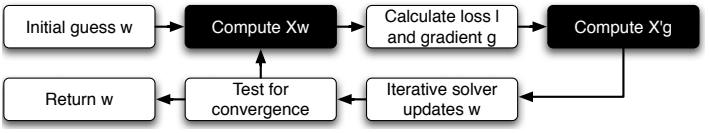


**Fig. 1.** Iterative solver algorithm. The black boxes refer to matrix-vector operations which could be accelerated by a GPU.

**Table 1.** Statistics for some typical ML datasets [3]

| Domain | Dataset | Rows | Columns | Nonzero Elements | Density |
|---|---|---|---|---|---|
| Intrusion Detection | KDDCup99 | 3,398,431 | 127 | 55,503,855 | 12.86% |
| Ranking | NetFlix | 480,189 | 17,770 | 100,480,507 | 1.17% |
| Text Categorization | Reuters C11 | 804,414 | 47,236 | 60,795,680 | 0.16% |
| Text Categorization | Arxiv astro-ph | 62,369 | 99,757 | 4,977,395 | 0.08% |

instance, Teo et al. [3] proposed a scalable convex solver for such problems. It is an iterative algorithm that involves guessing a solution vector $w$, using this to evaluate a loss function $l(x, y, w)$ and its derivative $g = \partial_w l(x, y, w)$, and then updating $w$ accordingly. This process is repeated until a desired level of convergence is achieved (see Fig. 1). As mentioned above the majority of time is spent evaluating the matrix vector products, and the elements of matrix $(X)$ do not change between iterations.

Many ML datasets are very sparse, as shown in Table 1. Exploiting the sparsity decreases the memory footprint of the matrix as well as the the number of floating point operations required for the matrix vector product. Unfortunately it also introduces random memory access patterns and indirect addressing, which is likely to result in less efficient utilization of a GPU's hardware.

### 2.2 NVIDIA 8800 GTX Hardware

Figure 2 illustrates the architecture of the GeForce 8800 GTX used in this work. At the heart of the device lies the Streaming Processor Array (SPA) consisting of 8 Texture Processor Cluster (TPC) units. Each TPC contains 2 Streaming Multiprocessor (SM) units and a texture unit. The SM in turn consists of 8 Stream Processors (SP) clocked at a default of 1.35 GHz. When running CUDA applications each SP is able to issue one multiply-add (MAD) instruction per cycle. This gives each SM a peak performance of 21.6 GFLOPS, and the GeForce 8800 GTX with 16 SMs an aggregate performance of 345.6 GFLOPS.

The SPA is connected to 768 MB of GDDR3 memory through a 384-bit (48 byte) wide interface. Clocked at 900 MHz (1800 MHz effective double data rate) by default, the frame buffer memory has a peak bandwidth of 84.375 GB/s. More details of the NVIDIA hardware can be found in [4].

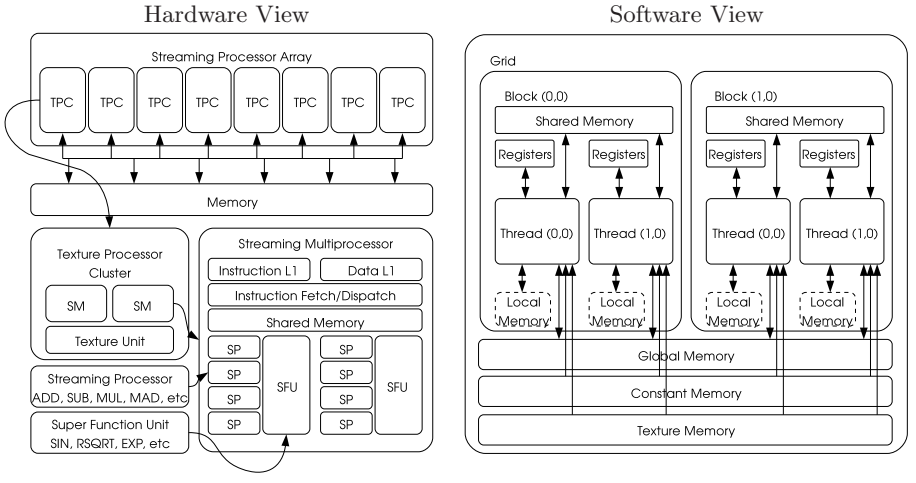


**Fig. 2.** GeForce 8800 GTX architecture and CUDA memory model

### 2.3 CUDA

The Compute Unified Device Architecture (CUDA), is a hardware and software architecture that enables the issuing and managing of computations on the GPU as a data-parallel device without the need to map the computations to a graphics API. CUDA transforms the hardware's personality from a graphics card to a multi-threaded coprocessor. Provided with CUDA are Basic Linear Algebra Subprograms (BLAS) and Fast Fourier Transform (FFT) implementations, however NVIDIA only provides a C/C++ API for these.

CUDA executes that part of the application that runs on the GPU using hundreds or thousands of threads. These threads are organized into a grid of blocks. The grid can be either one or two dimensional, while each block can be a one, two or three dimensional group of threads. The grid and block dimensions can be set at runtime with each thread able to retrieve its own thread and block id. Each block of threads is executed on one physical SM, with NVIDIA hardware only allowing synchronization and access to fast shared memory for threads in the same block. An illustration of the CUDA memory model is given in Fig. 2. A programming guide [1] providing additional information is available from NVIDIA (`http://www.nvidia.com`).

### 2.4 Sparse Matrices on GPUs

A popular representation for sparse matrices is compressed sparse row (CSR) [5] storage. Non-zero elements are arranged into a dense vector *val*. For each value in *val*, its column index from the original matrix is stored in a dense vector of the same size *ind* at the same offset. A third pointer array (*ptr*) carries the offset of the first element in every row. CSR storage and associated pseudo code for a sparse matrix vector product are shown in Fig. 3.

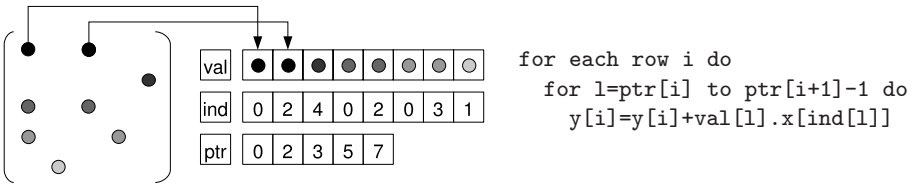


**Fig. 3.** CSR format and pseudo-code for matrix vector product

Sparse matrix vector products (SpMV) have been implemented on older GPU hardware [6,7,8], but were limited by the graphics API and hardware constraints; Bolz et al. [6] and Krüger et al. [8] achieved 9 and 110 MFLOPS respectively in 2003. Ujaldon et al. [9] achieved 222 MFLOPS in 2005 and recently Sengupta et al. [10] achived 215 MFLOPS with CUDA on a GeForce 8800 GTX in 2007.

## 3 Experimental Setup

The GeForce 8800 GTX was hosted in a 2 GHz dual core AMD Athlon64 3800+ system with 2GB of PC3200 DDR memory. The processor has 128KB of L1 cache and 1 MB L2 cache and it has a theoretical peak performance of 8 GFLOPS.

For all benchmarks and experiments, the code was run for 100 iterations and the average time was used to calculate bandwidth and FLOPS. Results for the GPU include the time required to transfer the vector to the GPU and the resulting product vector back to the host. For the sparse matrix vector product FLOPS were calculated as $(2 \times nonzero\ elements \div time)$.

For dense matrix vector products the CUDA BLAS library (CUBLAS) was used on the GPU, while ATLAS[1] was used on the host. Both specialist matrix vector routines (SGEMV) and general matrix matrix routines (SGEMM) were considered, although for ATLAS SGEMV always outperformed SGEMM, since SGEMM is optimized for matrix-matrix multiplications, and thus results for ATLAS SGEMM will not be given. ATLAS permits the matrix to be given in either row or column major format, while CUDA only supports matrices in column major format. Results are given for both normal (N) and transpose (T) ordering of the matrix as both these are required (see Fig. 1).

As yet CUBLAS doesn't support sparse matrices, so our own sparse matrix vector implementation was written (described later). Sparse test matrices were generated using the following code with the condition that each row contains at least one non-zero element.

```
s: chosen sparsity (0% to 99%)
for each row i do
    for each column j do
        if s <= random(0,99) do matrix[i][j] = 1.0
```

[1] Automatically Tuned Linear Algebra Software, http://math-atlas.sourceforge.net

**Table 2.** Host initiated memory transfer rates (GB/s)

| | Latency $\mu s$ | 1KB | 1MB | 100MB |
|---|---|---|---|---|
| Main Memory to GPU | 22 | 0.03 | 0.80 | 1.10 |
| Main Memory (pinned) to GPU | 18 | 0.04 | 2.70 | 3.10 |
| GPU to Main Memory | 18 | 0.04 | 0.40 | 0.50 |
| GPU to Main Memory (pinned) | 15 | 0.05 | 2.80 | 3.00 |
| GPU Memory to GPU Memory | 12 | 0.14 | 50.59 | 71.17 |

# 4 Results

## 4.1 Memory Transfer Rates

Rates for various memory transfer operations are given in Table 2. All transfers are initiated by a CUDA call on the host, with the time recorded from before this call until after the transfer was complete. Hence all benchmarks involve communication over the PCIe bus which has a maximum bandwidth of 4 GB/s. For host to GPU transfers with large data sizes only ∼25% of the PCIe bandwidth is achieved. CUDA, however, allows for the allocation of non-pageable pinned memory on the host, and when this is used approximately 75% of the peak PCIe rate is achieved. When using unpinned memory transfer rates from the GPU to main memory are significantly less than from main memory to GPU, but these become roughly equivalent when using pinned memory. All transfers were found to have a latency of $\sim 20\mu$s, probably reflecting the latency of the PCIe bus. The bandwidth for transferring data from GPU memory to GPU memory was also measured and found to have an asymptotic value close to the 84.4 GB/s peak, with nearly 60% of this achieved for a 1 MB transfer.

## 4.2 Dense Matrix Vector Performance

Using CUBLAS matrix dimensions that were not a multiple of 16 were found to have significantly lower performance; since for ML applications padding can be done once, only results for matrices that are a multiple of 16 will be reported here. Performance data for square matrices of ascending sizes are given in Fig. 4. These show that on the host system performance is roughly constant for all matrix sizes and that normal ordering significantly out performs transpose ordering. On the GPU performance is much more varied. For normal ordering SGEMV performance increases dramatically as the dimension increases, but for transpose ordering it is roughly constant. Thus, while transpose SGEMV ordering is over twice as fast as normal ordering when N=1024, by the time N=5120 it is 30% slower. In almost all cases use of SGEMM instead of SGEMV is found to be slower. At best use of the GPU is $\sim 4.5\times$ faster than use of the host processor.

To observe the effect of matrix shape on performance the total size of the matrix was set to ∼100 MB ($5120 \times 5120$ or 26,214,400 elements) while the number of rows and columns was varied. The results, given in Fig. 5, show a degradation in performance when the number of columns exceeds the number

| Dimension | Host | | GPU | | | |
|---|---|---|---|---|---|---|
| | RowMajor | | ColMajor | | | |
| | SGEMV | | SGEMV | | SGEMM | |
| | N | T | N | T | N | T |
| 1024 | 2.7 | 1.2 | 3.6 | 7.8 | 6.9 | 6.5 |
| 2048 | 2.9 | 1.2 | 7.2 | 9.2 | 6.8 | 7.1 |
| 2816 | 2.9 | 1.1 | 9.5 | 9.0 | 6.4 | 7.0 |
| 3200 | 2.9 | 1.1 | 10.6 | 10.0 | 6.2 | 6.9 |
| 4480 | 3.0 | 1.1 | 13.0 | 8.7 | 6.8 | 7.6 |
| 5120 | 3.0 | 1.2 | 13.6 | 9.9 | 7.0 | 7.8 |

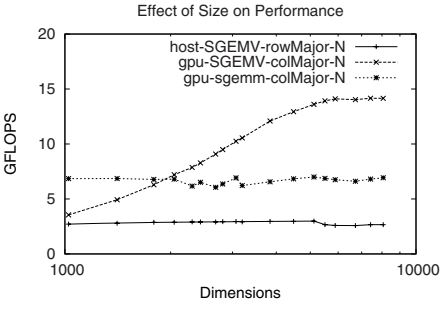


**Fig. 4.** Performance (GFLOPS) for square matrix vector products

| Columns | Rows | Host | | GPU | | | |
|---|---|---|---|---|---|---|---|
| | | RowMajor | | ColMajor | | | |
| | | SGEMV | | SGEMV | | SGEMM | |
| | | N | T | N | T | N | T |
| 204800 | 128 | 1.7 | 0.5 | 12.5 | 9.3 | 6.0 | 6.8 |
| 51200 | 512 | 2.5 | 1.1 | 13.4 | 9.9 | 6.7 | 7.6 |
| 25600 | 1024 | 2.8 | 1.1 | 12.8 | 9.7 | 6.9 | 7.8 |
| 10240 | 2560 | 2.9 | 1.2 | 12.0 | 10.2 | 7.0 | 7.8 |
| 5120 | 5120 | 3.0 | 1.2 | 13.6 | 9.9 | 7.0 | 7.8 |
| 2560 | 10240 | 2.7 | 1.1 | 8.9 | 8.3 | 7.0 | 7.9 |
| 1024 | 25600 | 2.7 | 1.2 | 3.9 | 10.1 | 7.1 | 7.7 |
| 512 | 51200 | 2.7 | 1.2 | 2.0 | 5.3 | 7.1 | 7.9 |
| 128 | 204800 | 2.7 | 0.9 | 0.5 | 1.3 | 1.8 | 2.0 |

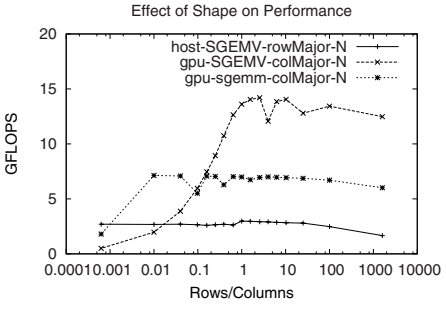


**Fig. 5.** Performance (GFLOPS) as function of shape for matrix vector product

of rows, particularly when using transpose ordering. Although a similar effect occurs when using SGEMM this only happens when the difference is 2 orders of magnitude. In summary the results given here suggest that there is further scope to optimise the performance of the SGEMV routine in CUBLAS.

### 4.3 Initial Sparse Matrix Vector Implementation

The approach taken here stores the matrix in CSR and is parallelized by assigning rows to threads such that each thread multiplies all the elements in a given row by the corresponding elements in the vector before writing the sum to the relevant element in the result vector. The following issues were considered:

**Memory Loads:** While CUDA only supports scalar operations, it also supports upto 128-bit wide vector data types. Loading a float or a float4 from memory costs the same. For a $3000 \times 3000$ matrix at 95% sparsity and 32 threads/block, an SpMV took 827 and 442 $\mu s$ when using float and float4 data types respectively.

**Use of Different Memory Types:** The GeForce 8800 GPU offers different types of memory (Fig. 2). The key to optimizing the SpMV code on the GPU is determining the most efficient use of each memory type.

- ▷ Global Memory. On our card there was 768 MB of global memory. This memory is not cached, but can be read and written to by any thread.
- ▷ Shared Memory. Each SM has 16KB of shared memory that threads in the same block can use to share data. We do not currently use this.
- ▷ Constant Memory. There is 64KB of cacheable read only memory that is initialized each time a GPU kernel is started. Storing the vector in constant memory limits vector sizes to $\sim$ 16000 but further reduces the SpMV duration to 242 $\mu s$ for a $3000 \times 3000$ matrix at 95% sparsity and 32 threads/block.
- ▷ Texture References. CUDA allows binding of global memory to a texture reference. Our initial results suggest this may be useful for the matrix, but only with large row dimensions. Results given here do not use texture references.

The performance of the SpMV implementation for square matrix vector products with a variety of different numbers of threads per block is given in Figure 6 for matrices with 75% and 95% sparsity. The results show significant variation in performance as a function of number of threads per block, particularly at 95% sparsity. The optimal number of threads per block changes with the size of the matrix, and although there is a general trend suggesting more threads per block for large matrices this is not always true. Secondly, beyond some key dimension the performance drops markedly and becomes roughly the same regardless of thread count. At best a performance of 3-4 GFLOPS is observed. By comparison recent work of Gahvari et al. [11] using a range of sparse matrices on a 1.4 GHz Opteron gave a maximum performance of $\sim$400 MFLOPS for an unblocked CSR SpMV and a median performance of $\sim$180 MFLOPS (On current hardware these values would probably increase by a factor of 3).

To determine when it is preferable to use SpMV over dense matrix vector products we plot in Fig. 7 the speedup of SpMV at 75% and 95% sparsity over the equivalent SGEMV runs. This shows that for 75% sparsity using SpMV can be advantageous for dimensions upto around 4000, while for 95% sparsity using SpMV is always an advantage.

## 5 Discussion and Further Directions

The results from Section 4.1 show that it is possible to transfer a large ML dataset to global memory on the GPU card at around 3GB/s. On the GPU card used in this work there was 768 MB of memory, so if 600 MB of this were used to store the ML dataset it would take $\sim$0.2 s to transfer the data from host memory to GPU memory. This is the minimum amount of time that must be saved when using the GPU instead of the host to perform the computational work. Achieving this is most likely to be possible if the dataset can be copied to the GPU once, left there and re-used in each iteration of the convex solver (Fig. 1). From Table 1 and using CSR storage this should be possible for the intrusion detection and two text categorization datasets. For larger datasets an alternative strategy would be to divide the problem/dataset over multiple GPU cards, or to use double buffering to overlap movement of data to the GPU with computation on the GPU.

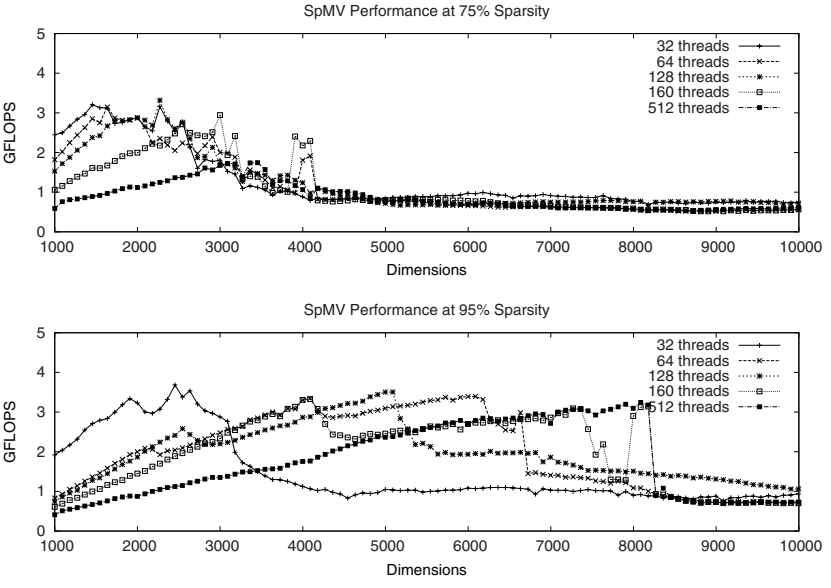


**Fig. 6.** SpMV performance at 75% and 95% sparsity

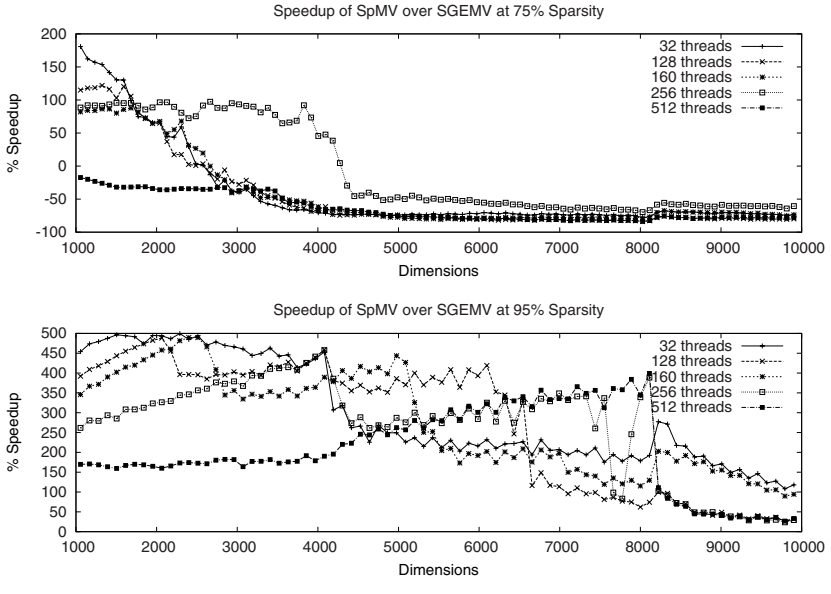


**Fig. 7.** SpMV speedup over SGEMV at 75% and 95% sparsity

Results for dense matrix vector products show a potential speedup of $3-5\times$ from use of the GPU for matrices of dimension $2500\times2500$ and above. To obtain this performance from the current version of CUBLAS requires, however, that the number of rows in the (column major) matrix be a multiple of 16. While $3-5\times$ is a useful performance gain, the cost of moving the dataset to the GPU could easily erode this advantage. As a consequence it is not clear which would be a better option if, for example, it were a choice between buying a dual core host with an NVIDIA GTX 8800 as a coprocessor or a quad core host.

Performance for our initial sparse matrix vector product is significantly better than we originally expected, achieving similar $3-5\times$ speedups over the host CPU, even for small $1000\times1000$ matrices if the sparsity is over 90%. Since many ML datasets are sparse this suggests that it would be advantageous to place further effort into optimising the SpMV routine for CUDA, and in particular, trying to eliminate the performance drop observed after certain dimensions and trying to determine automatically the optimal number of threads per block to use for a given problem size. While other approaches exist that may offer performance gains in specific areas (such as the use of coalesced memory reads), they also introduce complexities. We are in the process of evaluating such approaches. Finally, for easier integration with existing software it would be useful to implement an OSKI[2] (Optimized Sparse Kernel Interface) front-end.

## References

1. NVIDIA: NVIDIA CUDA Programming Guide. 1.0 edn. (2007)
2. Vapnik, V.N.: Statistical Learning Theory. John Wiley & Sons, Inc., Chichester (1998)
3. Teo, C.H., Smola, A., Vishwanathan, S.V., Le, Q.V.: A scalable modular convex solver for regularized risk minimization. In: KDD 2007: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 727–736. ACM, New York (2007)
4. NVIDIA: NVIDIA GeForce 8800 GPU Architecture Overview (2006)
5. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Oxford (1986)
6. Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the gpu: conjugate gradients and multigrid. ACM Trans. Graph. 22(3), 917–924 (2003)
7. Buck, I.: Data parallel computation on graphics hardware (2003)
8. Krüger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. In: SIGGRAPH 2003: ACM SIGGRAPH 2003 Papers, pp. 908–916. ACM, New York (2003)
9. Ujaldon, M., Saltz, J.: The gpu on irregular computing: Performance issues and contributions. In: CAD-CG 2005: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG 2005), pp. 442–450. IEEE Computer Society, Washington DC (2005)
10. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Graphics Hardware, pp. 97–106. ACM, New York (2007)
11. Gahvari, H., Mark Hoemmen, J.D., Yelick, K.: Benchmarking sparse matrix-vector multiply in five minute. In: SPEC Benchmark Workshop (2007)

[2] http://bebop.cs.berkeley.edu/oski/