Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2010) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2010

Runtime sparse matrix format selection

Warren Armstrong[a,*], Alistair P. Rendell[a]

[a]*School of Computer Science, ANU College of Engineering and Computer Science, Australian National University, ACT 0200, Australia*

Abstract

There exist many storage formats for the in-memory representation of sparse matrices. Choosing the format that yields the quickest processing of any given sparse matrix requires considering the exact non-zero structure of the matrix, as well as the current execution environment. Each of these factors can change at runtime. The matrix structure can vary as computation progresses, while the environment can change due to varying system load, the live migration of jobs across a heterogeneous cluster, etc. This paper describes an algorithm that learns at runtime how to map sparse matrices onto the format which provides the quickest sparse matrix-vector product calculation, and which can adapt to the hardware platform changing underfoot. We show multiplication times reduced by over 10% compared with the best non-adaptive format selection.

Keywords: Sparse matrix formats, Runtime tuning

1. Introduction

Sparse matrices are stored in a variety of formats, all aiming to reduce space and computation requirements by not storing or processing zero-valued elements. Architectural factors such as cache sizes affect different storage formats in different ways, so the best format to use is architecture dependent. The best format also depends on the matrix itself, as some matrices contain structure that can be taken advantage of to reduce indexing overhead. Processing sequences of matrices optimally requires adapting to the different non-zero structure of the matrices. It may also require adapting to a changing execution environment. The execution environment could change because of other applications running on the same machine, competing for memory bandwidth, or because of migration between nodes in virtualised heterogeneous clusters (if an application is migrated between machines with different cache sizes, the relationship between matrix structure and format performance can change).

In this paper we investigate the use of reinforcement learning for selecting sparse matrix storage formats. Our system monitors the performance of matrix-vector multiplication in a variety of storage formats and learns how to map matrices to optimal formats. The system is able to detect changes in the execution environment and to re-learn mappings when this occurs.

Using our learning algorithm, we have demonstrated a reduction in simulated sparse matrix-vector multiplication times of between 12% to 14% on a set of sample matrices. Our algorithm is used to select different storage formats

*Corresponding author.
Email address: warren.armstrong@anu.edu.au (Warren Armstrong)

for a subset of the Florida Sparse Matrix Collection [1] based on observations of performance. The performance improvement is relative to the minimum time required to process all matrices in a single format.

This paper is structured as follows: Section 2 provides background on sparse matrix formats, reinforcement learning and a few statistical techniques, as well as reviewing related work. Section 3 describes our learning algorithm, while the framework used to evaluate it is described in Section 4. Section 5 presents and discusses our results.

2. Background

2.1. Sparse matrices and storage formats

Many applications use matrices that have a large proportion of elements with a value of zero. Since the value of these elements can be inferred when needed, memory can be saved by not explicitly storing them in memory. In particular, zero-valued elements can be ignored when performing matrix-vector multiplication. There are many different formats used to store only the non-zero elements of a matrix. This paper makes use of three common storage formats: *coordinate format* (COO), *compressed sparse row* (CSR) and *blocked compressed sparse row* (BCSR). In coordinate format, each non-zero is represented by three quantities: its row index, its column index and its value. The other two formats are shown in Figure 1. In CSR format, three arrays are used to store the matrix data. One array stores the non-zero values, the second stores the column index associated with each nonzero. The third array stores the index at which non-zero values starting a row are stored.

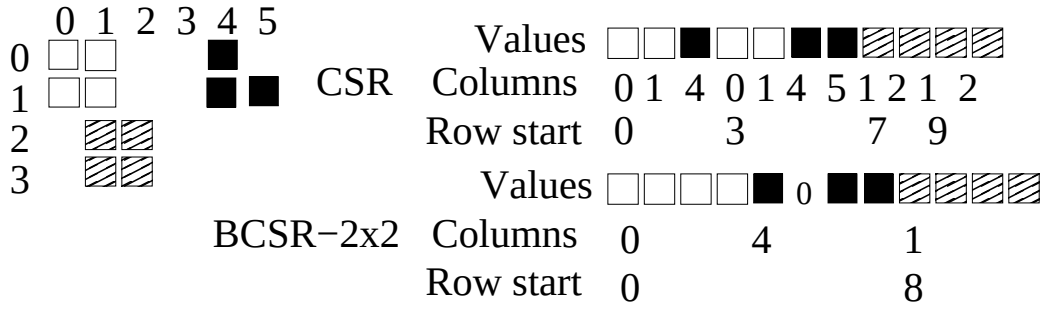

Figure 1: Two formats in which sparse matrices are stored.

The CSR format makes no assumptions about the structure of the matrix - every non-zero value is explicitly indexed. However, many matrices have clusters of non-zero elements. The BCSR format takes advantage of this to reduce the number of indirect memory accesses required; where CSR must look up one column index per non-zero processed, the BCSR format only needs to look up one index per block of non-zero values. However, if the non-zero elements are not perfectly blocked, the BCSR format must store zero values to pad out the blocks. Excessive numbers of such zeroes will make BCSR slower than CSR despite the reduced overhead (CSR's more detailed indexing means no explicit zeros need ever be stored). In this paper, the notation BCSR-$r\times c$ denotes the BCSR format for blocks with r rows and c columns. We will consider 64 different versions of BCSR from BCSR-1×1 to BCSR-8×8.

There is no one optimal format for storing sparse matrices. The speed of any format depends on the non-zero structure of the matrix as well as the attributes of the hardware platform performing the multiplication. This is illustrated in Figure 2, drawn from timing data obtained on two different platforms: a 2.2 GHz AMD Opteron 848 and a 2.2 GHz two-core Intel Core2 Duo. The sparse matrices used are those described in Section 4. The figure demonstrates that not only are different formats optimal for different matrices, but the mapping of matrix to optimal format is influenced by the architecture.

2.2. Reinforcement Learning Concepts

Reinforcement learning [2] algorithms aim to learn from experience. The learning entity (usually called an *agent*) receives perceptions of the environment. These perceptions may be incomplete or noisy. The agent has access to a set of actions, from which it must choose which one to employ. The action will affect the environment, possibly in a

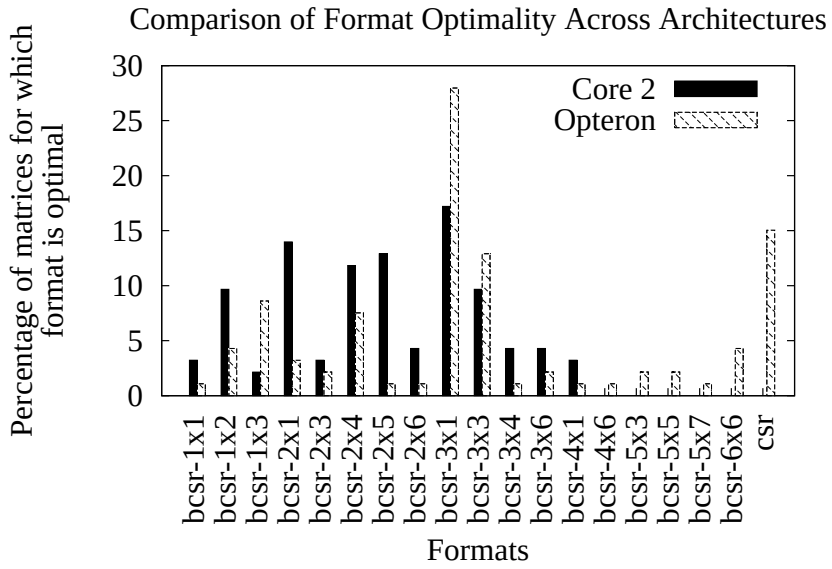

Figure 2: This figure shows the percentage of the matrix corpus for which different formats were optimal. Percentages are shown for two architectures.

delayed manner. This sequence of perception and action continues for the lifetime of the agent; the task of the agent is to learn how to map perceptions onto beneficial actions.

The learning is driven by a special input called the *reward*. This is a particular measure which the agent's designer has chosen to indicate the desirability of the environmental state. The agent tries to select actions that maximise the reward accumulated over its lifetime. To do this, the agent must make a trade-off between two classes of action: *exploitation* and *exploration*.

When choosing an exploitation action, the agent makes use of its current knowledge. It chooses that action which is predicted to yield the best payoff. In contrast, an exploration action is predicted to yield a sub-optimal payoff. It is chosen to assess the correctness of its estimated payoff- it enables the agent to try different actions in case one of them is actually more profitable than the agent hitherto believed. Pure exploitation is vulnerable to fluctuations in the environment, while pure exploration never uses its learning. Successful agents must therefore strike a balance between exploration and exploitation.

2.3. t-Statistics and p-Values

In this work, the algorithm is required to judge the significance of performance differences. This is achieved using *t*-statistics and *p*-values. A brief explanation of these concepts follows. For more details, see Moore, McCabe and Craig [3].

A two-sample *t*-statistic measures the difference in two sample means, scaled by the variation present in the sample values. The probabilities of different *t*-statistic are bell-shaped and symmetric around zero. Given a specific *t*-statistic τ, the *p*-value of τ is the area under the probability density curve for all values $\geq \tau$; it is the chance of calculating a *t*-statistic of τ if the underlying population means are actually equal. The *p*-value is used with a *significance level* α. If the *p*-value is less than α, then the two population means are declared different. Otherwise, the the chance of calculating a *t*-statistic of τ with two samples from identical underlying population means is too high. This paper uses $\alpha = 0.01$.

2.4. Related Work

In the area of automatic tuning of matrix computations, perhaps the most well-known work is the ATLAS package [4]. ATLAS is an implementation of the BLAS-specified routines for operations on matrices. It has the ability

to tune itself at installation time by using benchmarks to measure the architecture on which it is running. It does not adapt at runtime.

Runtime adaptation for sparse matrix operations has been implemented in several systems. Three examples are the systems named SPARSITY [5], OSKI [6] and AcCLES [7]. Each system builds a model of matrix performance when using different storage formats based on installation time benchmarks. Input matrices are then characterised and used as input to the model to determine the best choice of storage format. The three systems differ in the nature of their models. The main difference to our system is that our system derives its format selection rules at runtime, rather than at installation time. This gives it the advantage of flexibility, but at a cost of sub-optimal performance while selection rules are learned from experience.

Reinforcement learning was proposed for sparse matrix format selection by Armstrong and Rendell [8]. This current work differs in the learning algorithm employed, the number of formats available (65 rather than 3) and the matrices used for evaluation. Armstrong and Rendell used synthetic matrices, while this work draws from a public matrix collection. This work also differs in presenting performance improvements from the use of sparse matrix format selection - the work by Armstrong and Rendell only demonstrated that correct predictions could be achieved.

3. Design of the Learning System

This section presents a system for runtime sparse matrix format selection. The system is a library called by applications to service requests for matrix-vector multiplication. Each request contains three components: a matrix, a vector, and a number of multiplications [1]. The system first constructs an *attribute vector* for the matrix. This vector contains five measurements: *row count*, *column count*, *non-zero count*, *mean neighbour count* and *npr deviation*. The first three of these need no explanation. The *neighbour count* of a non-zero element is the number of adjacent positions which also have non-zero values. It ranges from 0 to 8 inclusive. The fourth attribute is the arithmetic mean of the neighbour count for all non-zero values. Finally, the *npr deviation* attribute is the standard deviation in the number of non-zero elements across all matrix rows.

The attribute vector and the number of matrix-vector products required are passed to the learning agent as an environmental perception. The agent responds with an action to carry out n multiplications in a specific format F. The time required to do this is measured and fed back to the agent as a reward signal. If the quantity n is equal to the total number of iterations, then the original multiplication request has been fulfilled and the library returns control to the invoking application. Otherwise, n is deducted from the total number of iterations, and the cycle begins again with the attribute vector and outstanding iteration count being passed to the agent. This process is shown graphically in Figure 3.

The rest of this section will provide more detail on the learning agent. The agent is conceptually split into two layers, the *multi-step learning layer* and the *learning support layer*. The former contains the logic for selecting actions and processing rewards, while the latter maintains the data structures used by the former.

3.1. The Multi-Step Learning Layer

The multi-step learning layer exists to provide the learning algorithm with short term memory. Every action of the learning system defines a single choice of format and iteration count. Format comparison requires data from multiple format choices. Further, making the comparison with confidence requires many timing results for each format. This layer orchestrates choices of actions to achieve these goals. It is driven by the two state machines, one to handle action selection and one to handle reward processing.

The action selection state machine is invoked with two arguments: a matrix M represented as an attribute vector and a number of multiplications R. The state machine is designed to string together multiple successive actions into a *multi-step action*. On every invocation, the machine may be in the middle of such an action. This situation will be discussed shortly. If the state machine is not in the middle of an action sequence, then the first order of business is to probabilistically select two formats (designated hereafter F_1 and F_2) and a random number (r) drawn from a uniform distribution on $[0, 1)$. If $F_1 = F_2$ and r is no less than a parameter ϵ, then a multi-step action is not undertaken. Rather,

[1] One application in which the number of multiplications is available in advance is an iterative solver, where heuristics enable estimates of the number of iterations required for convergence.

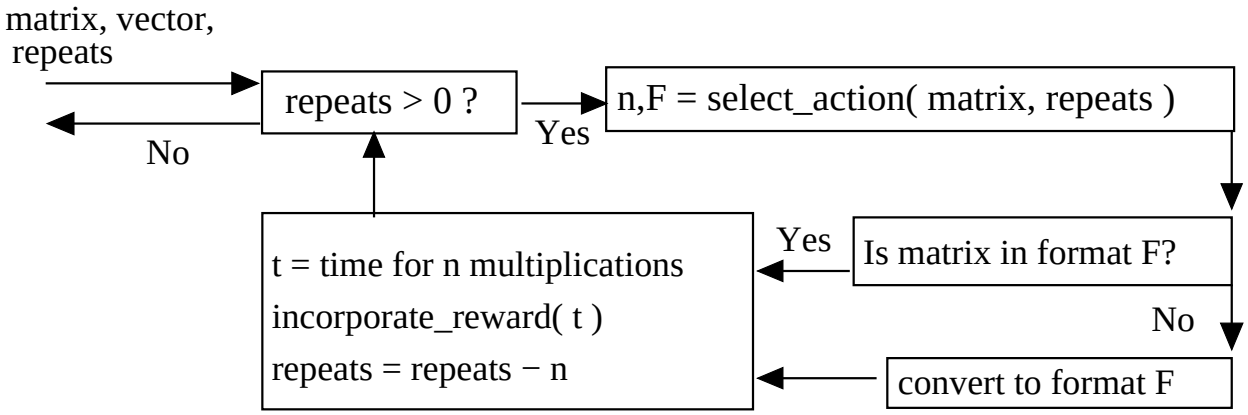

Figure 3: The way in which a learning agent directs the processing of a matrix

the current knowledge of the agent is exploited, and the invoker is instructed to undertake R multiplications using format F_1. If, however, $r < \epsilon$ then the agent forces F_2 to be distinct from F_1. The selection of two distinct formats enables their effects on matrix M to be explored. This requires a multi-step action to be undertaken.

Multi-step actions are not visible to the code which invokes the multi-step learning layer. Every invocation by that code results in one action being taken. Multi-step actions require the multi-step learning layer to schedule sequences of actions which are returned, one by one, on successive invocations. The schedule beings with an action specifying one matrix-vector multiplication in format F_1. This will force the matrix to be stored in format F_1 (the learner does not know the current matrix format, so it is conservative). The next step is to decide how many multiplications are required for every timing measurement. To take one example, if the system timer has a resolution of 1 μs and multiplying some matrix A in format F_1 requires 500 ns, then around 10 multiplications are needed to produce meaningful timing data. On the other hand, if multiplication required 500 ms, then using 10 multiplications for each timing result is overkill, and incurs an excessive penalty if F_1 turns out not to be optimal. This is handled by scheduling a sequence of actions, based on the reward provided after each action. The first action specifies one multiplication in F_1. If the reward from that action is less than a predefined minimum [2], then another action is scheduled, this time specifying two multiplications in F_1. This cycle continues, doubling the number of multiplications in successive actions, until the reward received exceeds the minimum. Let S denote the number of actions that yielded an acceptable reward. The schedule then continues with K actions, each specifying S multiplications in format F_1. The reward from each action is divided by S and the result is recorded for future use. Once the K actions have been completed, the schedule continues by carrying out the same steps for format F_2. When this is complete, the learner will possess two data sets, each containing K timing results.

For each data set, the mean and variance are calculated, and from these quantities are derived a t-statistic and a p-value. These two measures enable a choice to be made as to whether there is a statistically significant difference between the iteration times for the two formats. If there is, then the probability of choosing the slower format is reduced by a predefined algorithm parameter δ. No action is taken for the faster format, because to do so would be to raise it above other formats, elevation for which this comparison provides no justification.

As well as constructing the data sets mentioned above, the reward processing state machine has one other function which needs to be mentioned. Whenever a format weighting may change, the mean (μ_{prob}) and standard deviation (σ_{prob}) of all format probabilities for the current matrix are calculated. These quantities are checked against two threshold values, $\mu_{trigger}$ and $\sigma_{trigger}$, given as parameters to the algorithm. If $\mu_{prob} < \mu_{trigger}$ and $\sigma_{prob} < \sigma_{trigger}$, then more exploration is required. This inference flows from the following line of reasoning: Weights are never increased by the comparison of timing results. The weight of format A is reduced when A is measurably worse than some other format B. If $\mu_{trigger}$ is far enough from the initial weights then sufficiently small variation around $\mu_{trigger}$ means

[2] The minimum is calculated based on measuring the system timer.

every format has been judged significantly worse than some other format. This can only happen if the context of the judgement (for example, the underlying architecture) has changed during the learning process. To deal with this, the chance of every format on every matrix is reset to its starting value. The above reasoning does not represent a mathematical proof, because we cannot rule out corner cases where timing results on a single platform would induce the appropriate drops in μ_{prob} and σ_{prob}. However, we have not encountered such problems in practice.

3.2. The Learning Support Layer

This layer supports the multi-step learning layer by providing generalisation and managing format weights.

Generalisation is important because it allows inference. A learner that can judge states *A* and *B* through testing *A* alone can learn faster than one that must also test *B*. Generalisation also reduces memory requirements - data need only be stored about groups of states.

The learning support layer generalises by grouping matrices according to their attributes. For the i^{th} attribute a_i, the layer is supplied with bounds $[L_i, H_i]$ on the values of a_i, as well as a user-supplied granularity g_i. It calculates a chunk size $s_i = \frac{H_i - L_i}{g_i}$. Input matrices will have their values of a_i mapped onto one of g_i chunks: $[L_i, L_i + s_i)$, $[L_i + s_i, L_i + 2s_i)$, ...

The second role of this layer is to manage format selection weights. Matrix attributes are generalised to chunks, and every set of chunked attributes is associated with a set of formats. Every format is assigned a weight w_i, initially set to $\frac{1}{65}$ as there 65 possible formats. The total weight is given by $T = \sum_{i=1}^{65} w_i$. To convert the weights into probabilities, let S_k denote the partial sum $\sum_{i=1}^{k} \frac{w_i}{T}$, and define $S_0 = 0$. Let r be a random value drawn from a uniform distribution on the interval $[0, 1)$. Define m so that $S_m \leq r$ and $S_{m+1} > r$. Then the format m is the one selected.

Once the multi-step learner has decided to reward a particular action, it informs the learning support layer, which must then adjust the format's weighting. The weight is incremented by the reward, and then bounded to remain within the interval $[10^{-6}, 1]$. The lower bound is used to avoid formats falling into complete disuse. If a format weight reached zero, for instance, it would never be selected, and so the system would have no way of noticing when changing conditions made it a good choice. Without the upper bound, a format's weighting could grow so large that it is chosen long after changed circumstances have rendered it non-optimal.

4. Design of the evaluation environment

To enable rapid exploration of different parameter settings, the algorithm in Section 3 was evaluated using a simulator. The simulator is provided with pre-measured timing data for matrix vector products performed on a group of M matrices using all 65 formats running on two different hardware platforms. Using these results workloads of size N are constructed, where every element of the workload is a sparse matrix which is simulated to undergo Q matrix-vector multiplications.

The simulator is driven by a set of data files, one file per simulated architecture. Each file has one line per matrix. That line contains, for each format, the mean and standard deviation of the time required to perform one multiplication of that matrix in that format, as well as the time required to convert to that format from coordinate format. To form the simulated workload, command line parameters are used to set the values of N and Q, and to seed a random number generator. For each data file, a set of N random integers uniformly distributed on $[0, M)$ are generated. These numbers determine the sequence of matrices used in the evaluation. Each matrix in the sequence is initially set to be stored in coordinate format. The workloads for each architecture are executed sequentially.

Evaluation of the algorithm using the generated sequence proceeds by using the matrices one at a time. Each matrix is passed to the learning algorithm, which returns the format to use. If the matrix is not currently stored in that format, then the format conversion cost is added to the simulated elapsed time, and the matrix's current format is updated. The time required to multiply the matrix is taken from the input data files, which supplies the mean I_{mean} and the standard deviation I_{dev} of the time required for one multiplication. The elapsed time is calculated as $QI_{mean} + \sum_{j=1}^{Q} Z_j$, where Z_j are random numbers drawn from a Gaussian distribution with mean zero and standard deviation I_{dev}.

The simulator tracks, for every format, the total cost of using only that format for every matrix. This enables comparison of the dynamic format selection strategy to what is attainable without such selection. The simulator also

tracks the cost incurred if the best format is chosen for each matrix. This provides another point of comparison: the best possible performance for this workload, given perfect tuning and no need to learn.

The matrices used in our experiments come from the Florida Sparse Matrix Collection [1]. The initial selection culled those matrices that were deemed to be too small, too large or too dense. Of the remaining 576 matrices, two optimal formats (CSR and BCSR-*1×1*) dominated. To create a more even distribution of optimal formats, we removed some matrices for which CSR or BCSR-*1×1* were optimal.

This may seem to contradict the premise of this paper: that there is no one optimal format, and finding the best format for specific circumstances requires runtime tuning. However, this is not the case - it is certainly possible to construct groups of matrices where there is an optimal format. A machine and matrix library dedicated to only one application may well not require tuning. However, there do exist matrix workloads on non-dedicated machines where the components of the workloads require different formats for optimality. We have merely constructed such a workload, to demonstrate this fact and to show how to tune the processing of these workloads.

4.1. Parameters used in this paper

All experiments in this paper have been performed with the parameters of the learning layers fixed to the following values: $\mu_{trigger} = 0.0001$, $\sigma_{trigger} = 0.0001$, $\delta = 0.95$, $\epsilon = 0.01$. (Unreported experimentation has guided some of these choices, with others based solely on intuition.) The attributes of the matrix corpus are given in Table 1.

Attribute	**Low bound**	**High bound**	**Granularity**
row count	1005	17281	100
column count	1005	17281	100
density	5.04×10^{-4}	3.75×10^{-2}	100
row deviation	4.50×10^{-1}	1.28×10^{2}	100
mean neighbours	0	8	10

Table 1: Attributes of the matrices used in this paper.

5. Results and Discussion

Figure 4 compares three format selection agents on the task of processing 180,000 matrices 10,000 times each, with the first 90,000 on a 2.2 GHz Intel Core2 Duo and the second 90,000 on a 2.2GHz AMD Opteron. The *dynamic* agent implements the algorithm described in Section 3. The *static* agent uses a single format for all matrices - the chosen format is the one that accumulated the least simulated execution time. The third agent is an oracle which performs dynamic format selection, but has no need to learn - it always knows the best format for every possible matrix. It serves to show the upper bound on attainable performance.

The results show that after processing many matrices, the dynamic agent is 13% faster than the static agent, and within a few percent of the oracle. The graph also shows a brief upward surge in the $\frac{dynamic}{static}$ and $\frac{dynamic}{oracle}$ ratios, at the point where the architecture changes. At this point, the learning agent makes incorrect choices, incurring more execution time. This situation lasts until exploration forced via ϵ reduces μ_{prob} and σ_{prob} sufficiently to trigger a resetting of format weights. This allows selection chances to be shaped to the changed environment. A further point to note from Figure 4 is that, after approximately 18,000 of the 90,000 matrices have been processed on the Core2 architecture, the dynamic agent has already incurred less time than the static agent. When the architecture changes, the agent takes a while to adapt, and selects formats not optimal on the Opteron. Within 18,000 matrices of the change, though, the agent has adapted and is once more making optimal decisions.

It has been demonstrated that dynamic format selection can be profitable. To show that this outcome is not dependent on a particular sequencing of the matrix corpus, we generated four different matrix sequences. These sequences showed similar speedups of between 11% and 14%.

There are two caveats to the results shown so far: the number of iterations per matrix (Q) is large, and the number of matrices processed may appear large. The latter is not really an issue as a matrix multiplication library that can

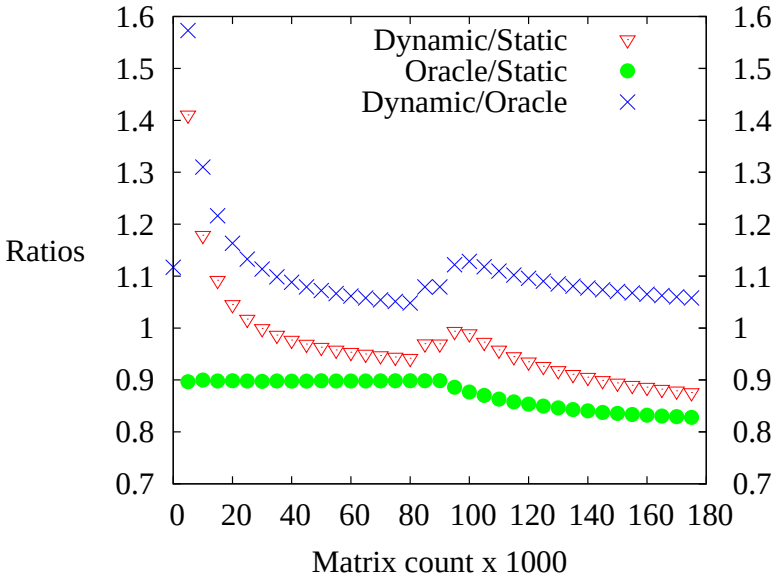

Figure 4: Incremental ratios of simulated execution time for three different format selection strategies.

save and restore learned state could well see such numbers occurring. The former issue needs to be addressed to make this solution more practical.

The value of Q has the effect of amortising the cost of format conversion. We hypothesise that changing the cost of format conversion relative to the cost of multiplication should allow smaller values of Q to be profitable. To check this, we took the databases which drove the simulator and scaled the format conversion costs. We then ran the simulator using the scaled databases, a workload size of 32,000 and different values of Q. The results are given as the points in Figure 5. The lines in the figure are models of the form $f(x) = mx + b$ fitted to the points. It appears that the slope of the models decreases with increasing Q, a fact shown numerically in Table 2. The models with the shallower slope stayed below the break-even ratio of 1.0 for higher scaling factors. This suggests that the hypothesis is correct: low values of Q are less tolerant of expensive format conversions.

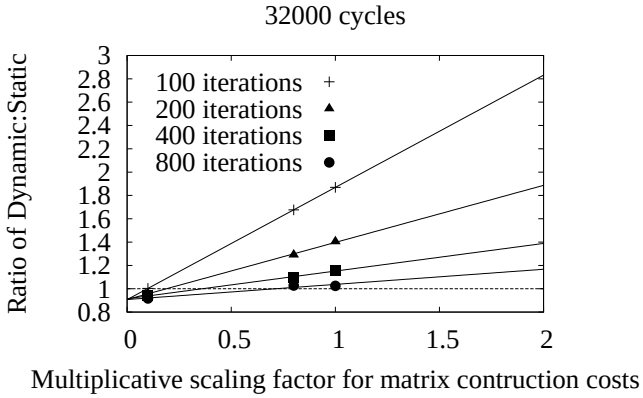

Figure 5: The relationship between format conversion expense (relative to iteration cost) and performance gain by the dynamic format selection algorithm.

Q	100	200	400	800
Gradient	0.96	0.49	0.24	0.13
Offset	0.91	0.91	0.91	0.91

Table 2: Parameters of linear functions fitted to a plot of final ratios versus scaling of conversion costs.

To further assess the applicability of these techniques, we measured the performance impact of the learning process. The results are plotted in Figure 6, which was generated using the same parameters as Figure 4. Figure 6 shows the ratio between execution time of the learning agent (measured using the `gettimeofday` system call) and the simulated time spent in sparse matrix-vector operations. The experiment was run on the Core2 system, which is the system for which the processing of the first 90,000 matrices was simulated. Therefore, only the first 90,000 matrices are shown in Figure 6. The time required for SpMV is dominant, indicating that the learning code itself is not a major factor in system performance.

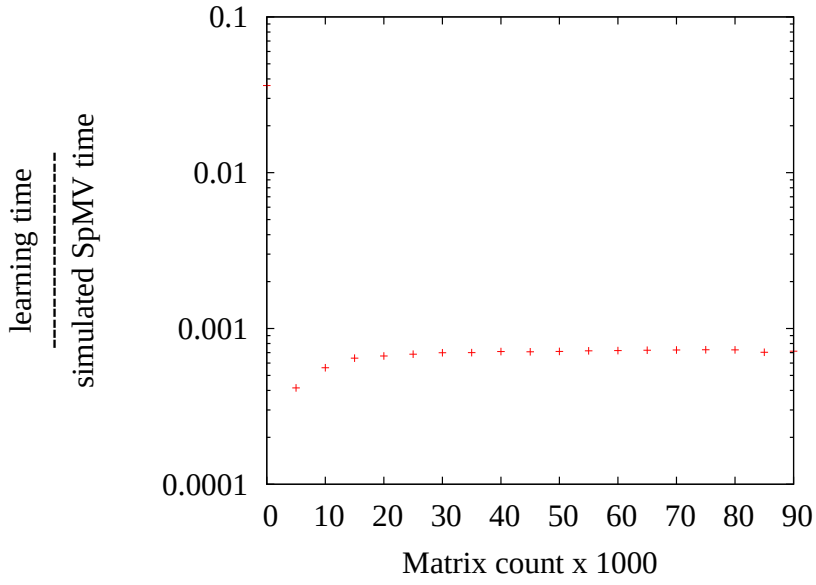

Figure 6: The relative cost of executing the learning code and executing sparse matrix-vector (SpMV) operations.

6. Conclusions and Future Work

In this paper, we have presented an algorithm for runtime selection of sparse matrix storage formats. We showed that, using a simulation driven by experimental matrix-vector multiplication times from two architectures, our system can adapt to changes in the underlying architecture as well as to the variations in non-zero structure of the matrices it is presented with. We demonstrated a speedup of 12% to 14% over the best single-format choice, and analysed the sensitivity of this result to the relative cost of format conversion.

In future work we will investigate the sensitivity of our results to factors such as the attributes used to represent matrices, the method of chunking matrix attributes, different methods of generalisation over matrices and formats and different environments, including hybrid CPU/GPU computational platforms. We will minimise the computational cost of the learning algorithm, and replace simulation with concrete computation.

References

[1] T. Davis, University of florida sparse matrix collection, NA Digest 97 (23).
URL http://www.cise.ufl.edu/research/sparse/matricies
[2] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, The MIT Press, 1998.
[3] D. S. Moore, G. P. McCabe, B. A. Craig, Introduction to the practice of statistics, 6th Edition, 2009.
[4] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimizations for software and the ATLAS project, Parallel Computing 27 (1-2) (2001) 3–35.
[5] E.-J. Im, K. Yelick, R. Vuduc, SPARSITY: Optimization framework for sparse matrix kernels, The International Journal of High Performance Computing Applications 18 (1) (2004) 135–158.
[6] R. Vuduc, J. W. Demmel, K. A. Yelick, OSKI: A library of automatically tuned sparse matrix kernels, in: Proceedings of SciDAC 2005, Journal of Physics: Conference Series, Institute of Physics Publishing, San Francisco, CA, USA, 2005.
[7] A. Buttari, V. Eijkhout, J. Langou, S. Filippone, Performance optimization and modeling of blocked sparse kernels, International Journal of High Performance Computing Applications 21 (4) (2007) 467–484. doi:10.1177/1094342007083801.
[8] W. Armstrong, A. Rendell, Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection, in: International Workshop on Automatic Performance Tuning, 2008, pp. 411–420. doi:10.1109/CLUSTR.2008.4663802.