Evaluating Matrix Representations for Error-Tolerant Computing

Pareesa Ameneh Golnari and Sharad Malik
Princeton University, {amene, sharad}@princeton.edu

***Abstract*—We propose a methodology to determine the suitability of different data representations in terms of their error-tolerance for a given application with accelerator-based computing. This methodology helps match the characteristics of a representation to the data access patterns in an application. For this, we first identify a benchmark of key kernels from linear algebra that can be used to construct applications of interest using any of several widely used data representations. This is then used in an experimental framework for studying the error tolerance of a specific data format for an application.**

As case studies, we evaluate the error-tolerance of seven data-formats on sparse matrix to vector multiplication, diagonal add, and two machine learning applications i) principal component analysis (PCA), which is a statistical technique widely used in data analysis and ii) movie recommendation system with Restricted Boltzmann Machine (RBM) as the core. We observe that the `Dense` format behaves well for complicated data accesses such as diagonal accessing but is poor in utilizing local memory. Sparse formats with simpler addressing methods and a careful selection of stored information, e.g., `CRS` and `ELLPACK`, demonstrate a better error-tolerance for most of our target applications.

I. Introduction

There is a growth in accelerator-based platforms because of their greater computational speed and energy efficiency. Further, many applications utilizing these platforms, such as inference applications, have large and often sparse data sets. Thus, they might benefit from using sparse data representations. While the speed benefits of sparse data formats has been studied in previous works [8], [12], their error-tolerance has not been studied before. This is increasingly relevant in late- and post-CMOS technologies where hardware failure threats are increasing [4], which has prompted the study of hardware failure effects for processors [7], [13]. Further, many applications such as media and inference applications are error-tolerant in that they can tolerate some level of errors in their results and it is not immediately obvious which of several possible data-representations may be suitable for a particular application in terms of providing the highest error-tolerance.

We address this question by presenting a methodology for accelerator-based computing to decide the most suitable data format for a specific application in terms of its error-tolerance. This methodology helps match the characteristics of a representation to the data access patterns in an application. For this, we first identify a benchmark of key kernels from linear algebra that can be used to construct applications of interest using any of several widely used data representations. This is then used in an experimental framework for studying the error tolerance of desired applications. As case studies, we evaluate the error-tolerance of seven data-formats on sparse matrix to vector multiplication (SpMV), diagonal add, and two machine learning applications i) principal component analysis (PCA), which is a statistical technique widely used in data analysis and ii) movie recommendation system with Restricted Boltzmann Machine (RBM) as the core. Our experimental results show that (i) `CRS` and `ELLPACK` demonstrate better error-tolerance than other formats most of the time because of their simple addressing methods and the useful information they store about the matrix. (ii) Sparse formats demonstrate higher error-tolerance for uncomplicated data access patterns. (iii) While the `Dense` format is better for complex access patterns it has poor utilization of local memory.

Overall, we make the following contributions: (i) We discuss the parameters that impact the choice of the data-format for a given application. (ii) We present a benchmark of key kernels to help evaluate the error-tolerance of data-formats for different applications. (iii) We evaluate the error-tolerance of seven data-formats for SpMV, diagonal add, and machine learning applications of PCA and RBM.

II. Matrix Formats

Among many data-formats in the literature [2], we selected `Dense` format and six popular unstructured sparse formats.

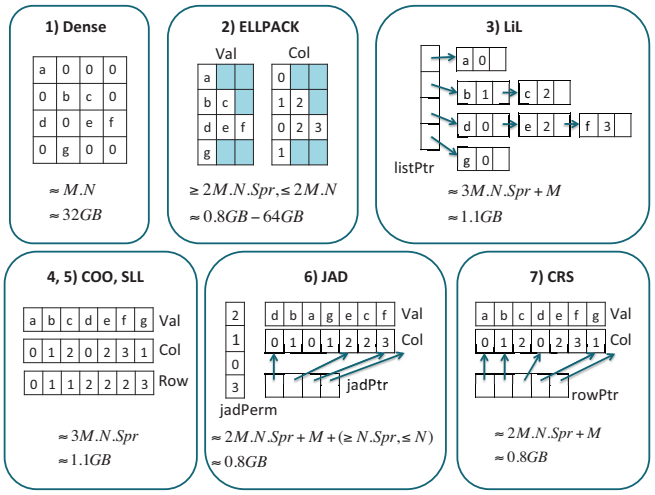

Fig. 1: Formats in decreasing order of storage requirements.

A. Sparse Matrix Formats

Compressed row storage (`CRS`) stores the non-zero values, the column indexes, and pointers to the beginning of each row.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

Co-ordinate list (COO) stores the non-zero values and their row and column indexes in a 2-D matrix.

Single linear list (SLL) stores same information as COO in a linked list or three separate vectors. The storage types of SLL and COO are different with differences in their address generator and memory access.

List of lists (LIL) stores the non-zeros of each row in a separate linked list. Vector $listPtr$ in Fig. 1 provides pointers to the beginning of each row.

ELLPACK requires two matrices. One matrix stores the non-zero values of row i in its i^{th} row and the second matrix stores the column indexes of the non-zeroes in the same way.

Jagged diagonal (JAD) sorts the rows of the matrix by the length of each row in decreasing order [6]. $jadPerm$ in Fig. 1 stores the permutation of the rows. Val stores the non-zero values of the matrix starting with the first non-zeros of all the rows, then the second non-zeros of the rows and so on. Col stores the column indexes of the non-zeros and $jadPtr[i]$ provides a pointer to the i^{th} non-zero of the first row.

B. Memory Requirements

Fig. 1 gives an estimate of the storage requirements for a matrix with M rows, N columns and Spr sparsity (the ratio of the non-zeros to all elements). For a numerical comparison, we assumed the Netflix data-set providing ratings for 480,189 users (M) on 17,770 movies (N) with 0.012 sparsity (Spr) [10] is stored in these formats.

C. Preprocessing

Sparse formats may require significant preprocessing time to read or modify a matrix. For instance, to modify a matrix that is stored in LIL or ELLPACK formats, we only need to access the elements of the respective row and add or remove the desired element. But in case of CRS, COO, and SLL formats, all non-zeros after the target element might need to be shifted. The worst case scenario is with the JAD format. Since the change in the matrix might change the order of the rows, JAD might require significant processing to rearrange the matrix. That is the reason this format is not used for the applications that need to modify the matrix. That is the reason we excluded JAD for diagonal add experiments.

III. Benchmark of Data-Access Kernels

Based on different data access patterns and matrix computations found in linear algebra libraries, we identified nine benchmark operational kernels for use as accelerator primitives. These kernels cover most data-access patterns of common linear algebra operations:

(i, ii, iii) $\boldsymbol{Fetch_{row}}$, $\boldsymbol{Fetch_{col}}$, and $\boldsymbol{Fetch_{diag}}$: Read a row, a column, or the diagonal respectively.

(iv, v) $\boldsymbol{Add_{row}}$ and $\boldsymbol{Add_{col}}$: Append a row or a column to the matrix, which increases the matrix size.

(vi) $\boldsymbol{Remove_{col}}$: Removes a column from the matrix.

(vii) $\boldsymbol{Update_{diag}}$: Updates the diagonal values.

(viii) $\boldsymbol{Op_{vec-vec}}$: Performs operations on two vectors resulting a new vector, e.g., adding two vectors.

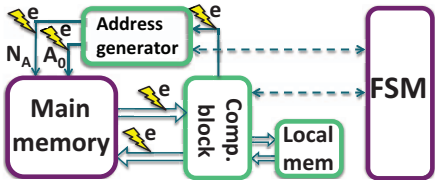

Fig. 2: Simulation setup. Color green distinguishes the accelerator's boundry.

(ix) $\boldsymbol{Red_{vec}}$: Is vector reduction, e.g., sum of the vector, standard deviation, maximum or minimum, *etc.*.

We built a software model for these kernels, which supports the seven discussed data-formats and allows for simulating the error-effects on the respective accelerators for these kernel operations. For instance, SpMV is implemented by composing $Fetch_{row}$, $Op_{vec-vec}$, and Red_{vec}. This provides an experimental framework for studying the effect of errors on applications that can be composed using these kernels.

A. Case Studies

We used the proposed software model to evaluate the error-tolerance of seven data-formats on selected numerical linear algebra operations and two machine learning inference applications that often use sparse data: RBM and PCA.

Our selected set of operations cover many of the linear algebra computations. This set includes: **matrix multiplication**, **SpMV**, **matrix accumulation**, **diagonal accumulation** ($A + \alpha.I$), **submatrix**, **matrix transposition**, and **matrix row or column permutation**. However, because of space limitations, we only present the results of SpMV and diagonal accumulation. These two operations cover simple and complex (row-based and diagonal) data access in matrix computation.

RBM is a two-layer artificial neural network, which is a good example of stochastic learning applications and PCA is a statistical technique widely used in data analysis including dimension reduction, feature extraction, and clustering. Thus, this set provides relevant test-cases for our study.

IV. Simulation Setup

We consider the implementation of our target applications on embedded accelerators. As we are primarily interested in the *relative* error-tolerance of data formats, we consider a high-level accelerator and error model for our studies. This is in contrast to a low-level (e.g. gate-level) model which would be needed for determining absolute error-tolerance.

Fig. 2 depicts our simulation setup, where we model the main characteristics of an accelerator including: data access, computation, control flow, and error-effects. Algorithm 1 shows an example of reading element $A[i][j]$ (stored in CRS format) based on this setup.

In this piece of code, address generator generates address of $rowPtr[i]$ and $rowPtr[i+1]$ after receiving the pointer to the beginning of the $rowPtr$ array, the type of the elements in that array, and the index i from the computational block. The address of the beginning of the data block and the size of the block (A_0 and N_A in Fig. 2) are passed to the main memory.

Algorithm 1 Find the i^{th} row then search through its elements.

```
begin-row_i ← rowPtr[i]
end-row_i ← rowPtr[i + 1]
for begin-row_i ≤ k < end-row_i do
    if col[k] = j then
        data ← val[k]
        break
    end if
end for
```

After reading $begin\text{-}row_i$ and $end\text{-}row_i$, the range for k is determined. Then for each k, the address generator computes the address of $col[k]$ and the data is read and passed to the computational block.

A. Local and Main Memory Blocks

We assume that the system has both a main memory and a local memory. The computational block uses the local memory to store the data read from the main memory, its partial results and its local parameters. The importance of a separate local memory for large data-set accelerators has been highlighted in recent research [5]. This type of local memory is comparable to an L1 data cache with the order of $\approx$ 64KB size [5]. In our setup, this means that the local memory can contain non-zeros of a few sparse vectors, but it cannot fit a whole row or column assuming that the matrix is very large and very sparse. For instance, considering the Netflix data-set and assuming each element occupies 4B, one whole row and column occupy $\approx$ 2MB while one sparse row and column occupy $\approx$ 24KB. Moreover, since the local memory is comparable to a data-cache, we assume that accessing this memory is direct, i.e., without requiring the address generator, and the memory and the logic accessing it are error-free, e.g. using error-correcting codes (ECC) as with cache memory. On the other hand, the main memory is large and with error-prone access.

B. Error Injection

The arrows labeled "**e**" in Fig. 2 indicate error injection. The arrows on the transitions to or from the main memory block capture memory access errors while the rest of the arrows capture logic errors. Errors are injected on the transitions between the blocks to model the effect of the faults that occur inside the source block. These errors follow some known distribution (uniform in our experiments). In this setup, we consider partial error injection, i.e., we target the data stored in sparse formats and the respective computations and inject error only in those parts. Since the injected errors could propagate to the control flow, we assume minimal required protection (studied in [13]) to avoid the control flow from crashing.

C. Application Implementation

1) Movie Recommendation with RBM: We modified the existing implementation of RBM in the CortexSuite [11] for our experiments to run faster and use a larger available data-set with a higher sparsity. We used 456,210 ratings of 61,459 users on 100 movies (sparsity $\approx$ 7.4%) taken from the MovieLens data-set [1] and managed to achieve prediction with $\approx$ 96.4% accuracy in error-free experiments.

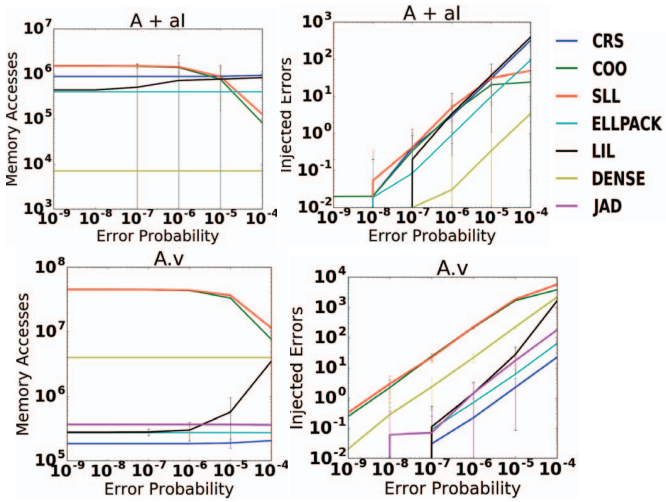

Fig. 4: More memory access results in receiving more errors.

2) PCA: We modified the existing implementation of PCA in CortexSuite [11] for our experiments by adding support for sparse formats and error-prone computation. We also rearranged the computation steps to keep the data sparse through more of the computation to benefit from the sparse storage. We used the UCI data-set [3] with records of 1000 words in 1000 documents with 11.47% sparsity.

The C code for the kernels and the applications is available at https://github.com/SparseFormats/Kernels-and-Apps.

V. Experimental Results

We used the proposed methodology to evaluate the relative error tolerance of different data representations for our target applications, which perform different types of computations and data access patterns. For instance, SpMV and RBM access data in row order but PCA accesses data in column order.

We modeled the target applications using the software model proposed in Section III based on the simulation setup in Fig. 2. We performed each experiment 100 times for different error rates and with uniform error distribution. The size of the selected data-sets was limited by the low speed of simulations, especially in presence of errors, and the high storage requirements for formats like `ELLPACK` and `Dense`.

Fig. 3 shows the accuracy of the results in the presence of error. We measured the signal to noise power ratio $(SNR) = \frac{|B_{exp}|^2}{|B_{res}-B_{exp}|^2}$, where B_{exp} and B_{res} are the expected and measured results respectively. They could be either scalar values, matrices, or vectors depending on the application.

In all the plots, the error probability on the X-axis shows the probability of error injection at the locations shown in Fig. 2 and the error bars show 90% confidence interval.

A. Observations

Among the sparse formats, `JAD` shows a good error tolerance for PCA and SpMV. `ELLPACK` shows the best error tolerance most of the time, after that are `LIL`, `CRS`, `COO`

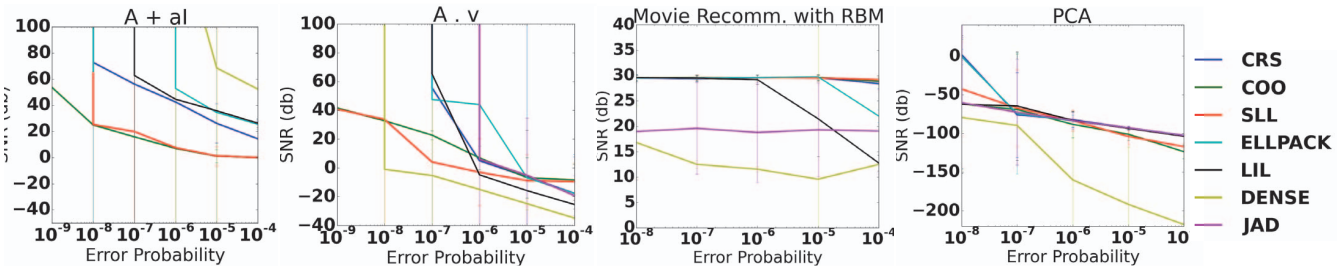

Fig. 3: Accuracy of linear algebra matrix operations using different sparse formats and in presence of errors.

and SLL in that order. The accuracy of Dense for diagonal access in $A + \alpha I$ is better than sparse formats because of the straightforward addressing. Otherwise, as discussed later, Dense shows the worst results because of its limitations in utilizing the local memory. Fig. 4 is helpful to better understand the reason behind this behavior of data-formats. It shows that the number of reads-and-writes and number of received errors grow similarly. This is expected, because more memory access results in higher probability of error injection.

In the following, we discuss the results and study the parameters that affect error-tolerance and are important in deciding the appropriate data-format for an accelerator.

B. Discussion

1) Features of Data-Formats and Error-Tolerance: **Fixed addressing method** in Dense format requires also storing zeros, which restricts the benefit of local memory storage. When a row or column of the matrix is fetched, the whole vector does not fit in local memory and all or part of it is stored in main memory. This causes increased memory accesses and this increases the probability of error. Note that this type of local memory is comparable to L1, therefore it is not feasible to increase its size from the order of tens of Kilo Bytes to the order of Mega Bytes. Thus, while Dense format benefits from straightforward addressing for complex data accesses, it does not utilize the local memory well and it shows a relatively poor error tolerance for simple data accesses.

Simplifying the addressing method by storing more information is generally a possible way to improve the error-tolerance. For instance, in ELLPACK, the start of each row is simply given. While in CRS, we need to first fetch $rowPtr[i]$ to find the beginning of the desired row. However, a wise format design could break this trade-off between cost (memory size and bandwidth) and error-tolerance. For instance, CRS stores $rowPtr$ instead of row in COO and SLL, thus, CRS occupies less memory but tolerates errors better than those formats, which indicates that CRS stores less but *more useful* information about the data.

2) Error-Resilience of Learning Applications: Most of the sparse formats show a very good error-tolerance for PCA and RBM (Fig. 3). Also, the flat SNR in the low error rates shows the inherent error-tolerance of these applications. This interesting feature has also been observed in other learning applications [9]. There it was noted that these applications implicitly learn the error-statistics during the training phase.

VI. Conclusions

We presented a methodology for selecting a suitable data-format for implementing an error-tolerant application on an embedded accelerator in the presence of hardware faults. We proposed a benchmark consisting of nine kernels that cover most of data-access related operations in different applications. We then used the proposed benchmark to study the features of data-formats and showed the importance of them in matching a particular data format to a particular application. For instance, we observed that the Dense format behaves better for complex data accesses, such as diagonal, but it has poor utilization of local memory. Sparse formats with simpler addressing method and better selection of stored information (i.e., CRS and ELLPACK) showed better error tolerance. We also observed that learning applications benefit from inherent error-tolerance, especially when utilizing sparse formats.

Acknowledgment

This work was supported in part by SONIC, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] MovieLens. http://grouplens.org/datasets/, 2016.
[2] SPARSKIT. http://www-users.cs.umn.edu/~saad/, 2016.
[3] UCI. http://archive.ics.uci.edu/ml/datasets/, 2016.
[4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO*, 25(6):10–16, 2005.
[5] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *DAC*. ACM, 2015.
[6] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. LAPACK working note 74: A sparse matrix library in C++ for high performance architectures. Technical report, University of Tennessee, 1994.
[7] A. Golnari, Y. Vizel, and S. Malik. Error-tolerant processors: Formal specification and verification. In *ICCAD*. IEEE, 2015.
[8] M. R. Hugues and S. G. Petiton. Sparse matrix formats evaluation and optimization on a GPU. In *HPCC*. IEEE, 2010.
[9] K.-H. Lee, Z. Wang, and N. Verma. Hardware specialization of machine-learning kernels: Possibilities for applications and possibilities for the platform design space. In *SiPS*, 2013.
[10] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML*. ACM, 2007.
[11] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. Bedford Taylor. Cortexsuite: A synthetic brain benchmark suite. In *IISWC*. IEEE, 2014.
[12] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, Citeseer, 2003.
[13] Y. Yetim, M. Martonosi, and S. Malik. Extracting useful computation from error-prone processors for streaming applications. In *DATE*. IEEE, 2013.