Towards Reversible Basic Linear Algebra Subprograms: A Performance Study

Kalyan S. Perumalla and Srikanth B. Yoginath

Oak Ridge National Laboratory[⋆],
Oak Ridge, TN 37831-6085, USA
perumallaks@ornl.gov,yoginathsb@ornl.gov

Abstract. Problems such as fault tolerance and scalable synchronization can be efficiently solved using reversibility of applications. Making applications reversible by relying on computation rather than on memory is ideal for large scale parallel computing, especially for the next generation of supercomputers in which memory is expensive in terms of latency, energy, and price. In this direction, a case study is presented here in reversing a computational core, namely, Basic Linear Algebra Subprograms (BLAS), which is widely used in scientific applications. A new Reversible BLAS (RBLAS) library interface has been designed, and a prototype has been implemented with two modes: (1) a memory-mode in which reversibility is obtained by checkpointing to memory, and (2) a computational-mode in which nothing is saved, and restoration is done entirely via inverse computation. The article is focused on detailed performance benchmarking to evaluate the runtime dynamics and performance effects, comparing reversible computation with checkpointing on both traditional CPU platforms and recent GPU accelerator platforms. For BLAS Level-1 subprograms, data indicates over an order of magnitude speed up of reversible computation compared to checkpointing. For BLAS Level-2 and Level-3, a more complex tradeoff is observed between reversible computation and checkpointing, depending on computational and memory complexities of the subprograms.

Keywords: Reversible computation, linear algebra, checkpointing, runtime performance, memory effects

1 Introduction

1.1 Reversible Computing

Reversible computing is a computing paradigm which uses a bidirectional execution capability as the basis for computation, in stark contrast to all traditional

[⋆] This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Dept. of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

computing that is essentially built on unidirectional execution. Reversible computing has been evolved over the past few decades, primarily driven by the need to lower the energy consumed by computation. Besides low-power computing (sometimes also called *adiabatic* computing, or, more precisely, *asymptotically isentropic* computing [7]), an important additional benefit of reversible computig arises in parallel computing. Reversible computing is useful in parallel computing to address critical problems such as rollback recovery-based fault tolerance and optimistic synchronization. For example, reversible computation has been recently shown to support extremely efficient rollback-based recovery [5, 4] for fault-tolerant execution due to its many desirable properties such as low memory footprint, minimal cache pollution, and significant reduction of congestion at file systems [14, 12]. Similarly, reverse computation has been successfully applied in parallel discrete event simulations to efficiently realize Time Warp-based synchronization on large-scale parallel computing platforms with over 10^5 processors [3, 13]. While these indicate the significant benefits of reversible computation, introducing reversibility into any complex application remains a major challenge.

1.2 Motivation

To reap the benefits from reversibility in parallel computing, parallel programs need to be augmented or transformed for reversible execution, which is a very complex endeavor. While automation can help to an extent, the highest runtime- and memory-efficiency for reversibility is only achieved via special runtime support, especially for extant codes. Complex subroutine libraries are used by parallel programs, whose efficient reversal is needed before the benefits of reversible computing can be fully obtained. In this direction, here we examine the runtime support for efficiently adding reversibility to one of the core computational blocks, namely, Basic Linear Algebra Subprograms (BLAS), which is a performance-critical, optimized library widely used in many parallel scientific applications that rely on fast linear algebra operations.

In the case of linear algebra operations, every routine modifies the entire vector or matrix in general, limiting the effectiveness of well-known optimizations to checkpointing (such as incremental or differential checkpointing). This implies that reversible computation can significantly relieve the pressure on the memory system in comparison to any checkpointing-based solution. Checkpointing also has the undesirable overhead of memory copying operations performed even in forward execution; this overhead can be pronounced for larger vectors and matrices. Reversible computation, on the other hand, can proceed at almost the same speed as irreversible execution with negligible overheads in runtime or memory. This aspect of reversible computation can deliver a significant gain in parallel execution in which the probability for reversal is low. However, while checkpointing can guarantee exact restoration of values, the numerical accuracy of reversible computation becomes an important consideration.

A specific aim of our approach here is to relieve reliance on memory (checkpointing) in enabling reversibility. Current and projected hardware trends portend the so-called “memory wall” effect which observes that, in general, the processor speed to compute a floating point operation is increasing significantly faster than the speed to access random access memories. Further, for very large (supercomputer) computing installations, the high energy usage and dollar costs of memory chips are a major problem in future “exa-scale” computing. In the context of hybrid/accelerator-based computing that uses a cluster of graphical processing units (GPUs) and/or multi-core central processing units (CPUs), the reversible computation is even more relevant because the relatively abundant and inexpensive computational capacity (in comparison to memory capacity).

1.3 BLAS Overview

The BLAS interface is a de facto standard application programming interface (API) for basic linear algebra primitives [6, 10]. The BLAS API is a library of subroutines to perform a range of well-defined operations on vectors and matrices of real and complex numbers of varying precision. Interfaces are available in popular languages such as **C** and **FORTRAN**. Many implementations are available, most being vendor-optimized for the fastest execution on various platforms (e.g., ACML[2] from AMD Inc., GotoBLAS[8] from U. Texas, and CUBLAS[1] for NVIDIA GPUs). Here, we focus on two: ACML for CPU-based multi-core execution, and CUBLAS for CUDA-based execution on NVIDIA GPUs.

BLAS is broadly organized into three levels: Level 1 (L1), Level 2 (L2), and Level 3 (L3). L1 contains vector operations (and a few scalar operations, which are insignificant for reversibility), L2 contains matrix-vector operations, and L3 contains matrix-matrix operations. Vectors and matrices can be made of single-precision (S) or double-precision (D) real numbers, or of single-precision complex (C) or double-precision complex (Z) numbers.

1.4 BLAS Reversal

Reversibility of BLAS implies that, after any given computational sequence of invocations to BLAS routines in the normal (forward) direction is made, a reverse sequence (i.e., an opposite order of forward sequence) of corresponding inverse routines can be invoked to nullify the side-effects of the forward sequence on all variables of the program, effectively undoing the original forward sequence. We introduce a new library called Reversible BLAS (RBLAS) to enable such reversibility. For every routine `B()` in BLAS, a pair of routines `FB()` and `RB()` is defined in RBLAS such that `FB()` is the counterpart of `B()` and `RB()` is the perfect reversal of `FB()`, that is, executing `FB()` followed by `RB()` is a no-op as far as the program state is concerned. For example, corresponding to the BLAS routine `zROT()`, RBLAS includes two routines, `FzROT()` and `RzROT()`, as the forward and reverse pair. Internally, our RBLAS implementation makes use of the native BLAS for efficiently realizing the forward and reverse computation.

1.5 Numerical Reversibility

Numerical reversibility is a major concern with inverse computation-based reversal of linear algebra operations. The ideal reversal requirement is to recover all the bits of an overwritten matrix or vector, or, alternatively to recover to a user-defined (application-specific) precision that may not be as high as full machine precision. The precision of inverse computation varies with each subprogram, depending on its sequence of arithmetic instructions, and considerations of (non-)associativity and (non-)commutativity of floating point arithmetic. Here, we only focus on the runtime performance potential and do not provide any theoretical treatment of the arithmetic precision properties of reversal. A comprehensive treatment is relagated to future work. Some prior methods on mixed precision BLAS implementations [11] are useful starting points for such theoretical treatment. The issues and solutions in reproducibility of floating point arithmetic-based parallel applications is also relevant [9]. Also, entirely new approaches such as fixed point arithmetic that is reversible by design [14] may be useful in the longer term.

Nevertheless, it is unclear how significant the accuracy concern is in reality, with actual codes as opposed to pathological conditions. Indeed, with *non-pathological* inputs in our performance study of the RBLAS prototype interface and implementation, a verification of numerical accuracy in the experiments (see Section 3.7) shows zero loss of precision for vector subprograms, and negligible loss of precision for matrix-vector subprograms. Perceivable loss of precision seems to arise only for large matrix sizes for matrix-matrix subprograms.

1.6 Organization

The rest of the article is organized as follows. In Section 2, the high-level approach to the reversal of BLAS is presented. In Section 3, a detailed performance study is presented. The results are summarized and future work is described in Section 4.

2 RBLAS Approach

In this section, we present the overall reversal approach to arrive at RBLAS from BLAS. Key observations underlying the reversal method are presented first. This is followed by the design of the reversible interface of RBLAS that relaxes the forward-only interface of BLAS. The analysis of memory and computational complexities of memory-based checkpointing and inverse-based reversible computation methods for reversibility are then presented.

2.1 Key Observations for Reversal

The design of the reversible computation-based solution in RBLAS is based on the following observations.

- **Input/output Types** Formal arguments to each routine can be in (read-only), out (write), or inout (read/write). Thus, for reversal, it is sufficient to restore the out and inout arguments to pre-invocation values. For example, `zAXPY()` accepts as input two vectors x and y and a scalar α, and overwrites y with $\alpha x + y$. To reverse this routine, it is sufficient to restore y to its pre-forward values.
- **Constructive versus Destructive Routines** When considered from a reversibility standpoint, we observe that almost all BLAS routines are "constructive" in nature, that is, information is not destroyed per se by each routine by itself. Hence, it becomes possible to reverse them via recreation of previous state from current state, rather than by saving to and restoring from memory. Thus, we were able to develop reversible computation-based reversals for almost all routines, with one exception. Only one L1 routine, namely `xCOPY`, is destructive in nature because its invocation blindly overwrites the pre-invocation values of a vector. The reversibility of this routine can be addressed in three different ways: (1) checkpointing can be used to save a copy of the vector that would be overwritten, (2) require the overwritten vector to contain all zero values, (3) prohibit this routine from being available in a reversible setting. The first choice is the easiest backward-compatible solution, but also the least desirable with respect to efficiency of reversal. The second is an interface specification which is reasonable to introduce because the blind overwriting of a vector has the semantics of reuse of a memory location for two different purposes, which can be avoided by explicit separation by the programmer. The third is in fact a more sensible approach in the longer-term because a copy operation in fact can be translated into an equivalent "variable renaming" or relabeling operation. For simplicity and short-term backward-compatibility, we adopt the first solution.
- **Reversible Ranges of Arithmetic Constants** Similar considerations are applied to other routines in which some extreme values result in destructive overwriting of values. For example, the `xSCAL` operation $x \leftarrow \alpha x$ with $\alpha = 0$ blindly resets the x vector. Thus, reversible computation is used only when α is non-zero, and checkpointing is employed when α is zero. Considering the precision limitations of computer arithmetic, this condition needs to be generalized to include precision-defined limits to account for underflow and overflow conditions of multiplication, such that $\epsilon \leq |\alpha| \leq \mathcal{E}$, where ϵ and $\mathcal{E}$ are platform-specific small and large constants outside of which multiplication operations lose reversibility.
- **Side-effects** Only those routines that have side-effects need to be reversed (side-effects are any changes to program state outside the routine, such as global variables). In the **FORTRAN** interface of BLAS, only those specified as `SUBROUTINE` need to be considered for reversal, while those specified as `FUNCTION` are not applicable for reversal (since they do not have any effect on memory that needs to be undone). Thus, for example, while `xROT` (rotation of a vector) is a subroutine that needs reversal, `xDOT` (dot product of two vectors) is a function that has no memory side-effects and hence does not need reversal.

2.2 RBLAS Forward and Reverse Routine Pairs

The RBLAS routines (forward and reverse) are shown in the following tables: L1 routines in Table 1; L2 routines in three groups – group 1 in Table 2, group 2 in Table 3, and group 3 in Table 4; L3 routines in Table 5.

Call	Forward	**Reversal**	Condition	Types	Notes
`xCOPY`	$y \leftarrow x$	$y \leftarrow y'$	y' is known (e.g., 0)	S,D,C,Z	Destructive copy
`xSWAP`	$x \leftrightarrow y$	$y \leftrightarrow x$	None	S,D,C,Z	Self-inverse
`xSCAL`	$x \leftarrow \alpha x$	$x \leftarrow \frac{1}{\alpha}x$	$\epsilon \leq \|\alpha\| \leq \mathcal{E}$	S,D,C,Z	Bounded scale
`xAXPY`	$y \leftarrow \alpha x + y$	$y \leftarrow -\alpha x + y$	None	S,D,C,Z	Constructive

Table 1. Level 1

Call	Forward	**Reversal**	Condition	Types	Notes
`xGEMV`	$y \leftarrow \alpha Ax + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Ax + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	S,D,C,Z	General matrix
`xGBMV`	$y \leftarrow \alpha Bx + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Bx + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	S,D,C,Z	Banded matrix
`xHEMV`	$y \leftarrow \alpha Ax + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Ax + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	C,Z	General hermitian
`xHBMV`	$y \leftarrow \alpha Bx + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Bx + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	C,Z	Banded hermitian
`xHPMV`	$y \leftarrow \alpha Px + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Px + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	C,Z	Packed hermitian
`xSYMV`	$y \leftarrow \alpha Ax + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Ax + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	S,D	Symmetric matrix
`xSYMV`	$y \leftarrow \alpha Bx + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Bx + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	S,D	Symmetric banded
`xSPMV`	$y \leftarrow \alpha Px + \beta y$	$y \leftarrow -\frac{\alpha}{\beta}Px + \frac{1}{\beta}y$	$\epsilon \leq \|\beta\| \leq \mathcal{E}$	S,D	Symmetric packed

Table 2. Level 2 Group 1

Call	Forward	**Reversal**	**Inverse**	Condition	Types	Notes
`xTRMV`	$x \leftarrow Ax$	$x \leftarrow A^{-1}x$	`xTRSV`	Non-singular A	S,D,C,Z	General triangular
`xTBMV`	$x \leftarrow Bx$	$x \leftarrow B^{-1}x$	`xTBSV`	Non-singular B	S,D,C,Z	Banded triangular
`xTPMV`	$x \leftarrow Px$	$x \leftarrow P^{-1}x$	`xTPSV`	Non-singular P	S,D,C,Z	Packed triangular
`xTRSV`	$x \leftarrow A^{-1}x$	$x \leftarrow Ax$	`xTRMV`	Non-singular A	S,D,C,Z	General triangular
`xTBSV`	$x \leftarrow B^{-1}x$	$x \leftarrow Bx$	`xTBMV`	Non-singular B	S,D,C,Z	Banded triangular
`xTPSV`	$x \leftarrow P^{-1}x$	$x \leftarrow Px$	`xTPMV`	Non-singular P	S,D,C,Z	Packed triangular

Table 3. Level 2 Group 2, for non-singular A, B, P or their transposes/conjugates

The new reversible application programming interface (API) retains identical formal arguments for all subprograms. However, new subprogram names

Call	Forward	**Reversal**	Types	Notes
`xGER`	$A \leftarrow \alpha xy^T + A$	$A \leftarrow -\alpha xy^T + A$	S,D	General
`xGERU`	$A \leftarrow \alpha xy^T + A$	$A \leftarrow -\alpha xy^T + A$	C,Z	General
`xGERC`	$A \leftarrow \alpha xy^H + A$	$A \leftarrow -\alpha xy^T + A$	C,Z	General
`xHER`	$A \leftarrow \alpha xx^H + A$	$A \leftarrow -\alpha xx^H + A$	C,Z	Hermitian
`xHPR`	$A \leftarrow \alpha xx^H + A$	$A \leftarrow -\alpha xx^H + A$	C,Z	Packed Hermitian
`xHER2`	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	$A \leftarrow -\alpha xy^H - y(\alpha x)^H + A$	C,Z	Hermitian
`xHPR2`	$P \leftarrow \alpha xy^H + y(\alpha x)^H + P$	$P \leftarrow -\alpha xy^H - y(\alpha x)^H + P$	C,Z	Packed Hermitian
`xSYR`	$Y \leftarrow \alpha xx^T + Y$	$Y \leftarrow -\alpha xx^T + Y$	S,D	Symmetric
`xSPR`	$P \leftarrow \alpha xx^T + P$	$P \leftarrow -\alpha xx^T + P$	S,D	Packed
`xSYR2`	$Y \leftarrow \alpha xy^T + \alpha yx^T + Y$	$Y \leftarrow -\alpha xy^T - \alpha yx^T + Y$	S,D	Symmetric
`xSPR2`	$P \leftarrow \alpha xy^T + \alpha yx^T + P$	$P \leftarrow -\alpha xy^T - \alpha yx^T + P$	S,D	Packed

Table 4. Level 2 Group 3

Call	Forward	**Reversal**	Condition	Types	Notes
`xGEMM`	$C \leftarrow \alpha AB + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} AB + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	S,D,C,Z	General
`xSYMM`	$C \leftarrow \alpha YB + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} YB + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	S,D,C,Z	Symmetric
	$C \leftarrow \alpha BY + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} BY + \frac{1}{\beta} C$			
`xHEMM`	$C \leftarrow \alpha HB + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} HB + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	C,Z	Hermitian
	$C \leftarrow \alpha BH + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} BH + \frac{1}{\beta} C$			
`xSYRK`	$C \leftarrow \alpha YY^T + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} YY^T + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	S,D,C,Z	Symmetric
	$C \leftarrow \alpha Y^TY + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} Y^TY + \frac{1}{\beta} C$			
`xHERK`	$C \leftarrow \alpha AA^H + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} AA^H + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	C,Z	Hermitian
	$C \leftarrow \alpha A^HA + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} A^HA + \frac{1}{\beta} C$			
`xSYR2K`	$C \leftarrow \alpha YB^T + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} YB^T + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	S,D,C,Z	Symmetric
	$C \leftarrow \alpha Y^TB + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} Y^TB + \frac{1}{\beta} C$			
`xHER2K`	$C \leftarrow \alpha AB^H + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} AB^H + \frac{1}{\beta} C$	$\epsilon \le \lvert\beta\rvert \le \mathcal{E}$	C,Z	Hermitian
	$C \leftarrow \alpha A^HB + \beta C$	$C \leftarrow -\frac{\alpha}{\beta} A^HB + \frac{1}{\beta} C$			
`xTRMM`	$B \leftarrow \alpha AB$	$B \leftarrow \frac{1}{\alpha} A^{-1}B$	$\epsilon \le \lvert\alpha\rvert \le \mathcal{E}$	S,D,C,Z	Triangular
`xTRSM`	$B \leftarrow \alpha A^{-1}B$	$B \leftarrow \frac{1}{\alpha} AB$	$\epsilon \le \lvert\alpha\rvert \le \mathcal{E}$	S,D,C,Z	Triangular

Table 5. Level 3 with options to specify transposes and conjugates of supplied matrices

are used in order to enable reversible operation: for every original BLAS routine `routine()`, a new pair of forward and reverse routines is defined with the following naming convention:

`[f|r][s|d|c|z]_routine()`

The notation `[x|y]` denotes choice between `x` and `y`. The prefix `f` denotes forward-mode and `r` denotes reverse-mode of the BLAS subprogram named `routine()`. The conventional data types codes are used: `s` for single precision floating point, `d` for double precision floating point, `c` for single precision complex, and `z` for double precision complex numbers. For example, `fzrot()` and `rzrot()` are the forward and reverse mode RBLAS subprograms corresponding to the BLAS subprogram `zrot()`.

2.3 Runtime and Memory Analysis

Since checkpointing (CP) relies on making a copy of values before they are overwritten, its memory complexity for any given BLAS routine is proportional to the byte size of values being modified in that routine. For reversal, CP runtime complexity remains proportional to the memory size because the saved value is simply copied back. On the other hand, RBLAS based on reversible computation (RC) incurs zero *additional* cost (memory or runtime) over the underlying BLAS cost for forward execution, but incurs *computation time* cost for reversal.

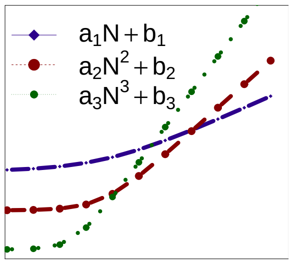


Fig. 1. Illustration of varying cut-off points across different complexities

Figure 1 illustrates this performance cut-off point for different complexities, where a higher order complexity of $O(N^2)$ (say, for recovering a vector via RC) may in fact perform better than a lower order complexity of $O(N)$ (say, for saving to memory via CP), due to the computer-specific constants of fixed and variable costs per computational or memory operation. Due to this essential nature of the difference between reversible computation and checkpointing, the runtime complexity and memory complexity of the two approaches differ, both in forward execution as well as in reversal. Since the amortized cost per memory operation is typically orders of magnitude larger than that per computational (floating point) operation, non-linear tradeoff points arise in the choice between the two methods.

- In L1 with vectors of size N, the memory cost of CP is $O(N)$, while that for RC is 0. Also, the forward overhead is $O(N)$ for CP, but 0 for RC. The reversal overhead is $O(N)$ for both, but with vastly different constants originating from memory speeds for CP and computational speeds for RC.
- For general (dense) matrices in L2 Group 1 (e.g., `xGEMV`), the memory cost of CP is $O(N)$ to save a vector, while that for RC is 0. The forward overhead is also $O(N)$ for CP, but 0 for RC. However, the reversal overheads are vastly different: $O(N)$ with memory speed constants for CP, and $O(N^2)$ with computational speed constants for RC. For routines on banded matrices (e.g., `xGBMV`) whose number of bands is $O(1)$, the reversal costs for RC becomes $O(N)$, which makes it significantly faster than CP.

– In L2 Group 2, the `xGER` routine has a large memory overhead of $O(N^2)$ with CP to save and restore the matrix A, whereas RC incurs the same quadratic complexity albeit at computational speeds as opposed to memory speeds. This results in large savings for reversibility in both memory usage and runtime.
– The L3 routines are $O(N^2)$ in data size but $O(N^3)$ in computational cost, which makes it expensive for RC during reversal. Forward execution is still slightly better with RC because no data is saved in forward mode.

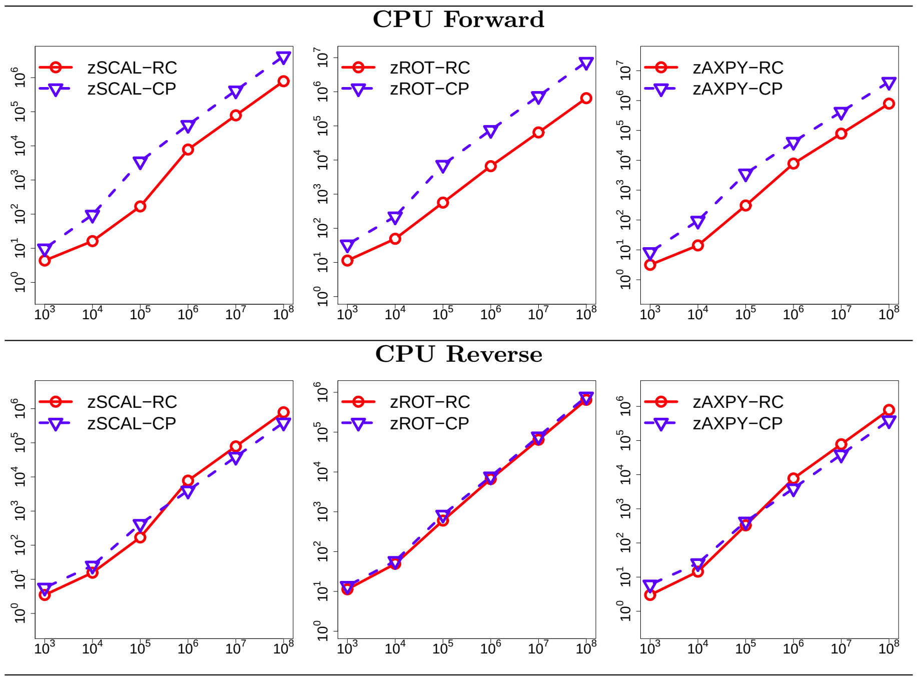


Fig. 2. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L1 routines on an N-sized double-complex (**z**) vector using ACML on a CPU.

3 Performance Study

In this section, we present a detailed performance study to understand the relative gains of reversible comptuation versus checkpointing in RBLAS. The implementation is described followed by the experiment setup and a description of the hardware and software employed.

Two popular computational platforms are studied: one which is based on a traditional high performance processor and the other based on an accelerator

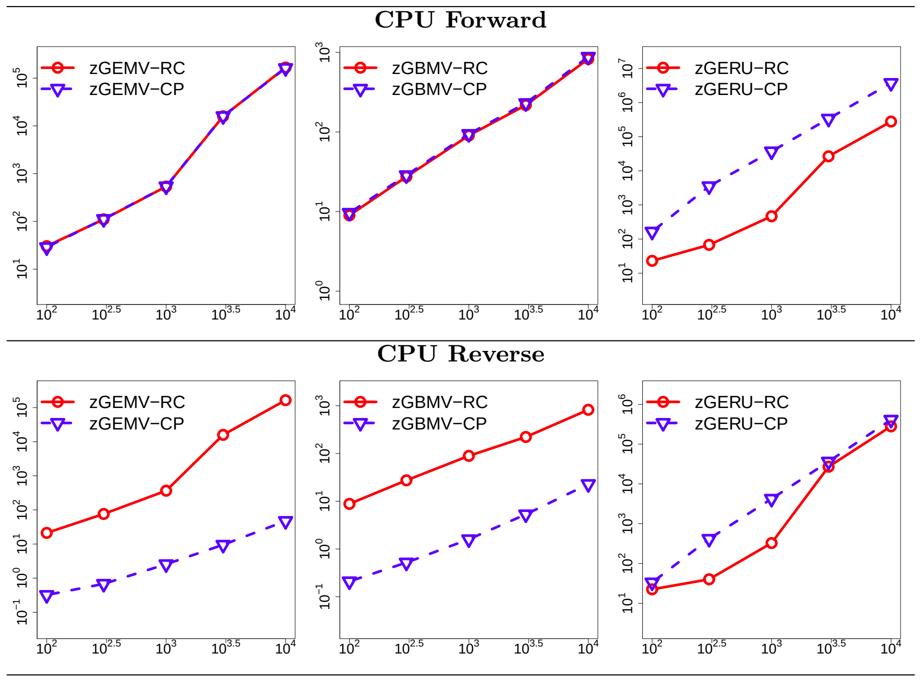


Fig. 3. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L2 routines on an N-sized double-complex ($\mathbf{z}$) vector using ACML on a CPU.

(graphical processing unit or GPU) architecture. For the high performance processor, the memory subsystem performance is studied in terms of cache behavior of RC and CP to explain the large runtime gain of RC compared to CP. For both platforms, the overall speed ratios of RC over CP are compared. Finally, numerical precision data are presented showing low loss of precision due to reversibility on most subprograms.

3.1 Implementation

We have implemented a prototype of the RBLAS library that includes all the BLAS routines. Further, it is organized in such a way as to be able to utilize any native BLAS implementation as an efficient building block. The portable implementation currently supports the ACML BLAS implementation on multi-core CPUs, and the CUBLAS implementation for CUDA-based NVIDIA GPUs. Additional native BLAS implementations can also be easily incorporated in the future. While our RBLAS implementation supports all BLAS routines, due to space constraints, the performance study here only focuses on a few routines (3 routines in each level).

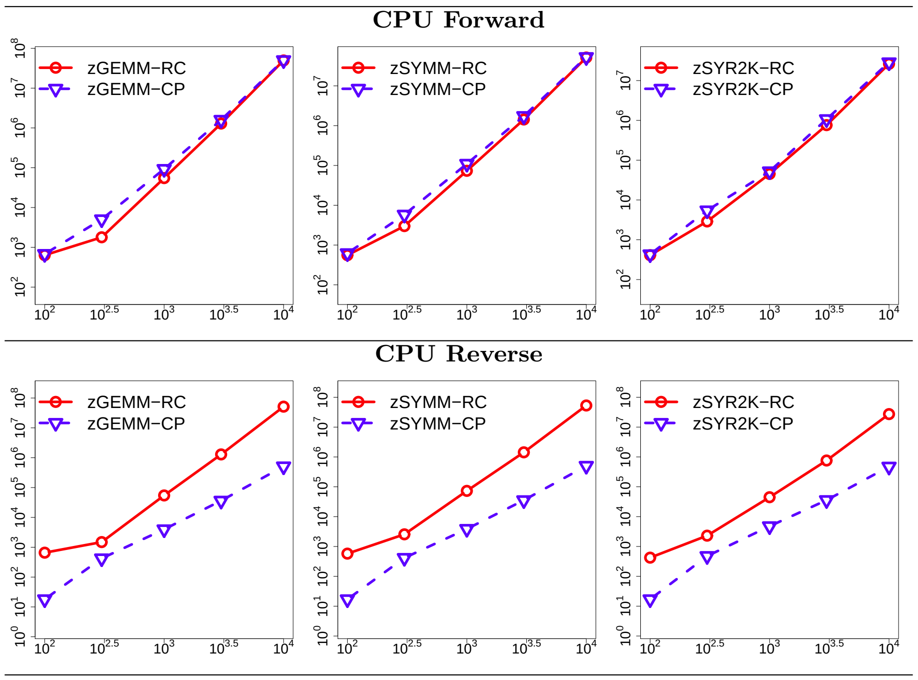


Fig. 4. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L3 routines on an $N \times N$-sized double-complex (`z`) matrix using ACML on a CPU.

3.2 Experiment Setup

The performance of RBLAS with checkpointing (CP) and reversible computation (RC) implementations are tested using a benchmark. In the benchmark, for every BLAS call `B()`, n invocations of `FB()` followed by n invocations of `RB()` are made. The average time for each forward and reverse invocation is measured. This is repeated for r trials, to account for runtime variations. In the presented charts, the observations are from $n = 10$ and $r = 10$. Matrices are filled with values generated as $a + Rb$ where a and b are real numbers and $R \in [0..1]$ is a uniformly distributed random number. In the presented charts, $a = b = 1000$. The accuracy of any routine `B` is measured by comparing the orginal input to `FB()` and the restored input after reversal by `RB()`. The accuracy measure is the root mean square (RMS) difference between the input and restored input values. Thus, an RMS value of e implies a *relative error* (on average) of $2e/((a + b))$ in the restored value. In the experiments (Figure 11), an RMS value of $e = 0.001$ with $a = b = 1000$ represents a relative error of $1e - 6$.

BLAS routines are typically used in various combinations in applications. In order to isolate the performance effects on a per-routine basis, here we focus on an experiment design in which the same routine is called multiple times. Thus, if A, B, and C are three routines, we focused on runs R_A, R_B, and R_C, where $R_A = A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \cdots \rightarrow A_n$, $n \geq 1$, is a sequence of n invocations to the

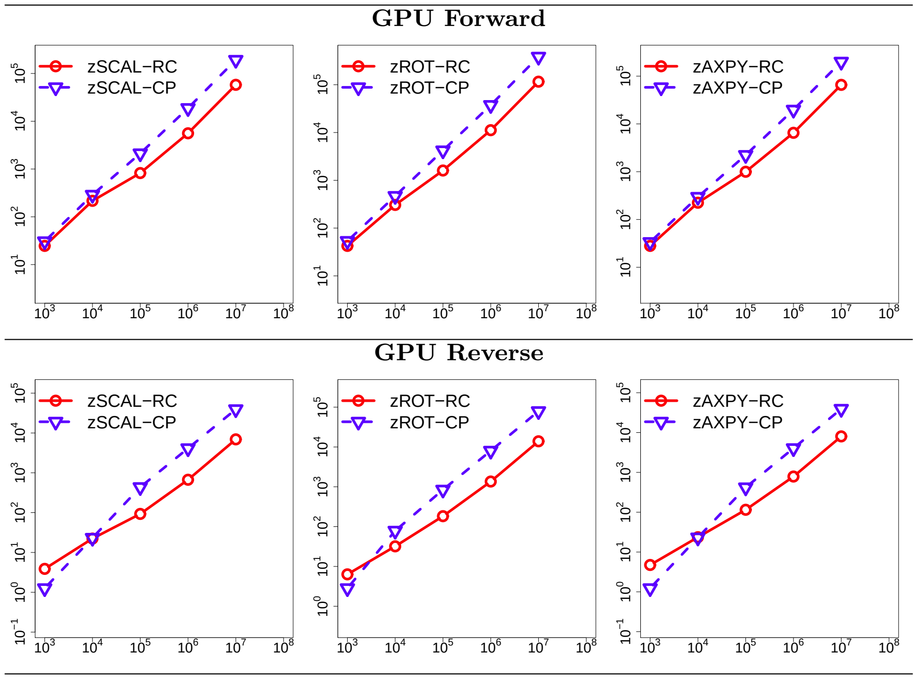


Fig. 5. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L1 routines on an N-sized double-complex (**z**) vector using CUBLAS on a GPU.

routine A, and so on. Each sequence is fully reversed before another sequence is initiated. Thus, after R_A is executed, its inverse, namely, $R_A^{-1} = A_n^{-1} \rightarrow \cdots A_3^{-1} \rightarrow A_2^{-1} \rightarrow A_1^{-1}$, is executed prior to making any other calls, such as R_B or R_C. The performance of combination of different calls together is in fact interesting, which we will pursue in future work.

In each sequence of n invocations to a routine, the output of each invocation is used as input of its subsequent invocation. This circumvents the need for input erasure; however, in other cases, if inputs are unrelated and need to be reset across calls, there would be an erasure cost, which may need to be added in an expanded performance study.

3.3 Hardware and Software

All experiments were performed on a multi-core CPU platform and a GPU platform.

– **GPU (CUBLAS)**: The test system contains an NVIDIA GeForce GTX 580 GPU, used with the CUDA compilation tools (release 4.2, V0.2.1221) and CUBLAS library. Each GPU has 16 multiprocessors, supporting 32 CUDA cores per streaming multiprocessor, with a total global memory of 3.2 GB,

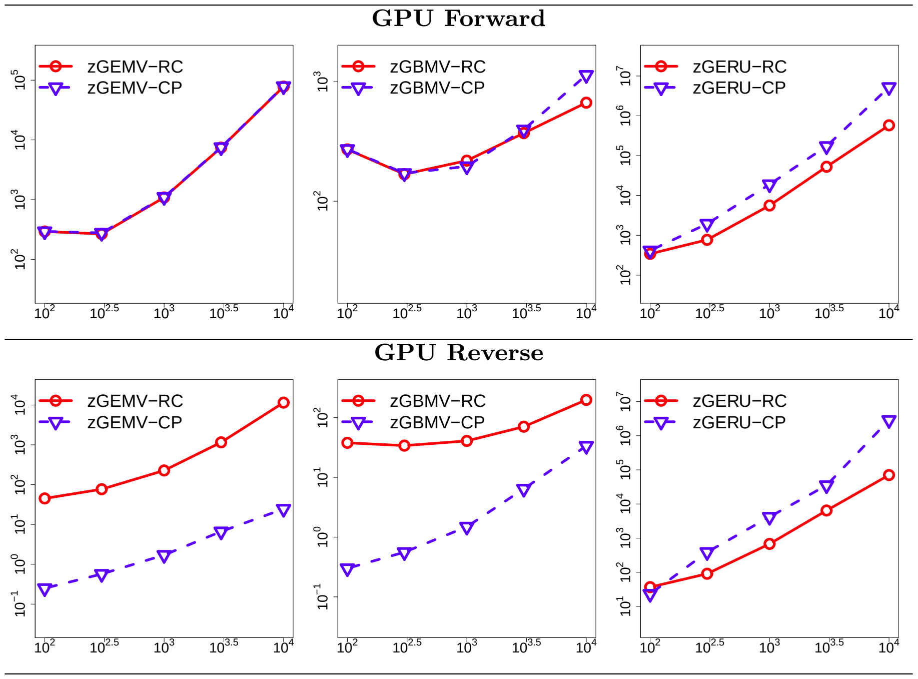


Fig. 6. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L2 routines on an N-sized double-complex (`z`) vector using CUBLAS on a GPU.

and shared memory of 49 KB per block. The system runs Linux using 16 GB of memory and an AMD Phenom(tm) II X6 1100T Processor with 6 cores each clocking at 3.3GHz.

– **CPU (ACML)**: The test system is a Linux machine running two 16-core (32 cores in total) AMD(tm) Opteron 6276 processors at 2.3 Ghz sharing 256GB of memory. AMD's `libacml_fma4_mp` library with multithreading support was used in our benchmarks and a maximum of 16 threads were exercised in our benchmark runs.

3.4 ACML (CPU) Runtime Performance

The CPU (ACML) runtime performance is shown in Figure 2 for L1, Figure 3 for L2, and Figure 4 for L3.

A significantly faster execution of RC is observed for all L1 routines, with some faster than CP by an order of magnitude. Reversal of RC, on the other hand, is on-par with CP. With L2, the forward costs are similar for RC and CP; as expected, the reversal costs are higher for RC on GEMV and GBMV. With GERU, both forward and reverse are significantly faster with RC, but the gains for reversal cost diminish with increasing N. For L3, the lower cost due to elimination of memory copying in forward execution is evident because memory

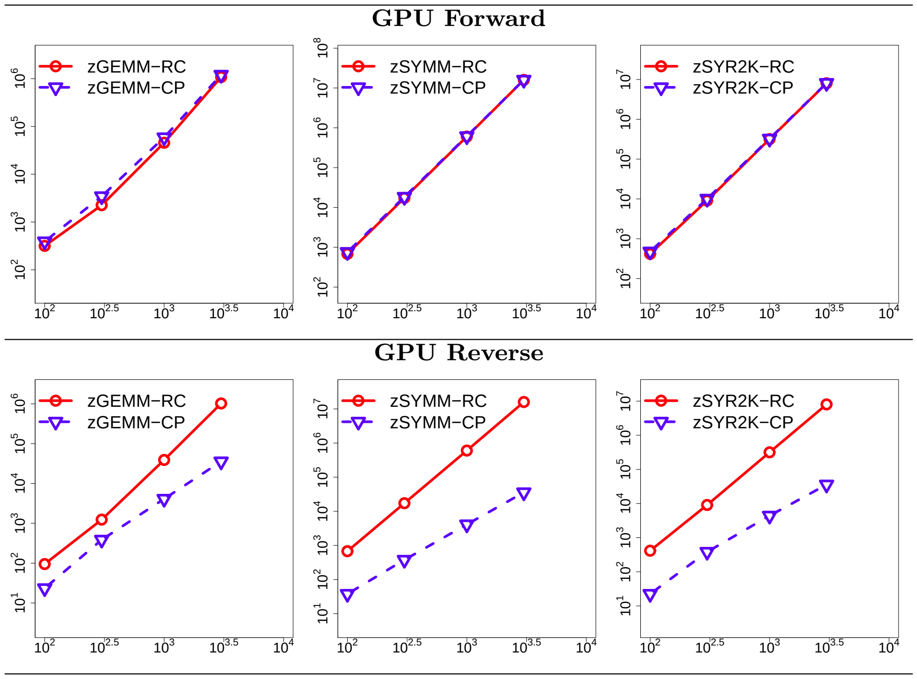


Fig. 7. Forward and reverse times (μsec) as a function of N (x-axis) for RBLAS L3 routines on an $N \times N$-sized double-complex (**z**) matrix using CUBLAS on a GPU.

size of $O(N^2)$ is significantly large. However, CP has much lower reversal cost. Thus, for L3, RC can be used to reduce memory usage, albeit at a significant runtime cost (only during reversal) for than CP.

3.5 Memory Cache Effects

The performance gain of RC over CP can be explained by the superior memory subsystem behavior of RC compared to that of CP. This can be verified using low-level processor performance counters for measuring the number of memory accesses and cache-misses at different cache levels (L1 and L2 [1] and at the Translation Lookaside Buffer (TLB)). These counters are plotted for the subprograms in Figure 8 for L1 cache, Figure 9 for L2 cache, and Figure 10 for TLB. Since even a slightly better cache performance (lower number of accesses and/or misses) results in a superior runtime, it is clear that a large gap between the cache numbers between RC and CP accounts for the overall runtime differences.

[1] The overlap in conventional terminology of “L1” and “L2” between BLAS levels and cache levels is unfortunately unavoidable.

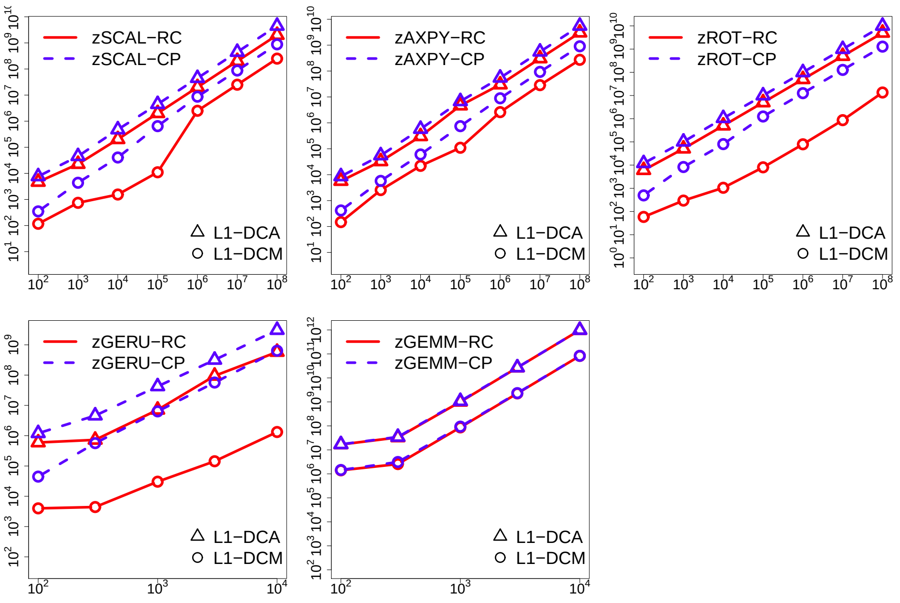


Fig. 8. L1-Cache performance: Number of cache misses and accesses (y-axis) against matrix dimension (x-axis)

3.6 CUBLAS (GPU) Runtime Performance

The GPU (CUBLAS) runtime performance is shown in Figure 5 for L1, Figure 6 for L2, and Figure 7 for L3.

On a GPU, two factors weigh heavily in favor of RC: (1) very fast computations with heavy parallel execution within the GPU, thus helping reliance on computation for reversal by RC, and (2) very large overhead of copying memory between GPU (device) and CPU (host) memories, thus hurting the reliance of CP on memory. The trends of the differences are similar to that for CPU, except that for all calls on which RC is faster with the CPU, RC is observed to fare even better on the GPU. Overall, RC performs as well as (and sometimes significantly better than CP) for all forward execution. During reversal, RC works better than CP for L1 and for L2 Group 3, on both CPUs and GPUs, but even markedly better on the GPUs. However, for reversal of L2 Group 1 and Group 2, and for L3, CP fares much better. In all cases, RC does not need any additional memory, whereas CP incurs additional memory needs.

3.7 Numerical Error and Empirical Results of Accuracy

A verification of numerical accuracy in the experiments is obtained by comparing the values before forward execution and after reversed execution. The accuracies obtained with reversals of several routines for both ACML and CUDA versions of RBLAS are shown in Figure 11.

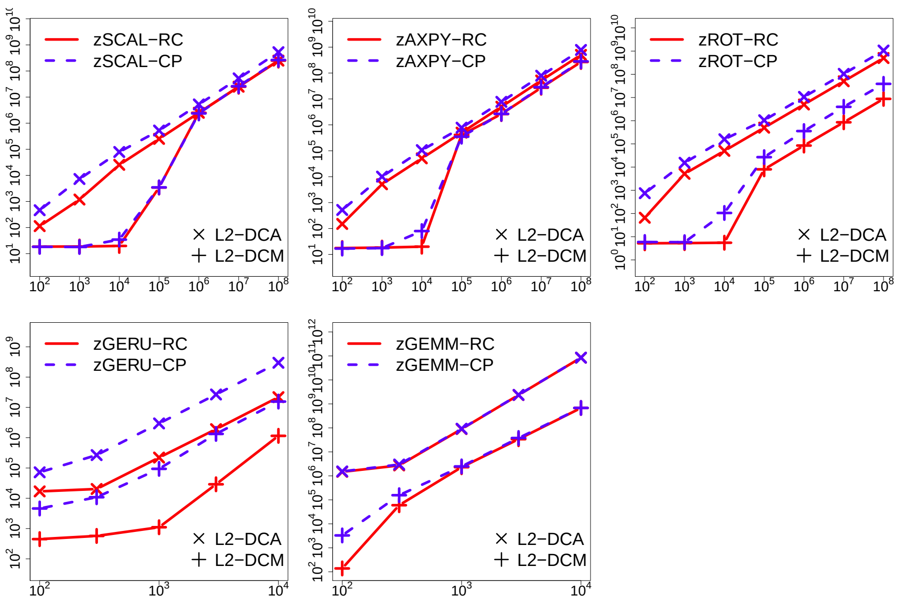


Fig. 9. L2-Cache performance: Number of cache misses and accesses (y-axis) against matrix dimension (x-axis)

With the normalized values used for matrix and vector elements in the experiments, the reversal of BLAS level 1 subprograms is essentially lossless upto vector sizes as large as 10^7; the RMS error is neglible even at a larger vector size of up to 10^8. Reversal of BLAS level 2 subprograms is observed to be lossless until square matrix sizes of $10^{3.5}$, and continue to have negligible loss even at 10^4. For BLAS level 3, as expected, lossless reversal is achieved on a slightly smaller matrix size of 10^3–$10^{3.5}$ However, lossy reversal sets in at the next size of 10^4.

4 Summary and Future Work

The design, analysis, and performance study of a reversible library for basic linear algebra subprograms has been presented. Two major reversibility approaches have been explored, namely, checkpointing and reversible computation. Reversible computation (RC) is found to be perfectly suited to overcome the detrimental memory effects due to RC reliance of computation (which is "cheap") as opposed to on memory (which is "expensive" in both energy and cost). The gains are found to be especially pronounced on GPUs. RC is found to be over an order of magnitude faster for L1 and some L2 of BLAS, *both in forward and reverse*; and never slower than checkpointing in forward. For GEMV, GEMM, forward is slightly faster but the primary benefit is the avoidance of memory needs for reversal. Savings are significant for larger matrices because

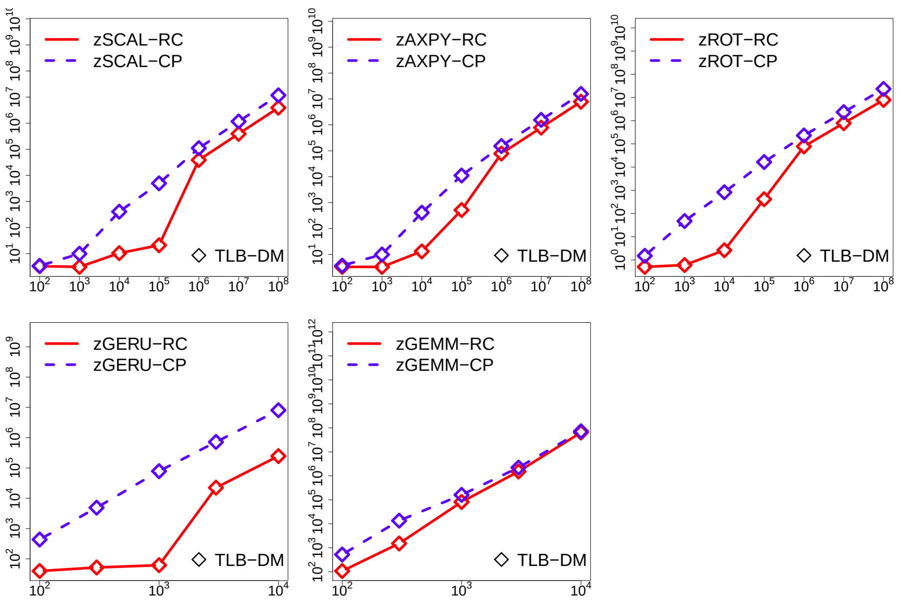
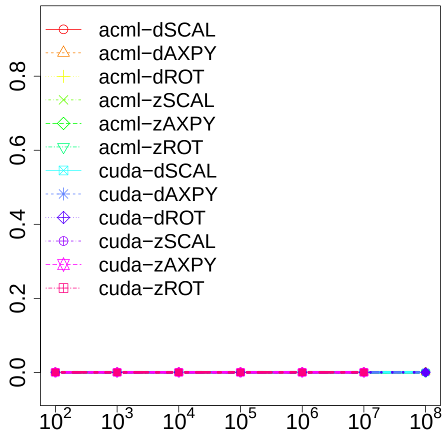


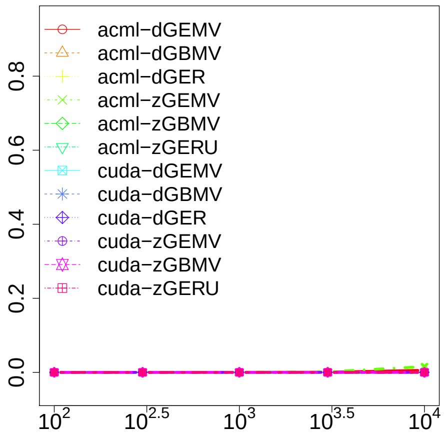
Fig. 10. TLB performance: Number of TLB misses (y-axis) against matrix dimension (x-axis)

memory use can be large; in fact, memory limits are encountered for $N = 10^4$ on GPU.

Future work includes the exercise of RBLAS in an actual application (scientific applications) and exploration of fault tolerant execution at very large scale. Since applications include invocations to various combinations of the BLAS routines, it would be interesting to perform benchmarking experiments on the performance of RBLAS on different combinations of routines.

A comprehensive theoretical treatment is needed on the precision issues and computer arithmetical aspects of reverse code for each BLAS routine on a case-by-case basis, especially to account for unexpected or pathological cases. Of interest is also hardware-level realization of reversible arithmetic in general, and specialized reversible linear algebra in particular. Additionally, it would be interesting to follow the reversible application programming interface (API) methodology to explore reversal of other popular computational libraries at the software-level.





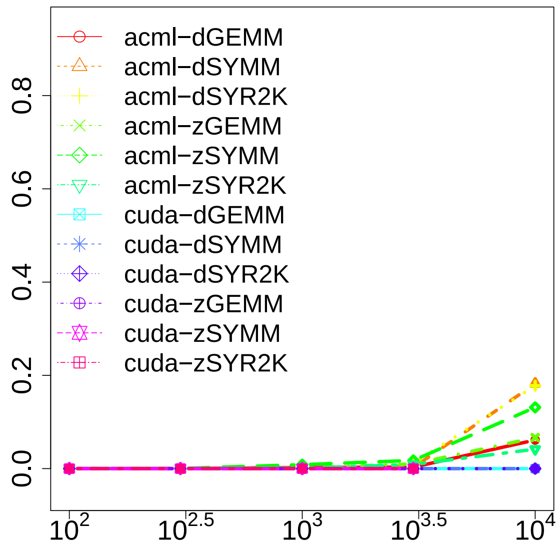


Fig. 11. Root Mean Square (RMS) values denoting numerical deviation of values before and after reversal (of N values for L1 and L2, and N^2 values for L3), for increasing N (x-axis), with ACML on a CPU and CUBLAS on a GPU

References

1. Cublas: Common Unified Data Architecture Basic Linear Algebra Subprograms. `http://developer.nvidia.com/cublas` (2012)
2. Acml: Advanced micro devices core math library. `http://developer.amd.com` (2013)
3. Barnes, P., Carothers, C., Jefferson, D., LaPre, J.: Warp speed: Executing time warp on 1,966,080 cores. In: Proceedings of the ACM SIGSIM Principles of Advanced Discrete Simulation (2013)
4. Besseron, X., Gautier, T.: Impact of over-decomposition on coordinated checkpoint/rollback protocol. In: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 7156, pp. 322–332. Springer Berlin Heidelberg (2012)
5. Bessho, N., Dohi, T.: Comparing checkpoint and rollback recovery schemes in a cluster system. In: Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science, vol. 7439, pp. 531–545. Springer Berlin Heidelberg (2012)
6. Dongarra, J., Duff, I., DuCroz, J., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (1989)
7. Frank, M.: Introduction to reversible computing: Motivation, progress, and challenges. In: International Workshop on Reversible Computing (Special Session at ACM Computing Frontiers) (2005)
8. Goto, K., Van De Geijn, R.: High-performance implementation of the level-3 blas. ACM Trans. Math. Softw. 35(1), 4:1–4:14 (Jul 2008)
9. He, Y., Ding, C.: Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. Springer Journal of Supercomputing 18, 259–277 (2001)
10. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic linear algebra subprograms for fortran usage. ACM Transactions on Mathematical Software (5), 308–325 (1979)
11. Li, X., Demmel, J., Baile, D., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kapur, A., Martin, M.C., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision blas. ACM Transactions on Mathematical Software 28(2), 206–238 (2002)
12. Perumalla, K., Park, A.: Reverse computation for rollback-based fault tolerance in large parallel systems. Cluster Computing pp. 1–11 (2013), `http://dx.doi.org/10.1007/s10586-013-0277-4`
13. Perumalla, K., Park, A., Tipparaju, V.: Discrete event execution with one-sided and two-sided gvt algorithms on 216,000 processor cores. ACM Transactions on Modeling and Computer Simulation (to appear) (2014)
14. Perumalla, K.S.: Introduction to Reversible Computing. Computational Science Series, Chapman Hall/CRC Press (2013), ISBN 978-1439873403