A Mixed Precision Randomized Preconditioner for the LSQR Solver on GPUs

Vasileios Georgiou[1], Christos Boutsikas[2], Petros Drineas[2], and Hartwig Anzt[1,3]

[1] Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Germany {vasileios.georgiou, hartwig.anzt}@kit.edu
[2] Purdue University, West Lafayette, USA {cboutsik,pdrineas}@purdue.edu
[3] Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, USA

Abstract. Randomized preconditioners for large-scale regression problems have become extremely popular over the past decade. Such preconditioners are known to accelerate large-scale regression solvers both from a theoretical and a practical perspective. In this paper, we present a mixed precision randomized preconditioner for LSQR solvers, focusing on overdetermined, dense least squares problems. We implement and evaluate our method on GPUs and we demonstrate that it outperforms the standard double precision version of randomized, preconditioned LSQR by up to 20% on the NVIDIA A100. We present extensive numerical experiments utilizing the half-precision and tensorcore units to demonstrate that, in many cases, constructing the preconditioner in *reduced precision* does not affect the convergence of LSQR solvers. This leads to important speedups without loss of accuracy.

Keywords: Mixed Precision · Randomized Preconditioners · Over-determined Least Squares · LSQR · GPUs

1 Introduction

Solving overdetermined least squares problems is a common yet computationally expensive challenge in scientific computing. Standard approaches include a variety of direct and iterative methods. The former rely either on computing the QR factorization of the input matrix or on solving the so-called normal equations. Orthogonalization methods used for factorization utilize variants of the Gram-Schmidt algorithm [5,8,10,33], Householder reflectors [21,32,41], or Givens rotations [7,25]. Additionally, the Cholesky factorization is often used to solve the normal equations [42]. Among the iterative solvers that have been proposed to tackle least squares, LSQR [38] is one of the most popular methods mainly because of its numerical robustness. Alternatives include GMRES [28,37] and CGLS [9,23,35]. The main factor determining the runtime of iterative methods is the number of iterations required in order to converge to the specified tolerance. Several techniques for transforming the original problem to one which is easier to solve, i.e., one that requires fewer iterations, have been developed over the years. The most important family of such techniques are the *preconditioning* methods, which are essential in both theory and practice of least-squares solvers.

Preconditioning includes a spectrum of techniques ranging from those tailored to a specific application to general purpose, "black-box" methods, which are broadly applicable but more inefficient in special cases. It is worth noting that constructing the preconditioner could be relatively expensive compared to the overall runtime of the solver, which often argues against using the "tailored" approach, unless the problem input has very specific characteristics. Iterative least squares solvers are often popular for solving *sparse* problems, which has led to a variety of preconditioners based on sparse approximations [16,17] and incomplete factorizations [6,11,18]. On the other hand, there are significantly fewer results for preconditioning *dense* overdetermined least squares problems. Over the past decade, randomized "black-box" preconditioners have emerged as a robust way to solve large-scale regression problems, outperforming dense QR-based approaches [3,4].

Randomization has often been used as a resource in tackling data-intensive linear algebra problems. A popular example is performing principal components analysis (PCA) on massive datasets by sketching or sampling the input matrix. Another example has been randomized preconditioning, which first creates a "sketch" of the input matrix that is used to compute the preconditioner [3]. Theoretical analyses of such methods provide error guarantees that depend on the number of samples or the size of the sketch, which are typically independent of the data size. Such methods effectively reduce the dimensionality of the original data, a process that is somewhat akin to processing a noisy version of the input matrix. This makes randomized linear algebra algorithms perfect candidates for incorporating elements of mixed precision computations, taking advantage of modern hardware to achieve speedups without significant loss of accuracy.

The introduction of native support for 16-bit precision formats on modern GPUs has led to increased interest in mixed-precision versions of numerical methods. Mixed precision algorithms use at least two different precision formats, performing the most computationally intensive steps in lower precision to benefit from faster execution on hardware accelerators. Another way to speedup memory-bound computations is by communicating the data in reduced precision while executing the computations in the original (or higher) precision [2,27]. This is beneficial since for memory-bounded problems the cost of communicating data across devices dominates the overall runtime.

Even though early work on mixed precision numerical algorithms was mostly focused on the solution of linear systems of equations, this has changed over time. Some notable mixed precision methods for solving least squares problems include [15,29], as well as iterative refinement approaches [14] and scaling techniques [30] for recovering (at least partially) the accuracy which is inevitably lost when converting to lower precision. There has also been some work on mixed precision preconditioners in [22,26]. However, to the best of our knowledge, there has not been much progress in the development and implementation of mixed precision randomized preconditioners for least-squares problems.

In this paper, we address the aforementioned gap. We develop a mixed precision randomized preconditioner to be used with our novel LSQR implementation for solving dense overdetermined least squares problems on GPUs. Despite constructing the preconditioner in lower precision, our results show that this loss in precision does not negatively affect the convergence of LSQR. This leads to significant speedups of up to 140% in terms of the runtime required for constructing the preconditioner, and up to to 20% in terms of the overall runtime, without any loss of accuracy. In our analysis we provide some insights, regarding the factors that affect the performance of the preconditioner. Both the randomized preconditioner and the LSQR solver are implemented in C++ using the MAGMA and the CUDA runtime libraries and operate exclusively on the GPU. This is the first implementation and systematic evaluation of mixed-precision, randomized preconditioned LSQR on GPUs.

The rest of the paper is structured as follows: In Section 2, we provide some background on randomized preconditioners. In Section 3, we explain the details of the implementation of our method, and in Section 4 we showcase performance results from our experiments on different datasets. Lastly, in Section 5, we summarize our findings and discuss potential extensions.

2 Background

Given a coefficient matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, and a right-hand side vector $\boldsymbol{b} \in \mathbb{R}^m$, the overdetermined ($m \geq n$) least-squares (LS) solution is the vector $\mathbf{x}^\star$ which minimizes the Euclidean norm residual

$$\mathbf{x}^\star = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \|\boldsymbol{b} - \boldsymbol{A}\mathbf{x}\|_2. \tag{1}$$

For large linear systems, iterative solvers are usually preferred for solving (1). However, such solvers can become impractical and exhibit slow convergence if the condition number of the input matrix $\boldsymbol{A}$ is large (ill-conditioned systems). One potential remedy for this challenge is to transform (1) into a mathematically equivalent problem with more favorable properties. Such a transformation is called *preconditioning*, and in particular, the right preconditioned LS system is given by

$$\boldsymbol{y}^\star = \arg\min_{\boldsymbol{y}\in\mathbb{R}^n} \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{y}\|_2,\ \boldsymbol{y}^* = \boldsymbol{M}\mathbf{x}^\star. \tag{2}$$

The matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ is called the *preconditioner*. We can design $\boldsymbol{M}$ having various requirements in mind (e.g., spectral properties, approximating the pseudoinverse, etc.). In practice, we are mostly interested in decreasing the condition number of $\boldsymbol{A}\boldsymbol{M}^{-1}$ (at least compared to the condition number of $\boldsymbol{A}$) and being able to solve linear systems with $\boldsymbol{M}$ inexpensively. In this paper, we solve (2) using the LSQR (Algorithm 1), which is theoretically equivalent to applying conjugate gradients on $\boldsymbol{A}^T\boldsymbol{A}$, but with better numerical properties [38].

Algorithm 1 Preconditioned LSQR

```
Input: matrix $\boldsymbol{A}$, initial solution $\mathbf{x}_0$, right-hand side $\boldsymbol{b}$, tolerance tol, maximum
    number of iterations maxiter, preconditioner $\boldsymbol{M}$
Output: solution $\mathbf{x}$, relative residual relres
1: procedure [$\mathbf{x}$, RELRES] = LSQR($\boldsymbol{A}$, $\mathbf{x}_0$, $\boldsymbol{b}$, tol, maxiter, $\boldsymbol{M}$)
2:     $\beta = \|\boldsymbol{b}\|_2, \mathbf{u} = \boldsymbol{b}/\beta$
3:     $\mathbf{v} = (\boldsymbol{M}^\top)\backslash(\boldsymbol{A}^\top \mathbf{u})$
4:     $\alpha = \|\mathbf{v}\|_2, \mathbf{v} = \mathbf{v}/\alpha$
5:     $\mathbf{w} = \mathbf{v}$
6:     $\bar{\phi} = \beta, \bar{\rho} = \alpha$, iter $= 0$
7:     while (1) do
8:         $\mathbf{u} = \boldsymbol{A}(\boldsymbol{M}\backslash\mathbf{v}) - \alpha\mathbf{u}$
9:         $\beta = \|\mathbf{u}\|_2, \mathbf{u} = \mathbf{u}/\beta$
10:        $\mathbf{v} = \boldsymbol{M}^\top\backslash(\boldsymbol{A}^\top \mathbf{u}) - \beta v$
11:        $\alpha = \|\mathbf{v}\|_2, \mathbf{v} = \mathbf{v}/\alpha$
12:        $\rho = \sqrt{\bar{\rho}^2 + \beta^2}$
13:        $c = \bar{\rho}/\rho$
14:        $s = \beta/\rho$
15:        $\theta = s \cdot \alpha$
16:        $\bar{\rho} = -c \cdot \alpha$
17:        $\phi = c \cdot \bar{\phi}$
18:        $\bar{\phi} = s \cdot \bar{\phi}$
19:        $\mathbf{x} = \mathbf{x} + \boldsymbol{M}\backslash((\phi/\rho)\mathbf{w})$
20:        $\mathbf{w} = \mathbf{v} - (\theta/\rho)\mathbf{w}$
21:        $\mathbf{r} = \boldsymbol{b} - \boldsymbol{A}\mathbf{x}$, relres $= \|\mathbf{r}\|_2/\|\boldsymbol{b}\|_2$, iter $+= 1$
22:        if ((iter == maxiter)||(relres < tol)) then
23:            break
24:        end if
25:    end while
26: end procedure
```

2.1 Related work

Over the last two decades, Randomized Linear Algebra has left its mark on constructing preconditioners through sketching-based methods. Rokhlin and Tygert [40] developed a preconditioner for overdetermined systems by applying a Subsampled Randomized Fourier Transform (SRFT) on the input matrix and then pivoted-QR on the preconditioned system. Similar to that setting, Avron *et al.* [3] constructed the randomized solver *Blendenpik* which consists of four steps:

1. Mix the rows of $\boldsymbol{A}$ by premultiplying it by an appropriate random matrix (i.e., the Randomized Hadamard Transform matrix, the Randomized Discrete Cosine Transform matrix, etc.). Let $\boldsymbol{G} \in \mathbb{R}^{m\times m}$ be this random matrix.
2. Sample s rows (uniformly at random) from the "mixed" matrix $\boldsymbol{GA}$ to create the sampled matrix $(\boldsymbol{GA})_s \in \mathbb{R}^{s\times n}$.
3. QR factorization on $(\boldsymbol{GA})_s$ to construct the preconditioner $\boldsymbol{M}$.

4. Call Algorithm 1 to solve (2).

Intuitively, the “mixing” procedure of step (1) distributes the importance of the rows, thus improving the accuracy guarantees of uniform sampling in the following steps. In other words, the mixing procedure uniformizes the so-called *leverage scores* of the rows of the input matrix $\boldsymbol{A}$; leverage scores play a crucial role in regression problems and random sampling and sketching [20,31]; It is known that the aforementioned transformation *reduces* the maximum leverage score (*coherence*). The *Blendenpik* algorithm is actually a general template for designing randomized preconditioners. For example, [36] proposes the use of a Gaussian matrix instead of the Randomized Hadamard Transform, followed by an alternative approach to the QR decomposition using the Singular Value Decomposition. More recently, Tropp *et al.* [24] described a preconditioner for Conjugate Gradient (CG) via a randomized low-rank Nyström approximation.

The concept of employing mixed-precision arithmetic to improve performance has been recently applied to a range of problems [1]. Furthermore, it has been a well-established approach for linear systems. The recent work of Carson and Dauickait, [13] provides an analysis of a Nyström mixed-precision preconditioner for CG. In [12], the authors use a combination of 32-bit and 64-bit floating point arithmetic for iterative refinement of dense linear systems. Also recently, Lindquist *et al.* [34] presented mixed-precision restarted GMRES for sparse linear systems. However, their work differs from ours in various ways: they provide a mix of single and double-precision implementation but do not focus on half precision. Moreover, they construct each preconditioner in double precision and then store it in single precision for the reduced-precision algorithm, unlike our work (see Section 3).

3 Design and implementation of the mixed precision preconditioner

Our mixed precision implementation uses a Gaussian random matrix $\boldsymbol{G} \in \mathbb{R}^{s\times m}$ in order to sketch the input matrix by computing $\boldsymbol{A}_s = \boldsymbol{G} \cdot \boldsymbol{A}$. For the preconditioner we use the triangular factor of the economy `qr` factorization of the matrix $\boldsymbol{A}_s$, following the approach proposed in [3]. In Matlab notation, this is computed as $[\sim, \boldsymbol{M}] = \mathrm{qr}(\boldsymbol{A}_s, 0)$. In Algorithm 2, we present the mixed precision version of this preconditioner. The `demote` and `promote` functions convert the matrix entries between the required precisions. All the steps of the algorithm are executed on the GPU, using MAGMA [43] routines for the linear algebraic operations and custom CUDA kernels to perform the conversions to different precisions. The Gaussian matrices are generated using the cuRAND functions [4]. You can access our implementation at https://github.com/vasilisge0/randLS/.

In Algorithm 2, we store matrices in high or low precision, as indicated by the types `high_prec` and `low_prec`. The only floating point format for high precision

[4] cuRand v12.0.0 https://docs.nvidia.com/cuda/curand/index.html

we consider in this paper is double, or fp64. For the low precisions, we experimented with the following types: single or fp32; half or fp16; and TensorFloat-32 or tf32. The latter is a 19-bit representation for which NVIDIA provides native support on the Ampere architecture. It uses eight bits for representing the exponent (the same as fp32), but only ten bits for the mantissa (the same as fp16). An additional bit is required to store the sign. Table 1 depicts the precisions used by our implementations of the preconditioner and the solver.

	high precision	low precision
preconditioner	fp64	fp64, fp32, tf32, fp16
solver	fp64	fp64

Table 1: Precisions used in implementing our preconditioner and the LSQR solver.

Algorithm 2 Mixed precision gaussian preconditioner

```
Input: $m \times n$ matrix $\boldsymbol{A}$, number of samples $s$, precision types high_prec, low_prec
Output: $s \times n$ preconditioner $\boldsymbol{M}$
1: procedure $[\boldsymbol{M}]$ = GENERATE_PRECOND($\boldsymbol{A}$, $s$, high_prec, low_prec)
2:     generate $s \times m$ Gaussian matrix $\boldsymbol{G}$
3:     $\hat{\boldsymbol{G}}$ = demote($\boldsymbol{G}$, low_prec)
4:     $\hat{\boldsymbol{A}}$ = demote($\boldsymbol{A}$, low_prec)
5:     $\hat{\boldsymbol{A}}_s = \hat{\boldsymbol{G}}\hat{\boldsymbol{A}}$
6:     $\boldsymbol{A}_s$ = promote($\hat{\boldsymbol{A}}_s$, high_prec)
7:     $[\sim, \boldsymbol{M}] = \mathrm{qr}(\boldsymbol{A}_s, 0)$
8: end procedure
```

The central components underlying the construction of the preconditioner and the solver are BLAS operations. The dominant computation for generating the preconditioner is one matrix-matrix multiplication, while the dominant computation for the solver are dense matrix-vector multiplications. For this purpose, we decided to use the MAGMA library [19,43], which ports BLAS operations on various GPU architectures. In this paper, we want to target specifically NVIDIA devices following the Ampere architecture, in order to test the fp16 and tf32 precision formats. Choosing MAGMA instead of vendor-specific libraries like cuBLAS [5] will allow us to extend our implementation to different architectures in future work. It should be noted that MAGMA provides BLAS functionality either by calling custom CUDA kernels or by directly calling cuBLAS. Mechanisms to make such decision on the fly are also provided.

[5] cuBlas v12.0 https://developer.nvidia.com/cublas

The following code snippet is our implentation of Algorithm 2. We use `value_type_internal` as the reduced precision type for performing the compute-intensive operations and `value_type` for the original precision of the input data. When `value_type_internal` and `value_type` are different, the entries of the input matrix and the sketch matrix are converted to the precision indicated by `value_type_internal` and the matrix multiplication `dmtx_rp = sketch_mtx` $\times$ `mtx` is performed. The output is then converted back into the original precision. If `value_type_internal` is the same as `value_type` then no conversion is required. Afterwards, the economy QR factorization is computed in `value_type` precision and the preconditioner is stored in `dr_factor`.

– **preconditioner::gaussian::generate()**

```
// Generates the preconditioner and measures runtime.
template <typename value_type_internal, typename value_type,
          typename index_type>
void generate(index_type num_rows_sketch, index_type num_cols_sketch,
              value_type* dsketch, index_type ld_sketch,
              index_type num_rows_mtx, index_type num_cols_mtx,
              value_type* dmtx, index_type ld_mtx, value_type* dr_factor,
              index_type ld_r_factor,
              state<value_type_internal, value_type, index_type>&
                  precond_state,
              detail::magma_info& info, double* runtime, double* t_mm,
              double* t_qr)
{
    // Performs matrix-matrix multiplication in value_type_internal
    // precision and promotes output to value_type precision.
    if (!std::is_same<value_type_internal, value_type>::value) {
        cuda::demote(num_rows_mtx, num_cols_mtx, dmtx, num_rows_mtx,
            precond_state.dmtx_rp, num_rows_mtx);
        cuda::demote(num_rows_sketch, num_cols_sketch, dsketch,
            num_rows_sketch, precond_state.dsketch_rp, num_rows_sketch);
        blas::gemm(MagmaNoTrans, MagmaNoTrans, num_rows_sketch,
            num_cols_mtx, num_rows_mtx, 1.0, precond_state.dsketch_rp,
            num_rows_sketch, precond_state.dmtx_rp, num_rows_mtx, 0.0,
            precond_state.dresult_rp, num_rows_sketch, info);
        cuda::promote(num_rows_sketch, num_cols_mtx, precond_state.
            dresult_rp, num_rows_sketch, dr_factor, num_rows_sketch);
    } else {
        // value_type_internal == value_type -> no conversions required
        blas::gemm(MagmaNoTrans, MagmaNoTrans, num_rows_sketch,
            num_cols_mtx, num_rows_mtx, 1.0, dsketch, num_rows_sketch,
            dmtx, num_rows_mtx, 0.0, dr_factor, ld_r_factor, info);
    }

    // Performs qr factorization in value_type precision.
    magma_int_t info_qr = 0;
    blas::geqrf2_gpu(num_rows_sketch, num_cols_mtx, dr_factor,
        ld_r_factor, tau, &info_qr);
    if (info_qr != 0) {
        magma_xerbla("geqrf2_gpu", info_qr);
    }
}
```

Listing 1.1: Generate preconditioner.

The object `state<value_type_internal, value_type, index_type>` is a struct containing the input matrix, the sketch matrix and their product computed in `value_type_internal` precision. It also contains the array `tau`, which is allocated on the cpu and used by the QR factorization.

– **state<value_type_internal, value_type, index_type>**

```
template <typename value_type_internal, typename value_type,
          typename index_type>
struct state{
    value_type_internal* dmtx_rp = nullptr;
    value_type_internal* dsketch_rp = nullptr;
    value_type_internal* dresult_rp = nullptr;
    value_type* tau = nullptr;

    void allocate(index_type ld_mtx, index_type num_cols_mtx,
        index_type num_rows_sketch, index_type num_cols_sketch,
             index_type ld_sketch,
        index_type ld_r_factor) {
        memory::malloc(&dmtx_rp, ld_mtx * num_cols_mtx);
        memory::malloc(&dsketch_rp, ld_sketch * num_cols_sketch);
        memory::malloc(&dresult_rp, ld_r_factor * num_cols_mtx);
        memory::malloc_cpu(&tau, num_rows_sketch);
    }

    void free() {
        memory::free(dmtx_rp);
        memory::free(dsketch_rp);
        memory::free(dresult_rp);
        memory::free_cpu(tau);
    }
};
```

Listing 1.2: State used for storing reduced precision information.

The following code snippet is a our high level implementation of Algorithm 1.

– **solver::lsqr::run()**

```
template <typename value_type_internal, typename value_type,
          typename index_type>
void run(index_type num_rows, index_type num_cols, value_type* mtx,
          value_type* rhs, value_type* init_sol, value_type* sol,
          index_type max_iter, index_type* iter, value_type tol,
          double* resnorm, value_type* precond_mtx,
          index_type ld_precond, magma_queue_t queue)
{
    temp_scalars<value_type, index_type> scalars;
    temp_vectors<value_type_internal, value_type, index_type> vectors;
    initialize(num_rows, num_cols, mtx, rhs,
               precond_mtx, ld_precond, iter, scalars,
               vectors, queue, t_solve);
    while (1) {
        step_1(num_rows, num_cols, mtx, precond_mtx, ld_precond, scalars,
               vectors, queue);
        step_2(num_rows, num_cols, mtx, rhs, sol, precond_mtx,
               ld_precond, scalars, vectors, queue);
        if (check_stopping_criteria(num_rows, num_cols, mtx, rhs, sol,
                                    vectors.temp, iter, max_iter, tol,
                                    resnorm, queue)) {
            break;
        }
    }
    finalize(vectors);
}
```

Listing 1.3: High level implementation of the LSQR solver.

Similar to `preconditioner::gaussian::generate()` the type `value_type_internal` is associated with the precision used in computing the most compute-intensive operations, which, in this case, are the `MV` operations. In this paper, we consider `value_type_internal` and `value_type` to be the same for the solver. The variables `scalars` and `vectors` contain all linear algebraic objects associated with the LSQR algorithm. From an implementation standpoint, Algorithm 1 can be dissected into three consecutive parts: lines 8-11 are implemented in `step_1` and compute the new basis vectors; lines 12-20 are implemented by `step_2` and update the current solution; finally, lines 21-24 are implemented by the `check_stopping_criteria` function, which tests whether convergence has been reached.

4 Numerical Experiments

4.1 Experiment setup

We evaluate the effectiveness and performance of our preconditioned LSQR implementation as follows: We use a selection of $m \times n$ (with $m \gg n$) matrices $\boldsymbol{A}$ and we set the "true" least squares solution to $\mathbf{x} = \text{randn}(n, 1)$, in Matlab notation, with $\boldsymbol{b} = \boldsymbol{A}\mathbf{x}$. This allows us to modify the tolerance in the LSQR algorithm, in order to stress-test the effectiveness of the preconditioner. For our numerical experiments, we use the following datasets (Table 2): (a) a human genetics dataset from the Human Genome Diversity Panel and (b) the CIFAR image dataset.

HGDP: HGDP_1 dataset has emerged from a population genetics application; see [39] and references therein for details. The coefficient matrix related to the regression problem is a tall-and-thin matrix whose entries are -1, 0, 1, 2. Exact details of the underlying genetic application are not relevant for our work here, since the matrix is only used for numerical evaluations. As regards HGDP_2, we modify HGDP_1 to get an ill-conditioned matrix ($\kappa(\boldsymbol{A}) \approx 10^6$) with different dimensions as follows: Initially, we get the first 6000 rows of HGDP_1 and subsequently, we add a few columns by randomly picking existing ones and change a tiny fraction of their elements ($< 1\%$). We carefully act on every change to preserve each entry to be {-1,0,1,2}. The dimensions of the respective datasets are in Table 2.

CIFAR: The CIFAR dataset consists of $60,000$ 32×32 color images belonging in ten (non-overlapping) classes. In our setting, each row represents an image (we vectorize each $32 \times 32 \times 3 = 3,072$ matrix). Our CIFAR_2 dataset consists of $20,000$ randomly chosen images. We normalize all grayscale values to belong in the $[0, 1]$ interval. For the CIFAR_1 dataset, we created a somewhat "thinner" tall and thin matrix by randomly choosing for each image $1,000$ pixels out of the $3,072$.

datasets	rows	columns	cond	aspect ratio
HGDP_1	643,862	425	$O(10^3)$	1.5e3
HGDP_2	60,000	1,000	$O(10^6)$	6.0e1
CIFAR_1	20,000	1,000	$O(10^3)$	2.0e1
CIFAR_2	20,000	3,072	$O(10^4)$	6.5e0

Table 2: Matrices used in experimental evaluation.

Our experiments were conducted on a system, equipped with AMD EPYC 7742 64-Core Processor cluster CPUs and A100 80GB SXM NVIDIA GPUs. Our tests were run exclusively on a single node and utilized one GPU. The NVIDIA A100, which we ran our tests on, has native support for operations in fp16 and tf32 formats, and features tensor cores for matrix operations in fp64, fp16, and tf32. We used as termination criterion for LSQR, the relative residual norm $\frac{\|b-Ax^{(i)}\|}{\|b\|}$, setting the tolerance to 10^{-10} for our numerical experiments with the HGDP and 10^{-12} for the experiments with the CIFAR dataset. For the reported results, GCC 11.3.0, CUDA 14.4.4 and MAGMA 2.6.2 were used.

The goals of our experiments are three-fold: We seek to demonstrate that *(i)* constructing the preconditioner in reduced precision does not severely affect the convergence of the LSQR solver, and *(ii)* modest speedups can be achieved in constructing the preconditioner, which eventually lead to reductions of the total runtime of the preconditioned solver. In our analysis we also attempt to *(iii)* determine the factors that affect the performance of preconditioned LSQR. Those factors are related to properties of the input matrix, but also on implementation choices and underlying hardware.

For each matrix, we report (for varying values of the *sampling coefficient*) *(a)* the breakdown into preconditioner generation cost and solver iteration cost. This plot forms a *runtime profile* for each test matrix; *(b)* the iteration count of the LSQR solvers using different preconditioners; *(c)* the corresponding runtimes; and *(d)* the speedup with regard to the double precision reference preconditioned LSQR solver. The sampling coefficient controls the number of rows of the sketched matrix (and of the resulting preconditioner) as `rows_sampled = sampling_coeff` $\times$ `rows_mtx`. As the value of the sampling coefficient increases, more random samples are generated leading to preconditioners which are more effective, but also more expensive to generate. The data presented in the plots have been averaged over five executions and collected after five warmup runs.

4.2 Discussion

Figures 1, 2, 3, 4 correspond to a problem with a unique combination of *runtime profile* and matrix aspect ratio, i.e., the fraction $\frac{\#\text{rows}}{\#\text{columns}}$. The matrices in descending aspect ratio order are, HGDP_1, HGDP_2, CIFAR_1 and CIFAR_2, (HGDP_1 having the largest and CIFAR_2 the smallest aspect ratio). Their *runtime profiles*, as indicated by the solver to preconditioner-generation runtimes, range from the solver dominating the total runtime (HGDP_1), runtimes being proportional (HGDP_2, CIFAR_1) and preconditioner-generation dominating the total runtime (CIFAR_2). In all of our tests, computing the preconditioner in fp32 is slower than the fp64 implementation. This is related to the lack of specialized hardware units for executing single precision matrix operations (the A100 GPU features tensor cores for fp64, fp16, and tf32 operations but not fp32).

Figure 1 depicts the outcomes of our experimental evaluation for matrix HGDP_1. Firstly, we observe that the convergence of preconditioned LSQR is not affected when the preconditioner is generated in fp32, tf32 or fp16 formats, as depicted in the top-right plot. The corresponding runtimes of the preconditioner generation step are shown in the bottom-left plot. For scaling coefficients

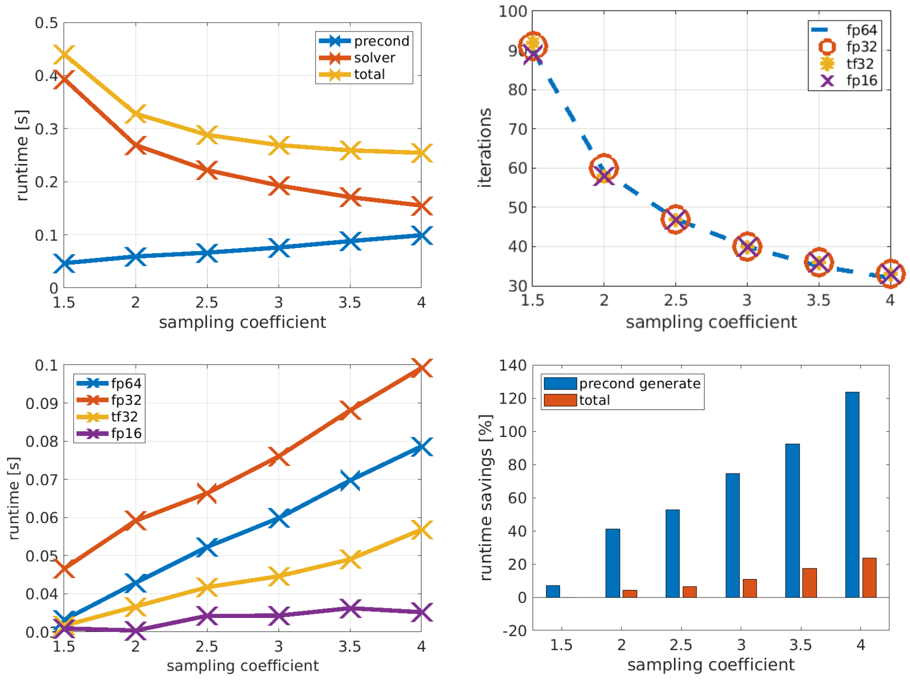

Fig. 1: Evaluation of the mixed precision preconditioner for the HGDP_1 test matrix. Top left: Runtime breakdown of the LSQR algorithm; Top right: Convergence of LSQR using a mixed precision preconditioner; Number of iterations for the 4 precisions overlap. Bottom left: Runtime of the preconditioner generation; Bottom right: Speedup when generating the preconditioner in fp16. Tolerance: 1e-10.

greater than 1.5, we notice a significant reduction in the preconditioner generation runtime when tf32 and fp16 are used. The bottom-right plot depicts the speedup for the preconditioner generation and the overall runtime of the fp16 implementation. Despite the 2.4× speedup for the preconditioner generation, we only see a moderate 1.20× overall algorithm speedup. This is because of the costly solver iteration phase for the HGDP_1 problem (see top-left plot).

Figure 2 presents the evaluation results for the HGDP_2 matrix. This matrix is generated by manipulating HGDP_1 as described in Section 4.1. Computing the preconditioner in fp32 and tf32 formats does not affect the convergence of LSQR but generating the preconditioner in fp16 requires 3× as many LSQR iterations to reach convergence. In the bottom-left plot, we present the runtimes for preconditioner generation and on bottom right the speedup for generating the preconditioner in the tf32 format. The observed speedups of the preconditioner generation step for HGDP_2 are smaller in comparison to HGDP_1. However, the preconditioner overtakes the solver runtime for sampling coefficients greater than 2.5. As a result, the overall speedups are similar to those reported for HGDP_1.

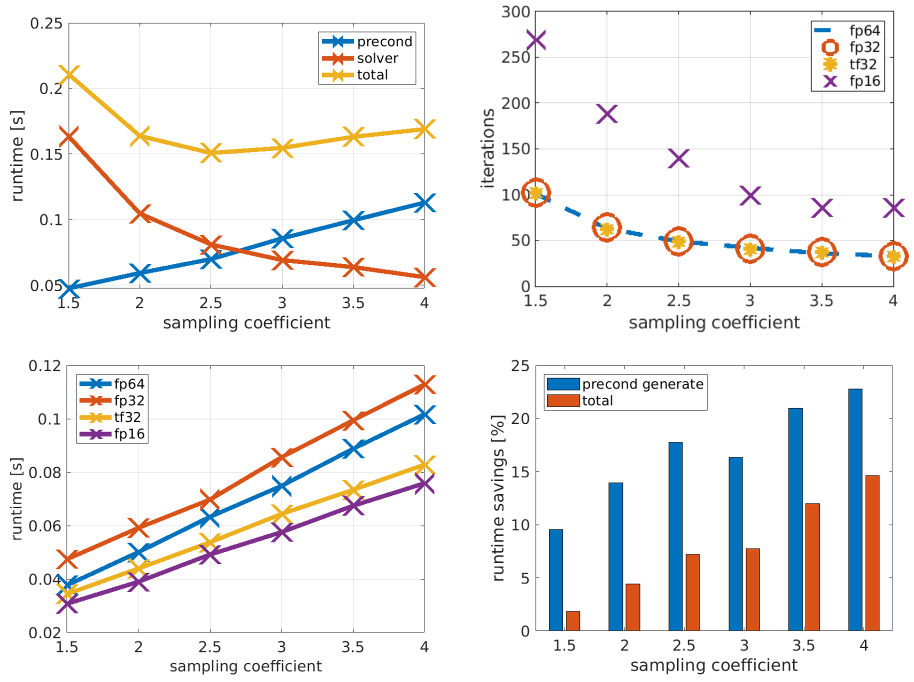

Fig. 2: Evaluation of the mixed precision preconditioner for the HGDP_2 test matrix. Top left: Runtime breakdown of the LSQR algorithm; Top right: Convergence of LSQR using a mixed precision preconditioner; Number of iterations overlap for fp64, fp32 and tf32 precisions; Bottom left: Runtime of the preconditioner generation; Bottom right: Speedup when generating the preconditioner in tf32. Tolerance: 1e-10

In Figures 3 and 4, we present experimental results for CIFAR_1 and CIFAR_2 matrices. In both cases, for moderate and large sampling coefficients, the preconditioner generation step becomes more expensive than the solver iteration phase. Convergence is not affected for CIFAR_1 when changing the precision format. Conversely, for CIFAR_2, the convergence suffers when generating the preconditioner in fp16. For CIFAR_2, the preconditioner generation cost

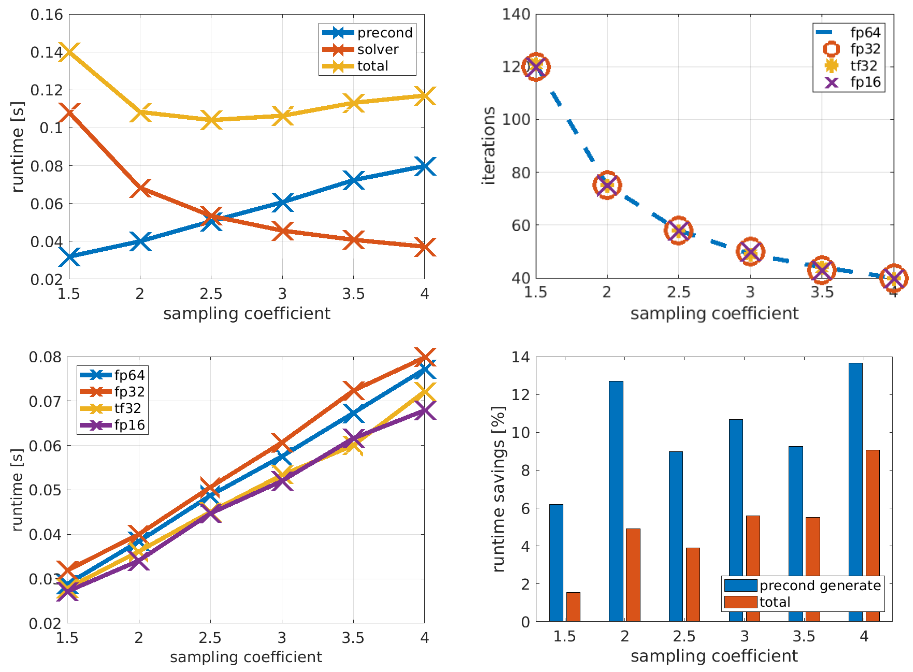

Fig. 3: Evaluation of the mixed precision preconditioner for the CIFAR_1 test matrix. Top left: Runtime breakdown of the LSQR algorithm; Top right: Convergence of LSQR using a mixed precision preconditioner; Iteration plots overlap across different formats; Bottom left: Runtime of the preconditioner generation; Bottom right: Speedup when generating the preconditioner in fp16. Tolerance: 1e-12

is almost independent of the precision format used. This behaviour can be explained by taking into consideration the following; Firstly the aspect ratio of the matrix is too small (approximately 6.5 for CIFAR_2 compared to over 1,500 for HGDP_1), making the theoretical complexity of `qr`, $O(sn^2)$, similar to that of the `matrix multiplication`, $O(smn)$, since m becomes proportional to n. This effect is further amplified by the implementation of the preconditioner on GPU. Even though those components (i.e. `matrix multiply` and `qr`) have similar complexity, implementations of `matrix-matrix multiplication` achieve

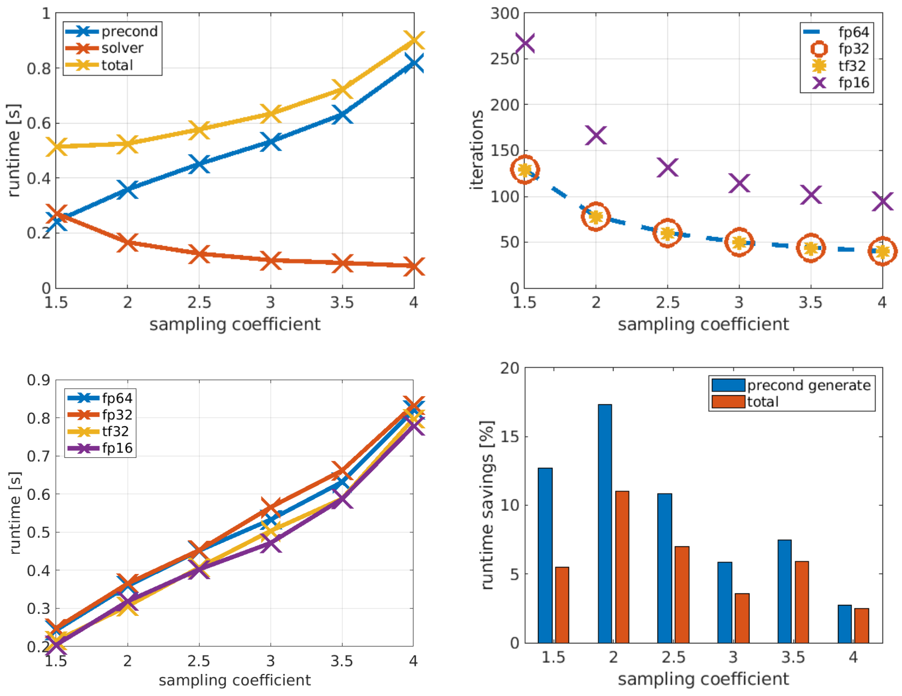

Fig. 4: Evaluation of the mixed precision preconditioner for the CIFAR_2 test matrix. Top left: Runtime breakdown of the LSQR algorithm; Top right: Convergence of LSQR using a mixed precision preconditioner; Iteration plots overlap for fp64, fp32 and tf32 precisions; Bottom left: Runtime of the preconditioner generation; Bottom right: Speedup when generating the preconditioner in tf32. Tolerance: 1e-12

better performance on GPUs. On the other hand, `qr` is harder to parallelize, because it requires operating on the columns of a matrix in a sequential fashion. The above suggest that QR factorization becomes the dominant component of the preconditioner generation when the aspect ratio of the matrix is small, and since it is always computed in double precision, the speedup observed is modest at best. This is also evident from Figure 5, where the runtimes of the the major preconditioner components, namely the matrix-matrix multiplication and the `qr` factorization are presented. We observe that only for the case of HGDP_1 the matrix multiplication is the dominant operation of the preconditioner generation stage.

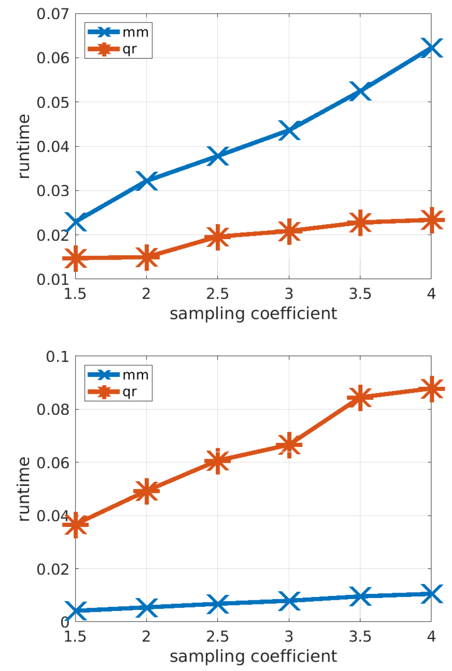

Fig. 5: Runtimes of preconditioner components. Top left: HGDP_1; Top right: HGDP_2; Bottom left: CIFAR_1; Bottom right: CIFAR_2.

5 Conclusion

In this paper, we describe a mixed precision implementation of a randomized preconditioner for solving the dense overdetermined least squares problem and present results on the NVIDIA A100 GPU. In our numerical experiments with matrices from the HGDP and CIFAR datasets, we show that convergence is not affected when using the tf32 format for generating the preconditioner, but we may experience delayed convergence when using fp16 in the preconditioner generation step. Part of our analysis explores how performance is affected by the properties of the input matrix. Attractive runtime savings can be achieved for matrices with high aspect ratio, since mixed precision is applied on the dominant operation of the preconditioner generation stage. Speedups can also be achieved for matrices with balanced row/column ratio, because the preconditioner generation stage requires a significant portion of the total runtime.

In future work, we are interested in combining our preconditioner with a mixed precision implementation of the LSQR solver. This will allow us to further investigate the effect of mixed precision computations on the solution of regression problems. Lastly, we would like to explore the use of mixed precision randomized preconditioning for potentially accelerating sparse least squares solvers and uncovering the factors that impact performance on GPUs.

Acknowledgements. PD and CB were partially supported by NSF grants CCF-2209509, CCF- 1814041, DMS-1760353, and DOE grant DE-SC0022085. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors would like to thank the Innovative Computing Lab at University of Tennessee, for providing access to their compute cluster, to run the numerical experiments. They are also grateful to the reviewers for their insightful comments that helped improve this paper. CB and VG would like to thank Eugenia Kontopoulou for motivating them to pursue the topic of this paper and Efstratios Gallopoulos for introducing them to the *Blendenpik* algorithm.

References

1. Abdelfattah, A., Anzt, H., Boman, E.G., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., Higham, N.J., Li, X.S., et al.: A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. The International Journal of High Performance Computing Applications **35**(4), 344–369 (2021)
2. Aliaga, J.I., Anzt, H., Grtzmacher, T., Quintana-Ort, E.S., Toms, A.E.: Compressed basis gmres on high-performance graphics processing units. The International Journal of High Performance Computing Applications **0**(0), 10943420221115140 (0). https://doi.org/10.1177/10943420221115140, https://doi.org/10.1177/10943420221115140
3. Avron, H., Maymounkov, P., Toledo, S.: Blendenpik: Supercharging lapack's least-squares solver. SIAM Journal on Scientific Computing **32**(3), 1217–1236 (2010). https://doi.org/10.1137/090767911, https://doi.org/10.1137/090767911
4. Baboulin, M., Becker, D., Bosilca, G., Danalis, A., Dongarra, J.: An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems. Parallel Computing **40**(7), 213–223 (2014). https://doi.org/https://doi.org/10.1016/j.parco.2013.12.003, https://www.sciencedirect.com/science/article/pii/S0167819113001488, 7th Workshop on Parallel Matrix Algorithms and Applications
5. Balabanov, O., Grigori, L.: Randomized gram–schmidt process with application to gmres. SIAM Journal on Scientific Computing **44**(3), A1450–A1474 (2022). https://doi.org/10.1137/20M138870X, https://doi.org/10.1137/20M138870X
6. Benzi, M., Tuma, M.: A robust preconditioner with low memory requirements for large sparse least squares problems. SIAM Journal on Scientific Computing **25**(2), 499 –512 (2003). https://doi.org/10.1137/S106482750240649X, https://doi.org/10.1137/S106482750240649X
7. Bindel, D., Demmel, J., Kahan, W., Marques, O.: On computing givens rotations reliably and efficiently. ACM Trans. Math. Softw. **28**(2), 206238 (jun 2002). https://doi.org/10.1145/567806.567809, https://doi.org/10.1145/567806.567809
8. Björck, A.: Solving linear least squares problems by gram-schmidt orthogonalization. BIT Numerical Mathematics **7**, 1–21 (1967). https://doi.org/10.1007/BF01934122, https://doi.org/10.1007/BF01934122

9. Björck, r., Elfving, T., Strakos, Z.: Stability of conjugate gradient and lanczos methods for linear least squares problems. SIAM Journal on Matrix Analysis and Applications **19**(3), 720–736 (1998). https://doi.org/10.1137/S089547989631202X, https://doi.org/10.1137/S089547989631202X
10. Björck, .: Numerics of gram-schmidt orthogonalization. Linear Algebra and its Applications **197-198**, 297–316 (1994). https://doi.org/https://doi.org/10.1016/0024-3795(94)90493-6, https://www.sciencedirect.com/science/article/pii/0024379594904936
11. Björk, A.: Ssor preconditioning methods for sparse least squares problems. p. 2125 (1979)
12. Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., Kurzak, J.: Mixed precision iterative refinement techniques for the solution of dense linear systems. The International Journal of High Performance Computing Applications **21**(4), 457–466 (2007)
13. Carson, E., Daužickaitė, I.: Single-pass nyström approximation in mixed precision (2022). https://doi.org/10.48550/ARXIV.2205.13355, https://arxiv.org/abs/2205.13355
14. Carson, E., Higham, N.J.: Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM Journal on Scientific Computing **40**(2), A817–A847 (2018). https://doi.org/10.1137/17M1140819, https://doi.org/10.1137/17M1140819
15. Carson, E., Higham, N.J., Pranesh, S.: Three-precision gmres-based iterative refinement for least squares problems. SIAM Journal on Scientific Computing **42**(6), A4063–A4083 (2020). https://doi.org/10.1137/20M1316822, https://doi.org/10.1137/20M1316822
16. Cui, X., Hayami, K.: Generalized approximate inverse preconditioners for least squares problems. Japan Journal of Industrial and Applied Mathematics **26**(1) (2008). https://doi.org/https://doi.org/10.1007/BF03167543
17. Cui, X., Hayami, K., Yin, J.F.: Grevilles method for preconditioning least squares problems. Advances in Computational Mathematics **35** (2011). https://doi.org/10.1007/s10444-011-9171-x, https://doi.org/10.1007/s10444-011-9171-x
18. Davis, T.A.: Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. ACM Trans. Math. Softw. **38**(1) (dec 2011). https://doi.org/10.1145/2049662.2049670, https://doi.org/10.1145/2049662.2049670
19. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., Yamazaki, I.: Accelerating numerical dense linear algebra calculations with gpus. Numerical Computations with GPUs pp. 1–26 (2014)
20. Drineas, P., Mahoney, M.W., Muthukrishnan, S.: Sampling algorithms for l 2 regression and applications. In: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm. pp. 1127–1136 (2006)
21. Dubrulle, A.A.: Householder transformations revisited. SIAM Journal on Matrix Analysis and Applications **22**(1), 33–40 (2000). https://doi.org/10.1137/S0895479898338561, https://doi.org/10.1137/S0895479898338561
22. Flegar, G., Anzt, H., Cojean, T., Quintana-Ortí, E.S.: Adaptive precision block-jacobi for high performance preconditioning in the ginkgo linear algebra software. ACM Trans. Math. Softw. **47**(2) (apr 2021). https://doi.org/10.1145/3441850, https://doi.org/10.1145/3441850

23. Fletcher, R.: Conjugate gradient methods for indefinite systems. In: Watson, G.A. (ed.) Numerical Analysis. pp. 73–89. Springer Berlin Heidelberg, Berlin, Heidelberg (1976)
24. Frangella, Z., Tropp, J.A., Udell, M.: Randomized nyström preconditioning. arXiv preprint arXiv:2110.02820 (2021)
25. George, A., Liu, J.W.: Householder reflections versus givens rotations in sparse orthogonal decomposition. Linear Algebra and its Applications **88-89**, 223–238 (1987). https://doi.org/https://doi.org/10.1016/0024-3795(87)90111-X, https://www.sciencedirect.com/science/article/pii/002437958790111X
26. Göbel, F., Grützmacher, T., Ribizel, T., Anzt, H.: Mixed precision incomplete and factorized sparse approximate inverse preconditioning on gpus. In: Sousa, L., Roma, N., Tomás, P. (eds.) Euro-Par 2021: Parallel Processing. pp. 550–564. Springer International Publishing, Cham (2021)
27. Grtzmacher, T., Anzt, H., Quintana-Ort, E.S.: Using ginkgo's memory accessor for improving the accuracy of memory-bound low precision blas. Software: Practice and Experience **53**(1), 81–98 (2023). https://doi.org/https://doi.org/10.1002/spe.3041, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3041
28. Hayami, K., Yin, J.F., Ito, T.: Gmres methods for least squares problems. SIAM Journal on Matrix Analysis and Applications **31**(5), 2400–2430 (2010). https://doi.org/10.1137/070696313, https://doi.org/10.1137/070696313
29. Higham, N.J., Pranesh, S.: Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. SIAM Journal on Scientific Computing **43**(1), A258–A277 (2021). https://doi.org/10.1137/19M1298263, https://doi.org/10.1137/19M1298263
30. Higham, N.J., Pranesh, S., Zounon, M.: Squeezing a matrix into half precision, with an application to solving linear systems. SIAM Journal on Scientific Computing **41**(4), A2536–A2551 (2019). https://doi.org/10.1137/18M1229511, https://doi.org/10.1137/18M1229511
31. Ipsen, I.C., Wentworth, T.: The effect of coherence on sampling from matrices with orthonormal columns, and preconditioned least squares problems. SIAM Journal on Matrix Analysis and Applications **35**(4), 1490–1520 (2014)
32. Kaufman, L.: The generalized householder transformation and sparse matrices. Linear Algebra and its Applications **90**, 221–234 (1987). https://doi.org/https://doi.org/10.1016/0024-3795(87)90314-4, https://www.sciencedirect.com/science/article/pii/0024379587903144
33. Leon, S.J., Björck, ., Gander, W.: Gram-schmidt orthogonalization: 100 years and more. Numerical Linear Algebra with Applications **20**(3), 492–532 (2013). https://doi.org/https://doi.org/10.1002/nla.1839, https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.1839
34. Lindquist, N., Luszczek, P., Dongarra, J.: Accelerating restarted gmres with mixed precision arithmetic. IEEE Transactions on Parallel and Distributed Systems **33**(4), 1027–1037 (2021)
35. Ludwig, R.: Ausgleichung vermittelnder und bedingter Beobachtungen, pp. 58–79. Vieweg+Teubner Verlag, Wiesbaden (1969). https://doi.org/10.1007/978-3-322-98459-3_4, https://doi.org/10.1007/978-3-322-98459-3_4
36. Meng, X., Saunders, M.A., Mahoney, M.W.: Lsrn: A parallel iterative solver for strongly over- or underdetermined systems. SIAM Journal on Scientific Computing **36**(2), C95–C118 (2014). https://doi.org/10.1137/120866580, https://doi.org/10.1137/120866580

37. Paige, C.C., Rozložník, M., Strakos, Z.: Modified gram-schmidt (mgs), least squares, and backward stability of mgs-gmres. SIAM Journal on Matrix Analysis and Applications **28**(1), 264–284 (2006). https://doi.org/10.1137/050630416, https://doi.org/10.1137/050630416
38. Paige, C.C., Saunders, M.A.: Lsqr: An algorithm for sparse linear equations and sparse least squares. ACM Transactions on Mathematical Software (TOMS) **8**(1), 43–71 (1982)
39. Paschou, P., Lewis, J., Javed, A., Drineas, P.: Ancestry informative markers for fine-scale individual assignment to worldwide populations. Journal of Medical Genetics **47**(12) (2010). https://doi.org/10.1136/jmg.2010.078212
40. Rokhlin, V., Tygert, M.: A fast randomized algorithm for overdetermined linear least-squares regression. Proceedings of the National Academy of Sciences **105**(36), 13212–13217 (2008). https://doi.org/10.1073/pnas.0804869105, https://www.pnas.org/doi/abs/10.1073/pnas.0804869105
41. Rotella, F., Zambettakis, I.: Block householder transformation for parallel qr factorization. Applied Mathematics Letters **12**(4), 29–34 (1999). https://doi.org/https://doi.org/10.1016/S0893-9659(99)00028-2, https://www.sciencedirect.com/science/article/pii/S0893965999000282
42. Terao, T., Ozaki, K., Ogita, T.: Lu-cholesky qr algorithms for thin qr decomposition. Parallel Computing **92**, 102571 (2020). https://doi.org/https://doi.org/10.1016/j.parco.2019.102571, https://www.sciencedirect.com/science/article/pii/S0167819119301620
43. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing **36**(5-6), 232–240 (Jun 2010). https://doi.org/10.1016/j.parco.2009.12.005