Undergraduate Research Report Spring 2024 (2024)

Santa Clara University

ARTICLE

Spring Quarter Undergraduate Research Report

L. Calder*

Santa Clara University School of Engineering
*Corresponding author. Email: lcalder@scu.edu

Abstract

A summary of all my undergraduate research during the Spring 2024 Quarter at Santa Clara University. This quarter's focus was on RandBLAS and analyzing how varying matrix S sizes, as well as threads, would impact performance and runtime. Performance counting was conducted with PAPI, whereas visualization of data was displayed with Dashing.

The following work was performed on Santa Clara University's Wave HPC, a multi-core cluster of high-performance computing servers. All work is uploaded to a GitHub repository, with respective instructions regarding how to replicate results or conduct similar analyses. You may view that repository here: *https://github.com/logancalder/COEN193*.

1. Download and Installation of RandBLAS

RandBLAS had multiple dependencies that were challenging to get working on the Wave HPC. This included multiple new libraries that had to be downloaded and linked, including `lapackpp`, `random123`, and `OpenBLAS`.

I ran into several issues here regarding the installation of `RandBLAS`. Namely, it failed to locate linked libraries, even if the correct directory was passed. Further, it complained of a `CUDA` dependency, but this should not be needed as we are not dealing with GPU calculations. The issue was eventually resolved with the successful re-installation and linking of all dependencies and libraries.

2. Performance Counter Analysis With PAPI

Before analyzing the effect of varying parameters for `RandBLAS` sketching, I had to configure a way to effectively (and correctly) analyze data on a library that I was already familiar with. Given this task, I chose to measure and analyze a parallelized nested loop matrix multiplication. This method of multiplying matrices using loops follows the General Matrix Multiply (GEMM) routine [1]. This is defined as:

$$C = \alpha(AB) + \beta C \tag{1}$$

To create this algorithm, we may create a parallelized function that multiplies two matrices together, each one stored in a 1D array. This method is much more efficient than storing matrices in a 2D

array, which takes longer to access. Rather, we may "jump" through each row/column as we know how wide/long each is.
Below is the respective loop-based algorithm for multiplying two matrices, parallelized using `OpenMP`:

```
double *matmul(double *A, double *B, double *C, int alpha, int beta, int m,
    int n, int k)
{
#pragma omp parallel for collapse(2)
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            C[m * j + i] *= beta;

            for (int x = 0; x < k; ++x)
            {
                C[m * j + i] += alpha * A[m * i + x] * B[k * x + j];
            }
        }
    }

    return C;
}
```

Here we may see how the matrix multiplication is parallelized, thus leading the number of threads allocated to its computation to significantly impact its runtime and performance counters. Of course, there is no analysis implemented yet, which shall be conducted with Pathway Active Profiling (`PAPI`).

2.1 GEMM Implementation of PAPI

To analyze how the number of threads impacts performance, I first implemented `PAPI`'s functions in a more user-friendly format. This would allow me to easily implement the analysis on other scripts, with `RandBLAS` being the end goal. The implementation of `PAPI` is quite simple, with the basic steps being:

1. Initialization
2. Start Analysis
3. Stop Analysis
4. Cleanup

All of these steps needed to be parallelized, so using `OpenMP` I was able to adjust each of these steps accordingly. My approach to performance analysis was to run multiple calculations for each parameter, permitting me to average the data to obtain better results. I would run this amount of calculations per performance counter, as I could only measure 1-2 at a time. While this was slow, it yielded accurate results and ensured that `PAPI` did not become overloaded and crash. Data for each performance counter would then be stored into an array, and later deposited into a .csv file, being stored for later analysis. My methodology was as follows:

1. `EventSet` array creation, with the length being set to the current number of threads.
2. For each event analysis, add it to the `EventSet`.
3. Starting `PAPI` before the desired calculation to be measured is run.
4. Stopping `PAPI` after the calculation has finished.
5. Storing the analysis data into a "pre-average" array, which is then averaged once all calculations have finished.
6. Storing this new averaged data into the current matrices and thread's output .csv file.
7. Repeat for all other parameters.

The functions and implementation for this approach may be viewed under */COEN193/dgemm/*, listed in the filed *da.h* and *dgemm.cpp*.

2.2 Analysis on RandBLAS' Dense Sketching

For the Dense Sketches in `RandBLAS`' example files, we opted for the following parameters:

1. Thread allocation of 1-4, 5, 10, 20
2. Matrix sizes of 10000x10000, 10000x2000, 10000x500, 50000x10000, 50000x2000, and 50000x200.

Provided the amount of parameters that were being considered in these experiments (thread count and matrix side), I opted for a fully automated approach to running the calculations using a shell script. This would loop through each parameter and thread, writing the data as described above. However, this approach to writing the results is not directly readable by `Dashing`, the visualization software used to analyze which parameters have the strongest effect on performance. To navigate this issue, I developed a Python script that would reformat the data into readable rows for the software.

3. What is Dense Sketching?

Dense sketching is a technique used in linear algebra and randomized numerical linear algebra to approximate large-scale linear algebra problems efficiently. The main idea is to compress a large matrix into a smaller "sketch" while preserving certain properties of the original matrix, allowing for faster computation with reduced memory usage.

3.1 How Does RandBLAS Function?

`RandBLAS` is a C++ library for dimension reduction via random linear transformations. These random linear transformations are called *sketching operators*. The act of applying a sketching operator – that is, the act of *sketching* – is of fundamental importance to randomized numerical linear algebra [2].

4. Performance Counter Results of RandBLAS' Dense Sketching

The areas that I analyzed regarding `RandBLAS`' Dense Sketching were the following functions:

1. Fill Dense, a parallelized subroutine of RandBLAS:

```
RandBLAS::fill_dense(S);
```

2. Sketch General, which conducts the sketching calculation:

```
RandBLAS::sketch_general<double>(
                blas::Layout::ColMajor, // Matrix storage layout of AB and
    SAB
                blas::Op::NoTrans,      // NoTrans => \op(S) = S, Trans =>
    \op(S) = S^T
                blas::Op::NoTrans,      // NoTrans => \op(AB) = AB, Trans
    => \op(AB) = AB^T
                sk_dim,                 // Number of rows of S and SAB
                n + 1,                  // Number of columns of AB and SAB
                m,                      // Number of rows of AB and columns
     of S
                1,                      // Scalar alpha - if alpha is zero
    AB is not accessed
                S,                      // A DenseSkOp or SparseSkOp
    sketching operator
                AB,                     // Matrix to be sketched
                m,                      // Leading dimension of AB
                0,                      // Scalar beta - if beta is zero
    SAB is not accessed
```

```
                SAB,                        // Sketched matrix SAB
                sk_dim                      // Leading dimension of SAB
            );
```

3. GESDD: Computes the Singular Value Decomposition (SVD) of a matrix:

```
lapack::gesdd(lapack::Job::AllVec, sk_dim, (n + 1), SAB, sk_dim, svals, U,
    sk_dim, VT, n + 1);
```

Analyzing the results of these subroutines, it is apparent that *RandBLAS::fill_dense(S)* would be the only occurrence to be affected by thread count, as it is the only one that is parallelized. Provided this, we may view how matrix size and thread count has affected performance counters over our varying instances of running and measuring the subroutine (see Figures 1-3).

4.1 Observing RandBLAS' Fill Dense Subroutine

The data below indicates trends among certain performance events and thread count along varying matrix sizes.

Figure 1. Sunburst chart showing the importance of each region's contribution to the execution of the subroutine.

4.2 Analyzing RandBLAS' Fill Dense Subroutine

From this data, we can draw reasonable conclusions about varying events.

4.2.1 Runtime

The time which it takes to perform the *RandBLAS::fill_dense(S)* subroutine grows proportionally to matrix size. Examining Figure 1, we may view how the 50000x2000 matrix took the longest to execute, whereas 10000x500 had the shortest runtime. Further, thread count corresponded inversely to runtime, with an increased number of threads resulting in shorter runtime.

Table 1. Matrix Sizes vs. % of Total Program Runtime

Matrix Size	% Of Tot. Program Runtime
10000x500	2.40
10000x1000	4.77
10000x2000	9.52
50000x500	11.94
50000x1000	23.87
50000x2000	47.50

Table 2. Matrix Sizes vs. % of Total Program Runtime

Matrix Size	1 Thread Runtime (s)	5 Threads Runtime (s)	10 Threads Runtime (s)	20 Threads Runtime (s)
10000x500	41.4378	8.53591	4.32923	2.95882
10000x1000	82.8848	17.0584	8.59445	5.86295
10000x2000	165.754	34.0628	17.0819	11.6207
50000x500	207.547	42.759	21.9763	14.7215
50000x1000	415.135	85.3193	42.7878	29.0994
50000x2000	830.103	170.474	85.3078	57.9403

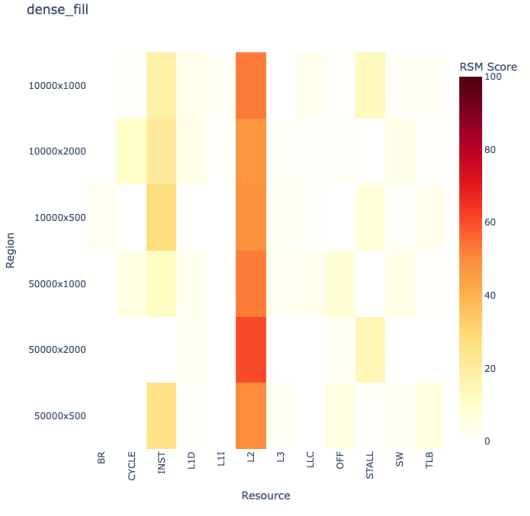

Figure 2. Heatmap showing what resources are used heavily in corresponding regions.

4.2.2 *Optimizations*

Looking at Figure 2, it becomes apparent that `L2` is a key event in optimization (signified by the vertical orange/red bar). The events specifically within `L2` that resulted in the highest Percent Error Reduced (PER) were `L2_RQSTS:CODE_RD_HIT` (30.68%) and `L2_RQSTS:ALL_CODE_RD` (33.41%) for a PER of 60.88%.

5. Performance Counter Results of RandBLAS' Sparse Sketching

For sparse sketching, I focused on analyzing three areas, all of which being parallel versions of the dense functions mentioned in **4)**. These are:

1. Fill Sparse, a parallelized subroutine of RandBLAS:

```
RandBLAS::fill_sparse(S);
```

2. Sketch General, which conducts the sketching calculation:

```
RandBLAS::sketch_general<double>(
                blas::Layout::ColMajor, // Matrix storage layout of AB and
    SAB
                blas::Op::NoTrans,      // NoTrans => \op(S) = S, Trans =>
    \op(S) = S^T
                blas::Op::NoTrans,      // NoTrans => \op(AB) = AB, Trans
    => \op(AB) = AB^T
                sk_dim,                 // Number of rows of S and SAB
                n + 1,                  // Number of columns of AB and SAB
                m,                      // Number of rows of AB and columns
     of S
                1,                      // Scalar alpha - if alpha is zero
    AB is not accessed
                S,                      // A DenseSkOp or SparseSkOp
    sketching operator
                AB,                     // Matrix to be sketched
                m,                      // Leading dimension of AB
                0,                      // Scalar beta - if beta is zero
    SAB is not accessed
                SAB,                    // Sketched matrix SAB
                sk_dim                  // Leading dimension of SAB
            );
```

3. TLS: Performs total least squares on sketched matrix. Note: *lapack::gesdd* and *blas::scal* are of interest and were also analyzed for parallel computing optimization:

```
lapack::gesdd(lapack::Job::AllVec, sk_dim, (n+1), SAB, sk_dim, svals, U,
    sk_dim, VT, n+1);

for (int i = 0; i < n; i++) {
    X[i] = VT[n + i*(n+1)]; // Take the right n by 1 block of V
}

// Scale X by the inverse of the 1 by 1 bottom right block of V
blas::scal(n, -1/VT[(n+1)*(n+1)-1], X, 1);
```

Inversely from Dense sketching, it was observed that *RandBLAS::sketch_general<double>* was parallelized, whereas *RandBLAS::fill_dense(S)* did not display any parallel computing characteristics. Further analysis will be conducted on this in the Fall to solve this discrepancy. It is highly unlikely that the two would not reflect the same results, and is more likely that an analysis error has occurred. An initial revision of Sparse data confirmed results to be correct, therefore placing the likelihood of error within Dense sketching analysis.

5.1 Observing RandBLAS' Fill Sparse Subroutine

The data below is from the specified sections of code listed above, analyzed using `PAPI`. We will first analyse *TLS*, comprised of its respective subroutines *RandBLAS::sketch_general<double>* and *RandBLAS::fill_dense(S)*.

5.1.1 GESDD

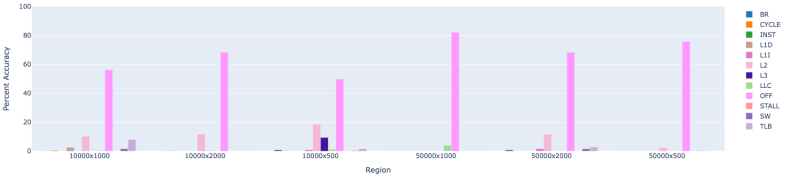

Figure 3. Bar graph showing event impact on *lapack::gesdd*, which is a subroutine of *TLS*.

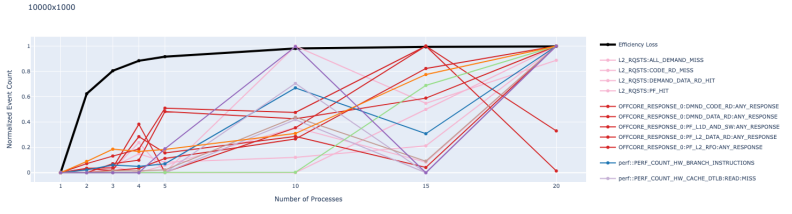

Figure 4. Line graph displaying event impact on *lapack::gesdd* over thread count for 10000x10000 matrices.

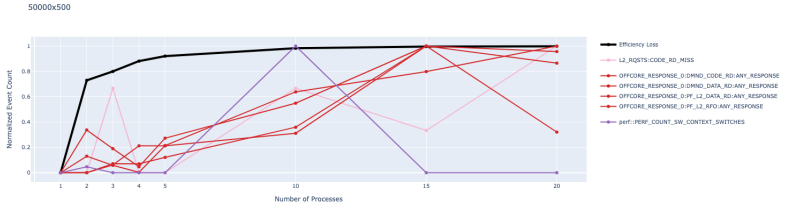

Figure 5. Line graph displaying event impact on *lapack::gesdd* over thread count for 50000x500 matrices.

These results show a recurring pattern where *L1I* events tend to have the highest percent accuracy over varying matrix sizes. Further, *OFFCORE_RESPONSE_0* events seemingly correspond to efficiency loss, increasing simultaneously.

5.1.2 *Scal*

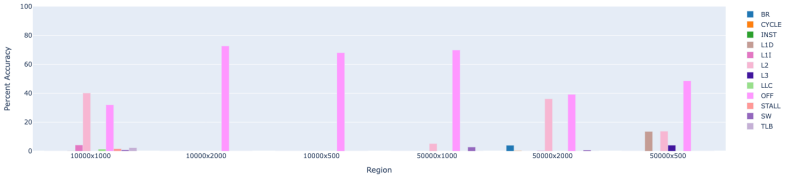

Figure 6. Bar graph showing event impact on *blas::scal*, which is a subroutine of *TLS*.

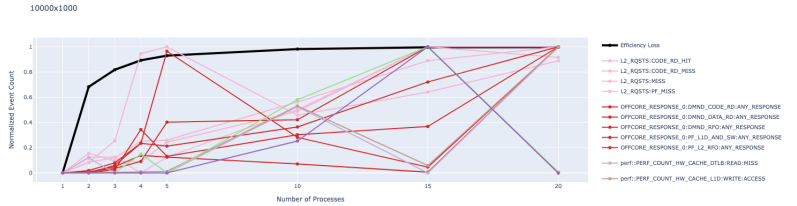

Figure 7. Line graph displaying event impact on *blas::scal* over thread count for 10000x10000 matrices.

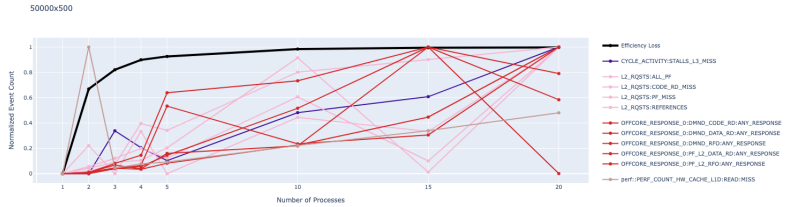

Figure 8. Line graph displaying event impact on *blas::scal* over thread count for 50000x500 matrices.

The subroutine *blas::scal* seemingly displays the same patterns as *lapack::gesdd*, where *L1I* relates to higher percent accuracy and *OFFCORE_RESPONSE_0* results in higher efficiency loss.

5.1.3 *TLS Subroutine Conclusions*

While these subroutines displayed results suggesting that they are not parallelized using *OpenMP*, it is possible that they could be parallelized another way, thus making my analysis of them incorrect. From these results, we may view how thread count does affect events and efficiency loss, suggesting another parallelization method.

5.1.4 TLS

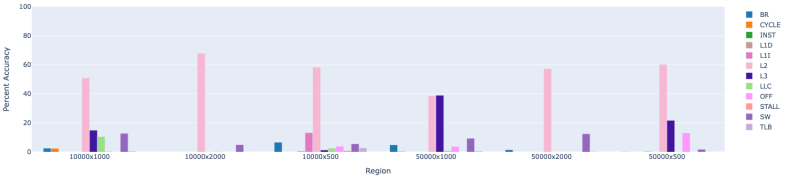

Figure 9. Bar graph showing event impact on *TLS* (includes *blas::scal* and *lapack::gesdd*.

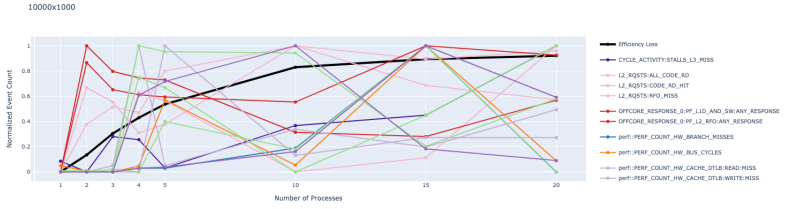

Figure 10. Line graph displaying event impact on *TLS* over thread count for 10000x10000 matrices.

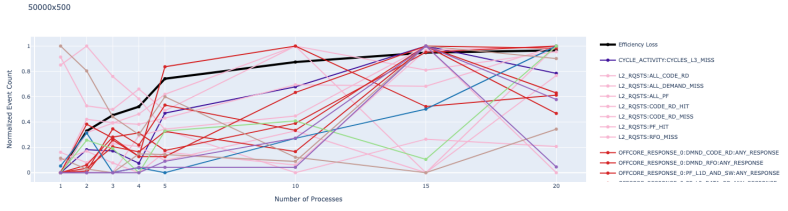

Figure 11. Line graph displaying event impact on *TLS* over thread count for 50000x500 matrices.

5.1.5 TLS Conclusions

The data obtained for the *TLS* subroutine was generally inconclusive, with the only significant data obtained being the consistent spike of *L2* events correlating to percent accuracy. Line charts did not suggest much correlation of events to efficiency loss.

6. Conclusion & Plans For Fall Quarter

Provided that the data obtained for these thread counts resulted in a clear pattern, I aim to work on optimizing `RandBLAS`' Dense and Sparse Sketching next Quarter. I also hope to find the parallelization method for some subroutines discussed prior.

6.1 Remarks

I found this Quarter to be much more challenging than the last, but also more fulfilling. I was very pleased to finally obtain performance results, and to have begun to analyze the data. Last Quarter, my

work mainly revolved around the installation and setup of software. While this Quarter did begin with that same process, I was able to write performance analysis functions to analyze how varying matrix size and thread count affect the runtime and events for loop-based matrix multiplication and dense sketching.

This is also a significant achievement for me as I had no prior experience or knowledge of parallel computing. I did not take a course to learn about `pragma omp`, and thus had to self-learn and make changes alongside my peers.

7. Acknowledgments

I would like to credit Anthony Bryson (Santa Clara University) for all his help this Quarter with my research, as he aided me greatly with learning about parallel computing and the utilization of `Dashing`'s analysis software. I would also like to thank Gaurav P. (Santa Clara University) for his help in aiding my installation and troubleshooting of RandBLAS and its dependencies.

References

[1] NVIDIA Corporation. Matrix Multiplication Background User's Guide. *Accessed 05.30.2024*, 2024.

[2] Riley J. Murray, Burlen Loring. RandBLAS: sketching for randomized numerical linear algebra. networks. *Accessed 05.30.2024*, 2024.