Undergraduate Research Report Fall 2024 (2024)

Santa Clara University

ARTICLE

Fall 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 Fall 2024 Quarter at Santa Clara University. This quarter's focus was on RandBLAS's Total Least Squares (TLS) Dense and Sparse Sketch routines, observing how various thread, memory, and thread mapping configurations would impact performance and runtime. Performance counting was conducted with PAPI, whereas visualization of data was displayed with Dashing and Plotly.JS.

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. Performance Counter Analysis With PAPI

Before analyzing the effect of varying parameters for `RandBLAS`' TLS Dense/Sparse 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`).

1.1 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 `TLS Dense/Sparse` sketches 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 file *da.h*.

1.2 Analysis on RandBLAS' TLS Dense and Sparse Sketches

For the TLS Dense and Sparse Sketching in `RandBLAS`' example files, we opted for the following parameters:

1. Thread allocation of 4, 8, 12 ... 48.
2. Matrix sizes of 50000x1000 and 50000x2000.

3. Thread Mapping of parameters: Master, Close, Spread, True, and False.

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 graph results under their respective code section, with the X axis representing thread count and Y axis representing runtime. Within these graphs would be five lines, each representing a different thread mapping configuration. To plot my results, I used Plotly.JS, a library that prohibits easy data visualization. For an in-depth analysis of performance counters, I used Dashing, which visually displays patterns of resource usage by the programs.

2. 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. What is Sparse Sketching?

Sparse sketching in linear algebra refers to techniques used to approximate or compress large-scale matrices, vectors, or linear transformations while retaining essential structural information. These methods are particularly useful in high-dimensional data processing and optimization, enabling computational efficiency and memory savings.

Sparse sketching often uses **sparse random matrices** (matrices with most entries being zero) to project data into a lower-dimensional space.

3.1 How Does TLS Sketching function Work in RandBLAS?

`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. Analyzed Subsections of RandBLAS' TLS Dense Sketching

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

1. Fill Dense, a parallelized subroutine of RandBLAS TLS Dense:

```
RandBLAS::DenseDistName dn = RandBLAS::DenseDistName::Gaussian;

// Initialize dense distribution struct for the sketching operator
RandBLAS::DenseDist Dist(sk_dim,   // Number of rows of the sketching
    operator
                         m,        // Number of columns of the sketching
    operator
                         dn);     // Distribution of the entires

//Construct the dense sketching operator
RandBLAS::DenseSkOp<float> S(Dist, seed);
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, which 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);
```

5. Analyzed Subsections of RandBLAS' TLS Sparse Sketching

1. Fill Sparse, a parallelized subroutine of RandBLAS TLS Sparse:

```
// Initialize sparse distribution struct for the sketching operator
uint32_t seed = 0;      // Initialize seed for random number generation
RandBLAS::SparseDist Dist = {.n_rows = sk_dim,
    // Number of rows of the sketching operator
                             .n_cols = m,
    // Number of columns of the sketching operator
                             .vec_nnz = 4,                                  //
     Number of non-zero entires per major axis
                             .major_axis = RandBLAS::MajorAxis::Short
    // Defines the major axis of the sketching operator
                            };

//Construct the sparse sketching operator
RandBLAS::SparseSkOp<double> S(Dist, seed);
RandBLAS::fill_sparse(S);
```

2. Sketch General, which conducts the sketching calculation (same as Dense).
3. GESDD, which computes the Singular Value Decomposition (SVD) of a matrix (also same as Dense).

5.1 Analyzing RandBLAS' TLS Sketch Dense

5.1.1 fill_dense and create_dense_sketch

The time which it takes to perform the fill and create dense subroutine decreases proportionally with accordance to thread count, with the exception of the thread mapping selection, `master`. This mapping results in a linear increase in runtime with a larger number of threads. The likely reasoning for an increased runtime for `master` when combined with larger thread counts is due to threads being bound to the same cores as the master thread. This results in an uneven work distribution and resource contention.

While an increased thread count does result in a shorter execution time, the tradeoff for system resources and performance becomes negligible as we increase allocated thread count. Provided this, the

optimal allocation of threads that we suggest would be **24**, with the thread mapping selection of `false`.

Looking at Figure 1, it becomes apparent that `L2` displays high cache contention and misses, likely resulting in the increased runtime displayed in Figure 2. Viewing Figure 8, we observe worsening efficiency loss with higher thread count for `master`, which leads us to draw the conclusion of `spread` and `close` being optimal thread mapping configurations (See Figure 2).

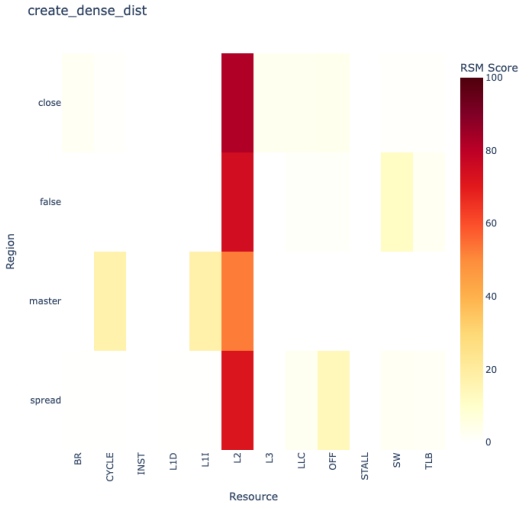

Figure 1. Heatmap showing resource usage per thread map selection of createdense sketch

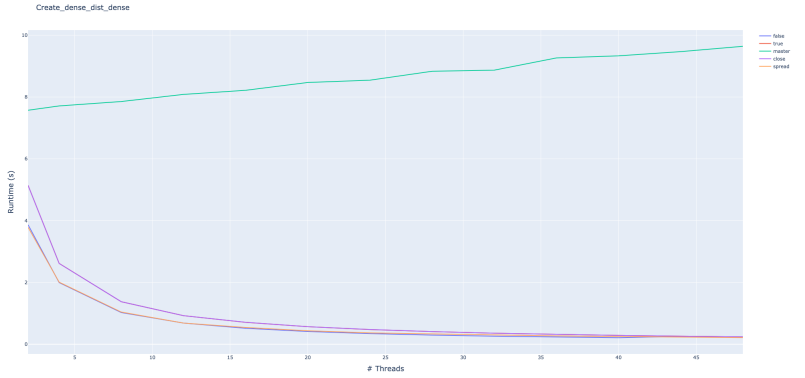

Figure 2. Runtime analysis for create dense dist

5.1.2 sketch general

Within this subsection of code, we observe a relative tie among the varying thread mapping configurations, with the exception of `master` being highly inefficient (Figure 3). This setting displays a jagged pattern of runtime increasing proportionally to thread count, suggesting a conflict over resources allocated to each thread. Further, `L2` once again proves to be highly used solely by `master`, suggesting high contention and misses (Figure 9).

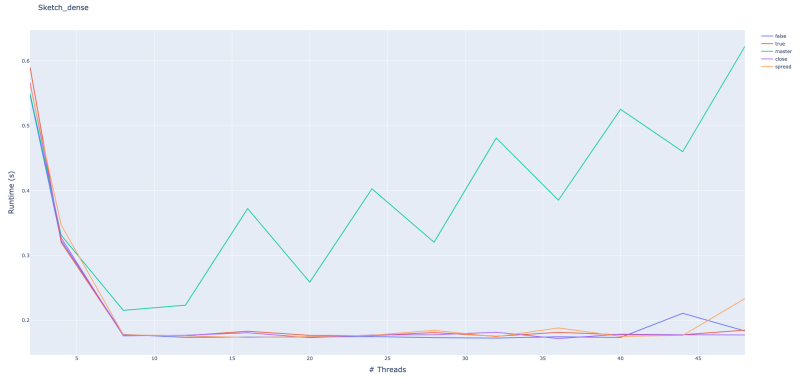

Figure 3. Runtime analysis of thread mapping for sketch general (dense)

Provided Figures 3 and 9, we draw the conclusion that the optimal settings for running `sketch_general` (for Dense) would be **8** threads ran on either `false`, `true`, `close`, or `spread`.

5.1.3 gesdd

This function was by far the most conclusive of the Dense subsections, with a clear advantage of using the thread mapping configuration, `false`. Observing Figures 4 and 11, we view a consistent pattern of `false` being faster than any other selection, running in less time for all thread counts. Further, while `false` is most optimal, all settings do plateau at 8 threads (with the exceptions of a few performance spikes, likely due to resource conflicts), which allows us to suggest the optimal settings of **8** threads ran on `false` for `gesdd`.

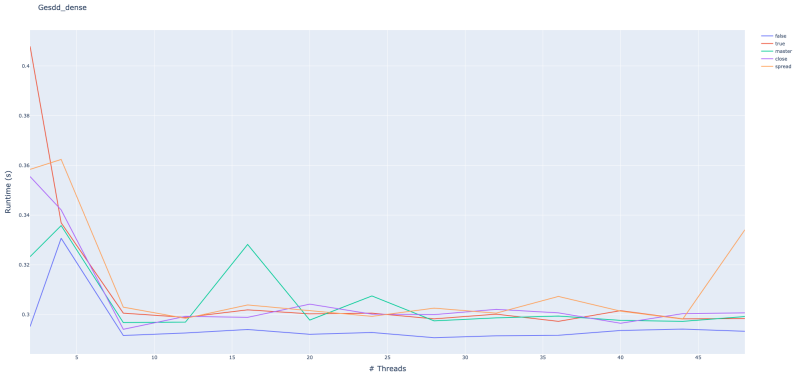

Figure 4. Runtime analysis for gesdd (dense)

5.2 50000x2000 Analysis

It should be noted that while experiments were ran on 50000x2000 matrix sketches (See Figures 12-14), the results were very similar to 50000x1000, simply with larger runtimes. This data follows the same trends, optimal thread count, and best thread mapping settings as the 50000x1000 experiment.

5.2.1 Concluding Optimizations for TLS Dense

For the most optimal performance of TLS Dense Sketch, we suggest the following settings: **8** threads ran on `false` thread affinity policy.

5.3 Analyzing RandBLAS' TLS Sketch Sparse

5.3.1 fill sparse

The results for `fill_sparse` are relatively inconclusive, with intersecting lines at nearly every thread count (Figure 5). However, we may still draw conclusions for the most optimal settings for reduced runtime.

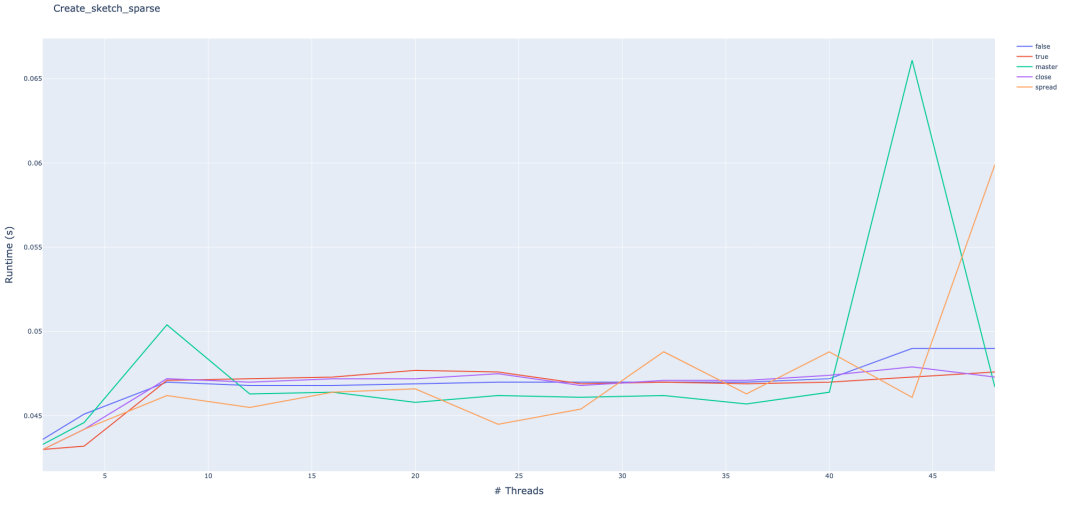

Figure 5. Runtime analysis for fill sparse

If we would like to have the most optimal performance with least amount of threads utilized, we may select **4** threads with thread mapping configuration of `true`. This results in an average runtime of **0.028** seconds, opposed to other settings at the same thread count which average 0.035 seconds.

5.3.2 sketch general

Analyzing sketch general, we observe that `master` is highly inefficient, displaying a linear pattern of increased runtime with increased thread count (Figure 6). This is very similar to `sketch_general` (dense) and `create_dense`. Once again, this is very likely due to high resource contention from the sharing of cores with the master thread.

Further, from this data, we may conclude that `false` and `spread` result in the lowest runtimes, with the exact same results for each thread count. `false` is expected to be low, as it allows threads to dynamically request resources and bind to cores. `spread`, on the other hand, is very efficient here as we are now dealing with a sparse workload (in contrast to dense). A sparse workload avoids thread concentration on the same cores, and so specifying `spread` accomplishes this default behavior (it should be noted that `false` is the default setting).

Concluding from our data, we recommend an optimal setting of **8–12** threads and using either `false` or `spread` for thread mapping.

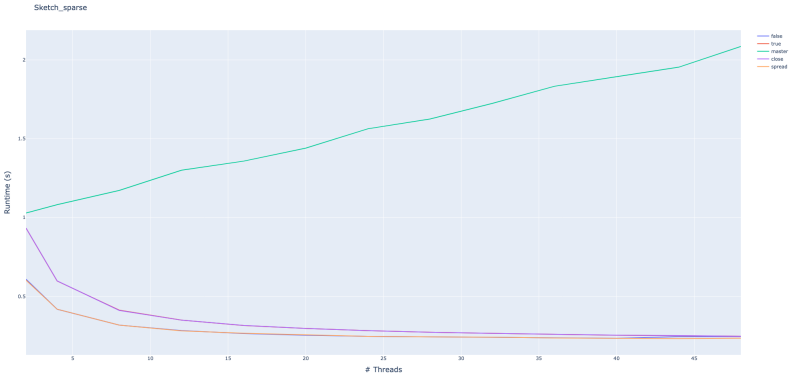

Figure 6. Runtime analysis for sketch general (sparse)

5.3.3 gesdd

Gesdd also displays an inefficient runtime for `master`, whereas all other selections nearly tie in efficiency (Figure 7). Further, performance boosts from thread increases plateau around 8 threads, similarly to all other experiments. Analyzing Figure 10, we may view performance counter analysis. The `OMP_PROC_BIND=false` setting mitigates some bottlenecks compared to `master`, leading to reduced efficiency loss. However, it still suffers from cache contention and stalls at high process counts.

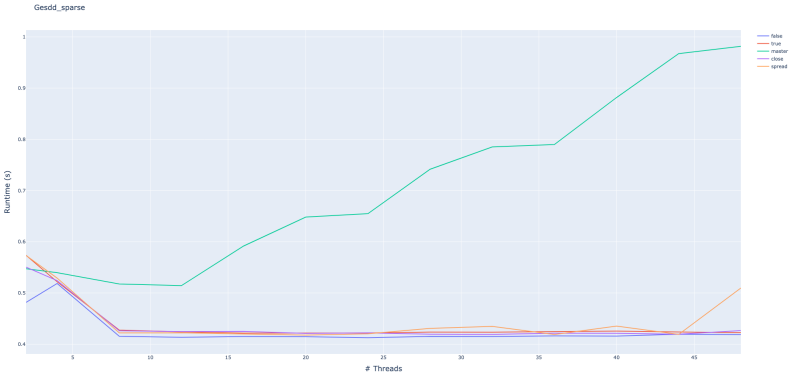

Figure 7. Runtime analysis of gesdd (sparse)

Given that `gesdd` is not a sparse or dense specific function, we do not expect `spread` to be the most optimal setting. However, it will differ as the parameters passed into it will indeed be different, thus affecting the thread mapping performance.

From our results, we strongly recommend **8** threads and `false`, `spread`, `close`, and `true` for most optimal performance.

5.3.4 Concluding Optimizations for TLS Sparse

While `true` and 4 threads may be best for `fill_sparse`, the `spread` thread mapping selection combined with **8–12** threads has been found to be the most optimal selection for efficiency when decreasing runtime. This is namely due to `spread` being highly efficient when working on sparse programs, as the two avoid thread concentration on the same cores.

5.4 50000x2000 Analysis

Results for `gesdd` and `sketch_general` were near-identical to the 50000x1000 experiment, with `master` following the same upwards linear trend, whilst other settings plateaued with around 8 threads allocated. However, interestingly, `master` displayed the best performance for creating the sketch, with the lowest runtime at a thread count of 4. While this pattern is unique to the 50000x2000 experiment, this data is not extremely relevant due to `create_sketch` only compensating for roughly 10-20 percent of runtime.

6. Conclusion

I am quite content with the progress and results I obtained from this quarter's work. While previously in earlier quarters I solely worked on varying matrix size and thread count, diving deeper into utilizing the entire CPU Node (48 threads, 96 cores) and its memory (512GB RAM), as well as varying thread mapping configurations, I was able to discover patterns that would have otherwise gone undetected, and provide solutions for the most optimal performance. I personally found it very interesting how thread mapping configurations mattered for TLS Dense and Sparse uniquely, as Sparse corresponded strongly to the `spread` setting. I prior was unaware of such a similarity, but after discovering so, the possibility for better optimizations has arisen.

6.1 Remarks

As this marks my last quarter researching at SCU, I would like to thank Dr. Cho for his mentorship throughout my last year in this position. I have learned a great deal not just about sketching, but also multi-threading and the analysis of heap amounts of data. This opportunity was extremely insightful to me and I am very grateful to have been granted such an opportunity. I would also really like to thank each member that was a part of my research group, as they are all very intelligent and bright individuals who aided me in my studies. I wish them nothing but the best and would like to acknowledge their research as well, as it was extremely interesting to learn about.

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.

7. Additional Figures

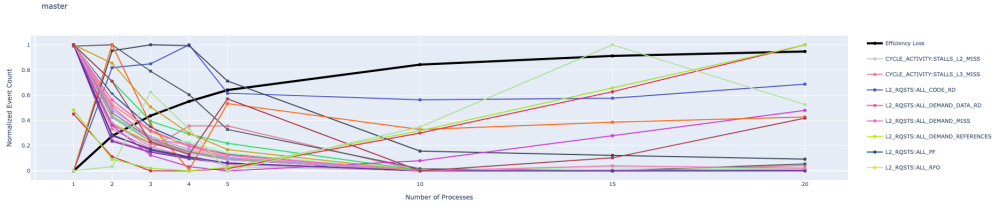

Figure 8. Performance counter visualization for "master" thread mapping on Dense

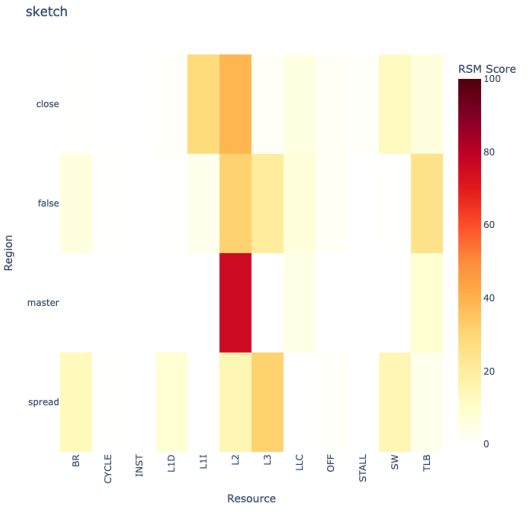

Figure 9. Heat map for sketch general (dense)

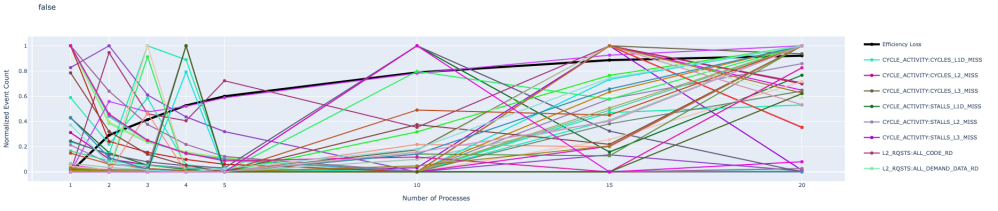

Figure 10. Performance counter analysis for "false" thread mapping on Sparse

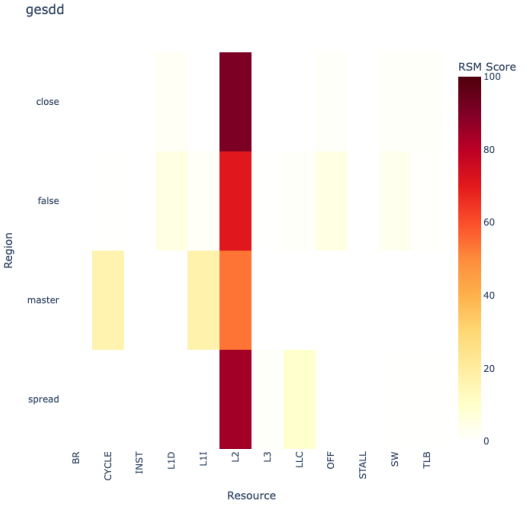

Figure 11. Heatmap displaying resource usage per thread mapping configuration for gesdd (dense)

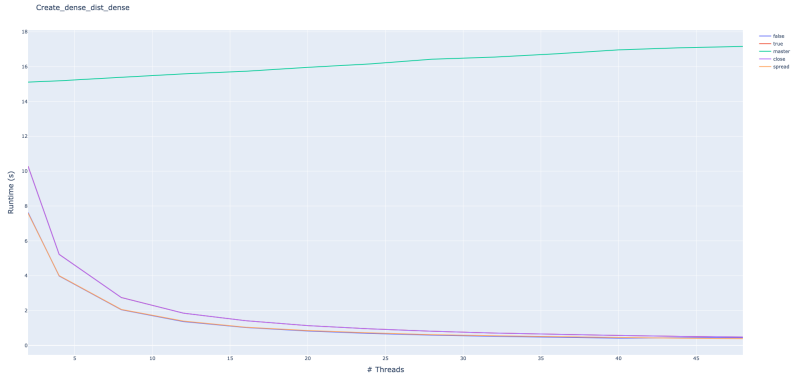

Figure 12. 50000x2000 runtime analysis for creating dense distribution

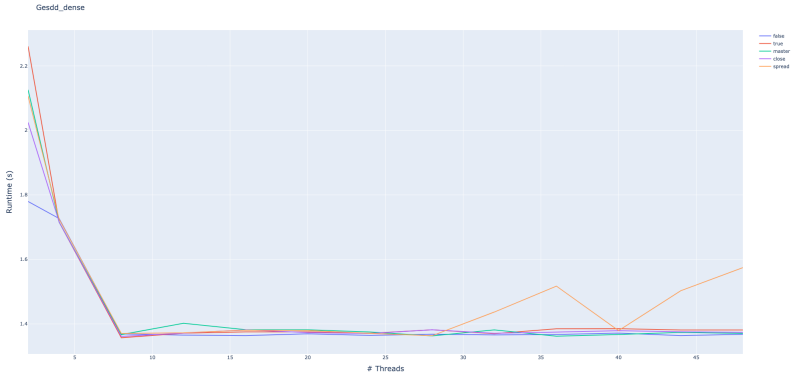

Figure 13. 50000x2000 runtime analysis for dense gesdd

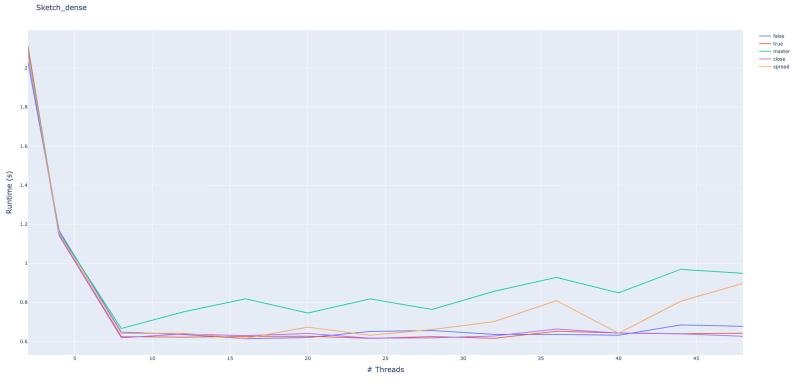

Figure 14. 50000x2000 runtime analysis for sketch dense general

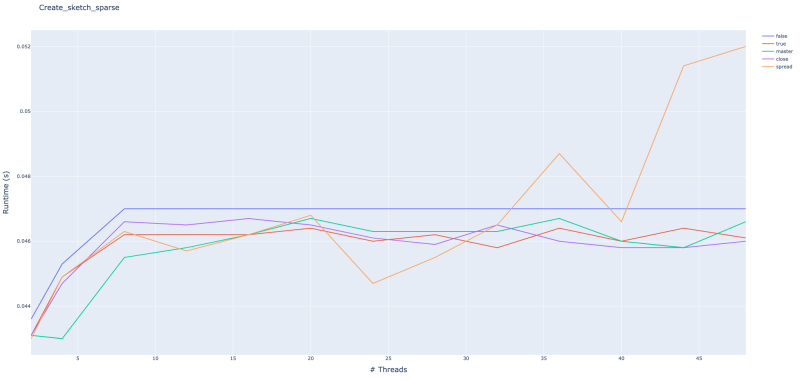

Figure 15. 50000x2000 runtime analysis for creating sparse distribution

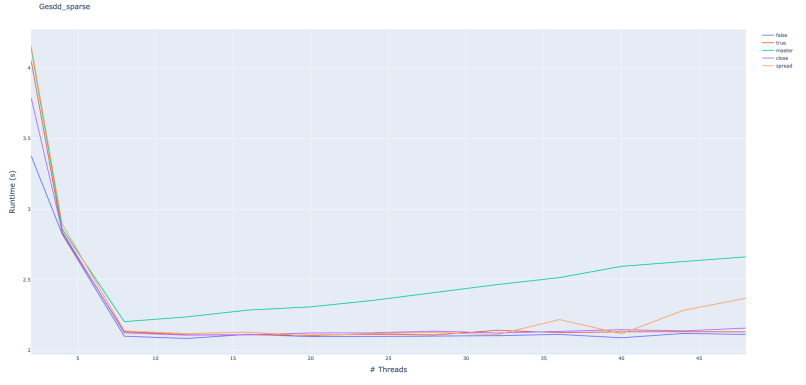

Figure 16. 50000x2000 runtime analysis for sparse gesdd

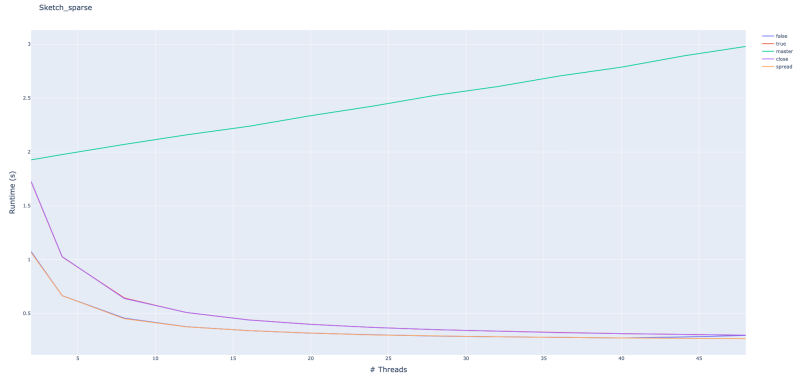

Figure 17. 50000x2000 runtime analysis for sketch sparse general