SCAMP

Release 4.0.0

Zach Zimmerman

Jan 22, 2024

CONTENTS:

1 Introduction **1**

2 Motivation **3**

3 Why use SCAMP? **5**

4 When should you use SCAMP? **7**

5 When is SCAMP not the right choice? **9**

6 SCAMP Profile Types **11**

7 SCAMP CLI **13**
- 7.1 Building the SCAMP CLI . . . 13
- 7.2 Using the SCAMP CLI . . . 13
- 7.3 Input Format . . . 14
- 7.4 Required Arguments . . . 14
- 7.5 Common Optional Arguments . . . 14
- 7.6 Advanced Optional Arguments . . . 15
- 7.7 Build Configuration Options . . . 16
- 7.8 Specifying a Compliler . . . 16
- 7.9 Forcing CUDA (or No CUDA) . . . 16

8 Environment **17**
- 8.1 Notes on GPU Support . . . 17

9 Performance **19**
- 9.1 Notes on CPU performance . . . 19
- 9.2 Precomputation performance . . . 19
- 9.3 Benchmarks . . . 19
- 9.4 Performance Comparisons . . . 20
- 9.5 Performance Comparisons with other Matrix Profile Libraries . . . 20

10 Interpreting SCAMP Output **23**
- 10.1 Interpreting the Matrix Profile . . . 23
- 10.2 Missing Data . . . 23
- 10.3 Flat Regions . . . 23
- 10.4 SCAMP Output Precision . . . 24

11 Using SCAMP's Docker image **25**

12 pyscamp **27**
12.1 Installation . . . 27
12.2 Installation Troubleshooting . . . 28
12.3 pyscamp System Resource Usage . . . 29
12.4 Python Example . . . 29

13 pyscamp API documentation **31**
13.1 pyscamp: Python bindings for SCAMP . . . 31

14 Keyword Arguments for pyscamp Methods **39**

15 Distributed Operation **41**
15.1 Limitations . . . 42
15.2 Sharded implementation . . . 42

16 Frequently Asked Questions **43**
16.1 Missing libcufft.so on Linux . . . 43
16.2 CUDA/GPUs won't work . . . 43
16.3 SCAMP's output looks wrong . . . 43

17 Indices and tables **45**

Python Module Index **47**

Index **49**

CHAPTER

ONE

INTRODUCTION

SCAMP is a CPU/GPU implementation of the SCAMP algorithm. SCAMP takes a time series as input and computes the matrix profile for a particular window size. You can read more about the matrix profile at the Matrix Profile Homepage This is a much improved framework over GPU-STOMP which has the following additional features:

- Tiling for large inputs
- Computation in fp32, mixed fp32/fp64, or fp64 (double is recommended for most datasets, single precision will work for some)
- fp32 version should get good performance on GeForce cards
- AB joins (you can produce the matrix profile from 2 different time series)
- Distributable on the cloud.
- More types of matrix profiles! See *Profile Types*.
- Extremely Efficient Implementation
- Extensible to adding optimized versions of custom join operations.
- CPU Support (Only enabled for double precision; does not support KNN joins yet)
- Handles missing data as NaN input values. The matrix profile will be computed while excluding any subsequence with a NaN value
- Gracefully handles flat regions in data.
- Python module: Use SCAMP in Python with pyscamp
- conda-forge integration: Install pyscamp seamlessly with conda.
- Extensive testing: SCAMP has thousands of input configurations tested with every PR.
- Automatic benchmarking: Helps ensure performance doesn't slip with future updates.
- Extremely Efficient Implementation

CHAPTER

TWO

MOTIVATION

The matrix profile is expensive to compute. SCAMP aims to utilize specialized kernels and a tiled approach to create an extensible, scalable framework for computing the matrix profile.

CHAPTER

THREE

WHY USE SCAMP?

- It is *faster* than any other matrix profile library. It is 10x to 100x faster than almost all other implementations out there currently.
- It is very easy to install using conda and has very few dependencies.
- It handles real data: very large inputs, missing values, and flat regions with little issue.
- It can compute various other types of matrix profiles, including efficiently computing KNN matrix profiles, and matrix summaries (a.k.a. mplots). And can be extended to compute other types of profile efficiently.

CHAPTER

FOUR

WHEN SHOULD YOU USE SCAMP?

- You want to go fast. :)
- You want to compute very large matrix profiles. i.e. more than 50K-100K datapoints. The larger the dataset, the more advantage SCAMP has over other exact methods.
- You want to compute matrix profiles using an NVIDIA GPU. With a seamless install experience.
- You want a library that will handle real data.

CHAPTER

FIVE

WHEN IS SCAMP NOT THE RIGHT CHOICE?

- SCAMP does not currently support architectures other than x86_64 (sorry Apple M1 users, you'll need to build from source). SCAMP can build on other architectures but they are not explicitly supported. Eventually support will be added but it is not currently being worked on.
- SCAMP does not currently provide a rich API for doing things with the matrix profile once you have it. Some support for things like this is on the roadmap, but there are other libraries you can use for post processing in the meantime.
- You want to generate matrix profiles on edge devices (sensor systems, smartwatches, raspberry pis, smartphone, etc.), these devices usually have exotic architectures (eg. 32-bits or ARM) not fully supported by SCAMP. The preference on these systems is to do some kind of approximation to reduce power usage and save on-chip resources. You might try looking into LAMP for something like this.

CHAPTER

SIX

SCAMP PROFILE TYPES

SCAMP can compute various types of matrix profiles. Each has slightly different semantics and provide different 'views' of the distance matrix. All of these profile types support AB-joins.

Nearest Neighbor and Index:
This is the default profile type and the normal definition of matrix profile, it will produce the nearest neighbor distance/correlation of every subsequence as well as the index of the nearest neighbor

- CLI flag: `--profile_type=1NN_INDEX`
- pyscamp functions: selfjoin, abjoin

Nearest Neighbor Only:
This is a slightly faster version of the default profile type, but it only returns the nearest neighbor distance/correlation not the index of the nearest neighbor

- CLI flag: `--profile_type=1NN`
- pyscamp functions: Currently unsupported.

Sum of correlations above threshold [correlation histogram]:
Rather than finding the nearest neighbor, this profile type will compute the sum of the correlations above the specified threshold (`--threshold`) for each subsequence. This is like a frequency histogram of correlations.

- CLI flag: `--profile_type=SUM_THRESH`
- pyscamp functions: selfjoin_sum, abjoin_sum

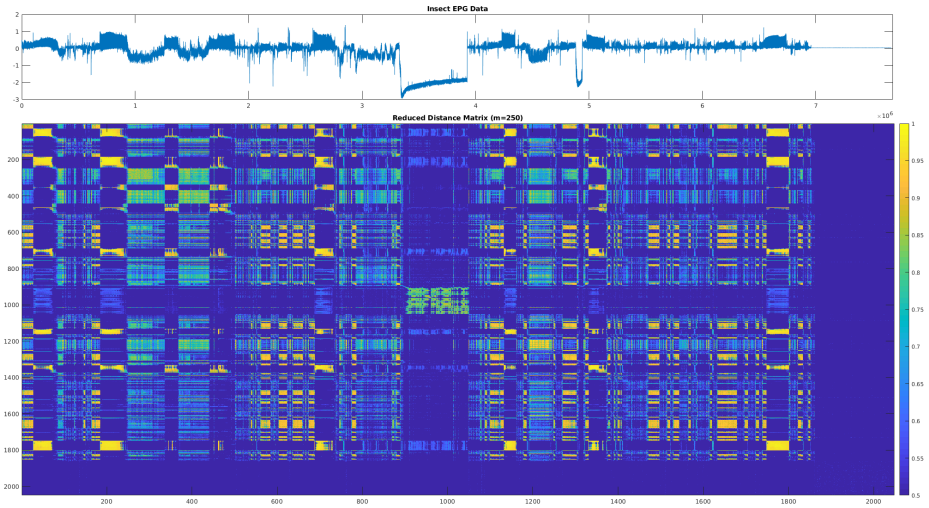
Approximate K nearest neighbors:
[EXPERIMENTAL, GPU ONLY, DISTRIBUTED UNSUPPORTED] This returns the approximate K (`--max_matches_per_column`) nearest neighbors and their correlations/indexes for each subsequence. A threshold (`--threshold`) can be used to accelerate the computation by ignoring matches below the threshold. This is an approximation, because the output may miss results which are too close together (up to ~1000 datapoints apart). The results will be provided in the output file where each row is a tuple of (subsequence index, match index, correlation/distance)

- CLI flag: `--profile_type=ALL_NEIGHBORS`
- pyscamp functions: selfjoin_knn, abjoin_knn

Pooled distance matrix summary:
[EXPERIMENTAL, DISTRIBUTED UNSUPPORTED] This returns a max-pooled summary (see example below) of the distance matrix using the specified summary matrix height (`--reduced_height`) and width (`--reduced_width`). When using GPUs there are limits to the resolution of the output. The output matrix height and width must be approximately 1000x smaller than the input size, otherwise you can get patchy results. If you have a small time series (~100K elements or less), it is recommended you use the CPU version for now as that is fast enough). Additionally, the entire output matrix must be small enough to fit in your system/GPU's memory.

- CLI flag: `--profile_type=MATRIX_SUMMARY`
- pyscamp functions: selfjoin_matrix, abjoin_matrix

Below is an example distance matrix summary, as you can see there is structure exposed via the visualization of the distance matrix.



CHAPTER

SEVEN

SCAMP CLI

SCAMP provides a command line interface which can be used to generate matrix profiles from ascii files. The command line interface provides the most flexibility in terms of how you can execute `SCAMP`, but `pyscamp`, is likely easier to interact with programmatically.

7.1 Building the SCAMP CLI

Clone the repository and submodules; make a build directory:

```
git clone https://github.com/zpzim/SCAMP
cd SCAMP
git submodule update --init --recursive
mkdir build && cd build
```

Build SCAMP on Mac/Linux/Windows:

```
# cmake will look in your $PATH for the cuda/c++ compilers
# If you have problems with cmake, you may need to specify a
# cuda or c++ compiler as shown above
cmake ..
cmake --build . --config Release
```

7.2 Using the SCAMP CLI

Once built, you can use the CLI as follows

For a traditional matrix profile:

```
./SCAMP --window=window_size --input_a_file_name=input_A_file_path
```

This will generate two files: mp_columns_out and mp_columns_out_index, which contain the matrix profile and matrix profile index values respectively.

For a sum/histogram profile:

```
./SCAMP --window=window_size --input_a_file_name=input_A_file_path \
   --profile_type=SUM_THRESH \
   --threshold=[optional, correlation threshold for matches]
```

This will generate one file: mp_columns_out, which contains histogram matrix profile (in sum of correlations above threshold).

For a knn matrix profile:

```
./SCAMP --window=window_size --input_a_file_name=input_A_file_path \
  --profile_type=ALL_NEIGHBORS --max_matches_per_column=[K] \
  --threshold=[optional, correlation threshold]
```

This will generate one file: mp_columns out, which contains the all matches discovered as [matrix col (subseq index), matrix row (match index), distance] tuples.

For a matrix summary:

```
./SCAMP --window=window_size --input_a_file_name=input_A_file_path \
  --profile_type=MATRIX_SUMMARY --reduced_height=[H] --reduced_width=[W] \
  --threshold=[optional, correlation threshold for matches]
```

This will generate one file: mp_columns_out, which contains the matrix summary.

7.3 Input Format

Input files should have one value per line. The parser expects newlines at the end of each value. You can find examples of this on github in the test/SampleInput directory.

7.4 Required Arguments

--input_a_file_name=[/path/to/file]:
The time series input to compute the matrix profile for, the input files should follow the format listed above.

--window=[subsequence length]:
The subsequence length to use for computing the matrix profile.

7.5 Common Optional Arguments

--gpus=[list of device numbers to use]:
Allows you to specify which gpus to use on the machine, by default we try to use all of them. The device numbers must be valid cuda devices on your system. You can chain these to add more gpus. Example: `--gpus="0 1"` will use gpu 0 and gpu 1 on the system.

--input_b_file_name=[/path/to/file]:
Allows a second input file which acts as the second time series for an AB join. An AB join compares every subsequence in input A with every subsequence in input B, the length of the matrix profile produced by this operation is always determined by input A, but the matrix profile index's values will reference subsequences in input B. Providing this parameter implies that SCAMP will compute an AB join.

--no_gpu:
SCAMP will not use GPUs even if they are available. Must be combined with `--num_cpu_workers` or no computation will happen.

--num_cpu_workers=[number of threads]:
Allows you to specify the number of cpu threads to compute with, by default we use none. For now, if you

don't have gpus, we recommend setting this to the number of cores on your system for best performance. It is possible to perform hetrogeneous GPU/CPU computation using this flag, but you will likely see very little speedup compared to GPU only as GPUs are much faster. If you have GPUs on your system and don't want to use them you should also use the `--no_gpu` argument.

`--output_a_file_name=[/path/to/output/file]`:
This is the name of the output file where SCAMP will write its output (except index values in 1NN_INDEX profiles) [default: mp_columns_out]

`--output_a_index_file_name=[/path/to/output/file]`:
This is the name of the output file where SCAMP will write the indexes for 1NN_INDEX profiles [default: mp_columns_out_index]

`--output_b_file_name=[/path/to/output/file]`:
Only used when `--keep_rows` is specified, this is the name of the file where SCAMP will write the row-wise matrix profile [default: mp_rows_out]

`--output_b_index_file_name=[/path/to/output/file]`:
Only used when `--keep_rows` is specified, this is the name of the file where SCAMP will write the row-wise matrix profile indexes [default: mp_rows_out_index]

`--output_pearson`:
SCAMP will output pearson correlation rather than z-normalized euclidean distance.

`--print_debug_info`:
By default SCAMP runs in silent_mode with no output, this option prints debugging information to stdout.

`--profile_type=[Type of profile to compute]`:
Determines the type of matrix profile to compute. See the examples above and in *Profile Types*.

`--reduced_height=[height of output matrix]`:
For matrix summary profiles, the height of the reduced resoulution distance matrix output.

`--reduced_width=[width of output matrix]`:
For matrix summary profiles, the width of the reduced resoulution distance matrix output.

`--threshold=[correlation threshold in the interval [0,1] ]`:
For sum / histogram / knn / matrix_summary profiles. Correlations below this value will be ignored from the final output.

7.6 Advanced Optional Arguments

One of [`--ultra_precision`, `--double_precision`, `--mixed_precision`, `--single_precision`]:
Changes the precision mode of SCAMP, default is double precision (and is the only one available on CPU joins), mixed precision will work on many datasets but not all, single precision will work for some simple datasets, but may prove unreliable for many. See test/SampleInput/earthquake_precision_test.txt for an example of a dataset that fails in mixed/single precision. The single precision mode is about 2x faster than double precision, mixed_precision falls in the middle, but can sometimes be as slow as double precision".

Ultra Precision uses double precision everywhere and also computes the norms of each subsequence during the precomputation step with a more precise, but potentially slower formula with complexity **O(len(timeseries)*sublen)**, this should be tried if you run into issues with the standard method in double precison and need more precise calculations. Also uses a new, more stable formula for computing the matrix profile.

`--aligned`:
For ab-joins only, indicates that A and B may start with the same sequence and must consider an exclusion zone.

--keep_rows:
Informs SCAMP to compute the "rowwise mp" and output in a a separate file specified by the flag `--output_b_file_name`.

1. In self-joins, specifying this flag results in "output_a_file_name" containing the right matrix profile and "output_b_file_name" containing the left, these can be used to compute time series chains.
2. This is also useful when computing a distributed self-join, so as to not recompute values in the lower-trianglular portion of the symmetric distance matrix.

--global_col=[global col of the distance matrix in a distributed join]:
Informs SCAMP that this join is part of a larger distributed join which starts at this column in the larger distance matrix, this allows us to pick an appropriate exclusion zone for our computation if necessary.

--global_row=[global row of the distance matrix in a distributed join]:
Informs SCAMP that this join is part of a larger distributed join which starts at this row in the larger distance matrix, this allows us to pick an appropriate exclusion zone for our computation if necessary.

--max_tile_size=[integer tile size]:
Allows you to specify the max tile size used by the SCAMP tile scheme. By default this is set to 128K, but you can adjust this as desired. Note that a tile size smaller than 128K will likely fail to saturate the compute resources of newer GPUs

7.7 Build Configuration Options

7.8 Specifying a Compliler

You can set the environment variables `CUDACXX=/path/to/nvcc`, `CXX=/path/to/cpp/compiler`, and `CC=/path/to/c/compiler` to manually specify a compiler. By default cmake will look for cuda at the /usr/local/cuda symlink on linux and mac.

On Windows you may also need to specify the generator to cmake:

```
# Build with Visual Studio 2015 tools
cmake -G "Visual Studio 14 2015" ..
# Build with Ninja (requires ninja)
cmake -G "Ninja"
```

Windows CUDA builds will only work using Visual Studio tools (and the CUDA visual studio extensions). This is due to the fact that the visual studio toolchain is the only suppored toolchain for compiling cuda on windows, changing the C++ compiler will cause nvcc to fail. Therefore you can only use other generators for C++ only builds.

7.9 Forcing CUDA (or No CUDA)

If you desire explicit CUDA support, you can make the build fail using the flag FORCE_CUDA=1 if cuda is not found:

```
cmake -DFORCE_CUDA=1 ..
```

The same is true if you want to disable CUDA support using FORCE_NO_CUDA=1, this will cause CUDA not to be used, even if it is found on the system:

```
cmake -DFORCE_NO_CUDA=1 ..
```

CHAPTER

EIGHT

ENVIRONMENT

Currently builds under Windows/Mac/Linux using msvc/gcc/clang and nvcc (if CUDA is available) with cmake.

Base dependancies (required for all builds of SCAMP):

- cmake 3.18 or greater
  - This version is not available directly from all package managers so you may need to install it manually, the easist way to do this is with python via `pip install cmake` or you can download it manually from here
- C/C++ compiler (e.g. gcc/clang/Visual Studio Build tools)
- SCAMP is only tested currently on x86_64 systems. 32-bit systems are not supported. Though SCAMP may build on them, other 64-bit architectures are not currently tested or optimized for.

For GPU support (required for any SCAMP build which will use a GPU):

- cuda toolkit v11.0 or greater
  - Available here
- NVIDIA GPU with CUDA (compute capability 3.5+) support.
  - You can find a list of CUDA compatible GPUs here
  - Highly recommend using a Pascal/Volta or newer GPU as they are much better (V100 is ~10x faster than a K80 for SCAMP, V100 is ~2-3x faster than a P100)

For python support:

- Only Python 3 is supported.

Recommended Compiler:

- If you are using CPUs, using a newer version of clang is recommended as it tends to have better performance.

8.1 Notes on GPU Support

You need to have a cuda development environment set up in order to build SCAMP with GPU support. If you install SCAMP (or pyscamp) and it does not detect CUDA during installation it will install using CPU support only. cmake must detect your cuda installation, this can be especially tricky when using Windows and MSVC as you need to have the CUDA extensions for visual studio installed.

I have only gotten Windows CUDA builds to work under MSVC and the Visual Studio Generators. There are some issues with cmake/nvcc/msvc that make it very difficult to install outside of this configuration.

You can use the *configuration option* FORCE_CUDA=1, to force SCAMP to build with CUDA (or fail). This works when installing pyscamp as well using `FORCE_CUDA=1 pip install pyscamp`.

CHAPTER

NINE

PERFORMANCE

SCAMP is extremely fast, especially on Tesla series GPUs. I belive this repository contains the fastest code in existance for computing the matrix profile. If you find a way to improve the speed of SCAMP, or compute matrix profiles any faster than SCAMP does, please let me know, I would be glad to point to your work and incorporate any improvements that can be made to SCAMP.

9.1 Notes on CPU performance

SCAMP's CPU performance is very good. However, how performant it is depends heavily on the compiler and compiler flags used to build the SCAMP binary. Newer compilers are better at autovectorization, clang newer than v6 and gcc newer than v7 are best. MSVC tends to be much slower. Even with microoptimizations to generate better code based on the compiler, there can be up to a 10x (perhaps more) difference depending on the compiler you use. Though most of the time the variance is in the ballpark of less than 2x-3x difference.

The distributed pyscamp-gpu and pyscamp-cpu conda packages should have consistent good performance, as they are built with a modern compiler.

9.2 Precomputation performance

When enabling the `--ultra_precision` flag in the SCAMP CLI, or specifying the `precision='ultra'` option in pyscamp, the method for precomputing the necessary statisics for the matrix profile computation uses an O(nm) algorithm to compute the subsequence means and norms. This computation can become a bottleneck if you specify an extremely large subsequence length.

The timing results below do not use this option. All experiments were performed in double precision.

9.3 Benchmarks

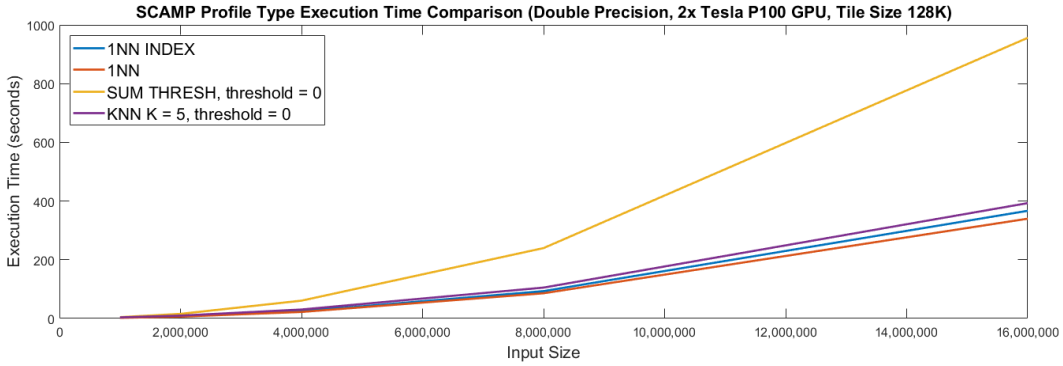
SCAMP has automated benchmarks running. Here is a link to recent GPU performance results:

- GPU NVIDIA Tesla P100 (1x), Input length 512K datapoints, default parameters
- CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz, Single Thread, Input length 32K datapoints, default parameters

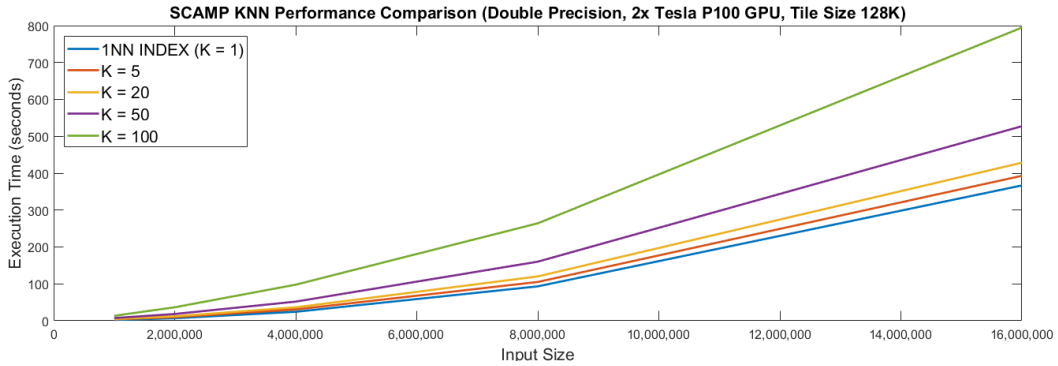
Note that the charts are not totally optimized for human consumption yet. But you can see a benchmark of each profile type.

9.4 Performance Comparisons

The included performance tests showcase SCAMP's performance up to an input size of 16M datapoints; however, as we have shown in our publications SCAMP is scalable to hundreds of millions of datapoints and even billions of datapoints with the right hardware.



In the figure above we show the runtime in seconds for SCAMP's various profile types (self-join) on 2 P100 GPUs.



In the figure above we show the runtime in seconds for SCAMP's approximate KNN (`--profile_type=ALL_NEIGHBORS`) matrix profile, while varying K and the input size on 2 P100 GPUs.

You can see that SCAMP maintains good performance relative to the baseline 1NN_INDEX matrix profile up to at least K=20, which should be sufficient for almost all practioners. All measurements were made with random data with the initial threshold set to 0 correlation (close to the worst case for KNN).

9.5 Performance Comparisons with other Matrix Profile Libraries

As mentioned before, SCAMP is extremely fast. This section contains experiments comparing other libraries to SCAMP in terms of performance to show quantitatively how fast SCAMP is. Note that these numbers reflect the performance of these libraries at a snapshot in time (June 2022) and implementations can change. If you want to reproduce these results, or generate new performance numbers in the future, the scripts used to generate the tables below are provided in the SCAMP repository here.

System 1	• WSL Ubuntu under Windows 11 • CPU: Intel(R) Core(TM) i9-10850K CPU @ 3.60GHz • GPU: NVIDIA GeForce RTX 3080
System 2	• Linux Ubuntu 18.04 • CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz • GPU: 2x NVIDIA Tesla P100

Note: Both CPUs have SSE2/AVX/AVX2/FMA enabled.

9.5.1 pyscamp vs stumpy Performance

stumpy is a very popular matrix profile library which has reimplemented many of the algorithms published by Eamonn's time series lab at UC Riverside.

stumpy claims to have superior performance for the matrix profile algorithms they have reimplemented. However, these performance comparisons are done in bad faith. They compare across several generations of GPU/CPU hardware instead of making a fair apples to apples comparison, they simply copy numbers published in our papers and use hardware that is multiple generations newer, and throw many times more resources at the problem.

I contacted the stumpy maintainer years ago asking for these bad faith comparisons to be removed, but they refused. To set the record straight, here is a fair comparison of pyscamp and stumpy done on the same system.

The tables below shows that pyscamp is faster than stumpy by a factor of 20x or more on the CPU, 6x faster on GeForce GPUs, and 60x faster using Tesla Series GPUs (pyscamp is even faster here because the bottleneck on GeForce cards is fp64 compute, GeForce cards are optimized for lower precision computation). Pyscamp has a single precision mode for GPU compute which makes GeForce performance better, but this is not reported to keep the playing field level.

Both systems are using the following dependencies installed from conda-forge: `pyscamp-gpu v4.0.0` `stumpy v1.11.1` `python v3.9.12` `cudatoolkit v11.6.0` `numba v0.55.1` `numpy v1.21.6` `scipy v1.8.1`

All GPU compute is done in FP64, FP32 numbers aren't reported.

pyscamp vs stumpy (System 1: 20 logical cores, 1x GeForce RTX 3080)

input_size	stumpy CPU	pyscamp CPU	pyscamp CPU speedup	stumpy GPU	pyscamp GPU	pyscamp GPU speedup
8192	0.027	0.009	3.037	1.138	0.06	18.984
16384	0.1	0.018	5.561	2.415	0.119	20.309
32768	0.388	0.053	7.324	5.011	0.231	21.725
65536	1.662	0.186	8.951	9.749	0.434	22.439
131072	9.178	0.767	11.963	18.268	1.068	17.101
262144	48.947	2.804	17.456	36.344	2.576	14.107
524288	272.687	11.266	24.204	72.449	7.59	9.545
1048576	Not Computed	44.647	Not Computed	144.59	24.86	5.816
2097152	Not Computed	184.232	Not Computed	539.172	88.054	6.123

pyscamp vs stumpy (System 2: 12 logical cores, 2x Tesla P100)

input_size	stumpy CPU	pyscamp CPU	pyscamp CPU speedup	stumpy GPU	pyscamp GPU	pyscamp GPU speedup
8192	6.812	0.01	669.938	1.747	0.01	181.297
16384	0.189	0.03	6.232	2.26	0.016	141.965
32768	0.748	0.102	7.331	3.356	0.036	92.378
65536	3.075	0.399	7.709	5.31	0.055	95.707
131072	13.951	1.487	9.382	9.711	0.101	96.535
262144	72.197	5.922	12.192	18.185	0.221	82.26
524288	367.535	22.867	16.073	36.501	0.573	63.678
1048576	Not Computed	93.814	Not Computed	111.844	1.958	57.11
2097152	Not Computed	359.328	Not Computed	439.938	7.151	61.524

9.5.2 pyscamp vs MPF

Matrix Profile Foundation's matrixprofile is a python library which implements many of the matrix profile algorithms. It only supports CPU computation.

There are two algorithms in this library compared against:

- **MPX**: The mpx algorithm implemented in this library is very similar to what SCAMP uses and is also highly optimized, hence performance is similar here.
- **SCRIMP++**: I show SCRIMP++ performance here for comparison even though it is an approximate algorithm and could be made faster by changing parameters. It is a common misconception that SCRIMP++ is always faster than exact algorithms like mpx and pyscamp. There are overheads assoicated with SCRIMP++ that have high constant factor overhead (e.g. repeated FFT computation) which high-performing exact algorithms like pyscamp don't have. This can make pyscamp competetive with SCRMP++ in all but the most highly approximated scenarios.

Comparisons were done with 20 threads, SCRIMP++ was configured with 10% sampling and 25% step.

Packages installed: `pyscamp-gpu v4.0.0` `matrixprofile v1.1.10` `python v3.8.13` `numpy v1.22.4` `scipy v1.8.1`

pyscamp vs mpf (System 1: 20 logical cores, 1x GeForce RTX 3080)

input_size	mpx CPU	scrimp++ CPU	pyscamp CPU	pyscamp CPU speedup vs mpx	pyscamp CPU speedup vs scrimp++
8192	0.013	0.207	0.007	1.87	30.494
16384	0.046	0.683	0.017	2.686	39.988
32768	0.151	2.442	0.054	2.814	45.474
65536	0.59	9.307	0.189	3.124	49.245
131072	2.215	43.715	0.701	3.162	62.403
262144	10.846	182.173	2.793	3.883	65.219
524288	69.426	884.057	11.046	6.285	80.037

CHAPTER

TEN

INTERPRETING SCAMP OUTPUT

10.1 Interpreting the Matrix Profile

When using z-normalized euclidean distance (the default). Peaks in the matrix profile represent time series discords (anomalies) and valleys represent motifs in the data.

When using Pearson Correlation (CLI flag: `--output_pearson`, pyscamp argument: `pearson=True`), then the opposite is true; Peaks in the matrix profile correspond to motifs and valleys correspond to discords.

Please see the Matrix Profile Tutorial (slides available here for more information on how to interpret and use the matrix profile.

For more research background on the matrix profile you can find links to the research on the Matrix Profile Homepage.

10.2 Missing Data

Missing data is specified using NaNs. Any subsequence containing NaNs will not match any other subsequence. The nearest neighbor of a subsequence containing NaN is defined as NaN. If there is no match found for a particular subsequence, it's nearest neighbor index will be set to -1.

10.3 Flat Regions

Some datasets contain flat regions. In SCAMP we consider a flat region any region of the time series input that remains constant for at least one whole subsequence. Constant regions have a mean-centered norm of 0, which means that their Pearson Correlation and z-normalized euclidean distance are **undefined**. Totally flat regions will not match any subsequence and will output NaN as their nearest neighbor distance.

There can also be regions that are **almost flat**, these regions contain values that are on average very close to the mean value of the subsequences and so have a mean-centered norm which is very small. These sequences can be problematic as floating point roundoff errors can cause problems with SCAMP's output. Internally SCAMP produces the Pearson Correlation by dividing intermediate output by these precomputed norms. If a norm is extremely small, this can cause large floating point roundoff errors. If you are using Pearson Correlation and you encounter a nearest neighbor with correlation greater than 1.0, this is likely the cause for that.

We have tried to prevent catastrophic roundoff error in SCAMP by using an epsilon (currently hardcoded to 1e-13). If the sum of squared error from the mean for the values in a particular subsequence is less than this epsilon, then SCAMP considers the subsequence totally flat (as defined above). This will cause NaN to be output for that subsequence.

By default we use compensated arithemtic to compute the subsequence means and a fast method to compute the subsequence norms during precomputation. Optionally, you can use a high-precision brute force O(nm) approcach for computing the subsequence norms and means. With this approach precomputation can be expensive when a very large

subsequence length is chosen in combination with a large input size. If you would like to use this method of computation in exchange for speed you can specify (CLI flag: `--ultra_precision`, pyscamp arg: `precision='ultra'`) as an option to use this slower, more precise method. Using the Ultra precision option also enables a new more stable update formula.

10.4 SCAMP Output Precision

While distance computation is done in the recommended 64 bit precision by default. Most of SCAMP's profile types output with 32 bit precision. Intermediate output is compared with 32-bit precision for many of the profile types, therefore SCAMP can only output in 32-bit precision.

Due to the above any nearest neighbors correlations/distances that are indistinguishable in 32 bit precision could be output as the nearest neighbor in SCAMP.

CHAPTER

ELEVEN

USING SCAMP'S DOCKER IMAGE

Rather than building from scratch you can run SCAMP via nvidia-docker using the prebuilt image on dockerhub.

The docker image uses an ideal environment for SCAMP and builds the project with a new compiler, both CPU/GPU performance should be optimal in the docker container.

The docker image is useful if you want to utilize SCAMP in a distributed environment and start Linux VMs which can run SCAMP as needed.

In order to expose the host GPUs nvidia-docker must be installed correctly. Please follow the directions provided on the nvidia-docker github page. The following example uses docker 19.03 functionality:

```
docker pull zpzim/scamp:latest
docker run --gpus all \
  --volume /path/to/host/input/data/directory:/data \
  --volume /path/to/host/output/directory:/output \
  zpzim/scamp:latest /SCAMP/build/SCAMP \
    --window=<window_size> --input_a_file_name=/data/<filename> \
    --output_a_file_name=/output/<mp_filename> \
    --output_a_index_file_name=/output/<mp_index_filename>
```

CHAPTER

TWELVE

PYSCAMP

`pyscamp` is a python module which uses the SCAMP CUDA/C++ library to compute the matrix profile efficiently, inheriting the speed and functionality of SCAMP.

12.1 Installation

There are two ways to install pyscamp, via its conda package, or directly from source.

12.1.1 Installing on conda-forge

In an anaconda environment, you can install the pyscamp conda packages as follows:

To install pyscamp with gpu support (Windows/Linux) `conda install -c conda-forge pyscamp-gpu`.

To install pyscamp without gpu support (Windows/Linux/MacOS) `conda install -c conda-forge pyscamp-cpu`.

Note that `pyscamp-gpu` can be installed and used even if you don't have a GPU, it will simply fall back to using your CPU. However, `pyscamp-cpu` is preferrable if you don't have a GPU because it builds with a newer compiler and does not require installing the `cudatoolkit` depencency.

If you run into problems using GPUs with `pyscamp-gpu` make sure your NVIDIA drivers are up to date. This is the most common cause of issues.

12.1.2 Installing from source

You can install pyscamp from source to maximize performance on your system. A source distribution for pyscamp is available on pypi.org.

The python module supports python3 and requires that cmake 3.15 or greater is installed in order to build properly.

You can install cmake using `pip install cmake`

If you want GPU support for pyscamp, you must have CUDA installed and available to the default cmake compiler on your system.

Please look at the *environment setup guide* for more information about how to set up the environment for SCAMP.

Once you have cmake and/or cuda installed, you can use `pip install pyscamp` to download and install the module.

pyscamp allows you to specify certain build options to pip using environment variables:

- `FORCE_CUDA=1` will allow you to force pyscamp to build with cuda (or fail if it can't find CUDA)
- `FORCE_NO_CUDA=1` will force pyscamp to build without cuda, even if it is found on the system.

- CC=<compiler> will force pyscamp to use a c compiler other than the default compiler detected by cmake, this works only for builds not using visual studio tools.
- CXX=<compiler> will force pyscamp to use a c++ compiler other than the default compiler detected by cmake, this works only for builds not using visual studio tools
- CUDACXX=<nvcc location> will point pyscamp to the specified cuda compiler to build with, useful if your cuda compiler is not detected. This works only for builds not using visual studio tools
- CMAKE_GENERATOR=<generator tag ("Ninja", etc.)> useful for windows builds to specify a generator other than visual studio tools
- CMAKE_GENERATOR_TOOLSET=<toolset tag ("llvm", etc.)> for specifying a non-default compiler toolchain when using visual studio tools
- CMAKE_GENERATOR_PLATFORM=<platform tag (x64)> for specifying a non-default platform arch when using visual studio tools.
- PYSCAMP_USE_EXTERNAL_PYBIND11=ON pyscamp will install using the pybind11 package installed on the system. Used to build the conda-forge package.
- PYSCAMP_PYTHON_EXECUTABLE_PATH=<Path to the python executable to build for> defaults to the execuatable invoking setup.py. If you want to build pyscamp for a specific python executable you can point this env variable to it.
- PYSCAMP_ADD_CMAKE_ARGS=<args> Allows for passing additional cmake arguments during pyscamp's build.

Example:

```
# Force a cuda build using clang++ (on the system path) as the cpp compiler
FORCE_CUDA=1 CC=clang CXX=clang++ pip install pyscamp
```

Installation Notes: If you do not specify any of the above options, cmake will use the default compilers available to it, will try to find cuda and install GPU support if it is found.

12.2 Installation Troubleshooting

If you have problems building the module (or getting GPU support to work) please submit an issue on github. I don't have access to all build environments so help in addressing these issues is appreciated.

Figuring out what went wrong: You can use `pip install -v pyscamp` to print the output of the cmake configuration and build.

Getting CUDA to work:

When installing from conda-forge: If you have installed the `pyscamp-gpu` conda-forge package and you are having trouble with CUDA the most common issue is that the NVIDIA drivers on the system need to be updated to work with the newest versions of CUDA. Please try to update your GPU drivers.

When building from source: Ensure pyscamp is built with cuda using `FORCE_CUDA=1 pip install -I --no-cache-dir pyscamp`. If this fails, that means cmake was unable to detect your cuda installation or it wasn't new enough (see *environment setup guide* for which versions of cuda are supported). Some general troubleshooting steps you can try are:

- If you installed pyscamp previously and you have since installed cuda, make sure to add the `-I` and `--no-cache-dir` flags to pip install just to make sure you are reinstalling correctly.
- Use `pip install -v` to get more information about the build configuration and make sure it is using the compilers and cuda like you expect.

- On Mac/Linux make sure nvcc (the CUDA compiler, usually located at /usr/local/cuda/bin), is in your PATH. You can also specify a cuda compiler using `CUDACXX=/path/to/cuda/compiler pip install pyscamp`
- On Windows, I have only gotten CUDA to work using using the visual studio toochains **with the appropriate visual studio plugins installed** so make sure cuda is installed with these plugins. (see *GPU support* for more information and links to the cuda installation guide)
  - This means that it is not currently possible to use a compiler other than MSVC to build SCAMP with CUDA support on Windows.

Using a different compiler:

- On Mac/Linux: You can install clang v6 or greater and point pyscamp to it using `CXX=path/to/compiler pip install pyscamp`
- On Windows: You can use Ninja (or another generator) to build with `CMAKE_GENERATOR=Ninja CXX=path/to/compiler pip install pyscamp`

12.3 pyscamp System Resource Usage

When a pyscamp method is invoked with the default arguments. The following logic is followed to determine how to use resources on the system:

1. Check if GPUs are available, if so use them, do not use CPU resources to do compute heavy work.
2. If GPUs are not available, pyscamp will use cpu threads equal to the number of available cores to do compute work.

This logic is followed by default, but can be changed with the `gpus` and `threads` pyscamp kwargs:

- If you want to opt out of gpu execution, specify an empty list e.g. `gpus=[]`.
- If you want to use a non-default number of threads, specify the number in `threads=N`. Note that this is not recommended when GPUs are being used by default, so you should also specify `gpus=[]` so that you don't mix CPU/gpu resources. The only exception to this is if you want to use all resources available to compute results on a very large input. Otherwise, mixing cpu/gpu resources will probably end up slower than simply using GPU resources alone.

12.4 Python Example

```
import pyscamp as mp

# Allows checking if pyscamp was built with CUDA and GPUs are available.
has_gpu_support = mp.gpu_supported()

# Self join.
profile, index = mp.selfjoin(a, sublen)
# AB join using 4 threads and no gpus.
profile, index = mp.abjoin(a, b, sublen, threads=4, gpus=[])
# Sum thresh
corr_sum = mp.abjoin_sum(a, b, sublen, threshold=0.9)

# Matrix summary (100x100) with threVshold, outputting pearson correlation
matrix = mp.abjoin_matrix(a, b, sublen, mwidth=100, mheight=100, threshold=0.5,␣
↪pearson=True)
```

(continues on next page)

(continued from previous page)

```
# Approximate KNN is supported with GPUs + CUDA only for now.
if has_gpu_support:
  knn = mp.selfjoin_knn(a,sublen, k)
  # KNN with threshold
  knn = mp.selfjoin_knn(a, sublen, k, threshold=0.85)
  # KNN Ab join with threshold, outputting pearson correlation
  knn = mp.abjoin_knn(a, b, sublen, k, threshold=0.90, pearson=True)
```

CHAPTER

THIRTEEN

PYSCAMP API DOCUMENTATION

13.1 pyscamp: Python bindings for SCAMP

selfjoin(a, m, **kwargs)	Computes the matrix profile for time series A.
abjoin(a, b, m, **kwargs)	For each subsequence in time series A, finds the nearest neighbor in time series B.
selfjoin_sum(a, m, **kwargs)	Returns the sum of the correlations above specified threshold (default 0) for each subsequence in a time series.
abjoin_sum(a, b, m, **kwargs)	For each subsequence in time series a, returns the sum of the correlations to subsequences in time series b above specified threshold (default 0).
selfjoin_knn(a, m, k, **kwargs)	[GPU ONLY, EXPERIMENTAL] Returns the approximate k nearest neighbors for each subsequence in a time series
abjoin_knn(a, b, m, k, **kwargs)	[GPU ONLY, EXPERIMENTAL] For each subsequence in time series A, returns its Approximate K nearest neighbors in time series B
selfjoin_matrix(a, m, **kwargs)	[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance
abjoin_matrix(a, b, m, **kwargs)	[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance

13.1.1 pyscamp.selfjoin

pyscamp.**selfjoin**(*a: List[float]*, *m: int*, ***kwargs*) → Tuple[numpy.ndarray[numpy.float32], numpy.ndarray[numpy.int32]]

Computes the matrix profile for time series A.

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.

Returns
A tuple containing the matrix profile as the first element and the indices as a the second element.

Return type
Tuple of np.ndarray[float32] and np.ndarray[int32]

13.1.2 pyscamp.abjoin

pyscamp.**abjoin**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → Tuple[numpy.ndarray[numpy.float32], numpy.ndarray[numpy.int32]]

For each subsequence in time series A, finds the nearest neighbor in time series B.

Parameters

- **a** (*1D array*) – Time series, b will be queried for subsequences in a.
- **b** (*1D array*) – Time series in which to search for matches for subsequences in a.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.

Returns
A tuple. First element: The nearest neighbor distance of subsequences in a to time series b. Second element: The index (in b) of each nearest neighbor.

Return type
Tuple of np.ndarray[float32] and np.ndarray[int32]

13.1.3 pyscamp.selfjoin_sum

pyscamp.**selfjoin_sum**(*a: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float64]

Returns the sum of the correlations above specified threshold (default 0) for each subsequence in a time series.

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
For each subsequence in A, returns the sum of correlations above the the specified threshold to other subesequences in A.

Return type
np.ndarray[float64]

13.1.4 pyscamp.abjoin_sum

pyscamp.**abjoin_sum**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float64]

For each subsequence in time series a, returns the sum of the correlations to subsequences in time series b above specified threshold (default 0).

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **b** (*1D array*) – Time series to search for matches.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns

For each subsequence in A, returns the sum of correlations above the the specified threshold in B.

Return type

np.ndarray[float64]

13.1.5 pyscamp.selfjoin_knn

pyscamp.**selfjoin_knn**(*a: List[float]*, *m: int*, *k: int*, ***kwargs*) → List[Tuple[int, int, float]]

[GPU ONLY, EXPERIMENTAL] Returns the approximate k nearest neighbors for each subsequence in a time series

Parameters

- **a** (*1D array*) – Time series to compute the KNN matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **k** (*int*) – Number of neighbors to return for each subsequence
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns

List of tuples (col, row, distance) containing the matches (up to K) for each column of the distance matrix, row is the index of the match, and d is the distance between the two subsequences

Return type

List of tuple[int, int, float]

13.1.6 pyscamp.abjoin_knn

pyscamp.**abjoin_knn**(*a: List[float]*, *b: List[float]*, *m: int*, *k: int*, ***kwargs*) → List[Tuple[int, int, float]]

[GPU ONLY, EXPERIMENTAL] For each subsequence in time series A, returns its Approximate K nearest neighbors in time series B

Parameters

- **a** (*1D array*) – Time series to compute the KNN matrix profile for.`
- **b** (*1D array*) – Time series in which to search for matches.

- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **k** (*int*) – Number of neighbors to return for each subsequence
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
List of tuples (col, row, distance) containing the matches (up to K) for each column of the distance matrix, col is the index in A, row is the index in B of the match, and d is the distance between the two subsequences

Return type
List of tuple[int, int, float]

13.1.7 pyscamp.selfjoin_matrix

pyscamp.**selfjoin_matrix**(*a: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float32]

[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **mheight** (*int, optional*) – Height of the pooled distance matrix to output. Default 50
- **mwidth** (*int, optional*) – Width of the pooled distance matrix to output. Default 50
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
A 2D array of height of mheight and width of mwidth. This is a pooled version of the full distance matrix.

Return type
2D array

13.1.8 pyscamp.abjoin_matrix

pyscamp.**abjoin_matrix**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float32]

[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance

Parameters

- **a** (*1D array*) – Time series corresponding to the columns of the distance matrix.
- **b** (*1D array*) – Time series corresponding to the rows of the distance matrix.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **mheight** (*int, optional*) – Height of the pooled distance matrix to output. Default 50
- **mwidth** (*int, optional*) – Width of the pooled distance matrix to output. Default 50
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
A 2D array of height of mheight and width of mwidth. This is a pooled version of the full distance matrix.

Return type
2D array

pyscamp.**abjoin**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → Tuple[numpy.ndarray[numpy.float32], numpy.ndarray[numpy.int32]]

For each subsequence in time series A, finds the nearest neighbor in time series B.

Parameters

- **a** (*1D array*) – Time series, b will be queried for subsequences in a.
- **b** (*1D array*) – Time series in which to search for matches for subsequences in a.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.

Returns
A tuple. First element: The nearest neighbor distance of subsequences in a to time series b. Second element: The index (in b) of each nearest neighbor.

Return type
Tuple of np.ndarray[float32] and np.ndarray[int32]

pyscamp.**abjoin_knn**(*a: List[float]*, *b: List[float]*, *m: int*, *k: int*, ***kwargs*) → List[Tuple[int, int, float]]

[GPU ONLY, EXPERIMENTAL] For each subsequence in time series A, returns its Approximate K nearest neighbors in time series B

Parameters

- **a** (*1D array*) – Time series to compute the KNN matrix profile for.`
- **b** (*1D array*) – Time series in which to search for matches.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **k** (*int*) – Number of neighbors to return for each subsequence
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
List of tuples (col, row, distance) containing the matches (up to K) for each column of the distance matrix, col is the index in A, row is the index in B of the match, and d is the distance between the two subsequences

Return type
List of tuple[int, int, float]

pyscamp.**abjoin_matrix**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float32]

[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance

Parameters

- **a** (*1D array*) – Time series corresponding to the columns of the distance matrix.
- **b** (*1D array*) – Time series corresponding to the rows of the distance matrix.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **mheight** (*int, optional*) – Height of the pooled distance matrix to output. Default 50

- **mwidth** (*int, optional*) – Width of the pooled distance matrix to output. Default 50
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
A 2D array of height of mheight and width of mwidth. This is a pooled version of the full distance matrix.

Return type
2D array

pyscamp.**abjoin_sum**(*a: List[float]*, *b: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float64]

For each subsequence in time series a, returns the sum of the correlations to subsequences in time series b above specified threshold (default 0).

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **b** (*1D array*) – Time series to search for matches.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
For each subsequence in A, returns the sum of correlations above the the specified threshold in B.

Return type
np.ndarray[float64]

pyscamp.**gpu_supported**() → bool

Returns true if both 1) The module was compiled with GPU support and 2) GPUs are available.

pyscamp.**selfjoin**(*a: List[float]*, *m: int*, ***kwargs*) → Tuple[numpy.ndarray[numpy.float32], numpy.ndarray[numpy.int32]]

Computes the matrix profile for time series A.

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.

Returns
A tuple containing the matrix profile as the first element and the indices as a the second element.

Return type
Tuple of np.ndarray[float32] and np.ndarray[int32]

pyscamp.**selfjoin_knn**(*a: List[float]*, *m: int*, *k: int*, ***kwargs*) → List[Tuple[int, int, float]]

[GPU ONLY, EXPERIMENTAL] Returns the approximate k nearest neighbors for each subsequence in a time series

Parameters

- **a** (*1D array*) – Time series to compute the KNN matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **k** (*int*) – Number of neighbors to return for each subsequence

- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
List of tuples (col, row, distance) containing the matches (up to K) for each column of the distance matrix, row is the index of the match, and d is the distance between the two subsequences

Return type
List of tuple[int, int, float]

pyscamp.**selfjoin_matrix**(*a: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float32]

[EXPERIMENTAL] Returns a pooled version of the distance matrix with HxW of [mheight x mwidth], pooling operation is max() for Pearson Correlation and min() for Euclidian Distance

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **mheight** (*int, optional*) – Height of the pooled distance matrix to output. Default 50
- **mwidth** (*int, optional*) – Width of the pooled distance matrix to output. Default 50
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
A 2D array of height of mheight and width of mwidth. This is a pooled version of the full distance matrix.

Return type
2D array

pyscamp.**selfjoin_sum**(*a: List[float]*, *m: int*, ***kwargs*) → numpy.ndarray[numpy.float64]

Returns the sum of the correlations above specified threshold (default 0) for each subsequence in a time series.

Parameters

- **a** (*1D array*) – Time series to compute matrix profile for.
- **m** (*int*) – Subsequence length to use for computing the matrix profile.
- **threshold** (*float, optional*) – Correlation threshold [0,1] (Default 0), matches which have a correlation less than the threshold will be ignored

Returns
For each subsequence in A, returns the sum of correlations above the the specified threshold to other subesequences in A.

Return type
np.ndarray[float64]

CHAPTER

FOURTEEN

KEYWORD ARGUMENTS FOR PYSCAMP METHODS

pyscamp methods support several different keyword arguments.

threshold=[float]:
Distance threshold used for various profile types, correlations found below this threshold will be ignored

pearson=[bool]:
Output Pearson Correlation rather than Z-normalized euclidean distance

threads=[int]:
Number of CPU threads to use with SCAMP (if using gpus it is recommended to not use this flag). If you want to prevent GPUs from being used, pass gpus=[] as a kwarg.

gpus=[list of integers]:
Cuda device ids of gpus to run on, by default we run on all gpus if you have any. To opt out of gpu execution, specify an empty list here.

precision=[string]:
One of ['single', 'mixed', 'double', 'ultra'] default is double precision, ultra and double precision are supported on CPU and GPU, mixed and single precision are only supported on GPU.

mwidth=[int]:
For matrix summaries, the width of the output matrix (default 50)

mheight=[int]:
For matrix summaries, the height of the output matrix (default 50)

verbose=[bool]:
Enable verbose output. This will log to stdout. (default False)

CHAPTER

FIFTEEN

DISTRIBUTED OPERATION

SCAMP has a client/server architecture built using grpc. Tested on GKE but should be possible to get working on Amazon EKS as well. To use distributed functionality, build the client and server executables via:

```
git submodule update --init --recursive
mkdir build && cd build
# requires golang and libz
cmake -DBUILD_CLIENT_SERVER=1 ..
make -j8
```

This will produce three executables in build/src/distributed:

- "SCAMPserver": This is the SCAMP server. It accepts jobs via grpc and handles divying them up among worker clients.
- "SCAMPclient": Run this on worker nodes, it must be configured with the hostname and port where the SCAMPserver is. This is the workhorse of the computation, it will utilize all gpus or cpus on the host system to compute work handed to it by the server. Each worker node should have only one client executable running at a time. Though not completely necessary, these clients should have high bandwidth to the server for best performance.
- "SCAMP_distributed": This behaves similarly to the SCAMP executable above, except that it issues jobs to the server via rpc instead of computing them locally. use the –hostname_port="hostname:port" to configure the address of the server. Currently does not support any kind of authentication, so it will need to be run inside any firewalls that would block internet traffic to the compute cluster.

The server/clients can be set up to run under kubernetes pods using the Dockerfile in this repo. The docker image zpzim/scamp will contain the latest version of the code ready to deploy to kubernetes.

src/distributed/config contains a sample script which will create a GKE cluster using preemptible GPUs and autoscaling as well as sample configuration files for the scamp grpc service, client, and server pods. You should edit these scripts/configuration files to suit your application.

You can use this script to run and execute your own SCAMP workload on GKE as follows:

```
cd src/distributed/config && ./create_gke_cluster.sh
# Once cluster is up and running you can copy your desired input to the server
kubectl cp <local SCAMP input file> <SCAMP server container name>:/
# Now you can run SCAMP_distributed on the server and wait for the job to finish
kubectl exec <SCAMP server container name> -c server -- /SCAMP/build/src/distributed/
↪SCAMP_distributed <SCAMP arguments>
# Copy the results back to a local storage location
kubectl cp <SCAMP server container name>:/mp_columns_out .
```

Note: The configuration above runs SCAMP_distributed on the server, this is not required and is actually not the desired functionality. We would like to be able to run this remotely. While this is currently possible to do it is not reflected in our example.

The above example works on GKE but it should be simple to produce an example that works on Amazon EKS as well.

15.1 Limitations

- Server currently does not periodlically save state, so if it dies, all jobs are lost. This will eventually be fixed by adding sever checkpointing.
- Server currently handles all work in memory and does not write intermediate files to disk. For this reason the server requires a lot of memory to operate on a large input. Eventually the server will operate mostly on files on disk rather than keep all intermediate data in memory.
- All neighbors profiles and distance matrix summaries are not yet supported in distributed workloads.

15.2 Sharded implementation

The original distributed implementation used AWS batch and shards the time series to Amazon S3. This approach avoids the above limitations of our in-memory SCAMPserver, however our initial implementation was very limited in scope and was not extensible to other types of SCAMP workloads, so it is obsolete. The old scripts can be found *here <https://github.com/zpzim/SCAMP/tree/v2.1.0/aws>* for posterity. Though these would be strictly for inspiration, as there are AWS account side configurations required for operation that cannot be provided.

CHAPTER

SIXTEEN

FREQUENTLY ASKED QUESTIONS

This section contains solutions to common troubleshooting issues and questions about the SCAMP repository.

16.1 Missing libcufft.so on Linux

If when trying to run the SCAMP CLI or import pyscamp, you recieve an error message that looks something like:

```
libcufft.so:  cannot open shared object file:  No such file or directory
```

It is likely that CUDA has not been set up correctly on your system. You need to make sure your LD_LIBRARY_PATH is set correctly. You can put the following into your .bashrc or similar:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

16.2 CUDA/GPUs won't work

A common issue is that CUDA does not get picked up properly when building SCAMP or pyscamp. There are tips on how to deal with this in the documentation but some common tips are:

- Make sure your NVIDIA drivers are up to date.
- Make sure your *environment* is set up properly and wherever CUDA is installed is on your PATH. For example on linux usually this is: /usr/local/cuda/bin
- Set FORCE_CUDA=1 (See CLI and pyscamp docs for more information) to validate that CUDA is being detected during installation.
- See the *pyscamp* and *SCAMP CLI* documentation for more troubleshooting information.

16.3 SCAMP's output looks wrong

By default, SCAMP uses double precision to compute the matrix profile, for some datasets this may not be sufficient. SCAMP provides an ultra precision option which can help reduce floating point roundoff error. You can specify this option by specifying:

- SCAMP CLI: `--ultra_precision`
- pyscamp: `precision='ultra'` as a kwarg to any pyscamp function

Even this option will not be sufficient for some datasets, but it should work for the vast majority. For more information please see the documentation on *interpreting output*.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p
pyscamp, 31

INDEX

A

abjoin() (*in module pyscamp*), 32, 35
abjoin_knn() (*in module pyscamp*), 33, 35
abjoin_matrix() (*in module pyscamp*), 34, 35
abjoin_sum() (*in module pyscamp*), 33, 36

G

gpu_supported() (*in module pyscamp*), 36

M

module
    pyscamp, 31

P

pyscamp
    module, 31

S

selfjoin() (*in module pyscamp*), 31, 36
selfjoin_knn() (*in module pyscamp*), 33, 36
selfjoin_matrix() (*in module pyscamp*), 34, 37
selfjoin_sum() (*in module pyscamp*), 32, 37