Praktikum im SS22
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Vincent Bode, MSc., Minh Chung, MSc., Dennis-Florian Herr, MSc., Bengisu Elis, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Project 02 – Due: July 21st, 2022

1. **Background**

   In this project, we explore and evaluate the modern architectures for an state-of-the-art representative data mining application. For this, we choose time series mining applications, specifically the class of matrix profile analysis [1]. We provide you with the source code for the serial implementation of matrix profile computation and you will implement and study it in various target systems in BEAST exploiting the particular HW features of each platform, e.g., vectorization, threading and so on.

2. **High Performance Data Mining; Matrix Profile**

   Matrix profile is a similarity indexing data structure for time series: Matrix profile summarizes correlations and nearest neighbor information of local features in time series datasets.

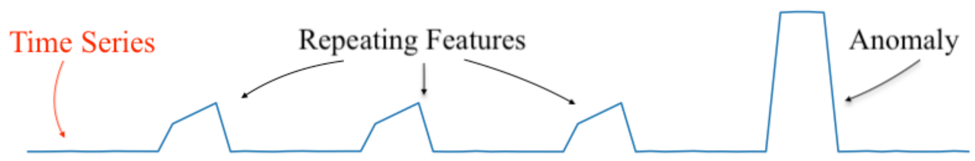

Figure 1: Illustration of a simple time series: matrix profile analysis is used to detect the repeating patterns (features) and anomalies in time series.

Formulation To better describe what a matrix profile is we provide a few simplified definitions to help you understand the underlying algorithm:

Definition 1 *A time series T is an ordered sequence of real-valued numbers: $T = (v_1, v_2, \ldots, v_n)$.*

Definition 2 *A subsequence T_i^m of a time series T is a continuous ordered subset of m values of T of starting from position i: $T_i^m = (v_i, v_{i+1}, \ldots, v_{i+m-1})$.*

Definition 3 *A matrix profile P of a time series T is a vector of distances of subsequences to their nearest neighbors. A matrix profile index I of a time series T is a vector of pointers to nearest (indices of) neighbor subsequences.*

In essence, matrix profile analysis tends to compute the distances among all the local features (see Figure 1) and find the most similar features by finding the ones with minimum distance.

We can further clarify these definitions by explaining how a trivial algorithm for computing P and I works: In a trivial approach, a full distance matrix among all subsequences is constructed. P and I are computed by finding the minimum, and argminumum values of rows of this distance matrix.

Algorithm To compute the matrix profile, various schemes and methods are proposed in the literature. The most efficient and accurate formulation for the computation of matrix profiles is developed by Zhu et al. [2].

In this formulation, the following four steps are included:

1. The *first* row of the distance matrix is computed using a trivial Euclidean distance formulation (see left-most illustration in Figure 2).
2. The following rows of the distance matrix are computed using a streaming dot product formulation [2]. With this formulation, the new rows are computed in-place (see middle-left illustration in Figure 2).
3. For each row of the matrix, row- and column-wise `min` and `argmin` operations are performed to construct *matrix profile* and *matrix profile index* (see middle-right illustration in Figure 2).
4. the `min` and `argmin` among all the elements of the row is computed and stored in two separate vectors (see right-most illustration in Figure 2).

Using this algorithm, in each iteration, only one row of the distance matrix is computed and the nearest neighbors information is computed on the fly and updated accordingly. As a result, unlike the computational cost which is $O(N^2)$ the memory consumption is $O(N)$.

Also note that due to the symmetrical structure of distance matrix (in case of self-join matrix profile [1]), we only compute the upper triangular part of the matrix.

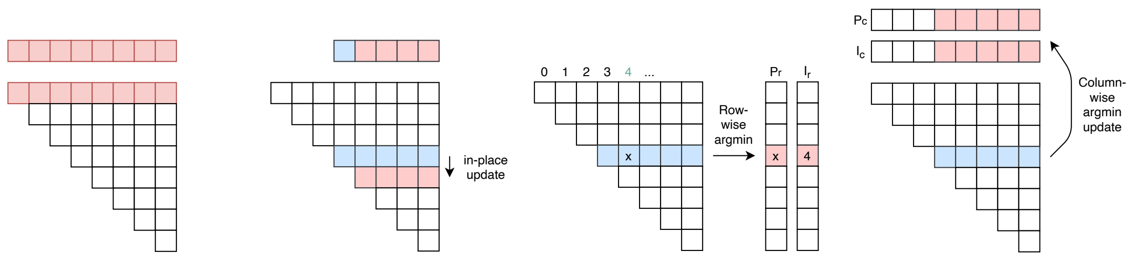

Figure 2: Illustration for the steps of computation. In the first step (left-most image), the first row of the distance matrix is computed (Step 1). Next figure shows the in-place update of the distance matrix (Step 2). Next figure illustrates the row-wise `argmin` computation (Step 3) and the right-most figure shows the column-wise `argmin` (Step 4). During the computation, the elements highlighted in red are active for computations or updates. The elements in blue are pre-computed values from last stage (last row) used for computation of red elements at this stage.

Implementation Listing 1 provides a vanilla implementation for the computation of matrix profile using the scheme that is described above. The main difference of the implementation in Listing 1 and the illustration in Figure 2 is that in the implementation we are using the same buffer to write the results of both row- and element-wise argmin operations (which is valid for the specific use case of self-join matrix profile [1]).

```
// loop over rows
for (int i=0; i<mlen-sublen; i++){
    // loop over columns
    for (int j=i+sublen; j<mlen; j++){
        if (i!=0) // except for the first row
            // streaming dot product - Pearson's correlation (first
               part)
            QT[j-i-sublen] +=  df[i] * dg[j] + df[j] * dg[i];

        // streaming dot product - scaling (second part)
        double cr = QT[j-i-sublen] * norm[i] * norm[j];

        // row-wise argmin
        if (cr > mp[i]){
            mp[i]=cr;
            mpi[i]=j;
        }

        // column-wise argmin
        if (cr > mp[j]){
            mp[j]=cr;
            mpi[j]=i;
        }
    }
}
```

Listing 1: matrix profile main kernel

3. Hints and Ideas for Optimization

(a) Parallelization: the outer loop in Listing 1 is a good candidate for coarse-grained parallelization, e.g., threading, while the inner loop is a candidate for SIMD. Analyze the loop-carried dependencies in the kernel provided in Listing 1. Discuss these dependencies and describe how you parallelize and optimize this kernel?

(b) Reduction: reduction is a type of operation that is commonly used in parallel programming to reduce multiple values—e.g., the local values in different threads—into a single result. In this exercise, you will need to use *User-Defined Reduction* operations in OpenMP—.e.g, to implement an `argmin` reduction—for correct parallelization and vectorization of the kernels.

(c) Blocking: creating smaller blocks of iteration space i, j as triangular or parallelogram tiles (see Figure 3) in the distance matrix calculation and running the blocks in parallel, similar to the tiling techniques you used in the previous assignments might be useful. Although in such case, additional initialization of the first row in each tile might be a punishing extra computation as it needs to be done using the non-optimized Euclidian distance kernel, but overall it might help to optimize the code. Moreover, You might

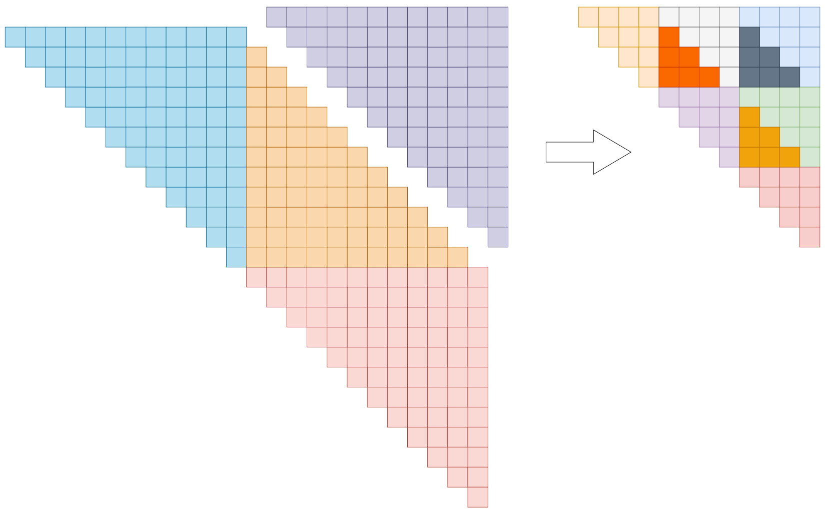

Figure 3: Triangular tiling of distance matrix computation. Left part: each tile is represented by a unique color and is executed by a different device. The final results need to be aggregated using User Defined Reduction operations. Right part: each tile is represented by a unique color and is executed in different compute units, e.g, by a thread. You might need similar reduction operations to aggregate the results in different threads.

need to implement a hierarchy of nested tiles for the distribution of workload to multiple devices, teams and threads.

4. **Tasks for CPU Architectures**
   (a) SIMD - User-Defined Reduction in OpenMP: In this task, we auto-vectorize the inner loop of the above kernel using the `simd` directive of OpenMP. While for `minimum` operations we can exploit the OpenMP `min` operation for reduction, this is not the case for `argmin` operation. Use the custom reductions in OpenMP and proper directives to vectorize the kernels.
      - Instrument the performance of the code before and after SIMD vectorization. Compare two.

   (b) SIMD - Intrinsic: In this task, we take one step forward and use the SIMD intrinsics that the processor offers for vectorization, e.g., AVX2 on Rome, AVX512 on Skylake, Neon on thx, and SVE on A64FX. You will need to use these ISA extensions to implement vectorized code for the innermost loop. This means that you would need to implement the distance computation, and reduction operations using the intrinsics. You can check the links in the footnotes[1] [2] [3], as well as the slides provided by ARM for information about SIMD in general.
      - *Bonus: Instrument the performance of the vectorized code using intrinsics.*
      - *Bonus: Compare the performance of the vectorized code usign intrinsics to auto-vectorized code.*

   (c) Multi-threading - OpenMP: In this task, we use OpenMP directives to parallelise the outer loop of the kernel. You need to use a tiling scheme similar to the ones that were introduced before, and work around the dependencies.

[1]SIMD Lecture from : https://www.moodle.tum.de/course/view.php?id=63406

[2]https://developer.arm.com/documentation/100891/0612/coding-considerations/using-sve-intrinsics-directly-in-your-c-code

[3]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

  - Use appropriate optimization techniques you learned in the previous lectures, e.g., thread pinning, NUMA configurations and
  - Study the effect of these techniques to explain their effects.
  - How does your implementation scale with number of threads?

(d) Profiling : Use appropriate profiling tools to get more insight about the performance of your implementations

  - Use appropriate profiling tools to acquire flop rates and memory throughput that your implementations can achieve. Compare the performance of this application to your vector triad and matrix multiplication microbenchmarks. (50 GFlops is a good performance for large time series, e.g., with 1M samples. However, you should observe very high memory bandwidth utilization)

5. **Tasks for GPU Architectures**

(a) OpenMP: use OpenMP offloading to run the kernel in Listing 1 on GPUs.

  - Apply the optimization techniques you used in previous assignments.
  - Find the suitable number of teams and threads similar to previous assignments.

(b) *Bonus: Other programming models and vendor development toolkits: Implement your code in another programming model of your choice like CUDA/HIP/OpenCL etc. For CUDA implementations you can find some vendor provided slides for quick start, pushed into your repository together with this pdf.*

  - *Optimize the reduction operations; you have the options to use a math library or vendor intrinsics to implement reduction operation.*
  - *Keep in mind memory access pattern optimizations.*

(c) Multi-GPU: Utilize multiple GPUs for both versions of your code.

(d) Profiling : Use appropriate profiling tools to get more insight about the performance of your implementations

  - Profile your code by using the tools and methods you learned in assignment #6 and identify the bottlenecks in your code.
  - Try to optimize the bottlenecks and measure the overall time of your code.
  - *Bonus: Measure and compare the overall performance of the two implementations (OpenMP vs. vendor development toolkits).*
  - Plot time series input length vs. Flop rates, and estimate the flop rates and memory throughput that your implementations can achieve (200 GFlops is a good performance for large time series, e.g., with 1M samples. however, you should observe very high memory bandwidth utilization).
  - Compare the performance of this application to your vector triad and matrix multiplication microbenchmarks.

References

[1] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, “Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets,” in 2016 IEEE 16th International Conference on Data Mining (ICDM), Dec 2016, pp. 1317–1322.

[2] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh, "Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins," Knowledge and Information Systems, vol. 54, no. 1, pp. 203–236, 2018.