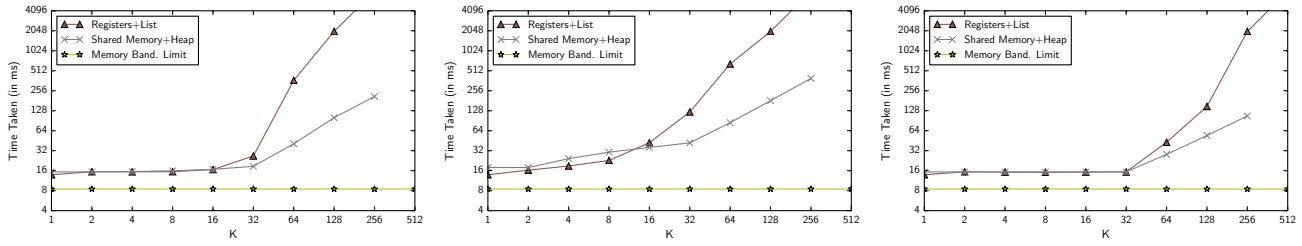

(a) Uniform (b) Increasing (c) Decreasing

Figure 18: Different Per-Thread Top-K Approaches

Algorithm 5: CPU Bitonic Top-K Thread

Input : Input Parititon S of length n; int k
Output : List O of the top-k elements per thread

```
1  int numElements ← n;
2  int numVectors ← numElements / vectorSize;
3  int temp[2][n/16];
4  int current ← 0;
5  for i ← 0; i < numVectors; i += 1 do
6      SortReducer(S, temp[current], i, k)
7  numElements ← numElements / 16;
8  while numElements >= vectorSize do
9      for i ← 0; i < numVectors; i += 1 do
10         BitonicReducer(temp[current], temp[1-current], i, k);
11     numElements ← numElements / 16;
12     numVectors ← numElements / vectorSize;
13     current ← 1 - current;
14 O ← sort(temp[current], numElements);
```

top-k elements. To make use of all the cores available, we partition the input array into equal sized partitions and let each core independently process the partition to emit the top-k. The top-k elements emitted by the individual core are combined in a final global step to find the global top-k.

On each core, we further break down the input partition into vectors of fixed size (in the implementation we use 2048 elements as the vector size). We process the input partition in phases. The first phase does the function of the SortReducer. It reads in the unsorted input paritition, one vector at a time and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. The subsequent phases do the function of BitonicReducer. They read in the input containing bitonic sequences of length k, one vector at a time, and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. Algorithm 5 shows the pseudocode.

On the GPU, each vector is processed in parallel by a thread block. Each thread of the thread block reads in 16 elements from shared memory and runs a combined step and outputs it back to shared memory. However, on the CPU, on each core, we process the vector in a single threaded fashion. The thread reads in 16 elements at a time from main memory and runs a combined step and outputs it back to shared memory. The reason we process small sized vectors (here of size 2048) is so that the data is cached in L1 cache. This allows random accesses in the vector to not incur latency of main memory read.

Modern CPUs also have support for Single Input Multiple Data(SIMD) instructions. The bitonic sorting network used to process a combined step can be implemented using SIMD instructions for improved performance. In the implementation we use 128-bit SSE-based implementation of [8]. Also, some of the optimzations details in Section 4.3 are not needed on the CPU. In particular, padding and chunk permutation are not useful on the CPU as there is no notion of bank conflict.

The bitonic top-k algorithm is not work-efficient. It does $O(n(logk)^2)$ number of comparisons as shown in Section 3.2. This is strictly worse than heap-based methods which do $O(nlogk)$ number of comparisons. However, bitonic top-k can leverage SIMD instructions to improve performance. Overall, in the case when lots of heap insertions occur (e.g.: when the input data is sorted) , the performance of bitonic top-k is close to that of heap-based methods despite the larger number of comparisons. Further, bitonic top-k could be better on platforms with wider vector instruction support like AVX-512 in Intel Knights Landing processors. We plan to explore this in the future.