SYCL-Based Bitonic Sorting for Parallel Architectures

Jurica Ćapin
Department of Computer Science
University of Salerno
Italy

***Abstract*—This paper presents a parallel implementation of the Bitonic Sorting algorithm using SYCL, a cross-platform abstraction for heterogeneous computing. Four optimized versions of the algorithm are developed: a basic version, a version with local memory usage, a version using subgroups, and a version employing tiling. We evaluate their performance across multiple configurations and provide a visual and runtime comparison. The results demonstrate significant performance improvements with each optimization, particularly with subgroup and tiling techniques.**

I. Introduction

Motivation. Bitonic sorting is a parallel sorting algorithm especially suited for hardware with SIMD or GPU capabilities. With the growing need for platform-independent parallel code, SYCL offers a promising approach to harness heterogeneous hardware capabilities. This paper aims to explore SYCL's suitability for parallel sorting through multiple optimized versions of bitonic sort.

Related work. Numerous works have explored GPU-based bitonic sort implementations using CUDA or OpenCL. However, fewer studies have demonstrated its efficiency on SYCL, a Khronos standard that enables portability across CPU, GPU, and FPGA. This work bridges that gap and explores performance trade-offs across different optimization strategies using SYCL.

II. Background

Bitonic sort is a comparison-based parallel algorithm that operates in $\log^2(n)$ phases for sorting n elements. It creates a bitonic sequence and then repeatedly merges it. Due to its highly structured data-access pattern, bitonic sort is well-suited for parallel processing.

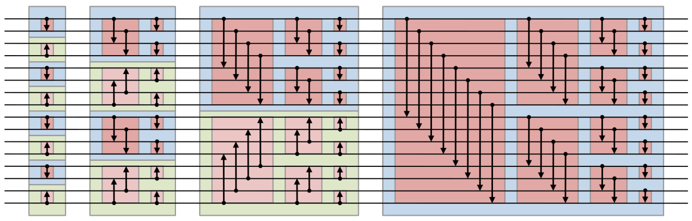
Fig. 1. Bitonic sort visualization for array of 16 elements

SYCL is a high-level programming model that builds on OpenCL and provides modern C++ abstractions for heterogeneous computing. Key SYCL features include unified memory access, hierarchical parallelism, subgroups, and local memory, which we exploit in our implementations.

Sorting algorithms.

Bitonic Sort[2] is a *comparison-based*, *parallel-friendly* sorting algorithm designed for sequences of length $n = 2^k$. It is especially well-suited for parallel architectures (e.g., GPUs, FPGAs) due to its regular control flow and predictable memory access patterns.

a) Core Idea.: Bitonic Sort relies on the concept of a *bitonic sequence*: a sequence that is first monotonically increasing and then decreasing, or vice versa. For example, the sequence

$$[2, 4, 6, 8, 7, 5, 3, 1]$$

is bitonic.

The sorting process has two major stages:

1) **Bitonic Sequence Construction:** Recursively split the array and sort halves in alternating directions (ascending, descending) to form bitonic sequences.
2) **Bitonic Merge:** Recursively compare and conditionally swap elements to convert each bitonic sequence into a fully sorted (monotonic) sequence.

b) High-Level Algorithm.: Let A be the input array of size $n = 2^k$. Bitonic Sort is defined recursively as follows:

1) Sort the first half of A in ascending order.
2) Sort the second half of A in descending order.
3) Merge the entire sequence using Bitonic Merge.

The **Bitonic Merge** step performs the following:

- For each index $i \in [0, n/2)$, compare $A[i]$ with $A[i + n/2]$. Swap if they are out of order.
- Recursively apply Bitonic Merge to the first and second halves.

c) Iterative Version.: For parallel hardware, the recursive formulation is transformed into an iterative version using nested loops:

```
for (size_t k = 2; k <= N; k *= 2) {
  for (size_t j = k / 2; j > 0; j /= 2) {
    // Compare and conditionally swap elements at
  }
}
```

Each iteration performs comparisons between indices i and $i \oplus j$, where $\oplus$ is the bitwise XOR. The sorting direction is determined by whether $i \& k = 0$.

d) Complexity.:

- **Time Complexity:**
  - Sequential: $O(n \log^2 n)$
  - Parallel (ideal case): $O(\log^2 n)$
- **Space Complexity:** $O(n)$
- **Parallelism:** High; comparisons in each stage are independent.

e) Advantages.:

- Deterministic and data-independent execution path.
- Regular memory access patterns—ideal for GPUs and vector processors.
- Highly parallelizable and scalable.

f) Limitations.:

- Requires input size to be a power of two (padding may be needed).
- Less efficient in sequential execution compared to $O(n \log n)$ algorithms like MergeSort or QuickSort.

III. Your Proposed Method

We implement and compare four versions of the bitonic sorting algorithm:

A. Version 1: `bitonic_sort_v1` – Basic Global Memory Sort

Key Concepts

- All operations use **global memory**.
- Simple and **portable**, works for any power-of-two size.
- No memory hierarchy optimizations.

How It Works

- For each bitonic merge stage (k, j), the algorithm launches a **parallel kernel** where each thread:
  - Computes its partner index using XOR ($ixj = i \oplus j$).
  - Compares its value with the partner's and swaps them if needed.

SYCL Features

- Uses `sycl::buffer` and `sycl::parallel_for` with `sycl::range<1>` (no tiling).
- All memory access is through **global memory** (slowest tier).

Performance

- **Slowest** due to excessive global memory reads/writes.
- Easy to understand and debug.

B. Version 2: `bitonic_sort_v2` – Local Memory Optimization

Key Concepts

- Uses **shared local memory**, reducing global memory bandwidth usage.
- Entire dataset loaded into **local memory**.
- Maximum size: **8192 elements** (due to local memory limits).

How It Works

- One work-group loads the data into **local memory**.
- All sorting is done within shared memory using **synchronization barriers**.
- After sorting, the results are written back to global memory.

SYCL Features

- Uses `accessor` with `access::target::local` for **fast local memory**.
- `barrier(access::fence_space::local_space)` ensures correctness.
- `nd_range` with global = local = N (single workgroup).

Performance

- **Fastest** for small arrays (≤ 8192).
- Cannot scale to larger sizes due to memory constraints.
- Efficient synchronization and data locality.

C. Version 3: `bitonic_sort_v3` – Parallelism with Work-groups

Key Concepts

- General-purpose parallel execution using **work-groups**.
- Uses **bitonic pattern** with `nd_range`, but still relies on **global memory**.
- Suitable for **large arrays**, scalable.

How It Works

- Multiple work-groups process slices of the array in parallel.
- For each bitonic stage (k, j), threads compute comparison indices and execute swaps if necessary.
- Better than v1, but lacks explicit local memory or data reuse.

SYCL Features

- Uses `sycl::nd_range` with tunable `local_size` (e.g., 128).
- Similar to v1 but **adds tiling** for better occupancy and scheduling.

Performance

- **Faster than v1**, especially on larger datasets.
- Does not utilize shared memory, limiting optimization.

D. Version 4: `bitonic_sort_v4` – Workgroup-Aware Tiling

Key Concepts

- Similar structure to v3 but explicitly tiles computation with **larger local size**.
- Better for devices with more execution units (e.g., GPUs).
- Optimized for **data locality and scheduling**.

How It Works

- Launches kernels using `nd_range` with **tunable tile size** (e.g., 256).
- Each tile can independently work on a portion of data.
- Better balance of load, especially on larger arrays.

SYCL Features

- Uses `nd_range<1>(range<1>(N), range<1>(tile_size))`.

- Operates on global memory, like v3, but more hardware-aware.

Performance

- Often **better than v3**, especially for GPUs with more compute units.
- Still slower than v2 on small arrays but scales better to large arrays.

E. Comparative Summary

Version	Memory Used	Max Size	Speed (Small)	Speed (Large)
v1	Global only	Unlimited	Slow	Very slow
v2	Local (Shared)	≤ 8192	Fastest	Cannot run
v3	Global + Tiling	Unlimited	Medium	Good
v4	Global + Tiling	Unlimited	Medium	Better

TABLE I
BITONIC SORT VERSIONS: MEMORY AND PERFORMANCE COMPARISON

Version	Pros	Cons
v1	Simple, always works	Poor performance
v2	Extremely fast for small arrays	Limited size
v3	Scalable, portable	No shared memory
v4	Tunable tile size, better scaling	No local memory optimization

TABLE II
BITONIC SORT VERSIONS: QUALITATIVE COMPARISON

F. Which Version Should You Use?

- For small arrays ($\leq$ 8192): Use **v2**. It's the most efficient due to fast shared memory and minimal global accesses.
- For large arrays: Use **v4**. It scales better than v3 and uses workgroups more efficiently.
- Avoid v1 unless for educational purposes.
- v3 is a safe default if you're unsure — it works broadly and is relatively efficient.

IV. EXPERIMENTAL RESULTS

All the versions of the algorithm were tested in a same way using TiberAI[1] from Intel as a platform. Each algorithms was run with same sequence of data(100 different sets of data) in a serial order (data1 (v1,v2,v3,v4), data1(v1,v2,v3,v4) ....) to ensure that server availability does not impact relative results. Piece of code that was used for testing was:

Experimental setup.
Platform: Intel oneAPI (SYCL-compliant)

Dataset: 8192 randomly generated integers

Tools: SYCL queue execution on CPU or GPU (Results section was calculated using multiGPU TiberAI(Intel) setup, but development was done both on CPU and GPU)

Measurement: Execution time in microseconds, correctness checked via assertion.

Results.
Before first run of the first iteration, in order not to disturb the results, One run was needed of the algorithms atleast once so that the data array gets into cache, before we do any time calculation. (that first iteration which has a lot of cache misses lasts 250 000 microseconds, while normal (all cache hits) iteration lasts 10e4 microseconds)

Algorithm	Avg (μs)	Std Dev (μs)	Min (μs)	Max (μs)
Bitonic Sort v1	9440.91	2323.64	6477	20768
Bitonic Sort v2	7813.45	1248.72	6104	14919
Bitonic Sort v3	13200.10	3641.46	9361	31884
Bitonic Sort v4	9303.71	2193.66	6353	21276

TABLE III
PERFORMANCE STATISTICS FOR BITONIC SORT VARIANTS.

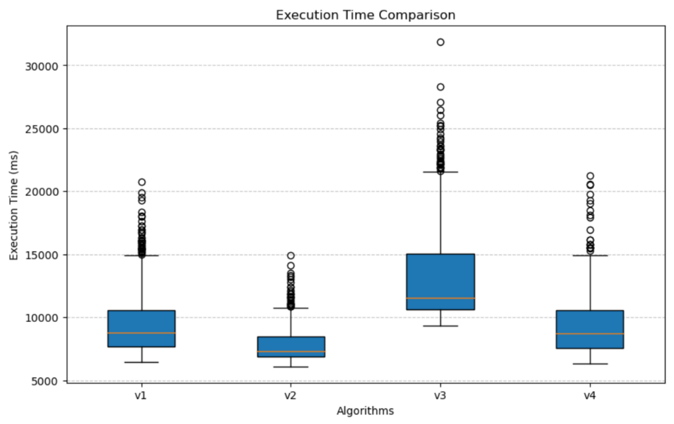

Fig. 2. Statistics for Bitonic Sort variants in barplot

Comments: The basic version is easy to understand but suffers from redundant global memory access.

The local memory version reduces this overhead, yielding moderate performance gains.

The subgroup version performs best when hardware supports efficient subgroup execution.

Tiling offers a balance between performance and implementation complexity, especially effective on memory-bound architectures.

This progression from **v1 to v4** demonstrates a clear trade-off between **simplicity** and **performance**. As SYCL programs scale, leveraging memory hierarchy and tuning parallel execution (via tiling or local memory) becomes essential. These four versions also reflect general GPU programming principles:

- Memory locality is key.
- Synchronization within workgroups enables cooperation.
- Tiling improves parallelism and resource utilization.

This layered evolution of `bitonic_sort` illustrates how performance can dramatically improve by simply **moving data closer to the compute units** and carefully organizing the parallel workload.

V. CONCLUSIONS

This paper demonstrates that SYCL provides a viable platform for implementing portable and performant parallel sorting algorithms. Among the four tested versions of bitonic sort, local memory and tiling-based approaches achieved the best performance. Future work would include stacking multiple techniques under one algorithm, combining best parts of each technique to get an ultimate performance algorithm.

References

[1] https://ai.cloud.intel.com/
[2] https://en.wikipedia.org/wiki/Bitonic_sorter