Hybrid Bitonic Sort on NVIDIA H100 GPU

Abhishek Revinipati

September 26, 2025

1 Introduction

Bitonic sort is a parallel sorting algorithm well-suited for GPUs because of its structured compare–exchange pattern. A naïve GPU implementation, however, suffers from heavy global memory traffic and synchronization overhead. This project implements a **hybrid bitonic sort** that combines shared memory kernels, global memory kernels, and warp shuffle intrinsics to balance performance and scalability.

2 Naïve Global Memory Implementation

Algorithm 1 Naïve Bitonic Sort (Global Memory Only)

Require: Input array $A[0 \dots n-1]$, direction up

for $s = 2$ to n step $\times 2$ **do**
  {Subsequence size}
  for $d = s/2$ down to 1 step $/2$ **do**
    {Partner distance}
    launch **GLOBAL_STAGE**(A, n, s, d, up)
  end for
end for

Algorithm 2 Kernel **GLOBAL_STAGE**

$tid \leftarrow threadIdx + blockIdx \times blockDim$
for $i = tid$; $i < n$; $i \leftarrow i + gridDim \times blockDim$ **do**
  $j \leftarrow i \oplus d$ {Partner index via XOR}
  if $j > i$ and $j < n$ **then**
    $dir \leftarrow ((i \,\&\, s) == 0)$
    if $(A[i] > A[j]) == (dir == up)$ **then**
      swap$(A[i], A[j])$
    end if
  end if
end for

Problems:

- All compare–exchange operations are performed directly in global memory, leading to high memory traffic and poor data locality.
- Each stage of the algorithm requires a separate kernel launch, resulting in a large number of launches.
- Data transfers between host and device are serialized, and the absence of pinned memory further increases h2d and d2h latency.

The performance metrics for the naive kernel is shown in Table 1.

3 Optimized Implementation

Our optimized implementation combines shared memory, warp shuffling, and global kernels to exploit the different abstractions of the CUDA programming model.

Metric	Value
H2D Transfer Time + kernel padding (ms)	150.4
D2H Transfer Time (ms)	92.3
Kernel Time (ms)	160.5
GPU sort speed (MEPS)	268
Achieved Occupancy (%)	66.64
Memory Throughput (% of peak)	15

Table 1: GPU Bitonic Sort Performance Metrics

Algorithm 3 Hybrid Bitonic Sort (CUDA Implementation)

Require: Input array $A[0 \dots n-1]$, direction up

$threadsPerBlock \leftarrow \min(1024, n)$
$blocksPerGrid \leftarrow \min(\lceil n/threadsPerBlock \rceil$, numSMs*4)
$paddedSize \leftarrow threadsPerBlock + threadsPerBlock/32$
{— Phase A: Local sort within blocks —}
launch **bitonic_sort_shared**(A, n, up)
Each block loads its chunk into shared memory with padding
Perform bitonic sort locally using:
- Shared memory for $d \geq 32$ (with __syncthreads)
- Warp shuffle for $d < 32$ (no sync needed)

Write results back to global memory
{— Phase B: Expand subsequences —}
for $s = threadsPerBlock \times 2$ to n; step $\times 2$ **do**
  for $d = s/2$ down to $threadsPerBlock$; step $/2$ **do**
    launch **bitonic_sort_global**(A, n, s, d, up)
    Each thread processes multiple pairs via grid-stride loop
  end for
  launch **bitonic_sort_shared_fused**$(A, n, s, threadsPerBlock/2, up)$
  Handles remaining stages with d less than 32 within shared memory
end for
return sorted array A

The shared-memory kernel handles compare distances that fit within the size of a single CUDA block (up to 1024 elements). Each block cooperatively loads its assigned subsequence into shared memory, with padding to reduce bank conflicts.

The kernel then performs the bitonic merge in stages:

- For distances where partner threads lie in different warps ($32 \leq d1024$), each thread reads its own value and its partner's value from shared memory. After applying the bitonic comparison rule, the updated result is written back. Synchronization (`__syncthreads()`) is used between steps to ensure correctness.

- For distances confined within a warp ($d < 32$), the kernel switches to warp shuffle (`__shfl_xor_sync`), allowing threads to exchange values directly through registers. This eliminates shared memory traffic and the need for barriers at these fine-grained stages.

Finally, each thread writes its sorted value back to global memory. This kernel efficiently performs all bitonic stages that fit within a block, combining shared memory for inter-warp comparisons and warp shuffle for intra-warp comparisons, thereby reducing global memory traffic and synchronization overhead.

4 CUDA Optimizations

Grid-Stride Loops with Occupancy-Aware Launch

The global kernel uses grid-stride loops so that each thread processes multiple elements. Instead of launching one thread per element, the grid size is limited to approximately 4× the number of streaming multiprocessors (SMs), with 1024 threads per block. This ensures high occupancy across the GPU while avoiding excessive kernel launch overhead. Each thread covers more work through the stride loop, making the kernel scalable to very large arrays.

Shared Memory Padding

Shared memory accesses can suffer from bank conflicts when consecutive threads in a warp map to the same memory bank. To mitigate this, each warp's data is offset by one element (`paddedIdx = tid + tid/32`). This padding spreads accesses across different banks and reduces replay events in the shared memory pipeline, improving throughput.

Pinned Memory

Host-to-device (H2D) and device-to-host (D2H) transfers are accelerated by using pinned (page-locked) memory instead of pageable memory. Pinned memory enables direct DMA transfers to the GPU, reducing transfer latency and increasing PCIe bandwidth utilization.

Asynchronous Transfers with `cudaMemcpyAsync`

Data transfers are further overlapped with kernel execution by using `cudaMemcpyAsync` in multiple CUDA streams. This enables concurrent H2D/D2H copies and kernel launches, hiding transfer latency behind computation. For large problem sizes, this overlap is critical to achieving high end-to-end throughput.

Technique	**Goal**	**Effectiveness**
Shared Memory Padding	Reduce bank conflicts	Replay counts decreased significantly
Warp Shuffle Intrinsics	Remove barriers inside warps	Very effective; reduced stalls and memory ops
Grid-Stride Loops	Scale to large arrays	Highly effective; coalesced and scalable
Hybrid Strategy	Balance shared and global kernels	Critical; achieved both speed and scalability
Occupancy Tuning	Keep SMs saturated	Effective; ensured high utilization

Metric	**Value**
H2D Transfer Time (ms)	21.1
D2H Transfer Time (ms)	7.2
Kernel Time (ms)	78.8
GPU Sort Speed (MEPS)	933.0
Achieved Occupancy (%)	91.74
Memory Throughput (% of peak)	52.23

Table 2: Optimized GPU Bitonic Performance Metrics

Compared to the naïve global-memory version, it significantly reduces memory traffic and transfer overhead, delivering over 150× speedup against CPU baselines.