CS 259 Lab 1

I. Bitonic Sort Implementation

A. Implementation

Our implementation of Bitonic sort algorithm on the GPU takes advantage of the independent nature of these comparison-and-swap operations, allowing them to be parallelized across multiple threads. Each thread is responsible for comparing and potentially swapping a pair of elements, speeding up the process.

The sorting of n elements happens in a series of log(n) phases or passes, with multiple comparisons and swaps occurring concurrently in each phase. This is controlled using two nested loops:

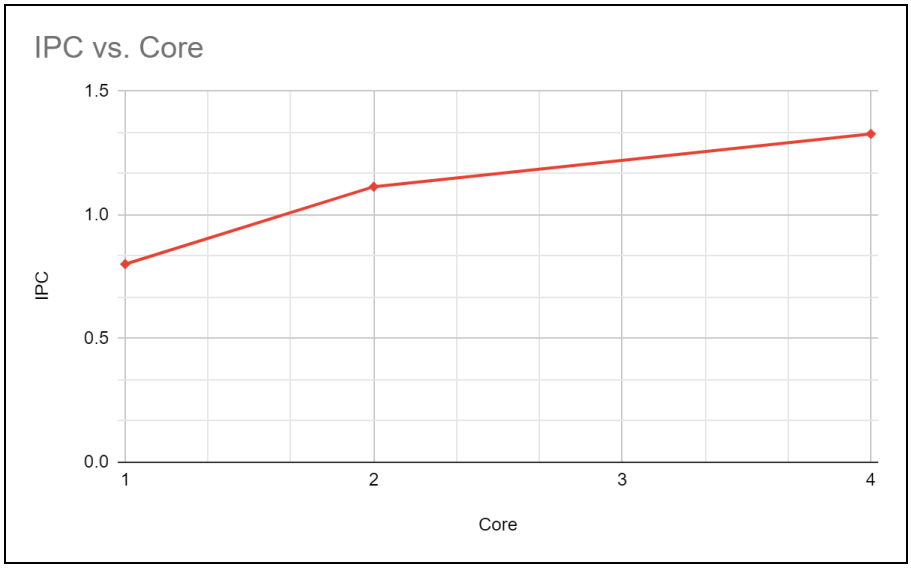
1. **Outer loop (k) for passes**:
   The value of k doubles with each pass, controlling the size of the bitonic subsequences that are compared and sorted. The outer loop iterates through larger subsequences, preparing them for sorting in parallel.
2. **Inner loop (j) for distances**:
   The inner loop controls the distance between the elements being compared. Starting with k/2, it progressively halves the distance with each iteration. As the value of j decreases, the comparisons become more fine-grained within the subsequences.

For each combination of k and j, vx_spawn_threads() is called to start threads to perform compare-and-swap operations for elements in parallel. After each thread completes its operations, a local barrier (vx_barrier()) is used to synchronize the threads within a warp. This maintains correctness and prevents data overwrites. To ensure memory consistency, vx_fence() is used after the swaps, to ensure that all memory operations are completed before progressing.

When executing across multiple cores, global synchronization is needed to ensure that no core gets ahead of the rest during a particular stage and avoid race conditions. This is ensured using a global barrier through vx_barrier() in the kernel_main. Before the sorting process begins, the entire array is copied from src_addr to dst_addr, and sorting is performed in-place in dst_addr. The copying is done on the GPU itself, with each thread responsible to copy one element.

B. Evaluation

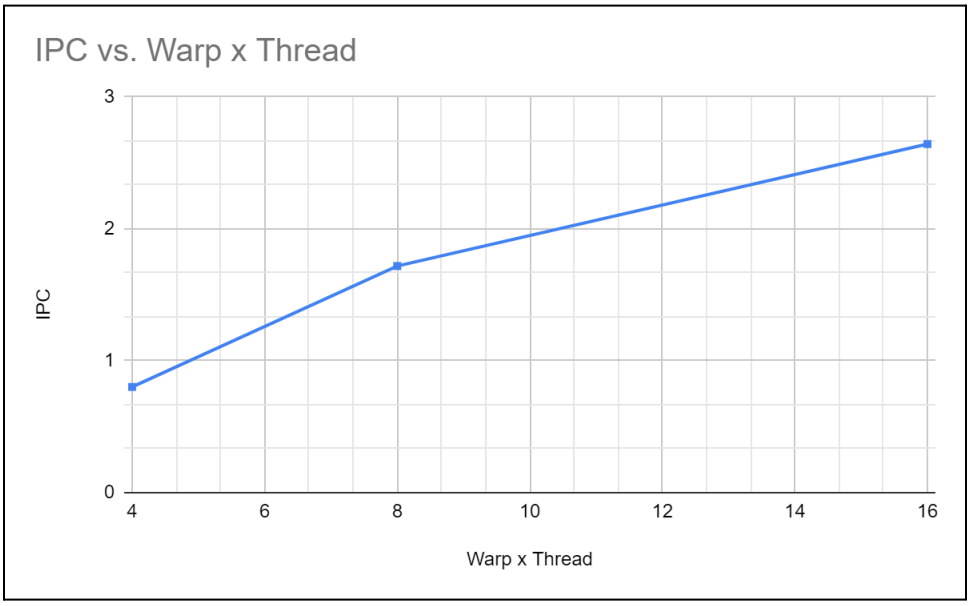
IPC vs. Core				IPC vs (Warps x Threads)			
Cores	**Instr**	**Cycles**	**IPC**	**W x T**	**Instr**	**Cycles**	**IPC**
1	17894292	22335662	0.801153	4x4	17894292	22335662	0.801153
2	17980320	16139543	1.114054	8x8	18124568	10548291	1.718247
4	18152376	13672964	1.327611	16x16	19042464	7204535	2.643122



Input array size = 4096, run on AMD Ryzen 7 Host

As the number of cores increases, the IPC also rises, indicating better parallel execution. However, the gains diminish as cores increase from 2 to 4, likely due to resource contention. The system becomes less efficient with each additional core, as the overhead of managing multiple cores begins to impact performance.

IPC vs. (Warp-x-Thread) for single core



Input array size = 4096, run on AMD Ryzen 7 Host

Increasing the number of warps and threads significantly boosts IPC, demonstrating efficient parallelism within a single core. The jump from 4x4 to 8x8 warps/threads shows a large gain, but then shows diminishing returns as system resources may become bottlenecked. This highlights effective scaling, but with some limits as the number of threads grows.

II. Performance Counter

A. Implementation

Our performance counter implementation uses 2 CSRs called ACTIVE_THREADS and SCHED_FIRES. Active_threads tracks how many threads in the thread mask are activated each time a warp is scheduled, and sched_fires tracks how many times a warp is scheduled. Calculating the ratio of active_threads / (sched_fires * threads_per_warp) results in the average warp efficiency.

The CSRs are defined in the Class 1 PERF MPM address space and included in the sched_perf struct inside the pipeline_perf_if interface. The scheduler in VX_schedule.sv updates both the active_threads and sched_fires counters whenever a new warp is scheduled on a core. Logic for the active_threads uses a POP_COUNT hardware module provided in vortex to sum the bits of the thread mask of any warp being scheduled and accumulates that number of active threads over time, meanwhile the sched_fires simply increments by 1 every time a new warp gets scheduled. Both counters are aggregated across cores in utils.cpp where the final warp efficiency is calculated.

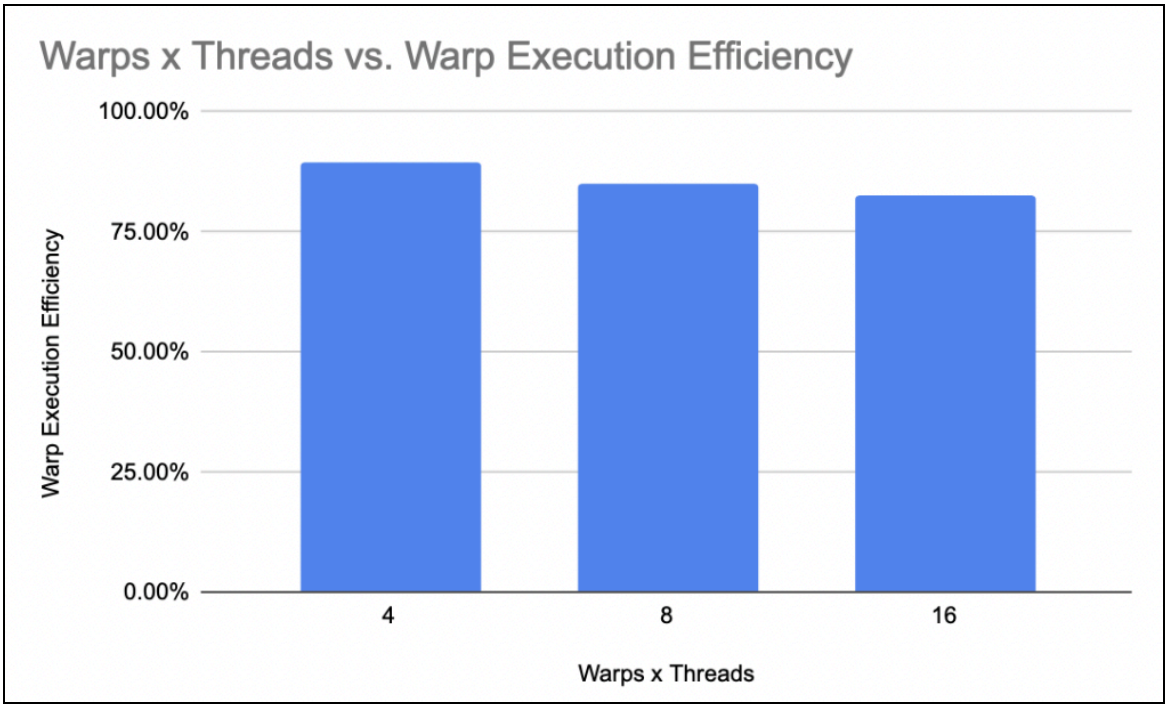
B. Evaluation

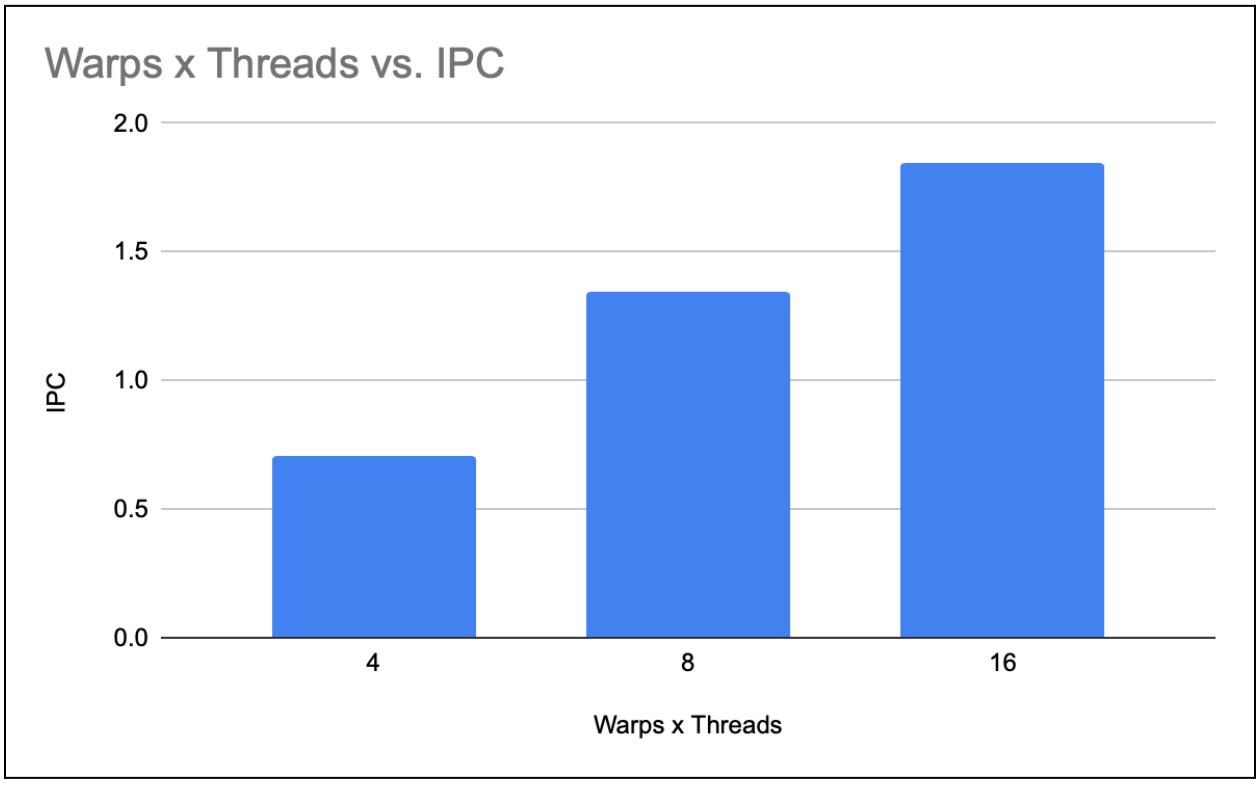
It appears that the number of threads that are activated are always marginally greater than the number of instructions executed, showing that some overhead exists to start up the GPU cycles before the kernel can execute.

Since this portion was tested on a Docker container on an M1 Mac, the emulator had trouble producing results for 4096 points, thus, the following graphs are compiled using data for 1024 points. Configuration => 1 core, 1024 points, issue width = 4.

Warps	Threads	IPC	Scheduler Fires	Activated Threads	Warp Efficiency
4	4	0.702639	821218	2944723	89.64%
8	8	1.34395	453756	3090587	85.14%
16	16	1.847703	277738	3671756	82.63%

The increasing IPC trend indicates that increasing warp and thread counts significantly speeds up the kernel at a cost of decreasing warp efficiency. The decreasing warp execution efficiency likely stems from each scheduled instruction not necessarily needing an entire warp's thread count to perform its intended function. From a tradeoff perspective, however, the 2.5x IPC boost for only a 7% decrease in warp execution efficiency provides justification for why GPU providers want to increase warp sizes to boost throughput and thereby revenue. Clearly with this type of parallelized hardware, brute forcing greater throughput generates more value than optimizing for perfect efficiency or maximum utilization.


Warps x Threads vs. Warp Execution Efficiency
100.00%
75.00%
50.00%
25.00%
0.00%
Warp Execution Efficiency
4
8
16
Warps x Threads


Warps x Threads vs. IPC
2.0
1.5
1.0
0.5
0.0
IPC
4
8
16
Warps x Threads