CS 8803 GPU Hardware & Software
Project #2

Objectives

The objective of this assignment is to further advance your understanding of parallel programming using CUDA. You will write CUDA code to perform parallel integer sorting and gain a deeper understanding of parallelism in GPU programming. This assignment enhances your CUDA programming skills and provides an exercise to get familiar with the NSight Compute profiler.

The goal of this assignment is to implement parallel sorting for arrays with the assistance of the NSight profiler, to achieve high performance on NVIDIA H100 GPU.

Instructions

Input to the program is a 1-D array. You have been provided with the initial framework for the code in `kernel.cu` on Canvas.

```
# Compile the code
nvcc -x cu kernel.cu
```

```
# Run an individual array (example: array size 10K)
./a.out 10000
```

Note: For debugging CUDA code, you will want to compile with both flags -g -G, then use cuda-gdb. This will allow you to use gdb within your kernel code.

All your implementation must be within the marked code region (`// ==== START OF YOUR CODE ==== ... // ==== END OF YOUR CODE ====).` Failure to do so may result in a penalty.

Background

The straightforward implementation of merge sort on a GPU can exhibit suboptimal runtime due to the nature of the algorithm. As each iteration reduces the active threads by half and the last iteration involves merging the entire array, it leads to inefficient parallelization. This reduction in active threads hinders the GPU's ability to fully exploit its parallel processing capabilities. As an exercise (not required for the assignment), you can write a CUDA program to perform a straightforward parallelization of the mergesort algorithm using <<< N, M >>> kernels. What does the 'Achieved Occupancy' look like for kernel launches in the later iterations on NSight?

Divide and conquer, an effective paradigm for parallel algorithms, involves breaking a problem into smaller subproblems solved recursively, enabling concurrent processing. Mergesort, an optimal sequential sorting algorithm utilizing divide and conquer, serves as inspiration for parallel sorting algorithms like Bitonic sort. **Bitonic sort** efficiently maintains parallelism, making it well-suited for GPU architectures. Another notable approach is Batcher's odd-even merge sort, leveraging a sorting network for effective parallelism in sorting operations.

Bitonic Sort Explained

Bitonic Sort creates a bitonic sequence, which is a sequence that starts as ascending and then becomes descending (or vice versa). The algorithm recursively sorts subsequences of the bitonic sequence until the entire sequence is sorted. It works on sequences with lengths that are powers of 2.

Bitonic split: of a bitonic sequence $L = x_0, x_1, x_2 .. x_{n-1}$ is defined as decomposition of L into

$$L_{min} = \min(x_0, x_{n/2}), \min(x_1, x_{(n/2)+1}) \ . \ . \ . \ \min(x_{(n/2)-1}, x_{n-1})$$
$$L_{max} = \max(x_0, x_{n/2}), \max(x_1, x_{(n/2)+1}) \ . \ . \ . \ \max(x_{(n/2)-1}, x_{n-1})$$

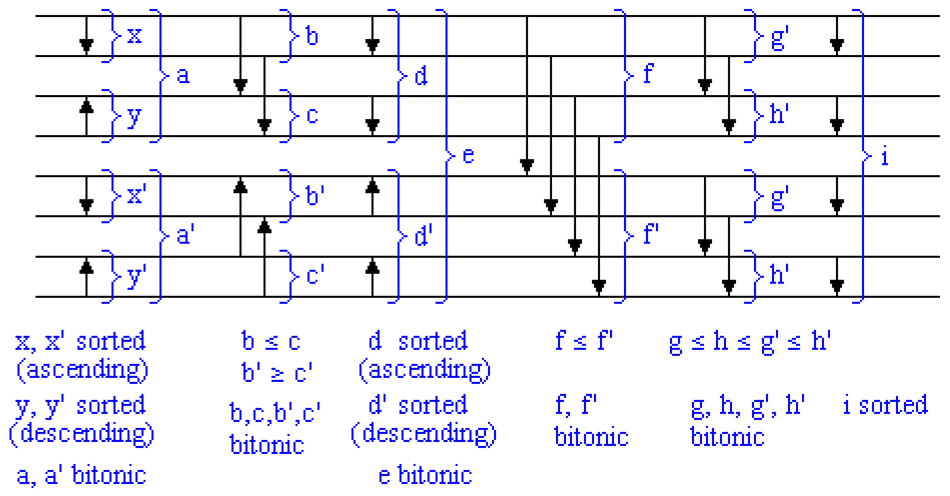
L_{min} and L_{max} are also bitonic sequences with $\max(L_{min}) \leq \min(L_{max})$

Bitonic merge: Turning a bitonic sequence into a sorted sequence using repeated bitonic split operations. Successive bitonic split operations are applied on the decomposed subsequences until their size reduces to 2, culminating in a fully sorted sequence.

$$BM(n) = BS(n) + BS(n/2) + . . BS(2) = O(\log n)$$

To achieve a bitonic sequence, we start with a sequence of length 2 and apply Bitonic merge to obtain bitonic sequences of length 2. The process involves alternating between ascending and descending order to transform an unsorted array into a bitonic sequence. This serves as the initial step before applying further Bitonic merge operations to eventually achieve a fully sorted sequence.

$$BitonicSort(n) = BM(2) + BM(4) + . . BM(n) = O(\log^2 n)$$



Task #1

You need to implement bitonic sort in CUDA. Pseudo code for bitonic sort algorithm to sort n elements:

```
for i=1 to (log n) do
     for j=i-1 down to 0 do
          for k=0 to n do #loop through the array
            a = arr[k]; b = arr[k XOR $2^j$];
            if (k XOR $2^j$) > k then # (a,b) are compared so skip (b,a) case
                if ($2^i$ & k) is 0 then
                     Compare_Exchange↑ with (a, b)
                else
                     Compare_Exchange↓ with (a, b)

          endfor
     endfor
endfor
```

The outer loop sequentially traverses the stages of the Bitonic sort algorithm. Each iteration corresponds to the execution of the BM(2^i) operation, resulting in the generation of sorted sequences with lengths of 2^i. The inner loop performs multiple bitonic splits essential for completing a bitonic merge operation.

The **compare exchange** swaps elements. The sorting strategy ((2^i & k) = 0 check) arranges even chunks (sub-sequences of length 2^i) in ascending order and odd chunks in descending order, forming a 2^{i+1} bitonic sequence for the subsequent ith iteration.

```
     Compare_Exchange↑: if (arr[i] > arr[j])
                            arr[i], arr[j] = arr[j], arr[i]
```

The **XOR operation** determines the indices of the two elements to be compared during the jth iteration. For instance, if $i = 2$ and $j = 2$ (first sub-stage of BM(8)), XORing 0 with 4 yields 4, 1 gives 5, and so forth. To gain a better understanding, it is encouraged to experiment with various values of i and j and compare your results with the image in the previous page. In simpler terms, XOR strides the rank by 2^j, providing the indices required for comparing elements during the algorithm's iterations.

Task #2

CUDA Optimizations

We move on to optimizing our parallel program using our learnings from lectures and NSight profiler observations to improve memory and compute efficiency. The method described in Task#1 leads to excessive kernel launches and prolonged global memory access times. NSight may indicate:

"This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device."

Shared memory to rescue: If a subsequence of size 2^s fits into the shared memory of a block, process steps `s,s-1 . . .1` of the bitonic sort on the shared memory to reduce global accesses and also kernel launches. Break down your kernel into two: one with the shared memory and another with the global memory.

We will evaluate a couple of times and take the best run to avoid any server load issues. We will run your code on the H100 in PACE. Be sure to explicitly select the H100 when creating an instance.

Please try to optimize the programs to make # of million elements per second (meps) as high as possible. You will get full points when achieving 900 million elements per second.

To encourage you to get familiar with the NVIDIA profiling toolkit, we also evaluate the 'Memory Throughput' and 'Achieved Occupancy'. If you have multiple kernels in your implementation, we will take the average of all the kernels. You are expected to get Memory Throughput higher than 80% and 'Achieved Occupancy' higher than 70% on an H100 GPU.

You can use NVIDIA ncu to get these metric:

```
# Memory Throughput
ncu -- metric
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed
--print-summary per-gpu a.out 10000000

# Achieved Occupancy
ncu --metric sm__warps_active.avg.pct_of_peak_sustained_active --
print-summary per-gpu a.out 10000000
```

What to submit:

You will upload two files to Canvas. Do not create a zip file containing these two files. The files must have the following names or point deductions will be applied.

- report.pdf – The report must be a .pdf file in order to receive credit.
- kernel.cu - Your bitonic sort implementation.

Grading Policy

Total points: 20 plus up to 2 extra points based on meeting performance metrics

The program will be graded on the correctness and usage of the required parallel programming features. You should also use a good programming style and add comments to your program so that it is understandable.

The script used to grade is provided (`grade.py`) to evaluate the score locally. **All submissions are evaluated using H100 GPUs in the PACE-ICE cluster. Be sure to explicitly select the H100 when creating an instance.**

Start the assignment early to circumvent a last-minute rush to secure a node on the pace-ice cluster.

Grading consists of the following components:

1. Functional correctness **(5 pts)**

   We will check whether the sorting results match the expected results for the input array of sizes 2K, 10K, 100K, 1M, 10M.

   Each test case of array size will give 1 pts. "FUNCTIONAL SUCCESS" is printed on the terminal for passing cases.

   If your code is not parallel code, you will get only 20% of functional correctness (i.e., 1 pt)

2. Performance **(16 pts)**
   We will go through your code to make sure the appropriate parallel programming practices discussed in the class are being followed along with the right CUDA functions being called. The evaluation metrics will be run for **100M** array size. Please make sure your implementation is robust at this scale.

   Note: off-loading compute operations (e.g. final 'merge') to CPU is not allowed.
   Note: You can only get performance points when your implementation is functionally correct.

Evaluation metric	Max Credit	Calculation
Achieved Occupancy	1	Achieved Occupancy >= 70%
Memory Throughput	1	Memory Throughput >= 80%
Performance Option 1		

Million elements per second (meps)	14	if meps > 900
		min((meps/1000)*14, 14)

Performance Option 2		
Kernel time (ms)	10	min((80/kernelTime)*10, 10)
Memory Transfer Time D2H + H2D	4	min((30/memTime)*4, 4)

Note: Performance Options (14 points)

- Option 1 is million elements processed per second. If your meps score is 900 or over, then you are eligible for this score. Full credit is obtained with a meps score of 1000.
- Option 2 is a combination of time spent in the GPU kernel doing calculations where 80ms or less is ideal and full credit of 10 points is given and time spent transporting data to and from the GPU where 30ms or less is ideal and full credit of 4 points is given.
- You will receive the higher score of the two options.

Note: Points will be deducted for ignoring performance protocols; **serialization in the program will lead to a zero on the whole assignment.**

3. Report **(1 pt)**

   When describing the project, what you did, what you tested, etc., you can assume that the reader is familiar with the topic. Also, be sure to refer to the syllabus for expectations on Assignment Quality.

   Report contents:

   - Your report will discuss project implementation and performance counter analysis
   - Discussion of performance optimization techniques implemented and effectiveness of each
   - Report length must be between two and three pages (including charts and graphs.) You will use 11 pt. Times New roman or Arial fonts. These rules are in place so that you write enough (but no too much) in your report.
   - **You must submit a report to receive a grade. Submissions without a report will earn zero points for this project.**

NSight Compute

Profilers are tools that sample and measure performance characteristics of an executable across its runtime. This information is intended to aid program optimization and performance engineering. Nsight Compute - Provides an in depth level assessment of individual GPU kernel performance and how various GPU resources are utilized across many different metrics. Use NSight Compute NvProf to report the following numbers as well:

- Number of global memory accesses
- Number of local memory accesses
- Number of divergent branches
- Achieved Occupancy

This YouTube video can serve as a good starting point to using the profiler visualizations to optimize code: `https://developer.nvidia.com/nsight-compute`

Running NSight Compute:

```
    ncu ./<program name> # stats for each kernel on stdout
    ncu -o profile ./<program name> # output file for NSight Compute GUI
```

Command to list all existing metrics:

```
    ncu --query-metrics --query-metrics-mode all
```

To check the metrics you can use this command

```
    ncu --metrics [metric_1],[metric_2],... ./<program name>
```

You can read ncu documentation to understand what each metric means. Here are some metrics that you probably need:

```
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum
l1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum

smsp__thread_inst_executed_per_inst_executed.ratio
smsp__thread_inst_executed_pred_on_per_inst_executed.ratio

sm__maximum_warps_per_active_cycle_pct
sm__warps_active.avg.pct_of_peak_sustained_active
```

Additional Resources

- Bitonic Sort
- Prof. Vuduc's bitonic sort lecture: items 23 (Comparator networks) through 28
- Batcher's Odd-Even Merge Sort
- Improved GPU Sorting
- PACE ICE cluster guide
- NVIDIA CUDA Toolkit Documentation
- CUDA Programming Guide

Version: Aug 31 2025