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 L40S 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.

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

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

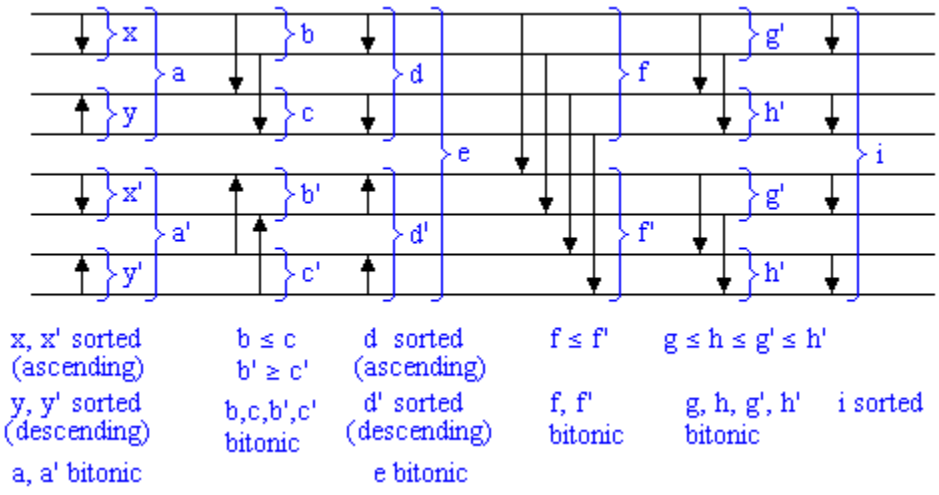
L_{min} and L_{max} are also bitonic sequences with $max(L_{min}) \leq min(L_{max})$

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 i^{th} 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 j^{th} 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.

For evaluation, we will run your solution with 10M (10,000,000) array sizes. 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 L40S in PACE. Be sure to explicitly select the L40S 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 650 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 L40S 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:

report.pdf and kernel.cu file. Please submit a pdf file format for the report.

Grading Policy

Total points: 20 (bonus 3 pts)

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 the submissions are evaluated on the L40S in the pace-ice cluster. Be sure to explicitly select the L40S 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., 2 pts)

2. Performance **(14 pts + Bonus 3 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 10M array size.
   Receiving bonus points are subject to the contents of the report.

   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%
Million elements per second (meps)	15	if meps <400: 0 else: min(15, (meps/650)*15)

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)**
    - Implementation and performance counter analysis (max 3 pages including figures)
    - Performance optimization techniques and discussions

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

FAQs

1. **Can I implement an algorithm that is based on a paper or should I come up with something new?**
   You can implement algorithms based on others. Please cite appropriately.

2. **In CUDA SDK, sorting algorithms are already provided. Can I use it?**

   While you are encouraged to explore alternative implementations, a substantial part of your code should be independently crafted. In cases where your code borrows or references external sources, it is essential to provide proper citations and clearly indicate the portions that are derived from other source code.

3. **Can I start the code from CUDA SDK?**

   Yes, especially the code on 0_simple can be a good starting point.

4. **Isn't the merge sort performance based on the input?**

   The parallel algorithms described in Task#2 are data-independent.

5. **If my code runs faster on my local machine, leveraging a newer GPU architecture, what performance metrics should I present to showcase the improvements?**

   While the optimization is commendable, grading will be contingent on performance results from the pace-ice cluster as described in the Grading section. Ensure to validate the performance gains on the pace cluster.

6. **My code works for array sizes of N but fails for > N ?**

   There could be several reasons for this failure. As a sanity check, it is useful to check for any memory leaks.

7. **How to check for memory leaks in my implementation ?**

   You can consider using the `compute-sanitizer` provided by the CUDA toolkit

   ```
   nvcc -Xcompiler -rdynamic -lineinfo -x cu kernel.cu
   compute-sanitizer a.out 10000
   ```

8. **Bitonic sort algorithm expects the array size to be a power of 2 while the instructions specify multiples of 10?**

   To resolve this issue, modify the array size to a power of 2. After GPU sorting is complete, ensure a proper adjustment to the array to maintain the final sorted array's size equivalent to the input array,

9. **Why is the bitonic merge failing on the sorted arrays computed from the shared memory block in Task#2?**

   Ensure that the sequences involved in the merge operation exhibit a 'bitonic' nature. It is crucial to prevent the inadvertent merging of non-bitonic sequences that may arise after the shared memory bitonic sort.

10. **nvcc results in 'command not found' error on pace-ice cluster?**

    Load the module: `module load cuda gcc/12.3.0`
    You can also put this command in your bash file (~/.bashrc) to avoid typing this for every new terminal session.

11. **"This site can't be reached" for** https://ondemand-ice.pace.gatech.edu/ **?**

    Make sure you are connected to Georgia Tech VPN.

12. **Are there resources to learn about Bitonic Sort?**

    You may find the following resources helpful

    - For a nice visual explanation: https://www.youtube.com/watch?v=uEfieI0MumY

- Wikipedia - https://en.wikipedia.org/wiki/Bitonic_sorter

13. Possible performance instability due to Cluster GPU

The cluster GPU resource is shared among other users, so when multiple students are using the GPUs (especially when the deadline approaches), the performance numbers show differences from run to run. When the assignment is graded, we will choose a time when the GPU is idle.

To make sure you are on the right track, it is also okay to share your numbers (speedup, throughput, occupancy) with classmates to get a sense of whether your results are mostly aligned with other students. Please use the Project 2 - Performance thread to post your values.

14. Analysis based on NSight compute is somewhat expected in the report. Are there any resources available that shows how to use the tool?

Resources for using NSight compiler are provided in the document, attaching here: Nsight Compute | NVIDIA Developer | NVIDIA Developer

These videos provide a basic walkthrough of using the tool and also examples of using it to improve performance.

15. Is the bitonic sort algorithm supposed to handle any array size?

Bitonic sort as an algorithm works on arrays with lengths that are a power of 2 (2, 4, 8, …, 1024, 2048, etc). However, for project 2, we expect your entire implementation to work on input arrays of any size. As such, you will need to pad the input array until its length is the next power of 2 before running bitonic sort. For example, if the input array given is of length 1000, you should pad it until it has 1024 elements, and then run bitonic sort.

16. Is there a required citation format that we need to use?

Feel free to use any citation format.

17. I noticed this line in the project description: "Break down your kernel into two: one with the shared memory and another with the global memory". Does this mean we'd submit our code with both kernels, but only call one (I assume the shared memory kernel)?

The project description suggests (not enforced) breaking down the kernel into two phases: one utilizing shared memory to sort as many integers as possible that fit within the shared memory constraints, followed by gathering these individually sorted arrays and sorting them using a global kernel. Both kernels must be invoked to obtain a fully sorted array.

18. Compiler Options and Performance

When compiling your code to test for performance, make sure you are not using the -g and -G flags. These debug flags will slow down your code.

19. What is the report format?

The report should be a standard report with the format of your choosing. Content could follow the basic format of a research paper. Things we will be looking for include:

- Brief intro and summary of the problem and how you approached it
- Any important details in your implementation that we should know about.
- Base performance data that highlights things you discovered. Could include GPU to CPU comparisons, GPU performance comparisons for different data sizes, etc.
- Optimizations that you made (good ones and bad ones), their effects and your analysis

- Observations you made from the project
- Citations for anything other than course assigned materials
- Possible next steps.

  You have three pages with figures/tables so it will be a challenge to figure out what not to include and yet still tell your story. Make the assumption that the reader is familiar with the topic (a fellow student or TA, for instance) and document what you did and how it turned out. By default, we will grant nearly full credit for the report. However, the report will be carefully graded in two specific cases: when determining eligibility for bonus points or when deciding whether to deduct a significant portion of points from project #2, which could impact your final grade.

  Don't stress over the report too much. The idea is to have you show us how you achieved what you did with the final program and what you learned along the way.

Version: Aug 28 2024