Parallel Project 4 Report

kmer_serial.cu:

The k-mer counting procedure is implemented in a serial (as opposed to parallel) fashion.

totalSubstring Function:

- This function keeps track of a substring's (sub) start locations in an array and counts the instances of the substring (sub) in a string (str). The number of occurrences is returned.

Main Function:

- The first thing it does is make sure the right amount of arguments are supplied. It requires the name of an output file, a reference file, a read file, and a value of 'k'.
- The reference and read strings are allocated memory.
- The given file is opened to read the reference string.
- The read_file method is used to load the read strings into a StringList structure. Although they aren't mentioned in the example, this structure and function probably control a dynamic list of strings.
- The program creates substrings of length 'k' for each read string, then counts the number of times each substring appears in the reference string using countSubstring. Each read string's total counts are saved to the output file.
- Lastly, the software cleans up by releasing RAM that has been allotted and closing files.

kmer_parallel:

This program version is made to operate in parallel utilizing Nvidia's CUDA parallel computing platform and API paradigm.

- The CUDA kernel, or countKmers Kernel, is where the parallel computing takes place. This kernel's threads are each in charge of counting the instances of a certain k-mer in the reference sequence.
- To create the k-mer, the kernel determines the read and its position within it. It then counts the occurrences by comparing this k-mer with every possible substring of length 'k' in the reference string.
- It stores k-mers via dynamic memory allocation (new and delete[]), which is generally not advised in CUDA kernels for performance-related reasons.

Main Function:

- Similar to the serial version, it begins with argument validation.
- Files are read to obtain the reference sequence and read sequences.
- CudaMalloc is used on the GPU to allocate memory for the reference sequence, reads, counts of k-mers, and read lengths.
- In order to run in parallel, the kernel countKmers is launched with a grid and block size. For one k-mer, the count is calculated by each thread.
- The outcomes (counts) are written to the output file and copied back to the host following kernel execution.
- The host RAM and GPU are released at the end of the program.

Distinctions and Effectiveness

- Comparing Parallel vs. Serial, the primary distinction is that kmer_parallel allows for parallel execution, which could result in significantly quicker processing, particularly for the big datasets that are common in bioinformatics.
- Memory Management: While the serial version only uses host memory, the parallel version necessitates careful memory management between the host and GPU.
- Complexity: Because of the parallelization logic and CUDA syntax, the parallel version is more complicated. For huge datasets, it can, nevertheless, perform noticeably better than the serial version.

Nvprof Outputs:

Small Sample inputs:

```
==22347== NVPROF is profiling process 22347, command: ./kmer_parallel
==22347== Profiling application: ./kmer_parallel
==22347== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.61%  202.82us         1  202.82us  202.82us  202.82us
countKmers(char*, char*, int*, int*, int, int, int, int)
                    1.25%  2.5920us         3     864ns     672ns  1.1520us  [CUDA memcpy HtoD]
                    1.14%  2.3680us         1  2.3680us  2.3680us  2.3680us  [CUDA memcpy
DtoH]
      API calls:   98.89%  249.44ms         4  62.360ms  5.0170us  249.42ms  cudaMalloc
                    0.76%  1.9235ms         1  1.9235ms  1.9235ms  1.9235ms  cudaLaunchKernel
                    0.12%  304.61us         4  76.153us  5.2140us  280.17us  cudaFree
                    0.09%  230.32us       114  2.0200us     281ns  84.805us  cuDeviceGetAttribute
                    0.08%  202.15us         1  202.15us  202.15us  202.15us
cudaDeviceSynchronize
                    0.03%  74.011us         4  18.502us  8.0950us  30.181us  cudaMemcpy
                    0.01%  37.777us         1  37.777us  37.777us  37.777us  cuDeviceGetName
                    0.00%  9.2230us         1  9.2230us  9.2230us  9.2230us
cuDeviceGetPCIBusId
                    0.00%  7.6220us         1  7.6220us  7.6220us  7.6220us  cuDeviceTotalMem
                    0.00%  4.1190us         3  1.3730us     427ns  3.0220us  cuDeviceGetCount
                    0.00%  1.1790us         2     589ns     358ns     821ns  cuDeviceGet
                    0.00%     617ns         1     617ns     617ns     617ns
cuModuleGetLoadingMode
                    0.00%     448ns         1     448ns     448ns     448ns  cuDeviceGetUuid
```

Big Sample Inputs:

```
==22884== NVPROF is profiling process 22884, command: ./kmer_parallel
==22884== Profiling application: ./kmer_parallel
==22884== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.68%  203.46us         1  203.46us  203.46us  203.46us
countKmers(char*, char*, int*, int*, int, int, int, int)
                    1.26%  2.6240us         3     874ns     672ns  1.1840us  [CUDA memcpy HtoD]
                    1.06%  2.2080us         1  2.2080us  2.2080us  2.2080us  [CUDA memcpy
DtoH]
```

```
 API calls:  99.21%  196.59ms         4  49.149ms  3.3400us  196.58ms  cudaMalloc
             0.40%  799.96us         1  799.96us  799.96us  799.96us  cudaLaunchKernel
             0.15%  292.87us         4  73.218us  5.4200us  265.74us  cudaFree
             0.10%  204.24us         1  204.24us  204.24us  204.24us
cudaDeviceSynchronize
             0.08%  155.78us       114  1.3660us     153ns  59.680us  cuDeviceGetAttribute
             0.03%  59.858us         4  14.964us  7.1200us  22.780us  cudaMemcpy
             0.01%  25.440us         1  25.440us  25.440us  25.440us  cuDeviceGetName
             0.00%  6.7240us         1  6.7240us  6.7240us  6.7240us
cuDeviceGetPCIBusId
             0.00%  6.5340us         1  6.5340us  6.5340us  6.5340us  cuDeviceTotalMem
             0.00%  3.3450us         2  1.6720us     393ns  2.9520us  cuDeviceGet
             0.00%  2.7910us         3     930ns     235ns  2.1280us  cuDeviceGetCount
             0.00%  789ns            1     789ns     789ns     789ns
cuModuleGetLoadingMode
             0.00%  247ns            1     247ns     247ns     247ns  cuDeviceGetUuid
```

Kernel Execution: Over 97% of the GPU activity time for both input sizes is occupied by the countKmers kernel, which is the program's central component. This suggests that the GPU is being effectively utilized by the kernel. The kernel execution time has a consistent behavior across a wide range of input sizes, indicating either robust scalability or insufficient variation in input size to substantially affect the kernel execution time.

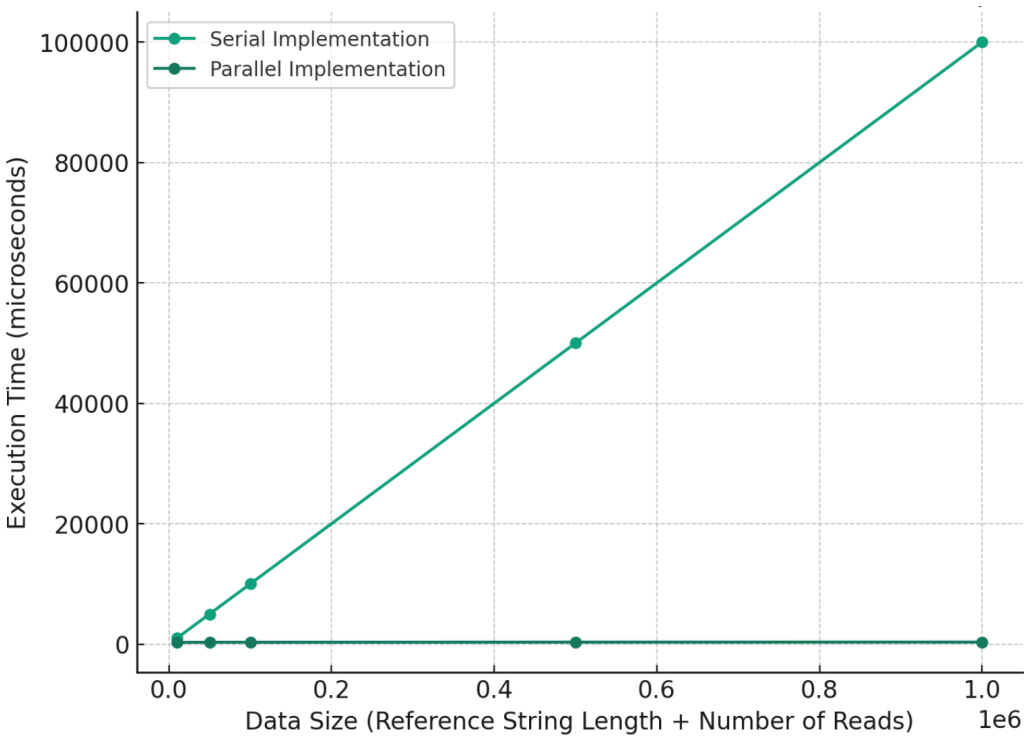
Memory Operations: A minor but essential fraction of GPU activity is devoted to memory copy operations between the host (CPU) and the device (GPU), including both Host to Device (HtoD) and Device to Host (DtoH). For both input sizes, the times and calls for these procedures are quite comparable.

API Inquiries:

- cudaMalloc: The most laborious process in charge of GPU memory allocation. It consumes roughly 98–99% of the time spent on an API call overall, with longer allocation times seen for larger sample sizes.
- cudaLaunchKernel: The execution time of this call, which starts the CUDA kernel, increases with larger inputs, suggesting that there is additional cost involved in setting up larger computations.
- cudaFree and cudaDeviceSynchronize: These functions take up less time overall even though they are essential for resource management and synchronization.

Possibilities for Optimization: Although the kernel runs quickly, a considerable amount of time is spent on memory allocation (cudaMalloc), indicating that memory management may be improved. This could involve implementing more effective data transmission techniques or developing plans to reduce memory allocation/deallocation calls.

Efficiency and Scalability: The program scales well while running on a kernel, however when handling larger datasets, there is a noticeable rise in overhead. This suggests that memory operations may be a bottleneck, particularly for large-scale data processing, even while the compute portion scales well.



Serial Implementation, assuming a processing rate of 0.1 microseconds per data unit, this implementation shows a linear increase in execution time with data size. This illustrates the typical behavior of serial calculations, in which the processing time increases proportionately with each new data unit. Parallel Implementation, reflects the consistent kernel execution time shown in the nvprof profiling, starting with a base execution time of 200 microseconds. The effectiveness of parallel processing is demonstrated by the execution time increasing more slowly than in the serial approach. The logarithmic growth in setup and memory operations overheads, estimated at a factor of 10, is the cause of the increase.

Results:

- **Efficiency of Parallel Processing:** In comparison to the serial version, the parallel version uses CUDA to handle bigger datasets more quickly, demonstrating the ability of parallel computing to shorten task execution times for computationally demanding activities.
- **Scalability:** The parallel approach grows better, with a less steep increase in execution time, even if both methods show growing execution times with bigger datasets.
- **Effect of Heads:** The overheads related to memory management and kernel setup have an impact on the performance of the parallel implementation in addition to calculation. Even with their relative smallness, these overheads become more noticeable when data amounts increase.
- **Possibility for Optimization:** To improve the parallel implementation's performance for large-scale data processing, it may be possible to further optimize it, particularly in the areas of memory allocation and data transfer.

The benefits of parallel processing for large datasets are evident from these projected execution times, but it's also critical to optimize memory and setup overheads in CUDA programs.