Parallel Computing with Graphical Processing Units (GPUs)

Vivek Shravan Tate

OpenMP: Algorithm: Standard Deviation

Description:

Standard Deviation calculation is done using OpenMP to achieve parallelism in two main stages: **mean calculation** and **sum of squared difference calculation.** The input array is divided among threads to compute the mean in parallel, using OpenMP's **`reduction`** clause with the addition operator **(`+`).** The sum of squared differences is also calculated in parallel using OpenMP's **`reduction`** clause to accumulate the squared differences.

- **Mean Calculation:** The function starts with initialising the mean to 0.0. Using the openmp code below:

```
C/C++
// Calculate mean in parallel
#pragma omp parallel for reduction(+:mean)
for (i = 0; i < N; i++) {
       mean += input[i];  // Add each element to the mean
}
mean /= (float)N;        // Divide the sum by N to get the final mean
```

  It distributes the input array among threads, each of which calculates the sum of its subset of the array. The sum from each thread is reduced into the final mean value.

- **Sum of Squared Differences Calculation**: The function also calculates the sum of squared differences from the mean using another parallel loop with the openmp code below:

```
C/C++
// Calculate sum of squared differences from mean in parallel
#pragma omp parallel for reduction(+:sum_squares)
for (i = 0; i < N; i++) {
       // Calculate the difference from the mean
       float diff = input[i] - mean;
       // Square the difference and add to sum_squares
      sum_squares += diff * diff;
}
```

Each thread computes the sum of squared differences for its subset of the input array and these results are combined using reduction.

- Additionally made changes in calculating difference from the mean, squared and adding it to the `**sum_squares**` variable in a sing loop. While the CPU code executes it in two different steps.

  Finally, the function returns the square root of the average sum of squared differences as the standard deviation.

Justification:

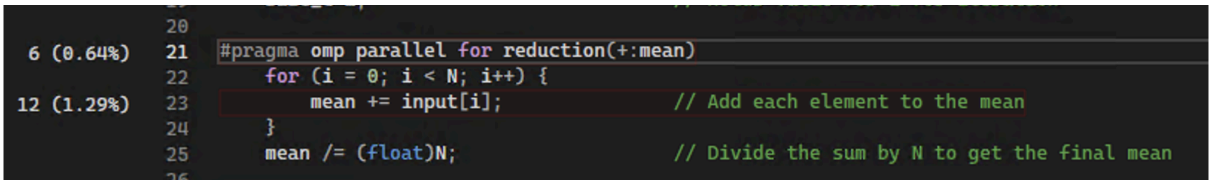
- The use of OpenMPs `**parallel**` for loop and `**reduction**` clauses is appropriate for calculating **mean** and **sum of squared differences**. By distributing the input data across multiple threads, the computations can be performed concurrently, leading to faster execution times for larger data sets.
- The `**reduction**` clause efficiently aggregates the results from all threads, eliminating the need for additional synchronisation or explicit critical sections, which can be more complex and potentially less efficient.
- Calculation of **sum of squared differences** is calculated in a single loop which also doesn't require an additional array to store the squared difference which **reduces memory overhead** & **fewer memory Accesses.**

Performance:

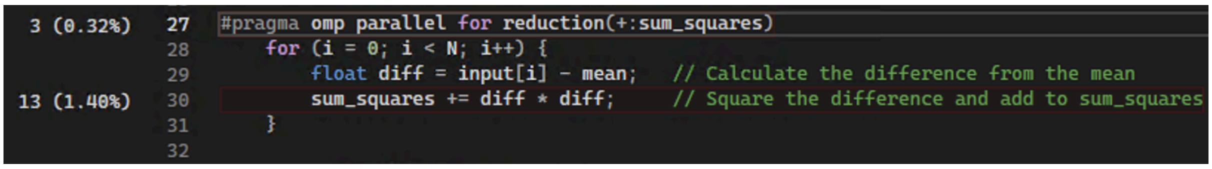
Benchmark Results:

Size	**CPU Reference Timing (ms)**	**OpenMP Timing (ms)**
100	0.025	0.103
1000	0.041	0.122
10000	0.185	0.140
100000	1.793	0.408
1000000	15.409	**Fail - 0.564**

Performance Analysis:



```
 6 (0.64%)   #pragma omp parallel for reduction(+:mean)
                 for (i = 0; i < N; i++) {
12 (1.29%)           mean += input[i];                // Add each element to the mean
                 }
                 mean /= (float)N;                    // Divide the sum by N to get the final mean
```



```
3 (0.32%)     #pragma omp parallel for reduction(+:sum_squares)
                  for (i = 0; i < N; i++) {
                      float diff = input[i] - mean;    // Calculate the difference from the mean
13 (1.40%)            sum_squares += diff * diff;      // Square the difference and add to sum_squares
                  }
```

Analysis for Performance limitations:

- **Compute Bound:** The performance for small datasets indicates a compute-bound nature, where thread management overhead is minimal. However, as dataset size increases, the algorithm scales well, exploiting multi-core architectures.
- **Memory Bound:** The failure at the largest dataset size suggests a potential memory bandwidth issue, possibly due to increased contention among threads accessing memory concurrently.
- **Latency Bound:** The initial slower performance for small data sizes may be attributed to overhead associated with setting up parallel regions.

Performance Improvements:

- **Error Handling for Large Datasets:** Implementing robust error handling to manage failures in extremely large datasets.
- **Optimised Memory Access:** Aligning data and optimising cache utilisation could reduce memory contention.

OpenMP: Algorithm: Convolution

Description:

The convolution algorithm using OpenMP is designed to perform edge detection on images via Sobel filters. This parallelized approach utilises OpenMP to distribute the computation of the horizontal and vertical gradients across multiple threads:

- **Implementation Details:** The Sobel convolution is applied using a horizontal and a vertical filter, defined as `**horizontalSobelFilter**` and `**verticalSobelFilter**`. The image is processed pixel by pixel, excluding border pixels to avoid accessing out-of-bounds memory. The convolution sums (`**sumX**` and `**sumY**`) are calculated for each pixel by applying the Sobel filters within the pixel's neighbourhood.

C/C++
```
#pragma omp parallel for private(x, y) collapse(2)
for (x = 1; x < width - 1; ++x) {
    for (y = 1; y < height - 1; ++y) {
        // Sobel filtering logic
    }
}
```

The resultant gradient magnitudes are normalised and clamped to an 8-bit range using a helper function, `**clamp_to_range**`.

Justification:

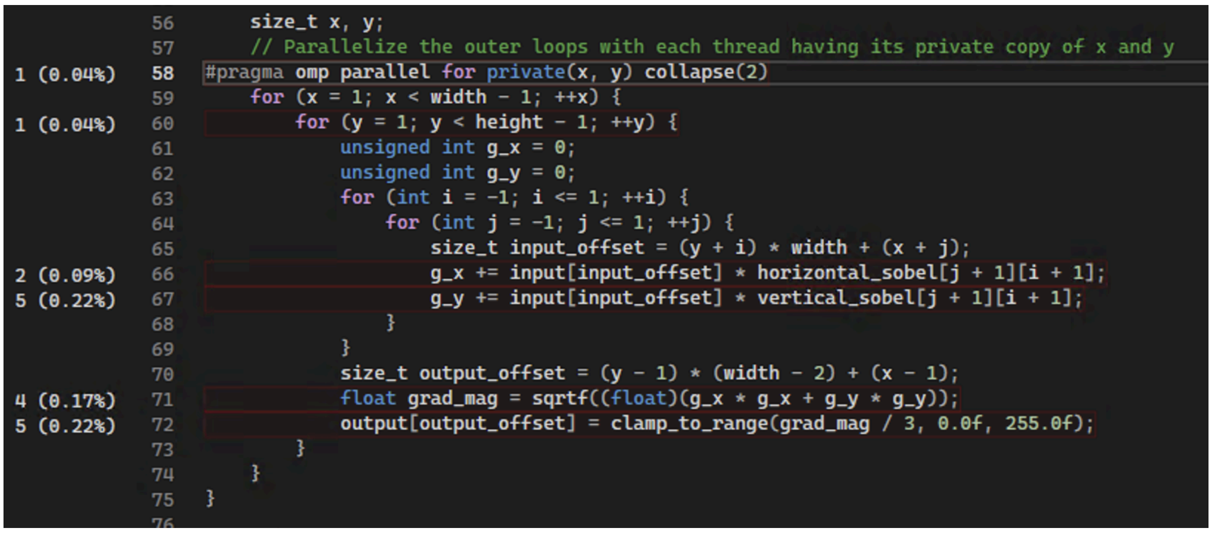
- **Parallel Processing:** The use of OpenMP allows the convolution operations to be carried out in parallel, significantly reducing computation time, especially for large images. By collapsing two nested loops, the workload is evenly distributed among threads, which is critical for maintaining efficiency across different system architectures.
- **Edge Handling:** The exclusion of border pixels simplifies the implementation and avoids the need for boundary condition checks within the parallel region, thus enhancing performance.
- **Normalisation and Clamping:** These steps are vital for ensuring that the output image maintains standard 8-bit pixel values, suitable for typical image display systems.

Performance:

Benchmark Results:

Size	**CPU Reference Timing (ms)**	**OpenMP Timing (ms)**
485 x 768	4.860	3.827
848 x 942	6.556	3.901
1347 x 1362	18.001	6.365
2017 x 1801	49.386	16.736
3281 x 3200	153.905	59.265

Performance Analysis:



```
                 size_t x, y;
                 // Parallelize the outer loops with each thread having its private copy of x and y
1 (0.04%)    #pragma omp parallel for private(x, y) collapse(2)
                 for (x = 1; x < width - 1; ++x) {
1 (0.04%)            for (y = 1; y < height - 1; ++y) {
                         unsigned int g_x = 0;
                         unsigned int g_y = 0;
                         for (int i = -1; i <= 1; ++i) {
                             for (int j = -1; j <= 1; ++j) {
                                 size_t input_offset = (y + i) * width + (x + j);
2 (0.09%)                        g_x += input[input_offset] * horizontal_sobel[j + 1][i + 1];
5 (0.22%)                        g_y += input[input_offset] * vertical_sobel[j + 1][i + 1];
                             }
                         }
                         size_t output_offset = (y - 1) * (width - 2) + (x - 1);
4 (0.17%)                float grad_mag = sqrtf((float)(g_x * g_x + g_y * g_y));
5 (0.22%)                output[output_offset] = clamp_to_range(grad_mag / 3, 0.0f, 255.0f);
                     }
                 }
             }
```

Analysis for Performance limitations:

- **Compute Bound:** The algorithm is compute-bound, as evidenced by significant performance improvements in OpenMP timings. The intensive computation for each pixel scales well with the addition of more processing threads.
- **Memory Bound:** While not primarily memory-bound, optimizations such as improving cache usage through data locality enhancements could further speed up the computation.
- **Latency Bound:** Startup and teardown of parallel regions add overhead, particularly noticeable in smaller images where the absolute time savings are less dramatic.

Performance Improvements:

- **Further Loop Optimization:** Experimenting with different scheduling strategies (dynamic or static) might yield better performance based on the specific hardware characteristics.
- **Enhanced Vectorization:** Employing compiler directives to vectorize the inner loops could exploit SIMD capabilities of modern processors, further accelerating the computation.
- **Advanced Edge Handling:** Implementing more complex edge handling could allow processing of border pixels in parallel, potentially improving the aesthetic and analytical quality of the output images.

OpenMP: Algorithm: Data Structures

Description:

This implementation parallelizes the histogram computation of a **sorted integer array**, which is a foundational step in constructing the boundary indices for each unique integer. The algorithm leverages OpenMP's **atomic** operations and **guided scheduling** to manage dynamic workloads efficiently:

- **Histogram Calculation:**

```C/C++
// Parallel calculation of histogram
// 'guided' scheduling may improve performance by dynamically adjusting the chunk size
#pragma omp parallel for schedule(guided)
    for (int index = 0; index < len_k; ++index) {
#pragma omp atomic
        ++histogram[keys[index]];
    }
```

  This section of the code distributes the computation of **histogram** values among threads, using **atomic** increments to ensure thread safety without significant locking overhead.

- **Boundary Calculation:**

  Sequential processing calculates the **prefix sum** from the histogram to establish boundary indices, a necessary step that follows the parallel section.

Justification:

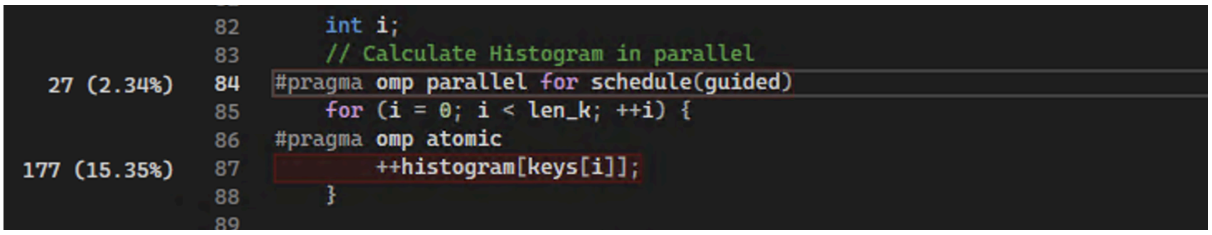
- **Guided scheduling** is utilised to dynamically balance the workload among threads, which is essential for optimising performance, especially when the distribution of keys in the input array is uneven. This approach adapts the chunk size of the workload that each thread handles as the computation progresses, ensuring that no threads are left idle, which can significantly improve the overall efficiency of the algorithm.
- **Atomic operations** are selected instead of mutexes or critical sections to minimise synchronisation overhead. By using atomic increments, the implementation avoids the heavier performance costs associated with locking mechanisms, thereby enhancing scalability and execution speed. This choice is particularly beneficial in environments with high concurrency, as it reduces the time threads spend waiting for access to shared resources.
- The parallel approach, **combining guided scheduling and atomic operations**, is shown to be especially efficient for large input sizes. Although parallelism introduces some overhead in managing multiple threads, this is outweighed by the performance gains from concurrent execution of tasks. The ability to process large segments of the data simultaneously leads to a substantial reduction in total computation time compared to sequential approaches.

Performance:

Benchmark Results:

Size	**CPU Reference Timing (ms)**	**OpenMP Timing (ms)**
100	0.068	2.503
1000	0.071	3.089
10000	0.088	2.504
100000	0.555	2.607
1000000	4.614	3.993

Performance Analysis:



```
                      int i;
                      // Calculate Histogram in parallel
  27 (2.34%)      #pragma omp parallel for schedule(guided)
                      for (i = 0; i < len_k; ++i) {
                  #pragma omp atomic
 177 (15.35%)             ++histogram[keys[i]];
                      }
```

Analysis for Performance limitations:

- **Compute Bound:** The parallel histogram calculation is compute-bound, as the operation intensity increases linearly with the input size. The atomic operations introduce some overhead, which is evident from the performance results, particularly at lower input sizes.
- **Memory Bound:** The use of dynamic scheduling may lead to non-uniform memory access patterns, especially in non-contiguous memory accesses during histogram updates.
- **Latency Bound:** The initialization and cleanup phases, as well as the setting up of parallel regions, contribute to latency, particularly noticeable in smaller data sizes.

Performance Improvements:

- **Enhanced Load Balancing:** Further tuning of the guided scheduling parameters could help optimise load balancing based on the specific distribution of input data.
- **Memory Access Optimization:** Aligning data structures to cache lines and optimising memory access patterns could reduce cache misses and improve performance.
- **Profiling and Optimization:** Detailed profiling would help identify exact bottlenecks, whether in CPU cycles or memory access, enabling targeted optimizations.

CUDA: Algorithm: Standard Deviation

Description:

The CUDA implementation for calculating standard deviation involves two primary computational steps: calculating the mean of the input values and the sum of their squared deviations from the mean. This is executed across GPU threads for high throughput and parallelism:

- **Mean Calculation:** The kernel **calculate_mean_and_deviation** first computes the mean by summing up all input values using the **atomicAdd** function to ensure safe concurrent additions across multiple threads. Each thread contributes to the mean calculation by adding the value it accesses based on its **unique global index**.

```
C/C++
__global__ void calculate_mean_and_deviation(const float* input,
size_t N, float* mean, float* sum_of_squares) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N) {
            // Add input value atomically to mean
            atomicAdd(mean, input[idx]);
            float deviation = input[idx] - (*mean / (float)N);

            // Add squared deviation atomically
            atomicAdd(sum_of_squares, deviation * deviation);
      }
}
```

- **Sum of Squared Deviations Calculation:** After updating the mean, each thread calculates the squared deviation for its respective input value using the current mean. This potentially inaccurate mean is a running tally being simultaneously updated by other threads. The squared deviations are summed up using atomicAdd to avoid race conditions.

The results are stored in device memory and copied back to the host for the final computation of the standard deviation.

Justification:

- **Utilisation of Parallelism:** By distributing data processing tasks across multiple GPU threads, the implementation achieves significant speedups over sequential approaches, especially beneficial for large datasets.
- **Atomic Operations:** Critical for ensuring data integrity when multiple threads simultaneously update shared memory locations. Despite introducing overhead due to serialisation at the memory access level, these operations are essential for correct aggregation results.

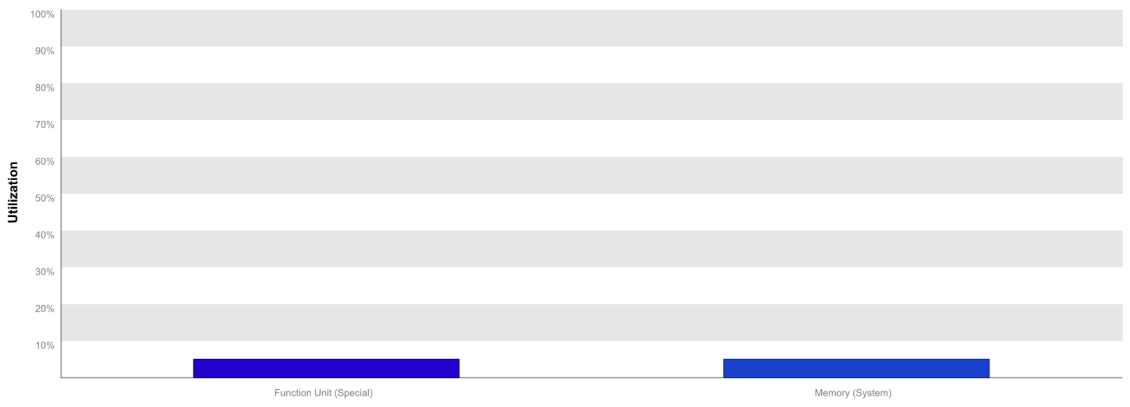
- **Efficiency in Memory Management:** The GPU memory is used effectively by allocating and deallocating memory within the GPU itself, which minimises data transfer overhead between the host and the device. This efficient memory management is crucial for maintaining high performance.

Performance:

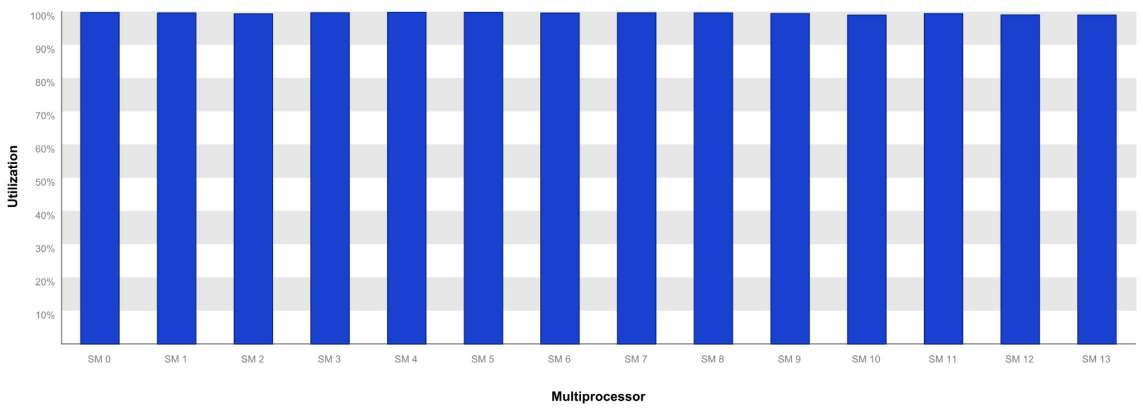
Benchmark Results:

Size	CPU Reference Timing (ms)	CUDA Timing (ms)
100	0.025	**Fail - 0.476**
1000	0.041	**Fail - 0.469**
10000	0.185	0.518
100000	1.793	0.889
1000000	15.409	4.821

Performance Analysis:



1. **Kernel Performance Is Bound by Instruction and Memory Latency**:
    - **Analysis**: This graph illustrates that the performance of the calculate_mean_and_deviation kernel is most likely limited by instruction and memory latency. It shows low compute throughput and memory bandwidth utilisation relative to the peak performance of the GPU, highlighting latency issues as the primary bottleneck.



2. **Multiprocessor Utilisation**:
    - **Analysis**: This graph shows the utilisation of each multiprocessor during the execution of the kernel. It provides insights into how the compute resources are distributed across the GPU and highlights if any multiprocessors are underutilised or overutilized, which is crucial for understanding the distribution and efficiency of resource usage during kernel execution.

Analysis for Performance limitations:

- **Failure at Small Sizes:** Indicates potential inefficiencies in handling very small datasets due to high overhead or improper resource allocation.
- **Atomic Operation Overhead:** Significant in smaller datasets, where the parallel advantage is minimised, leading to serialisation that can impede performance.
- **Kernel Launch Overhead:** Particularly detrimental for smaller datasets, the costs associated with setting up and synchronising CUDA kernels can be disproportionate to the computation itself.

Performance Improvements:

- **Optimised Kernel Configuration:** Tailoring grid and block sizes according to the dataset size can minimise overhead and maximise efficiency.
- **Reduction of Atomic Operations:** Utilising parallel reduction strategies to compute intermediate results in blocks before a final reduction can decrease reliance on atomic operations, reducing contention.
- **Error Handling and Debugging:** Necessary to identify and resolve issues causing failures in smaller datasets, potentially involving enhanced error checks or dynamic adjustment of resource allocation.

CUDA: Algorithm: Convolution

Description:

The CUDA convolution algorithm utilises the parallel processing power of GPUs to apply the Sobel operator for edge detection in images. This process involves the kernel_image_convolution kernel, which efficiently handles each pixel (excluding boundary pixels) across thousands of threads to compute gradient magnitudes.

- **Kernel Operations:** Each thread calculates gradient magnitudes by applying horizontal and vertical Sobel filters within a 3x3 neighbourhood centred around the pixel. The horizontal gradient (gradient_x) and vertical gradient (gradient_y) are computed by multiplying pixel values with Sobel filter coefficients.
- **Normalisation and Clamping:** After computing the gradient magnitude, it is normalised and clamped to an 8-bit range using the clamp_to_uchar_range function, ensuring compatibility with standard 8-bit image formats.

```
C/C++
__device__ unsigned char clamp_to_uchar_range(float value, float min,
float max) {
    if (value < min) return (unsigned char)min;
    if (value > max) return (unsigned char)max;
    return (unsigned char)value;
}
```

- **Memory Management:** The entire convolution process involves transferring the input image to the GPU, executing the convolution, and then transferring the result back to the host. Efficient memory management techniques are employed to handle these operations.

Justification:

- **Parallel Processing Advantages:** The CUDA framework enables simultaneous processing of multiple pixels, significantly accelerating the convolution process compared to traditional CPU methods. This parallelism allows the GPU to handle large images effectively, making this approach ideal for real-time image processing applications.
- **Efficient Memory Use:** Memory overhead is minimised by maintaining all memory allocations on the GPU, reducing latency from data transfers. Additionally, the potential use of shared memory for Sobel filters could optimise memory access patterns and decrease global memory accesses.
- **Boundary Handling:** Excluding boundary pixels simplifies the kernel's implementation and avoids complex conditional checks, leading to cleaner and more efficient code. However, this results in slightly smaller output images.
- **Approximate Normalisation Method:** Normalisation and clamping are simplified to maintain processing speed while ensuring that the output image maintains visual integrity, crucial for real-time applications.

Performance:

Benchmark Results:

Size	CPU Reference Timing (ms)	CUDA Timing (ms)
485 x 768	4.860	0.676
848 x 942	6.556	0.916
1347 x 1362	18.001	2.250
2017 x 1801	49.386	6.173
3281 x 3200	153.905	19.238

Analysis for Performance limitations:

- **Scaling with Image Size:** Performance gains are notable as image size increases, showcasing GPU efficiency in handling large data volumes. However, kernel launch and management overhead becomes more noticeable with larger images.
- **Memory Transfer Overhead:** The time required for transferring data between host and device memory increases with image size, potentially hindering performance.

Performance Improvements:

- **Optimise Memory Transfers:** Implementing CUDA streams and asynchronous transfers could lessen the impact of memory transfer times.
- **Kernel Optimization:** Varying block and grid sizes could enhance performance, particularly for larger images.
- **Shared Memory Utilisation:** Using shared memory to store pixel neighbourhoods can reduce global memory accesses and accelerate the convolution process.
- **Advanced Convolution Techniques:** Exploring tiled convolution or FFT-based methods may offer further performance benefits, especially for very large images.

CUDA: Algorithm: Data Structures

Description:

The CUDA data structure algorithm is engineered to compute histograms and boundary indices efficiently using GPU parallel processing, making it crucial for tasks like digital signal processing and statistical analysis. Key components of the algorithm include:

- **Histogram Computation:** The compute_histogram CUDA kernel processes each key in the dataset independently. Using atomic operations, it updates the histogram bins to ensure thread-safe modifications without race conditions.
  cuda

C/C++
```
_global_  void  compute_histogram(const  unsigned  int*  keys,  size_t
len_k, unsigned int* histogram) {

    // Calculate histogram values in parallel
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len_k) {

      // Atomically increment the histogram for key
      atomicAdd(&histogram[keys[idx]], 1);
    }
}
```

- **Boundary Computation:** The `**compute_boundaries**` kernel is then executed to compute the exclusive **prefix sum of the histogram** to determine boundary indices. Since this operation requires **global synchronisation** to ensure all histogram updates are complete before calculation, it is typically run with a single thread block.

C/C++
```
_global_  void  compute_boundaries(const  unsigned  int*  histogram,
unsigned int* boundaries, size_t len_b) {
    // Calculate boundary values in serial
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        for (size_t i = 0; i <= len_b - 2; ++i) {
            boundaries[i + 1] = boundaries[i] + histogram[i];
        }
    }
}
```

- **Memory Management:** Memory for the keys, histogram, and boundaries is allocated on the GPU to minimise data transfer overheads and maximise computational speed. After processing, results are transferred back to host memory for further use or analysis.

Justification:

- **Parallel Histogram Calculation:** Leveraging atomic operations allows the algorithm to efficiently handle concurrent updates, significantly outperforming sequential CPU methods, especially in large datasets where parallelism is crucial.
- **Atomic Operations for Data Integrity:** Atomic additions ensure data accuracy when multiple threads update the same histogram bin, crucial for maintaining the integrity of computation results.

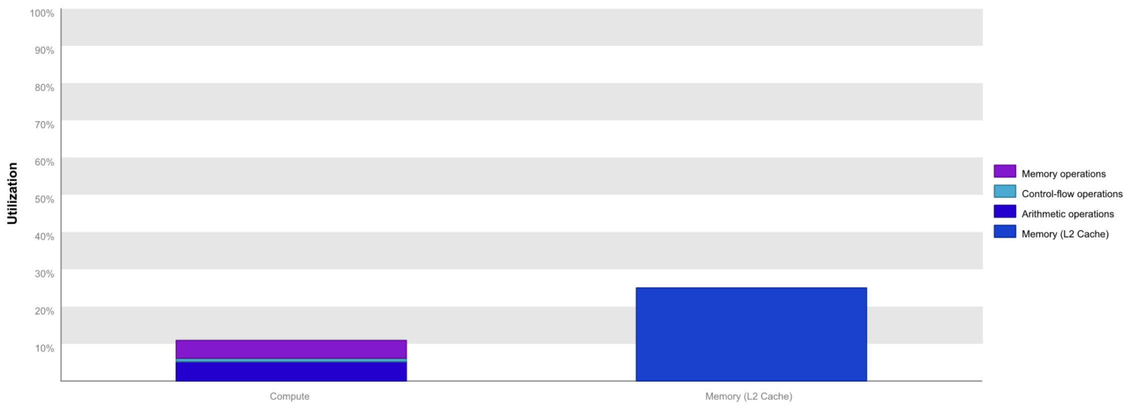
- **Single-Threaded Prefix Sum Calculation:** Using a single thread block for boundary computation avoids the complexity of parallel scans, which is practical given the lower computational intensity compared to histogram updates.
- **Optimised Memory Transfers:** Memory operations on the GPU reduce synchronisation penalties commonly associated with frequent data movement between host and device, critical for maintaining high performance.

Performance:

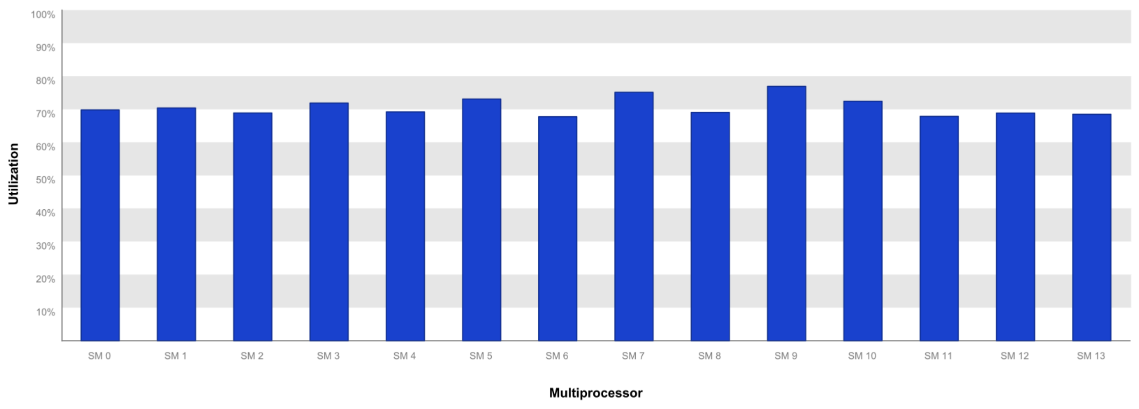
Benchmark Results:

Size	CPU Reference Timing (ms)	CUDA Timing (ms)
100	0.068	0.599
1000	0.071	0.575
10000	0.088	0.866
100000	0.555	2.670
1000000	4.614	24.244

Performance Analysis:



1. **Kernel Performance Is Bound by Instruction and Memory Latency**:
   - **Analysis**: This graph illustrates that the performance of the compute_histogram kernel is most likely limited by instruction and memory latency. It indicates low compute throughput and memory bandwidth utilisation relative to the peak performance of the GPU.



2. **Multiprocessor Utilisation**:

- **Analysis**: This graph shows the utilisation of each multiprocessor during the execution of the kernel. It provides insights into how the compute resources are distributed across the GPU and highlights if any multiprocessors are underutilised or overutilized.
- **Graph**: Include the chart showing the multiprocessor utilisation. This chart is crucial for understanding the distribution and efficiency of resource usage during kernel execution.

Analysis for Performance limitations:

- **Overhead of Small Datasets:** The algorithm shows considerable overhead for small datasets due to the GPU's parallel computation capabilities not being fully utilised, exacerbated by initialization and synchronisation costs.
- **Memory Transfer Overhead:** As dataset sizes increase, so does the time required for data transfer between host and device, noticeably affecting performance.
- **Atomic Operation Overhead:** The use of atomic operations introduces serialisation in memory access, becoming a bottleneck with increasing dataset sizes.
- **Kernel Launch and Synchronisation Costs:** Costs linked with CUDA kernel operations and necessary synchronisation post-execution are significant, especially
- for larger datasets.

Performance Improvements:

- **Optimised Memory Transfers:** Implementing pinned memory and asynchronous transfers could significantly enhance data transfer efficiency.
- **Kernel Configuration Optimization:** Adjusting grid and block configurations according to dataset size and GPU architecture could minimise overhead and maximise parallel execution efficiency.
- **Utilisation of Shared Memory:** Employing shared memory for storing intermediate results or frequently accessed data could decrease global memory traffic and boost performance.

- **Algorithm Refinement:** Splitting histogram computation into phases or employing advanced parallel algorithms could minimise synchronisation requirements and enhance scalability.