HPC Tutorial 8 Report
CS22B2012 K Aditya Sai

This report analyses the performance of CUDA parallel code on 10 million double precision floating-point number addition. The values range from 0.0 to 1000.0.

The floating point numbers are loaded into an array from a file.

Serial Code :

```
double sum = 0.0;
clock_t start = clock();
for (int i = 0; i < N; i++){
    sum += arr[i];
}
clock_t end = clock();
double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
```

The parallel code uses **1024** threads per block and the number of blocks is calculated as **(N + threads_per_block - 1) / threads_per_block**

Parallel Code(Atomic) :

```
__global__ void sum_n(double *a, double *sum){
    __shared__ double temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;

    if(threadIdx.x < N){
        temp[threadIdx.x] = a[index];
    }
    else{
        temp[threadIdx.x] = 0.0;
    }
    __syncthreads();
    if(threadIdx.x == 0){
        double block_sum = 0.0;
        for(int i = 0; i < blockDim.x; i++){
            block_sum += temp[i];
        }
        atomicAdd(sum, block_sum);
    }
}
```

Parallel Code :

```
__global__ void sum_n(double *numbers, double *result, int n)
{
    __shared__ double sharedSum[THREADS_PER_BLOCK];
    int tid = threadIdx.x;
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    sharedSum[tid] = (i < n) ? numbers[i] : 0.0;
    __syncthreads();

    for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
    {
        if (tid < stride)
        {
            sharedSum[tid] += sharedSum[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0)
    {
        atomicAdd(result, sharedSum[0]);
    }
}
```

```
double *d_a, *d_final_sum;
cudaMalloc(&d_a, N * sizeof(double));
cudaMemcpy(d_a, a, N * sizeof(double), cudaMemcpyHostToDevice);

cudaMalloc(&d_final_sum, sizeof(double));
cudaMemset(d_final_sum, 0, sizeof(double));

cudaDeviceSynchronize();
clock_t start = clock();

int block = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
sum_n<<<block, THREADS_PER_BLOCK>>>(d_a, d_final_sum, N);

cudaDeviceSynchronize();
clock_t end = clock();

cudaMemcpy(final_sum, d_final_sum, sizeof(double), cudaMemcpyDeviceToHost);

printf("Sum: %lf\n", *final_sum);
printf("Time taken: %lf seconds\n", ((double)(end - start)) / CLOCKS_PER_SEC);
```

Output:

Serial:

```
PS C:\Users\adity\OneDrive\Documents\HPC\Tut8> .\tut8serial.exe
Serial Sum = 4999938176.753002
Serial Time = 0.053000 seconds
```

Parallel with Atomic:

```
C:\Users\adity\OneDrive\Documents\HPC\Tut8>nvcc -arch=sm_86 tut8.cu -o p
tut8.cu
tmpxft_0000409c_00000000-10_tut8.cudafe1.cpp
   Creating library p.lib and object p.exp

C:\Users\adity\OneDrive\Documents\HPC\Tut8>.\p
Sum: 4999938176.752375
Time taken: 0.012000 seconds
```

Parallel Reduction:

```
C:\Users\adity\OneDrive\Documents\HPC\Tut8>nvcc -arch=sm_86 tut8.cu -o p
tut8.cu
tmpxft_00002208_00000000-10_tut8.cudafe1.cpp
   Creating library p.lib and object p.exp

C:\Users\adity\OneDrive\Documents\HPC\Tut8>.\p
Sum: 4999938176.752374
Time taken: 0.001000 seconds
```

Speedup (Atomic):

S = T(1) / T(p) = 0.053 / 0.012 = 4.41

Speedup (Reduction):

S = 0.053 / 0.001 = 53

Observations:

- Due to the immense size of the problem, the atomicAdd operations and the addition operations are expensive to synchronize the threads and blocks.
- To solve this, parallel reduction has been used in the block, where, using a stride, we halve the size of the array during addition, further reducing the time complexity.
- In the usage of the stride, at each iteration we divide the array into two halves and find the sum of corresponding elements and store it in one half of the array which will be halved again.

Time Complexity Analysis:

- Reduction - O(log THREADS_PER_BLOCK) + O(n / THREADS_PER_BLOCK)
- Atomic - O(THREADS_PER_BLOCK) + O(n / THREADS_PER_BLOCK)