

PPAR: GPU Lab1 and Lab2

Team

Ishan Tiwari

Shashi Mohan Reddy Ravula

Question1:

Below is the program for computing log2_series

```
float log2_series(int n)
{
    int i; float sum1 = 0;
    for (i = 0; i<n; i++)
    {
        if (i % 2 == 0)
        {
            sum1 += 1 / float(i + 1);
        }
        else
        {
            sum1 += -1 / float(i + 1);
        }
    }
    return sum1;
}
```

Question2:

When we perform the computation by summing the terms in increasing and decreasing order of indices. We see that the actual value is 0.693147, but in increasing number of indices we get 0.693138 and in decreasing number of indices we get 0.693147 which is the exact value. This variation is due to the order of summing the terms, as we studied in arithmetic that if we sum the series that has negative and positive numbers in it, then the order of summation will affect the result.

Below is the function for summing of terms in decreasing order.

```
float log2_series(int n)
{
    int i; float sum1 = 0;
    for (i = n-1; i>=0; i--)
    {
        if (i % 2 == 0)
        {
            sum1 += 1 / float(i + 1);
        }
    }
}
```

```

        }
        else
        {
            sum1 += -1 / float(i + 1);
        }
    }
    return sum1;
}

```

Question3

Strategy1: Every thread is adding `nthreads` number of elements.

```

int tid = threadIdx.x + blockIdx.x*blockDim.x;
int nthreads = blockDim.x * gridDim.x;
int begin = tid * nthreads ;
int end = begin + nthreads;
for (int i=begin; i<end; i=i++)

```

Strategy 2: Every thread is adding elements parted with a distance of `nthreads` number of elements.

```

int tid = threadIdx.x + blockIdx.x*blockDim.x;
int nthreads = blockDim.x * gridDim.x;
for (int i=tid; i<data_size; i=i+nthreads)

```

Question4

The strategy is to divide the elements in two arrays and sending both the arrays in the GPU and then adding one array element with other array element.

// Allocating output data on CPU

```

float final_cpu;
float * partial_cpu = (float*)malloc(blocks_in_grid*sizeof(float));
float * data_out_cpu = (float*)malloc(data_size*sizeof(float));

```

// Allocating output data on GPU

```

float * data_gpu, *partial_gpu, *block_partial, *final_gpu;
cudaMalloc((void**)&final_gpu, sizeof(float));
cudaMalloc((void**)&partial_gpu, blocks_in_grid*sizeof(float));
cudaMalloc((void**)&data_gpu, num_threads*sizeof(float));

```

// Copying from Host to Device

```

cudaMemcpy(partial_cpu,partial_gpu,blocks_in_grid*sizeof(float),
cudaMemcpyDeviceToHost);
cudaMemcpy(&final_cpu, final_gpu, sizeof(float), cudaMemcpyDeviceToHost);

```

// Get results back, since we have used partial_gpu to store the partial results from each block and then final sum in final_gpu and copying them in partial_cpu and final_cpu respectively.

```
// Finish reduction
```

```
float sum = 0.0;
for (int i = 0; i < blocks_in_grid; i++)
{
    printf("value of %f", partial_cpu[i]);
    sum += partial_cpu[i];
}
```

Question5

So we are getting two arrays, which we are adding together to get the results. We are adding the sum in the first array so we can use `data_out1` to get the results on cpu.

```
// GPU kernel
```

version 1: Strategy2

```
__global__ void summation_kernel1(int data_size, float * data_out)
{
    // TODO
    float sum = 0;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int nthreads = blockDim.x * gridDim.x;

    data_out[tid] = 0;
    for (int i = tid; i < data_size; i = i + nthreads)
    {
        if (i % 2 == 0)
        {
            sum = sum + 1 / float(i + 1);
        }
        else if (i % 2 == 1)
        {
            sum = sum - 1 / float(i + 1);
        }
    }
    data_out[tid] = sum;
}
```

version 2: Strategy1

```
__global__ void summation_kernel1_version2(int data_size, float * data_out)
{
    // TODO
    float sum = 0;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int nthreads = blockDim.x * gridDim.x;
    int begin = tid * nthreads;
    int end = begin + nthreads;
    data_out[tid] = 0;
    for (int i = begin; i < end; i = i++)
    {
        if (i % 2 == 0)
        {
            sum = sum + 1 / float(i + 1);
        }
    }
}
```

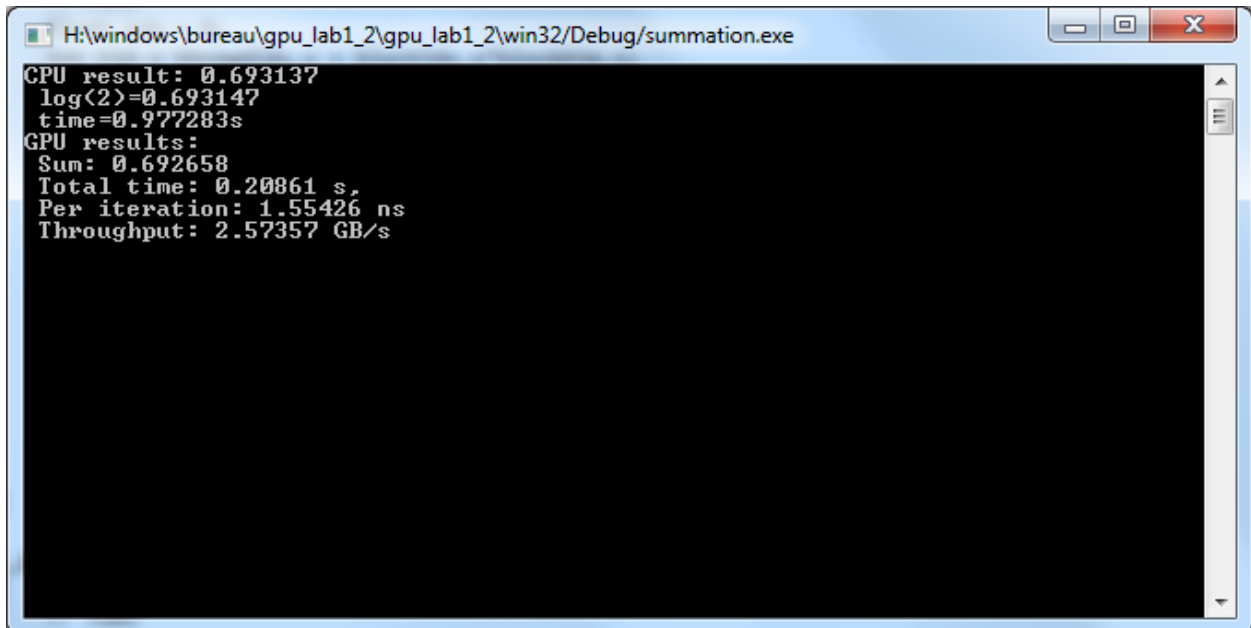
```

    }
    else if (i % 2 == 1)
    {
        sum = sum - 1 / float(i + 1);
    }
}
data_out[tid] = sum;
}

```

Question 6

From the snapshot below we can see the time taken by CPU is 0.977283 secs whereas time taken by GPU is 0.20861. This is because in the CPU version, there is only one thread running to add all the elements serially whereas in the GPU version we can see that the time is pretty less since it is using many threads at a time to compute the same parallelly.



The screenshot shows a Windows command prompt window with the title bar "H:\windows\bureau\gpu_lab1_2\gpu_lab1_2\win32\Debug\summation.exe". The window contains the following text:

```

CPU result: 0.693137
log(2)=0.693147
time=0.977283s
GPU results:
Sum: 0.692658
Total time: 0.20861 s.
Per iteration: 1.55426 ns
Throughput: 2.57357 GB/s

```

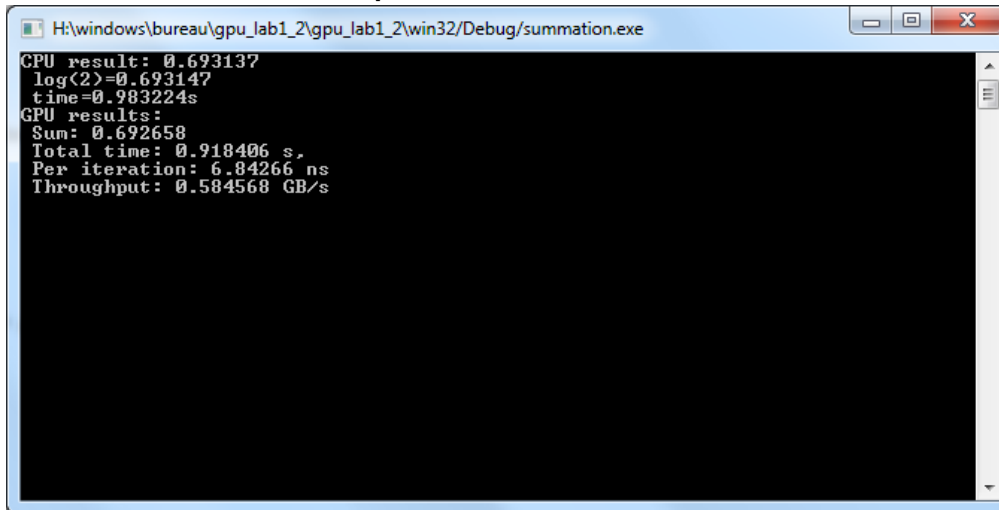
When we compared the two versions which we were using to compute the sum on the kernel we found that there was slight difference in the time but the result from strategy 2 was faster than the strategy one.

Question7

We changed blocks and threads and found that the best result when we tried to increase the number of blocks, whereas if we decreased the number of blocks it was taking little more time. Also, if we want to run the maximum number of threads per block available there was not much difference in the time.

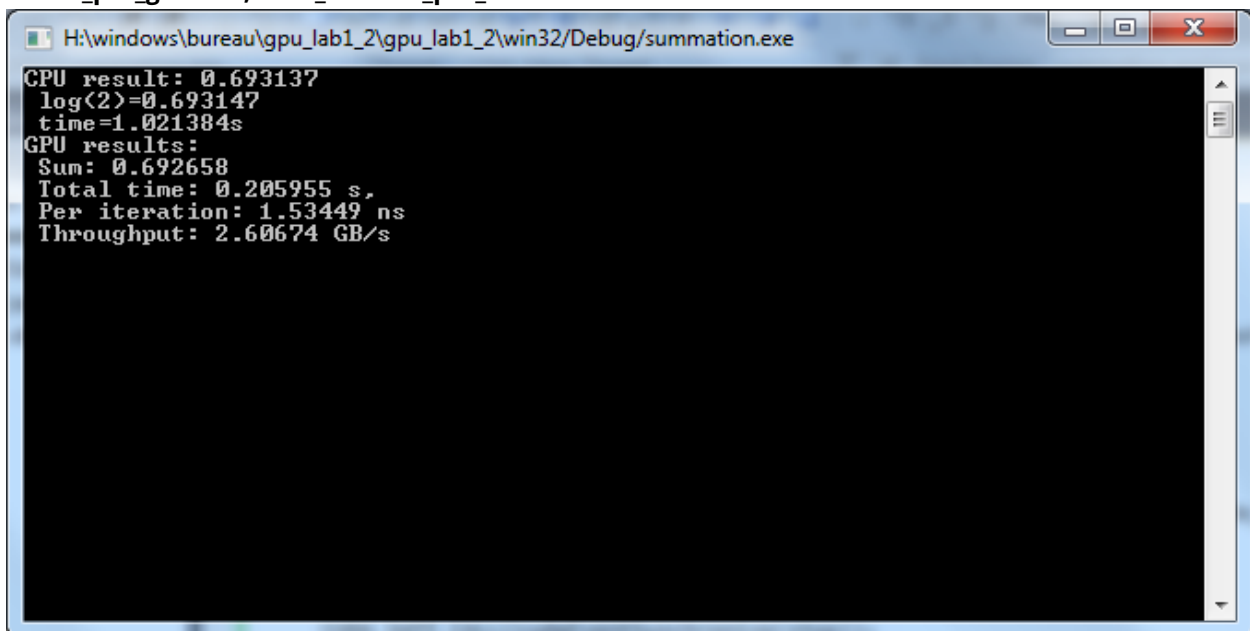
Eventually, the best time was to increase the number of blocks but that too in a limit so that the GPU can run the different number of blocks quickly. So we found **blocks_per_grid * 8** gave the best result.

Scenario1# one block, thread per blocks



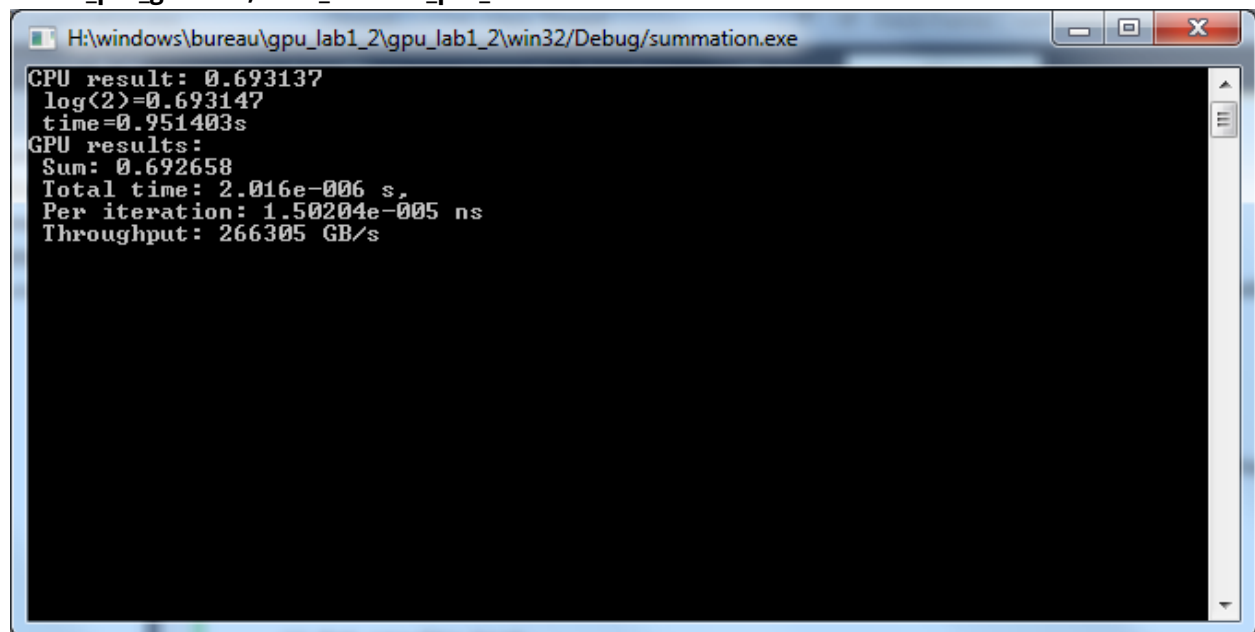
```
H:\windows\bureau\gpu_lab1_2\gpu_lab1_2\win32\Debug\summation.exe
CPU result: 0.693137
log(2)=0.693147
time=0.983224s
GPU results:
Sum: 0.692658
Total time: 0.918406 s.
Per iteration: 6.84266 ns
Throughput: 0.584568 GB/s
```

blocks_per_grid * 8 , num_threads_per_block



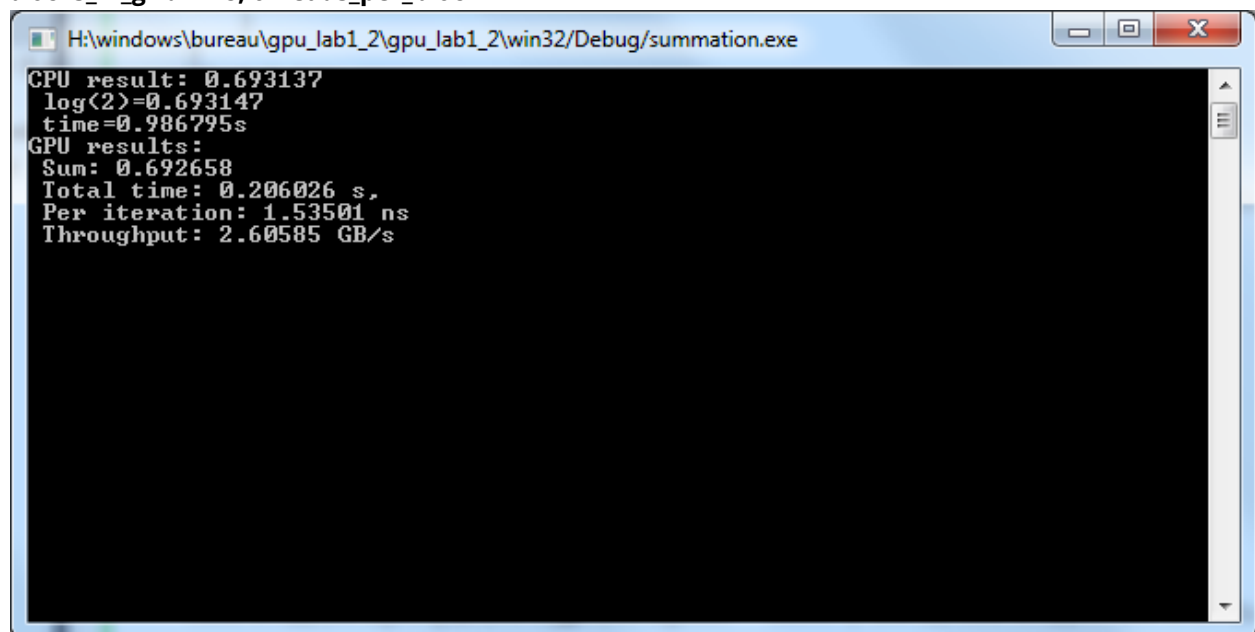
```
H:\windows\bureau\gpu_lab1_2\gpu_lab1_2\win32\Debug\summation.exe
CPU result: 0.693137
log(2)=0.693147
time=1.021384s
GPU results:
Sum: 0.692658
Total time: 0.205955 s.
Per iteration: 1.53449 ns
Throughput: 2.60674 GB/s
```

blocks_per_grid * 8 , num_threads_per_block*64



```
H:\windows\bureau\gpu_lab1_2\gpu_lab1_2\win32\Debug\summation.exe
CPU result: 0.693137
log(2)=0.693147
time=0.951403s
GPU results:
Sum: 0.692658
Total time: 2.016e-006 s,
Per iteration: 1.50204e-005 ns
Throughput: 266305 GB/s
```

blocks_in_grid*128, threads_per_block



```
H:\windows\bureau\gpu_lab1_2\gpu_lab1_2\win32\Debug\summation.exe
CPU result: 0.693137
log(2)=0.693147
time=0.986795s
GPU results:
Sum: 0.692658
Total time: 0.206026 s,
Per iteration: 1.53501 ns
Throughput: 2.60585 GB/s
```

Question 8

To reduce the amount of data transfer back we can use the shared memory of GPU which can be used to store the partial results. To get the one value per block we made a strategy to make another *summation_kernel2* where we will store the sum from each block in shared variable *sdata*. So we made one more where we gave input array from the *summation_kernel1* output array *data_gpu*.

```
summation_kernel1 << <blocks_in_grid, threads_per_block >> >(data_size, data_gpu);  
summation_kernel2 << <blocks_in_grid, threads_per_block, threads_per_block *  
sizeof(float) >> >(data_size, data_gpu, partial_gpu);
```

Question 9

To perform the whole computation on GPU we used one more kernel function *summation_kernel#* to take the one value per block and then adding them back to get the final result. So in this case the input array is *partial_gpu* from the previous kernel *summation_kernel2*.

```
summation_kernel1 << <blocks_in_grid, threads_per_block >> >(data_size, data_gpu);  
summation_kernel2 << <blocks_in_grid, threads_per_block, threads_per_block *  
sizeof(float) >> >(data_size, data_gpu, partial_gpu);  
summation_kernel3 << <1, blocks_in_grid, sizeof(float) >> >(partial_gpu, final_gpu);
```