Parallel Summation of N Double Precision Floating-Point Numbers: Serial C Code vs. CUDA

Problem Statement

The objective of this report is to analyze and compare the performance of serial and parallel techniques for summing large arrays of double-precision floating-point numbers. The problem involves writing a CUDA parallel program to compute the sum of N double-precision floating-point numbers, where N is at least 1 million. The input values can be stored in a file and read from it during the computation.

The goal is to:

- Implement a serial version of the code to compute the sum of the array elements.
- Implement a parallel version of the code using CUDA, leveraging the GPU to accelerate the summation process.
- Estimate the time taken by the serial code for summing the elements.
- Estimate the time taken by the parallel CUDA code.
- Calculate the speedup achieved by using the parallel approach compared to the serial one.

The purpose is to evaluate the performance improvements from parallelization, especially when working with very large arrays.

Compute Power of the System

The system is powered by the **NVIDIA GeForce GTX 1650** graphics card, based on the *TU117* GPU architecture. Below are the detailed specifications of the GPU:

Graphics Processor Specifications

- **GPU Name:** TU117
- **GPU Variant:** TU117-300-A1
- **Architecture:** Turing
- **Process Size:** 12 nm
- **Die Size:** 200 mm²
- **Transistors:** 4.7 billion
- **DirectX Support:** 12
- **Shader Model:** 6.8
- **CUDA:** 7.5
- **NVENC:** 5th Gen
- **NVDEC:** 4th Gen
- **PureVideo HD:** VP10
- **VDPAU:** Feature Set J

Core Configuration

- **Shading Units (CUDA Cores):** 896
- **Texture Mapping Units (TMUs):** 56
- **Render Output Units (ROPs):** 32
- **Streaming Multiprocessors (SM):** 14
- **L1 Cache (per SM):** 64 KB
- **L2 Cache:** 1024 KB

Memory Specifications

- **Memory Size:** 4 GB GDDR5
- **Memory Bus:** 128 bit
- **Memory Clock:** 2001 MHz (8 Gbps effective)
- **Memory Bandwidth:** 128.1 GB/s

Clock Speeds

- **Base Clock:** 1485 MHz
- **Boost Clock:** 1665 MHz
- **Memory Clock:** 2001 MHz (8 Gbps effective)

Performance Metrics

- **Pixel Rate:** 53.28 GPixel/s
- **Texture Rate:** 93.24 GTexel/s
- **FP32 Performance (float):** 2.984 TFLOPS
- **FP16 Performance (half):** 5.967 TFLOPS
- **FP64 Performance (double):** 93.24 GFLOPS

Power and Thermal Design

- **Thermal Design Power (TDP):** 75 W
- **Recommended PSU:** 250 W
- **Slot Width:** Dual-slot
- **Power Connectors:** None

Display and Connectivity

- **Outputs:** 1x DVI, 1x HDMI 2.0, 1x DisplayPort 1.4a
- **Bus Interface:** PCIe 3.0 x16
- **Supported Resolutions:** 1920x1080, 2560x1440, 3840x2160

Relative Performance

At launch, the GeForce GTX 1650 performed similarly to other mid-range graphics cards in the market, with a launch price of 149 USD. It offers an excellent balance of price and performance for mainstream gaming and light productivity tasks.

Serial Code

The serial implementation of the summation computes the sum of N double-precision floating-point numbers stored in an array. The array is populated with the numbers from 1 to N, and the sum is computed using a simple loop.

Code Explanation

The serial code allocates memory for an array of size N and fills the array with values from 1 to N. The summation is performed in a loop over all the elements. The time taken for the summation is measured using the `clock()` function.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000000  // Number of elements (10 million)

int main() {
    double *data = (double*)malloc(N * sizeof(double));
    double sum = 0.0;

    // Generate numbers from 1 to N
    for (int i = 0; i < N; i++) {
        data[i] = i + 1;
    }

    // Serial summing of numbers
    clock_t start_time = clock();
    for (int i = 0; i < N; i++) {
        sum += data[i];
    }
    clock_t end_time = clock();

    // Print the sum and execution time
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Serial Sum: %f\n", sum);
    printf("Serial Time: %f seconds\n", time_taken);

    free(data);
    return 0;
}
```

Output

The output of the serial summation code is as follows:

```
gcc sumOfN_serial.c
./a.out
Serial Sum: 50000005000000.000000
Serial Time: 0.042723 seconds
```

Parallel Code using CUDA

The parallel implementation utilizes CUDA, a parallel computing platform, and API model, to offload the summation computation to the GPU. The parallel code divides the task of summing the array into smaller sub-tasks, where each thread in the GPU computes part of the sum and the results are later combined.

Code Explanation

In the parallel code, the array is allocated and populated in the same way as the serial code. The main difference is the use of CUDA for parallelization. A kernel function `sum_kernel` is defined to compute the sum of the array in parallel across multiple threads. The kernel performs reduction within each block using shared memory and then combines the results from all blocks using atomic addition.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

#define N 10000000  // Number of elements (10 million)
#define THREADS_PER_BLOCK 256

__global__ void sum_kernel(double *data, double *result, int n) {
    __shared__ double shared_data[THREADS_PER_BLOCK];

    int thread_id = threadIdx.x + blockIdx.x * blockDim.x;
    int index = thread_id;

    // Initialize shared memory with 0
    shared_data[threadIdx.x] = (index < n) ? data[index] : 0.0;

    // Synchronize threads within a block
    __syncthreads();

    // Perform reduction in shared memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (threadIdx.x < stride) {
            shared_data[threadIdx.x] += shared_data[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Only one thread in each block writes the sum of the block to the global result
    if (threadIdx.x == 0) {
        atomicAdd(result, shared_data[0]);
    }
}

int main() {
    double *data = (double*)malloc(N * sizeof(double));
    double *d_data, *d_result, result = 0.0;

    // Generate numbers from 1 to N
    for (int i = 0; i < N; i++) {
        data[i] = i + 1;
    }

    // Allocate memory on the GPU
    cudaMalloc((void**)&d_data, N * sizeof(double));
    cudaMalloc((void**)&d_result, sizeof(double));
    cudaMemcpy(d_data, data, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_result, &result, sizeof(double), cudaMemcpyHostToDevice);

    // Calculate grid and block sizes
    int block_size = THREADS_PER_BLOCK;
    int grid_size = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

    // Launch CUDA kernel to compute the sum
    clock_t start_time = clock();
    sum_kernel<<<grid_size, block_size>>>(d_data, d_result, N);
    cudaDeviceSynchronize();
    clock_t end_time = clock();

    // Copy the result back to the host
    cudaMemcpy(&result, d_result, sizeof(double), cudaMemcpyDeviceToHost);

    // Print the sum and execution time
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Parallel Sum: %f\n", result);
    printf("Parallel Time: %f seconds\n", time_taken);

    // Free memory
    free(data);
    cudaFree(d_data);
    cudaFree(d_result);

    return 0;
}
```

Output

The output of the parallel summation code is as follows:

```
nvcc -arch=sm_60 sumOfN_cuda.cu
./a.out
Parallel Sum: 50000005000000.000000
Parallel Time: 0.002711 seconds
```

Explanation of the Parallel Reduction Loop

The purpose of the following CUDA kernel loop is to perform a parallel reduction using shared memory. In this case, we will explain how the loop works step-by-step, particularly focusing on how the reduction is achieved by reducing the number of active threads in a logarithmic fashion.

```
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
    if (threadIdx.x < stride) {
        shared_data[threadIdx.x] += shared_data[threadIdx.x + stride];
    }
    __syncthreads();
}
```

1. Initialization of the `stride` Variable

The loop begins by initializing the `stride` variable to half of the block size (`blockDim.x / 2`). The `stride` will control how far apart the thread pairs are in each iteration, and will decrease logarithmically (`stride >>= 1`) in each iteration. This ensures that the number of active threads will be halved in each step, and the reduction will continue in logarithmic time.

$$\text{stride} = \frac{\text{blockDim.x}}{2}$$

2. Combining Data (Parallel Reduction)

The reduction is achieved through the following statement:

```
if (threadIdx.x < stride) {
    shared_data[threadIdx.x] += shared_data[threadIdx.x + stride];
}
```

- `threadIdx.x` is the thread's index within the block, and each thread works on its own corresponding element in the `shared_data` array. - The condition `if (threadIdx.x < stride)` ensures that only the first `stride` threads participate in the reduction step. - The `shared_data[threadIdx.x]` holds the value that will be reduced, and `shared_data[threadIdx.x + stride]` is the value of the thread `stride` positions away. - The threads add the values of the corresponding positions together:

$$\text{shared\_data[threadIdx.x]} \mathrel{+}= \text{shared\_data[threadIdx.x + stride]}$$

This allows each pair of threads to combine their data, effectively halving the number of active threads.

3. Synchronization

The `__syncthreads()` function ensures that all threads in the block finish their work (i.e., the addition operation) before proceeding to the next iteration:

```
__syncthreads();
```

Without synchronization, some threads might try to read data that other threads have not yet written, leading to incorrect results. This synchronization ensures that the next step starts only after all updates to the shared memory are completed.

4. How the Reduction Works Step-by-Step

Consider the following example where there are 8 threads in the block (`blockDim.x = 8`). We will walk through the reduction process.

Initial Shared Data:

$$\text{shared\_data} = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

First Iteration (stride = 4)

In the first iteration, `stride` is 4. The threads `0, 1, 2, 3` will combine their data with the threads `4, 5, 6, 7`, respectively:

$$\text{Thread 0 adds: shared\_data[0]} + \text{shared\_data[4]} = 1 + 5 = 6$$
$$\text{Thread 1 adds: shared\_data[1]} + \text{shared\_data[5]} = 2 + 6 = 8$$
$$\text{Thread 2 adds: shared\_data[2]} + \text{shared\_data[6]} = 3 + 7 = 10$$
$$\text{Thread 3 adds: shared\_data[3]} + \text{shared\_data[7]} = 4 + 8 = 12$$

After the first iteration, the shared memory looks like this:

$$\text{shared\_data} = \{6, 8, 10, 12, 5, 6, 7, 8\}$$

Second Iteration (stride = 2)

In the second iteration, `stride` is 2. The threads `0, 1` will now combine their data with threads `2, 3`, respectively:

$$\text{Thread 0 adds: } \text{shared\_data}[0] + \text{shared\_data}[2] = 6 + 10 = 16$$

$$\text{Thread 1 adds: } \text{shared\_data}[1] + \text{shared\_data}[3] = 8 + 12 = 20$$

After the second iteration, the shared memory looks like this:

$$\text{shared\_data} = \{16, 20, 10, 12, 5, 6, 7, 8\}$$

Third Iteration (stride = 1)

In the third iteration, `stride` is 1. Thread 0 combines its value with thread 1:

$$\text{Thread 0 adds: } \text{shared\_data}[0] + \text{shared\_data}[1] = 16 + 20 = 36$$

After the third iteration, the shared memory looks like this:

$$\text{shared\_data} = \{36, 20, 10, 12, 5, 6, 7, 8\}$$

At this point, thread 0 holds the final result of the reduction (in this case, the sum of all elements). This result is then written to global memory.

Summary

In summary, this loop performs a parallel reduction in shared memory. By halving the number of threads in each iteration (`stride >>= 1`), the threads combine their data in a pairwise fashion, reducing the active threads logarithmically. After the reduction, only thread 0 in each block holds the final reduced value, which is written to the global result.

Using the `nvcc` Compiler with the `-arch` Flag

The `nvcc` (NVIDIA CUDA Compiler) command is used to compile CUDA programs. When specifying the `-arch` flag with an architecture version (like `sm_60`), it instructs the compiler to generate code optimized for a specific compute capability, which corresponds to the architecture of the target GPU.

In this section, we explain the command `nvcc -arch=sm_60 your_program.cu` and its significance in targeting GPUs with different compute capabilities.

Syntax of the `nvcc` Command

```
nvcc -arch=sm_60 your_program.cu
```

Where:

- `nvcc`: This is the NVIDIA CUDA Compiler. It compiles the CUDA code into machine code that can run on an NVIDIA GPU.
- `-arch=sm_60`: This flag specifies the target architecture. In this case, `sm_60` refers to the compute capability 6.0, which corresponds to the **Pascal** architecture.
- `your_program.cu`: This is the source file containing your CUDA code. The `.cu` extension indicates that it is a CUDA C++ source file.

What is `sm_60`?

`sm_60` refers to **compute capability 6.0**, which is associated with the **Pascal** architecture, introduced with the Tesla P100 GPUs. Each compute capability corresponds to a version of NVIDIA's GPU architecture and determines what features and optimizations are available.

The compute capability of a GPU is identified by the `sm_` prefix followed by a version number. For example:

- `sm_30`: Kepler architecture (e.g., Tesla K40, K80)
- `sm_35`: Kepler architecture (e.g., Tesla K20)
- `sm_50`: Maxwell architecture (e.g., Tesla M40, P4)
- `sm_60`: Pascal architecture (e.g., Tesla P100, P40)
- `sm_70`: Volta architecture (e.g., Tesla V100)
- `sm_80`: Ampere architecture (e.g., A100)

Each newer compute capability introduces more advanced features, including support for atomic operations, more registers, and additional CUDA instructions.

How Does `-arch=sm_60` Affect the Compilation?

The `-arch` flag tells `nvcc` to compile the code for a specific GPU architecture. By specifying `-arch=sm_60`, you are instructing the compiler to generate code optimized for GPUs with compute capability 6.0 (i.e., Pascal-based GPUs). This ensures that the compiled code will take advantage of the features and optimizations of the Pascal architecture, such as:

- Double-precision atomic operations (important for scientific computing).
- Higher number of registers per thread.
- Improved performance for certain types of calculations.

Why is `-arch=sm_60` Important for `atomicAdd(double)`?

The function `atomicAdd` supports atomic operations for data types like `int`, `float`, and some others. However, **atomic operations on `double` are supported only on GPUs with compute capability 6.0 and higher**. Therefore, specifying `-arch=sm_60` ensures that the compiled code will be optimized for GPUs that support `atomicAdd` for `double`-precision floating-point numbers.

If you were targeting a GPU with a lower compute capability (e.g., `sm_35`), `atomicAdd` would not support `double`, and this would lead to compilation errors. By specifying `sm_60`, you ensure that the code will compile and run on GPUs with compute capability 6.0 or higher, which do support atomic operations for `double`.

Example of Using the `nvcc` Command

Consider a simple CUDA program `my_kernel.cu` that you want to compile for a GPU with compute capability 6.0:

```
nvcc -arch=sm_60 my_kernel.cu -o my_kernel
```

Here:

- `nvcc`: Invokes the CUDA compiler.
- `-arch=sm_60`: Targets the Pascal architecture (compute capability 6.0).
- `my_kernel.cu`: The source file containing the CUDA kernel.
- `-o my_kernel`: Specifies the output executable name.

This command tells `nvcc` to compile `my_kernel.cu` with optimizations for GPUs that have compute capability 6.0, generating the executable file `my_kernel`.

Targeting Different Compute Capabilities

If you want to target different architectures, you can specify different compute capabilities. For example:

- `-arch=sm_70`: Targets GPUs with compute capability 7.0 (Volta architecture, e.g., Tesla V100).
- `-arch=sm_80`: Targets GPUs with compute capability 8.0 (Ampere architecture, e.g., A100).

By choosing the correct architecture, you ensure that your program is optimized for the hardware capabilities of the GPU you are targeting.

Conclusion

Using the `-arch` flag with `nvcc` allows you to target specific GPU architectures and optimize the CUDA code for the features and capabilities of that architecture. For instance, `-arch=sm_60` targets GPUs with compute capability 6.0, which is essential for enabling features like double-precision atomic operations. Understanding the compute capability of your target GPU helps ensure that your program can take advantage of the appropriate hardware features.

Performance Comparison

To evaluate the performance gain achieved by parallelization, we compare the execution times of the serial and parallel implementations.

Speedup Calculation

The speedup can be calculated using the formula:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

Using the values obtained from the outputs:

$$\text{Speedup} = \frac{0.042723}{0.002711} \approx 15.75$$

Thus, the parallel implementation is approximately 15.75 times faster than the serial implementation.

Conclusion

In this report, we implemented a serial and parallel version of the summation of large arrays using double precision floating-point numbers. The parallel implementation, utilizing CUDA, resulted in a significant speedup over the serial implementation. The parallel code achieved a speedup of approximately 15.75 times, demonstrating the efficiency of parallel computing for large-scale summation tasks.