Vector Dot Product Parallelization: Serial C Code vs. CUDA

Problem Statement

The task is to implement the vector dot product of two large vectors of double-precision floating point numbers using both serial and parallel methods. The serial method calculates the dot product sequentially, while the parallel method leverages CUDA to perform the dot product calculation using multiple threads on a GPU. The goal is to estimate the time taken by both implementations and calculate the speedup obtained by parallelizing the code.

For this task, we used vectors of size 10 million elements. The vectors are initialized with sequential values for simplicity, but this could be adapted to load large values from files.

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 mm^2
- **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 for Vector Dot Product

The serial implementation is straightforward. The dot product of two vectors A and B is calculated as:

$$\text{Dot Product} = \sum_{i=0}^{N-1} A[i] \times B[i]$$

Where N is the number of elements in the vectors.
The serial code is:

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

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

// Serial function for vector dot product
double vector_dot_product(double *A, double *B) {
    double dot_product = 0.0;
    for (int i = 0; i < N; i++) {
        dot_product += A[i] * B[i];
    }
```

```
    return dot_product;
}

int main() {
    double *A, *B;
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));

    // Initialize the vectors with sequential values
    for (int i = 0; i < N; i++) {
        A[i] = i + 1;
        B[i] = N - i;
    }

    // Measure the time taken by the serial dot product function
    clock_t start = clock();
    double result = vector_dot_product(A, B);
    clock_t end = clock();
    double serial_time = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Serial Dot Product: %f\n", result);
    printf("Serial Time: %f seconds\n", serial_time);

    // Cleanup
    free(A);
    free(B);

    return 0;
}
```

This code calculates the dot product of vectors A and B by iterating through the elements and summing the product of corresponding elements. It measures the time taken to execute this function and prints the result.

Parallel Code using CUDA

In the parallel implementation, the vector dot product is divided into multiple threads, each of which computes a partial product. The results are then reduced in parallel.

The parallel CUDA code consists of two main kernels: 1. The `vector_dot_kernel` computes the partial dot products. 2. The `reduce_sum` kernel performs a reduction to sum the partial results from all threads.

The parallel code is:

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

```
#include <cuda.h>

#define N 10000000  // Define the size of the vectors
#define THREADS_PER_BLOCK 256

// CUDA kernel for partial dot product
__global__ void vector_dot_kernel(double *A, double *B, double *C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] * B[idx];  // Calculate individual products
    }
}

// CUDA kernel to sum the partial results
__global__ void reduce_sum(double *C, double *sum) {
    __shared__ double shared_data[THREADS_PER_BLOCK];
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int tid = threadIdx.x;

    if (idx < N) {
        shared_data[tid] = C[idx];
    } else {
        shared_data[tid] = 0.0;
    }
    __syncthreads();

    // Perform reduction within the block
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (tid < stride) {
            shared_data[tid] += shared_data[tid + stride];
        }
        __syncthreads();
    }

    // Store the block's result in the global memory
    if (tid == 0) {
        atomicAdd(sum, shared_data[0]);
    }
}

int main() {
    double *A, *B, *C;
    double *d_A, *d_B, *d_C, *d_sum;
    double sum = 0.0;

    // Allocate memory for the vectors
```

```
A = (double*)malloc(N * sizeof(double));
B = (double*)malloc(N * sizeof(double));
C = (double*)malloc(N * sizeof(double));

// Initialize the vectors with sequential values
for (int i = 0; i < N; i++) {
    A[i] = i + 1;
    B[i] = N - i;
}

// Allocate device memory
cudaMalloc(&d_A, N * sizeof(double));
cudaMalloc(&d_B, N * sizeof(double));
cudaMalloc(&d_C, N * sizeof(double));
cudaMalloc(&d_sum, sizeof(double));

// Copy input vectors from host to device
cudaMemcpy(d_A, A, N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, N * sizeof(double), cudaMemcpyHostToDevice);

// Initialize sum on device
cudaMemcpy(d_sum, &sum, sizeof(double), cudaMemcpyHostToDevice);

// Launch kernel for partial dot product
int blocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

vector_dot_kernel<<<blocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C);

// Launch kernel to reduce the partial results
reduce_sum<<<blocks, THREADS_PER_BLOCK>>>(d_C, d_sum);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float parallel_time = 0.0f;
cudaEventElapsedTime(&parallel_time, start, stop);

// Copy the final sum from device to host
cudaMemcpy(&sum, d_sum, sizeof(double), cudaMemcpyDeviceToHost);

printf("CUDA Dot Product: %f\n", sum);
printf("CUDA Time: %f seconds\n", parallel_time / 1000.0);
```

```
    // Cleanup
    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_sum);

    return 0;
}
```

In this code: - `vector_dot_kernel` calculates the element-wise product of vectors A and B. - `reduce_sum` performs a reduction on the results to calculate the final dot product, using shared memory and atomic operations to sum the results from all blocks.

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:

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

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:

$$\text{Thread 0 adds: } \text{shared\_data}[0] + \text{shared\_data}[4] = 1 + 5 = 6$$

$$\text{Thread 1 adds: } \text{shared\_data}[1] + \text{shared\_data}[5] = 2 + 6 = 8$$

$$\text{Thread 2 adds: } \text{shared\_data}[2] + \text{shared\_data}[6] = 3 + 7 = 10$$

$$\text{Thread 3 adds: shared\_data}[3] + \text{shared\_data}[7] = 4 + 8 = 12$$

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:

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

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

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:

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

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.

CUDA Compilation Instructions

To compile the CUDA code that includes atomic operations (such as `atomicAdd`), it is important to specify the correct architecture flag, which indicates the compute capability of the target GPU. The architecture flag ensures that the code is compiled in a way that is compatible with the features and capabilities of the specific GPU.

Here are the key architecture flags and the corresponding GPUs:

- **sm_20 - Fermi** architecture (compute capability 2.x): This is the architecture for GPUs like Tesla C2050/C2070, which introduced CUDA support but has limited atomic operations.
- **sm_30 - Kepler** architecture (compute capability 3.x): Kepler introduced significant performance improvements and support for atomic operations. The GTX 600 and 700 series GPUs are based on the Kepler architecture. Atomic operations were supported from `sm_30` onward.
- **sm_35 - Kepler** architecture (compute capability 3.5): This includes GPUs like the Tesla K40/K80. It offered more advanced features, such as better memory management.
- **sm_50 - Maxwell** architecture (compute capability 5.x): Maxwell architecture, found in GPUs like the GTX 900 series, introduced power efficiency improvements, but it's still based on a relatively similar CUDA programming model compared to Kepler. Maxwell supports atomic operations.
- **sm_60 - Pascal** architecture (compute capability 6.x): Pascal-based GPUs, such as the GTX 10-series (e.g., GTX 1080), provide significant performance improvements in both computation and memory efficiency. This architecture supports atomic operations at a high level, and the `sm_60` flag is used for this architecture.
- **sm_70 - Volta** architecture (compute capability 7.x): Volta introduced major improvements in deep learning and AI workloads, with hardware support for tensor cores. It includes GPUs like the Tesla V100. The `sm_70` flag corresponds to this architecture.
- **sm_80 - Ampere** architecture (compute capability 8.x): The Ampere architecture, used in GPUs such as the RTX 30 series (e.g., RTX 3090), provides improved performance for a wide range of computational workloads, including AI, ray tracing, and parallel processing. The `sm_80` flag is used for this architecture.
- **sm_90 - Hopper** architecture (compute capability 9.x): The latest architecture, with GPUs like the A100, providing extreme performance for machine learning and large-scale computation tasks.

To compile the CUDA code with atomic operations (such as `atomicAdd`), you need to specify the architecture that matches your GPU. For example, if you're using a Pascal GPU, such as the GTX 1080, you would use the `-arch=sm_60` flag. If you're using a Turing GPU (e.g., RTX 2080), you should use `-arch=sm_70`.

Example Compilation Command:

To compile the CUDA code with atomic operations for a Pascal architecture GPU (with compute capability 6.0), use the following command:

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

Here, `-arch=sm_60` specifies that the code should be compiled for GPUs with compute capability 6.0 (Pascal architecture). If you are using a different architecture, simply replace `sm_60` with the corresponding architecture flag for your GPU. For instance:

- For Kepler, use `-arch=sm_30` or `-arch=sm_35`
- For Maxwell, use `-arch=sm_50`
- For Pascal, use `-arch=sm_60`
- For Volta, use `-arch=sm_70`
- For Ampere, use `-arch=sm_80`

Notes on Choosing the Right Architecture Flag:

- Always choose the flag corresponding to the GPU's compute capability. If you're unsure about the compute capability of your GPU, you can check it on NVIDIA's official website or use the `deviceQuery` tool provided by the CUDA toolkit.
- If you're targeting multiple architectures, you can use the `-arch=compute_60` flag and then specify the `-code` option to generate PTX code for multiple architectures (e.g., `-code=sm_60,sm_70`).

Once the code is compiled, you can run the resulting executable to perform the vector dot product computation on the GPU.

Results and Speedup Calculation

The results obtained from the serial and parallel implementations are as follows:

- **Serial Dot Product:** 166666716666670284800.000000
- **Serial Time:** 0.047956 seconds
- **CUDA Dot Product:** 166666716666769375232.000000
- **CUDA Time:** 0.005146 seconds

The speedup S achieved by parallelizing the dot product computation is calculated as:

$$S = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{0.047956}{0.005146} \approx 9.32$$

Thus, the parallel code is approximately 9.32 times faster than the serial code.

Explanation of Differences in Results

The slight difference between the serial and parallel results arises due to the non-associative nature of floating point arithmetic. In the serial version, the dot product is calculated in a single pass, with the operations performed in a fixed order. However, in the parallel implementation, the dot product is divided into multiple threads, and partial results are summed in parallel. This introduces small variations due to:

- **Floating-point precision issues:** The order of operations in parallel execution may affect the rounding errors, leading to minor discrepancies in the final result. For example, consider the sum of three numbers: $A = 1.0000001$, $B = 1.0000002$, and $C = 1.0000003$. In serial computation, the sum is computed as $(A + B) + C$, which yields 3.0000006. In a parallel computation, if the numbers are summed in a different order, such as $A+(B+C)$, the result could slightly differ due to the precision of floating-point operations.
- **Reduction in parallel code:** The atomic addition used to accumulate results might also introduce slight variations in precision. For instance, in the parallel reduction, each block may compute a partial sum of products, and when these partial results are combined using atomic addition, the order in which the sums are performed can influence the final result. This is because atomic operations are not associative, meaning the final sum may depend on the order in which partial results are added, introducing slight errors in the accumulation process.

These differences are generally negligible in most cases but can be observed when dealing with large numbers of operations and floating-point arithmetic. Such small discrepancies can be amplified when large vectors with millions of elements are involved, as in the case of the dot product computation here.

Conclusion

This report compared the serial and parallel implementations of the vector dot product. By utilizing CUDA, we achieved a speedup of approximately 9.32 times over the serial version. The parallel approach reduces computation time by distributing the task across multiple GPU threads.

Although a small discrepancy between the serial and parallel results was noted, primarily due to the non-associative nature of floating-point arithmetic, the parallel implementation remains highly efficient. Overall, CUDA parallelization significantly accelerates computationally intensive tasks like the vector dot product.