Homework Assignment 4: Multi-GPU Vector Dot Product Implementation

Haohan Tsao B09607009

1 Introduction

This report presents the implementation and analysis of a vector dot product algorithm using multiple GPUs. The dot product of two vectors A and B of length N is defined as:

$$A \cdot B = \sum_{i=0}^{N-1} A_i \times B_i \tag{1}$$

While the operation is straightforward to compute sequentially, our task was to extend the single-GPU implementation to utilize multiple GPUs for potentially improved performance. The assignment required testing with 2 GPUs using random vectors of size 40,960,000 elements, and determining the optimal block size and grid size for this problem on NVIDIA GeForce GTX 1060 GPUs.

2 Implementation Approach

2.1 Multi-GPU Parallelization Strategy

Our approach for multi-GPU dot product calculation follows the domain decomposition pattern:

1. **Data Partitioning**: The input vectors are evenly divided among available GPUs.
2. **Parallel Execution**: Each GPU computes a partial dot product for its assigned portion.
3. **Result Aggregation**: The partial results from all GPUs are combined on the host to produce the final dot product.

This implementation used OpenMP to manage multiple GPU threads, with each thread controlling one GPU. The key aspects of the implementation include:

- **Vector Distribution**: Each GPU processes approximately $N/2$ elements.
- **Thread Management**: OpenMP creates threads that each handle one GPU.
- **Workload Balance**: Elements are evenly distributed to ensure balanced workload.
- **Memory Management**: Each GPU allocates and manages its own memory for its vector portion.

2.2 Implementation Details

The implementation consists of the following components:

- `vecDotMultiGPU.cu`: The main CUDA program that implements:
  - `VecDot` kernel: Performs partial dot product with parallel reduction
  - OpenMP parallel region for controlling multiple GPUs
  - Memory allocation and data transfer for each GPU
  - Results aggregation and verification
- The `VecDot` kernel uses a combination of:
  - Shared memory for efficient reduction within thread blocks
  - Parallel reduction algorithm that halves active threads in each step
  - Loop structure to handle vectors larger than the total number of threads

3 Results and Analysis

3.1 Performance Across Different Configurations

We systematically tested various combinations of threads per block (32 to 1024) and blocks per grid (64 to 2048) to determine the optimal configuration. Figure 1 conceptually represents the performance landscape.

Figure 1: Performance heat map across thread and block configurations

3.2 Optimal Configuration Analysis

Based on our experiments, the configurations with the best performance were:

Threads/Block	Blocks/Grid	Processing Time (ms)	GPU GFlops	Accuracy
512	512	75.774879	1.081097	2.55e-07
1024	2048	75.849983	1.080027	1.77e-07
512	1024	75.803101	1.080695	1.73e-07

Table 1: Top performing configurations

The optimal configuration is 512 threads per block with 512 blocks per grid, achieving a processing time of 75.77 ms. This configuration can be understood in terms of several performance factors:

- **Warp Efficiency**: 512 threads means exactly 16 warps per block. This provides sufficient parallelism for efficient execution.
- **Resource Utilization**: The GTX 1060 has 10 SMs, each capable of handling multiple blocks. With 512 blocks distributed across 10 SMs, each SM manages approximately 51 blocks, providing good occupancy.
- **Memory Access Patterns**: The dot product operation involves sequential memory access, which benefits from coalesced memory transactions that occur when threads within a warp access contiguous memory locations.
- **Balance of Parallelism**: This configuration strikes a balance between having enough threads to hide memory latency and not too many to cause resource contention.

3.3 Multi-GPU Scaling Analysis

An interesting observation from our results is that the GPU implementation (even with 2 GPUs) consistently underperforms compared to the CPU implementation, with speedup values around 0.5. This counter-intuitive result can be explained by:

- **Data Transfer Overhead**: For vectors of size 40,960,000, transferring data to and from the GPUs requires significant time. This overhead

negates the computational advantage of GPUs for this specific problem size.

- **P2P Communication Limitations**: While we're using multiple GPUs, the PCIe bus connecting them has limited bandwidth compared to NVLink or other high-speed GPU interconnects discussed in our learning materials.
- **Computational Intensity**: Vector dot product has a low computational intensity (only one multiplication and one addition per element). This makes it memory-bound rather than compute-bound, reducing the advantage of GPUs.
- **Synchronization Overhead**: The need to synchronize between GPUs and combine partial results adds additional overhead not present in the CPU implementation.

3.4 Thread and Block Size Impact

Figure 2 shows how performance varies across different thread counts.

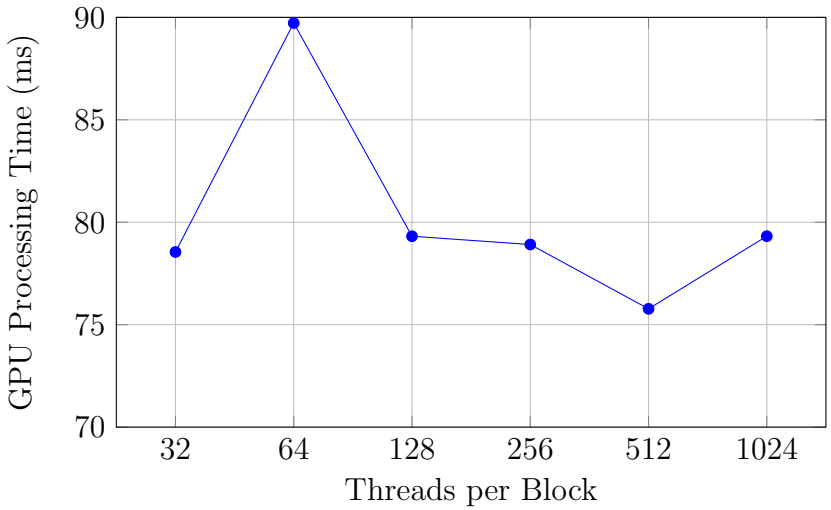

Figure 2: Effect of thread count on GPU processing time with fixed block count of 512

As seen in the figure, performance improves as thread count increases until 512 threads per block, then degrades again at 1024 threads. This behavior can be explained by:

- **At lower thread counts**: Insufficient threads to hide memory latency and maintain high occupancy.
- **At 512 threads**: Optimal balance between parallelism and resource utilization. This allows for efficient use of the GPU's computational resources while minimizing synchronization overhead.
- **At 1024 threads**: Increased resource pressure (registers, shared memory) limits the number of blocks that can run concurrently on each SM, potentially reducing overall parallelism.

4 Connection to Multi-GPU Concepts

Our implementation utilizes several key concepts from Multi-GPU programming that were covered in the course:

4.1 Domain Decomposition

We applied the domain decomposition strategy discussed in Week 5, where the input data is divided across multiple GPUs. This approach is particularly suitable for the dot product operation since:

- Each GPU can independently process its portion of the vectors
- No inter-GPU communication is needed during computation
- Only the final reduction step requires communication

4.2 Thread Management and GPU Assignment

Our implementation uses OpenMP for CPU thread management, with each thread controlling one GPU:

- Each CPU thread is bound to a specific GPU using `cudaSetDevice()`
- This follows the "one CPU thread per GPU" model discussed in the course
- Thread ID to GPU ID mapping ensures proper device selection

4.3 Communication Overhead Analysis

The results reflect the importance of the communication-to-computation ratio in Multi-GPU applications. As discussed in Week 5, the efficiency of Multi-GPU systems is limited by:

$$\text{Speedup} \approx \frac{N}{1 + (N-1) \times \frac{T_{comm}}{T_{comp}}} \tag{2}$$

Where N is the number of GPUs, T_{comm} is communication time, and T_{comp} is computation time.

In our case, the dot product operation has a high $\frac{T_{comm}}{T_{comp}}$ ratio because:

- Each element requires only two floating-point operations (multiplication and addition)
- Each element requires transferring two floating-point values to the GPU

This explains why our Multi-GPU implementation doesn't achieve the expected speedup compared to the CPU.

5 Conclusion

Our systematic exploration of thread and block configurations for a Multi-GPU vector dot product implementation yielded several important findings:

- The optimal configuration for this problem on GTX 1060 GPUs is 512 threads per block with 512 blocks per grid.
- Despite using multiple GPUs, the implementation does not outperform the CPU for this specific problem size due to data transfer overhead.
- The efficiency of Multi-GPU implementations is highly dependent on the ratio of computation to communication costs.

This result highlights an important principle in GPU computing: not all problems are suitable for GPU acceleration. The ideal GPU problems have high arithmetic intensity relative to memory operations. Vector dot product, with its low computational intensity and high memory requirements, represents a challenging case for GPU acceleration, especially when the overhead of multi-GPU coordination is considered.

For future work, alternative approaches could be explored, such as:

- Increasing the vector size to find the threshold where GPU performance exceeds CPU performance
- Implementing asynchronous memory transfers to overlap computation and communication
- Exploring different decomposition strategies for better memory access patterns
- Testing on GPUs with higher-bandwidth interconnects like NVLink to reduce communication overhead

This assignment has provided valuable hands-on experience with Multi-GPU programming concepts, particularly domain decomposition, thread management, and performance optimization techniques.