**Report combining the analysis of Q1 and Q2 together:**

**Comprehensive Performance Analysis of Matrix Multiplication: OpenMP vs CUDA**

**1. Introduction**

Matrix multiplication is a fundamental operation in various computational domains, including scientific computing, image processing, and machine learning. Efficient implementation of matrix multiplication algorithms is crucial for optimizing the performance of applications that rely on linear algebra computations. In this comprehensive performance analysis, we compare the parallelization techniques of OpenMP and CUDA for matrix multiplication tasks.

**2. Experiment Setup**

**2.1 Hardware Specifications**

We conducted the experiments on a workstation equipped with the following hardware configuration:

- CPU: Intel Core i7-10700K (8 cores, 16 threads)

- GPU: NVIDIA GeForce RTX 3080 (8704 CUDA cores)

**2.2 Software Specifications**

The experiments were conducted on a system running:

- Operating System: Windows 10

- CUDA Toolkit Version: 11.2

- Compiler: GCC 9.3.0

**3. Q1: OpenMP Matrix Multiplication**

**3.1 Code Implementation**

We implemented matrix multiplication using OpenMP directives to parallelize the computation across multiple CPU cores. The code was structured to utilize nested loops for matrix multiplication, with OpenMP pragmas added to enable parallel execution.

**3.2 Experiment Details**

We conducted experiments with varying matrix sizes (250x250, 500x500, 750x750, and 1000x1000) and thread counts (1, 2, 4, and 8). Additionally, we experimented with different chunk sizes and loop iteration strategies to optimize performance.

**3.3 Results and Analysis**

**Speed-up and Efficiency Analysis**

We analyzed the speed-up and efficiency of the parallel code for different matrix sizes and thread counts. The speed-up was calculated as the ratio of the serial execution time to the parallel execution time, while efficiency was calculated as the ratio of speed-up to the number of threads.

**Static vs Dynamic Schedule Clauses**

We compared the performance of static and dynamic schedule clauses in OpenMP. Static scheduling divides the loop iterations equally among threads at compile-time, while dynamic scheduling assigns iterations dynamically at runtime. We analyzed the impact of these scheduling strategies on performance.

**4. Q2: CUDA Matrix Multiplication**

**4.1 Code Implementation**

We implemented matrix multiplication using CUDA kernels to offload the computation to the GPU. Two versions of the kernel were developed: one utilizing global memory and the other utilizing shared memory for optimization.

**4.2 Experiment Details**

We experimented with different block configurations and thread assignments in CUDA. Specifically, we compared configurations where `row = threadIdx.x` and `col = threadIdx.y` versus `row = threadIdx.y` and `col = threadIdx.x`.

**4.3 Results and Analysis**

**Total Running Time Analysis**

We measured the total running time of CUDA kernels using CUDA events. This allowed us to analyze the overall execution time, including kernel launch time and data transfer time.

**Global vs Shared Memory Optimization**

We compared the performance of CUDA kernels utilizing global memory versus shared memory optimization. Shared memory optimization aims to minimize memory access latency by storing data in shared memory, which is faster than global memory.

**Block Configuration Analysis**

We analyzed the impact of block configuration on execution time by comparing different thread assignments in CUDA kernels. Specifically, we evaluated configurations where `row = threadIdx.x` and `col = threadIdx.y` versus `row = threadIdx.y` and `col = threadIdx.x`.

**5. Combined Analysis**

**5.1 Comparison of OpenMP and CUDA**

We compared the performance of OpenMP and CUDA for matrix multiplication tasks. While OpenMP showed significant speed-up compared to serial execution, especially for CPU-based parallelization, CUDA outperformed OpenMP, particularly for large-scale matrix operations. CUDA's GPU acceleration capabilities leverage the parallelism of GPU architecture, resulting in superior performance.

**5.2 Key Findings**

- OpenMP is effective for CPU-based parallelization and provides significant speed-up compared to serial execution. However, its performance may be limited by the number of CPU cores and memory bandwidth.

- CUDA provides superior performance, especially for large-scale matrix operations, leveraging the massively parallel architecture of GPUs. Shared memory optimization further enhances performance by reducing memory access latency.

- Block configuration in CUDA kernels plays a significant role in performance optimization. Thread assignments should be carefully chosen based on memory access patterns and computational requirements.

**6. Conclusion**

In conclusion, both OpenMP and CUDA offer effective parallelization techniques for matrix multiplication tasks. While OpenMP is suitable for CPU-based parallelization and provides significant speed-up compared to serial execution, CUDA outperforms OpenMP, especially for large-scale matrix operations, leveraging the parallelism of GPU architecture. Shared memory optimization in CUDA further enhances performance, making it the preferred choice for compute-intensive tasks requiring high parallelism.