Project 2

COMPILATION AND EXECUTION (IN CLUSTER)

CMake Configuration: First, configure the project using CMake to generate the necessary build files. Navigate to the project directory (../project1/build/)and run the following command to create the build files:

bash **cmake ..**

This step sets up the project's build environment and prepares it for compilation.

Compilation: Next, use the **make** command to compile the project. You can specify the number of parallel jobs to speed up compilation (e.g., **-j4** for four parallel jobs):

bash **make -j4**

This command compiles the source code, linking all necessary libraries and generating the executable binary.

Cluster Submission Script:
To run our program on the cluster, use a submission script. In this example, use the **sbatch** command to submit a job to the cluster. It specifies job parameters such as the number of CPU cores, memory requirements, and the name of the executable to run.

shell **sh src/scripts/sbatch_PartB.sh**

The submission script is responsible for launching our program on the cluster nodes with the specified configurations.

PARALLEL PROGRAMMING MODELS

Memory Locality:

- **Blocking (Tiling)**:
- The matrices **matrix1** and **matrix2** are divided into smaller blocks, often referred to as tiles. In this code, **BLOCK_SIZE** is set to 32, which means it processes the matrices in 32x32 tiles.
- The blocking technique ensures that the computation is done within a small block of the matrices, promoting cache locality and reducing cache misses.
- The code uses **std::min** to ensure that the loops do not go beyond the boundaries of the matrices.

- **Change the order of the triple loop:**
- Changing loop order improves data locality, minimizing cache misses, and accelerating matrix multiplication significantly.

Data-Level Parallelism:

- **Blocking (Tiling)**:
- The code dynamically adjusts the BLOCK_SIZE based on the matrix dimensions. If the matrix dimensions are smaller than 64x64, it uses a smaller BLOCK_SIZE (8x8). For larger matrices, it uses the original BLOCK_SIZE of 64x64.
- **Change the order of the triple loop:**
- **Use of SIMD intrinsic**
- The code uses SIMD instructions to parallelize the matrix multiplication. SIMD allows a single instruction to operate on multiple data elements simultaneously, operating on 256-bit wide data registers. The code optimizes memory access by loading data into SIMD registers, performing calculations, and then storing results back to memory.

Thread-Level Parallelism:

- **Blocking (Tiling)**:
- **Change the order of the triple loop:**
- **Use of SIMD intrinsic**
- **Multithreading with OpenMP**
- The code uses OpenMP directives to parallelize the outer loops. The "omp_set_num_threads" function sets the number of threads for parallel execution. This divides the matrix multiplication task among multiple threads, taking advantage of multi-core processors.

- OpenMP directives parallelize the computation within each block across multiple threads. This means each thread processes a portion of the matrix, further improving the computation speed.

Process-Level Parallelism:

- **Blocking (Tiling)**:
- **Change the order of the triple loop:**
- **Use of SIMD intrinsic**
- **Multithreading with OpenMP**
- **MPI processes**
- **Dynamic Load Balancing**: Rows of the matrix are distributed among MPI processes, and each process is responsible for a specific range of rows. Dynamic load balancing is achieved by adjusting the range based on the number of processes and the matrix dimensions.
- **Message Passing**: MPI processes communicate by sending and receiving data using the MPI communication routines. After computing their portion of the matrix multiplication, processes exchange results with the master process (rank 0).
- **Aggregation of Results**: The master process collects partial results from all other processes, aggregating them into the final result matrix.

CUDA Kernel for Matrix Multiplication:

- The matrix multiplication computation is offloaded to the GPU using a CUDA kernel named **matrixMultiply**.
- The kernel is executed in parallel on the GPU with each thread handling a specific element of the result matrix **C**.
- **Shared Memory Usage**:
- Shared memory is used to store submatrices (**subTileA** and **subTileB**) that are used for each tile of the result matrix. Shared memory is much faster than global memory, and this optimization reduces memory latency.
- **Loop Tiling**:

• The matrix multiplication is optimized using loop tiling. The computation is divided into smaller tiles, and shared memory is used for each tile, improving data locality and reducing global memory accesses.

Grid and Block Configuration:

• The dimensions of the thread blocks (**blockDim**) and the grid of thread blocks (**gridDim**) are configured based on the size of the matrices to ensure efficient parallel execution. This is essential for optimal GPU performance.

EXPERIMENT RESULTS

Methods	Matrices 1024*1024	Matrices 2048*2048
Naive	7998 ms	99693 ms
Memory Locality	4329 ms	36972 ms
SIMD + Memory Locality	390 ms	3203 ms
OpenMP + SIMD + Memory Locality (32 threads)	67 ms	334 ms
MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	62 ms	371 ms
CUDA	430 ms	5802 ms

EXPERIMENT RESULTS

SIMD + Memory Locality Matrix Multiplication (Optimized with -O3)

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 370 milliseconds

OpenMP + SIMD + Memory Locality Matrix Multiplication (Optimized with -O3)

Number of cores: 1

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 413 milliseconds

Number of cores: 2

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 412 milliseconds

Number of cores: 4

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 219 milliseconds

Number of cores: 8

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 132 milliseconds

Number of cores: 16

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 94 milliseconds

Number of cores: 32

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 100 milliseconds

MPI + OpenMP + SIMD + Memory Locality Matrix Multiplication (Optimized with -O3)

Number of Processes: 1, Number of Threads: 32

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 73 milliseconds

Number of Processes: 2, Number of Threads: 16

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 87 milliseconds

Number of Processes: 4, Number of Threads: 8

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 69 milliseconds

Number of Processes: 8, Number of Threads: 4

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 68 milliseconds

Number of Processes: 16, Number of Threads: 2

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 66 milliseconds

Number of Processes: 32, Number of Threads: 1

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 68 milliseconds

CUDA

Output file to: /nfsmnt/123100001/CSC4005-2023Fall/project2/src/../build/result.txt

Multiplication Complete!

Execution Time: 717 milliseconds

[perf record: Woken up 4 times to write data]

[perf record: Captured and wrote 0.558 MB ./perf_files/cuda.data (5743 samples)]

Performance counter stats for 'sh src/sbach.sh':

256,753,483	instructions:u	# 1.12 insn per cycle	(73.37%)
228,956,836	cpu-cycles:u		(72.37%)
2,442,770	cache-references:u		(73.25%)
989,904	cache-misses:u	# 40.524 % of all cache refs	(73.14%)
51,379,965	branch-instructions:u		(74.07%)
1,438,017	branch-misses:u	# 2.80% of all branches	(59.06%)
5,280,237	bus-cycles:u		(58.81%)

13.205533712 seconds time elapsed

0.247815000 seconds user

0.314141000 seconds sys