CSC4005 Project 1 - With CUDA IMPLEMENT

ID:120090712

CPU Part

How to execute the program:

Depress the zip file, the depressed folder is "120090712". Upload into the cluster. Open 120090712/src/sbatch.sh and change the following settings:

```
CURRENT_DIR="/path/to/120090712/src"
matrixA='matrix5.txt'
matrixB='matrix6.txt'
```

Then, in the cluster, use following commands to complie and subbmit the tasks:

```
cd /path/to/120090712
mkdir build && cd build
cmake ..
make -j4
cd ..
sbatch ./src/sbatch.sh
```

The location of the result will be "120090712/build/result.txt"

How does each parallel programming model do computation in parallel:

SIMD

```
SIMD allows multiple data elements to be processed simultaneously using a single
instruction. In SIMD programming, the same operation is performed on multiple data
elements in parallel. This is achieved by using special SIMD data types and
instructions. For example, in the case of matrix multiplication, SIMD instructions
are used to perform element-wise multiplication and addition on vectors, enabling
parallel computation of multiple elements at once.
```

OPENMP

```
OpenMP is a shared-memory parallel programming model. In OpenMP, parallelism is
achieved by dividing the computation among multiple threads that can run
```

```
concurrently on different cores or processors. The programmer uses directives,
such as "#pragma omp parallel for," to specify which parts of the code should be
executed in parallel. The work is automatically divided among the threads, and
each thread executes a portion of the computation independently. In the case of
matrix multiplication, the outer loop is parallelized using OpenMP, allowing
multiple threads to compute different rows of the result matrix simultaneously.
```

MPI

```
MPI is a message-passing parallel programming model used for distributed memory
systems. In MPI, the computation is divided among processes, each running on a
separate node or processor. Processes communicate with each other by exchanging
messages. The programmer explicitly specifies the message passing operations, such
as sending and receiving data between processes. In the case of matrix
multiplication, the computation is divided among different processes, with each
process responsible for a subset of rows. After each process completes its
computation, the results are exchanged using MPI communication operations, such as
MPI_Isend and MPI_Irecv, to gather the final result in the master process.
```

Kinds of optimizations to speed up my parallel program, and how does them work

COMPELLING

```
I use O3 instead of O3, O3 aims to improve the efficiency of parallel computing
compared to O2. It achieves this by applying advanced optimization techniques
specifically targeted at parallel code. These optimizations include automatic
vectorization, loop unrolling, and software pipelining. Loop unrolling is a
technique where the loop iterations are reduced by executing multiple iterations
in a single loop iteration. This reduces the overhead of loop control and improves
the efficiency of parallel execution.

Software pipelining is a technique that overlaps the execution of loop iterations
to exploit instruction-level parallelism. It reorders and schedules instructions
in such a way that the processor's pipeline is fully utilized, minimizing pipeline
stalls and improving overall parallel execution efficiency.

By applying these optimizations, O3 can effectively parallelize code, distribute
workload across multiple cores or processors, and exploit the capabilities of
modern hardware. This results in improved parallel computing efficiency compared
to O2, which does not have as extensive optimization techniques targeting
parallelism.

Basically, for some CPU parallel code, O3 can improve its efficiency by about 15%
than O2.
```

LOCALITY

1. The number of operands of matrix multiplication is O (n^3), but due to the discontinuity of data in different rows, it will increase the number of cache misses, and this will waste a lot of time. My code has adjusted the order of the loops. After I made the adjustments, my matrix multiplication will continuously take data from the same row during each multiplication, thereby reducing the number of cache misses.

2. In "naive" code, due to the large size of the matrix itself, it takes a lot of time to use Matrix [i] [j] to retrieve and store values. So in my code, I chose to use pointers to represent the first data of each row, and then use pointer calculations to access the data I want, saving a lot of time.

SIMD

My code also uses SIMD (Single Instruction, Multiple Data) optimization technique to improve the performance of matrix multiplication. SIMD allows multiple data elements to be processed simultaneously using a single instruction.

In this code, the "__m256i" data type and related SIMD instructions are used.

The "_mm256_set1_epi32" function sets all elements of a 256-bit vector to the same value.
The "_mm256_loadu_si256" function loads 256 bits (8 integers) from memory into a 256-bit vector.
The "_mm256_mullo_epi32" function performs element-wise multiplication of two 256-bit vectors, resulting in a vector of the products.
The "_mm256_add_epi32" function performs element-wise addition of two 256-bit vectors.
The "_mm256_storeu_si256" function stores a 256-bit vector into memory.
The code performs matrix multiplication in a loop using SIMD instructions. It loads elements from the matrices into SIMD vectors, performs multiplication and addition operations on the vectors, and stores the results back into memory.

By using SIMD, the code can process multiple elements at once, reducing the number of instructions needed and improving the overall performance of the matrix multiplication algorithm.

OPENMP

My code uses OpenMP (Open Multi-Processing) optimization technique to improve the performance of matrix multiplication by parallelizing the computation across multiple threads.

Since each loop in my loop is not associated and there are no potential duplicate read and write operations, I can directly optimize it using OPENMP.

In my code, the "#pragma omp parallel for" directive is used to parallelize the outermost loop, which iterates over the rows of the result matrix. This directive instructs the compiler to distribute the iterations of the loop across multiple threads, allowing for concurrent execution of the loop iterations.

By using OpenMP, the code can leverage the available computational resources by dividing the work among multiple threads. Each thread performs matrix multiplication for a subset of rows independently and updates the corresponding portion of the result matrix. This parallelization reduces the overall execution time by utilizing multiple cores or processors simultaneously.

With the help of OpenMP, my code achieves parallel execution of matrix multiplication, effectively utilizing the available resources and reducing the overall computation time.

MPI

My code uses MPI (Message Passing Interface) optimization technique to improve the performance of matrix multiplication by distributing the computation across multiple processes in a parallel and distributed manner.

In my code, MPI is used to divide the computation of matrix multiplication among multiple processes. The code starts by initializing MPI and obtaining the number of processes and the rank of each process. The input matrices are loaded by the master process.

The matrix multiplication is divided into subsets of rows, and each process performs matrix multiplication for a specific range of rows. The ranges are determined using the "cuts" array. The master process assigns different ranges of rows to different processes by using MPI_Isend to send the results back to the master process.

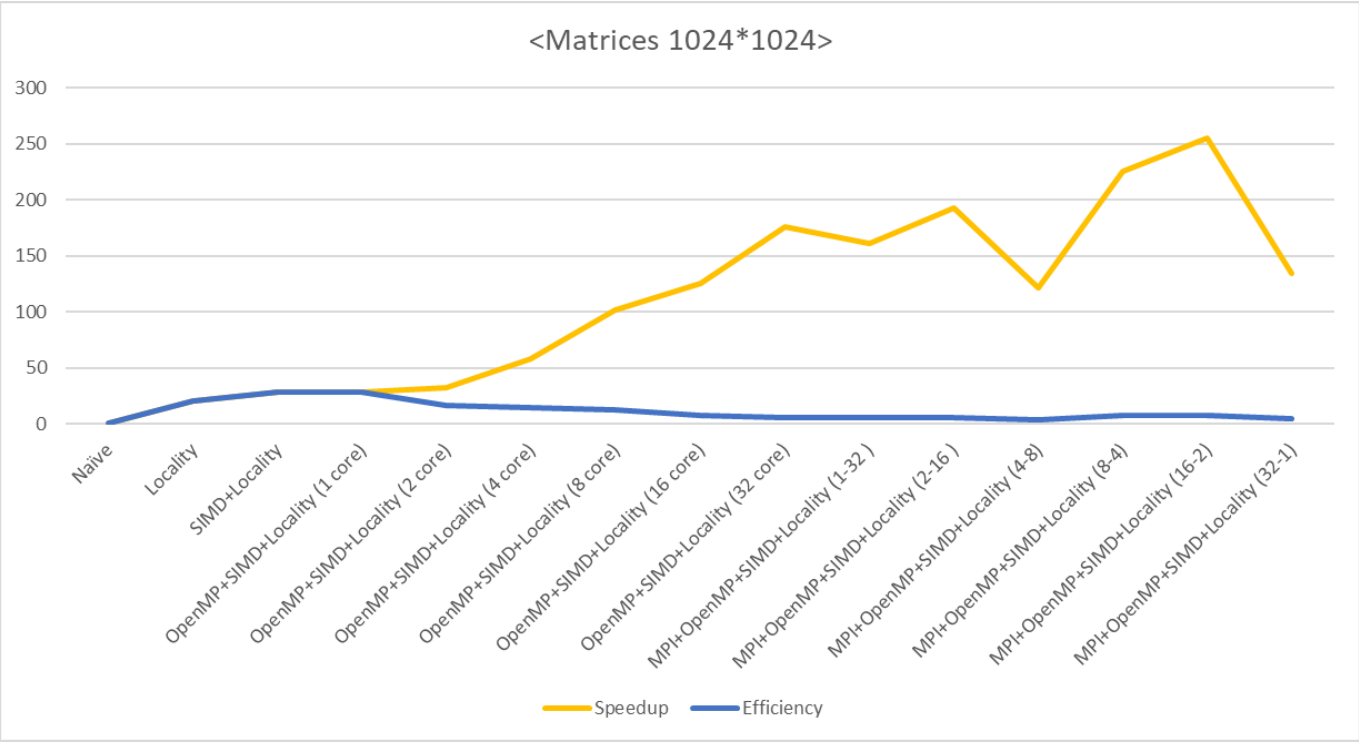
After the matrix multiplication is complete, the master process waits for the results from the other processes using MPI_Irecv. Once all the results are received, the master process saves the result to the output file.

By using MPI, the computation of matrix multiplication is distributed among multiple processes, allowing for parallel and distributed execution. This enables the utilization of multiple computational resources and reduces the overall computation time.

Experiment results and some numerical analysis

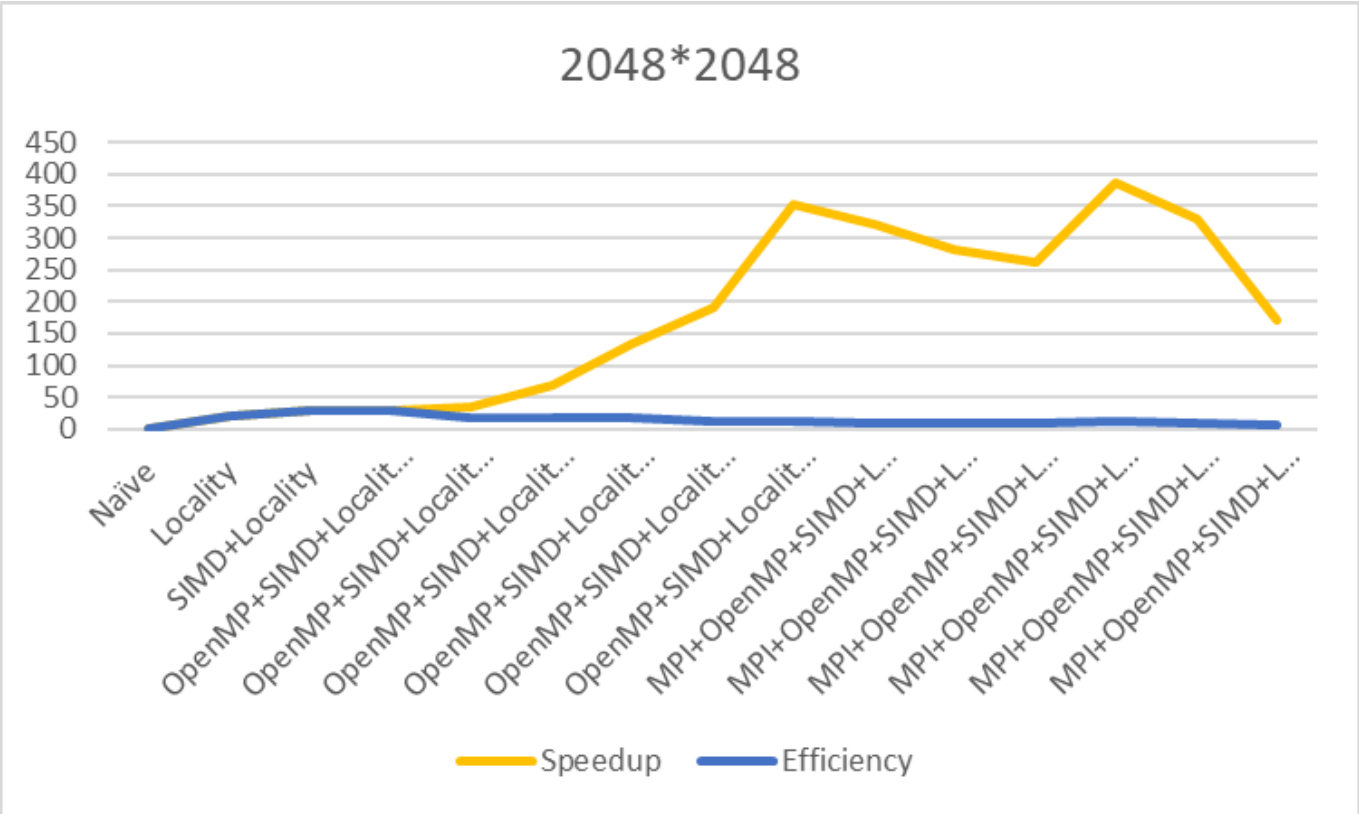
Matrix 1024*1024:

<Matrices 1024*1024>	Number of cores	Number of Threads	Time(ms)	Speedup	Efficiency
Naïve	1	1	7893	1	1
Locality	1	1	392	20.1377551	20.1377551
SIMD+Locality	1	1	275	28.70545455	28.70545455
OpenMP+SIMD+Locality	1	1	276	28.60144928	28.60144928
OpenMP+SIMD+Locality	2	1	243	32.48559671	16.24279835
OpenMP+SIMD+Locality	4	1	137	57.62043796	14.40510949
OpenMP+SIMD+Locality	8	1	78	101.2051282	12.65064103
OpenMP+SIMD+Locality	16	1	63	125.3015873	7.831349206
OpenMP+SIMD+Locality	32	1	45	175.4222222	5.481944444
MPI+OpenMP+SIMD+Locality	1	32	49	161.1020408	5.034438776
MPI+OpenMP+SIMD+Locality	2	16	41	192.5365854	6.016768293
MPI+OpenMP+SIMD+Locality	4	8	65	121.4461538	3.795192308
MPI+OpenMP+SIMD+Locality	8	4	35	225.5428571	7.048214286
MPI+OpenMP+SIMD+Locality	16	2	31	254.6451613	7.95766129
MPI+OpenMP+SIMD+Locality	32	1	59	133.7966102	4.181144068



Matrix 2048*2048:

<Matrices 2048*2048>	Number of cores	Number of Threads	Time(ms)	Speedup	Efficiency
Naïve	1	1	77462	1	1
Locality	1	1	3573	21.67982088	21.67982088
SIMD+Locality	1	1	2725	28.42642202	28.42642202
OpenMP+SIMD+Locality	1	1	2749	28.17824664	28.17824664
OpenMP+SIMD+Locality	2	1	2149	36.04560261	18.0228013
OpenMP+SIMD+Locality	4	1	1143	67.77077865	16.94269466
OpenMP+SIMD+Locality	8	1	583	132.8679245	16.60849057
OpenMP+SIMD+Locality	16	1	406	190.7931034	11.92456897
OpenMP+SIMD+Locality	32	1	220	352.1	11.003125
MPI+OpenMP+SIMD+Locality	1	32	241	321.4190871	10.04434647
MPI+OpenMP+SIMD+Locality	2	16	276	280.6594203	8.770606884
MPI+OpenMP+SIMD+Locality	4	8	295	262.5830508	8.205720339
MPI+OpenMP+SIMD+Locality	8	4	201	385.3830846	12.04322139
MPI+OpenMP+SIMD+Locality	16	2	235	329.6255319	10.30079787
MPI+OpenMP+SIMD+Locality	32	1	454	170.6211454	5.331910793



Efficiency and Speedup:

```
Efficiency and speedup metrics provide insight into how effectively resources are
utilized in parallel computation. In both datasets, the efficiency and speedup
values vary with the number of cores and threads. Typically, The speedup values
increases as the number of total threads inreases

While efficiency generally decreases with an increase in cores and threads, some
variations can be observed. For example, in the first dataset, the efficiency
decreases as the number of threads increases from 2 to 32, suggesting diminishing
returns from parallelization. However, when using MPI+OpenMP+SIMD+Locality with 2
cores and 16 threads, the efficiency increases compared to 32 threads, indicating
better resource utilization.
```

Analysis of Trends and Phenomena:

```
Change in Performance with Different Number of Cores and Threads:
In both datasets (1024*1024 and 2048*2048), the performance of the matrix
multiplication varies with different combinations of cores threads, despite the
same total number of MPI processes. This indicates that the distribution of
workload and the synchronization overhead impact the overall performance.

For example, in the first dataset, when using OpenMP+SIMD+Locality with 4 cores
and 1 thread, the execution time is reduced compared to 2 cores and 1 thread.
However, when increasing the number of cores to 8, the execution time further
decreases significantly. This suggests that distributing the workload among more
cores can effectively improve performance, up to a certain limit.

Similarly, in the second dataset, the execution time decreases as the number of
cores and threads increases until a certain point. However, when using a higher
number of threads with a fixed number of cores, the execution time may increase
due to increased thread synchronization overhead.
```

What have I found from the experiment results

Impact of SIMD and Locality Optimization:

```
The SIMD+Locality optimization technique aims to exploit data parallelism and
improve memory access patterns to enhance performance. In both datasets, the
SIMD+Locality method consistently demonstrates superior performance compared to
the Naïve and Locality methods. This indicates that utilizing SIMD instructions
and optimizing memory access patterns can significantly speed up the matrix
multiplication process.
```

Possible Reasons for Variations:

```
Several factors influence the observed variations in performance:

I. Workload Distribution: The workload distribution across cores/threads affects
the load balance and overall performance. Unequal distribution can result in idle
resources and suboptimal performance.

II. Thread Synchronization Overhead: Increasing the number of threads can
introduce additional overhead due to increased synchronization and communication
between threads.

III. Memory Access Patterns: Optimizing memory access patterns can enhance data
locality and reduce cache misses, leading to improved performance.

IV. Limits of Parallelization: Increasing the number of cores/threads beyond a
certain point may not always yield proportional performance gains due to
contention for resources and increased overhead.
```

Profiling Results & Analysis with `perf`

Part of useful result from the 'perf report' analysis:

cpu-cycles

```
      Count    Children      Self      Command   Shared Object        Program
23320351486      74.17%    74.16%        naive           naive          naive
 1675551637      67.25%    67.16%     locality        locality       locality
 1235770390      55.42%    55.42%         simd            simd           simd
 1234387701      55.21%    55.21%       openmp          openmp  openmp(1 pro)
 1781122790      67.49%    67.49%       openmp          openmp  openmp(2 pro)
...
 2060827124      54.44%    54.39%       openmp          openmp openmp(16 pro)
 2963926888      37.69%    37.67%       openmp          openmp openmp(32 pro)
```

cache-misses

```
      Count    Children      Self      Command   Shared Object        Program
     188974      36.10%    36.09%        naive           naive          naive
     134803      46.99%    46.99%     locality        locality       locality
     159888      50.02%    50.02%         simd            simd           simd
     154243      46.95%    46.95%       openmp          openmp  openmp(1 pro)
     152176      52.79%    52.79%       openmp          openmp  openmp(2 pro)
     ...
     357001      74.05%    74.05%       openmp          openmp openmp(16 pro)
     359017      74.39%    74.39%       openmp          openmp openmp(32 pro)
```

page-faults

```
Count                  Program
10113                    naive
 3717                  locality
 4172                      simd
 4330            openmp(1 pro)
 4407            openmp(2 pro)
...
 3979           openmp(16 pro)
 4086           openmp(32 pro)
```

For MPI:

1 core 32 threads:

```
2,871,905,846      cpu-cycles:u
      412,646      cache-misses:u
        5,106      page-faults:u
```

2 core 16 threads:

```
2,081,034,676      cpu-cycles:u
      501,413      cache-misses:u
        5,341      page-faults:u
```

4 core 8 threads:

```
1,381,536,883      cpu-cycles:u
      203,406      cache-misses:u
        5,281      page-faults:u
```

8 core 4 threads:

```
1,087,139,873      cpu-cycles:u
      599,779      cache-misses:u
        4,857      page-faults:u
```

16 core 2 threads:

```
     1,070,769,935      cpu-cycles:u
         1,236,246      cache-misses:u
             4,783      page-faults:u
```

32 core 1 threads:

```
     1,491,052,399      cpu-cycles:u
         1,654,722      cache-misses:u
             4,819      page-faults:u
```

Analysis

```
Compared to linear computing, as the number of processes increases, the CPU
cycles, cache misses, and page faults of each process generally show a downward
trend. However, I have noticed that when the number of processes is high, these
data fluctuates back and forth, and the proportion of resources used for matrix
computing functions may decrease (thread overhead increases)
```

GPU Part - CUDA

How to design:

```
The program takes three command-line arguments: the paths to two input matrices
and the path to the output matrix. It then loads the two input matrices from the
specified files using the Matrix class.

Before performing the matrix multiplication, the program verifies if the
dimensions of the two input matrices are compatible for multiplication. If not, an
error message is printed and the program exits.

Next, the program allocates memory on the GPU for the input matrices and the
result matrix using cudaMalloc. It also allocates memory on the CPU for temporary
arrays to flatten the input matrices and store the result matrix.

The input matrices are then flattened and copied from the CPU to the GPU using
cudaMemcpy.

The program then calculates the grid and block sizes for launching the kernel
function. The kernel function, matrix_multiply_kernel, is responsible for
performing the matrix multiplication. Each thread in the grid calculates a single
element of the result matrix using a nested loop. The result matrix is stored in
the global memory of the GPU.
```

```
After the kernel function completes, the result matrix is copied from the GPU to
the CPU using cudaMemcpy.

Finally, the program frees the memory allocated on the GPU, unflattens the result
matrix, saves it to the specified output file using the saveToFile function of the
Matrix class, and prints the execution time.
```

Performance:

```
As the TA has pointed out, the time for the first "cudaMalloc" can be excluded
because it is too heavy.
For 1024 * 1024, the cost time is about 12ms.
For 2048 * 2048, the cost time is about 70ms.

Basically, the time are used for data mallocing and passing.
```

Some findings:

```
1. I have found that kernel functions are often less important in reducing time in
CUDA programming, and how data is uploaded to threads often becomes a bottleneck.

2. A strange thing is that the first cudaMalloc will take a lot of time (about
900ms for 1024 * 1024), but the second cudaMalloc will not, even though they
accept the same size of data. This makes me very confused.
```