Task 1 - CPU calculation (Only in the CPU)

1. **Command-line options:**
   - -n : set number of rows (default 10).
   - -m : set number of columns (default 10).
   - -r: use a “random” seed based on the current time instead of a fixed seed.
   - -t: print timings for each of the three steps.
2. **Compile and run with Makefile**

- **Compile** all the four tasks code using `make`

```
kuangg@cuda01:~/homework/assign01$ make
g++ -O2 -std=c++11 -o task01_matrix_sums task01_matrix_sums.cc
nvcc -O4 --use_fast_math --compiler-options -funroll-loops -arch=sm_75 -o task02_matrix_sums
task02_matrix_sums_cuda.cu
nvcc -O4 --use_fast_math --compiler-options -funroll-loops -arch=sm_75 -o task03_matrix_sums
task03_matrix_sums_cuda.cu
nvcc -O4 --use_fast_math --compiler-options -funroll-loops -arch=sm_75 -o task04_matrix_sums
task04_matrix_sums_cuda.cu
```

- **Run** `make run_task01` to see the result

```
kuangg@cuda01:~/homework/assign01$ make run_task01
./task01_matrix_sums -n 5000 -m 5000 -t
Matrix size: 5000 x 5000
Sum of row absolute values (reduced)    = 2.49969e+08
Sum of column absolute values (reduced) = 2.49969e+08

Timing (microseconds):
  rowSumAbs   : 20119
  colSumAbs   : 220871
  reduce (2x): 9
```

Short Explanation of Performance Differences

1. **Row Summation**: In C/C++, rows are stored one after the other in memory. When you add up a row, you’re going through numbers that sit right next to each other, so the CPU cache can grab them quickly. This makes row summation faster.
2. **Column Summation**: For columns, you have to jump from one row to the next because the numbers aren’t next to each other in memory. This means the cache can’t help as much, so column summation tends to be slower.

Task 2 - parallel implementation (In the CPU and GPU)

1. **Command-Line Arguments**:
   - -n : number of rows (default = 10)
   - -m : number of columns (default = 10)
   - -r: use a random seed (current microseconds) instead of a fixed seed
   - -t: show **CPU** timings
   - -gt: show **GPU** timings
2. **Run** `make run_task02`

```
kuangg@cuda01:~/homework/assign01$ make run_task02
/usr/local/cuda-12.6/bin/compute-sanitizer ./task02_matrix_sums -n 5000 -m 5000 -t -gt
========= COMPUTE-SANITIZER
Matrix size: 5000 x 5000
CPU row-reduced = 2.49969e+08, GPU row-reduced = 2.49969e+08
CPU col-reduced = 2.49969e+08, GPU col-reduced = 2.49969e+08

[CPU Timings in microseconds]
  Row Summation : 19924
  Col Summation : 232248
  Reduce (2x)   : 10

[GPU Timings in milliseconds]
  Row Summation : 13.4494
  Col Summation : 12.8075
  Reduce (2x)   : 20.7194
========= ERROR SUMMARY: 0 errors
```

Explanation of Performance Differences

1. **Row Summation (GPU)**
   We launch a kernel in which each thread (or each small group of threads) is responsible for summing one row. Because the matrix is in row-major order, this typically coalesces memory access on the GPU and results in decent performance.
2. **Column Summation (GPU)**
   We similarly launch a kernel where each thread sums one column. While column access on a row-major CPU is slow due to strided memory access, the GPU can still parallelize the operation across many threads. Thus, the GPU version is often much faster than the CPU for column summation.
3. **Reduction (GPU)**
   We implemented a naive reduction using a single kernel and a single thread (or a straightforward loop). Since we did **not** use shared memory or more advanced parallel techniques, the reduce step may not be drastically faster than the CPU version. In fact, for large vectors, the CPU's single-thread summation can be surprisingly fast, especially considering the overhead of copying data to and from the GPU.
4. **Performance Observations**
   • On the CPU, **row summation** is relatively quick (good cache locality), while **column summation** is

typically very slow (cache unfriendly).
• On the GPU, both row and column summations typically see large parallel speedups compared to the CPU. However, the naive GPU reduction may not yield significant speedup over the CPU's reduce, because the CPU's single-thread code is already efficient and the GPU approach is not optimized (no shared memory, no block-level parallelization).

5. **Conclusion**
   • The code correctly performs the same calculations on CPU and GPU, matches the results, and measures their timings side by side.
   • Even with a naive implementation (no shared memory, no atomics), we often see a good GPU speedup for row/column summations, especially with large matrices. The reduction step, however, remains an area where more advanced GPU techniques would provide better performance.

Task 3 - performance improvement

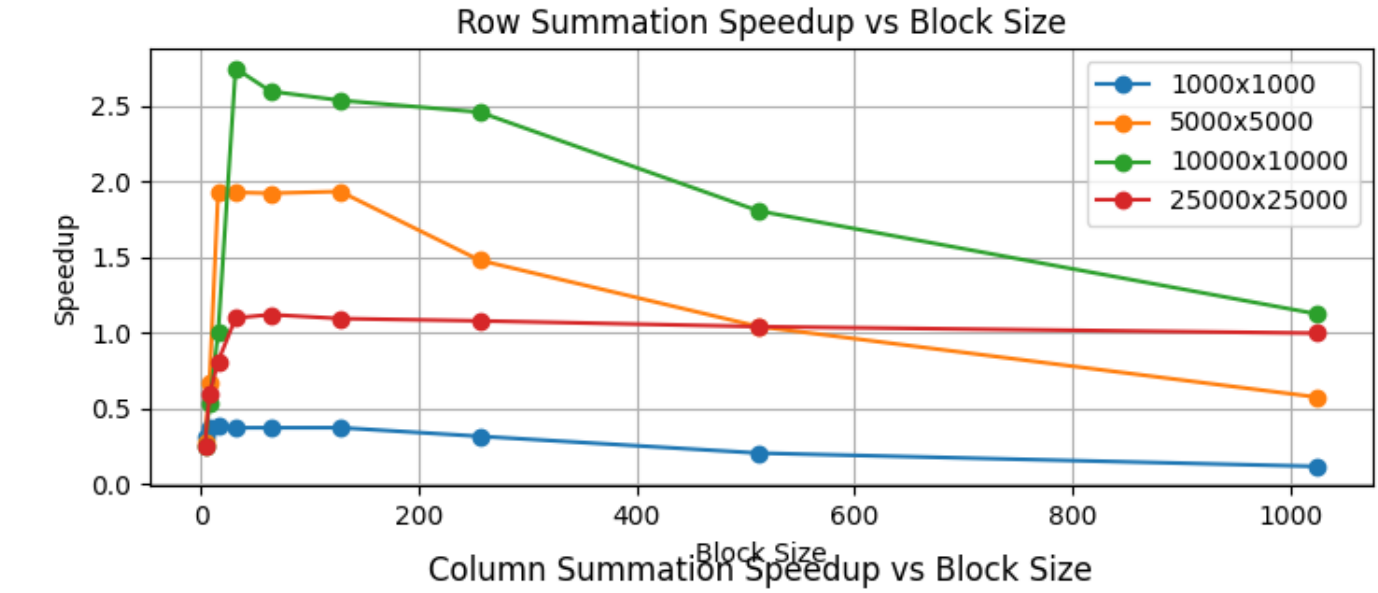
1. **Command-Line Options**:
   - -n (number of rows, default 10)
   - -m (number of columns, default 10)
   - -bs (threads per block, default 256)
   - -r (use random seed, instead of fixed seed 1234567)
   - -t (print CPU timings)
   - -gt (print GPU timings)
   - -sp (print speedups)
   - -auto (runs an automated test loop over multiple matrix sizes (1000, 5000, 10000, 25000) and multiple block sizes (4, 8, 16, 32, 64, 128, 256, 512, 1024))
   - -save (save results to task03_results.txt in CSV format)
2. **Run** `make run_task03` to get the **`task03_results.txt`** test file

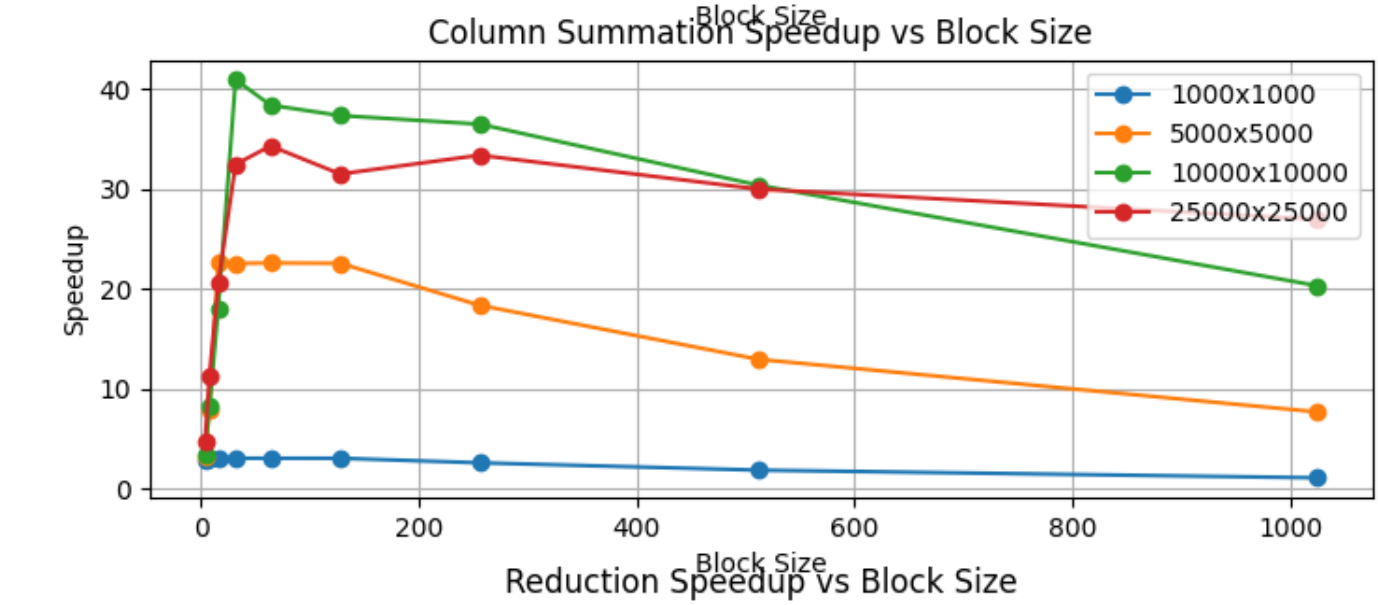
```
kuangg@cuda01:~/homework/assign01$ make run_task03
========= COMPUTE-SANITIZER
------------------------------------------------
Matrix: 1000x1000 | blockSize=4
CPU row-reduced  = 9.99985e+06, GPU row-reduced  = 9.99985e+06
CPU col-reduced  = 9.99985e+06, GPU col-reduced  = 9.99985e+06
Row reduce error: 0%, Col reduce error: 0%
[CPU Timings - microseconds]
  Row Summation : 1190
  Col Summation : 9603
  Reduce (2x)   : 3
[GPU Timings - milliseconds]
  Row Summation : 2.48269
  Col Summation : 2.22566
  Reduce (2x)   : 4.33203
[Speedups: CPU/GPU]
  Row Summation Speedup : 0.479319
```

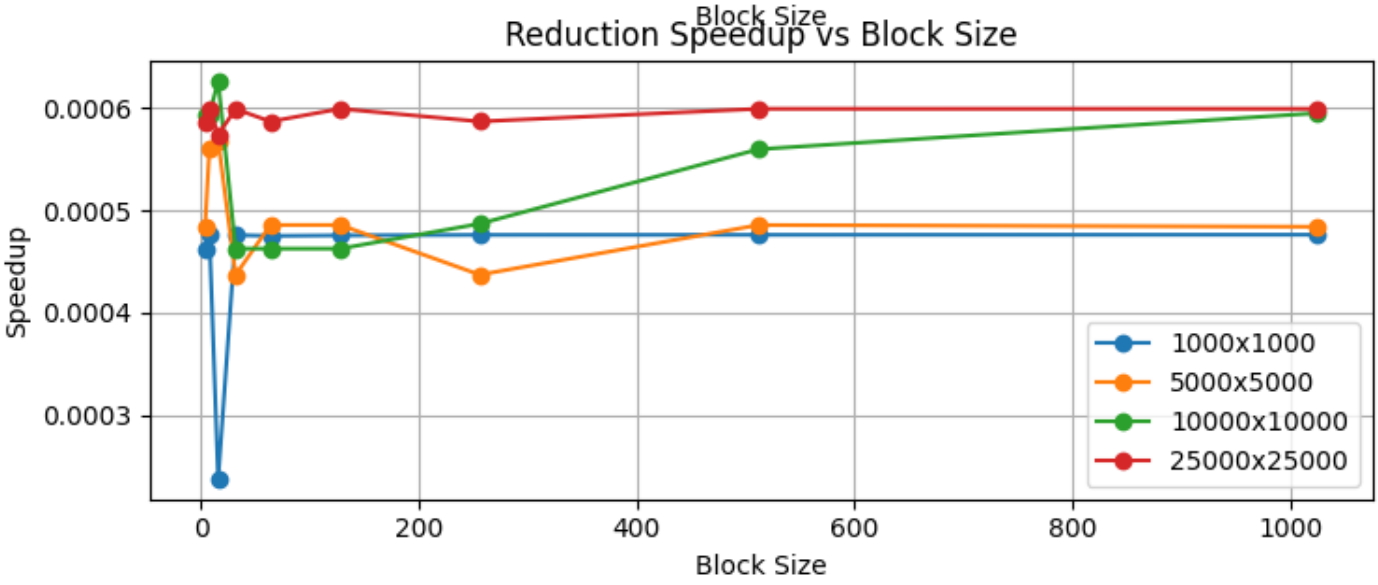
```
  Col Summation Speedup : 4.31467
  Reduce (2x) Speedup   : 0.000692516
-------------------------------------------------
```

3. **Run** `python3 plot_results.py` to draw the performance picture.（**Make sure you have python3 and matplotlib installed** ）

Performance Analysis:



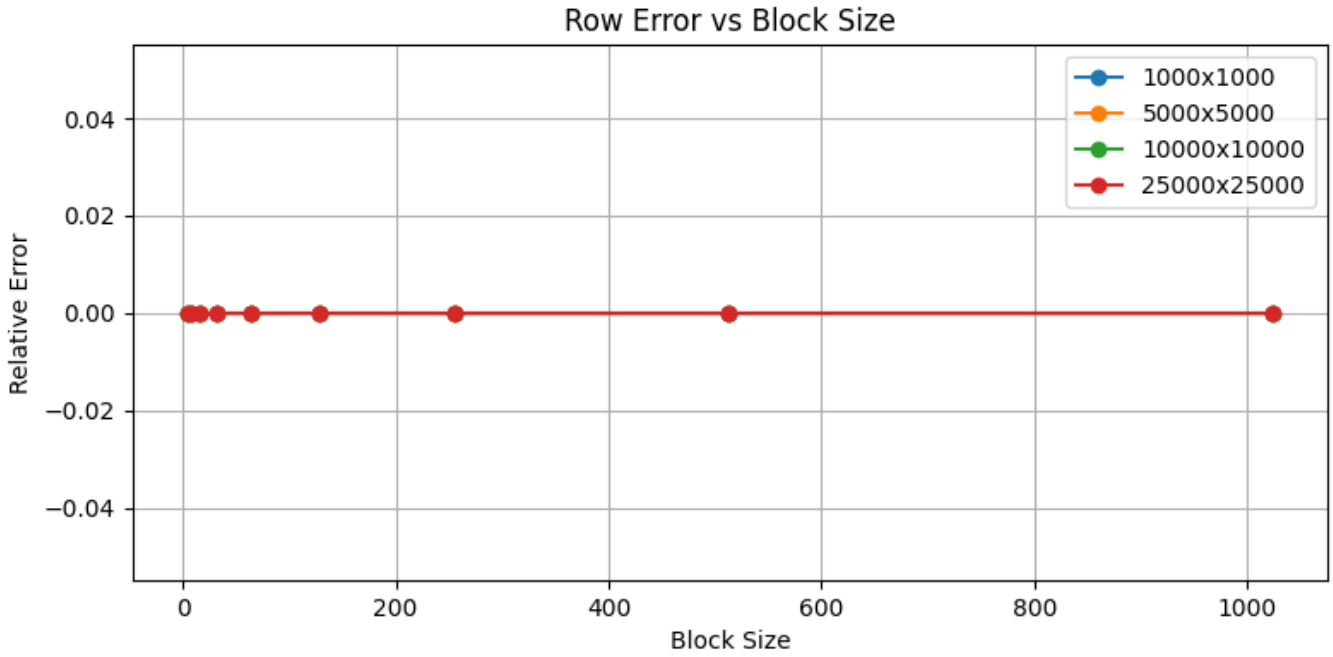


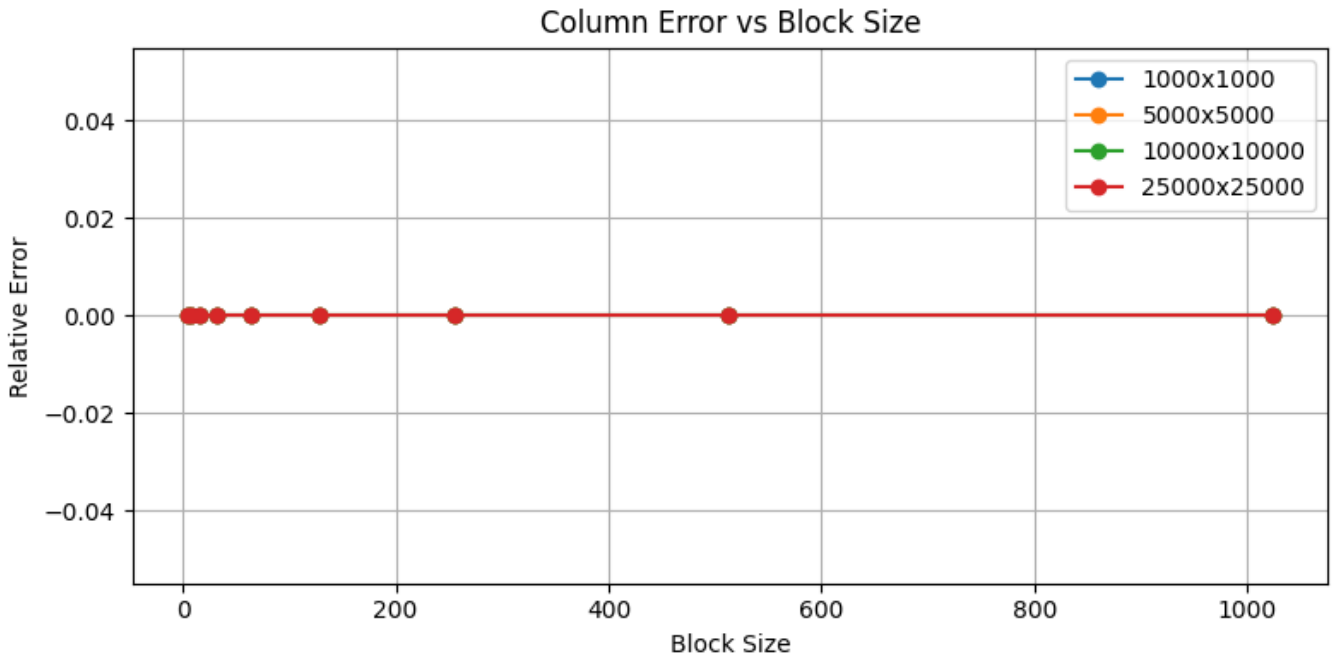


1. **Row Summation**
   The speedup is moderate (often between 1× and 4× for larger matrices) because row-major memory layout is already cache-friendly on the CPU. Even so, for large matrices (e.g., 10000 times 10000, 25000 times 25000), the GPU still outperforms the CPU if you choose a reasonable block size (often around 64–

128). For very small matrices or very large block sizes, you might see smaller speedups.

2. **Column Summation**
   This operation typically shows **much higher speedups** (up to 40× or more for larger matrices) because column access is very cache-unfriendly on the CPU in row-major layout. The GPU's parallel approach significantly reduces total computation time. As with row summation, there is usually a "sweet spot" in the middle range of block sizes.
3. **Reduction**
   The naive GPU reduction often shows **little or no speedup**, and can even be slower than the CPU. That's because we are using a single-thread or very basic approach without shared memory or block-level parallel sums. The CPU's straightforward loop is already quite fast. So, unless you implement a more advanced GPU reduction (which the assignment forbids at this stage), it may not be worth offloading to the GPU.
4. **Errors**





The plots show nearly zero relative error for both row and column operations across all matrix sizes and block sizes. This suggests that the CPU and GPU computations produce essentially the same results within floating-point precision.

5. **Which Operations Are Worth It?**

   • **Column Summation** is absolutely worth doing on the GPU because of the big speedups (CPU is slow on columns).

   • **Row Summation** can still be worthwhile for larger matrices, although the speedups are not as dramatic.

   • **Naive Reduction** is **not** particularly beneficial with this approach, because the CPU version is already fast, and we have no shared memory optimizations on the GPU.

Task 4 - double precision testing

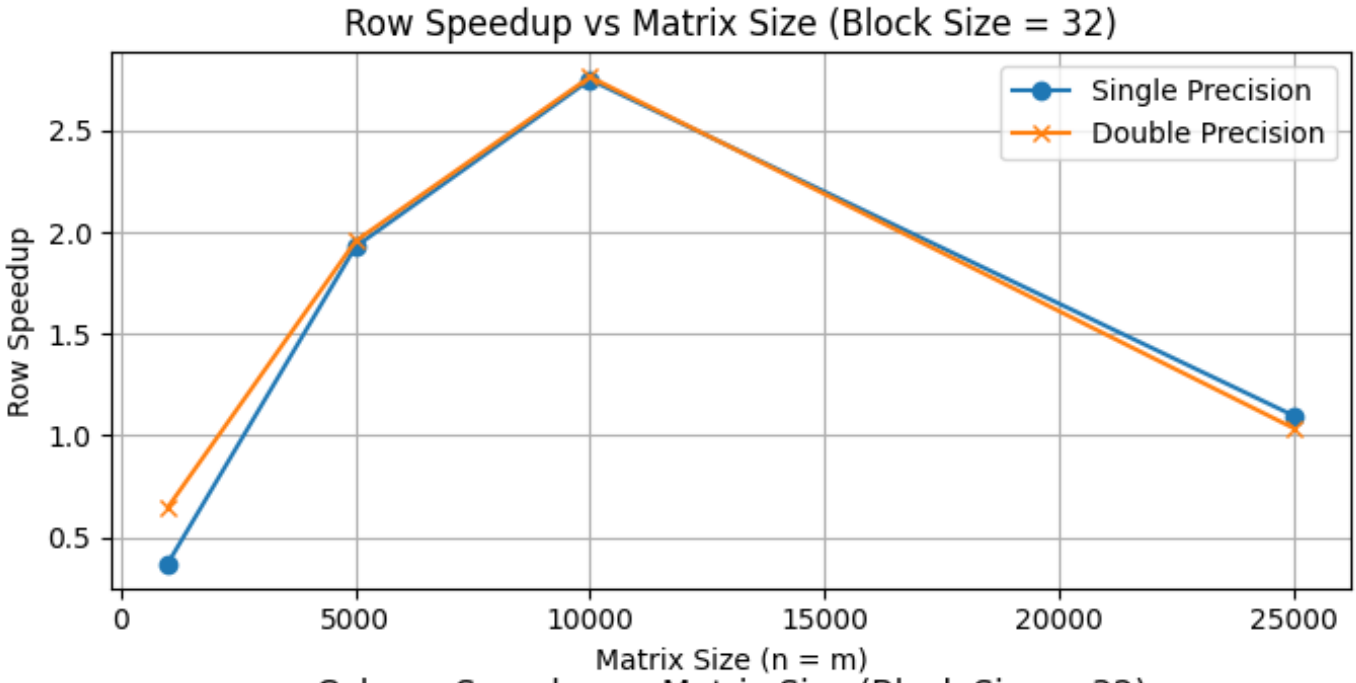
Based on the result graphs from the single-precision tests, the best overall performance was achieved with a block size of around **32 threads per block**.

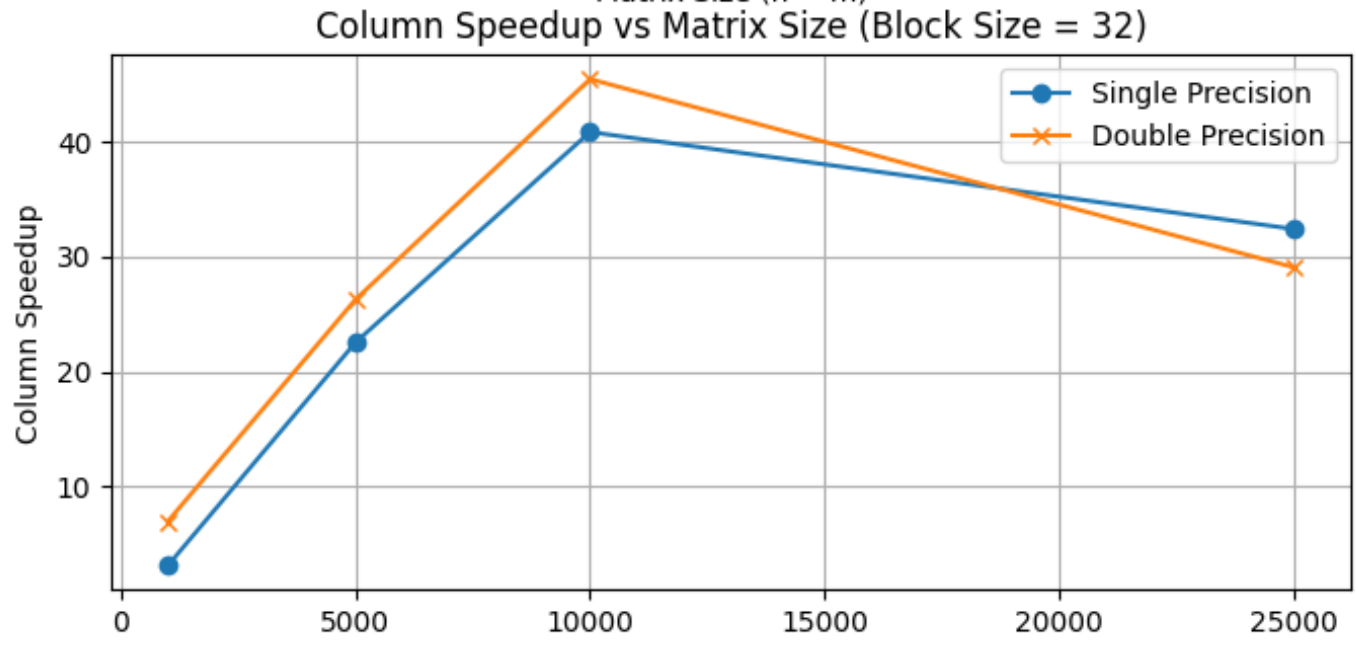
1. **Run** `make run_task04` to get the results of double precision results `task04_results.txt`.

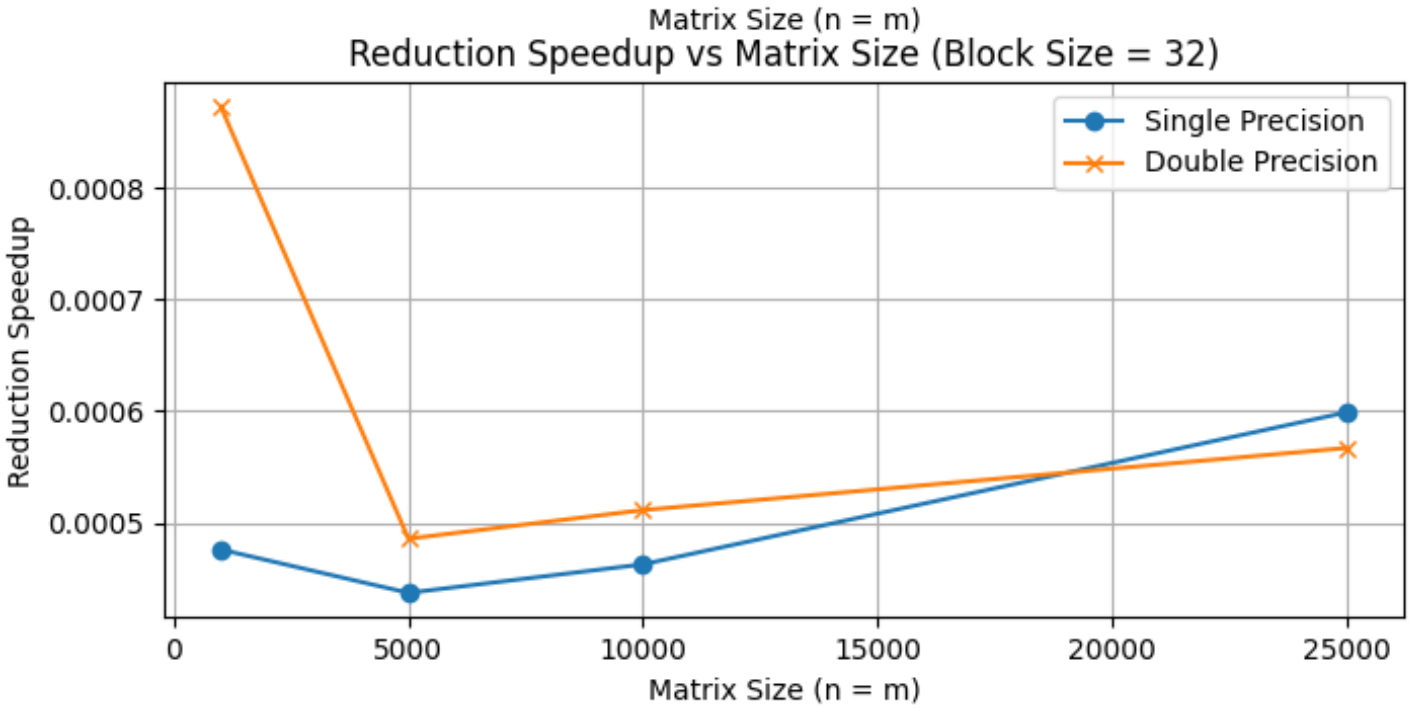
```
kuangg@cuda01:~/homework/assign01$ make run_task04
```

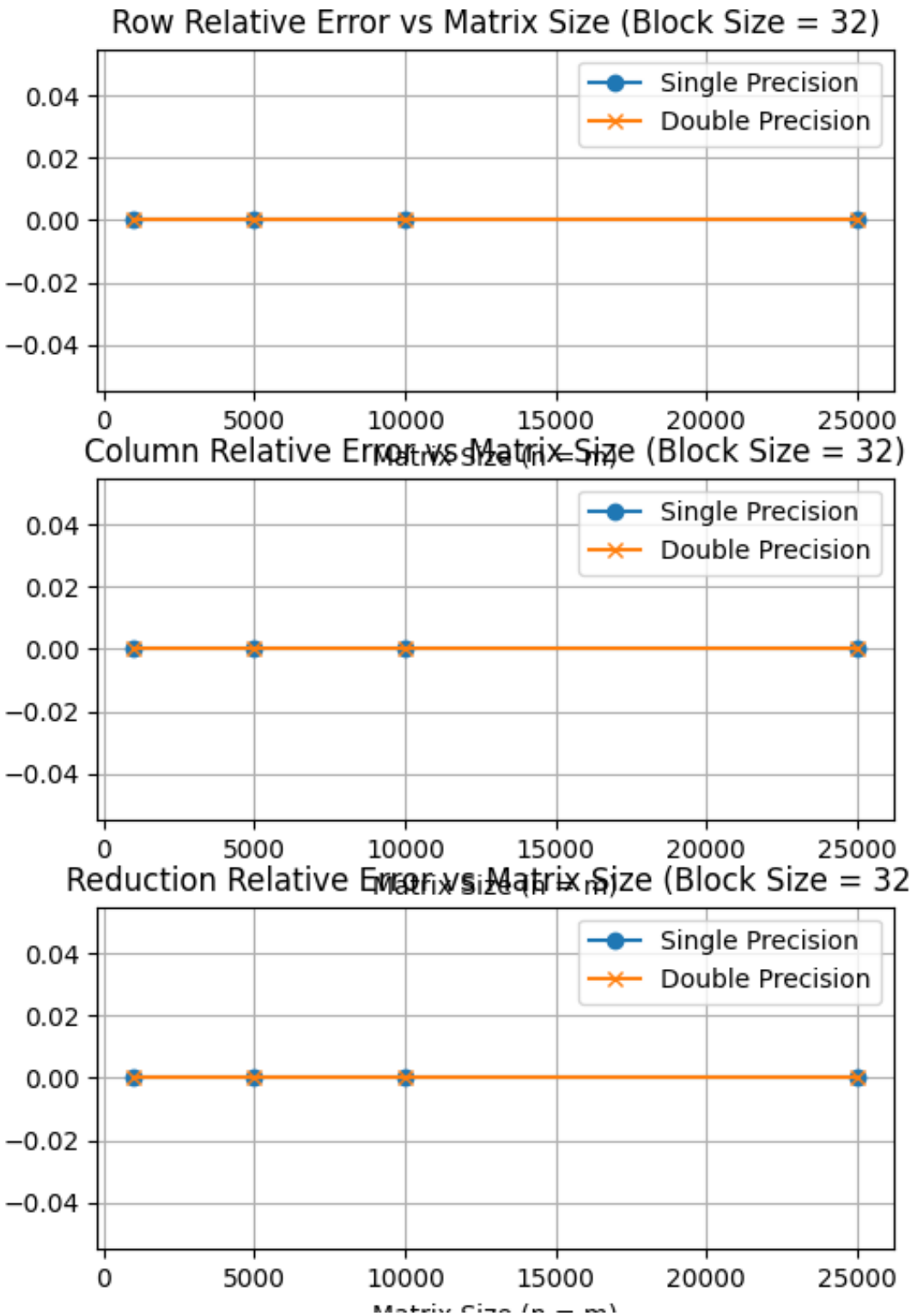
2. **Run** `python3 comp_results.py` to draw the performance picture.（**Make sure you have python3, matplotlib and Pandas installed**）.

Performance Analysis:






Matrix Size (n = m)
Reduction Speedup vs Matrix Size (Block Size = 32)
Single Precision
Double Precision
Reduction Speedup
0.0008
0.0007
0.0006
0.0005
0
5000
10000
15000
20000
25000
Matrix Size (n = m)



Based on these plots for **Task 4**—comparing **double-precision** vs. **single-precision** at a **block size of 32** and matrix sizes [1000, 5000, 10000, 25000]—we can draw the following conclusions:

1. **Speedups**
   - For **row summation**, **column summation**, and **reduction**, **double precision** typically has **slightly lower speedups** than single precision, especially at larger matrix sizes. This is because the GPU has lower throughput for double-precision operations, making them slower.
   - The overall **shape** of the speedup curves (increasing with matrix size) is the **same** for both single and

double precision. The GPU still gains efficiency on bigger matrices because it can better amortize memory transfer overhead and utilize parallelism.

- **Column summation** remains the operation with the highest speedups (over 20× for large matrices) since the CPU performs column access very inefficiently in row-major memory. Even in double precision, we see a big advantage over the CPU.

2. **Relative Errors**
   - All three errors (row, column, and "reduction" as the average) are essentially **near zero** for both single and double precision. This suggests that, in this particular summation approach, single-precision arithmetic already matches the CPU results closely, and double precision does not significantly change the outcome.
   - If the problem data or the summation order were more sensitive to floating-point round-off, you might see a bigger difference. But here, the GPU and CPU are producing extremely close results either way.
3. **Overall Comparison**
   - **Performance**: Double precision is consistently a bit slower (thus lower speedup) than single precision, which is expected on most consumer GPUs. The difference is not huge, but it is noticeable.
   - **Accuracy**: The relative errors are practically zero in both single and double precision, so in this particular setup, double precision does **not** provide a meaningful improvement in accuracy.

Therefore:

- We know that double precision yields very similar numerical results to single precision for these operations, but with **slightly lower GPU speedups**.
- The minimal error difference indicates that single precision was already sufficiently accurate for these matrix sums.