Sum Performance Analysis
Assignment #1, CSC 746, Fall 2024

Shun Usami[*]
SFSU

Abstract

This study investigates the impact of varying memory access patterns on the performance of different summation methods. We implemented three sum computation methods to measure performance: a direct sum with no memory access, a vector sum with sequential memory access, and an indirect sum with random memory access. We instrumented the code to measure elapsed time and derived performance metrics such as MFLOP/s, memory bandwidth utilization, and memory latency. The direct sum method proved to be most efficient in terms of MFLOP/s due to minimal memory access, while the vector sum outperformed the indirect sum in memory-related metrics because it can utilize spatial locality more efficiently. The results emphasize the importance of maximizing spatial locality to improve computational efficiency.

1 Introduction

The objective of this assignment is to examine different summation methods and evaluate their performance across various metrics on a supercomputing platform. We implemented three distinct summation approaches: direct sum, vector sum, and indirect sum, each designed to showcase different memory access patterns. Performance evaluation was conducted on the Perlmutter supercomputer at NERSC, utilizing C++ for implementations. Key performance metrics, such as MFLOP/s, memory bandwidth utilization, and memory latency, were derived from measuring the elapsed time of each method. In this report, we discuss the effect of memory access patterns on computational performance. Detailed implementations are provided in Section 2.

The results indicate that the direct sum method is the most efficient in terms of MFLOP/s because it avoids memory accesses, which are typically the most costly operations. In contrast, the vector sum method performs significantly better than the indirect sum in both memory bandwidth utilization and memory latency. This performance gap suggests that the indirect sum's memory access pattern is highly inefficient, leading to sub-optimal performance in these metrics.

2 Implementation

We implemented three sum computation methods to measure performance: direct sum with no memory access, vector sum with contiguous memory access, and indirect sum with random memory access.

2.1 Direct sum

The objective of implementing the direct sum is to measure performance when the program involves no memory access. This approach isolates the computational cost without the overhead of memory operations.

No extra setup is needed for the direct sum, and we used only a loop index and an accumulator variable, avoiding memory access for each iteration. The actual implementation is shown in Listings 1 and 2.

```
void setup(int64_t N, uint64_t A [])
{
    // do nothing
}
```

Listing 1: Direct Sum Setup: No setup required.

```
int64_t sum(int64_t N, uint64_t A [])
{
    int64_t sum = 0;
    for (int64_t i = 0; i < N; i++)
    {
        sum += i;
    }
    return sum;
}
```

Listing 2: Direct Sum: Computes the sum by directly adding the loop index values.

2.2 Vector sum

The objective of implementing the vector sum is to measure performance when the program accesses memory sequentially, maximizing spatial locality.

The setup method initializes an array of length N to contain the values 0 to $N-1$. The sum method calculates the sum of the values in the array by accessing each subsequent element in each iteration. The actual implementation is shown in Listings 3 and 4.

```
void setup(int64_t N, uint64_t A [])
{
   for (int64_t i = 0; i < N; i++)
   {
      A[i] = i;
   }
}
```

Listing 3: Vector Sum Setup: Initializes an array with sequential values from 0 to $N-1$.

[*]email:susami@sfsu.edu

```
int64_t sum(int64_t N, uint64_t A[])
{
    int64_t sum = 0;
    for (int64_t i = 0; i < N; i++)
    {
        sum += A[i];
    }
    return sum;
}
```

Listing 4: Vector Sum: Calculates the sum by sequentially accessing and adding the elements of the array.

2.3 Indirect sum

The objective of implementing the indirect sum is to measure performance when the program accesses memory randomly, which results in low spatial locality.

The setup method initializes an array of length N to contain the values 0 to $N-1$ and then creates a random permutation of these values. This permutation is used to form a single cycle in which each element points to the next element in the cycle, ensuring that every value from 0 to $N-1$ is visited exactly once. The sum method calculates the sum of the array's values by following the indices determined by this cycle, starting from the first index and moving to the next as dictated by the current value. The actual implementation is shown in Listings 5 and 6.

```
void setup(int64_t N, uint64_t A [])
{
   std::vector<int> walk_order(N);
   for (int64_t i = 0; i < N; i++)
   {
      walk_order[i] = i;
   }
   std::shuffle(walk_order.begin(), walk_order.end
    (), std::default_random_engine());
   // random walk
   int64_t index = walk_order[0];
   for (int64_t i = 0; i < N; i++)
   {
      int64_t next_index = walk_order[(i+1) % N];
      A[index] = next_index;
      index = next_index;
   }
}
```

Listing 5: Indirect Sum Setup: Initializes an array with values 0 to $N-1$ and shuffles them to create a single cycle for indirect access.

```
int64_t sum(int64_t N, uint64_t A[])
{
    int64_t sum = 0;
    for (int64_t i = 0; i < N; i++)
    {
        int64_t indirect_index = A[i];
        sum += A[indirect_index];
    }
    return sum;
}
```

Listing 6: Indirect Sum: Computes the sum by following a random walk through the array indices, accessing elements indirectly.

3 Evaluation

To evaluate the impact of different memory access patterns on performance, we conducted a series of experiments using the three sum computation methods. For each method, we measured the elapsed time and computed key performance metrics such as MFLOP/s, memory bandwidth utilization, and memory latency. These metrics allow us to understand how efficiently each method uses computational resources and memory subsystems under varying access patterns.

3.1 Computational Platform and Software Environment

The experiments were conducted on the Perlmutter supercomputer at NERSC. The key specifications are summarized in Table 1.

We used *gcc version 7.5.0 (SUSE Linux)* as the C++ compiler. For optimization, the following compiler flags were applied: *-O3 -DNDEBUG -Wall -pedantic -march=native*, aiming for maximum speed and strict standard compliance. Additionally, to assess the impact of compiler optimizations, we tested with the *-O0* flag, which disables optimizations, to provide a baseline for performance comparison.

Specification	Details
CPU	AMD EPYC 7763 (Milan) [2]
Instruction Set	AVX2 instruction set [2]
Number of Cores	64 [2]
Clock Rate	2.45 GHz [2]
L1 Cache	32 KB per core [1]
L2 Cache	512 KB per core [1]
L3 Cache	32 MB shared among 8 cores [1]
DRAM	512 GB DDR4 [2]
Memory Bandwidth	204.8 GB/s per CPU [2]
Max Memory Channels	8 per socket [1]
Computational Throughput	39.2 GFLOPS per core [2]
Operating System	x86_64 GNU/Linux

Table 1: Specifications of the Computational Platform Used for Experiments

3.2 Methodology

We evaluated the performance of different summation methods using problem sizes of 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27}, and 2^{28}. Performance was measured by calculating the elapsed time using an instrumentation code placed around the main summation code, as shown in Listing 7. From the elapsed time, we derived three key performance metrics:

- **MFLOP/s**: Measures computational throughput as operations per second, calculated as:
  - $MFLOP/s = \frac{\text{ops}}{\text{time}}$
  - ops = number of operations / $1M$
  - $time$ = runtime (seconds)
- **Memory Bandwidth Utilization (%)**: The percentage of theoretical peak memory bandwidth used, computed by:
  - $\%\ Memory\ Bandwidth = \frac{\text{bytes/time}}{\text{capacity}} \times 100$
  - $bytes$ = number of memory bytes accessed
  - $time$ = runtime (seconds)
  - $capacity$ = theoretical peak memory bandwidth of the system (see Table 1)
  - Since the direct sum method does not access memory, its memory bandwidth utilization is consistently 0%.
- **Average Memory Latency**: Measures the average time per memory access, defined as:

- *Avg Memory Latency* = $\frac{\text{time}}{\text{accesses}}$
- *time* = runtime (seconds)
- *accesses* = number of memory accesses by the program
- Since the direct sum method does not access memory, its memory latency is consistently 0.

```
for (int64_t n : problem_sizes)
{
    setup(n, &A[0]);

    start_time = get_current_time();
    result = sum(n, &A[0]);
    end_time = get_current_time();

    elapsed_time = end_time - start_time;
}
```

Listing 7: Measuring Elapsed Time:

3.3 Experiment: MFLOP/s

In this experiment, we aimed to evaluate the computational throughput (MFLOP/s) for different memory access patterns. The configuration details for this experiment are provided in Sec. 3.2. The results are summarized in Table 2 and Table 3, and visualized in Fig. 1 and Fig. 2.

The results show that the MFLOP/s is significantly higher with the *-O3* optimization flag compared to *-O0*. The direct sum method achieved the highest MFLOP/s, while the vector sum, though less efficient than the direct sum, performed much better than the indirect sum. The performance of the direct sum and vector sum methods remained relatively stable across different problem sizes, whereas the MFLOP/s of the indirect sum method decreased as the problem size increased.

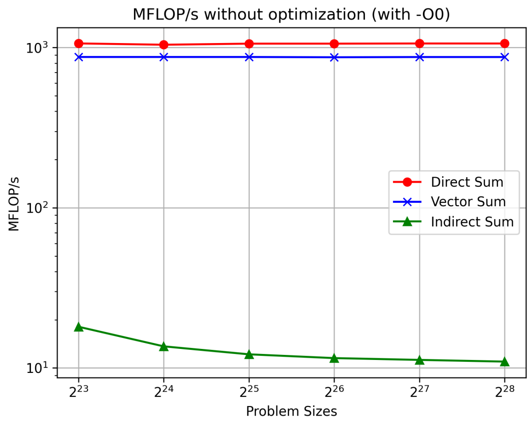

Figure 1: MFLOP/s without optimization (*-O0*).

3.4 Experiment: Memory Bandwidth Utilization (%)

In this experiment, we evaluated the memory bandwidth utilization for each memory access pattern. The configuration details for this experiment are provided in Sec. 3.2. The results are summarized in Table 4 and Table 5, and visualized in Fig. 3 and Fig. 4.

Since the direct sum method does not access memory, its memory bandwidth utilization is consistently 0%. The results indicate that,

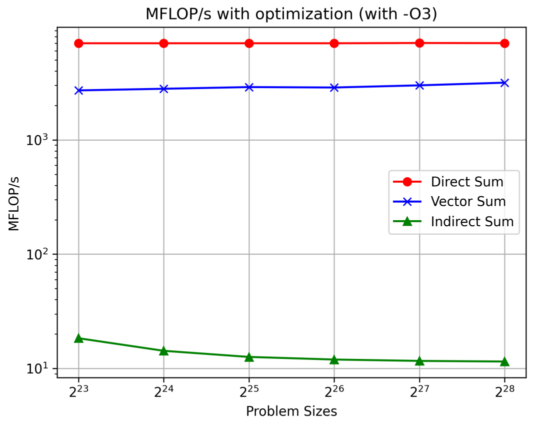

Figure 2: MFLOP/s with optimization (*-O3*).

Problem Size (N)	Direct Sum	Vector Sum	Indirect Sum
2^{23}	1,061.8	873.8	18.1
2^{24}	1,042.1	873.8	13.7
2^{25}	1,058.5	873.8	12.2
2^{26}	1,058.5	870.4	11.5
2^{27}	1,061.0	873.2	11.2
2^{28}	1,060.6	873.2	11.0

Table 2: MFLOP/s without optimization (with *-O0*)

Problem Size (N)	Direct Sum	Vector Sum	Indirect Sum
2^{23}	6,990.5	2,706.0	18.4
2^{24}	6,990.5	2,796.2	14.3
2^{25}	6,990.5	2,892.6	12.6
2^{26}	6,990.5	2,867.9	12.0
2^{27}	7,027.1	2,995.9	11.7
2^{28}	7,008.8	3,161.8	11.5

Table 3: MFLOP/s with optimization (with *-O3*)

without optimization (using *-O0*), the memory bandwidth utilization for the vector sum method remains stable across different problem sizes, while it slightly decreases for the indirect sum method as the problem size increases. With optimization (using *-O3*), a similar decreasing trend is observed for the indirect sum method, whereas the bandwidth utilization for the vector sum method increases as the problem size grows.

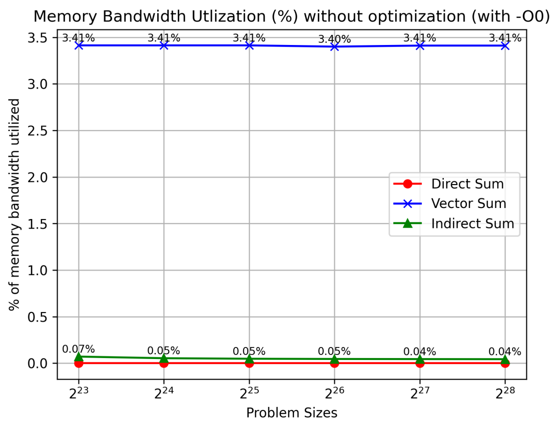

Figure 3: Memory Bandwidth Utilization (%) without optimization (-*O0*).

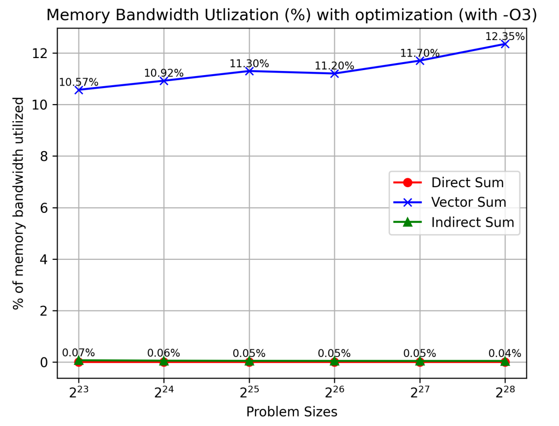

Figure 4: Memory Bandwidth Utilization (%) with optimization (-*O3*).

3.5 Experiment: Estimated Memory Latency

In this experiment, we estimated the memory latency for different memory access patterns across the three summation methods: direct sum, vector sum, and indirect sum. The configuration details for this experiment are provided in Sec. 3.2. The results are summarized in Table 6 and Table 7, and visualized in Fig. 5 and Fig. 6.

The results show that memory latency remains stable for the vector sum method across different problem sizes. In contrast, for the indirect sum method, memory latency increases as the problem size grows.

3.6 Findings and Discussion

3.6.1 MFLOP/s

The highest throughput achieved was 7,027.1 MFLOP/s with the direct sum method, which avoids memory access and thus is suitable

Problem Size (N)	Direct Sum	Vector Sum	Indirect Sum
2^{23}	0%	3.41%	0.07%
2^{24}	0%	3.41%	0.05%
2^{25}	0%	3.41%	0.05%
2^{26}	0%	3.40%	0.05%
2^{27}	0%	3.41%	0.04%
2^{28}	0%	3.41%	0.04%

Table 4: Memory Bandwidth Utilization (%) without optimization (with -*O0*)

Problem Size (N)	Direct Sum	Vector Sum	Indirect Sum
2^{23}	0%	10.57%	0.14%
2^{24}	0%	10.92%	0.11%
2^{25}	0%	11.30%	0.10%
2^{26}	0%	11.20%	0.09%
2^{27}	0%	11.70%	0.09%
2^{28}	0%	12.35%	0.09%

Table 5: Memory Bandwidth Utilization (%) with optimization (with -*O3*)

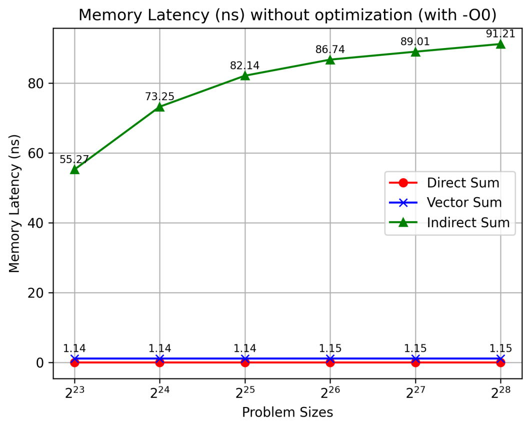

Figure 5: Estimated Memory Latency without optimization (-*O0*).

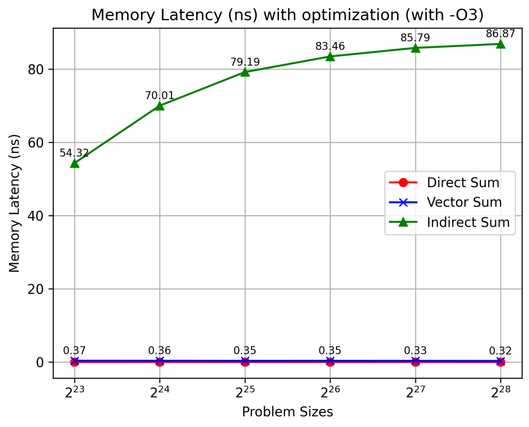

Figure 6: Estimated Memory Latency with optimization (-*O3*).

Problem Size (N)	Direct Sum (ns)	Vector Sum (ns)	Indirect Sum (ns)
2^{23}	0.00	1.14	55.27
2^{24}	0.00	1.14	73.25
2^{25}	0.00	1.14	82.14
2^{26}	0.00	1.15	86.74
2^{27}	0.00	1.15	89.01
2^{28}	0.00	1.15	91.21

Table 6: Memory Latency without optimization (with *-O0*)

Problem Size (N)	Direct Sum (ns)	Vector Sum (ns)	Indirect Sum (ns)
2^{23}	0.00	0.37	27.16
2^{24}	0.00	0.36	35.00
2^{25}	0.00	0.35	39.59
2^{26}	0.00	0.35	41.73
2^{27}	0.00	0.33	42.89
2^{28}	0.00	0.32	43.43

Table 7: Memory Latency with optimization (with *-O3*)

for measuring pure computational throughput (see Table 3). However, this is lower than the theoretical peak of 39.2 GFLOPS per core for the AMD EPYC 7763 processor (see Table 1).

This gap is due to several factors: the theoretical peak assumes full utilization of both threads per core and the AVX2 instruction set, which can perform eight single-precision operations per thread. The peak is calculated as $2.45\,\text{GHz} \times 2\,\text{threads} \times 8 = 39.2\,\text{GFLOPS}$. In contrast, our program uses only one thread, and the 'vpaddq' SIMD instruction in the compiled code, which we observed in the disassembled program, allows only two computations per cycle, limiting performance.

3.6.2 Memory Bandwidth

The minimum, maximum, and average memory bandwidth for each summation method are shown in Table 8 and Table 9. The direct sum method consistently shows the minimum bandwidth as it does not access memory. In contrast, the vector sum method achieves the highest utilization due to its efficient use of sequential memory access, which enhances spatial locality. However, even the maximum observed bandwidth of 25.29 GB/s falls short of the system's theoretical peak of 204.8 GB/s (see Table 1). The program might not fully utilize the multiple memory channels available on the system because it runs on a single thread, which potentially underutilizes the 8 memory channels available per socket (see Table 1).

	Direct Sum	Vector Sum	Indirect Sum
Min	0.00	6.96	0.09
Max	0.00	6.99	0.14
Avg	0.00	6.98	0.10

Table 8: Min/Max/Average Memory Bandwidth Utilization (GB/s) without optimization (with *-O0*)

	Direct Sum	Vector Sum	Indirect Sum
Min	0.00	21.65	0.09
Max	0.00	25.29	0.15
Avg	0.00	23.23	0.11

Table 9: Min/Max/Average Memory Bandwidth Utilization (GB/s) with optimization (with *-O3*)

3.6.3 Memory Latency

The estimated memory latency for each summation method is presented in Table 6 and Table 7. The direct sum method has the lowest latency as it does not involve memory access. In contrast, the indirect sum with the largest problem size (2^{28}) compiled with the no-optimization flag (*-O0*) shows the highest latency due to poor spatial locality from random memory accesses. As the problem size increases, cache hit rates decrease because cache lines are more likely to be evicted before being reused.

3.6.4 Compiler Optimization

The effects of compiler optimization (*-O3*) are significant when there is no memory access (direct sum) or when memory access is efficient (vector sum). In these cases, the compiler can effectively optimize computational instructions, leading to noticeable performance improvements. However, for the indirect sum method, the impact of compiler optimization is minimal. This is because the performance gains from compiler optimizations are only a few nanoseconds per loop, whereas the cache miss penalty, caused by the lack of spatial locality, is approximately 100 nanoseconds per loop. As a result, the benefits of optimization are overshadowed by the higher cost of cache misses in the indirect sum method.

ACKNOWLEDGMENTS

This paper was edited with the assistance of ChatGPT-4o (accessed September 2024), primarily for correcting grammatical errors and rephrasing. This research used resources from the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility using NERSC award DDR-ERCAP m3930 for 2024.

REFERENCES

[1] AMD. *HPC Tuning Guide for AMD EPYC 7003 Series Processors*, 2022. Accessed: September 2024.

[2] NERSC. Perlmutter architecture, 2024. Accessed: September 2024.