Summation Method Performance Analysis

Ben Klein
Computer Science Department
San Francisco State University
San Francisco, CA, USA
Email: bklein@sfsu.edu

***Abstract*—This paper is studying the performance of three different methods of summation. We use a direct summation, a vector summation, and an indirect summation that traverses a vector in a random order. Each implementation used timers and was run on CPU nodes in the Perlmutter supercomputer at NERSC. We measure performance using MFLOP/s, percentage of bandwidth utilized, and average memory latency, up to problem sizes of 268 million elements. The results show that direct and vector summation runs roughly two orders of magnitude faster than indirect summation methods that traverse arrays in random order. These findings show that serial array access is vastly quicker in practice than random array access.**

I. Introduction

In this paper, we compare the run-times of three different summation methods: a direct summation, a vector summation, and an indirect summation with random array traversal. We show that serial traversal of arrays is about two orders of magnitude faster than random traversal, showing that clever implementation that ensures serial traversal is extremely important.

II. Implementation

The objective of the implementations is to show how serial code execution times compare with random order execution times.

This section describes the three summation algorithms we implemented and tested: *direct sum*, *vector sum*, and *indirect sum*. Each implementation results in different performance characteristics which is detailed in the next sections.

A. Direct Sum

The direct sum algorithm computes a sum over N number of elements, without utilizing any memory. All summation is done directly within a register.

B. Vector Sum

The goal of the vector sum summation is similar to the direct sum, however this algorithm sums a vector with N elements, showing the effects of memory in the performance of the algorithm.

C. Indirect Sum

The objective of the indirect sum is to show the inefficiency of accessing random elements within an array. Accessing serial elements in an array is very efficient, but random access is around two orders of magnitude slower.

III. Evaluation

The computing environment used for this evaluation is Perlmutter as NERSC. Python version 3.12.11 is used for plotting the graphs used in this evaluation.

A. Computational platform and Software Environment

These tests utilize the Perlmutter supercomputer. The stats below are found at https://docs.nersc.gov/systems/perlmutter/architecture/

- 2 × AMD EPYC 7763 (Milan) CPUs
- 64 cores per CPU
- AVX2 instruction set
- 512 GB DDR4 memory total
- 204.8 GB/s memory bandwidth per CPU
- 1 × HPE Slingshot 11 NIC
- PCIe 4.0 NIC–CPU connection
- 39.2 GFLOPs per core
- 2.51 TFLOPs per socket
- 4 NUMA domains per socket (NPS=4)

B. Procedures

We implemented each of the three codes (direct sum, vector sum, and indirect sum) in C++. We used C++ `chrono` timers to measure elapsed clock time for each execution. Each run was performed on a CPU node on Perlmutter at NERSC to obtain consistent results.

C. Performance Metrics

We used the following performance metrics:

- **MFLOP/s**: $\frac{\text{ops}/10^6}{\text{time (s)}}$
- **Memory bandwidth utilization**: $\frac{\text{bytes}/\text{time}}{\text{capacity}}$
- **Average memory latency**: $\frac{\text{time}}{\text{accesses}}$

D. MFLOP/s Results

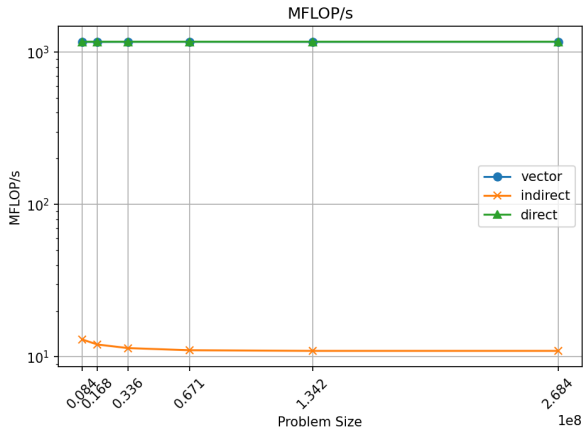

Fig. 1. MFLOP/s vs. problem size for direct, vector, and indirect kernels.

Vector sum shows 1 GFLOP/s; indirect is $\sim$100$\times$ slower. Direct is used mainly as a baseline, which for this example matches the vector sum.

E. % of Peak Memory Bandwidth Utilized

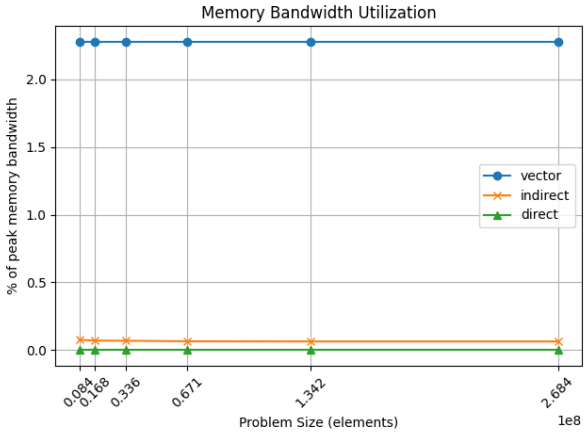

Fig. 2. Percent of Perlmutter memory bandwidth (204.8 GB/s) vs. problem size.

Vector reaches a few percent of peak; indirect is $< 1\%$ due to random access. Direct uses only registers and no memory and achieves similar performance to vector summation.

F. Estimated Memory Latency

TABLE I
ESTIMATED AVERAGE MEMORY LATENCY (NS/ACCESS). VECTOR USES N ACCESSES; INDIRECT USES $2N$; DIRECT HAS NO ARRAY ACCESSES.

Problem Size	Direct	Vector	Indirect
8,388,608	–	0.086	38.5
16,777,216	–	0.086	41.6
33,554,432	–	0.086	43.8
67,108,864	–	0.086	45.27
134,217,728	–	0.086	45.71
268,435,456	–	0.086	46.07

G. Experimental Design

We executed each code across a set of problem sizes that are powers of two, ranging from $2^{23} = 8{,}388{,}608$ up to $2^{28} = 268{,}435{,}456$ elements. For each problem size, we measured runtime and computed the metrics above.

H. Findings and Discussion

In this experiment, we found that we were able to achieve around 1100-1200 MFLOP/s for the direct and vector sum algorithms, but were around 100x slower with indirect sum, which showed around 11-12 MFLOP/s. The peak Perlmutter core CPU computation is 39.2 GFLops per core, but that is only under highly optimized conditions. In future work, we can work to optimize the code to achieve closer to this limit.

The lowest memory latency is direct sum which uses no memory and only registers, followed by the indirect sum which accesses one array. The reason why the indirect sum uses less memory latency than the vector sum is because the bottleneck for vector sum is in the arithmetic calculation rather than the memory access. The memory access for indirect looks better because the amount of time needed for indirect calculation is greater, but the bottleneck is in the calculation rather than memory access.