CPSC 524 Assignment 1: Divide and Vector Triad Performance

Rami Pellumbi*

September 21, 2023

*M.S., Statistics & Data Science

1 Introduction

This assignment investigated the characteristics of the processors on the Grace Linux Cluster. First, a program is written to approximate π by numerically integrating the function

$$f(x) = \frac{4.0}{1.0 + x^2}$$

from 0 to 1. The divide latency is estimated by assuming that floating point division cannot be pipelined and assuming that the calculation is dominated by the cost of the divide. Next, a program is written to benchmark the performance in MFlops of the vector triad kernel. The MFlops are plotted vs. $\log N$, where N is the size of the arrays in the vector triad benchmark. The development environment was VsCode on MacOS via SSH. The configuration is documented separately in `report/ssh_vscode_grace.pdf`.

2 Project Hierarchy

The project has the following folder hierarchy:

```
a1
├─ build/
├─ common
│  ├─ timing.h
│  └─ timing.c
├─ part1
│  ├─ integrate.c
│  ├─ integrate.h
│  └─ pi.c
├─ part2
│  ├─ data/
│  ├─ dummy.c
│  ├─ plot.rmd
│  ├─ triad.c
│  ├─ utilities.c
│  └─ utilities.h
├─ report/
├─ sbatch-out/
├─ build-run-pi.sh
├─ build-run-triad.sh
└─ Makefile
```

- **build**: The `build` directory is designated for housing the output files generated during the compilation process of both `part1` and `part2`. It acts as a repository for these compiled objects, allowing for easier access and management.
- **common**: The `common` directory serves as a repository for code that is shared between `part1` and `part2`. Specifically, this directory contains the file `timing.c`, which encapsulates the functionality related to timing measurements. The flag `-I common` must be in the `CFLAGS` of the `Makefile` to ensure this directory is part of the compilation.
- **part1**: The `part1` directory focuses on the task of numerically estimating the value of π. It utilizes a midpoint integration scheme, the details of which are implemented in the `integrate.c` file. The overall orchestration for estimating π and calculating the latency of division operations is carried out by the `pi.c` file.
- **part2**: The `part2` directory houses multiple files, each with distinct responsibilities. The `dummy.c` file serves the purpose of an opaque function used to ensure our benchmark code runs. The `triad.c` file is dedicated to performing the vector triad benchmark, an operation that evaluates the memory bandwidth and computational speed for vector operations. Lastly, the `utilities.c` file contains helper functions for array allocation, initialization with random numbers, and data output to a CSV file for subsequent graphical plotting.
- **report**: The `report` directory contains the tex

files used to write the report and VsCode instruction guide.

- **sbatch-out**: The `sbatch-out` directory serves as a centralized location for storing all output files generated from code executions on the compute node. The directory's internal folder structure is systematically organized based on the compiler options that led to each individual output file, facilitating easier debugging and performance analysis.
- **build-run-pi.sh**: This is a Bash script specifically crafted to automate the compilation and execution of the code residing in the `part1` directory. To deploy this script, navigate to the `a1` directory and execute the command `sbatch build-run-pi.sh`. This will submit the script to the compute node and subsequently initiate the execution of the Part 1 code.
- **build-run-triad.sh**: Similar to `build-run-pi.sh`, this Bash script is designed to compile and execute the code for Part 2. Once again, to utilize this script, position yourself in the `a1` directory and submit the command `sbatch build-run-triad.sh`. This action will queue the script for execution on the compute node, leading to the running of the Part 2 code.
- **Makefile**: This Makefile has undergone extensive modifications to support the unique build requirements of both `part1` and `part2`. It is engineered to handle the appropriate linking of shared libraries and integration of common code components, ensuring a seamless compilation process for both parts. By utilizing this Makefile, the user can effortlessly build and link the code, thereby simplifying the overall build process.

3 Building and Running the Code

This section outlines the steps required to build and execute the code for both Part 1 and Part 2 of the project. The provided Bash scripts automate the entire process, making it straightforward to compile and run the code.

3.1 Automated Building and Execution

- **Part 1: Numerical Estimation of π**
  1. Navigate to the `a1` directory if you haven't already.
  2. Run the following command to submit the Part 1 build and execution script:
     `sbatch build-run-pi.sh`

  This script employs a heavily modified `Makefile` to handle the compilation and linking of Part 1-specific code.
- **Part 2: Vector Triad Benchmark**
  1. Make sure you are still in the `a1` directory.
  2. Execute the following command to submit the Part 2 build and execution script:
     `sbatch build-run-triad.sh`

  Similar to Part 1, this script utilizes the same modified `Makefile` to manage the build process for Part 2.

3.2 Post-Build Objects

Upon successful compilation and linking, a `build` directory will be generated within the `a1` root directory. This directory will contain the compiled output files for both parts. Additionally, the executable files for running each part will be situated in the root of the `a1` directory.

3.3 Output Files

The output files generated from running the code will be stored in the root of the `a1` directory. They are manually moved to the `sbatch-out` directory under the relevant part and compiler options.

4 Division Performance

The first benchmark estimates the latency of the divide operation and assesses processor performance across different `icc` compiler options.

Integration and Timing Setup

The midpoint integration scheme is done by the function `integrate_midpoint_rule` implemented in `integrate.c` as follows:[1]

```c
double integrate_midpoint_rule(double start,
                               double end,
                               double num_steps,
                               double (*func)(double))
{
    // width of the interval at each midpoint
    double width_of_interval = 1.0 / num_steps;
    // initial midpoint is halfway between start and (start + 1/num_steps)
    double current_midpoint = (start + width_of_interval) / 2.0;
    // function value at the midpoint - initialized to the value at inital midpoint
    double current_function_value = func(current_midpoint);

    // initialize the sum value to the first rectangles area
    double sum = current_function_value * width_of_interval;

    // compute each midpoint, function value at that midpoint, and add rectangle area to the sum.
    for (int i = 1; i < num_steps; i++)
    {
        current_midpoint += width_of_interval; // 1 FLOP
        current_function_value = func(current_midpoint);
        sum += current_function_value * width_of_interval; // 2 FLOPS
    }

    return sum;
}
```

The function is invoked in `pi.c` with:

- `start = 0.0`.
- `end = 1.0`.
- `num_steps =` 1000000000.
- `func = function_to_integrate`.

where `function_to_integrate` is

```c
double function_to_integrate(double x)
{
    // 1 divide, 1 add, 1 multiply -> 3 FLOPS
    return 4.0 / (1.0 + x * x);
}
```

The results are timed as follows:

[1] All code shown is roughly similar to the submitted code. It is slighlty modified for clear reporting and the official code file should be considered as the final submitted solution as it typically is more detailed.

```c
double start_wc_time = 0.0, end_wc_time = 0.0;
double start_cpu_time = 0.0, end_cpu_time = 0.0;

timing(&start_wc_time, &start_cpu_time);
// Performing 6 FLOPS per iteration, and NUM_STEPS iterations.
double pi = integrate_midpoint_rule(START_X,
                                    END_X,
                                    NUM_STEPS,
                                    function_to_integrate);
timing(&end_wc_time, &end_cpu_time);
```

Estimating MFlops and Runtime

Since we perform 6 floating point operations per iteration of the midpoint integration scheme, we perform 6 billion floating point operations between the start of the wall clock time and the end of the wall clock time. Thus, we can estimate the runtime in seconds and performance in MFlops by

```c
// runtime
double elapsed_wc_time = end_wc_time - start_wc_time;

// performance in MFlops
double total_number_of_flops = 6.0 * 1e9;
double mega_flops_per_second = total_number_of_flops / elapsed_wc_time / 1.0e6;
```

Runtime of Algorithm Across Compiler Options: Performance in MFlops Using One Core of One Compute Node

The results of the π estimate, estimated MFlops, and elapsed wallclock time are shown for the following four compiler options:

1. `-g -O0 -fno-alias -std=c99`

```
pi estimate = 3.141593
elapsed wall clock time = 4.896937
Estimated MFLOPS = 1225.255732

real 0m4.901s
user 0m4.891s
sys  0m0.002s
```

2. `-g -O3 -no-vec -no-simd -fno-alias -std=c99`

```
pi estimate = 3.141593
elapsed wall clock time = 2.655869
Estimated MFLOPS = 2259.147565

real 0m2.660s
user 0m2.652s
sys  0m0.002s
```

3. `-g -O3 -fno-alias -std=c99`

```
pi estimate = 3.141593
elapsed wall clock time = 2.655767
Estimated MFLOPS = 2259.234369

real 0m2.661s
user 0m2.654s
sys  0m0.001s
```

4. `-g -O3 -xHost -fno-alias -std=c99`

```
pi estimate = 3.141593
elapsed wall clock time = 2.655834
Estimated MFLOPS = 2259.177175

real 0m2.670s
user 0m2.653s
sys  0m0.002s
```

Some notes on what these options mean and the differences in each run:

- The compiler optimizations do not impact the accuracy of the algorithm.
- All options include `-g -fno-alias -std=c99` which means we generate debuggin information, disable pointer aliasing, and use the C99 standard, respectively.
- The flag `-O0` disables compilor optimizations while the flag `-O3` offers aggressive and extensive optimizations.[2]
- The flags `-no-vec -no-simd` together disable all compiler vectorization.[3]
- The flag `-xHost` tells the compiler to generate instructions for the highest instruction set available on the compilation host processor. Given the host processor is an Intel(R) Xeon(R) Platinum 8268, using `-xHost` means the compiler may use AVX-512 instructions, fused multiply add instructions, and bit manipulation.[4]

Given the flag information and performance results, it can be concluded that:

- Options 2, 3, and 4 are very similar. We can reasonably conclude that vectorization is not a component of this algorithm.
- The `-O0` flag results in very slow execution, illustrating the impact of compiler optimizations.

The performance is what I expected for the first and fourth compiler option. Going into the problem, I expected disabling vectorization would decrease performance but this was not the case. This makes sense given the lack of vectorization in the program.

Divide Latency Estimate

Assuming the performance of the π calculation is dominated by the cost of the division operations, and assuming that division cannot be pipelined, we can estimate the latency of the divide operation by disabling all compiler optimization and computing the total number of cycles in our integration scheme divided by the total number of steps, e.g.,:

[2] Optimizations Compiler Documentation.
[3] SIMD Compiler Documentation.
[4] xHost Compiler Documentation. The CPU flags were assessed from running `lscpu` and inferring what the compiler may do.

```
// found via lscpu on the compute node in the "CPU MHz:" row.
// Assumed all cores run at this frequency.
double cpu_frequency_hertz = 3.5e9;
double total_number_of_cycles = elapsed_wc_time * cpu_frequency_hertz;
double estimated_divide_latency = total_number_of_cycles / num_steps;
```

The specific compiler option used was: `-g -O0 -fno-alia -no-vec -no-simd`. This resulted in an estimated divide latency of about 17 cycles. The compiler options do matter in estimating this latency. For example, when running option 4 from before, the estimated divide latency is about 9 cycles. When the compiler is able to do aggressive optimizations, our assumptions begin to have less validity and the estimation is not as accurate, e.g., if multiple divides are being concurrently performed in the optimized version we have to account for that in our latency estimate.

5 Vector Triad Performance

The second benchmark measures the performance in MFlops of the vector triad kernel:

$$\texttt{a[i]} = \texttt{b[i]} + \texttt{c[i]} * \texttt{d[i]},$$

where `a`, `b`, `c`, and `d` are double precision arrays of length `N` initialized with values in $[0, 100]$. The arrays are allocated memory and initialized via the following two helper functions:

```
void *allocate_double_array(size_t num_elements)
{
    // allocate uninitialized memory
    void *array = malloc(num_elements * sizeof(double));

    // exit if allocation failed
    if (array == NULL) exit(1);

    return array;
}

void initialize_array_with_random_numbers(double *array, size_t num_elements)
{
    // for each element in the array, store a random double in [0,100]
    for (size_t i = 0; i < num_elements; ++i)
    {
        array[i] = ((double)rand() / (double)RAND_MAX) * 100.0;
    }
}
```

Setting up the Benchmark

The performance benchmark ensures that the program runs for at least 1 second in duration. To ensure the operations in the kernel actually get executed, we insert a conditional call to an opaque function `dummy` residing in `dummy.c`. The expression in the conditional is such that the compiler can not easily determine the result of the conditional statement at compile time. The expression of choice is `a[N >> 1] > 0`, which is always `false` when a is initialized with positive numbers. The benchmark is summarized in the following snippet:

```c
int number_of_repetitions = 1;
double runtime = 0.0;
while (runtime < 1.0)
{
    // start the timing
    timing(&start_wc_time, &start_cpu_time);
    for (int r = 0; r < number_of_repetitions; ++r)
    {
        for (size_t i = 0; i < num_elements; i++)
        {
            a[i] = b[i] + c[i] * d[i];
        }

        // this if condition is always false when a is initiated with all positive values
        if (a[num_elements >> 1] < 0) dummy(a, b, c, d);
    }
    // end the timing
    timing(&end_wc_time, &end_cpu_time);
    runtime = end_wc_time - start_wc_time;
    number_of_repetitions *= 2;
}
number_of_repetitions /= 2;
```

The benchmark itself takes place in `part2/triad.c` and the script takes as input a positive integer `k`. The length `N` is then computed as

$$\mathtt{N} = \mathtt{floor}(2.1^k).$$

The benchmark is run for k in the inclusive range 3...25. The script `build-run-triad.sh` has a simple for-loop to provide the triad program with each value of k:

```bash
for k in {3..25}
do
  echo "Running with k = $k"
  time ./triad $k
done
```

Each call of the program will output the total megaflops per second and number of elements in the arrays to `part2/data/part_2_data.csv` via the helper function `write_data_to_file` located in `part2/utilities.c`. Since each iteration of the vector triad is two floating point operations, we compute the total floating point operations to be the number of repetitions times the number of elements in the array times 2 divided by the runtime converted to megaflops:

```c
double total_mega_flops = 2.0 * (double)N * number_of_repetitions / runtime / 1.0e6;
```

Running under the compiler options `-g -O3 -xHost -fno-alias -std=c99`, we produce the following application output for $k = 5, 11, 22$:

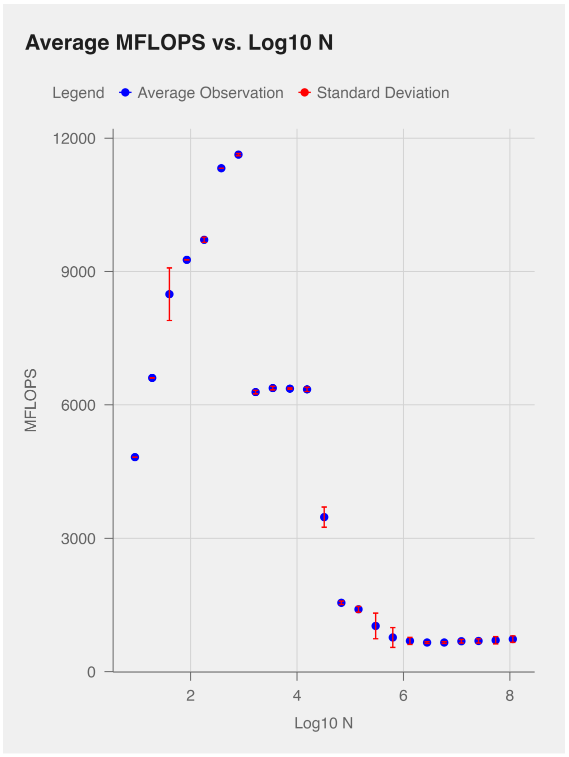
```
Running with k = 5

number of array elements: 40
134217728 repetitions performed
elapsed wall clock time = 1.234365
elapsed cpu time = 1.232172
estimated MFLOPS: 8698.738508
real  0m2.476s
user  0m2.464s
sys   0m0.002s
```

```
Running with k = 11

number of array elements: 3502
1048576 repetitions performed
elapsed wall clock time = 1.152193
elapsed cpu time = 1.149823
estimated MFLOPS: 6374.128172
real  0m2.306s
user  0m2.295s
sys   0m0.002s

Running with k = 22

number of array elements: 12269432
32 repetitions performed
elapsed wall clock time = 1.239635
elapsed cpu time = 1.235572
estimated MFLOPS: 633.447470
real  0m2.826s
user  0m2.739s
sys   0m0.073s
```

In total, 8 jobs were submitted to slurm via `sbatch`. Each job output the results of megaflop performance vs. `N` in `part2/data/part_2_data.csv` for every `k` value. The csv file was loading using the `plot.rmd` file to show the performance in MFlops versus `log N`.

Assessing Performance vs. Array Size

Having 8 data points for each value of `k`, the plot below shows the average MFlops versus $\log_2$ `N`. The red lines are the standard deviation from the mean.



The plot results are quite interesting. First, a bit of processor information. The Intel(R) Xeon(R) Platinum 8268 processor has:

- 24 × 32K `L1d` cache (stores data). This is 8-way set associative with write-back.
- 24 × 32K `L1i` cache (stores instructions for cores to execute). This is 8-way set associative.
- 24 × 1024K `L2` cache. This is 16-way set associative with write-back.
- 1 × 36608K `L3` cache. This is 11-way set associative with write-back. It is shared by every core on the processor.

Moreover,

- The `L2` cache is inclusive.
- The `L3` cache is shared by every core on the chip.
- The `L3` cache is not inclusive.

Knowing our cache information, the performance starts to make sense. The benchmark was run on 1 core of a single node and thus we have access to 1 of each of the level 1 and 2 caches.