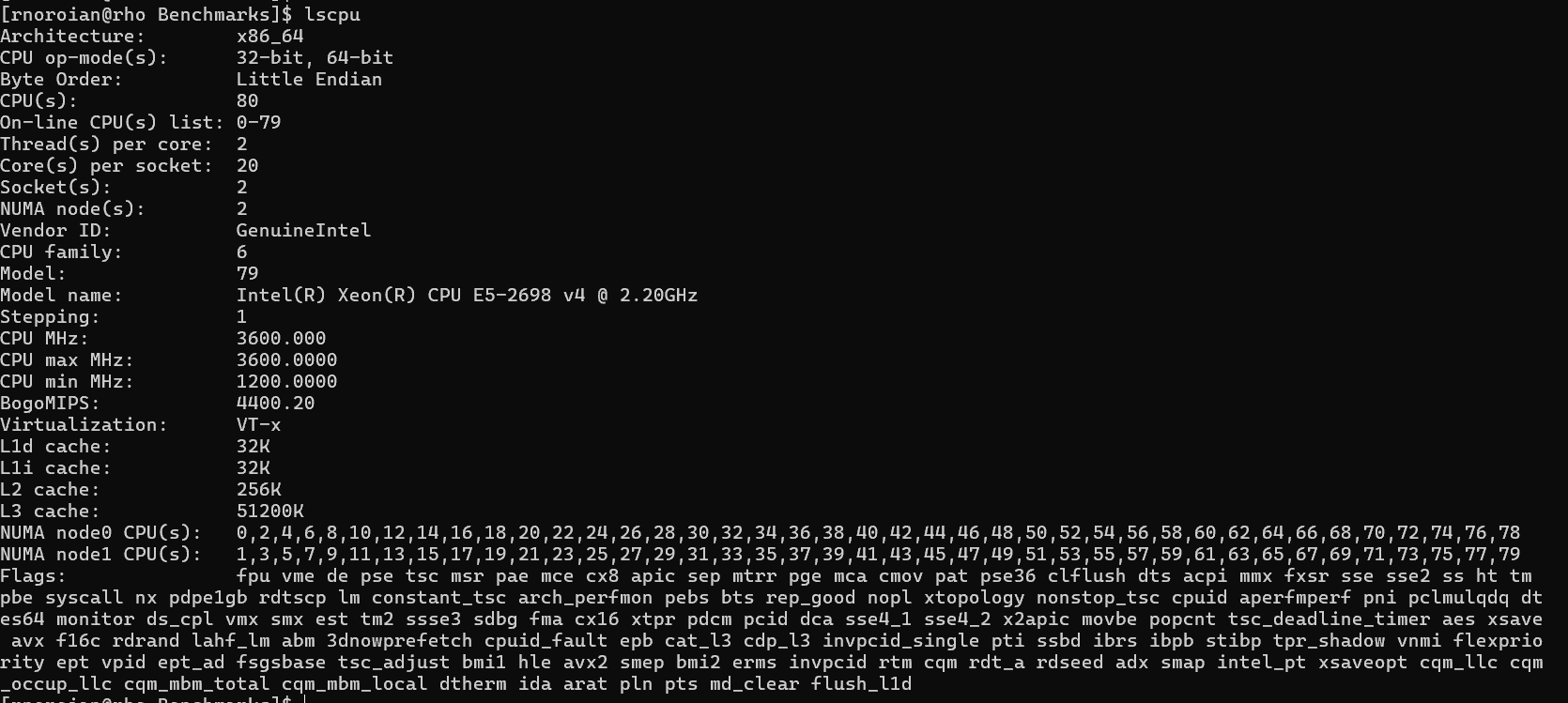
Q1 Single-Threaded Benchmarks:

The two systems I used were:

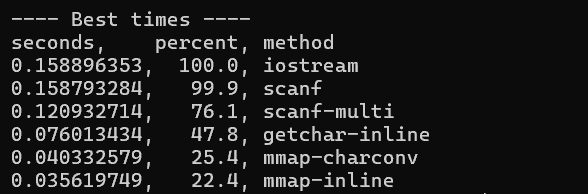
1) This is the COE vector CPU: 2) This is a CPU node, c0744, on the Explorer Cluster:



A&B)

Benchmark One: The first benchmark is called [parse-integers-benchmark,](https://github.com/max0x7ba/parse-integers-benchmark) which evaluates file reading and parsing for integers in a single threaded manner. It uses iostream, scanf, getchar, and mmap to test, and completes 10 tests for each. It then outputs the best performances.

Example Output:

A

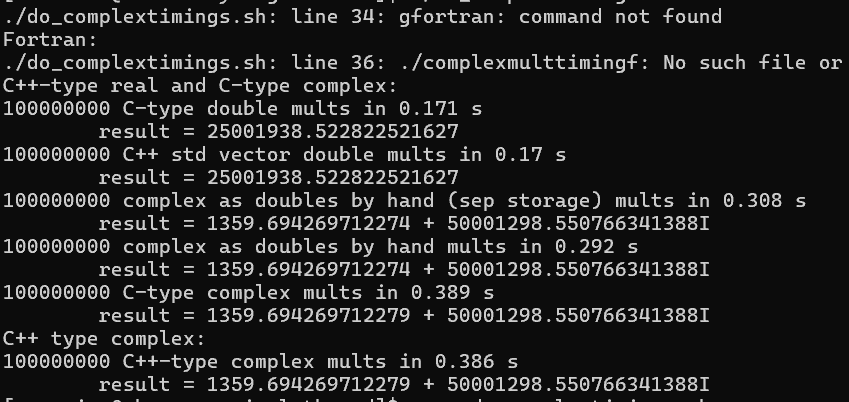
|  |  |  |
| --- | --- | --- |
| Average Times (s) | Vector CPU | Explorer CPU |
| IOStream | 0.158 | 0.184 |
| Scanf | 0.158 | 0.185 |
| Scanf | 0.121 | 0.145 |
| Getchar | 0.076 | 0.098 |
| Mmap | 0.04 | 0.055 |
| Mmap | 0.036 | 0.043 |

Explanation:

The performance results demonstrate that the Explorer CPU was slower than the Vector CPU across all metrics. The first CPU has 80 total threads, while the second has 28. This means that there are more threads in the Vector CPU to handle the benchmark’s single-threaded workload, as the benchmark involves repeatedly parsing data of varying sizes with different techniques.

Benchmark Two: This benchmark is [arraysinglethread](https://github.com/ahbarnett/floatingspeed/tree/master/arraysinglethread), which compares performance of arithmetic in C, C++, and Fortran, primarily analyzing differences in memory-bound tasks and incorporating float-point operations. Fortran failed to run on the Vector CPU, so C and C++ were compared here. It provides multiple different compile options, but I used #export FLAGS="-Ofast -funroll-loops -march=native" initially.

Example Output:



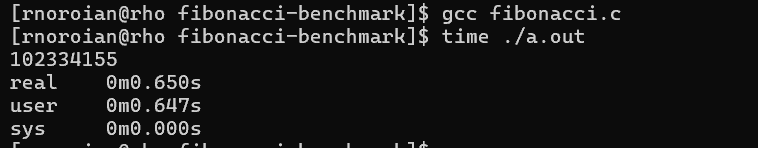
|  |  |  |
| --- | --- | --- |
| Average Times (s) | Vector CPU | Explorer CPU |
| C-type double mults | 0.171 | 0.176 |
| C++ vector double mults | 0.17 | 0.174 |
| C-type as double by hand mults | 0.308 | 0.334 |
| C++-type as doubles by hand complex mults | 0.292 | 0.337 |
| C-type complex mults | 0.389 | 0.256 |
| C++-type complex mults | 0.386 | 0.253 |

Explanation:

For this benchmark, the primary difference came between the C/C++ operations by hand, and the C/C++ type complex operations. The Vector CPU’s operations by hand were more efficient, while the Explorer CPU’s complex operations were more efficient. For complex operations, due to the Vector CPU’s greater number of cores, I believe the Explorer had better instruction pipelining and fewer memory accesses of more complex data structures, enabling its efficiency. For by hand operations, I believe the greater L1 cache size in the Vector CPU enabled it to store values nearer to the processor core, making it more efficient.

Benchmark Three: The final benchmark I selected was a single-threaded [fibonacci](https://github.com/yakkomajuri/fibonacci-benchmark) program. It executes a Fibonacci function recursively to determine the 40th Fibonacci number. I chose this benchmark to measure basic performance differences between the two CPUs.

Example Output:



|  |  |
| --- | --- |
| Vector CPU Time (s) | Explorer CPU Time (s) |
| 0.65 | 0.970 |
| 0.65 | 0.970 |
| 0.649 | 0.969 |
| 0.649 | 0.969 |
| 0.651 | 0.968 |
| 0.653 | 0.969 |
| 0.649 | 0.971 |
| Average: | Average: |
| 0.650 | 0.970 |

Explanation:

The Vector CPU, with greater cache sizes, and more cores, is about 33% faster than the Explorer CPU. Since the Fibonacci program is recursive, it involves a larger number of function calls pushing onto the stack. To keep track of a Fibonacci number of 40, memory access is frequent, and a greater cache size in the Vector CPU allows the CPU to fetch data quicker. This compounds for greater Fibonacci sequences, as they require more calls.

C) Compiler Optimizations Available for the Benchmarks

Benchmark 1:

This benchmark benefited from Ofast optimization, which lacks strict compliance with certain more complex operations. However, since these weren’t needed, this specification serves as an aggressive optimization towards program optimization in reading file data. Ofast enables faster loop unrolling, which is useful as that is how the parse-integers-benchmark processes most of the data through its various methods/tests.

Benchmark 2:

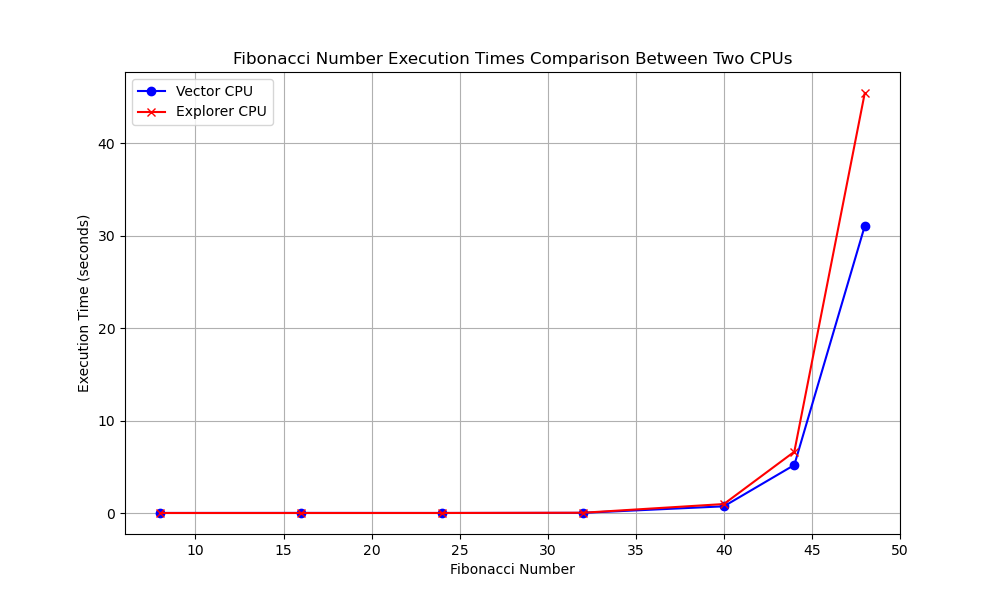
The Single Threaded array computation benefited from fcx-limited-range, which optimized floating-point computations, which was the datatype used for some of the benchmark tests. This optimization reduces the range of floating-point operations, which makes calculations more efficient and simpler. It can trade off precision for performance, but that is not the case as the values used in this benchmark aren’t particularly complex.

Benchmark 3:

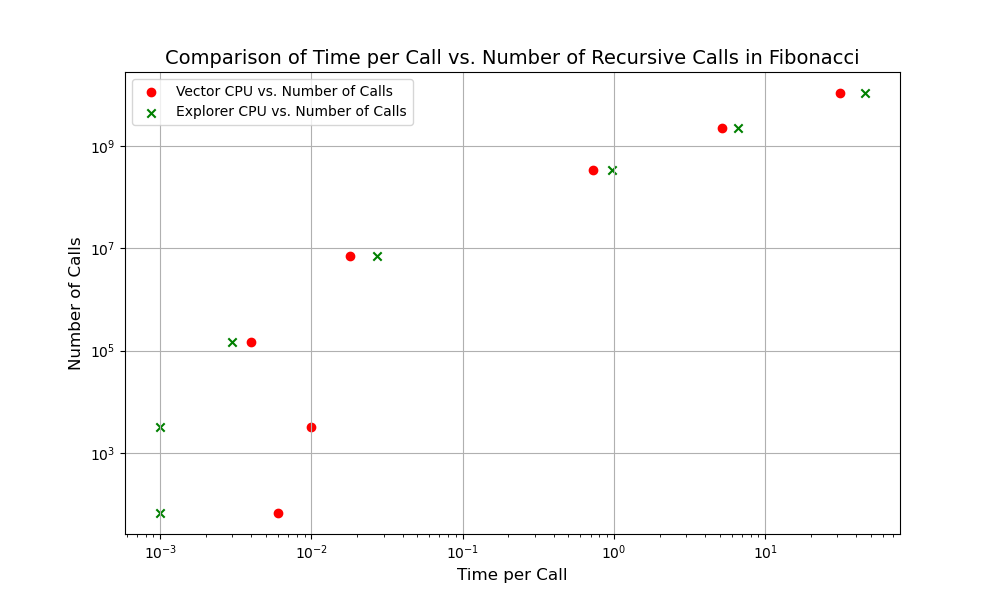
The Fibonacci benchmark was initially compiled and run with gcc and no additional specifications. I added –O2, which sped up the program by 50%, resulting in a 0.3 second completion time on the Vector CPU. This is because of O2’s tail call optimization, which reuses the current stack frame for recursive calls, preventing pushes to new calls when re-adding up all the values to get the next.

D) My Own Metric for Benchmark 3 – Fibonacci Recursion

To test each CPU’s performance with recursion, I developed a metric **time per recursive call.** As the Fibonacci number is originally 40, if it were modified to 20, or increased to 60, this metric tests the total execution and divides it by the number of recursive calls. This tests memory size and performance as recursive intensive programs, without optimization, leading to many calls to access data in the caches. If recursion depth is large, it may take longer to fetch data to continue computation of the desired Fibonacci number. The Fibonacci run time is O(2^n), and if linearity isn’t present in this metric, then there are different specifications in architecture.

Plot demonstrating pure execution time over a set of Fibonacci numbers:Plot for Time per Recursive Call:

To determine the number of recursive calls, I implemented a function to count calls. The plot is below:

It can be seen that for the lower number of recursive calls, the explorer CPU performs quicker. At scale, the Vector CPU slightly outperforms the Explorer CPU in times per call, allowing it to see much more efficient run times for large amounts of recursive calls.

|  |  |  |
| --- | --- | --- |
| Vector CPU Time | Explorer CPU Time | Fib(N), # Calls |
| 0.006 | 0.001 | 8, 67 |
| 0.01 | 0.001 | 16, 3193 |
| 0.004 | 0.003 | 24, 150049 |
| 0.018 | 0.027 | 32, 7049155 |
| 0.724 | 0.970 | 40, 331160281 |
| 5.196 | 6.633 | 44, 2269806339 |
| 31.066 | 45.452 | 48, 10779215329 |

E) Applying pthread to each Benchmark

Benchmark 1:

For several of the integer parsing tests, like methodscanf(), or method\_getchar\_inline(), for loops are used to collect data space by space. I believe pthreads can be utilized to turn these methods into thread tasks, where threads can be given offsets into the file, so that each saves a different part of the data. File inputs can be processed concurrently rather than sequentially, as they are currently done.

Benchmark 2:

For the array-single-thread benchmark, a lot of the calculations rely on previously calculated data. Regardless, pthread can be used to speed this up. To ensure that data is updated as the array index is incremented, a mutex can be used to synchronize the modification of the array, so that the right data can be accessed by the next thread for the next element. Though this won’t improve program efficiency with full concurrency, it will make function implementation much smoother as one method can be used, with creating and joining threads to modify the array.

Benchmark 3:

The third benchmark involves recursion, which is difficult to parallelize with pthread. To utilize pthread effectively, I believe it would be valuable to implement an iterative Fibonacci program, where each thread calculates the next element of the Fibonacci array. A mutex would be needed here as well to prevent newer threads from modifying greater Fibonacci elements before the values in previous elements have been set.