IN THE NAME OF ALAH

vectorization in C++

student : Amirreza Hosseini

student number : 996003086

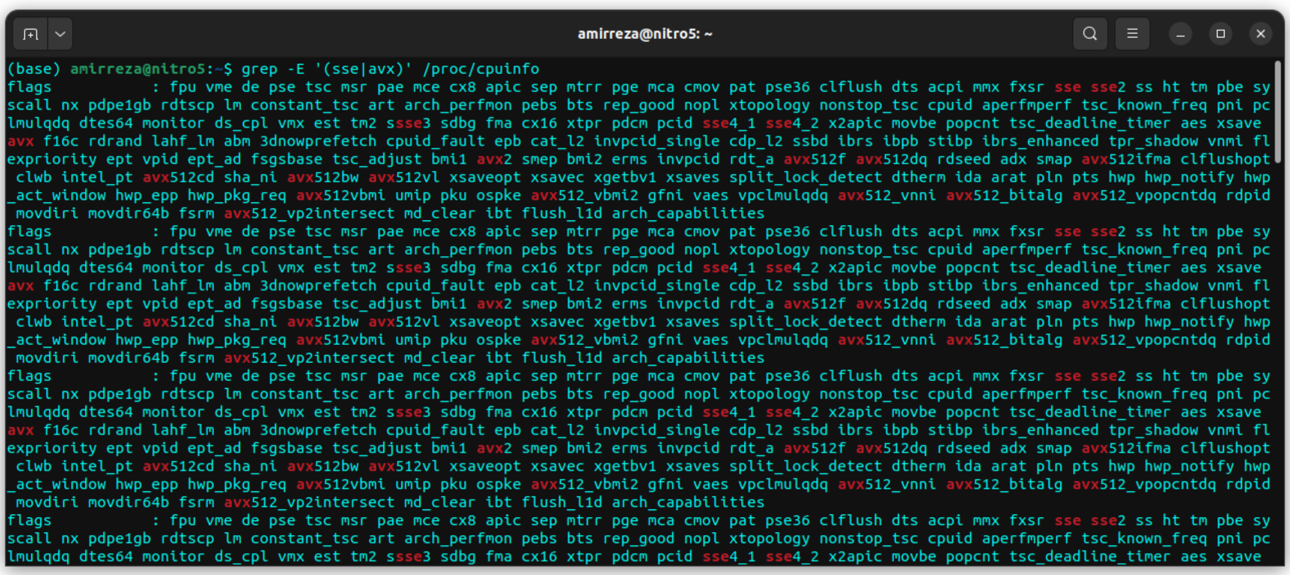
Vectorization in C++

Vectorization is a powerful technique in C++ that allows for the execution of operations on entire arrays, rather than individual elements. This can significantly improve the performance of your code, especially when working with large data sets.

The provided code demonstrates the use of vectorization in C++ through the use of different instruction set architectures (ISAs) - SSE2, SSE4.2, AVX2, and AVX512. These ISAs provide a set of instructions that can be used to perform operations on vectors, which are arrays of data.

SIMD

SIMD (Single Instruction, Multiple Data) is a type of parallel computing architecture that allows a single operation to be applied to multiple data points simultaneously. This can significantly improve the performance of your code, especially when working with large data sets. For checking for SIMD support in linux this command can help :
$ grep -E '(sse|avx)' /proc/cpuinfo



Code

main.cpp

You, 21 minutes ago | 1 author (You) | 💡 Click here to ask Blackbox to help you code faster |

```
#include <vector>
#include <string.h>
#include <iomanip>

#include "vectorization.cpp"

using namespace std;

#define TABLE_CELL_LEN 20

int64_t calculate_add_print(float *A, float *B, float *C, int size, int func)
{
    tick();
    add_vf(func)(A, B, C, size);
    auto dur = tock();
    return dur;
}

int64_t calculate_add_print(double *A, double *B, double *C, int size, int func)
{
    tick();
    add_vf_d(func)(A, B, C, size);        You, 21 minutes ago • Uncommitted changes
    auto dur = tock();
    return dur;
}

```

main.cpp

```cpp
int main()
{
    int size = 16 * 1024 * 12;
    float A[size];
    float B[size];
    float C[size];

    double Ad[size];
    double Bd[size];
    double Cd[size];

    set_default(A, size, 1.12f);
    set_default(B, size, 2.11f);

    set_default(Ad, size, 1.12);
    set_default(Bd, size, 2.11);

    vector<string> names;
    vector<int> funcs;
    vector<int64_t> floatf_durations;
    vector<int64_t> doublef_durations;

    funcs.push_back(SSE2);
    funcs.push_back(SSE4_2);
    funcs.push_back(AVX2);
    funcs.push_back(AVX512);

    names.push_back(SSE2_STR);
    names.push_back(SSE4_2_STR);
    names.push_back(AVX2_STR);
    names.push_back(AVX512_STR);

    for (int i = 0; i < funcs.size(); i++)
    {
        floatf_durations.push_back(calculate_add_print(A, B, C, size, funcs.at(i)));
        doublef_durations.push_back(calculate_add_print(Ad, Bd, Cd, size, funcs.at(i)));
    }

    cout << "size of arrayes : " << size << endl;
    cout << setw(TABLE_CELL_LEN) << left << "Function" <<
            setw(TABLE_CELL_LEN) << left << "Float Time(us)" <<
            setw(TABLE_CELL_LEN) << left << "Double Time(us)" << endl;
    for (int i = 0; i < names.size(); i++)
    {
        cout << setw(TABLE_CELL_LEN) << left << names.at(i) <<
                setw(TABLE_CELL_LEN) << left << floatf_durations.at(i) <<
                setw(TABLE_CELL_LEN) << left << doublef_durations.at(i) << endl;
    }
}
// g++ -mavx2 -mavx512f main.cpp -o main  && ./main
```

Code Explanation

The code begins by defining an array size and initializing two arrays, **A** and **B**, with default values. It then uses different ISAs to add the elements of these arrays together, storing the result in a third array, **C**. This is done for both **float** and **double** data types.

The time taken to perform these operations is measured and stored in **floatf_durations** and **doublef_durations** respectively. The function **tick()** is

called before the operation to start the timer, and **tock()** is called after the operation to stop the timer and return the elapsed time.
Finally, the code prints out the size of the arrays and a table showing the time taken for each operation. The table is formatted using the **setw()** function from the **<iomanip>** library, which sets the field width for output.

Output

```
● (base) amirreza@nitro5:~/Desktop/university/C_multitasking$ g++ -mavx2 -mavx512f main.cpp -o main  && ./main
  size of arrayes : 196608
  Function           Float Time(us)      Double Time(us)
  SSE2               535                 1228
  SSE4_2             613                 1234
  AVX2               337                 622
  AVX512             233                 440
```

Output Explanation

The output of the program shows the size of the arrays and the time taken for each operation in microseconds. The operations are performed using different ISAs, and the time taken for each is displayed in the table. The results show that AVX512 is the fastest for both **float** and **double** data types, followed by AVX2, SSE4.2, and SSE2.

This demonstrates the power of vectorization in C++ and the performance benefits that can be gained by using different ISAs. It's clear that using more advanced ISAs like AVX512 can significantly improve the performance of operations on large arrays.

Conclusion

Vectorization is a powerful technique in C++ that can significantly improve the performance of your code. By using different ISAs, you can optimize your code to perform operations on entire arrays, rather than individual elements. This can be especially beneficial when working with large data sets, as demonstrated by the provided code.

Please note that the actual performance may vary depending on the specific hardware and compiler optimizations. Always make sure to test your code on the target hardware to ensure optimal performance.