TP1 Parallel Programming

Notes

The goal of this work is to get started with SMID Note that `#include <immintrin.h>` must be used to work with it I am also using `std::chrono::high_resolution_clock` for time precision.

I also learnt quite late that the GCC compiler is way too smart. It is completely ignoring code blocks that have no impact on the rest of the program. Therefore, our functions blocks, which are doing something for the only sake of being chronometered, are not being used. To counter this effect, we are going to use optimization -O0 option of the GCC compiler.

CreateRandomVector

We are using an external function to initialize our vectors. A vector is a pointer to the initial element, and a size in memory, allocated with malloc. We use `#include <random>` to fill the vector.

Because I decided to work with separate files and such, I must be careful with headers, and add the files in the Makefile list.

ThreeVectorAverage

Each function initializes vectors and time.

scalar

```
start = std::chrono::high_resolution_clock::now();

for (unsigned long int i = 0; i < size; i++) {
    result[i] = (vector1[i] + vector2[i] + vector3[i])/3;
}

end = std::chrono::high_resolution_clock::now();

double duration = std::chrono::duration<double>(end-start).count();
```

It is a simple iteration over the vector's size.

vectorial

```
start = std::chrono::high_resolution_clock::now();


for (unsigned long i = 0; i < size; i+=8) {
    __m256 a = _mm256_loadu_ps(&vector1[i]);
    __m256 b = _mm256_loadu_ps(&vector2[i]);
    __m256 s = _mm256_add_ps(a, b);

    __m256 c = _mm256_loadu_ps(&vector3[i]);
    __m256 v = _mm256_add_ps(s, c);
    __m256 three = _mm256_loadu_ps(threes);
    __m256 r = _mm256_div_ps(v, three);
    _mm256_storeu_ps(&result[i],r);
}

end = std::chrono::high_resolution_clock::now();

double duration = std::chrono::duration<double>(end-start).count();
```

Here we are loading the two first vectors in memory using `_mm256_loadu_ps`, adding them with `_mm256_add_ps`.

All functions using _256 work with 6 32bits **floating numbers** here.

These are operations made especially for __256 types, that correspond to our processor. Then we load a third one and add it to the result. Finally we load a vector of 3s and use it for division. `_mm256_storeu_ps` permits to get the result back.

The vectorial function is slightly slower than its scalar counterpart, less than 10% so though. This is surprising at first, but G++ has auto vectorisation that is better than what we do manually. We still stay with the same range.

Two Vectors Scalar Product

Here we need to multiply element by element and then sum.

scalar

```
start = std::chrono::high_resolution_clock::now();

for (unsigned long int i = 0; i < size; i++) {
    scalarProduct += (vector1[i] * vector2[i]);
}
end = std::chrono::high_resolution_clock::now();

printf("%20f scalarproduct\n", scalarProduct);
double duration = std::chrono::duration<double>(end-start).count();
if (duration < min_duration) min_duration = duration;
```

vectorial

```
start = std::chrono::high_resolution_clock::now();

for (unsigned long i = 0; i < size; i+=8) {
    __m256 a = _mm256_loadu_ps(&vector1[i]);
    __m256 b = _mm256_loadu_ps(&vector2[i]);
    __m256 s = _mm256_mul_ps(a, b);

    __m256 t = _mm256_hadd_ps(s, s);
    __m256 u = _mm256_hadd_ps(t, t);
    _mm256_storeu_ps(partialResult, u);
    result += partialResult[0] + partialResult[4];
}

end = std::chrono::high_resolution_clock::now();

double duration = std::chrono::duration<double>(end-start).count();
if (duration < min_duration) min_duration = duration;
```

The first lines are the same thing as the exercise before. We are just multiplying bit by bit instead of adding.

However here, we use `_mm256_hadd_ps` to do the final sum in a little vectorized way. This functions sums the vectors by pairs, meaning if we have A and B, it returns `[A[0] + A[1], A[2] + A[3], B[0] + B[1], B[2] + B[3], ......]` So if we do it twice we will have the 8 elements summed in `partialResult[0] + partialResult[4]`.

This way we are only using 3 CPU cycles instead of 7 (to sum 8 elements). Even if this might not prove efficient, it is a nice way to familiarize with SMID thinking.

Min Max of a Vector

Sequential

```
start = std::chrono::high_resolution_clock::now();

for (unsigned long int i = 0; i < size; i++) {
    float element = vector[i];
    if ( element < min) {
        min = element;
    } else if ( element > max) {
        max = element;
    }
}
end = std::chrono::high_resolution_clock::now();

printf("%20f %20f min max\n", min, max);
double duration = std::chrono::duration<double>(end-start).count();
if (duration < min_duration) min_duration = duration;
```

This is very classical code to get min and max of a vector with two dancing references.

Vectorial

```
start = std::chrono::high_resolution_clock::now();

for (unsigned long i = 0; i < size; i+=8) {
    __m256 a = _mm256_loadu_ps(&vector[i]);
    __m256 minimum = _mm256_loadu_ps(minVec);
    minimum = _mm256_min_ps(a, minimum);
    __m256 maximum = _mm256_loadu_ps(maxVec);
    maximum = _mm256_max_ps(a, maximum);
    _mm256_storeu_ps(minVec, minimum);
    _mm256_storeu_ps(maxVec, maximum);
}
for (int i = 0; i < 8; i++) {
    if ( minVec[i] < min) {
        min = minVec[i];
    } else if ( maxVec[i] > max) {
        max = maxVec[i];
    }
}

end = std::chrono::high_resolution_clock::now();
```

Here we are comparing a eight value vector min to 8 values of the vector, and same with max. For instance, the `__m256 minimum` vector contains, for each of its 8 values, the minimum between the 8 values vector loaded `a`, and the minimum value we have found yet.

Therefore, for an integer i, the value minVec[i], which receives this result with `_mm256_storeu_ps`, represents `min (vector[0], vector[j], vector[j + 8], vector[j + 16] ..... vector[i])`. That is to said, the minimum of all the values of the initial vector at a position `j` that is `j % 8 = i`.

In the end, getting the minimum of the vector is just getting the minimum of minVec, which contains at index `i` these local minimums for every values which index can be devided by `i`.

Gaussian Filter

Sequential

```
// Side effects
result[0] = coefficients[1] * vector[0] + coefficients[2] * vector[1];
result[size] = coefficients[1] * vector[size] + coefficients[0] *
vector[size - 1];

for (unsigned long int i = 1; i < size - 1; i++) {
    result[i] = coefficients[0] * vector[i - 1] + coefficients[1] *
vector[i] + coefficients[2] * vector[i + 1];
}
```

Gaussian Filter is about using element before and element after, so I first need to deal with side effects.

Vectorial

```
/*
Setting up coefficients vector
This must be within clock, it's part of how I do the job
    */
for (int i = 0; i < 8; i++) {
    coef1[i] = coefficients[0];
    coef2[i] = coefficients[1] / 2; // ! Important divided by 2
    coef3[i] = coefficients[2];
}

// Side effects
result[0] = coefficients[1]*vector[0] + coefficients[2]*vector[1];
result[size] = coefficients[1] * vector[size] + coefficients[0] *
vector[size-1];

// Coef mm256, loaded before
__m256 m256Coef1 = _mm256_load_ps(coef1);
__m256 m256Coef2 = _mm256_load_ps(coef2);
__m256 m256Coef3 = _mm256_load_ps(coef3);

for (unsigned long i = 1; i < size-1; i++) {
    __m256 vec = _mm256_mul_ps(_mm256_load_ps(&vector[i]), m256Coef2);
    __m256 vecBefore = _mm256_mul_ps(m256Coef1, _mm256_load_ps(&vector[i-
1]));
    vec = _mm256_add_ps(vecBefore, vec);
    __m256 vecAfter = _mm256_mul_ps(m256Coef3, _mm256_load_ps(&vector[i +
1]));
    vec = _mm256_add_ps(vecAfter, vec);

    _mm256_storeu_ps(&result[i], vec);
}
```

Same here, all the setup must be in the chrono, because it's a longer setup than on the sequential code.

Here we are loading vec multiplying it by the main factor of the filter DIVIDED by 2, adding that to each vector, before and after. (Doing this by loading `vector[i-1]` and `vector[i+1]`).