Parallel Programming

CS575

Chao Zhang

Project #5

1. Source listing

nonsimd-re.cpp | nonsimd.cpp

```cpp
#include <stdio.h>
#include <stdlib.h> //
#include <omp.h>
#include "simd.p5.h"

int main(int argc, char const *argv[]) {
    int ARRAYS, NUMT;

    for (ARRAYS = 1024; ARRAYS <= 1024 * 1024 * 16; ARRAYS  *= 4)
    {
        float *A = (float *)malloc(sizeof(float)*ARRAYS);
        float *B = (float *)malloc(sizeof(float)*ARRAYS);
        float sum;

        double time_start = omp_get_wtime();  //time started
        for( long int i = 0; i < ARRAYS; i++ )
        {
            sum *= A[i]*B[i];
        }
        double time_end = omp_get_wtime();   //time ended

        double time_consumed = (time_end - time_start) * 1e6;
        printf("%.5lf\t", time_consumed);
        printf("%.5lf\n", (long int)ARRAYS / time_consumed);
    }

    return 0;
}
```

This is the code of non-simd and reduction. It calculates the sum of two array and print out the results so I can compare it with the simd and reduction's results.

nonsimd-re.cpp nonsimd.cpp

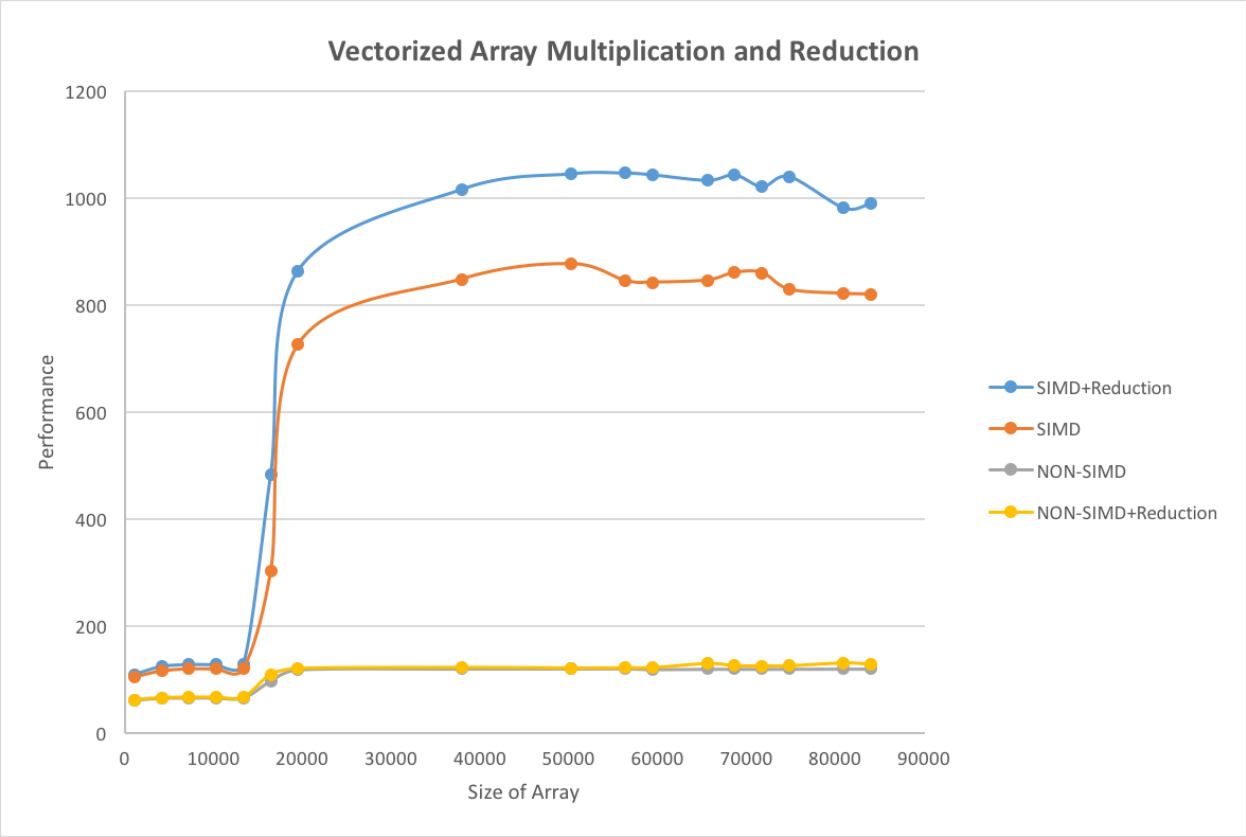
```cpp
#include <stdio.h>
#include <stdlib.h> //
#include <omp.h>
#include "simd.p5.h"

int main(int argc, char const *argv[])
{
    int ARRAYS;

    for (ARRAYS = 1024; ARRAYS <= 1024 * 1024 * 16; ARRAYS  *= 4)
    {
        float *A = (float *)malloc(sizeof(float)*ARRAYS);
        float *C = (float *)malloc(sizeof(float)*ARRAYS);
        float *B = (float *)malloc(sizeof(float)*ARRAYS);


        double time_start = omp_get_wtime();  //time started


        for( long int i = 0; i < ARRAYS; i++ )
        {
            C[i] = A[i]*B[i];
        }

        double time_end = omp_get_wtime();  //time ended

        double time_consumed = (time_end - time_start) * 1e6;
        printf("%.5lf\t", time_consumed);
        printf("%.5lf\n", (long int)ARRAYS / time_consumed);
    }
        return 0;
}
```

This is the code of non-simd. It calculates the sum of two array and print out the results so I can compare it with the simd results.

2. Result and analysis

Vectorized Array Multiplication and Reduction																	
Arraysize	1024	4096	7168	10240	13312	16384	19456	37888	50176	56320	59392	65536	68608	71680	74752	80896	83968
SIMD+Reduction	109.31712	124.02838	127.56835	127.52397	128.04374	482.18907	863.54732	1015.71513	1044.66663	1045.9088	1042.65438	1032.47724	1042.63121	1020.77998	1039.13236	980.92678	988.97876
SIMD	105.38787	116.73337	120.49442	120.50763	120.78563	302.41673	725.63051	847.77534	876.09643	844.69907	841.76585	845.88002	860.12804	859.28709	828.40266	820.8625	819.38266
NON-SIMD	61.17234	64.72284	65.40819	65.39652	65.45102	96.88608	118.38299	120.0504	120.26604	120.33412	119.13458	119.57108	119.69127	119.55741	119.72213	119.93505	119.94727
NON-SIMD+Reduction	61.64218	65.43933	66.39162	66.44378	66.46386	109.08936	120.48452	122.30555	121.28339	121.69512	121.89328	129.60071	125.60324	125.32597	125.43269	130.71299	128.09362

compare result table



compare results Graph

In the beginning, those four lines of performances are very similar, I think this is because the array size is small, so even the effective calculate cannot better too much. After the array size bigger than 10000, the size is big enough to show the different between them. So the SIMD and SIMD+reduction much better than the other two, because the simd make the process of multiple-data operation in a single instruction. And the nonsimd and nonsimd+reduction already reach the limit. The simd+reduction is faster than the simd is because the reduction makes it more effective. I think the speed up is based on the flip can the hardware, those will make the speed up higher the 4.