**Matrix-Matrix Multiplication using SIMD**

Name: Johannes Christ (#3616)

**I - Objective:**

The main goal of this project is to perform matrix-matrix multiplication of different sizes using 5 different technologies: normal c++ code, c++ code with automatic vectorization, SSE intrinsics instructions, inline SSE assembly instructions for mmx processors, and SSE objects that have operators overloaded.

**II – Approach:**

The main approach consisted in design a code that will break down big arrays into small processable units that can be multiplied together, in order to test the different approaches. The matrix-matrix multiplication consists in multiplying the first arrow of the matrix by the first column (element wise) and add all the values together in order to calculate the first element of the resulting array, do the same with the second column and so on. After this, do this recursively for each row to generate the rest of the elements. We have to cut this process in small parts in order to be able to try the SSE parallelization technology. So in this project we had bring down the amount of elements that need to be multiplied to 4. Then we can test the performance of doing this floating-point calculation using the five different methods mentioned. This is the main piece of code that is responsible for calculating the different elements of the resulting array and breaking down the amount of elements to be calculated at the same time to 4:

**void mmul\_cpp(const float \*a, const float \*b, float \*r, int size)**

**{**

**float result = 0;**

**int rcounter = 0;**

**for (int i = 0; i < size; i++){/// this will iterate thru every row**

**int index = 0;**

**float\* arraya = new float[size];**

**for (int row = size\*i; row <= (size\*(i+1)) -1; row++){**

**arraya[index] = a[row];**

**index++;**

**}**

**for (int kk = 0; kk< size; kk++){//this will itirate over every column**

**float\* arrayb = new float[size];**

**int index2 = 0;**

**for (int column = kk; column <= ((size\*size)-size) + kk; column+=size){**

**arrayb[index2] = b[column];**

**index2++;**

**}**

**result = 0;**

**////insert SSE code right here...where multiplying both vectors...4 at the time...**

**float number = 0;**

**//storing vectors for processing...(4 in lenght...)**

**for (int sind = 0; sind < size; sind+=4){**

**float\* a1 = new float[4];**

**float\* b1 = new float[4];**

**float\* results = new float[4];**

**//calculate each entry...using c++ code only.**

**for (int i = 0; i < 4; i++){**

**a1[i] = arraya[sind+i];**

**b1[i] = arrayb[sind+i];**

**results[i] = a1[i] \* b1[i];**

**number = number + results[i];**

**}**

**}**

**r[rcounter] = number;**

**rcounter++;**

**}**

**}**

**}**

The code that is the rectangle in computationally intensive and this code can be improved by using SSE technology.

**III – Performance of C++ code:**

The performance varies greatly according to which method is being used. If we only use the c++ approach then the computation has no gain from vectorization and shows the slowest performance using the **QueryPerformanceCounter Tool** provided by Microsoft. For an array-array multiplication of size 336 it needed approximate 10 seconds. The following chart shows the amount of time needed to multiply arrays of different sizes:

The main advantages are that this approach to solve the problem is straight forward but definitely not efficient.

**IV- Performance of C++ with Automatic Vectorization:**

There are many factors that are involved in automatic vectorization. In our example after turning on automatic vectorization in VS2012 ( Project -> project properties -> c/c++ -> code generation -> Enable Paralell code Generation -> Qpar) we have seen no increase in performance even given the fact that the loop is clearly done 4 times to multiply each value of the vector array. This approach is really easy to do, because it doesn’t require a modification of the code, but as we see in this code is not always reliable. Following is a chart presenting the time needed for calculating matrix-matrix multiplication of different sizes:

**V – Performance using SSE Objects:**

Next in our discussion we have the performance gain obtained by using the SSE objects provided in the header fvec.h. In this case we see a real gain in performance but working at its best. For this example we have generated the following code:

float number = 0;

for (int sind = 0; sind < size; sind+=4){

F32vec4 \*av=(F32vec4 \*) &arraya[sind];

F32vec4 \*bv=(F32vec4 \*) &arrayb[sind];

F32vec4 \*result=(F32vec4 \*) results;

\*result=\*av \* \*bv;

int i;

for (i = 0; i < 4; i++) number = number + results[i];

}

r[rcounter] = number;

rcounter++;

As we can see here the code generated uses the F32vec4 object that has the = and \* operators overloaded in order to provide an easy implementation of the code. As a result there is a decrease in performance, but generally a lot better that the c++ code only:

As we see here there is a great gain in overall performance, but not as good as the following two methods.

**VI – Performance using SSE Intrinsics:**

In this case we have used the SSE Intrinsics in order to obtain a big gain in performance. In practice this is one of the best tools to obtain maximum performance. The main problem was seen in the fact that it sometimes generate problems with the data alignment. In our example there are instances in which the code with generate segfaults but other instances in which it will not (using the same variables). The following code was generated for this purpose:

float number = 0;

for (int sind = 0; sind < size; sind+=4){

t0 = \_mm\_set\_ps(arraya[sind+3], arraya[sind+2], arraya[sind+1], arraya[sind+0]); //load aligned elements of float array...

t1 = \_mm\_set\_ps(arrayb[sind+3], arrayb[sind+2], arrayb[sind+1], arrayb[sind+0]);

t0 = \_mm\_mul\_ps(t0, t1); //multiply vectors...

\_mm\_store\_ps(results, t0); //store in resulting vector...

//compute the result for the entry..

int i;

for (i = -0; i < 4; i++) number = number + results[i];

}

This code generated top performance in our calculations:

**VII – Performance using SSE in line assembly instructions:**

This was in practice the fastest and more reliable approach to obtain the correct result for the calculations. Coding inline assembly gave total control over the operations and also ensure that mmx registers are accessed only when needed since all the code is manually written. It generates almost the same performance as SSE intrinsics:

float number = 0;

for (int sind = 0; sind < size; sind+=4){

float vectorIn1[] = {arraya[sind+0], arraya[sind+1], arraya[sind+2], arraya[sind+3]};

float vectorIn2[] = {arrayb[sind+0], arrayb[sind+1], arrayb[sind+2], arrayb[sind+3]};

float results[4] = {0.0, 0.0, 0.0, 0.0};

\_\_asm

{

movups xmm0, [vectorIn1] //move arrays into mmx's

movups xmm1, [vectorIn2]

mulps xmm0, xmm1 //multiply packed single fp #'s

movups [results], xmm0 //store xmm0 into results[]

}

int i;

for (i = 0; i < 4; i++) number = number + results[i];

}

r[rcounter] = number;

rcounter++;

**VIII – Conclusion:**

As we were able to appreciate, the utilization of SIMD technology makes a great impact on the performance of the software. There is a great increase in performance comparing the code generated by using only c++, and the one using SSE instructions, specially intrinsics and inline assembly.

**IX – Appendix:**

Attached to this document is the table generated with the different values for the 5 methods in comparison, as well as a graph comparing the performance between them.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | C++ no SSE | SSE Intrinsics | SSE Assembly | C++ Auto Vectoriz. | SSE Objects in C++ |
| 16 | 1.72E-03 | 2.90E-04 | 2.71E-04 | 1.05E-03 | 3.31E-04 |
| 32 | 8.87E-03 | 9.48E-04 | 9.47E-04 | 7.69E-03 | 2.19E-03 |
| 48 | 3.01E-02 | 2.67E-03 | 2.75E-03 | 2.69E-02 | 6.86E-03 |
| 64 | 6.64E-02 | 5.70E-03 | 5.38E-03 | 6.32E-02 | 1.61E-02 |
| 80 | 1.31E-01 | 1.02E-02 | 9.93E-03 | 1.24E-01 | 3.08E-02 |
| 96 | 2.40E-01 | 1.69E-02 | 1.57E-02 | 2.18E-01 | 5.41E-02 |
| 112 | 3.54E-01 | 2.58E-02 | 2.40E-02 | 3.54E-01 | 8.55E-02 |
| 128 | 5.97E-01 | 4.12E-02 | 3.83E-02 | 5.42E-01 | 1.28E-01 |
| 144 | 7.78E-01 | 5.45E-02 | 4.99E-02 | 7.77E-01 | 1.79E-01 |
| 176 | 1.41E+00 | 1.07E-01 | 9.29E-02 | 1.39E+00 | 3.31E-01 |
| 208 | 2.31E+00 | 1.70E-01 | 1.50E-01 | 2.27E+00 | 5.44E-01 |
| 272 | 5.22E+00 | 3.60E-01 | 3.37E-01 | 5.20E+00 | 1.20E+00 |
| 336 | 1.00E+01 | 7.20E-01 | 6.44E-01 | 1.03E+01 | 2.28E+00 |