



**Cairo University**

**Faculty of Engineering**

**Third Year – 2016/2017**

MileStone1 Report

***Submitted by:***

***Sec:* 1*, BN:* 25*, Name:* Hanna Nabil**

***Submitted to :***

**Dr. Meena AbdelMaseeh**

**Code :**

for(int i=0;i<MAX\_DIM;i+=1){

for(int j=0;j<MAX\_DIM;j+=4){

for(int k=0;k<MAX\_DIM;k+=4){

\_\_m128 result = \_mm\_load\_ps(&d[i][j]);

\_\_m128 a\_line = \_mm\_load\_ps(&a[i][k]);

\_\_m128 b\_line0 = \_mm\_load\_ps(&b[k][j+0]);

\_\_m128 b\_line1 = \_mm\_loadu\_ps(&b[k][j+1]);

\_\_m128 b\_line2 = \_mm\_loadu\_ps(&b[k][j+2]);

\_\_m128 b\_line3 = \_mm\_loadu\_ps(&b[k][j+3]);

result = \_mm\_add\_ps(result, \_mm\_mul\_ps(\_mm\_shuffle\_ps(a\_line, a\_line, 0x00), b\_line0));

result = \_mm\_add\_ps(result, \_mm\_mul\_ps(\_mm\_shuffle\_ps(a\_line, a\_line, 0x55), b\_line1));

result = \_mm\_add\_ps(result, \_mm\_mul\_ps(\_mm\_shuffle\_ps(a\_line, a\_line, 0xaa), b\_line2));

result = \_mm\_add\_ps(result, \_mm\_mul\_ps(\_mm\_shuffle\_ps(a\_line, a\_line, 0xff), b\_line3));

\_mm\_store\_ps(&d[i][j],result);

}

}

}

**Scalar Vs Vector (Single Precision) :**

Performance Gain obtained by vector parallelization, using MAX\_DIM = 100

Is 3 Times Faster than the Unoptimized Matrix Multiplication

So When

MAX\_DIM =100 ==🡺 Performance Gain = 4.22

MAX\_DIM =200 ==🡺 Performance Gain = 3.92

MAX\_DIM =300 ==🡺 Performance Gain = 4.21

MAX\_DIM =400 ==🡺 Performance Gain = 4.12

MAX\_DIM =500 ==🡺 Performance Gain = 4.89

MAX\_DIM =600 ==🡺 Performance Gain = 4.37

MAX\_DIM =700 ==🡺 Performance Gain = 5.16

**Double Precision :**

When using Double Precision I obtained the following performance Gain

MAX\_DIM =100 ==🡺 Performance Gain = 1.94 Faster than Scalar

MAX\_DIM =200 ==🡺 Performance Gain = 1.86

MAX\_DIM =300 ==🡺 Performance Gain = 1.9

MAX\_DIM =400 ==🡺 Performance Gain = 2.63

MAX\_DIM =500 ==🡺 Performance Gain = 2.6

**Using Vector Parallelism can be useful at :**

-One processor (the master) was programmed to be responsible for all of the work in the system; the other (the slave) performed only those tasks it was assigned by the master.

-The most successful applications have been for problems that can be broken down into many separate, independent operations on vast quantities of data.

-In [data mining](https://searchsqlserver.techtarget.com/definition/data-mining), there is a need to perform multiple searches of a static database.

-In [artificial intelligence](https://searchcio.techtarget.com/definition/AI), there is the need to analyze multiple alternatives, as in a chess game. Often systems are structured as clusters of processors

**Intrinsic Function I used :**

|  |  |
| --- | --- |
| \_\_m128 \_mm\_load\_ps(float \*src) | Load 4 floats from a 16-byte aligned address.  WARNING: Segfaults if the address isn't a multiple of 16! |
| \_\_m128 \_mm\_loadu\_ps(float \*src) | Load 4 floats from an unaligned address (about 4x slower!) |
| void \_mm\_store\_ps(float \*dest,\_\_m128 src) | Store 4 floats to an aligned address. |
| \_\_m128 \_mm\_add\_ps(\_\_m128 a,\_\_m128 b) | Add corresponding floats (also "sub") |
| \_\_m128 \_mm\_mul\_ps(\_\_m128 a,\_\_m128 b) | Multiply corresponding floats (also "div", but it's slow) |
| \_\_m128 \_mm\_shuffle\_ps(\_\_m128 lo,\_\_m128 hi,        \_MM\_SHUFFLE(hi3,hi2,lo1,lo0)) | Interleave inputs into low 2 floats and high 2 floats of output. Basically    out[0]=lo[lo0];    out[1]=lo[lo1];    out[2]=hi[hi2];    out[3]=hi[hi3]; For example, \_mm\_shuffle\_ps(a,a,\_MM\_SHUFFLE(i,i,i,i)) copies the float a[i] into all 4 output floats. |