Weifan Lin

Computer Science

Csc342 Section G

04/23/2015

Take-Home Exam

Title:

Matrix Multiplication

Objective:

The goal of this take-home exam was to implement matrix multiplication in different ways and to compare their performance. Firstly to create a simple c function to compute matrix-matrix multiplication, then to use DPPS instruction to improve performance. In addition, I also had some functions to compute the time of different performances for time analysis at the end.

C++ function to compute matrix multiplication

```
// function written in C++ language
float* MatrixVector_C(float* matrix, float* vector, int size)
{
    float *vectorResult=(float*) malloc (size*sizeof(float));
    float* matrix_ptr=matrix;

    for (int i=0; i<size; i++)
    {
        float* vector_ptr=vector;

        float Result=0.0;
        for (int j=0; j<size; j++)
        {
            Result+=(*matrix_ptr)*(*vector_ptr);
            vector_ptr++;
            matrix_ptr++;
        }
        vectorResult[i]=Result;
    }
    return (vectorResult);
}
```

Use DPPS instruction to Improve performance

```
// function written in assembler (SSE)
float* MatrixVector_SSE(float* refmatrixA,float* refvectorB, int size)
{

    int _length=size;
    int length1=size;
    int _nCol=size;
    int _nRow=size;

    float *vect=refvectorB;
    float *vect1=(float*) malloc (size*sizeof(float));
    float *matr=refmatrixA;

    int stride1=_nCol<<2;
    int stride2=stride1<<1;
    int stride3=stride2+stride1;
    int stride4=_nCol<<4;

    int nbre=_nRow>>2;
```

```
    int nbre1=_nCol>>2;
    int nbre2=nbre1;

    int resteRow=_nRow%4;
    int resteCol=_nCol%4;
    int nRow1=_nRow;
    int nRow2=_nRow;
    int nCol1=_nCol;

__declspec(align(16)) float valeur1[4];
__declspec(align(16)) float valeur2[4];
__declspec(align(16)) float valeur3[4];
__declspec(align(16)) float valeur4[4];

    float* derniersLigne=matr+(stride1>>2)-resteCol;
    float* derniersVect=vect-resteCol+length1;
    float zero=0;
    int stride=stride1-(resteCol<<2);
    nRow1-=resteRow;

// Is the matrix dimension bigger than 4x4?
    __asm
    {
        push    ebx             ; save EBX (warning :frame pointer register)
                                ; we save the data in EBX to avoid bugsdue to warnings C4731

        // Is the matrix dimension bigger than 3?
        cmp     _nCol, 3
        jle     inferieur
        cmp     _nRow, 3
        jle     inferieur

        /* Init of the result vector (all values to zero) */
        xorps   xmm0,xmm0
        mov     edi,vect1
        mov     eax,nbre
        mov     ecx, resteRow

init:   movups  [edi],xmm0
        add     edi,16
        dec     eax
        jnz     init

        cmp     ecx, 0
        je      debut
        // Init the end of the result vector if its size is not a multiple of 4
```

```
init_fin:
        movss   [edi], xmm0
        add     edi, 4
        dec     ecx
        jnz     init_fin

debut:
        // Init vectors' pointers
        mov         esi, vect
        mov         edi, vect1

        // Init matrix pointer and strides
        mov     ebx, matr
        mov     eax, stride1
        mov     ecx, stride2
        mov     edx, stride3

// the matrix is cutted in little matrix 4x4
// and the vector in little vector 4x1

ligne:  push    ebx     ; save ebx in the stack
        push    nbre    ; save nbre in the stack
        push    edi     ; save edi in the stack

        movups  xmm0, [esi]

        movaps  xmm1, xmm0
        movaps  xmm2, xmm0
        movaps  xmm3, xmm0
        shufps  xmm0, xmm0, 0x00
        shufps  xmm1, xmm1, 0x55
        shufps  xmm2, xmm2, 0xAA
        shufps  xmm3, xmm3, 0xFF

        movaps  valeur1, xmm0
        movaps  valeur2, xmm1
        movaps  valeur3, xmm2
        movaps  valeur4, xmm3

        // the vector 4x1 is loaded in the XMM0,XMM1,XMM2,XMM3

colonne:

        // Now we transpose the matrix in order to perform operations

        movlps  xmm4, [ebx]
```

```
movlps  xmm6, [ebx+0x08]
movlps  xmm3, [ebx+ecx]     // ->2*stride
movlps  xmm2, [ebx+ecx+8]   // ->2*stride+8

movhps  xmm4, [ebx+eax]     // ->stride
movhps  xmm6, [ebx+eax+8]   // ->stride+8
movhps  xmm3, [ebx+edx]     // ->3*stride
movhps  xmm2, [ebx+edx+8]   // ->3*stride+8


movaps  xmm5, xmm4
movaps  xmm7, xmm6
shufps  xmm4, xmm3, 0x88
shufps  xmm6, xmm2, 0x88

shufps  xmm5, xmm3, 0xDD
shufps  xmm7, xmm2, 0xDD

// we have the matrix transpose in registers XMM4 to XMM7

// load vector 4x1 values in  XMM0 to XMM3

movaps  xmm0, valeur1
movaps  xmm1, valeur2
movaps  xmm2, valeur3
movaps  xmm3, valeur4

// perform multiplication between matrix values and vector values
mulps      xmm0, xmm4
mulps      xmm1, xmm5
mulps      xmm2, xmm6
mulps      xmm3, xmm7

// perform addition
addps      xmm1, xmm0
addps      xmm2, xmm3
addps      xmm1, xmm2

movups  xmm2, [edi]
addps   xmm1, xmm2

/* the result (XMM1) is saved in the result vector */
movups  [edi], xmm1

// Next little matrix 4x4 and next vector 4x1
add     ebx, stride4
```

```
        add     edi, 16
        dec     nbre
        jnz     colonne

        pop     edi
        pop     nbre
        pop     ebx
        add     ebx, 16
        add     esi, 16

        dec     nbre1
        jnz     ligne

    // This section of the function performs the last values of a row
    // when the matrix dimension is not a multiple of 4
        cmp     resteCol, 0
        jz      fin
        mov     eax, nRow2
        mov     esi, derniersLigne
        mov     edi, vect1

ligne_suivante:

        mov     ecx, resteCol
        mov     ebx, derniersVect

        movss   xmm2, zero
fin_ligne:
        movss   xmm0, [esi]
        movss   xmm1, [ebx]
        mulss   xmm0, xmm1
        add     esi, 4
        add     ebx, 4
        addss   xmm2, xmm0
        dec     ecx
        jnz     fin_ligne
        movss   xmm3, [edi]
        addss   xmm3, xmm2
        movss   [edi], xmm3
        add     edi, 4
        add     esi, stride
        dec     eax
        jnz     ligne_suivante

fin:    cmp     resteRow, 0
        je      end
```

```
   }

     // Pointers used by the function to perfom the last coefficients of the result vector
     // used when the matrix dimension is not a multiple of 4
   float* dernieres_lignes=matr+(nRow2-resteRow)*nCol1;
   float* derniers_coeff=vect1+nRow2-resteRow;
   int resteCol4=resteCol<<2;

   __asm
   {
      mov     esi, dernieres_lignes
      mov     ebx, vect
      mov     edi, derniers_coeff
      mov     eax, resteRow
      mov     ecx, nbre2

debut_ligne:
      movss   xmm2, zero

prochains_coeff:

      movups  xmm0, [esi]
      movups  xmm1, [ebx]
      mulps   xmm0, xmm1
      movaps  xmm3,xmm0
      movaps  xmm4,xmm0
      movaps  xmm5,xmm0
      shufps  xmm3, xmm0, 0x55
      shufps  xmm4, xmm0, 0xAA
      shufps  xmm5, xmm0, 0xFF
      addps   xmm0, xmm3
      addps   xmm0, xmm4
      addps   xmm0, xmm5
      addps   xmm2, xmm0
      add     esi, 16
      add     ebx, 16
      dec     ecx
      jnz     prochains_coeff
      movss   xmm7, [edi]
      addss   xmm2, xmm7
      movss   [edi], xmm2
      add     edi, 4
      add     esi, resteCol4
      mov     ebx, vect
      mov     ecx, nbre2
      dec     eax
```

```
        jnz     debut_ligne

    }

// When the matrix dimension is lower than 4

    __asm
    {
        jmp     end

inferieur:
        mov     esi, matr
        mov     edi, vect1
        mov     ebx, vect
        mov     eax, nRow2
        mov     edx, nCol1

mise_zero:
        movss   xmm2, zero

lignes:
        movss   xmm0, [esi]
        movss   xmm1, [ebx]
        mulss   xmm0, xmm1
        add     ebx, 4
        add     esi, 4
        addss   xmm2, xmm0
        dec     edx
        jnz     lignes
        movss   [edi], xmm2
        add     edi, 4
        mov     ebx, vect
        mov     edx, nCol1
        dec     eax
        jnz     mise_zero
end:
        pop     ebx
    }
    return (vect1);
}
```

Functions For Time Analysis:

```
// Read the computer's timer RDTSC
__int64 GetTime()
{
    __int64 clock;

    __asm
    {
        rdtsc           // Resad the RDTSC Timer
        mov     dword ptr[clock], eax       // Store the value in EAX and EDX
        mov     dword ptr[clock+4], edx

    }
    return clock;
}

// Perform and display the time saved or lost between the C++ and assembly
void TimeImprove(__int64 timeC,__int64 timeSSE)
{
    float gain=0;

    if(timeC>timeSSE)
    {
        gain=(1-((float)timeSSE/(float)timeC))*100;
        printf("\nTime saved: %f %% \n",gain);

    }
    if(timeC==timeSSE)
    {
        printf("\nTemps saved: 0 %%\n");

    }
    if(timeC<timeSSE)
    {
        gain=((float)timeSSE/(float)timeC)*100-100;
        printf("\nTemps lost: %f %% \n ", gain);
    }
}
```

Comparison

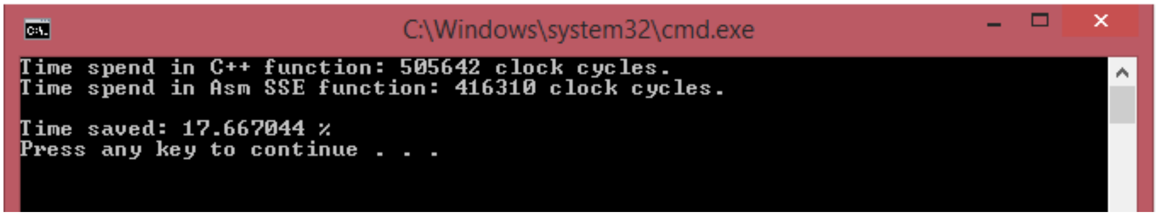
```
// The main contain the call to the functions, declarations, and time measurement
int main(int argc, char* argv[])
{
    //Enter the size of the matrix
    int size = 1024;
    int i;

    // Allocate memory
    float* matrix=(float*) malloc (size*size*sizeof(float));
    float* vector=(float*) malloc (size*sizeof(float));
    float* result=(float*) malloc (size*sizeof(float));

    float* matrix1=(float*) malloc (size*size*sizeof(float));
    for(i=0; i<size*size; i++)
        matrix1[i]=(float)i;

    // Writing values in the matrix and vector
    MatrixVectorWriting(matrix, vector, size);

    // Benchmark the two functions
    __int64 t1=GetTime();
    for(i=0; i<100; i++)
        result=MatrixVector_C(matrix,vector, size);
    __int64 t2=GetTime();
    __int64 time_C=t2-t1;
    printf("Time spend in C++ function: %d clock cycles.\n",time_C);

    __int64 t3=GetTime();
    for(i=0; i<100; i++)
        result=MatrixVector_SSE(matrix1, vector, size);
    __int64 t4=GetTime();
    __int64 time_SSE=t4-t3;
    printf("Time spend in Asm SSE function: %d clock cycles.\n",time_SSE);

    // Display the time improvement in percent
    TimeImprove(time_C,time_SSE);
  system("pause");
    return 0;
}
```

The variable size in the main function is the size of matrix, we can change it manually to compare different performances of matrix multiplication in different sizes.

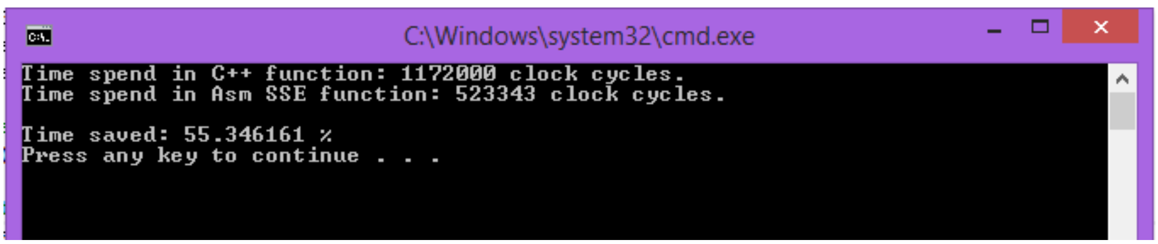
Time Analysis

I set the size of matrix from 8x8 to 512x512, below are result of each one of them

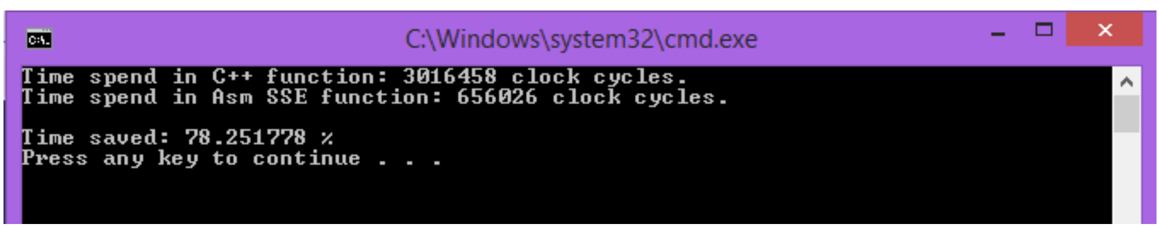
size=8x8:



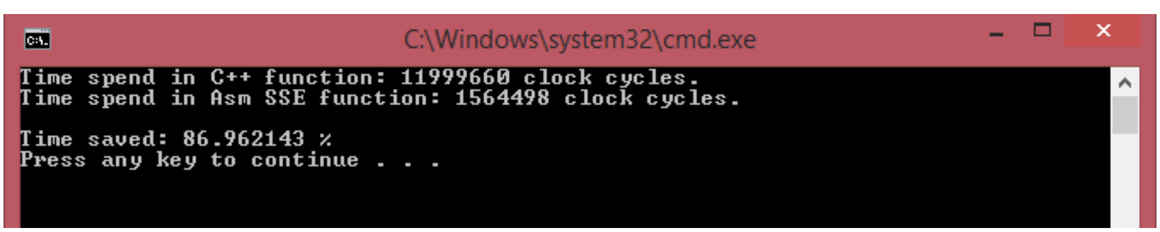
size=16x16:



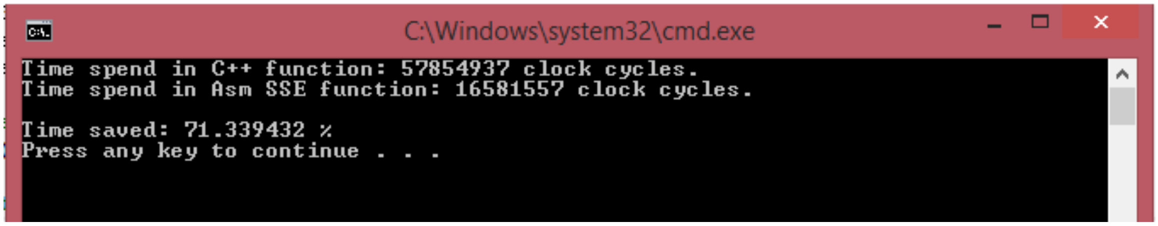
size=32x32:



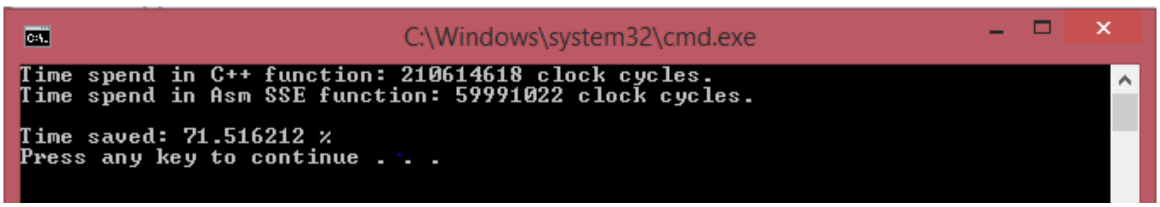
size=64x64:



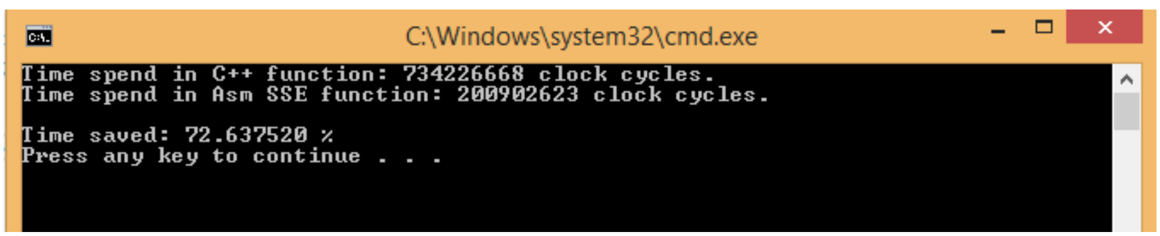
size=128x128:



size=256x256:



size=512x512:



Conclusion

As we can see the results from the time analysis, I have shown that for a large number of repeated operations, the use of vector processing is much faster than single instruction processing. Overall, this take-home exam helped me implement matrix multiplication in single instruction processing and in vector processing. Through the experiment I saw the performance of vector processing is faster as the size of matrix increases.