Take Home Test 4:
Matrix Multiplication

Shirong Zheng
Professor Izidor Gertner
April 19, 2017
CSC34200-G Spring 2017

Contents

❖1.Objective

The objective of take home test 4 is going to using the assembly code and C++ code which provide by Microsoft Visual Studio in 32 bit Window(debugger analysis) and GDB/GCC in 64 bit LINUX compiler. This topic is matrix multiplication, which we need to Write C++ function to compute Matrix-Matrix multiplication in Visual Studio environment. Matrix sizes should be powers of 2 (e.g. 16x16, 32x32, 64x64, ...512x512). Disable Automatic Parallelization, /Qpar, and Automatic Vectorization, /arch. Use QueryPerformanceCounter function to measure execution time.

❖2. SECTION 1 MS Visual Studio CPUID Instruction

The below is my matrix multiplication function. There are three static float, S will be be equal to the multiplication of size A and size B.

```
#include "stdafx.h"
#include<Windows.h>
#include<iostream>
using namespace std;

static const int size=8;
static float A[size][size],B[size][size],S[size][size];

void MatrixMultiplication(){
    for (int i=0; i<size; i++){
        for (int j=0;j<size;j++){
            for (int k=0;k<size;k++){
                S[i][j] += A[i][k] *B[k][j];
            }
        }
    }
}
```

It need to using the Windows.h library, then could successful using the QueryPerformanceCounter and QueryPerformanceFrequency to counting the exactly time base on the microseconds. QueryPerformanceCounter function Retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements. QueryPerformanceCounter function Retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements. Syntax C++ BOOL WINAPI QueryPerformanceCounter(_Out_ LARGE_INTEGER *lpPerformanceCount); Parameters lpPerformanceCount [out] A pointer to a variable that receives the current performance-counter value, in counts. Return value If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To

get extended error information, call GetLastError. On systems that run Windows XP or later, the function will always succeed and will thus never return zero. Retrieves the frequency of the performance counter. The frequency of the performance counter is fixed at system boot and is consistent across all processors. Therefore, the frequency need only be queried upon application initialization, and the result can be cached. A pointer to a variable that receives the current performance-counter frequency, in counts per second. If the installed hardware doesn't support a high-resolution performance counter, this parameter can be zero (this will not occur on systems that run Windows XP or later).

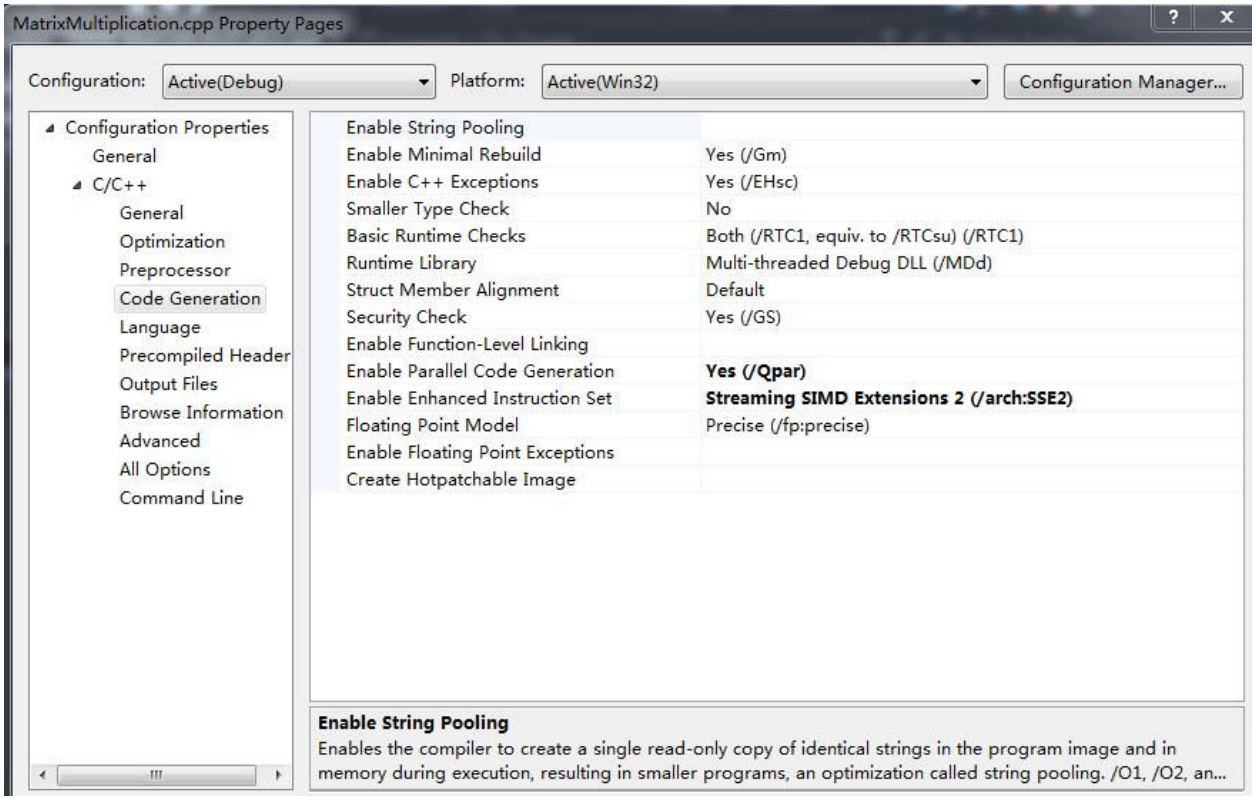
Then we using the formula " ((ctr2-ctr1)*1.0/freq) *1000000" to find the previous values in microseconds.

```
int main()
{
    _int64 ctr1=0, ctr2=0, freq=0;

    for(int i=0; i<size;i++){
        for(int j=0; j<size; j++){
            A[i][j]=0.0;
            B[i][j]=0.0;
            S[i][j]=0.0;
        }
    }

    if(QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) !=0){
        MatrixMultiplication();
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
        cout<<"Start Value:"<<ctr1<<endl;
        cout<<"End Value:"<<ctr2<<endl;
        QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
        cout<<"QueryPerformanceCounter minimum resolution: 1/" <<freq<<"Seconds,"<<endl;
        cout<<size<<"x"<<size<<"Matrix Multiplication:"<<((ctr2-ctr1)*1.0/freq) *1000000<<"Microseconds."<<endl;
    }
    else
    {
        DWORD dwError =GetLastError();
        cout<<"Error Value"<<dwError<<endl;
    }
    system("PAUSE");
    return 0;
}
```

First edit the property of the cpp file. Under the Code Generation part, Enable Function-Level Linking should be Yes and Enable Enhanced Instruction Set should be Streaming SIMD Extensions 2.



The "dword ptr" part is called a size directive. This page explains them, but it wasn't possible to direct-link to the correct section. Basically, it means "the size of the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the

contents of the ebp register and subtracting four with 0. It is a 32bit declaration. If you type at the top of an assembly file the statement [bits 32], then you don't need to type DWORD PTR.

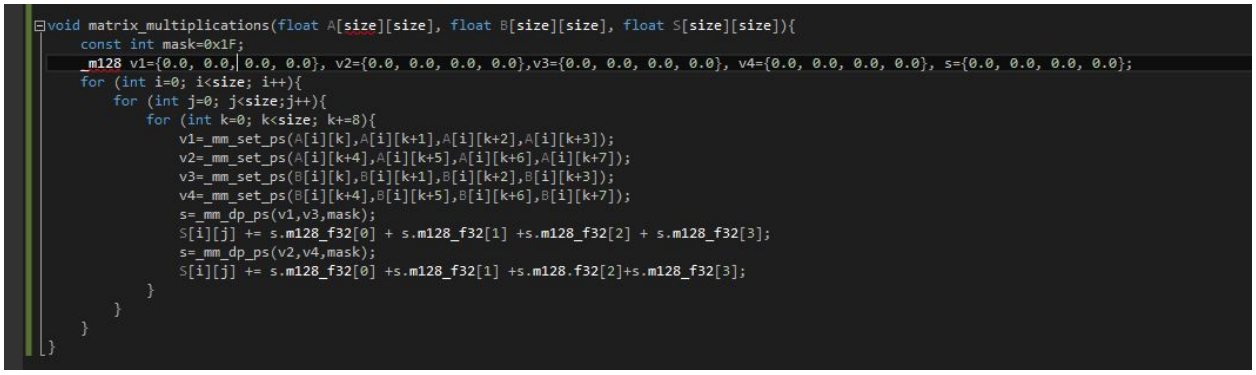
```
; 11   :               S[i][j] += A[i][k] * B[k][j];

    mov eax, DWORD PTR _i$2537[ebp]
    shl eax, 5
    mov ecx, DWORD PTR _i$2537[ebp]
    shl ecx, 5
    mov edx, DWORD PTR _k$2545[ebp]
    shl edx, 5
    mov esi, DWORD PTR _k$2545[ebp]
    fld DWORD PTR _A[ecx+esi*4]
    mov ecx, DWORD PTR _j$2541[ebp]
    fmul    DWORD PTR _B[edx+ecx*4]
    mov edx, DWORD PTR _j$2541[ebp]
    fadd    DWORD PTR _S[eax+edx*4]
    mov eax, DWORD PTR _i$2537[ebp]
    shl eax, 5
    mov ecx, DWORD PTR _j$2541[ebp]
    fstp    DWORD PTR _S[eax+ecx*4]

; 12   :           }
```

❖3. SECTION 2 MS Visual Studio DPPS Instruction

In the DPPS instruction, which is Dot Product of Packed Single Precision Floating-Point Values. Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1.



```
void matrix_multiplications(float A[size][size], float B[size][size], float S[size][size]){
    const int mask=0x1F;
    __m128 v1={0.0, 0.0, 0.0, 0.0}, v2={0.0, 0.0, 0.0, 0.0},v3={0.0, 0.0, 0.0, 0.0}, v4={0.0, 0.0, 0.0, 0.0}, s={0.0, 0.0, 0.0, 0.0};
    for (int i=0; i<size; i++){
        for (int j=0; j<size;j++){
            for (int k=0; k<size; k+=8){
                v1=_mm_set_ps(A[i][k],A[i][k+1],A[i][k+2],A[i][k+3]);
                v2=_mm_set_ps(A[i][k+4],A[i][k+5],A[i][k+6],A[i][k+7]);
                v3=_mm_set_ps(B[i][k],B[i][k+1],B[i][k+2],B[i][k+3]);
                v4=_mm_set_ps(B[i][k+4],B[i][k+5],B[i][k+6],B[i][k+7]);
                s=_mm_dp_ps(v1,v3,mask);
                S[i][j] += s.m128_f32[0] + s.m128_f32[1] +s.m128_f32[2] + s.m128_f32[3];
                s=_mm_dp_ps(v2,v4,mask);
                S[i][j] += s.m128_f32[0] +s.m128_f32[1] +s.m128.f32[2]+s.m128_f32[3];
            }
        }
    }
}
```

The Op/En is RVMI, the 64/32-bit Mode is v/v, the CPUID Feature Flag is AVX. CVTPS2PD is Convert Packed Single-Precision FP Values to Packed Double-Precision FP Values. Converts two or four packed single-precision floating-point values in the source operand (second operand) to two or four packed double-precision floating-point values in the destination operand (first operand). In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15). 128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operation is an XMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are unmodified. VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operation is a YMM register. The upper bits (VLMAX-1:128) of the corresponding YMM register destination are zeroed. VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operation is a YMM

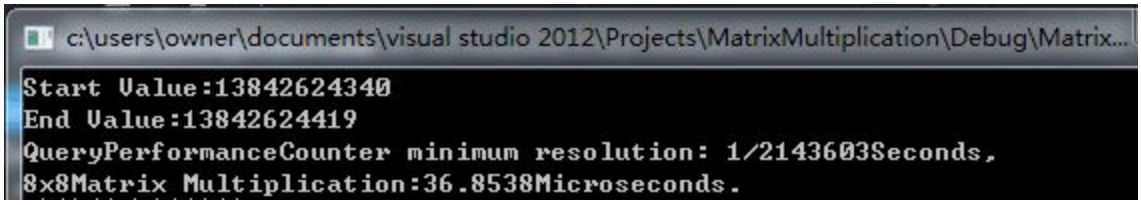
register. Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

```
; 35   :
; 36   :               S[i][j] += s.m128_f32[0] + s.m128_f32[1] + s.m128_f32[2] + s.m128_f32[3];

   mov eax, DWORD PTR _i$4451[ebp]
   shl eax, 5
   add eax, DWORD PTR _S$[ebx]
   movss   xmm0, DWORD PTR _s$[ebp]
   cvtps2pd xmm0, xmm0
   movss   xmm1, DWORD PTR _s$[ebp+4]
   cvtps2pd xmm1, xmm1
   addsd   xmm0, xmm1
   movss   xmm1, DWORD PTR _s$[ebp+8]
   cvtps2pd xmm1, xmm1
   addsd   xmm0, xmm1
   movss   xmm1, DWORD PTR _s$[ebp+12]
   cvtps2pd xmm1, xmm1
   addsd   xmm0, xmm1
   mov ecx, DWORD PTR _j$4455[ebp]
   movss   xmm1, DWORD PTR [eax+ecx*4]
   cvtps2pd xmm1, xmm1
   addsd   xmm1, xmm0
   mov edx, DWORD PTR _i$4451[ebp]
   shl edx, 5
   add edx, DWORD PTR _S$[ebx]
   xorps   xmm0, xmm0
   cvtsd2ss xmm0, xmm1
   mov eax, DWORD PTR _j$4455[ebp]
   movss   DWORD PTR [edx+eax*4], xmm0
```

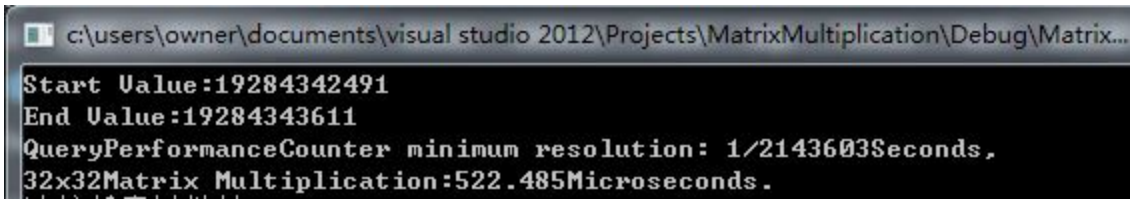
❖4. Running Time

8x8 Matrix

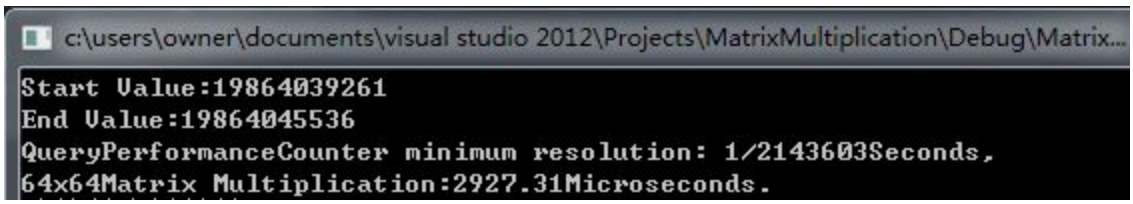


16x16 Matrix

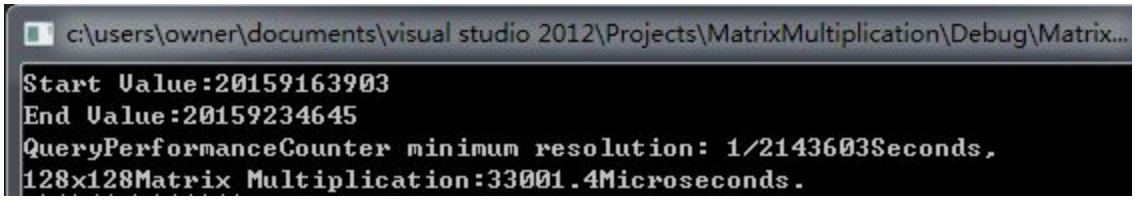
32x32 Matrix



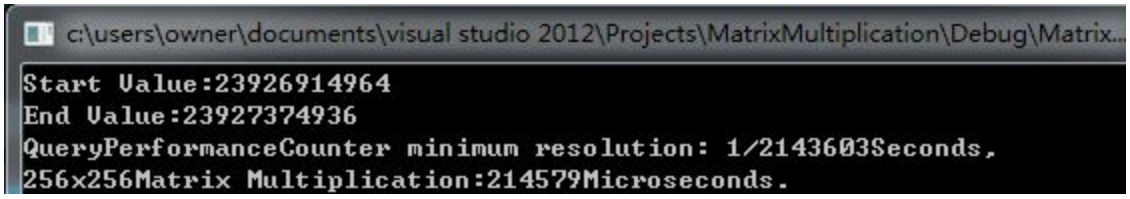
64x64 Matrix



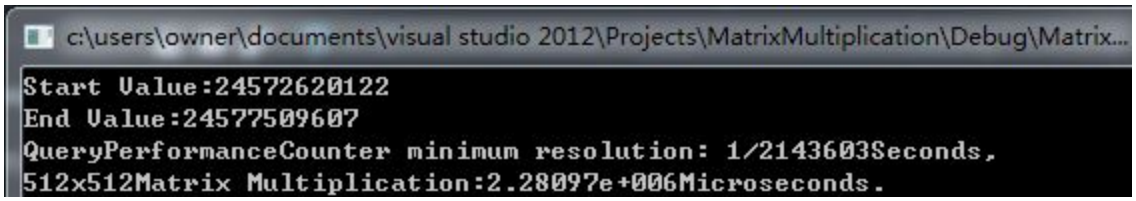
128x128 Matrix



256x256 Matrix



512x512 Matrix



They have same QueryPerformanceCounter minimum resolution time. The Start Value and End Value will has much higher depend on the the larger number of matrix.

❖5. Conclusions

In the conclusion, I learned how to debugging CPUID and DPPS instructions in MS Visual Studio. This take home test is going to run/debug the matrix multiplications. We should to compare the similarity and difference of all threes. In the end, this assignment required to find out the cost time of each matrix. I spent more than 10 hours to finish all the sections. In the suggestion, it is better to do the efficiently work in less time.