Matrix Multiplication, Compiler Generated Code Optimization

Brandon Chin

CSC342 - Instructor: Prof. Izidor Gertner

4/23/2015

Objective:

We are going to create compiler generated assembly code for a matrix multiplication function. Then we will modify this compiler generated assembly code using three different optimization techniques. First without any optimization, then with automatic parallelization and vectorization, and finally with DPPS instruction. In the end, we will organize and compare our results after execution of all methods.

Matrix Multiplication Function

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

void matrix_multiplication();
```

matrix_multiplication.h

```
#include "matrix_multiplication.h"

void matrix_multiplication()
{
    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];
            }
        }
    }
}
```

matrix_multiplication.cpp

Here, we can see the matrix multiplication function written C++. The function computes the product of two square matrices. We will start with two 8x8 matrices, where the size of the matrices will go up by powers of 2, for example, 8x8, 16x16, 32x32, 64x64, etc. Each element of the product matrix S is equal to the summation of the corresponding row of A multiplied by the corresponding column of B.

Un-optimized Compilation

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

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

    // initialize arrays
    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)
    {
        matrix_multiplication();        // call function
        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 << " Maxtrix Multiplication: " << ((ctr2 - ctr1)* 1.0 / freq) * 1000000 << "Microseconds." << endl;
    }
    else
    {
        DWORD dwError = GetLastError();
        cout << "Error value" << dwError << endl;
    }
    system("PAUSE");

    return 0;
}
```

main.cpp

The main function initializes all of the arrays and calls the matrix multiplication function. We are using the same "QueryPerformanceCounter" function that we used before to time the duration of the multiplication function.

Now let's link the compiler generated assembly code shown below:

```
_TEXT   SEGMENT
_k$2545 = -32                   ; size = 4
_j$2541 = -20                   ; size = 4
_i$2537 = -8                    ; size = 4
?matrix_multiplication@@YAXXZ PROC          ; matrix_multiplication, COMDAT

; 4    : {

    push    ebp
    mov ebp, esp
    sub esp, 228                ; 000000e4H
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-228]
    mov ecx, 57                 ; 00000039H
    mov eax, -858993460             ; ccccccccH
    rep stosd
```

```
; 5    :    for (int i = 0; i < size; i++)

    mov DWORD PTR _i$2537[ebp], 0
    jmp SHORT $LN9@matrix_mul
$LN8@matrix_mul:
    mov eax, DWORD PTR _i$2537[ebp]
    add eax, 1
    mov DWORD PTR _i$2537[ebp], eax
$LN9@matrix_mul:
    cmp DWORD PTR _i$2537[ebp], 8
    jge SHORT $LN7@matrix_mul

; 6    :    {
; 7    :        for (int j = 0; j < size; j++)

    mov DWORD PTR _j$2541[ebp], 0
    jmp SHORT $LN6@matrix_mul
$LN5@matrix_mul:
    mov eax, DWORD PTR _j$2541[ebp]
    add eax, 1
    mov DWORD PTR _j$2541[ebp], eax
$LN6@matrix_mul:
    cmp DWORD PTR _j$2541[ebp], 8
    jge SHORT $LN4@matrix_mul

; 8    :        {

; 9    :            for (int k = 0; k < size; k++)

    mov DWORD PTR _k$2545[ebp], 0
    jmp SHORT $LN3@matrix_mul
$LN2@matrix_mul:
    mov eax, DWORD PTR _k$2545[ebp]
    add eax, 1
    mov DWORD PTR _k$2545[ebp], eax
$LN3@matrix_mul:
    cmp DWORD PTR _k$2545[ebp], 8
    jge SHORT $LN1@matrix_mul

; 10   :            {

; 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   :            }

    jmp SHORT $LN2@matrix_mul
$LN1@matrix_mul:

; 13   :        }

    jmp SHORT $LN5@matrix_mul
$LN4@matrix_mul:
```

```
; 14   :    }

    jmp $LN8@matrix_mul
$LN7@matrix_mul:

; 15   : }

    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?matrix_multiplication@@YAXXZ ENDP          ; matrix_multiplication
_TEXT   ENDS
END
```

matrix_multiplication.asm

After executing this code, we get the following results shown below:

8x8 Matrices:

```
Start Value:2217602354317
End Value:2217602354323
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
8x8 Maxtrix Multiplication: 3.42109Microseconds.
Press any key to continue . . .
```

16x16 Matrices

```
Start Value:2217268365750
End Value:2217268365803
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
16x16 Maxtrix Multiplication: 30.2196Microseconds.
Press any key to continue . . .
```

32x32 Matrices

```
Start Value:2217692117032
End Value:2217692117460
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
32x32 Maxtrix Multiplication: 244.038Microseconds.
Press any key to continue . . .
```

64x64 Matrices

```
Start Value:2217781297281
End Value:2217781299886
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
64x64 Maxtrix Multiplication: 1485.32Microseconds.
Press any key to continue . . .
```

128x128 Matrices

```
Start Value:2217853457565
End Value:2217853478573
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
128x128 Maxtrix Multiplication: 11978.4Microseconds.
Press any key to continue . . .
```

256x256 Matrices

```
Start Value:2218262723369
End Value:2218262906207
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
256x256 Maxtrix Multiplication: 104251Microseconds.
Press any key to continue . . .
```

512x512 Matrices

```
Start Value:2218370626459
End Value:2218372117865
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
512x512 Maxtrix Multiplication: 850372Microseconds.
Press any key to continue . . .
```

Optimized with Automatic Parallelization and Vectorization

To enable these instructions, we must go to Properties -> C/C++ -> Code Generation. Next to "Enable Parallel Code Generation" click "**Yes (/Qpar)**" and next to "Enable Enhanced Instruction Set" click "**Streaming SIMD Extension 2 (/arch:SSE2)**". Now let's recompile the code and notice the xmm registers being used instead:

```
; 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]
```

matrix_multiplication.asm -- line 11 (before)

```
; 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]
    movss   xmm0, DWORD PTR _A[ecx+esi*4]
    cvtps2pd xmm0, xmm0
    mov ecx, DWORD PTR _j$2541[ebp]
    movss   xmm1, DWORD PTR _B[edx+ecx*4]
    cvtps2pd xmm1, xmm1
    mulsd   xmm0, xmm1
    mov edx, DWORD PTR _j$2541[ebp]
    movss   xmm1, DWORD PTR _S[eax+edx*4]
    cvtps2pd xmm1, xmm1
    addsd   xmm1, xmm0
    mov eax, DWORD PTR _i$2537[ebp]
    shl eax, 5
    xorps   xmm0, xmm0
    cvtsd2ss xmm0, xmm1
    mov ecx, DWORD PTR _j$2541[ebp]
    movss   DWORD PTR _S[eax+ecx*4], xmm0
```

matrix_multiplication.asm -- line 11(after)

8x8 Matrices:

```
Start Value:2207300004908
End Value:2207300004913
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
8x8 Maxtrix Multiplication: 2.85091Microseconds.
Press any key to continue . . .
```

16x16 Matrices

```
Start Value:2207545295847
End Value:2207545295885
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
16x16 Maxtrix Multiplication: 21.6669Microseconds.
Press any key to continue . . .
```

32x32 Matrices

```
Start Value:2207712622917
End Value:2207712623176
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
32x32 Maxtrix Multiplication: 147.677Microseconds.
Press any key to continue . . .
```

64x64 Matrices

```
Start Value:22078141317333
End Value:2207814133713
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
64x64 Maxtrix Multiplication: 1128.96Microseconds.
Press any key to continue . . .
```

128x128 Matrices

```
Start Value:2207887682881
End Value:2207887699834
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
128x128 Maxtrix Multiplication: 9666.28Microseconds.
Press any key to continue . . .
```

256x256 Matrices

```
Start Value:2207980354477
End Value:2207980495240
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
256x256 Maxtrix Multiplication: 80260.4Microseconds.
Press any key to continue . . .
```

512x512 Matrices

```
Start Value:22080585244422
End Value:2208059786203
QueryPerformanceCounter minimum resolution: 1/1753828Seconds,
512x512 Maxtrix Multiplication: 719444Microseconds.
Press any key to continue . . .
```

We can manually optimize our asm file slightly by modifying the loop executions. The compiler automatically reads from memory and moves the value into the eax register. Then it increment the value in the register by one, then moves it back into memory. This is done every time each loop iterates. Instead we can optimize this by keeping the value inside memory and incrementing the value right from memory. That way we never need to use the eax register for the loops.

```
; 7    :    for (int i = 0; i < size; i++)

    mov DWORD PTR _i$4440[ebp], 0
    jmp SHORT $LN9@matrix_mul
$LN8@matrix_mul:
    ;mov    eax, DWORD PTR _i$4440[ebp]
    ;add    eax, 1
    ;mov    DWORD PTR _i$4440[ebp], eax
    add     DWORD PTR _i$4440[ebp], 1
$LN9@matrix_mul:
    cmp DWORD PTR _i$4440[ebp], 8
    jge $LN7@matrix_mul


; 8    :    {
; 9    :        for (int j = 0; j < size; j++)

    mov DWORD PTR _j$4444[ebp], 0
    jmp SHORT $LN6@matrix_mul
$LN5@matrix_mul:
    ;mov    eax, DWORD PTR _j$4444[ebp]
    ;add    eax, 1
    ;mov    DWORD PTR _j$4444[ebp], eax
    add     DWORD PTR _j$4444[ebp], 1
$LN6@matrix_mul:
    cmp DWORD PTR _j$4444[ebp], 8
    jge SHORT $LN4@matrix_mul

; 10   :        {
; 11   :            for (int k = 0; k < size; k++)

    mov DWORD PTR _k$4448[ebp], 0
    jmp SHORT $LN3@matrix_mul
$LN2@matrix_mul:
    ;mov    eax, DWORD PTR _k$4448[ebp]
    ;add    eax, 1
    ;mov    DWORD PTR _k$4448[ebp], eax
    add     DWORD PTR _k$4448[ebp], 1
$LN3@matrix_mul:
    cmp DWORD PTR _k$4448[ebp], 8
    jge SHORT $LN1@matrix_mul
```

matrix_multiplication.asm -- lines 7-12

DPPS Vector Instruction Optimization

Using the smmintrin.h library, we can call the m_dp_ps function. We then can create four _m128 type vectors and load the matrix values using the _mm_set_ps function.

```
void matrix_multiplication(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];
            }
        }
    }
}
```

matrix_multiplication.cpp

Below is the compiler generated assembly code for the matrix multiplication function using dpps:

```
_TEXT   SEGMENT
$T4463 = -576                       ; size = 16
$T4464 = -544                       ; size = 16
$T4465 = -512                       ; size = 16
$T4466 = -480                       ; size = 16
$T4467 = -448                       ; size = 16
$T4468 = -416                       ; size = 16
_k$4459 = -196                      ; size = 4
_j$4455 = -184                      ; size = 4
_i$4451 = -172                      ; size = 4
_s$ = -160                      ; size = 16
_v4$ = -128                     ; size = 16
_v3$ = -96                      ; size = 16
_v2$ = -64                      ; size = 16
_v1$ = -32                      ; size = 16
_mask$ = -8                     ; size = 4
_A$ = 8                         ; size = 4
_B$ = 12                        ; size = 4
_S$ = 16                        ; size = 4
?matrix_multiplication@@YAXQAY07M00@Z PROC      ; matrix_multiplication, COMDAT

; 21   : {

    push    ebx
    mov ebx, esp
    sub esp, 8
    and esp, -16                ; fffffff0H
    add esp, 4
    push    ebp
    mov ebp, DWORD PTR [ebx+4]
    mov DWORD PTR [esp+4], ebp
    mov ebp, esp
    sub esp, 584                ; 00000248H
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-584]
    mov ecx, 146                ; 00000092H
    mov eax, -858993460             ; ccccccccH
    rep stosd

; 22   :    const int mask = 0x1F;

    mov DWORD PTR _mask$[ebp], 31       ; 0000001fH

; 23   :
; 24   :    __m128 v1 = { 0.0, 0.0, 0.0, 0.0 }, v2 = { 0.0, 0.0, 0.0, 0.0 },

    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v1$[ebp], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v1$[ebp+4], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v1$[ebp+8], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v1$[ebp+12], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v2$[ebp], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v2$[ebp+4], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v2$[ebp+8], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v2$[ebp+12], xmm0
```

```
; 25   :    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 };

    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v3$[ebp], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v3$[ebp+4], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v3$[ebp+8], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v3$[ebp+12], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v4$[ebp], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v4$[ebp+4], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v4$[ebp+8], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _v4$[ebp+12], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _s$[ebp], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _s$[ebp+4], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _s$[ebp+8], xmm0
    movss   xmm0, DWORD PTR __real@00000000
    movss   DWORD PTR _s$[ebp+12], xmm0

; 26   :    for (int i = 0; i < size; i++) {

    mov DWORD PTR _i$4451[ebp], 0
    jmp SHORT $LN9@matrix_mul
$LN8@matrix_mul:
    mov eax, DWORD PTR _i$4451[ebp]
    add eax, 1
    mov DWORD PTR _i$4451[ebp], eax
$LN9@matrix_mul:
    cmp DWORD PTR _i$4451[ebp], 8
    jge $LN7@matrix_mul

; 27   :        for (int j = 0; j < size; j++) {

    mov DWORD PTR _j$4455[ebp], 0
    jmp SHORT $LN6@matrix_mul
$LN5@matrix_mul:
    mov eax, DWORD PTR _j$4455[ebp]
    add eax, 1
    mov DWORD PTR _j$4455[ebp], eax
$LN6@matrix_mul:
    cmp DWORD PTR _j$4455[ebp], 8
    jge $LN4@matrix_mul

; 28   :            for (int k = 0; k < size; k += 8) {

    mov DWORD PTR _k$4459[ebp], 0
    jmp SHORT $LN3@matrix_mul
$LN2@matrix_mul:
    mov eax, DWORD PTR _k$4459[ebp]
    add eax, 8
    mov DWORD PTR _k$4459[ebp], eax
$LN3@matrix_mul:
    cmp DWORD PTR _k$4459[ebp], 8
    jge $LN1@matrix_mul
```

```
; 29   :               v1 = _mm_set_ps(A[i][k], A[i][k + 1], A[i][k + 2], A[i][k + 3]);

    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _A$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm0, DWORD PTR [eax+ecx*4+12]
    mov edx, DWORD PTR _i$4451[ebp]
    shl edx, 5
    add edx, DWORD PTR _A$[ebx]
    mov eax, DWORD PTR _k$4459[ebp]
    movss   xmm1, DWORD PTR [edx+eax*4+8]
    mov ecx, DWORD PTR _i$4451[ebp]
    shl ecx, 5
    add ecx, DWORD PTR _A$[ebx]
    mov edx, DWORD PTR _k$4459[ebp]
    movss   xmm2, DWORD PTR [ecx+edx*4+4]
    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _A$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm3, DWORD PTR [eax+ecx*4]
    unpcklps xmm0, xmm2
    unpcklps xmm1, xmm3
    unpcklps xmm0, xmm1
    movaps  XMMWORD PTR $T4463[ebp], xmm0
    movaps  xmm0, XMMWORD PTR $T4463[ebp]
    movaps  XMMWORD PTR _v1$[ebp], xmm0

; 30   :               v2 = _mm_set_ps(A[i][k + 4], A[i][k + 5], A[i][k + 6], A[i][k + 7]);

    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _A$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm0, DWORD PTR [eax+ecx*4+28]
    mov edx, DWORD PTR _i$4451[ebp]
    shl edx, 5
    add edx, DWORD PTR _A$[ebx]
    mov eax, DWORD PTR _k$4459[ebp]
    movss   xmm1, DWORD PTR [edx+eax*4+24]
    mov ecx, DWORD PTR _i$4451[ebp]
    shl ecx, 5
    add ecx, DWORD PTR _A$[ebx]
    mov edx, DWORD PTR _k$4459[ebp]
    movss   xmm2, DWORD PTR [ecx+edx*4+20]
    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _A$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm3, DWORD PTR [eax+ecx*4+16]
    unpcklps xmm0, xmm2
    unpcklps xmm1, xmm3
    unpcklps xmm0, xmm1
    movaps  XMMWORD PTR $T4464[ebp], xmm0
    movaps  xmm0, XMMWORD PTR $T4464[ebp]
    movaps  XMMWORD PTR _v2$[ebp], xmm0
```

```
; 31   :               v3 = _mm_set_ps(B[i][k], B[i][k + 1], B[i][k + 2], B[i][k + 3]);

    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _B$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm0, DWORD PTR [eax+ecx*4+12]
    mov edx, DWORD PTR _i$4451[ebp]
    shl edx, 5
    add edx, DWORD PTR _B$[ebx]
    mov eax, DWORD PTR _k$4459[ebp]
    movss   xmm1, DWORD PTR [edx+eax*4+8]
    mov ecx, DWORD PTR _i$4451[ebp]
    shl ecx, 5
    add ecx, DWORD PTR _B$[ebx]
    mov edx, DWORD PTR _k$4459[ebp]
    movss   xmm2, DWORD PTR [ecx+edx*4+4]
    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _B$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm3, DWORD PTR [eax+ecx*4]
    unpcklps xmm0, xmm2
    unpcklps xmm1, xmm3
    unpcklps xmm0, xmm1
    movaps  XMMWORD PTR $T4465[ebp], xmm0
    movaps  xmm0, XMMWORD PTR $T4465[ebp]
    movaps  XMMWORD PTR _v3$[ebp], xmm0

; 32   :               v4 = _mm_set_ps(B[i][k + 4], B[i][k + 5], B[i][k + 6], B[i][k + 7]);

    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _B$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm0, DWORD PTR [eax+ecx*4+28]
    mov edx, DWORD PTR _i$4451[ebp]
    shl edx, 5
    add edx, DWORD PTR _B$[ebx]
    mov eax, DWORD PTR _k$4459[ebp]
    movss   xmm1, DWORD PTR [edx+eax*4+24]
    mov ecx, DWORD PTR _i$4451[ebp]
    shl ecx, 5
    add ecx, DWORD PTR _B$[ebx]
    mov edx, DWORD PTR _k$4459[ebp]
    movss   xmm2, DWORD PTR [ecx+edx*4+20]
    mov eax, DWORD PTR _i$4451[ebp]
    shl eax, 5
    add eax, DWORD PTR _B$[ebx]
    mov ecx, DWORD PTR _k$4459[ebp]
    movss   xmm3, DWORD PTR [eax+ecx*4+16]
    unpcklps xmm0, xmm2
    unpcklps xmm1, xmm3
    unpcklps xmm0, xmm1
    movaps  XMMWORD PTR $T4466[ebp], xmm0
    movaps  xmm0, XMMWORD PTR $T4466[ebp]
    movaps  XMMWORD PTR _v4$[ebp], xmm0

; 33   :
; 34   :               s = _mm_dp_ps(v1, v3, mask);

    movaps  xmm0, XMMWORD PTR _v3$[ebp]
    movaps  xmm1, XMMWORD PTR _v1$[ebp]
    dpps    xmm1, xmm0, 31          ; 0000001fH
    movaps  XMMWORD PTR $T4467[ebp], xmm1
    movaps  xmm0, XMMWORD PTR $T4467[ebp]
    movaps  XMMWORD PTR _s$[ebp], xmm0

; 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

; 37   :
; 38   :                s = _mm_dp_ps(v2, v4, mask);

    movaps  xmm0, XMMWORD PTR _v4$[ebp]
    movaps  xmm1, XMMWORD PTR _v2$[ebp]
    dpps    xmm1, xmm0, 31              ; 0000001fH
    movaps  XMMWORD PTR $T4468[ebp], xmm1
    movaps  xmm0, XMMWORD PTR $T4468[ebp]
    movaps  XMMWORD PTR _s$[ebp], xmm0

; 39   :
; 40   :                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
```

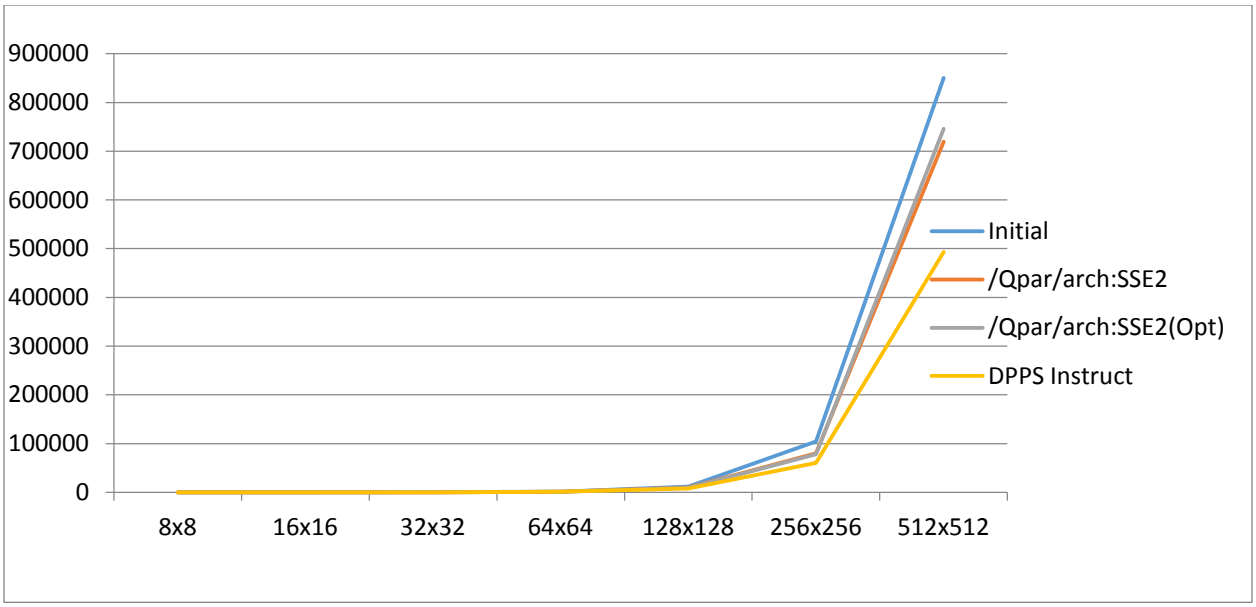
```
; 41   :            }

    jmp $LN2@matrix_mul
$LN1@matrix_mul:

; 42   :        }

    jmp $LN5@matrix_mul
$LN4@matrix_mul:

; 43   :    }

    jmp $LN8@matrix_mul
$LN7@matrix_mul:

; 44   : }

    pop edi
    pop esi
    mov esp, ebp
    pop ebp
    mov esp, ebx
    pop ebx
    ret 0
?matrix_multiplication@@YAXQAY07M00@Z ENDP      ; matrix_multiplication
_TEXT   ENDS
END
```

Results

Now let's compare all of our results with matrices ranging from 8x8 to 512x512:

	Size N						
	8x8	**16x16**	**32x32**	**64x64**	**128x128**	**256x256**	**512x512**
Before Optimization	3.42	30.22	244.04	1485.32	11978.40	104251.00	850372.00
/Qpar and /arch:SSE2	2.85	21.67	147.68	1128.96	9666.28	80260.40	719444.00
/Qpar and /arch:SSE2 (Optimized)	2.85	21.67	139.78	1017.18	8821.59	77767.20	746109.00
DPPS Instruct	2.28	12.59	128.88	1013.98	7891.66	60142.20	492825.00

(Table #1)



(Graph #1)

Conclusion

We tested the performance of various optimization methods on square matrix multiplication such as enabling automatic parallelization and vectorization, manipulating compiler generated assembly code, and implementing dpps functions. In the end, we timed, organized and compared the results of each of these optimization methods using the QueryPerformanceCounter. We then charted and graphed these results in order to easily analyze and compare the performance between them.