Final Take Home Test

Optimization of Matrix-Matrix Multiplication Using Vector Instructions

Zi Xuan Li

Professor Izidor Gertner

CSC 34200 Spring 2024

May 12th, 2024

Table of Contents

1. Objective..3

2. Task#1..3

3. Task#2..3-5

4. Task#3..5-7

5. Task#4..8

6. Task#5...8-12

7. Task#6...12-14

8. Task#7...14-16

9. Conclusion...16

Objective:

The aim of this concluding take-home exam is to enhance compiler-generated code for computing the product of two matrices using vector instructions and DPPS vector instruction. It builds upon the earlier test on matrix multiplication. The objective is to explore the performance contrast between vectorization and non-vectorization, assessed with Chrono, a high-resolution timer for execution time measurement.

Task#1: Use CPUID instructions to determine your processor vector processing capabilities.

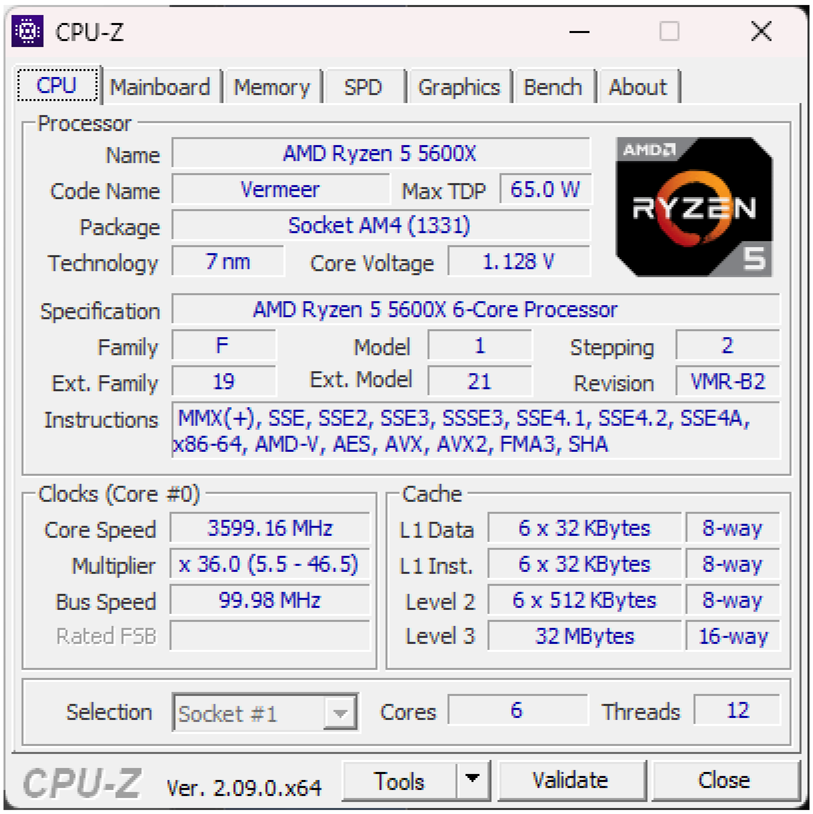
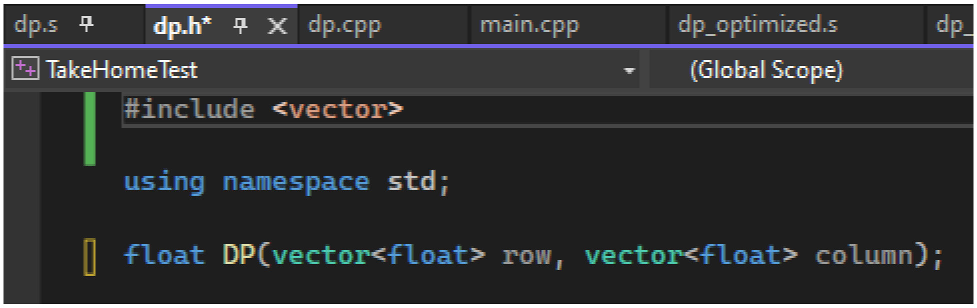


Figure 1: Information of my CPU using CPU-Z from the CPUID instructions

The processing capabilities of my CPU, the AMD Ryzen 5 5600x, can be found under the instructions tab in the window above. It supports MMX (+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AMD-V, AES, AVX, AVX2, FMA3, and SHA.

Task#2: Write C/C++ main () to compute Matrix-Matrix multiplication, can be taken from previous take-home test. The focus in this take home test is the most inner loop that computes dot product of row and column. You can use a function DP (row, column) from a previous take home test.

Place the function DP (row, column) in a separate file from main () that calls this function. Vector sizes should be powers of 2 (e.g. 16, 32, 64, ...512, ...216 etc.)

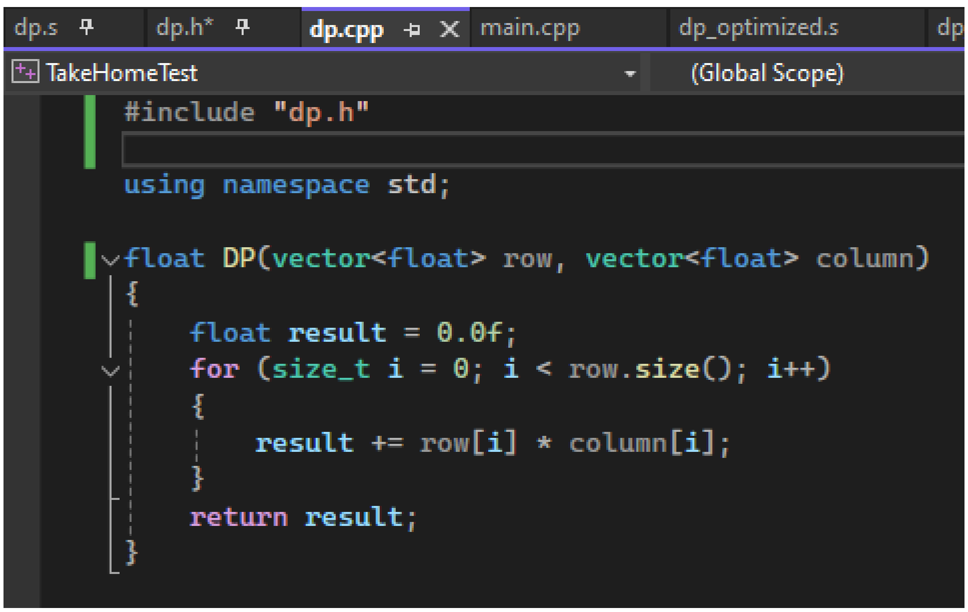



```cpp
#include <vector>

using namespace std;

float DP(vector<float> row, vector<float> column);
```

Figure 2: Image of the dp.h file




```cpp
#include "dp.h"

using namespace std;

float DP(vector<float> row, vector<float> column)
{
    float result = 0.0f;
    for (size_t i = 0; i < row.size(); i++)
    {
        result += row[i] * column[i];
    }
    return result;
}
```

Figure 3: Image of the dp.cpp file

```cpp
#include "dp.h"
#include <iostream>
#include <vector>
#include <stdio.h>
#include <random>
#include <chrono>

using namespace std;

void fill_matrix(vector<float>& matrix, size_t N) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dist(0.0f, 99.0f);
    for (size_t i = 0; i < N * N; ++i) {
        matrix[i] = dist(gen);
    }
}

void matrix_multiply(const vector<float>& A, const vector<float>& B, vector<float>& C, size_t N) {
    for (size_t i = 0; i < N; ++i) {
        for (size_t j = 0; j < N; ++j) {
            vector<float> row(N), column(N);
            for (size_t k = 0; k < N; ++k) {
                row[k] = A[i * N + k];
                column[k] = B[k * N + j];
            }
            C[i * N + j] = DP(row, column);
        }
    }
}

int main() {
    vector<size_t> sizes = { 16, 32, 64, 128, 256, 512, 1024 };
    for (auto N : sizes) {
        vector<float> A(N * N), B(N * N), C(N * N);

        fill_matrix(A, N);
        fill_matrix(B, N);

        auto start = chrono::high_resolution_clock::now();
        matrix_multiply(A, B, C, N);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> diff = end - start;
        cout << "Time to multiply two " << N << "x" << N << " float matrices: " << diff.count() << " seconds\n";
    }
    return 0;
}
```

Figure 4: Image of the main.cpp file.

Task#3: Compile code in §2 and create assembly code for function DP (row, column) only. Make sure that compiler generated vectorized code.

dp.h | dp.cpp | main.cpp* | dp.s | dp_optimized.s | dp_dpps.cpp

```
	.file	"dp.cpp"
	.text
	.p2align 4
	.globl	_Z2DPSt6vectorIfSaIfEES1_
	.def	_Z2DPSt6vectorIfSaIfEES1_;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z2DPSt6vectorIfSaIfEES1_
_Z2DPSt6vectorIfSaIfEES1_:
.LFB1021:
	.seh_endprologue
	movq	8(%rcx), %r8
	movq	(%rcx), %rax
	movq	%r8, %r9
	subq	%rax, %r9
	movq	%r9, %rcx
	sarq	$2, %rcx
	cmpq	%rax, %r8
	je	.L10
	movq	(%rdx), %rdx
	cmpq	$28, %r9
	jbe	.L11
	movq	%rcx, %r9
	xorl	%r8d, %r8d
	vxorps	%xmm0, %xmm0, %xmm0
	shrq	$3, %r9
	salq	$5, %r9
	.p2align 4
	.p2align 3
.L4:
	vmovups	(%rax,%r8), %ymm4
	vmulps	(%rdx,%r8), %ymm4, %ymm1
	addq	$32, %r8
	vshufps	$85, %xmm1, %xmm1, %xmm3
	vshufps	$255, %xmm1, %xmm1, %xmm2
	vaddss	%xmm1, %xmm0, %xmm0
	vaddss	%xmm3, %xmm0, %xmm0
	vunpckhps	%xmm1, %xmm1, %xmm3
	vextractf128	$0x1, %ymm1, %xmm1
	vaddss	%xmm3, %xmm0, %xmm0
	vaddss	%xmm2, %xmm0, %xmm0
	vshufps	$85, %xmm1, %xmm1, %xmm2
	vaddss	%xmm1, %xmm0, %xmm0
	vaddss	%xmm2, %xmm0, %xmm0
	vunpckhps	%xmm1, %xmm1, %xmm2
	vshufps	$255, %xmm1, %xmm1, %xmm1
	vaddss	%xmm2, %xmm0, %xmm0
	vaddss	%xmm1, %xmm0, %xmm0
	cmpq	%r8, %r9
	jne	.L4
	movq	%rcx, %r8
	andq	$-8, %r8
	testb	$7, %cl
```

dp.h | dp.cpp | main.cpp* | dp.s | dp_optimized.s | dp_dpps.cpp

```
        testb   $7, %cl
        je  .L22
        vzeroupper
.L3:
        movq    %rcx, %r9
        subq    %r8, %r9
        leaq    -1(%r9), %r10
        cmpq    $2, %r10
        jbe .L9
        vmovups (%rax,%r8,4), %xmm5
        movq    %r9, %r10
        vmulps  (%rdx,%r8,4), %xmm5, %xmm1
        andq    $-4, %r10
        addq    %r10, %r8
        andl    $3, %r9d
        vaddss  %xmm1, %xmm0, %xmm0
        vshufps $85, %xmm1, %xmm1, %xmm2
        vaddss  %xmm2, %xmm0, %xmm0
        vunpckhps   %xmm1, %xmm1, %xmm2
        vshufps $255, %xmm1, %xmm1, %xmm1
        vaddss  %xmm2, %xmm0, %xmm0
        vaddss  %xmm1, %xmm0, %xmm0
        je  .L1
        .p2align 4
        .p2align 3
.L9:
        vmovss  (%rax,%r8,4), %xmm1
        vmulss  (%rdx,%r8,4), %xmm1, %xmm1
        incq    %r8
        vaddss  %xmm1, %xmm0, %xmm0
        cmpq    %rcx, %r8
        jb  .L9
.L1:
        ret
        .p2align 4
        .p2align 3
.L10:
        vxorps  %xmm0, %xmm0, %xmm0
        ret
        .p2align 4
        .p2align 3
.L22:
        vzeroupper
        ret
.L11:
        xorl    %r8d, %r8d
        vxorps  %xmm0, %xmm0, %xmm0
        jmp .L3
        .seh_endproc
        .ident  "GCC: (Rev2, Built by MSYS2 project) 12.1.0"
```

Figure 5: Image of the compiler generated assembly code from dp.h

Task#4: Use high resolution timer to measure execution time (as in previous take-home test). Plot graph: time versus vector size.

```
Time to multiply two 16x16 float matrices: 9.2e-05 seconds
Time to multiply two 32x32 float matrices: 0.0002864 seconds
Time to multiply two 64x64 float matrices: 0.0009116 seconds
Time to multiply two 128x128 float matrices: 0.0057554 seconds
Time to multiply two 256x256 float matrices: 0.0376169 seconds
Time to multiply two 512x512 float matrices: 0.395068 seconds
Time to multiply two 1024x1024 float matrices: 4.69744 seconds
```

Figure 6: Execution times of various 2^n integer matrix sizes from n = 16 to n = 1024

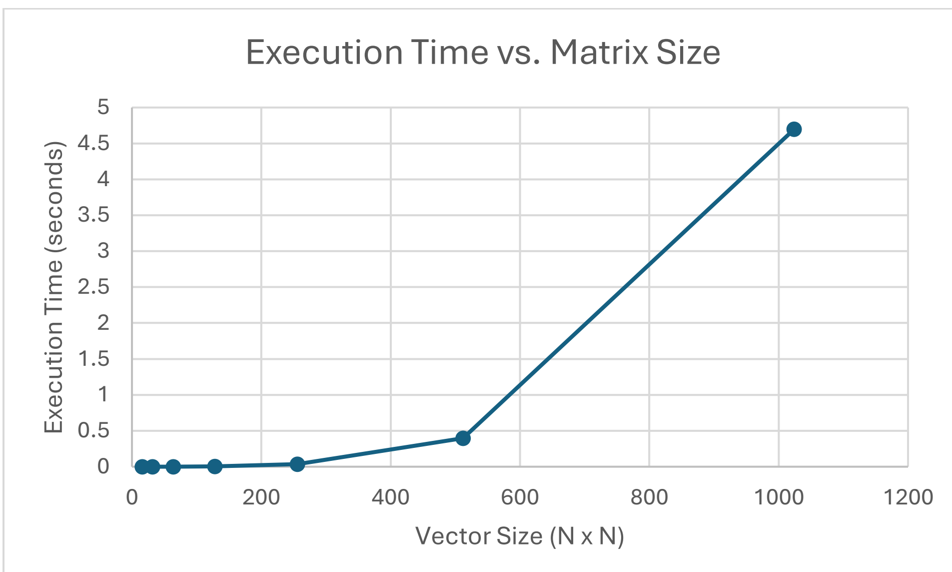


Figure 7: Graph measuring execution time vs. vector size for compiler generated code

Task#5: Create assembly code for function DP (row, column). The assembly code should contain vector instructions.

dp.h   dp.cpp   main.cpp   dp.s   dp_optimized.s   dp_dpps.cpp

```
	.file	"dp.cpp"
	.text
	.p2align 4
	.globl	_Z2DPSt6vectorIfSaIfEES1_
	.def	_Z2DPSt6vectorIfSaIfEES1_;	.scl	2;	.type	32;	.endef
	.seh_proc	_Z2DPSt6vectorIfSaIfEES1_
_Z2DPSt6vectorIfSaIfEES1_:
.LFB1021:
	.seh_endprologue
	movq	8(%rcx), %rax
	movq	(%rcx), %rcx
	movq	%rax, %r9
	subq	%rcx, %r9
	movq	%r9, %r8
	sarq	$2, %r8
	cmpq	%rcx, %rax
	je	.L9
	movq	(%rdx), %rdx
	cmpq	$28, %r9
	jbe	.L10
	movq	%r8, %r10
	xorl	%eax, %eax
	vxorps	%xmm0, %xmm0, %xmm0
	shrq	$3, %r10
	movq	%r10, %r11
	salq	$5, %r11
	andl	$1, %r10d
	je	.L4
	vmovups	(%rcx), %ymm4
	movl	$32, %eax
	vmulps	(%rdx), %ymm4, %ymm1
	vshufps	$85, %xmm1, %xmm1, %xmm5
	vunpckhps	%xmm1, %xmm1, %xmm3
	vshufps	$255, %xmm1, %xmm1, %xmm2
	vaddss	%xmm1, %xmm0, %xmm0
	vextractf128	$0x1, %ymm1, %xmm1
	vaddss	%xmm5, %xmm0, %xmm4
	vaddss	%xmm3, %xmm4, %xmm0
	vaddss	%xmm2, %xmm0, %xmm5
	vshufps	$85, %xmm1, %xmm1, %xmm0
	vunpckhps	%xmm1, %xmm1, %xmm2
	vaddss	%xmm1, %xmm5, %xmm3
	vshufps	$255, %xmm1, %xmm1, %xmm1
	vaddss	%xmm0, %xmm3, %xmm5
	vaddss	%xmm2, %xmm5, %xmm4
	vaddss	%xmm1, %xmm4, %xmm0
	cmpq	$32, %r11
	je	.L23
	.p2align 4
	.p2align 3
```

dp.h | dp.cpp | main.cpp | dp.s | **dp_optimized.s** | dp_dpps.cpp

```
        .p2align 3
.L4:
        vmovups (%rcx,%rax), %ymm3
        vmulps  (%rdx,%rax), %ymm3, %ymm5
        vshufps $85, %xmm5, %xmm5, %xmm1
        vunpckhps   %xmm5, %xmm5, %xmm3
        vshufps $255, %xmm5, %xmm5, %xmm2
        vaddss  %xmm5, %xmm0, %xmm0
        vextractf128    $0x1, %ymm5, %xmm5
        vaddss  %xmm1, %xmm0, %xmm4
        vmovups 32(%rcx,%rax), %ymm1
        vaddss  %xmm3, %xmm4, %xmm0
        vaddss  %xmm2, %xmm0, %xmm4
        vshufps $85, %xmm5, %xmm5, %xmm0
        vunpckhps   %xmm5, %xmm5, %xmm2
        vaddss  %xmm5, %xmm4, %xmm3
        vshufps $255, %xmm5, %xmm5, %xmm5
        vaddss  %xmm0, %xmm3, %xmm4
        vaddss  %xmm2, %xmm4, %xmm3
        vmulps  32(%rdx,%rax), %ymm1, %ymm4
        addq    $64, %rax
        vaddss  %xmm5, %xmm3, %xmm0
        vaddss  %xmm4, %xmm0, %xmm5
        vshufps $85, %xmm4, %xmm4, %xmm0
        vunpckhps   %xmm4, %xmm4, %xmm3
        vshufps $255, %xmm4, %xmm4, %xmm2
        vextractf128    $0x1, %ymm4, %xmm4
        vaddss  %xmm0, %xmm5, %xmm1
        vaddss  %xmm3, %xmm1, %xmm5
        vaddss  %xmm2, %xmm5, %xmm0
        vshufps $85, %xmm4, %xmm4, %xmm5
        vunpckhps   %xmm4, %xmm4, %xmm2
        vaddss  %xmm4, %xmm0, %xmm3
        vshufps $255, %xmm4, %xmm4, %xmm4
        vaddss  %xmm5, %xmm3, %xmm0
        vaddss  %xmm2, %xmm0, %xmm3
        vaddss  %xmm4, %xmm3, %xmm0
        cmpq    %rax, %r11
        jne .L4
.L23:
        movq    %r8, %r11
        andq    $-8, %r11
        testb   $7, %r8b
        je  .L26
        vzeroupper
.L3:
        movq    %r8, %r9
        subq    %r11, %r9
        leaq    -1(%r9), %r10
        cmpq    $2, %r10
```

dp.h dp.cpp main.cpp dp.s dp_optimized.s dp_dpps.cpp

```
        jbe .L7
        vmovups (%rcx,%r11,4), %xmm1
        movq    %r9, %rax
        vmulps  (%rdx,%r11,4), %xmm1, %xmm5
        andq    $-4, %rax
        addq    %rax, %r11
        andl    $3, %r9d
        vaddss  %xmm5, %xmm0, %xmm0
        vshufps $85, %xmm5, %xmm5, %xmm3
        vunpckhps   %xmm5, %xmm5, %xmm1
        vshufps $255, %xmm5, %xmm5, %xmm5
        vaddss  %xmm3, %xmm0, %xmm4
        vaddss  %xmm1, %xmm4, %xmm2
        vaddss  %xmm5, %xmm2, %xmm0
        je  .L1
    .L7:
        vmovss  (%rcx,%r11,4), %xmm3
        leaq    1(%r11), %r10
        leaq    0(,%r11,4), %r9
        vfmadd231ss (%rdx,%r11,4), %xmm3, %xmm0
        cmpq    %r8, %r10
        jnb .L1
        vmovss  4(%rcx,%r9), %xmm4
        addq    $2, %r11
        vfmadd231ss 4(%rdx,%r9), %xmm4, %xmm0
        cmpq    %r8, %r11
        jnb .L1
        vmovss  8(%rcx,%r9), %xmm1
        vfmadd231ss 8(%rdx,%r9), %xmm1, %xmm0
    .L1:
        ret
        .p2align 4
        .p2align 3
    .L9:
        vxorps  %xmm0, %xmm0, %xmm0
        ret
        .p2align 4
        .p2align 3
    .L26:
        vzeroupper
        ret
    .L10:
        xorl    %r11d, %r11d
        vxorps  %xmm0, %xmm0, %xmm0
        jmp .L3
        .seh_endproc
        .ident  "GCC: (Rev2, Built by MSYS2 project) 12.1.0"

```

Figure 8: Image of the optimized assembly code of dp.h

```
Time to multiply two 16x16 float matrices: 9.12e-05 seconds
Time to multiply two 32x32 float matrices: 0.0002748 seconds
Time to multiply two 64x64 float matrices: 0.0009819 seconds
Time to multiply two 128x128 float matrices: 0.0056039 seconds
Time to multiply two 256x256 float matrices: 0.0373567 seconds
Time to multiply two 512x512 float matrices: 0.392527 seconds
Time to multiply two 1024x1024 float matrices: 4.4821 seconds
```

Figure 9: Execution times of various 2^n integer matrix sizes from n = 16 to n = 1024

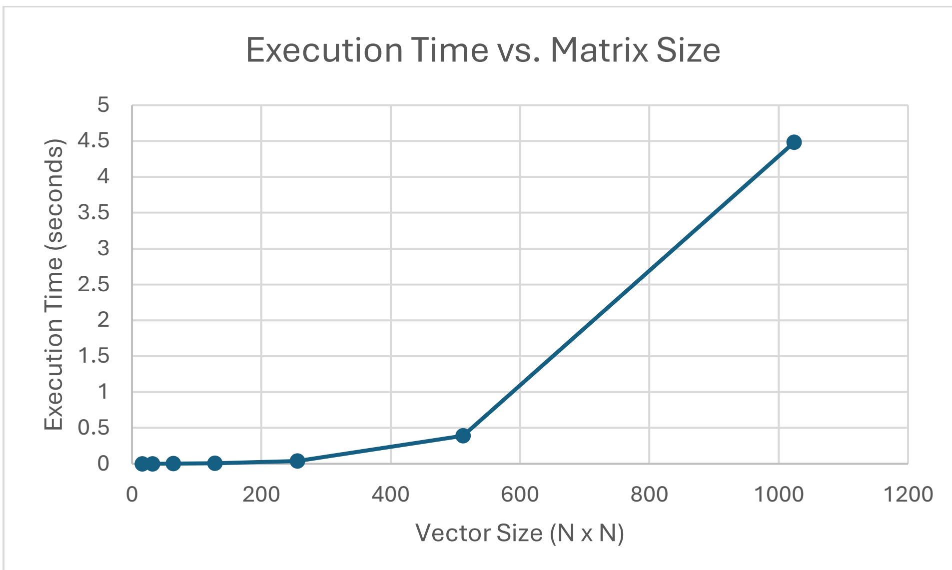


Figure 10: Graph measuring execution time vs. vector size for optimized code

Task#6: To optimize the code further, please try to use machine vector instruction DPPS to compute dot product.

dp.h | dp.cpp | main.cpp | dp.s | dp_optimized.s | dp_dpps.cpp*

TakeHomeTest

```cpp
#include "dp.h"
#include <immintrin.h>
#include <stdexcept>

using namespace std;

float DP(vector<float> row, vector<float> column)
{
    size_t size = row.size();
    if (size != column.size())
        throw std::invalid_argument("Vectors are not of the same size.");

    float result = 0.0f;
    size_t i = 0;

    __m256 sum_vec = _mm256_setzero_ps();
    for (; i <= size - 8; i += 8)
    {
        __m256 row_vec = _mm256_loadu_ps(&row[i]);
        __m256 col_vec = _mm256_loadu_ps(&column[i]);
        __m256 dp = _mm256_dp_ps(row_vec, col_vec, 0xF1);
        sum_vec = _mm256_add_ps(sum_vec, dp);
    }

    float temp[8];
    _mm256_storeu_ps(temp, sum_vec);
    for (int j = 0; j < 8; ++j)
    {
        result += temp[j];
    }

    for (; i < size; ++i)
    {
        result += row[i] * column[i];
    }
    return result;
}
```

Figure 11: Image of the dp_dpps.cpp file using machine vector instructions

```
Time to multiply two 16x16 float matrices: 9.08e-05 seconds
Time to multiply two 32x32 float matrices: 0.0002639 seconds
Time to multiply two 64x64 float matrices: 0.0007915 seconds
Time to multiply two 128x128 float matrices: 0.0045332 seconds
Time to multiply two 256x256 float matrices: 0.0294053 seconds
Time to multiply two 512x512 float matrices: 0.300061 seconds
Time to multiply two 1024x1024 float matrices: 3.83468 seconds
```

Figure 12: Execution times of various 2^n integer matrix sizes from n = 16 to n = 1024.

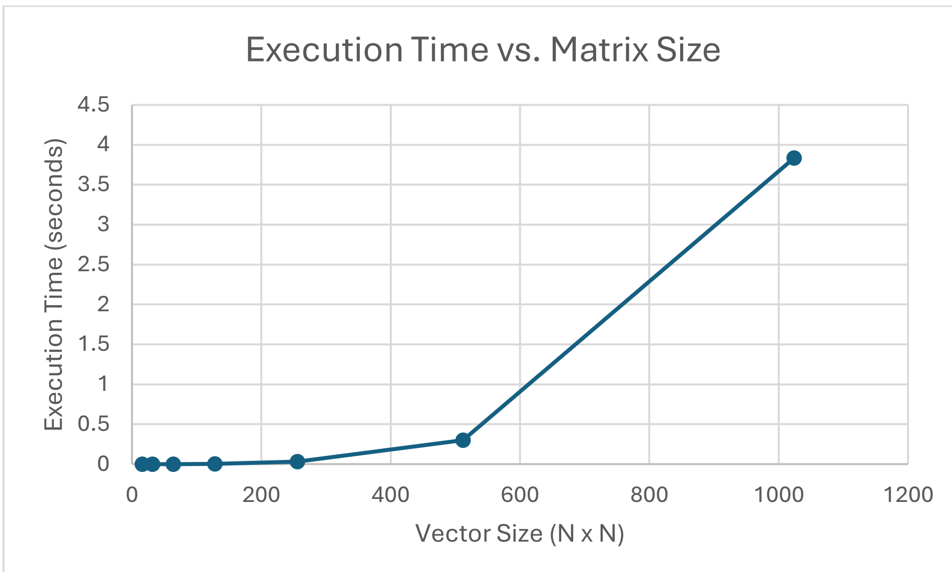


Figure 13: Graph measuring execution time vs. vector size using machine vector instruction DPPS code

Task#7: Compare all plots in one figure. Compare also to the performance plots from the previous take home test.

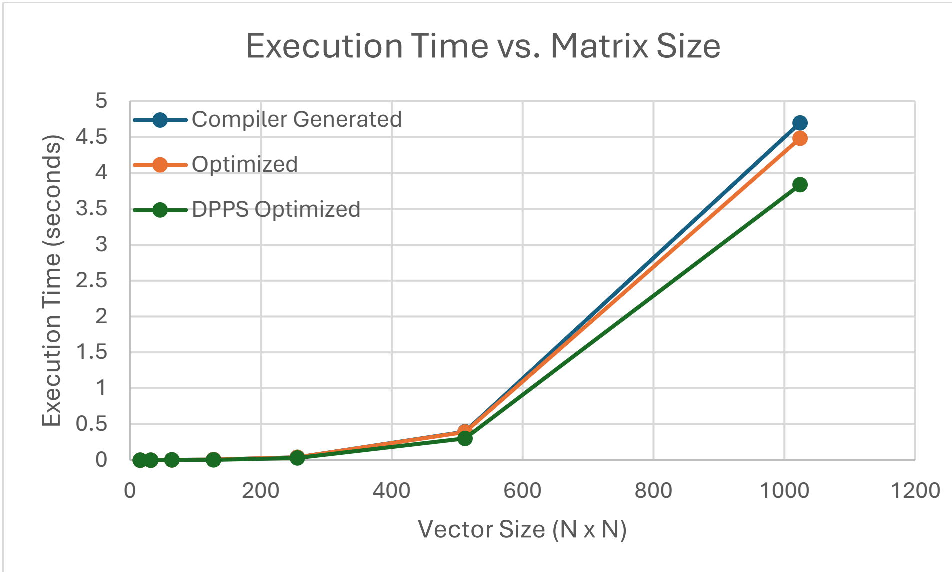


Figure 14: Graph comparing execution time vs. matrix size of compiler generated, optimized, and DPPS optimized assembly code

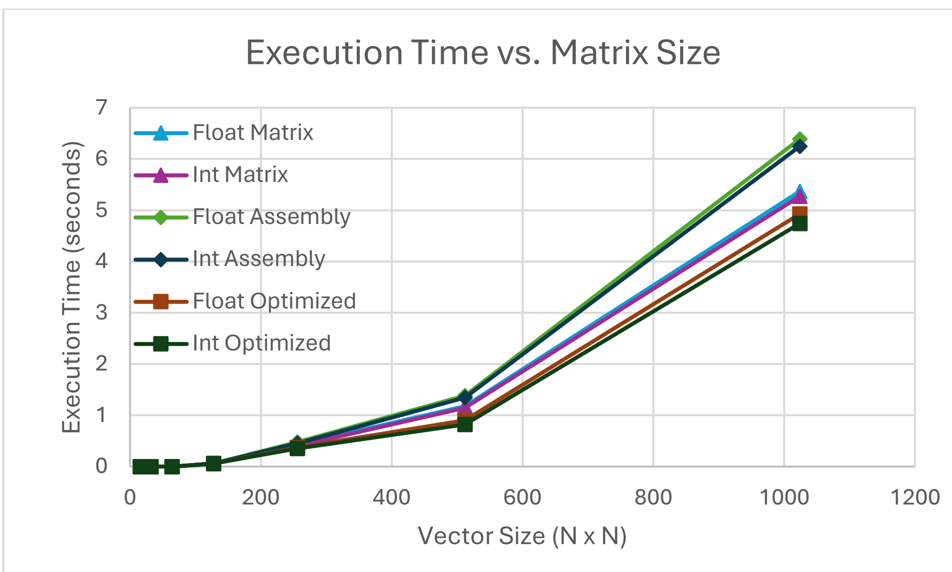


Figure 15: Graph comparing execution time vs. matrix size from previous take home test

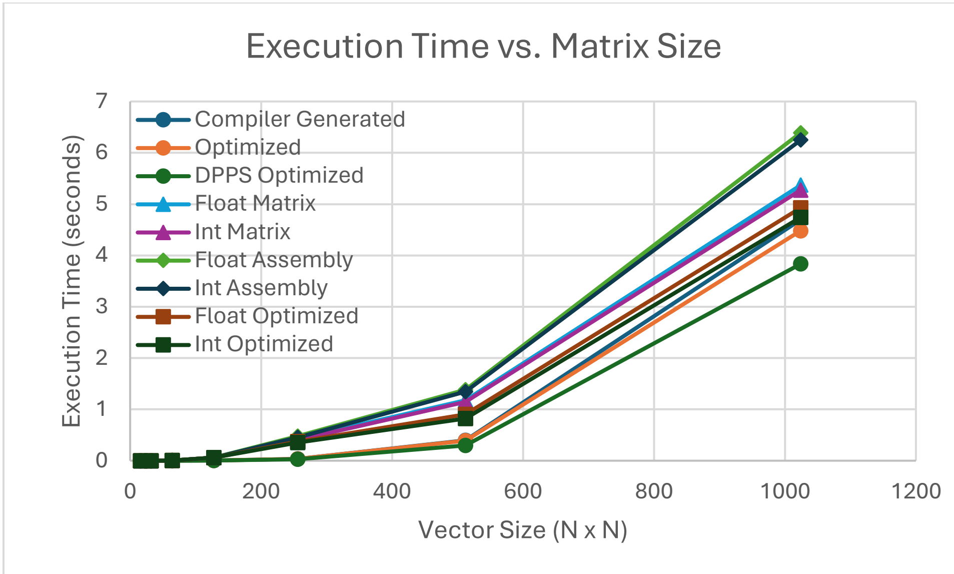


Figure 16: Graph comparing execution time vs. matrix size from the previous take home test and current take home test

Conclusion:

The primary goal of this take-home test was to employ vector instructions to enhance dot product computations in the dp.cpp file. Utilizing vectorization commands in g++, along with DPPS instructions, I compiled and executed the files and observed that the runtime improved with increasing optimization levels. The slowest runtime was without any optimization, followed by optimization with assembly, and finally optimization with DPPS.