FINAL TAKE HOME TEST

Meng Wai Chan
Professor Gertner
CSc 343
May 7, 2023

Meng Wai Chan
May 7, 2023

Table of Contents

Meng Wai Chan
May 7, 2023

I. Objective

The objective of this take home test is to optimize the compiler generated code for a program that computes the dot product using vector instructions. In this take home test we had to use the QueryPerformanceCounter function in the "Windows.h" library to measure execution time and observe the changes as we optimized the assembly code. First we write the function for the conventional dot product function in C++ and compile it without using any optimization. Then we enable Automatic Parallelization and Vectorization to see the changes. Then we optimize the dot product function using our own assembly code. Finally we wrote the assembly code using DPPS vector instruction to compare their differences. In this take home test we measure the runtime from when N = 2^3 to the maximum of N = 2^19. The simulation is run in Windows Visual Studio, then we use GCC compiler in a Linux environment to compare the difference.

Meng Wai Chan
May 7, 2023

II. AMD x86 compiler

CPU Vector Processing Capabilities using CPU-Z

Following the image below we can see that my processor is AMD Ryzen 5 3600 with 6 cores and 12 threads. In the instruction section we can see my processor support MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AMD-V, AES, AVX, AVX2, FMA3, and SHA instructions. In this take home test we will be using AVX2 vector instruction set.

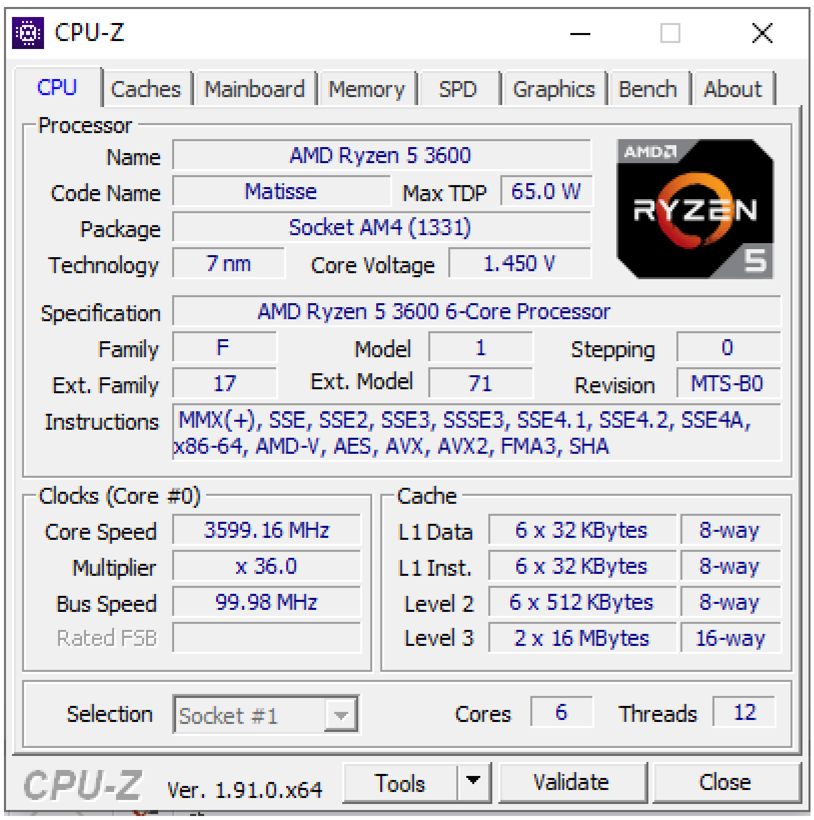


Figure 1: CPUID screenshot

Meng Wai Chan
May 7, 2023

Conventional Dot Product (No Optimization)

First we wrote a simple C++ function that computes dot product in a header file. Then you can see on Figure 3 we create an array and pass it onto the dotproduct function on line 28 to compute and time the output using QueryPerformanceCounter function as instructed by the assignment. This program will iterate from 2^3 to 2^19 for N number of arrays, to order to compute and compare the runtime.

```cpp

float dotproduct(float a[], float b[], int N) {
    float x = 0;
    for (int i = 0; i < N; i++) {
        x = x + (a[i] * b[i]);
    }

    return x;
}
```

Figure 2: Dot Product Function

```cpp
1  #include <iostream>
2  #include "dotproduct.h"
3  #include <windows.h>
4
5  using namespace std;
6
7  int main() {
8
9      _int64 ctr1 = 0, ctr2 = 0, freq = 0;
10     int N = 0;
11     int MAX = 19;
12
13     for (int i = 3; i <= MAX; i++) {
14         N = (int)pow(2, i);
15         freq = 0;
16
17         float* arr1 = new float[N];
18         float* arr2 = new float[N];
19
20         for (int j = 0; j < N; j++) {
21             arr1[j] = 8.12;
22             arr2[j] = 8.14;
23         }
24
25         if (QueryPerformanceCounter((LARGE_INTEGER*)&ctr1) != 0) {
26             cout << "Array Size: " << N << endl;
27
28             cout << " Dot Product: " <<dotproduct_asm(arr1, arr2, N)<<endl;
29
30             QueryPerformanceCounter((LARGE_INTEGER*)&ctr2);
31
32             QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
33
34             cout<<"Time: " <<(((ctr2 - ctr1) * 1.0) / freq) << endl;
35         }
36         else {
37             DWORD dwError = GetLastError();
38             cout << "Error value: = " << dwError << endl;
39         }
40
41         cout << endl;
42     }
43
44     return 0;
45 }
```

Figure 3: Dot Product main.cpp

```
Array Size: 8
Start Value: 767238909303
End Value: 767238913231
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 3928 counts.
Total time: 0.0003928 secs.

Array Size: 16
Start Value: 767238926222
End Value: 767238926821
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 599 counts.
Total time: 5.99e-05 secs.

Array Size: 32
Start Value: 767238937003
End Value: 767238938478
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 1475 counts.
Total time: 0.0001475 secs.
```

Figure 4: Example Dot Product main.cpp Output

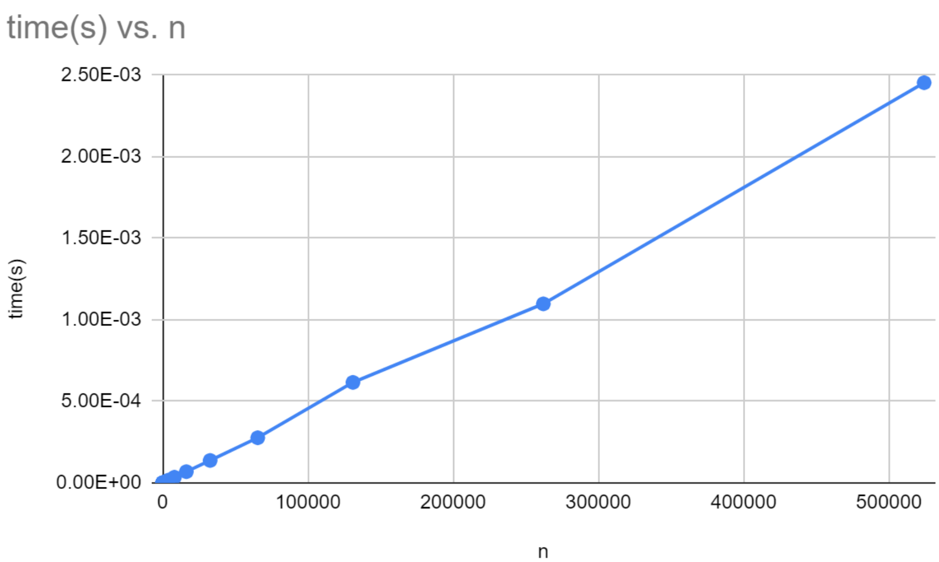


Figure 5: Runtime for conventional Method

Meng Wai Chan
May 7, 2023

n	time(s)
8	3.00E-07
16	2.00E-07
32	2.00E-07
64	3.00E-07
128	6.00E-07
256	1.10E-06
512	2.20E-06
1024	4.40E-06
2048	8.70E-06
4096	1.72E-05
8192	3.48E-05
16384	6.89E-05
32768	1.38E-04
65536	0.0002769
131072	0.0006163
262144	0.001098
524288	0.0024542

Meng Wai Chan
May 7, 2023

Conventional Dot Product (Automatic Parallelization, and Automatic Vectorization)

We changed the settings within Visual Studio to activate automatic parallelization and vectorization. Note that we are using AVX2 to optimize our initial program. As you can see from below the assembly code within the dot product function has changed and is more efficient compare to the conventional method.

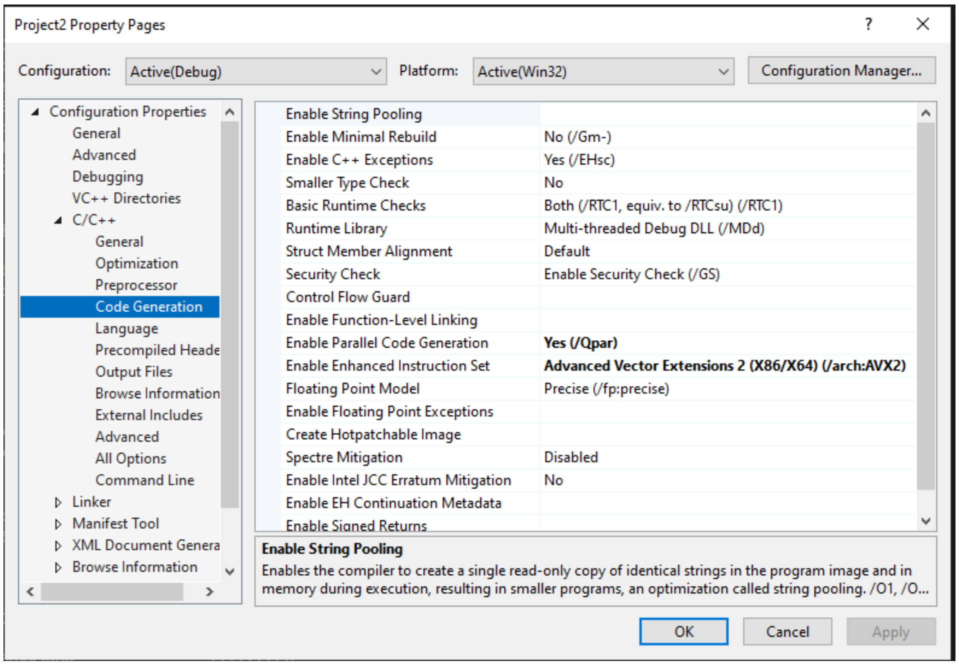


Figure 6: Visual Studio Settings

```
; Line 5
    mov DWORD PTR _i$1[ebp], 0
    jmp SHORT $LN4@dotproduct
$LN2@dotproduct:
    mov eax, DWORD PTR _i$1[ebp]
    add eax, 1
    mov DWORD PTR _i$1[ebp], eax
$LN4@dotproduct:
    mov eax, DWORD PTR _i$1[ebp]
    cmp eax, DWORD PTR _N$[ebp]
    jge SHORT $LN3@dotproduct
; Line 6
    mov eax, DWORD PTR _i$1[ebp]
    mov ecx, DWORD PTR _a$[ebp]
    mov edx, DWORD PTR _i$1[ebp]
    mov esi, DWORD PTR _b$[ebp]
    vmovss  xmm0, DWORD PTR [ecx+eax*4]
    vmulss  xmm0, xmm0, DWORD PTR [esi+edx*4]
    vmovss  xmm1, DWORD PTR _x$[ebp]
    vaddss  xmm0, xmm1, xmm0
    vmovss  DWORD PTR _x$[ebp], xmm0
; Line 7
    jmp SHORT $LN2@dotproduct
$LN3@dotproduct:
; Line 9
```

```
Array Size: 8
Start Value: 766722711801
End Value: 766722716665
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 4864 counts.
Total time: 0.0004864 secs.

Array Size: 16
Start Value: 766722736561
End Value: 766722740319
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 3758 counts.
Total time: 0.0003758 secs.

Array Size: 32
Start Value: 766722791919
End Value: 766722793483
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 1564 counts.
Total time: 0.0001564 secs.
```

Figure 7: AVX2 Output

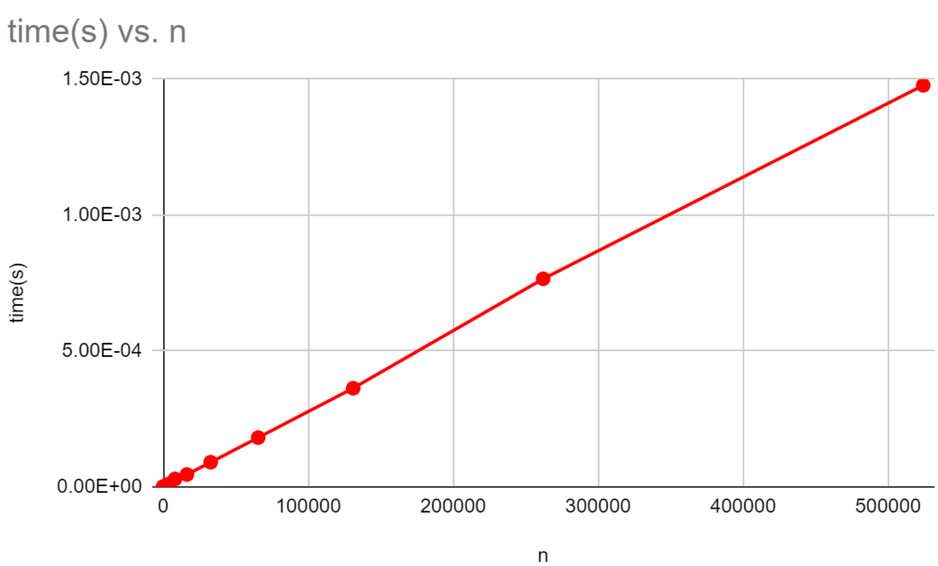


Figure 8: AVX2 Runtime graph

n	time(s)
8	4.00E-07
16	1.00E-07
32	4.00E-07
64	3.00E-07
128	5.00E-07
256	8.00E-07
512	1.50E-06
1024	2.90E-06
2048	5.80E-06
4096	1.14E-05
8192	2.93E-05
16384	4.55E-05
32768	9.08E-05
65536	0.0001815
131072	0.000363
262144	0.0007657
524288	0.0014775

Dot Product (Manual Optimization using vector instructions)

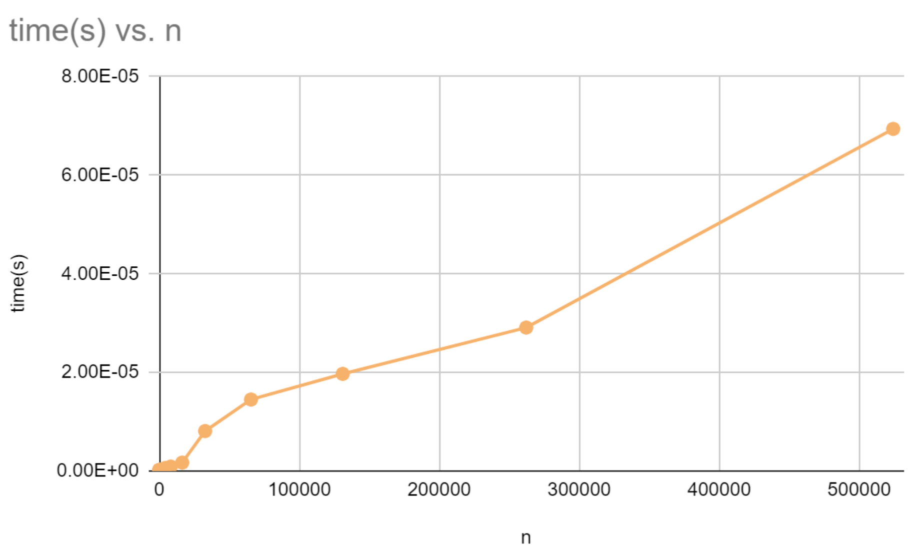
Now we optimize it by using our own assembly code to compile and generate a more efficiently as you can see from the graph below, by fixing the code generated by AVX2 reducing redundancy and problems that can affect runtime, making it more efficient when comparing to the previous methods.

```
float dotproduct_asm(float* a, float* b, int N) {
    float x = 0.0;
    _asm {
        vxorps ymm0, ymm0, ymm0;
        vxorps ymm1, ymm1, ymm1;
        vxorps ymm2, ymm2, ymm2;
        vxorps ymm3, ymm3, ymm3;

        mov eax, dword ptr[a]
        mov ebx, dword ptr[b]
        mov ecx, N

       mainloop :
       vmovups ymm1, [eax]
       vmovups ymm2, [ebx]

       vmulps ymm3, ymm1, ymm2
       vaddps ymm0, ymm3, ymm0

       add eax, 32
       add ebx, 32
       sub ecx, 8
       jnz mainloop

       vhaddps ymm0, ymm0, ymm0
       vhaddps ymm0, ymm0, ymm0

       vperm2f128 ymm3, ymm0, ymm0, 1
       vaddps ymm0, ymm3, ymm0
       vextractI128 xmm3, ymm0, 1
       movss dword ptr[x], xmm3
    }
    return x;
}
```

```
; Line 13
    vxorps  xmm0, xmm0, xmm0
    vmovss  DWORD PTR _x$[ebp], xmm0
; Line 15
    vxorps  ymm0, ymm0, ymm0
; Line 16
    vxorps  ymm1, ymm1, ymm1
; Line 17
    vxorps  ymm2, ymm2, ymm2
; Line 18
    vxorps  ymm3, ymm3, ymm3
; Line 20
    mov eax, DWORD PTR _a$[ebp]
; Line 21
    mov ebx, DWORD PTR _b$[ebp]
; Line 22
    mov ecx, DWORD PTR _N$[ebp]
$mainloop$3:
; Line 25
    vmovups ymm1, YMMWORD PTR [eax]
; Line 26
```

Meng Wai Chan
May 7, 2023

```
Array Size: 8
Start Value: 768316211365
End Value: 768316214274
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 2909 counts.
Total time: 0.0002909 secs.

Array Size: 16
Start Value: 768316228000
End Value: 768316228615
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 615 counts.
Total time: 6.15e-05 secs.

Array Size: 32
Start Value: 768316241999
End Value: 768316242787
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 788 counts.
Total time: 7.88e-05 secs.
```



n	time(s)
8	2.00E-07
16	2.00E-07
32	2.00E-07
64	2.00E-07
128	1.00E-07
256	2.00E-07
512	1.00E-07
1024	2.00E-07
2048	3.00E-07
4096	6.00E-07
8192	9.00E-07
16384	1.70E-06
32768	8.10E-06
65536	1.45E-05
131072	1.97E-05
262144	2.91E-05
524288	6.94E-05

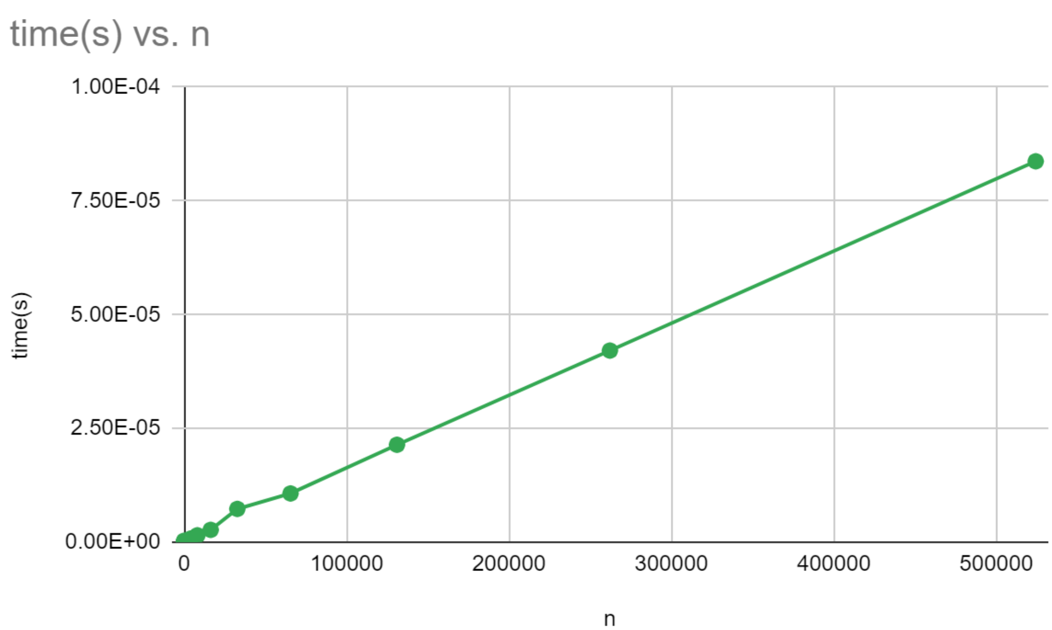
Dot Product (Optimized using DPPS Vector Instruction)

Finally, we optimized the dot product assembly code using the DPPS vector instruction, which essentially computes the dot product. So theoretically it should be faster as we do not need to perform addition and multiplication operations separately, instead it could be done all at once. My code that I wrote using the DPPS instruction is shown below.

```
47 float dotproduct_dpps(float* a, float* b, int N) {
48     float x = 0.0;
49     _asm {
50         vxorps ymm0, ymm0, ymm0;
51         vxorps ymm1, ymm1, ymm1;
52         vxorps ymm2, ymm2, ymm2;
53         vxorps ymm3, ymm3, ymm3;
54 
55         mov eax, dword ptr[a]
56         mov ebx, dword ptr[b]
57         mov ecx, N
58 
59         mainloop :
60         vmovups ymm0, [eax]
61         vmovups ymm1, [ebx]
62 
63         vdpps ymm2, ymm0, ymm1, 0xFF
64         vaddps ymm3, ymm2, ymm3
65         add eax, 32
66         add ebx, 32
67         sub ecx, 8
68         jnz mainloop
69 
70         vperm2f128 ymm0, ymm3, ymm3, 1
71         vaddps ymm3, ymm0, ymm3
72         vextractI128 xmm3, ymm3, 1
73         movss dword ptr[x], xmm3
74     }
75 
76     return x;
77 }
```

```
; Line 48
    vxorps  xmm0, xmm0, xmm0
    vmovss  DWORD PTR _x$[ebp], xmm0
; Line 50
    vxorps  ymm0, ymm0, ymm0
; Line 51
    vxorps  ymm1, ymm1, ymm1
; Line 52
    vxorps  ymm2, ymm2, ymm2
; Line 53
    vxorps  ymm3, ymm3, ymm3
; Line 55
    mov eax, DWORD PTR _a$[ebp]
; Line 56
    mov ebx, DWORD PTR _b$[ebp]
; Line 57
    mov ecx, DWORD PTR _N$[ebp]
```

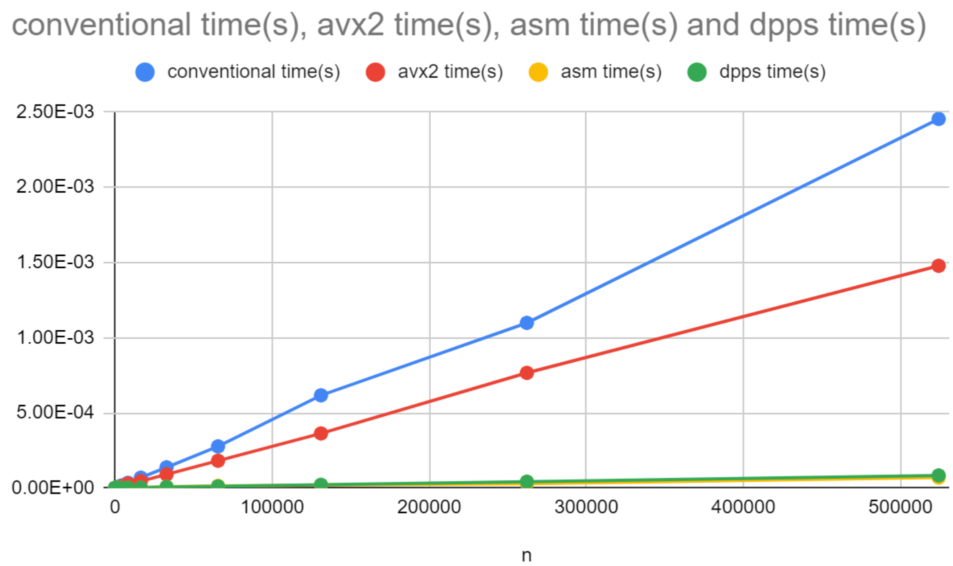
```
Array Size: 8
Start Value: 768811300958
End Value: 768811305735
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 4777 counts.
Total time: 0.0004777 secs.

Array Size: 16
Start Value: 768811320951
End Value: 768811321929
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 978 counts.
Total time: 9.78e-05 secs.

Array Size: 32
Start Value: 768811344297
End Value: 768811345247
QueryPerformanceFrequency: 10000000 counts per s.
QueryPerformanceCounter minimum resolution: 1/10000000 s.
begin - end: 950 counts.
Total time: 9.5e-05 secs.

Array Size: 64
Start Value: 768811358058
```



n	time(s)
8	2.00E-07
16	3.00E-07
32	1.00E-07
64	1.00E-07
128	1.00E-07
256	2.00E-07
512	2.00E-07
1024	3.00E-07
2048	4.00E-07
4096	8.00E-07
8192	1.50E-06
16384	2.70E-06
32768	7.30E-06
65536	1.07E-05
131072	2.14E-05
262144	4.21E-05
524288	8.37E-05

All Plots Comparison

The following is the comparison of all four methods as you can see the manual optimization and dpps is so much faster compared to the other two.

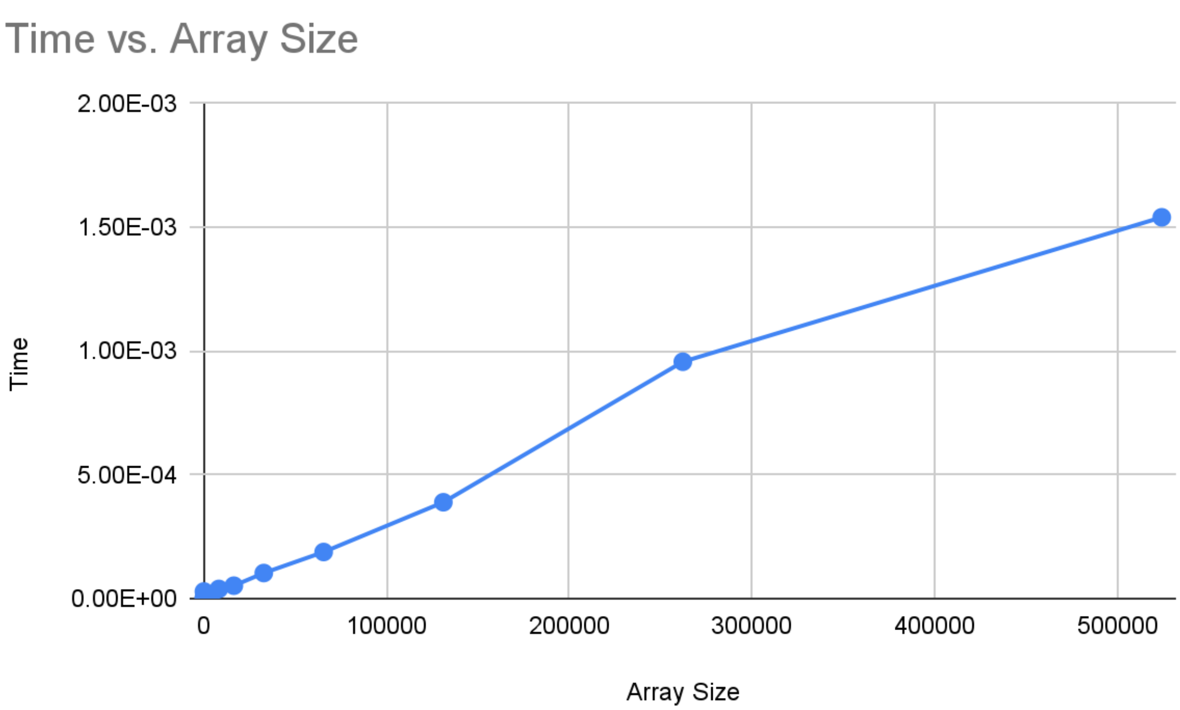


III. AMD x86 GCC compiler in Linux

Conventional Dot Product (No Optimization)

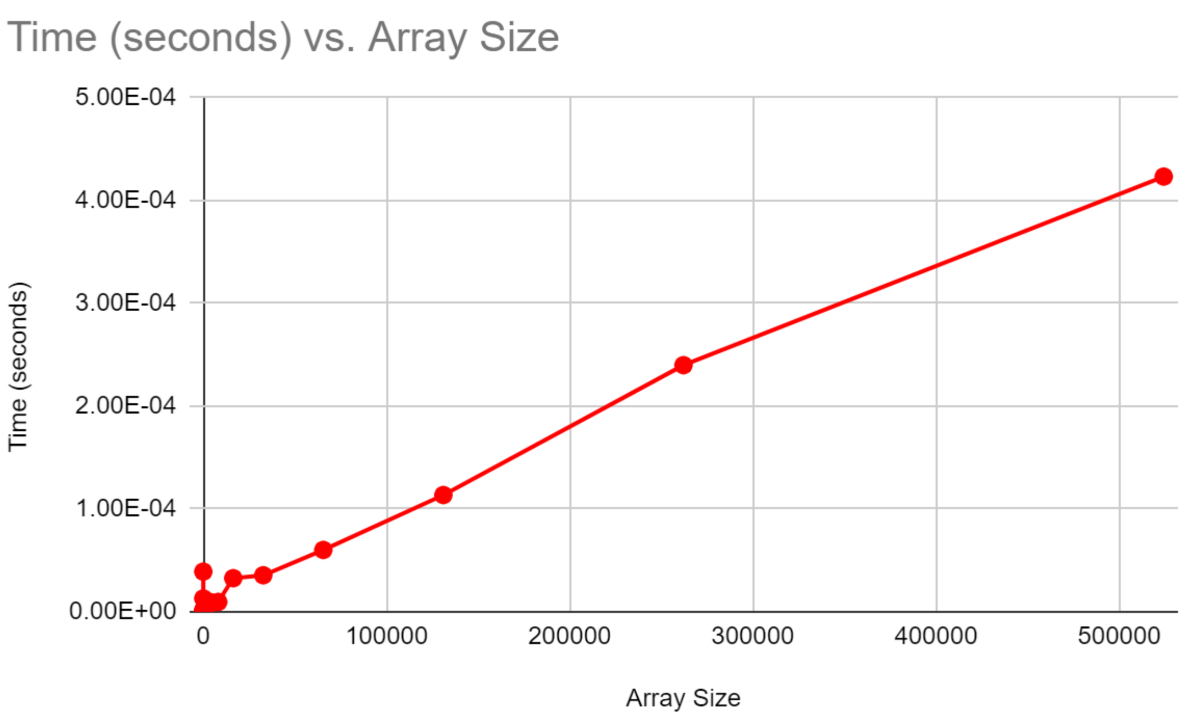
```cpp
#include <iostream>
#include "dotproduct.h"
#include <math.h>
#include <chrono>

using namespace std;

int main() {

    int N = 0;
    int MAX = 19;

    for (int i = 3; i <= MAX; i++) {
        N = (int)pow(2, i);

        float* arr1 = new float[N];
        float* arr2 = new float[N];


        for (int j = 0; j < N; j++) {
            arr1[j] = 8.12;
            arr2[j] = 8.14;
        }

        cout<<"Vector Size: "<<N <<endl;
        auto start = chrono::high_resolution_clock::now();

        dotproduct(arr1, arr2, N);

        auto end = chrono::high_resolution_clock::now();

        chrono::duration<double> diff = end - start;

        cout<<"Total Time: " <<diff.count() <<"seconds."<<endl;

        cout<<endl;
    }

    return 0;
}
```

```
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ g++ main.cpp -o main
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ ./main
Vector Size: 8
Total Time: 1.4e-07seconds.

Vector Size: 16
Total Time: 1.1e-07seconds.

Vector Size: 32
Total Time: 1.4e-07seconds.

Vector Size: 64
Total Time: 2.1e-07seconds.

Vector Size: 128
Total Time: 3.8e-07seconds.
```



Conventional Dot Product (Automatic Parallelization, and Automatic Vectorization)

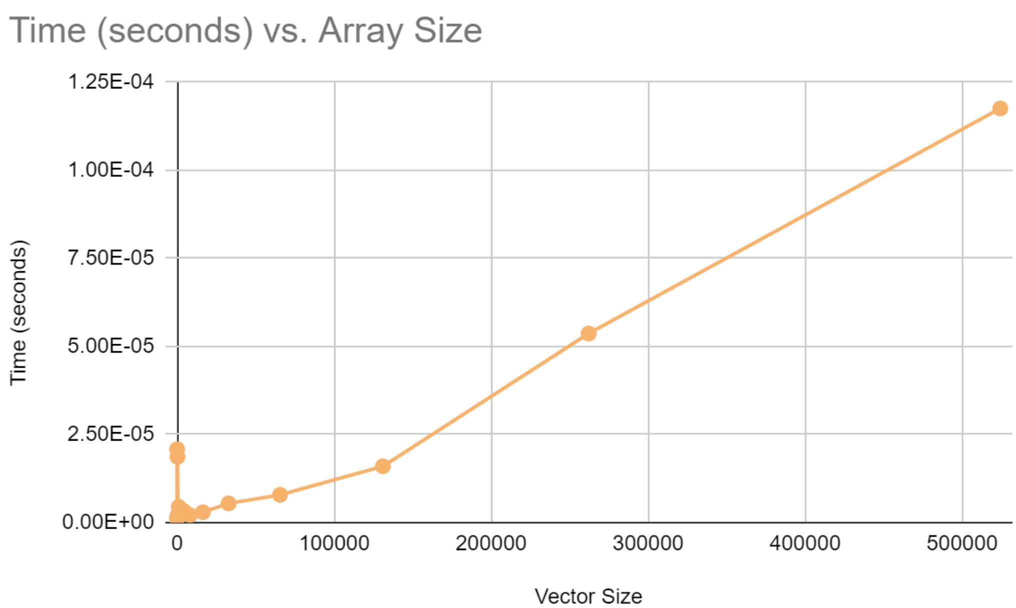
```
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ g++ main.cpp -O3 -ftree-parallelize-loops=4 -o main~
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ ./main
Vector Size: 8
Total Time: 1.7e-07seconds.

Vector Size: 16
Total Time: 1.3e-07seconds.

Vector Size: 32
Total Time: 1.1e-07seconds.

Vector Size: 64
Total Time: 2e-07seconds.

Vector Size: 128
Total Time: 3.8e-07seconds.

Vector Size: 256
Total Time: 7.3e-07seconds.
```



Dot Product (Manual Optimization using vector instructions)

```
float dotproduct_asm(float* a, float* b, int N) {
    float x = 0.0;
    __asm (
        "vpxor %ymm0, %ymm0, %ymm0\n"
        "vpxor %ymm3, %ymm3, %ymm3\n"

        ".mainloop:\n"
        "vmovups 0x0 (%rdi), %ymm1\n"
        "vmovups 0x0 (%rsi), %ymm2\n"

        "vmulps %ymm1, %ymm2, %ymm3\n"
        "vaddps %ymm0, %ymm3, %ymm0\n"

        "add $32, %rdi\n"
        "add $32, %rsi\n"
        "sub $8, %rdx\n"
        "jnz .mainloop\n"

        "vhaddps %ymm0, %ymm0, %ymm0\n"
        "vhaddps %ymm0, %ymm0, %ymm0\n"
        "vhaddps %ymm0, %ymm0, %ymm0\n"

        "vmovups %ymm0, (%rcx) \n"

    );
    return x;
}
```

```
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ g++ main.cpp -o main
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ ./main
Vector Size: 8
Total Time: 1.1e-07seconds.

Vector Size: 16
Total Time: 1.1e-07seconds.

Vector Size: 32
Total Time: 1e-07seconds.

Vector Size: 64
Total Time: 8e-08seconds.

Vector Size: 128
Total Time: 7e-08seconds.

Vector Size: 256
Total Time: 9e-08seconds.
```

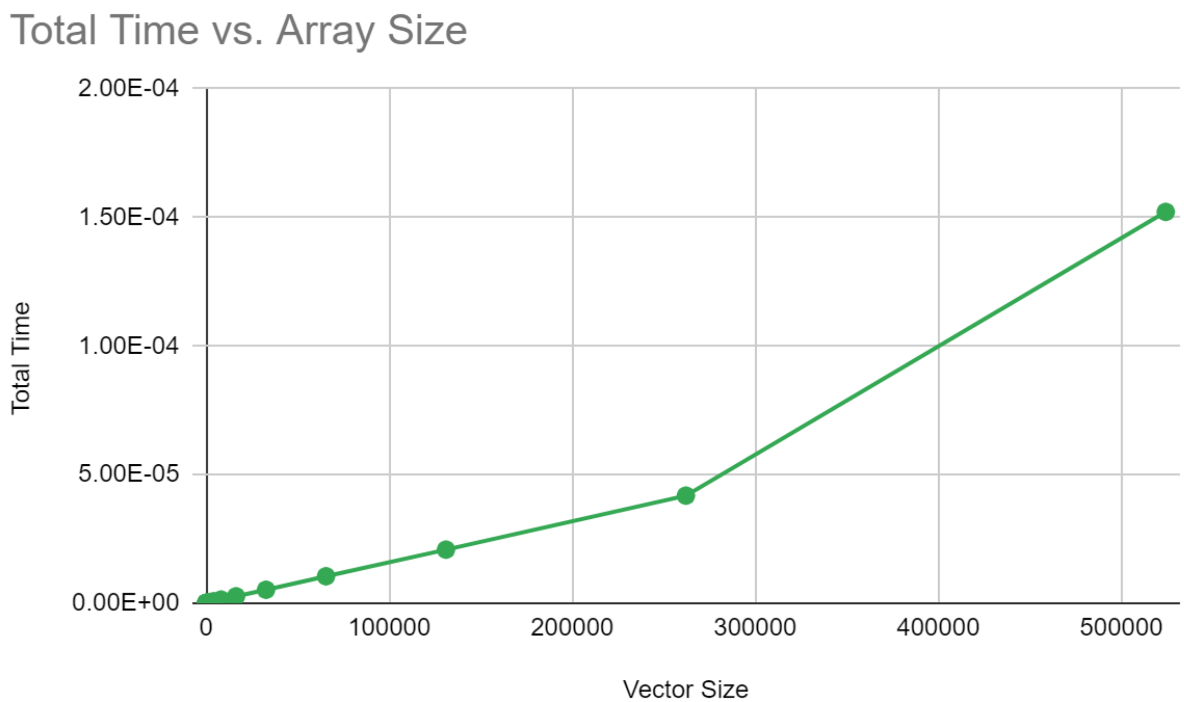


Meng Wai Chan
May 7, 2023

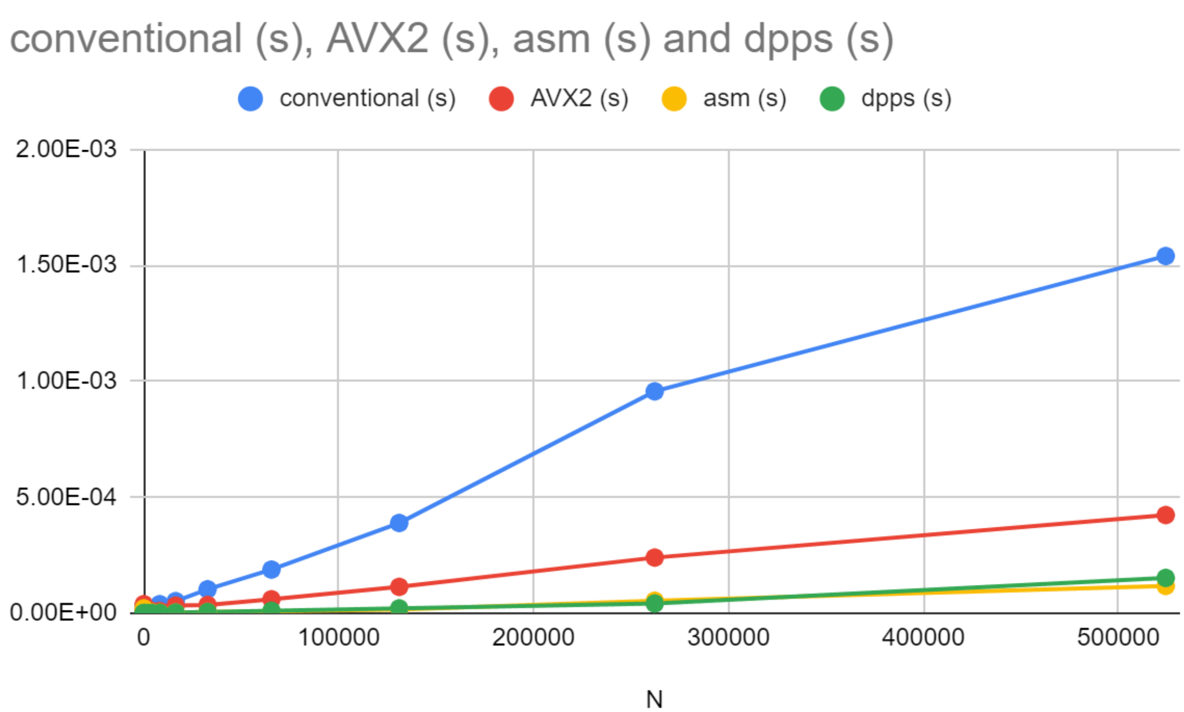
Dot Product (Optimized using DPPS Vector Instruction)

```
float dotproduct_dpps(float* a, float* b, int N) {
    float x = 0.0;
    __asm (
        "vpxor %ymm3, %ymm3, %ymm3\n"

        ".main:\n"
        "vmovups 0x0 (%rdi), %ymm1\n"
        "vmovups 0x0 (%rsi), %ymm2\n"
        "vdpps $0xFF, %ymm1, %ymm2, %ymm0\n"
        "vaddps %ymm0, %ymm3, %ymm3\n"

        "add $32, %rdi\n"
        "add $32, %rsi\n"
        "sub $8, %rdx\n"
        "jnz .main\n"

        "vhaddps %ymm3, %ymm3, %ymm3\n"

        "vmovups %ymm3, (%rcx) \n"
    );

    return x;
}
```

```
mengwai@mengwai-VirtualBox:~/Desktop/VSCode/csc342$ ./main
Vector Size: 8
Total Time: 1.6e-07seconds.

Vector Size: 16
Total Time: 1e-07seconds.

Vector Size: 32
Total Time: 7e-08seconds.

Vector Size: 64
Total Time: 8e-08seconds.

Vector Size: 128
Total Time: 1.3e-07seconds.

Vector Size: 256
Total Time: 1.5e-07seconds.
```



All Plots Comparison



IV. Conclusion

In this lab I learned about vector instructions and how they can be used to significantly improve the performance of a program. By using both Intel X86 32-bit compiler and Intel X86 64-bit GCC compiler in Linux, I learned how to vectorize and parallelize my code, even understanding how to automatically do it if the need arises. I also learned more about my processor and its capabilities for computation.