Take Home Test #3

Dot Product Optimization

Anthony Ramos

CSC342/43

May 16th, 2021

Professor Isidor Gertner

Contents

Objective 3
Introduction 3
My CPU Info 3
Dot Product Program 5
Test File 5
Dot Product (Array Indexing) (Not Optimized) 6
Dot Product (Array Pointer Arithmetic) (Not Optimized) 7
Enabling Compiler Optimizations 8
Dot Product (Array Indexing) (Compiler Optimized) 9
Dot Product (Array Pointer Arithmetic) (Compiler Optimized) 10
Comparing Dot Product Assembler Source 11
Dot Product (Vector Instructions) (Pointer) 14
Optimizing Dot Product in A Linux Environment 16
Enabling Compiler Optimization Flags 16
Test File 17
Comparing Assembly Source Files 19
Results 22

Objective

The objective of this assignment is to implement a program to compute the dot product using array indexing and pointer arithmetic. Next, optimization features will be applied (by software) followed by a manually optimized approach on the compiler generated assembly code. In all three cases, the execution times will be measured and analyzed. The simulation environments utilized were Intel X86 32-Bit compiler in MS Visual Studio and Intel X86 64-Bit GCC compiler in a Linux (18.04.5) environment.

Introduction

My CPU Info

Prior writing and analyzing any code, it was of interest to learn about the CPU capabilities in my hardware. To identify my hardware's processor, I utilized the *CPUID* application. Figure 1a shows information regarding my processor along with the instruction sets that are available (in red) for executing.

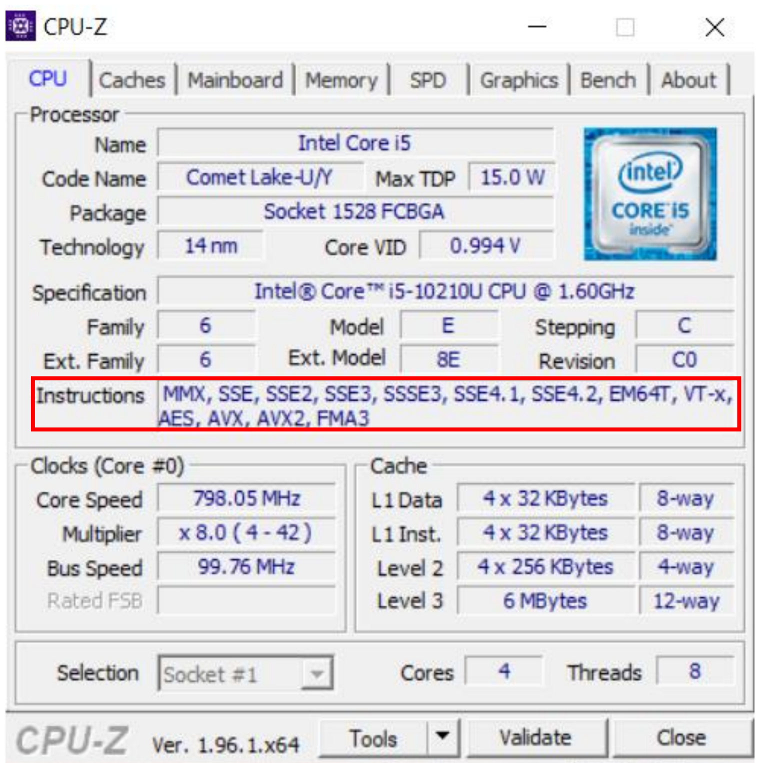


Figure 1a: My CPU Processor Info from CPUID Application

To confirm that my processor was properly detected, I checked for it in Linux. It can be confirmed that my processor is a 10th Generation Intel Core I5.

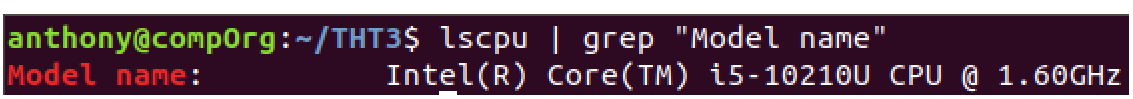


Figure 1b: My CPU Processor Info from CPUID Application

To determine my processor's capabilities, I did some research online and found data regarding my processor's performance in several categories as well as its performance relative to other popular processors.

CPU Test Suite Average Results for Intel Core i5-10210U @ 1.60GHz	
Integer Math	23,993 MOps/Sec
Floating Point Math	14,603 MOps/Sec
Find Prime Numbers	18 Million Primes/Sec
Random String Sorting	12 Thousand Strings/Sec
Data Encryption	2,176 MBytes/Sec
Data Compression	84.7 MBytes/Sec
Physics	424 Frames/Sec
Extended Instructions	5,513 Million Matrices/Sec
Single Thread	2,268 MOps/Sec

From submitted results to PerformanceTest V10 as of 13th of May 2021.

Figure 1c: My CPU Processor Capabilities

CPU Mark Relative to Top 10 Common Laptop CPUs

As of 14th of May 2021 - Higher results represent better performance

Processor	Average CPU Mark
AMD Ryzen 7 4800H	19,231
Apple M1 8 Core 3200 MHz	15,037
Intel Core i7-10750H @ 2.60GHz	12,573
Intel Core i7-9750H @ 2.60GHz	11,323
AMD Ryzen 5 4500U	11,264
Intel Core i7-8750H @ 2.20GHz	10,177
AMD Ryzen 5 3500U	7,147
Intel Core i7-7700HQ @ 2.80GHz	6,955
Intel Core i5-10210U @ 1.60GHz	6,497
Celeron Dual-Core T3500 @ 2.10GHz	786
Celeron Dual-Core T3100 @ 1.90GHz	531

Figure 1d: My CPU Processor Performance among Popular Processors

Dot Product Program

Two programs to compute the dot product will be written using two programming structures. One program will implement the dot product by utilizing array index notation as shown in figure 2a. The second will utilize pointer arithmetic as shown in figure 2b.

```
int DotProduct_Index(int a1[], int a2[], int n) {

	int sum = 0;

	for (int i = 0; i < n; i++)
		sum += (a1[i] * a2[i]);
	return sum;
}
```

Figure 2a: Dot Product Program (Array Index Notation)

```
int DotProduct_Pointer(int *a1, int *a2, int n) {

	int *a, *b;
	int sum = 0;

	for (a = &a1[0], b = &a2[0]; a < &a1[n]; a++, b++)
		sum += ((*a) * (*b));
	return sum;
}
```

Figure 2b: Dot Product Program (Pointer Arithmetic)

Test File

A third program (test file) will be utilized to serve as the *main* program that will call the previous two programs (one per run). It should be noted we care not for the actual result of the dot product. What matters is the performance (i.e., execution time of each program). To measure the execution time, we will utilize a highly accurate timing mechanism called the `QueryPerformanceCounter()` function.

```
if (QueryPerformanceCounter((LARGE_INTEGER*)&timeStart) != 0) {
		result = DotProduct_Index(arr1, arr2, arraySize);
		//result = DotProduct_Pointer(&arr1[0], &arr2[0], arraySize);
		QueryPerformanceCounter((LARGE_INTEGER*)&timeEnd);

		QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

		cout << "QueryPerformance frequency: " << freq << " counts per second" << 
endl;
		cout << "Array size: " << arraySize << endl;
		cout << "Elapsed Time: " << (timeEnd - timeStart) * 1.0 / freq << " 
seconds" << endl;
	}
	else {
		DWORD dwError = GetLastError();
		cout << "Error = " << dwError << endl;
	}
```

Figure 3: QueryPerformanceCounter()

Dot Product (Array Indexing) (Not Optimized)

In the first round of simulations, we disable some optimization features (*Automatic Parallelization* & *Automatic Vectorization*). Utilizing the `QueryPerformanceCounter()`, several measurements were taken to evaluate the program's performance. The average execution times of multiple runs for array sizes of $n = 2^4$ to 2^{20} will then be plotted. Figure 5 shows the plotted results are linear which is expected.

Array Size	Execution Time
16	0.00000036
32	0.00000041
64	0.00000054
128	6.71429E-07
256	0.00000124
512	0.00000198
1024	0.00000302
2048	0.00000616
4096	0.00001308
8192	0.00002178
16384	3.93556E-05
32768	0.00008305
65536	0.0001780
131072	0.0002889
262144	0.0007395
524288	0.001501317
1048576	0.00270965

Table 1: Execution times of dot product program via array indexing (Not-Optimized)

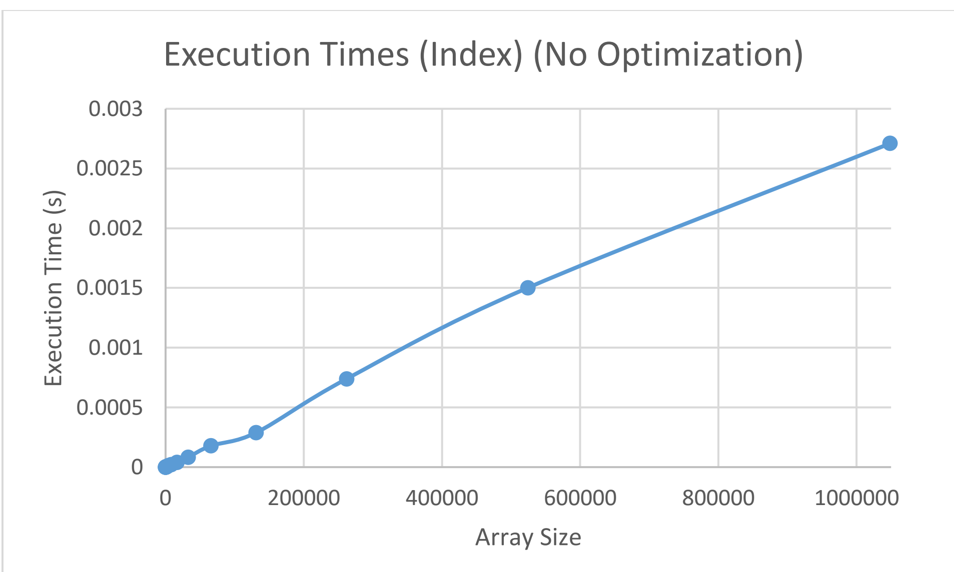


Figure 4: Execution Times of Dot Product Via Array Indexing (No Optimization)

Dot Product (Array Pointer Arithmetic) (Not Optimized)

The previous process is repeated for the second program using pointer arithmetic. The execution times are shown in table 2 and plotted results in figure 6. Again, the expected plot is linear.

Array Size	Execution Time
16	0.00000034
32	0.0000004
64	0.00000052
128	0.00000072
256	0.00000096
512	2.1660E-06
1024	3.6625E-06
2048	6.1375E-06
4096	1.38333E-05
8192	2.90111E-05
16384	0.00004735
32768	9.54111E-05
65536	1.92E-04
131072	4.52E-04
262144	8.12E-04
524288	1.65E-05
1048576	3.31E-05

Table 2: Execution times of dot product program via array pointer arithmetic (Not-Optimized)

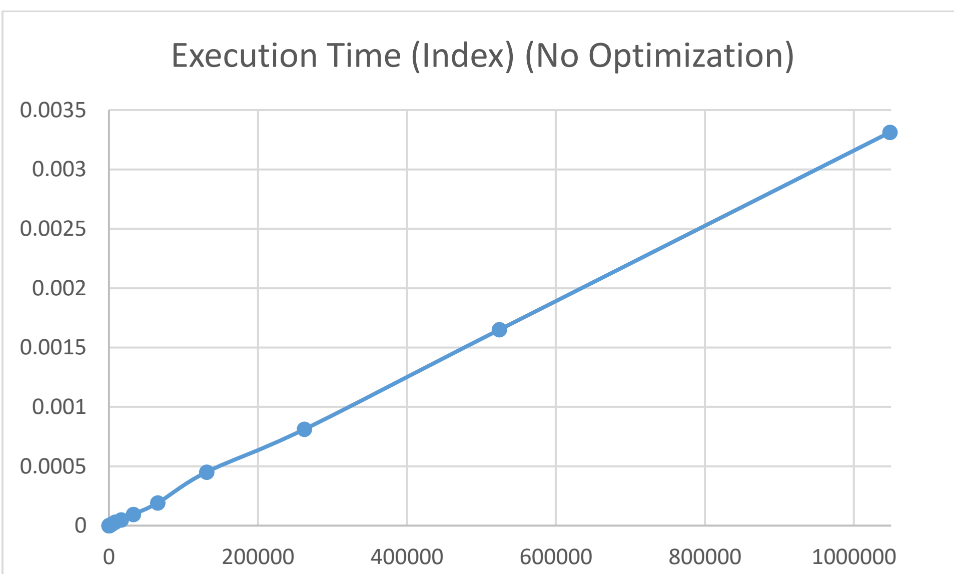


Figure 6: Execution Times of Dot Product Via Array Indexing (No Optimization)

Enabling Compiler Optimizations

Next, we wish to improve the execution time of each program by enabling the aforementioned optimization settings within Visual Studio. Further information regarding these settings can be found here.

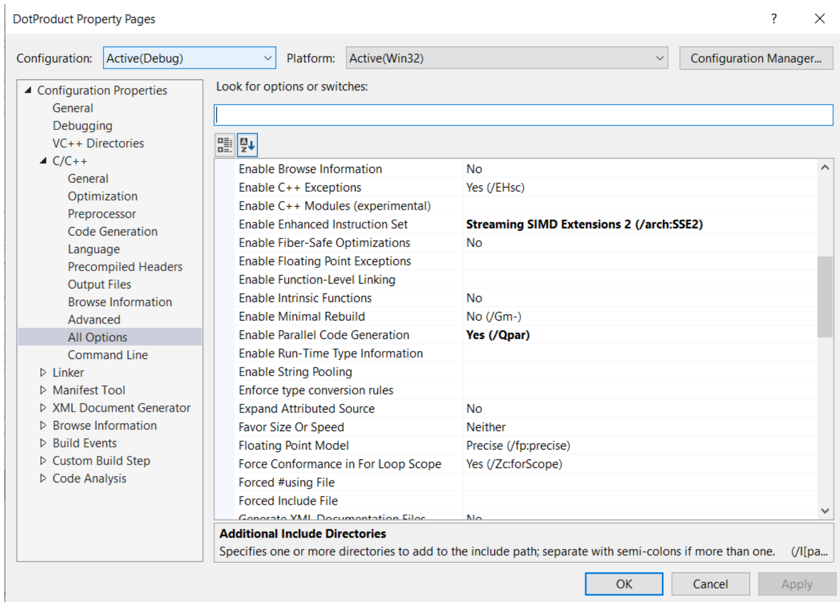


Figure 7: Enabling Optimization Features in Visual Studio

Dot Product (Array Indexing) (Compiler Optimized)

With optimization settings set, we repeat the same measurements for array sizes $n = 2^4$ to 2^{20} and tabulate the results as shown in table 3. Figure 8 shows the plotted results.

Array Size	Execution Time
16	0.000000325
32	0.00000039
64	0.00000052
128	0.00000065
256	0.00000111
512	0.00000167
1024	0.00000297
2048	0.00000502
4096	0.0000126
8192	0.00001731
16384	0.00003764
32768	0.00007569
65536	0.000171963
131072	0.00024365
262144	0.00065296
524288	0.001303629
1048576	0.002459667

Table 3: Execution Times of Dot Product Program Via Array Indexing (Compiler-Optimized)

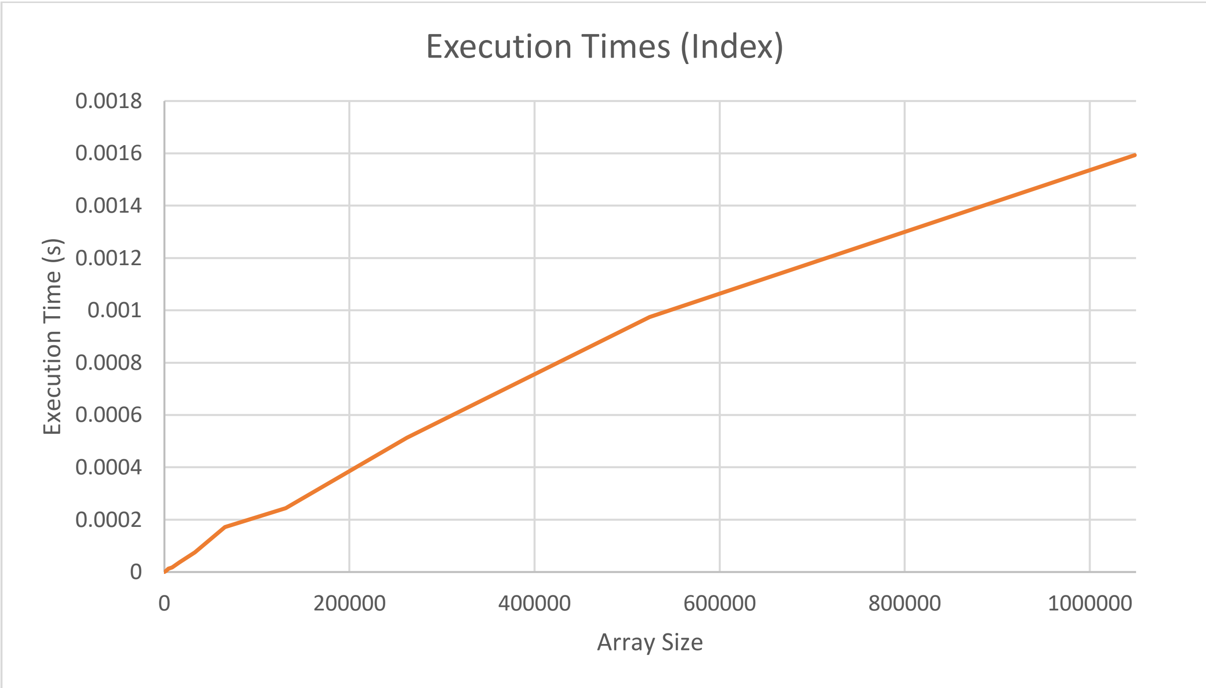


Figure 8: Execution Times of Dot Product Via Array Indexing (Compiler-Optimized)

Dot Product (Array Pointer Arithmetic) (Compiler Optimized)

Using the same optimization settings, we repeat the previous process and measure the execution times as described in table 4. The results are plotted in figure 9.

Array Size	Execution Time
16	0.0000003
32	0.00000042
64	0.00000042
128	0.00000066
256	0.00000096
512	0.00000153
1024	0.000002825
2048	0.000005825
4096	0.000009625
8192	2.17375E-05
16384	0.000038475
32768	8.09875E-05
65536	0.00018367
131072	0.0004259
262144	0.000789625
524288	0.001547517
1048576	0.00245714

Table 4: Execution Times of Dot Product Program Via Pointer Arithmetic (Compiler-Optimized)

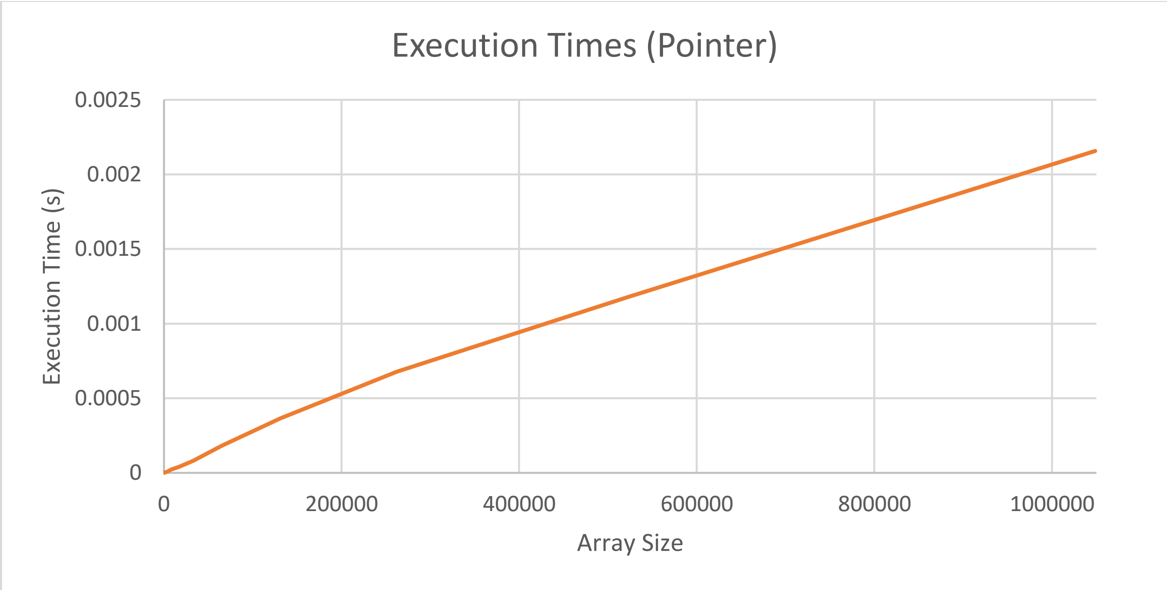


Figure 9: Execution Times of Dot Product Via Pointer Arithmetic (Compiler-Optimized)

Comparing Dot Product Assembler Source

The application of optimization settings often leads to a reduction in execution time (in our case). While the high-level programming remains unchanged, the assembly source is altered to yield better performance. Figure 10a shows the partial assembly code for the dot product program via array indexing for when no optimization (left) and optimization (right) is applied. It can be seen that, with optimization settings, the assembly source becomes partially vectorized. Although the assembly becomes longer and can be more difficult to debug, it greatly improves the programs performance by reducing execution time.

```
    int sum = 0;
00B11743  mov         dword ptr [sum],0
    #pragma loop(hint_parallel(8))
    for (int i = 0; i < n; i++)
00B1174A  mov         dword ptr [ebp-8],0
00B11751  jmp         DotProduct_Index+2Ch (0B1175Ch)
00B11753  mov         eax,dword ptr [ebp-8]
00B11756  add         eax,1
00B11759  mov         dword ptr [ebp-8],eax
00B1175C  mov         eax,dword ptr [ebp-8]
00B1175F  cmp         eax,dword ptr [n]
00B11762  jge         DotProduct_Index+4Fh (0B1177Fh)
        sum += (a1[i] * a2[i]);
00B11764  mov         eax,dword ptr [ebp-8]
00B11767  mov         ecx,dword ptr [a1]
00B1176A  mov         edx,dword ptr [ebp-8]
00B1176D  mov         esi,dword ptr [a2]
00B11770  mov         eax,dword ptr [ecx+eax*4]
00B11773  imul        eax,dword ptr [esi+edx*4]
00B11777  add         eax,dword ptr [sum]
00B1177A  mov         dword ptr [sum],eax
00B1177D  jmp         DotProduct_Index+23h (0B11753h)
    return sum;
00B1177F  mov         eax,dword ptr [sum]
}
```

```
    #pragma loop(hint_parallel(8))
    for (int i = 0; i < n; i++)
00221755  cmp         edi,8
00221758  jb          DotProduct_Index+0B6h (02217E6h)
0022175E  cmp         dword ptr [__isa_available (0BC2AD4h)],2
00221765  jl          DotProduct_Index+0B6h (02217E6h)
0022176B  mov         ecx,edi
0022176D  and         ecx,80000007h
00221773  jns         DotProduct_Index+4Ah (022177Ah)
00221775  dec         ecx
00221776  or          ecx,0FFFFFFF8h
00221779  inc         ecx
0022177A  mov         edx,dword ptr [a1]
0022177D  mov         esi,edi
0022177F  mov         ebx,dword ptr [a2]
00221782  sub         esi,ecx
00221784  mov         dword ptr [ebp-4],ebx
00221787  xorps       xmm3,xmm3
0022178A  sub         dword ptr [ebp-4],edx
0022178D  xorps       xmm2,xmm2
00221790  lea         ecx,[edx+10h]
00221793  mov         edx,dword ptr [ebp-4]
00221796  movups      xmm0,xmmword ptr [ebx+eax*4]
0022179A  add         eax,8
0022179D  lea         ecx,[ecx+20h]
        sum += (a1[i] * a2[i]);
002217A0  movups      xmm1,xmmword ptr [ecx-30h]
002217A4  pmulld      xmm1,xmm0
002217A9  paddd       xmm3,xmm1
002217AD  movups      xmm0,xmmword ptr [edx+ecx-20h]
002217B2  movups      xmm1,xmmword ptr [ecx-20h]
002217B6  pmulld      xmm1,xmm0
002217BB  paddd       xmm2,xmm1
002217BF  cmp         eax,esi
002217C1  jl          DotProduct_Index+66h (0221796h)
002217C3  paddd       xmm2,xmm3
002217C7  movaps      xmm0,xmm2
002217CA  psrldq      xmm0,8
002217CF  paddd       xmm2,xmm0
002217D3  movups      xmm0,xmm2
002217D6  psrldq      xmm0,4
002217DB  paddd       xmm2,xmm0
002217DF  movd        esi,xmm2
002217E3  mov         dword ptr [sum],esi
```

Figure 10a: Assembly Source for Dot Product via Array Indexing (Partial)

A similar observation can be seen for the dot product via pointer arithmetic. Although, the compiler optimized assembly source did not utilize vector instructions. Instead, it optimized the assembly code by reducing the number of machine instructions required to perform the same task.

```
    int *a, *b;
    int sum = 0;
008317C3  mov         dword ptr [sum],0
     #pragma loop(hint_parallel(8))
    for (a = &a1[0], b = &a2[0]; a < &a1[n]; a++, b++)
008317CA  mov         eax,4
008317CF  imul        ecx,eax,0
008317D2  add         ecx,dword ptr [a1]
008317D5  mov         dword ptr [a],ecx
008317D8  mov         edx,4
008317DD  imul        eax,edx,0
008317E0  add         eax,dword ptr [a2]
008317E3  mov         dword ptr [b],eax
008317E6  jmp         DotProduct_Pointer+4Ah (08317FAh)
008317E8  mov         eax,dword ptr [a]
008317EB  add         eax,4
008317EE  mov         dword ptr [a],eax
008317F1  mov         ecx,dword ptr [b]
008317F4  add         ecx,4
008317F7  mov         dword ptr [b],ecx
008317FA  mov         eax,dword ptr [n]
008317FD  mov         ecx,dword ptr [a1]
00831800  lea         edx,[ecx+eax*4]
00831803  cmp         dword ptr [a],edx
00831806  jae         DotProduct_Pointer+6Bh (083181Bh)
        sum += ((*a) * (*b));
00831808  mov         eax,dword ptr [a]
0083180B  mov         ecx,dword ptr [b]
0083180E  mov         edx,dword ptr [eax]
00831810  imul        edx,dword ptr [ecx]
00831813  add         edx,dword ptr [sum]
00831816  mov         dword ptr [sum],edx
00831819  jmp         DotProduct_Pointer+38h (08317E8h)
    return sum;
0083181B  mov         eax,dword ptr [sum]
}
```

```
    int *a, *b;
    int sum = 0;
     #pragma loop(hint_parallel(8))
    for (a = &a1[0], b = &a2[0]; a < &a1[n]; a++, b++)
002A18E9  and         eax,3FFFFFFFh
002A18EE  cmp         eax,2
002A18F1  jb          DotProduct_Pointer+66h (02A1926h)
002A18F3  mov         eax,esi
002A18F5  sub         eax,ecx
002A18F7  mov         dword ptr [a1],eax
002A18FA  mov         edx,eax
002A18FC  nop         dword ptr [eax]
        sum += ((*a) * (*b));
002A1900  mov         eax,dword ptr [esi]
002A1902  add         esi,8
002A1905  imul        eax,dword ptr [ecx]
002A1908  add         edi,eax
002A190A  mov         eax,dword ptr [edx+ecx+4]
002A190E  imul        eax,dword ptr [ecx+4]
002A1912  add         ecx,8
002A1915  mov         edx,dword ptr [ebp+8]
002A1918  add         ebx,eax
002A191A  mov         eax,dword ptr [ebp+10h]
002A191D  add         eax,0FFFFFFFCh
002A1920  cmp         ecx,eax
002A1922  jb          DotProduct_Pointer+40h (02A1900h)
002A1924  xor         edx,edx
```

Figure 10b: Assembly Source for Dot Product via Pointer Arithmetic (Partial)

Figures 11a and 11b compare the execution times for both dot product programs. The results confirm our earlier analysis. Optimization settings did reduce the execution time for both cases. This of course, was most noticeable for very large array sizes.

Figure 11a: Comparison of Execution Times for Dot Product via Array Indexing

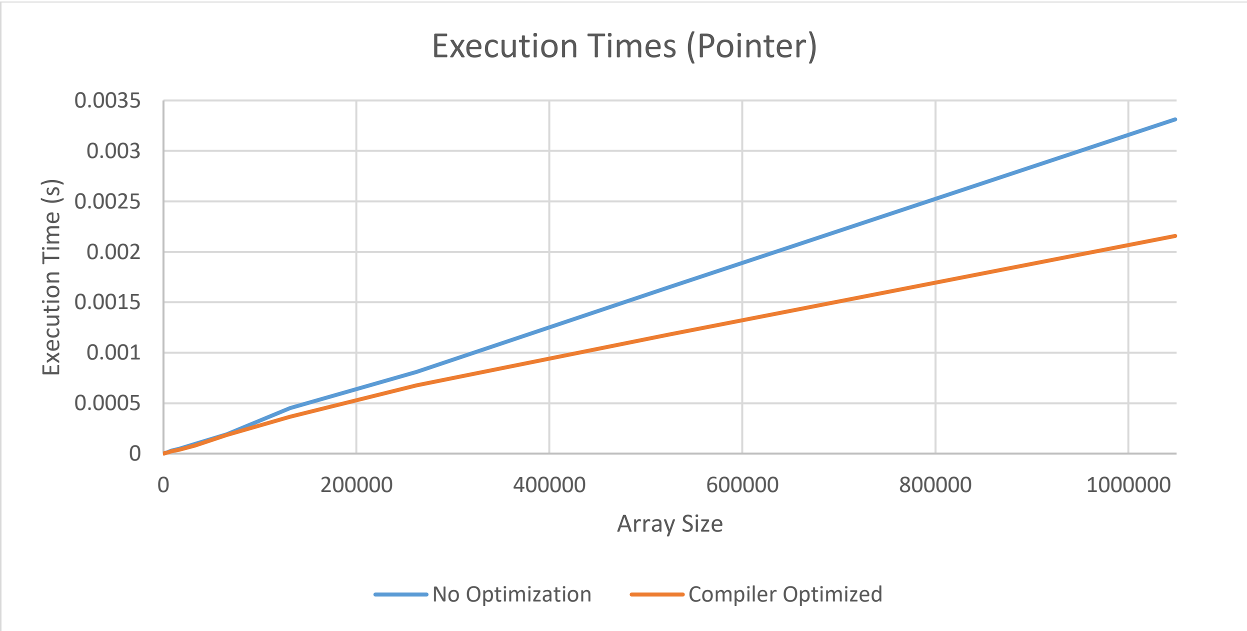


Figure 11b: Comparison of Execution Times for Dot Product via Pointer Arithmetic

Dot Product (Vector Instructions) (Pointer)

In measuring the execution times after applying built in optimization settings, it was now time to attempt to optimize the compiler generated code ourselves. However, difficulty arose, and I was not able to successfully optimize the existing code manually. Instead, I moved towards utilizing SSE3 vector instructions from previous Intel tutorial to implement the dot product from scratch whose code is shown below.

```
float DotProduct_Vector(float *a1, float *a2, int n) {
	float x = 0.0;
	_asm {
		pxor xmm0, xmm0; intialize xmm0 to 0 (xmm0 stores dot product result)

		mov eax, dword ptr[a1]; %eax points to arr1[]
		mov ebx, dword ptr[a2]; %ebx points to arr2[]
		mov ecx, dword ptr[n]; Array Size

		$MyLoop:
		movups xmm0, [eax]
			movups xmm1, [ebx]
			mulps  xmm1, xmm2; Compute arr1[i] * arr2[i]
			addps  xmm0, xmm1; Compute x + arr1[i] * arr2[i]

			add eax, 16
			add ebx, 16
			sub ecx, 4; Loop - 4
			jnz $MyLoop ; Loop if ecx is not zero

			haddps xmm0, xmm0 ; Horizontal Add
			haddps xmm0, xmm0 ; Horizontal Add
			movss dword ptr[x], xmm0 ; Store result in x
	}
	return 0;
}
```

Figure 12: Implementation of Dot Product Using Vector Instructions

We perform our previous measurements which yield the results in table 5.

Array Size	Execution Time
16	4.42857E-07
32	5.14286E-07
64	5.42857E-07
128	0.0000006
256	6.71429E-07
512	0.0000007
1024	1.06667E-06
2048	1.25556E-06
4096	0.0000022
8192	0.0000048
16384	8.27778E-06
32768	0.0000192
65536	3.98889E-05
131072	0.000105513
262144	0.000184338
524288	0.000427225
1048576	0.000837667

Table 5: Execution Times of Dot Product Program Via Pointer Arithmetic (SSE3 Vector Instructions)

Finally, the plot in figure 13 compares the previous plots with that obtained from the data in table 5. When applying the automatic parallelization and vectorization settings, the execution time is greatly reduced. This is most noticeable for larger inputs. The use of SSE3 vector instructions provided a significant advantage in reducing execution time. However, a similar implementation for the dot product via array indexing could not be realized.

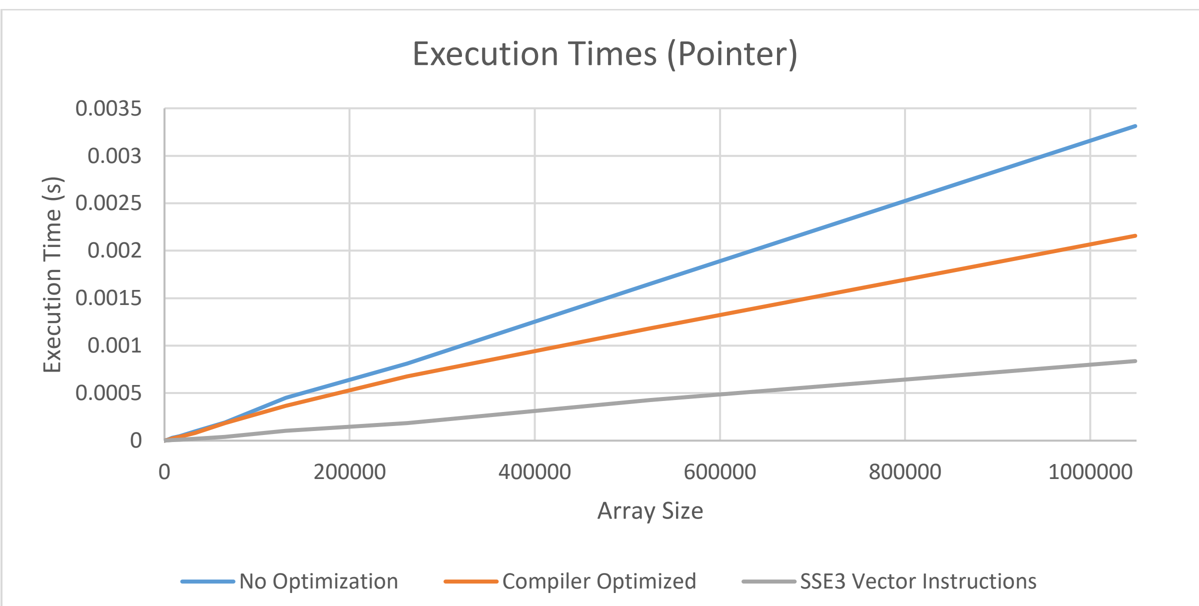


Figure 13: Summary of Dot Product (via Pointer Arithmetic) Execution Times

Optimizing Dot Product in A Linux Environment

We now move towards optimization in a Linux Environment. While one may attempt to manually optimize the compiler generated assembly code produced, GCC has readily available optimization features via *compiler flags*.

Enabling Compiler Optimization Flags

There are many optimization flags of which a complete listing can be found here. However, we will only consider the `-O` optimization flag. Specifically, we will consider the flags that reduce execution time of which there are three (and a no optimization default) and described in table 5 below. The '+' symbol indicates an increase an execution time while the '-' symbol indicates a decrease in execution time. The '--' and '---' indicate even further decreases in execution time.

Optimization Flag	Optimization Level	Execution Time	Notes
`-O0`	0 (Default)	+	No Optimization
`-O1`	1	-	Basic Optimization
`-O2`	2	--	Recommend Level
`-O3`	3 (Highest)	---	Optimization Not Guaranteed

Table 5: Summary of Optimization Flags

To set the optimization flag, we utilize the following syntax:

```
gcc -Olevel [source files] -o [output filename]
```

There are tradeoffs to applying these optimization flags – most notably in compilation time and memory usage. An increase in optimization leads to an increase in both these metrics. Utilizing higher optimization levels may not guarantee better performance and may even give incorrect or unreliable results according to sources found online. Nevertheless, we shall apply all four optimization levels and analyze the execution times.

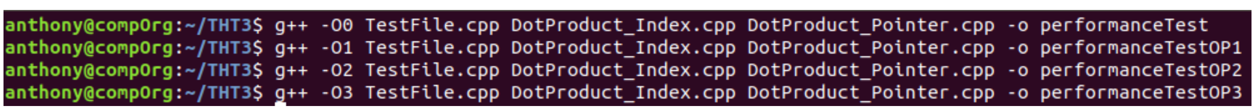


Figure 14a: Compiling Dot Product Programs Utilizing Optimization Flags

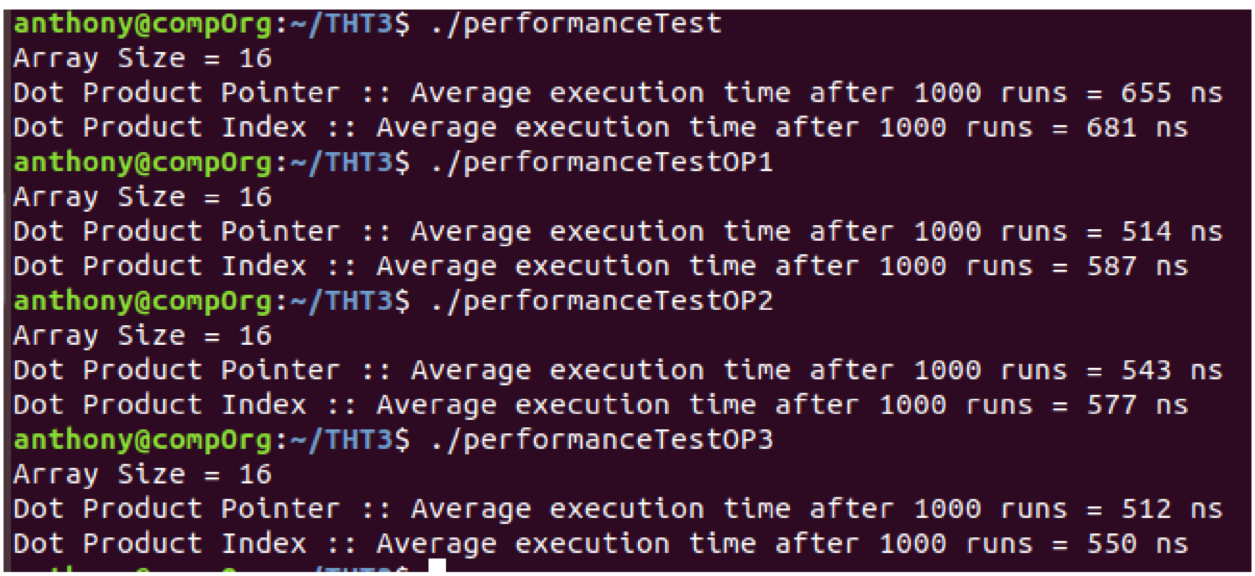
Test File

Figure 14b shows the test file utilized to measure the execution times of each dot product program. Specifically, for a given array size, 1000 measurements will be taken and the average execution time will be outputted.

```
#include <iostream>
#include <time.h>
#include <stdint.h>
#include <stdlib.h>

extern int DotProduct_Index(int a1[], int a2[], int n);
extern int DotProduct_Pointer(int *a1, int *a2, int n);

#define NANO 1000000000L
using namespace std;

const int runs = 1000;
const int arraySize = 256;
static int arr1[arraySize];
static int arr2[arraySize];

int main() {
        uint64_t elapsed_time;
        uint64_t PointerSum = 0;
        uint64_t IndexSum = 0;
        int result;
        struct timespec timeStart, timeEnd;

        for (int i = 0; i < arraySize; i++) {
                arr1[i] = i;
                arr2[i] = i/2;
        }

        cout << "Array Size = " << arraySize << endl;

        for (int i = 0; i < runs; i++) {
                        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timeStart);
// Begin Timer
                        result = DotProduct_Pointer(&arr1[0], &arr2[0],
arraySize);
                        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timeEnd); //
End Timer

                        elapsed_time = NANO * (timeEnd.tv_sec -
timeStart.tv_sec) + (timeEnd.tv_nsec - timeStart.tv_nsec);
                        //cout << "Execution Time = " << elapsed_time << "ns"
<< endl;
                        PointerSum += elapsed_time;
                        //cout << PointerSum << endl;

        }
```

```
cout << "Dot Product Pointer :: Average execution time after " << runs <<  "
runs = " << PointerSum/runs << " ns" << endl;

for (int j = 0; j < runs; j++) {
                clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timeStart); // Begin
Timer
                result = DotProduct_Index(arr1, arr2, arraySize);
                clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timeEnd); // End
Timer

                elapsed_time = NANO * (timeEnd.tv_sec - timeStart.tv_sec) +
(timeEnd.tv_nsec - timeStart.tv_nsec);
                //cout << "Execution Time = " << elapsed_time << "ns" <<
endl;
                IndexSum += elapsed_time;
                //cout << IndexSum << endl;

        }

        cout << "Dot Product Index :: Average execution time after " << runs
<< " runs = " << IndexSum/runs << " ns" << endl;
        return 0;

}
```

Figure 14b: Test Program to Measure Average Execution Times



```
anthony@compOrg:~/THT3$ ./performanceTest
Array Size = 16
Dot Product Pointer :: Average execution time after 1000 runs = 655 ns
Dot Product Index :: Average execution time after 1000 runs = 681 ns
anthony@compOrg:~/THT3$ ./performanceTestOP1
Array Size = 16
Dot Product Pointer :: Average execution time after 1000 runs = 514 ns
Dot Product Index :: Average execution time after 1000 runs = 587 ns
anthony@compOrg:~/THT3$ ./performanceTestOP2
Array Size = 16
Dot Product Pointer :: Average execution time after 1000 runs = 543 ns
Dot Product Index :: Average execution time after 1000 runs = 577 ns
anthony@compOrg:~/THT3$ ./performanceTestOP3
Array Size = 16
Dot Product Pointer :: Average execution time after 1000 runs = 512 ns
Dot Product Index :: Average execution time after 1000 runs = 550 ns
```

Figure 14c: Average Execution Time Measurements for Array Size = 16

Comparing Assembly Source Files

To better explain the decrease in execution times as optimization levels increase, we must look into the compiler generated assembler source files. Figure 14d shows the compiler assembler source both for when no optimization and level 1 optimization are applied for the dot product program utilizing array indexing. The first observation here is that the assembly generated via level 1 optimization is much smaller than the assembly with no optimization. This is done by reducing the assembly required to perform an instruction (which may be more difficult to debug).

```
	.file	"DotProduct_Index.cpp"
	.text
	.globl	_Z16DotProduct_IndexPiS_i
	.type	_Z16DotProduct_IndexPiS_i, @function
_Z16DotProduct_IndexPiS_i:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -24(%rbp)
	movq	%rsi, -32(%rbp)
	movl	%edx, -36(%rbp)
	movl	$0, -8(%rbp)
	movl	$0, -4(%rbp)
.L3:
	movl	-4(%rbp), %eax
	cmpl	-36(%rbp), %eax
	jge	.L2
	movl	-4(%rbp), %eax
	cltq
	leaq	0(,%rax,4), %rdx
	movq	-24(%rbp), %rax
	addq	%rdx, %rax
	movl	(%rax), %edx
	movl	-4(%rbp), %eax
	cltq
	leaq	0(,%rax,4), %rcx
	movq	-32(%rbp), %rax
	addq	%rcx, %rax
	movl	(%rax), %eax
	imull	%edx, %eax
	addl	%eax, -8(%rbp)
	addl	$1, -4(%rbp)
	jmp	.L3
.L2:
	movl	-8(%rbp), %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	_Z16DotProduct_IndexPiS_i, .-_Z16DotProduct_IndexPiS_i
	.ident	"GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
```

-O0
(No Optimization)

```
	.file	"DotProduct_Index.cpp"
	.text
	.globl	_Z16DotProduct_IndexPiS_i
	.type	_Z16DotProduct_IndexPiS_i, @function
_Z16DotProduct_IndexPiS_i:
.LFB0:
	.cfi_startproc
	testl	%edx, %edx
	jle	.L4
	leal	-1(%rdx), %eax
	leaq	4(,%rax,4), %r8
	movl	$0, %edx
	movl	$0, %eax
.L3:
	movl	(%rdi,%rdx), %ecx
	imull	(%rsi,%rdx), %ecx
	addl	%ecx, %eax
	addq	$4, %rdx
	cmpq	%r8, %rdx
	jne	.L3
	rep ret
.L4:
	movl	$0, %eax
	ret
	.cfi_endproc
.LFE0:
	.size	_Z16DotProduct_IndexPiS_i, .-_Z16DotProduct_IndexPiS_i
	.ident	"GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
	.section	.note.GNU-stack,"",@progbits
```

-O1
(Optimization Level 1)

Figure 14d: Comparing Assembly Source

In a similar sense, figure 14e shows the compiler generated assembly code for when optimization level 2 is applied. There are very minor differences between this code and the assembly source generated in the previous figure with level one optimizations. Most noticeably is use of `xorl %eax, %eax` instruction instead of `movl $0, %eax`. The use of `xorl` provides some optimization by requiring less space to encode as a machine instruction as it does not require the use of an immediate operand whereas the instruction via `movl` does and thus requires more space (4 Bytes).

```
	.file	"DotProduct_Index.cpp"
	.text
	.p2align 4,,15
	.globl	_Z16DotProduct_IndexPiS_i
	.type	_Z16DotProduct_IndexPiS_i, @function
_Z16DotProduct_IndexPiS_i:
.LFB0:
	.cfi_startproc
	testl	%edx, %edx
	jle	.L4
	leal	-1(%rdx), %eax
	xorl	%edx, %edx
	leaq	4(,%rax,4), %r8
	xorl	%eax, %eax
	.p2align 4,,10
	.p2align 3
.L3:
	movl	(%rdi,%rdx), %ecx
	imull	(%rsi,%rdx), %ecx
	addq	$4, %rdx
	addl	%ecx, %eax
	cmpq	%rdx, %r8
	jne	.L3
	rep ret
	.p2align 4,,10
	.p2align 3
.L4:
	xorl	%eax, %eax
	ret
	.cfi_endproc
.LFE0:
	.size	_Z16DotProduct_IndexPiS_i, .-_Z16DotProduct_IndexPiS_i
	.ident	"GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
	.section	.note.GNU-stack,"",@progbits
```

Figure 14e: Complier Generated Assembly Source at optimization level 2

Lastly, the application of level 3 optimization attempts to vectorize loops in an attempt to reduce execution time. However, not only does the assembly become more difficult to debug, but this application also yields the longest compilation time which may be an issue for larger programs. In addition, several sources warn against this optimization because vectorizing loops typically produces larger executables which might actually perform worse than lower optimization levels. Figure 14f shows the assembly source for level 3 optimization. Note the utilization of SSE instruction set.

```
        .file   "DotProduct_Index.cpp"
        .text
        .p2align 4,,15
        .globl  _Z16DotProduct_IndexPiS_i
        .type   _Z16DotProduct_IndexPiS_i, @function
_Z16DotProduct_IndexPiS_i:
.LFB0:
        .cfi_startproc
        testl   %edx, %edx
        jle     .L9
        movq    %rdi, %r9
        leal    -1(%rdx), %ecx
        movl    $5, %r8d
        shrq    $2, %r9
        pushq   %r12
        .cfi_def_cfa_offset 16
        .cfi_offset 12, -16
        pushq   %rbp
        .cfi_def_cfa_offset 24
        .cfi_offset 6, -24
        negq    %r9
        pushq   %rbx
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        andl    $3, %r9d
        leal    3(%r9), %eax
        cmpl    $5, %eax
        cmovb   %r8d, %eax
        cmpl    %eax, %ecx
        jb      .L10
        testl   %r9d, %r9d
        je      .L11
        movl    (%rdi), %eax
        movl    $1, %ebp
        imull   (%rsi), %eax
        cmpl    $1, %r9d
        movl    %eax, %r12d
        je      .L4
        movl    4(%rdi), %eax
        movl    $2, %ebp
        imull   4(%rsi), %eax
        addl    %eax, %r12d
        cmpl    $2, %r9d
        je      .L4
        movl    8(%rsi), %eax
        movl    $3, %ebp
        imull   8(%rdi), %eax
        addl    %eax, %r12d
.L4:
        movl    %edx, %ebx
        pxor    %xmm3, %xmm3
        subl    %r9d, %ebx
        salq    $2, %r9
        xorl    %ecx, %ecx
        movl    %ebx, %r11d
        leaq    (%rdi,%r9), %r10
        xorl    %r8d, %r8d
        shrl    $2, %r11d
        addq    %rsi, %r9
        .p2align 4,,10
        .p2align 3
.L7:
        movdqu  (%r9,%rcx), %xmm0
        addl    $1, %r8d
        movdqa  %xmm0, %xmm1
        psrlq   $32, %xmm0
        movdqa  (%r10,%rcx), %xmm2
        pmuludq (%r10,%rcx), %xmm1
        pshufd  $8, %xmm1, %xmm1
        addq    $16, %rcx
        psrlq   $32, %xmm2
        pmuludq %xmm2, %xmm0
        pshufd  $8, %xmm0, %xmm0
        cmpl    %r11d, %r8d
        punpckldq       %xmm0, %xmm1
        paddd   %xmm1, %xmm3
        jb      .L7
        movdqa  %xmm3, %xmm0
        movl    %ebx, %r8d
        andl    $-4, %r8d
        psrldq  $8, %xmm0
        paddd   %xmm0, %xmm3
        movdqa  %xmm3, %xmm0
        leal    (%r8,%rbp), %ecx
        psrldq  $4, %xmm0
        paddd   %xmm0, %xmm3
        movd    %xmm3, %eax
        addl    %r12d, %eax
        cmpl    %r8d, %ebx
        je      .L1
.L3:
        movslq  %ecx, %r9
        movl    (%rdi,%r9,4), %r8d
        imull   (%rsi,%r9,4), %r8d
        addl    %r8d, %eax
        leal    1(%rcx), %r8d
        cmpl    %r8d, %edx
        jle     .L1
        movslq  %r8d, %r8
        movl    (%rdi,%r8,4), %r9d
        imull   (%rsi,%r8,4), %r9d
        leal    2(%rcx), %r8d
        addl    %r9d, %eax
        cmpl    %r8d, %edx
        jle     .L1
        movslq  %r8d, %r8
        movl    (%rdi,%r8,4), %r9d
        imull   (%rsi,%r8,4), %r9d
        leal    3(%rcx), %r8d
        addl    %r9d, %eax
        cmpl    %r8d, %edx
        jle     .L1
        movslq  %r8d, %r8
        movl    (%rdi,%r8,4), %r9d
        imull   (%rsi,%r8,4), %r9d
        leal    4(%rcx), %r8d
        addl    %r9d, %eax
        cmpl    %r8d, %edx
        jle     .L1
        movslq  %r8d, %r8
        addl    $5, %ecx
        movl    (%rdi,%r8,4), %r9d
        imull   (%rsi,%r8,4), %r9d
        addl    %r9d, %eax
        cmpl    %ecx, %edx
        jle     .L1
        movslq  %ecx, %rcx
        movl    (%rdi,%rcx,4), %edx
        imull   (%rsi,%rcx,4), %edx
        addl    %edx, %eax
.L1:
        popq    %rbx
        .cfi_remember_state
        .cfi_def_cfa_offset 24
        popq    %rbp
        .cfi_def_cfa_offset 16
        popq    %r12
        .cfi_def_cfa_offset 8
        ret
        .p2align 4,,10
        .p2align 3
.L11:
        .cfi_restore_state
        xorl    %ebp, %ebp
        xorl    %r12d, %r12d
        jmp     .L4
        .p2align 4,,10
        .p2align 3
.L9:
        .cfi_def_cfa_offset 8
        .cfi_restore 3
        .cfi_restore 6
        .cfi_restore 12
        xorl    %eax, %eax
        ret
        .p2align 4,,10
        .p2align 3
.L10:
        .cfi_def_cfa_offset 32
        .cfi_offset 3, -32
        .cfi_offset 6, -24
        .cfi_offset 12, -16
        xorl    %ecx, %ecx
        xorl    %eax, %eax
        jmp     .L3
        .cfi_endproc
.LFE0:
        .size   _Z16DotProduct_IndexPiS_i, .-_Z16DotProduct_IndexPiS_i
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
                                                        170,1-8
```

Figure 14f: Complier Generated Assembly Source at optimization level 3

Results

Similar observations occurred when examining the program utilizing pointer arithmetic. Figures 51a and 15b show the plotted execution times utilizing various optimization levels for various array sizes.

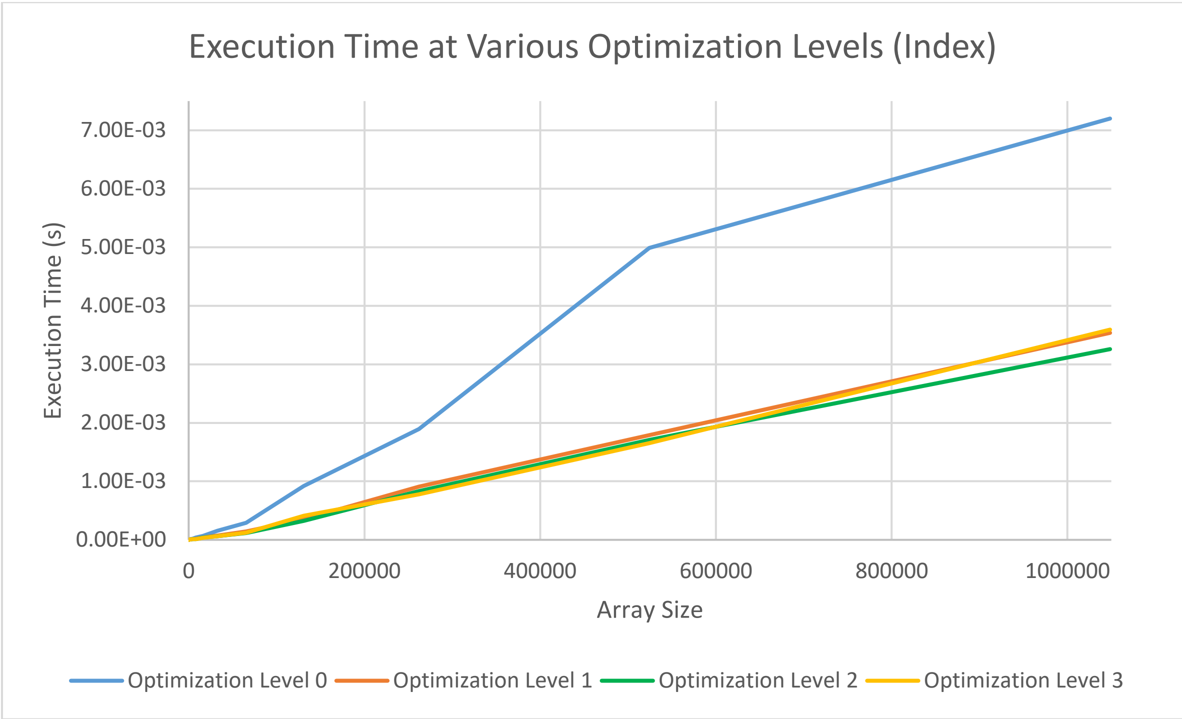


Figure 15a: Execution Times at Various Optimization Levels (Index)

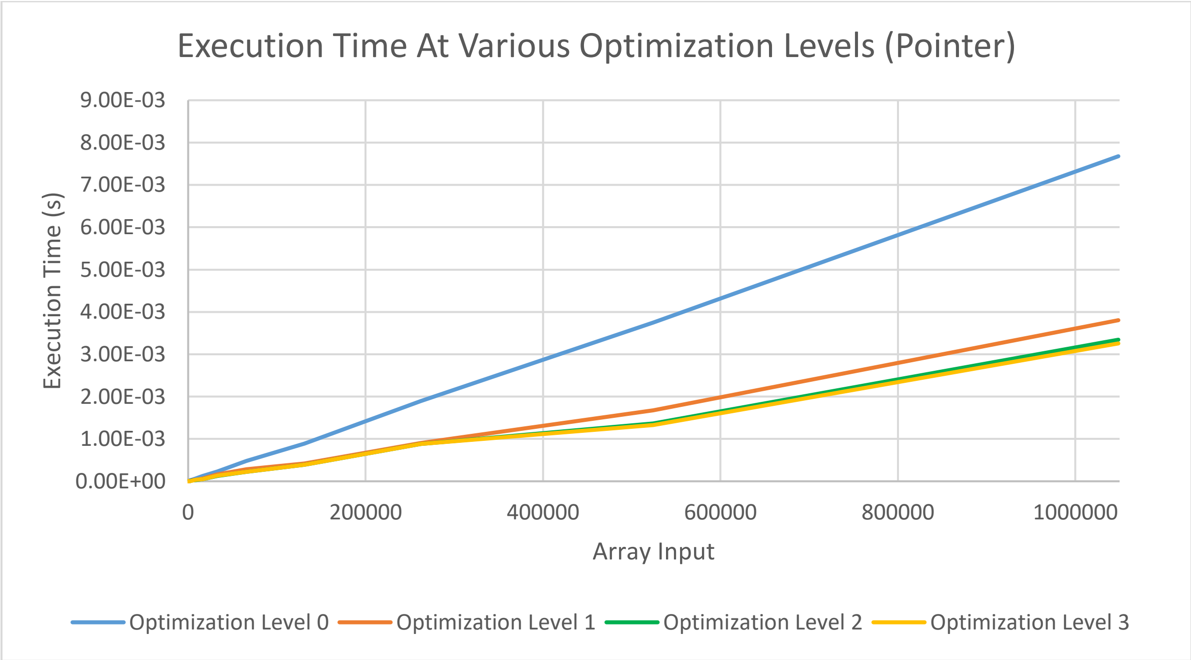


Figure 15b: Execution Times at Various Optimization Levels (Pointer)

Given these results, it is obvious that applying one level of optimization significantly improves the execution time of both programs. It can also be concluded that optimization at level 2 provides even faster execution times than level 1 optimization. However, the results obtained from optimization level 3, failed to prove any improvement over level 2 optimization in regard to execution time. In fact, there were some instances where this optimization level performed worse than level 2 optimization (but never worse than level 1). In addition to no improvement in execution time, we pay the price with higher compilation time which was noticeable for larger array sizes. This behavior confirms what online sources previously stated. Hence it could be concluded that level 2 optimization is the much better option in terms of reducing execution time (if a slight increase in compilation time is no issue). Regarding Level 1 optimization, it is a good choice if maximum optimization is not required.