Test 3: Performance between Indexing and Pointer Arithmetic In clearing Array and Performance of Dot product between Indexing, Pointers and Vector Instructions.

By Demetrios Doumas
Fall 2016
CSC 343
Prof. Gertner

Table of Contents

Objective: Compare performance of clearing an array and dot product by indexing arithmetic and by clearing the array using pointer arithmetic. Also, compare the performance of a dot product scalar code computation with the vector instruction code.

Indexing VS Pointers in Visual Studio 2013 Platform:

Using Indexing Arithmetic:

Main.cpp file:

```
#include <windows.h>
#include<iostream>
using namespace std;
//using namespace System;

//void main(int[], int);
void ClearUsingIndex(int Array[], int size);

static int Array[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, -1 };

int main(){
    int size = 10;
    ClearUsingIndex(Array, size);

return 0;
}
```

IndexArrayClear.cpp file:

```
void ClearUsingIndex(int Array[], int size) {
    int i;
    for (i = 0; i < size; i += 1)
        Array[i] = 0;
}
```

Compiler generated Assembly Code listing of Clear Array using Index function:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1

	TITLE	c:\Users\Demetri\documents\visual studio 2013\Projects\IndexArrayClear\IndexArrayClear\indexclear.cpp
	.686P
	.XMM
	include listing.inc
	.model	flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC	?ClearUsingIndex@@YAXQAHH@Z			; ClearUsingIndex
EXTRN	__RTC_InitBase:PROC
EXTRN	__RTC_Shutdown:PROC
;	COMDAT rtc$TMZ
rtc$TMZ	SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ	ENDS
;	COMDAT rtc$IMZ
rtc$IMZ	SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ	ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;	COMDAT ?ClearUsingIndex@@YAXQAHH@Z
_TEXT	SEGMENT
_i$ = -8						; size = 4
_Array$ = 8						; size = 4
_size$ = 12						; size = 4
?ClearUsingIndex@@YAXQAHH@Z PROC			; ClearUsingIndex, COMDAT
; File c:\users\demetri\documents\visual studio 2013\projects\indexarrayclear\indexarrayclear\indexclear.cpp
; Line 1
	push	ebp
	mov	ebp, esp
	sub	esp, 204				; 000000ccH
	push	ebx
	push	esi
	push	edi
	lea	edi, DWORD PTR [ebp-204]
	mov	ecx, 51					; 00000033H
	mov	eax, -858993460				; ccccccccH
```

continue…

```
	mov	eax, -858993460				; ccccccccH
	rep stosd
; Line 3
	mov	DWORD PTR _i$[ebp], 0
	jmp	SHORT $LN3@ClearUsing
$LN2@ClearUsing:
	mov	eax, DWORD PTR _i$[ebp]
	add	eax, 1
	mov	DWORD PTR _i$[ebp], eax
$LN3@ClearUsing:
	mov	eax, DWORD PTR _i$[ebp]
	cmp	eax, DWORD PTR _size$[ebp]
	jge	SHORT $LN4@ClearUsing
; Line 4
	mov	eax, DWORD PTR _i$[ebp]
	mov	ecx, DWORD PTR _Array$[ebp]
	mov	DWORD PTR [ecx+eax*4], 0
	jmp	SHORT $LN2@ClearUsing
$LN4@ClearUsing:
; Line 5
	pop	edi
	pop	esi
	pop	ebx
	mov	esp, ebp
	pop	ebp
	ret	0
?ClearUsingIndex@@YAXQAHH@Z ENDP			; ClearUsingIndex
_TEXT	ENDS
END
```

Functionality of Compiler generated Assembly Code listing of Clear Array using Index in the last page:

In line 26 to 28, the compiler stores the offset of the index of the loop, memory address of Array, and the size array at those particular offset from the base pointer of the stack. In line 32 to 41 the stack is being implemented. The index value in the for-loop gets initialized to one in line 43. The condition of the for-loop is translated in lines 50 to 52. The value at _i is the offset from the base pointer that store the index value I in the for-loop is store to register eax. Then it gets compared with the size of the array at an offset of _size from the base pointer. In line 4, the value of I in the for loop gets stored to eax and the beginning memory address is stored to the ecx register by the use of the offset _Array from the base pointer. The memory address in ecx get incremented by I times by 4 bytes, and initialize it to zero. The pattern then repeats until in line 3 where the compare instruction compares the index value at eax to the size of the array and if it is greater than then the instruction pointer then starts to deallocated the stack frame and bring the function to the end.

Optimized Assembly Code of Clear Array using Index function:

```
        .686P
        .XMM
        .model  flat

_TEXT   SEGMENT
_i$ = -8                            ; size = 4
_Array$ = 8                         ; size = 4
_size$ = 12                         ; size = 4
?ClearUsingIndex@@YAXQAHH@Z PROC                ; ClearUsingIndex, COMDAT

; Line 1
        push    ebp
        mov ebp, esp
        sub esp, 204                ; 000000ccH
        push    ebx
        push    esi
        push    edi
        lea edi, DWORD PTR [ebp-204]
        mov ecx, 51                 ; 00000033H
        mov eax, -858993460             ; ccccccccH
        rep stosd

        ;9 instructions
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        mov DWORD PTR _i$[ebp], 0
        mov eax, DWORD PTR _i$[ebp]
        $LN2@ClearUsing:
        cmp eax, DWORD PTR _size$[ebp]
        jge short exit
        mov eax, DWORD PTR _i$[ebp]
        mov ecx, DWORD PTR _Array$[ebp]
        mov DWORD PTR[ecx + eax*4], 0
        add DWORD PTR _i$[ebp],1
        jmp SHORT $LN2@ClearUsing
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        exit:
```

Continued…

```
36.
37.     exit:
38.
39. ;Line 5
40.     pop edi
41.     pop esi
42.     pop ebx
43.     mov esp, ebp
44.     pop ebp
45.     ret 0
46. ?ClearUsingIndex@@YAXQAHH@Z ENDP            ; ClearUsingIndex
47. _TEXT   ENDS
48. END
```

The code above is the optimized version from the compiled generated code. The optimized code has 9 lines of code for the body, not including the stack code, and the compilers version takes 12 lines. The first part of the code is to initialize the index i to zero. Move index (i) to the eax register. Then compare eax with the size of the array length. If eax is greater than the size of the array then jump to the remove stack code towards the end of the function. In the body of the loop we are instructing the compiler to generate array A at a certain index then initialize it to zero (A[i]=0). This happens by getting access to the current index and the address first (lines 30 and 32). Line 33 increments I by one. Then the procedure jumps back to loop and the body of the assemble code is repeated until eax is above the size of the Array being passed to the function.

Using Pointer Arithmetic:

Main.cpp

```
#include<iostream>
using namespace std;


void ClearUsingPointers(int *array, int size);

static int Array[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, -1 };

int main()
{
    int size = 10;

    for (int i = 0; i < 10; i++)
    {
        cout << Array[i] << endl;
    }

    cout << endl;
    cout << endl;
    ClearUsingPointers(Array, size);


    for (int i = 0; i < 10; i++)
    {
        cout << Array[i] << endl;
    }


    return 0;
}
```

PointerArrayClear.cpp file:

```
void ClearUsingPointers(int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

Compiler generated Assembly Code listing of Clear Array using Pointer function:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1

    TITLE   c:\Users\Demetri\documents\visual studio 2013\Projects\PointerArrayClear\PointerArrayClear\PointerArrayClear.cpp
    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC  ?ClearUsingPointers@@YAXPAHH@Z          ; ClearUsingPointers
EXTRN   __RTC_InitBase:PROC
EXTRN   __RTC_Shutdown:PROC
;   COMDAT rtc$TMZ
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;   COMDAT ?ClearUsingPointers@@YAXPAHH@Z
_TEXT   SEGMENT
_p$ = -8                        ; size = 4
_array$ = 8                     ; size = 4
_size$ = 12                     ; size = 4
?ClearUsingPointers@@YAXPAHH@Z PROC         ; ClearUsingPointers, COMDAT
; File c:\users\demetri\documents\visual studio 2013\projects\pointerarrayclear\pointerarrayclear\pointerarrayclear.cpp
; Line 1
    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
```

Continued…

```
    rep stosd
; Line 3
    mov eax, 4
    imul    ecx, eax, 0
    add ecx, DWORD PTR _array$[ebp]
    mov DWORD PTR _p$[ebp], ecx
    jmp SHORT $LN3@ClearUsing
$LN2@ClearUsing:
    mov eax, DWORD PTR _p$[ebp]
    add eax, 4
    mov DWORD PTR _p$[ebp], eax
$LN3@ClearUsing:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _array$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _p$[ebp], edx
    jae SHORT $LN4@ClearUsing
; Line 4
    mov eax, DWORD PTR _p$[ebp]
    mov DWORD PTR [eax], 0
    jmp SHORT $LN2@ClearUsing
$LN4@ClearUsing:
; Line 5
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?ClearUsingPointers@@YAXPAHH@Z ENDP        ; ClearUsingPointers
_TEXT   ENDS
END
```

In the last page is the visual studio compiler generated assembly code. The first line is the creation of the stack. The third line is where the body of the function is executed. There were a few step involved at line 3. The value 4 gets copied to the register eax. The register ecx is initialized with a value of zero. The next two lines, the register get the first address of the array and store it to the compiler variable _p$. A jump is executed to $LN3@ClearUsing. The array size is stored to register eax. The memory address of array size is stored in ecx register. The last address is computed and loaded to register edx. The address of _p$ is compared with the last address of the array. If the current address is not the same then the next line after the jump instruction is executed. In line 4, the value in _p$ is stored at eax. Then zero is moved to eax. The next instruction jumps to $LN2@ClearUsing. This is where the address is move to the next index by moving the value of _p$ to eax and then add 4. Then the address in eax gets saved back to _$p. This pattern repeats until eax is equal to edx or the last address of the array. If they are equal then the jump to remove the stack frame is called and the function ends.

Optimized generated Assembly compiler Code listing of Clear Array using pointer arithmetic function:

```
; Demetrios Doumas Optimized code  11/6/15
    .686P
    .XMM
    include listing.inc
    .model  flat

_TEXT   SEGMENT

;offset from base pointer
_p$ = -8                        ; size = 4
_array$ = 8                     ; size = 4
_size$ = 12                     ; size = 4

?ClearUsingPointers@@YAXPAHH@Z PROC         ; ClearUsingPointers, COMDAT

; Line 1
    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
    rep stosd
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; save the begining address of A to _p$  or point _p to the address the first address of array
    mov ecx, 0 ;
    add ecx, DWORD PTR _array$[ebp]
    mov DWORD PTR _p$[ebp], ecx

    ; need to save the last address to edx register
    mov eax, DWORD PTR _size$[ebp]   ; load the size to eax
    mov ecx, DWORD PTR _array$[ebp]  ; load the memory add of A into ecx
    lea edx, DWORD PTR [ecx+eax*4]

    ; Condition
    $loop:
    cmp DWORD PTR _p$[ebp], edx
    jae SHORT $exit
```

Continued…

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ; save the begining address of A to _p$  or point _p to the address the first address of array
    mov ecx, 0 ;
    add ecx, DWORD PTR _array$[ebp]
    mov DWORD PTR _p$[ebp], ecx

    ; need to save the last address to edx register
    mov eax, DWORD PTR _size$[ebp]   ; load the size to eax
    mov ecx, DWORD PTR _array$[ebp]  ; load the memory add of A into ecx
    lea edx, DWORD PTR [ecx+eax*4]

    ; Condition
    $loop:
    cmp DWORD PTR _p$[ebp], edx
    jae SHORT $exit

    ; Body of the Loop
    mov eax, DWORD PTR _p$[ebp]   ; give the current address of _p put it into eax
    mov DWORD PTR [eax], 0    ; initilaize it to zero
    add eax, 4                  ; Increment to the next value
    mov DWORD PTR _p$[ebp], eax     ; Move the new address to _p
    jmp SHORT $loop

    $exit:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Line 5
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?ClearUsingPointers@@YAXPAHH@Z ENDP          ; ClearUsingPointers
_TEXT   ENDS
END
```

The code in the last page and above is the optimized compiled generated assembly code. There are several of differences between the optimized compiler assembly code and the original compiler assembly listing code. In the complier assembly code there are 16 instruction and several of jumps or loops. The last address of the array is computed within the loop, where as the optimized code has the calculation of the last address outside the single loop. There are six instructions outside of the loop, where as in the complied generated code all the instruction has loops in between each other.

I followed the algorithm of the c++ code function and turn it to assemble code by editing the compile generated code. The first part is to declare the pointer. Make the pointer point to the first memory of address array. Next is to get the last address of the array. Then compare the current address with the last address of the array. If they are equal then jump to exit label and the stack begins to deallocate. If it does not exit, then increase the pointer and set the value of array at current index to zero. This program continues the same pattern until the current pointer points to the last address.

Performance Measurements between the Use of Indexing and Pointer Arithmetic in Visual Studio:

Time Measurement code for measuring performance of the clearing array with Index

```
#include "stdafx.h";
#include <windows.h>;
using namespace System;
using namespace std;
#include <fstream>

void ClearUsingIndex(int Array[], int size);

static int A[100000000];

int N[7] = {10, 100, 1000, 10000, 100000, 1000000, 10000000};

int main()
{

    int size = 10;
    _int64 ctrl = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;

    double Avg = 0.0;
    ofstream myfile;
    myfile.open("IndexClearArraydata.txt");

    for (int i = 0; i < 10; i++)
    {
        Console::WriteLine(A[i]);
    }

```

continued…

```

    double k = 0;
    for (int u = 0; u < 7; u++){
        for (int i = 0; i < 5; i++)
        {

            if (QueryPerformanceCounter((LARGE_INTEGER*)&ctrl) != 0){
                ClearUsingIndex(A, N[u]);
                QueryPerformanceCounter((LARGE_INTEGER*)&ctr2);
                QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

                k = ("{0}", (((ctr2 - ctrl)* 1.0) / freq)) + k;
                //Console::WriteLine("{0}", (((ctr2 - ctrl)* 1.0) / freq).ToString());
            }

        }
        Avg = k / 5;
        myfile << Avg << "\t" << N[u] << "\n";

    }

    Console::WriteLine(Avg);

    for (int i = 0; i < 10; i++)
    {
        Console::WriteLine(A[i]);// << endl;
    }
    return 0;

    myfile.close();
    Console::WriteLine();

    Console::Read();
}
```

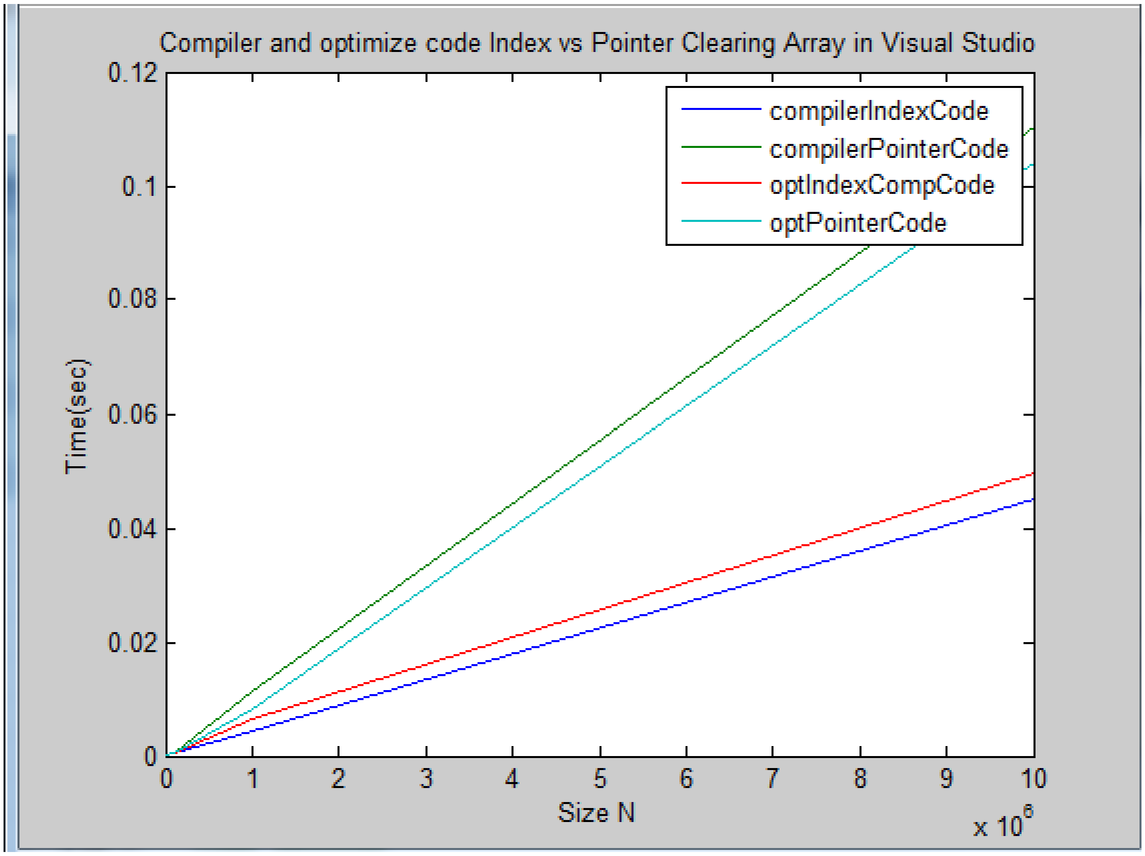
Time Measurement code for measuring performance of the clearing array with pointers:

```
#include "stdafx.h";
#include <windows.h>;
using namespace System;
using namespace std;
#include <fstream>

void ClearUsingPointers(int *array, int size);

static int A[100000000];

int N[7] = { 10, 100, 1000, 10000, 100000, 1000000, 10000000 };

int main()
{

    int size = 10;
    _int64 ctrl = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;

    double Avg = 0.0;
    ofstream myfile;
    myfile.open("PointerClearArraydata.txt");

    for (int i = 0; i < 10; i++)
    {
        Console::WriteLine(A[i]);
    }

```

Continued…

```
    double k = 0;
    for (int u = 0; u < 7; u++){
        for (int i = 0; i < 5; i++)
        {

            if (QueryPerformanceCounter((LARGE_INTEGER*)&ctrl) != 0){
                ClearUsingPointers(A, N[u]);
                QueryPerformanceCounter((LARGE_INTEGER*)&ctr2);
                QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

                k = ("{0}", (((ctr2 - ctrl)* 1.0) / freq)) + k;
                //Console::WriteLine("{0}", (((ctr2 - ctrl)* 1.0) / freq).ToString());
            }

        }
        Avg = k / 5;
        myfile << Avg << "\t" << N[u] << "\n";

    }

    Console::WriteLine(Avg);

    for (int i = 0; i < 10; i++)
    {
        Console::WriteLine(A[i]);// << endl;
    }
    return 0;

    myfile.close();
    Console::WriteLine();

    Console::Read();




}
```

Total performance:



The results were unexpected. Both the compiler and optimized code using pointers should of have the fastest time to clear the array. However, as shown above the compiler generated code to clear the array using indexing was the fastest. The optimize pointer code to clear the array runs faster than the compiler generated code using pointers. The optimize code for clearing the array with indexing arithmetic ran slower than the complier generated code using indexing to clear the array.

The data is display in the table below:

Size N	Opt Time (clear by Pointer) sec	Opt Time (clear by Index) sec	Comp CodeTime (clear by Pointer) sec	Comp CodeTimeTime (clear by Index) sec
10	4.09E-05	3.39E-05	2.91E-05	3.56E-05
100	4.28E-05	3.56E-05	3.09E-05	3.76E-05
1000	4.90E-05	4.36E-05	3.79E-05	4.40E-05
10000	9.16E-05	8.67E-05	8.76E-05	8.23E-05
100000	0.000516884	0.000507771	0.000560606	0.000503266
1000000	0.0083963	0.00635917	0.0114575	0.00450646
10000000	0.104164	0.0496016	0.110457	0.044986

Indexing VS Pointers in Linux Platform:

```
#include<iostream>
using namespace std;

void ClearUsingIndex(int Array[], int size);

static int Array[10] ={1,2,3,4,5,6,7,8,9,-1};

int main() {
int size = 10;


for(int i=0;i<size;i++)
{
cout<<Array[i]<<endl;
}

ClearUsingIndex( Array, size);


for(int i=0;i<size;i++)
{
cout<<Array[i]<<endl;
}

return 0;
}
```

```
void ClearUsingIndex(int Array[], int size) {
int i;
for (i = 0; i < size; i +=1)
Array[i] = 0;
}
```

```
void ClearUsingPointers ( int *array, int size) {
int *p;
for (p = &array[0]; p < &array[size]; p = p + 1)
*p = 0;
}
```

Linux Compiler Generated Assembly code below for index:

```
01.     .file   "indexcleararray.cpp"
02.     .text
03.     .globl  _Z15ClearUsingIndexPii
04.     .type   _Z15ClearUsingIndexPii, @function
05. _Z15ClearUsingIndexPii:
06. .LFB0:
07.     .cfi_startproc
08.     pushq   %rbp
09.     .cfi_def_cfa_offset 16
10.     .cfi_offset 6, -16
11.     movq    %rsp, %rbp
12.     .cfi_def_cfa_register 6
13.     movq    %rdi, -24(%rbp)
14.     movl    %esi, -28(%rbp)
15.     movl    $0, -4(%rbp)
16. .L3:
17.     movl    -4(%rbp), %eax
18.     cmpl    -28(%rbp), %eax
19.     jge .L4
20.     movl    -4(%rbp), %eax
21.     cltq
22.     leaq    0(,%rax,4), %rdx
23.     movq    -24(%rbp), %rax
24.     addq    %rdx, %rax
25.     movl    $0, (%rax)
26.     addl    $1, -4(%rbp)
27.     jmp .L3
28. .L4:
29.     nop
30.     popq    %rbp
31.     .cfi_def_cfa 7, 8
32.     ret
33.     .cfi_endproc
34. .LFE0:
35.     .size   _Z15ClearUsingIndexPii, .-_Z15ClearUsingIndexPii
36.     .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609"
37.     .section    .note.GNU-stack,"",@progbits
```

The code above is the Linux compiler Generated assembly code listing. The stack frame is created on lines 7 to 12. The registers rdi and esi are stored in the stack with a certain offset from the base pointer. Lines 17 to 19 are the condition of the for-loop. The size of the array is stored in the base pointer at an offset of -28. The value of I gets stored in the register eax. The next three lines increments the address. After that the value in the address gets initialize to zero. The index value I get incremented. The pattern of the execution continues until the condition for loop when I is greater than the size and jumps to the few lines of code that starts to deallocate memory from the stack.

Linux Optimized compiler Generated Assembly code below for Index:

```
01. .text
02.     .globl  _Z15ClearUsingIndexPii
03.     .type   _Z15ClearUsingIndexPii, @function
04. _Z15ClearUsingIndexPii:
05. .LFB0:
06.     .cfi_startproc
07.     pushq   %rbp
08.     .cfi_def_cfa_offset 16
09.     .cfi_offset 6, -16
10.     movq    %rsp, %rbp
11.     .cfi_def_cfa_register 6
12.
13.     movq    %rdi, -24(%rbp)
14.     movl    %esi, -28(%rbp)
15.     movl    $0, -4(%rbp)
16.
17. .Loop:
18.     movl    -4(%rbp), %eax
19.     cmpl    -28(%rbp), %eax
20.     jge .Exit
21.     leaq    0(,%rax,4), %rdx
22.     movq    -24(%rbp), %rax
23.     addq    %rdx, %rax
24.     movl    $0, (%rax)
25.     addl    $1, -4(%rbp)
26.     jmp .Loop
27.
28. .Exit:
29.     nop
30.     popq    %rbp
31.     .cfi_def_cfa 7, 8
32.     ret
33.     .cfi_endproc
34. .LFE0:
35.     .size   _Z15ClearUsingIndexPii, .-_Z15ClearUsingIndexPii
36.     .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609"
37.     .section    .note.GNU-stack,"",@progbits
```

The code above is the optimized version of the compiler generated assembly code for clearing the array using indexing arithmetic. Lines 6 to 12 is the creation of the stack frame. Lines 13 to 15 are different values stored on the stack at an offset from the base pointer. The index I I is stored at an offset of -4 from the base pointer. Lines 18 to 20 are the condition of the for-loop. The register eax store I and is being compared to the array size of an offset of -28 from the base pointer. The rest of the code is the same. The only difference is that there is no cltq instruction and line 20 was remove because it was unnecessary.

Linux Compiler Generated Assembly code below with Pointers to clear array:

```
01.         .file   "pointerclearrray.cpp"
02.         .text
03.         .globl  _Z18ClearUsingPointersPii
04.         .type   _Z18ClearUsingPointersPii, @function
05. _Z18ClearUsingPointersPii:
06. .LFB0:
07.         .cfi_startproc
08.         pushq   %rbp
09.         .cfi_def_cfa_offset 16
10.         .cfi_offset 6, -16
11.         movq    %rsp, %rbp
12.         .cfi_def_cfa_register 6
13.         movq    %rdi, -24(%rbp)
14.         movl    %esi, -28(%rbp)
15.         movq    -24(%rbp), %rax
16.         movq    %rax, -8(%rbp)
17. .L3:
18.         movl    -28(%rbp), %eax
19.         cltq
20.         leaq    0(,%rax,4), %rdx
21.         movq    -24(%rbp), %rax
22.         addq    %rdx, %rax
23.         cmpq    -8(%rbp), %rax
24.         jbe .L4
25.         movq    -8(%rbp), %rax
26.         movl    $0, (%rax)
27.         addq    $4, -8(%rbp)
28.         jmp .L3
29. .L4:
30.         nop
31.         popq    %rbp
32.         .cfi_def_cfa 7, 8
33.         ret
34.         .cfi_endproc
35. .LFE0:
36.         .size   _Z18ClearUsingPointersPii, .-_Z18ClearUsingPointersPii
37.         .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609"
38.         .section    .note.GNU-stack,"",@progbits
```

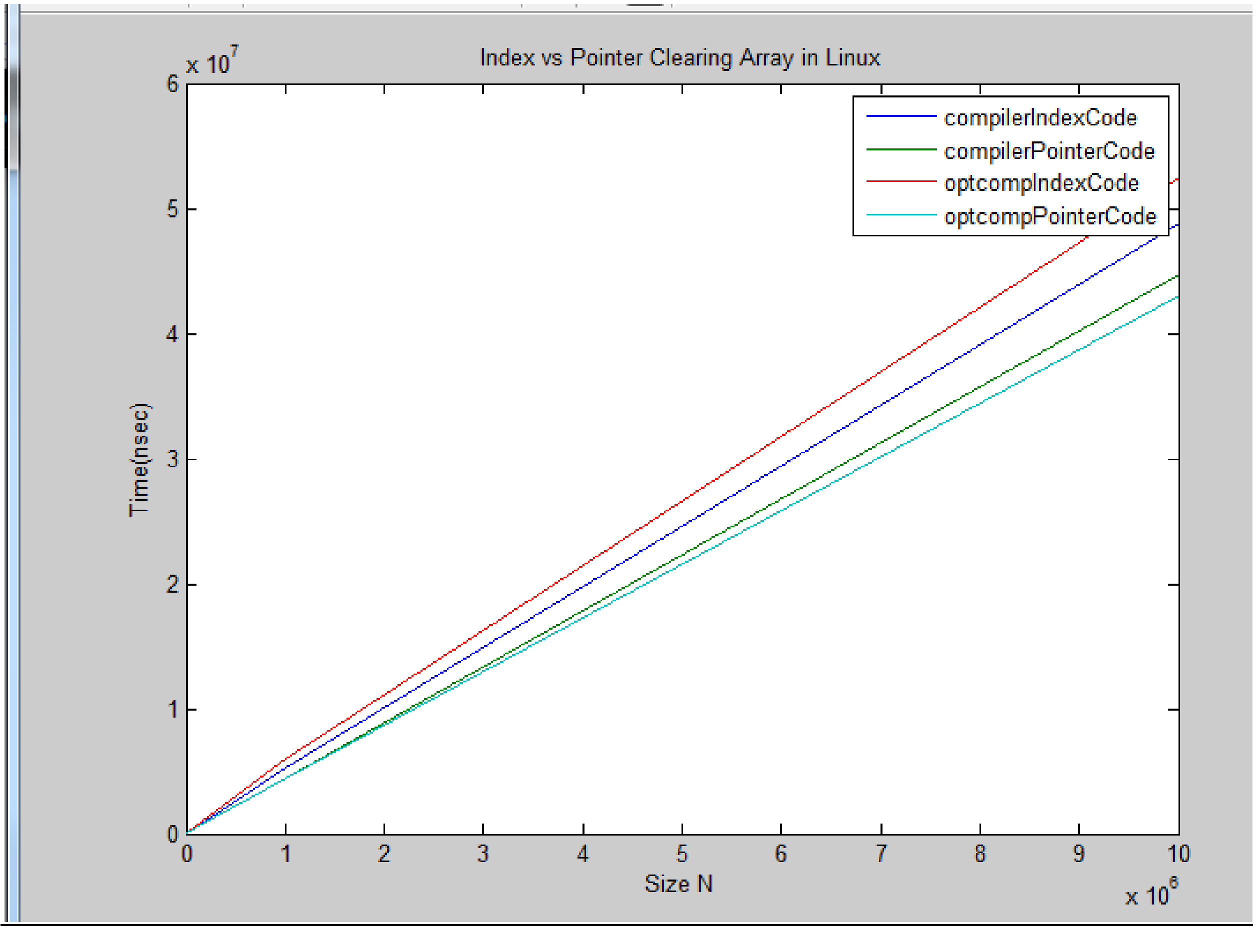
The code above is the compiler generated assembly code for clearing the array with pointer arithmetic. The stack frame is being created on lines 7 to 12. The values in line 13 to 16 are being stored on the stack by a particular offset from the base pointer. Lines 18 to 24 are the codes to increment the address of the pointer and compare the current address of the pointer to the last pointer of the array. Then the value zero gets copied to the current address which is stored in register rax. Then the execution of the code jumps back to incrementing the current address by 4 bytes. This pattern repeats until the register rax contains the last address of the array.

```
	.text
	.globl	_Z18ClearUsingPointersPii
	.type	_Z18ClearUsingPointersPii, @function
_Z18ClearUsingPointersPii:
.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)
	movl	%esi, -28(%rbp)
	movq	-24(%rbp), %rax
	movq	%rax, -8(%rbp)

.Loop:
	movl	-28(%rbp), %eax
	leaq	0(,%rax,4), %rdx
	movq	-24(%rbp), %rax
	addq	%rdx, %rax
	cmpq	-8(%rbp), %rax
	jbe .Exit
	movq	-8(%rbp), %rax
	movl	$0, (%rax)
	addq	$4, -8(%rbp)
	jmp .Loop

.Exit:
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	_Z18ClearUsingPointersPii, .-_Z18ClearUsingPointersPii
	.ident	"GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609"
	.section	.note.GNU-stack,"",@progbits
```

Linux Optimized Compiler Generated Assembly code below with Pointers to clear array:

The code above is the optimized compiler generated assembly code for clearing an array with pointers. The code did not change much. Only the cltq instruction was removed. The rest of the code has the same functionality as the previous compiler generated code.

Performance Measurements between the Use of Indexing and Pointer Arithmetic in Linux:



In the graph above shows the performance of the clearing the array functions using different techniques. The technique that perform the fastest is the clearing the array using pointers. The fastest performance was the optimized compiler generated assembly code using pointers. Clearing the array using indexing arithmetic took the longest to perform. The optimize code using indexing was slower than the compiler generated code using indexing arithmetic.

Dot Product Computation Using Scalar code:

Main.cpp

```cpp
#include<iostream>
#include<windows.h>;
using namespace std;

int DotProduct(int x[], int y[], int xsize, int ysize);

static int A[5] = { 1, 2, 3, 4, 5 };
static int B[5] = { 1, 2, 3, 4, 5 };
int main()
{
    int e = DotProduct(A, B, 5, 5);
    cout << e << endl;

    system("Pause");
    return 0;
}
```

Dot Product using Indexing Technique:

```cpp
int DotProduct(int x[], int y[], int xsize, int ysize)
{
    int result = 0;
    int j = 0;
    for (int i = 0; i < xsize; i++)
    {
        result = (x[i] * y[j]) + result;
        j++;
    }

    return result;

}
```

Compiler Assembly listing code for Dot Product function using Index arithmetic:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1

    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC  ?DotProduct@@YAHQAH0HH@Z            ; DotProduct
EXTRN   __RTC_InitBase:PROC
EXTRN   __RTC_Shutdown:PROC
;   COMDAT rtc$TMZ
;rtc$TMZ    SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
;rtc$TMZ    ENDS
;   COMDAT rtc$IMZ
;rtc$IMZ    SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
;rtc$IMZ    ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;   COMDAT ?DotProduct@@YAHQAH0HH@Z
_TEXT   SEGMENT
_i$1 = -32                          ; size = 4
_j$ = -20                           ; size = 4
_result$ = -8                           ; size = 4
_x$ = 8                             ; size = 4
_y$ = 12                            ; size = 4
_xsize$ = 16                            ; size = 4
_ysize$ = 20                            ; size = 4
?DotProduct@@YAHQAH0HH@Z PROC               ; DotProduct, COMDAT
; File c:\users\demetri\documents\visual studio 2013\projects\dotproduct\dotproduct\dotproduct.cpp
; Line 2
    push    ebp
    mov ebp, esp
    sub esp, 228                ; 000000e4H
    push    ebx
    push    esi
    push    edi
```

Continued…

```
        lea edi, DWORD PTR [ebp-228]
        mov ecx, 57                         ; 00000039H
        mov eax, -858993460                     ; ccccccccH
        rep stosd
    ; Line 3
        mov DWORD PTR _result$[ebp], 0
    ; Line 4
        mov DWORD PTR _j$[ebp], 0
    ; Line 5
        mov DWORD PTR _i$1[ebp], 0
        jmp SHORT $LN3@DotProduct
    $LN2@DotProduct:
        mov eax, DWORD PTR _i$1[ebp]
        add eax, 1
        mov DWORD PTR _i$1[ebp], eax
    $LN3@DotProduct:
        mov eax, DWORD PTR _i$1[ebp]
        cmp eax, DWORD PTR _xsize$[ebp]
        jge SHORT $LN1@DotProduct
    ; Line 7
        mov eax, DWORD PTR _i$1[ebp]
        mov ecx, DWORD PTR _x$[ebp]
        mov edx, DWORD PTR _j$[ebp]
        mov esi, DWORD PTR _y$[ebp]
        mov eax, DWORD PTR [ecx+eax*4]
        imul    eax, DWORD PTR [esi+edx*4]
        add eax, DWORD PTR _result$[ebp]
        mov DWORD PTR _result$[ebp], eax
    ; Line 8
        mov eax, DWORD PTR _j$[ebp]
        add eax, 1
        mov DWORD PTR _j$[ebp], eax
    ; Line 9
        jmp SHORT $LN2@DotProduct
    $LN1@DotProduct:
    ; Line 11
        mov eax, DWORD PTR _result$[ebp]
    ; Line 13
        pop edi
        pop esi
        pop ebx
```

Optimization of complier Code using Index arithmetic for Dot Product computation:

```
.686P
.model  flat
INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES
PUBLIC  ?DotProduct@@YAHQAH0HH@Z            ; DotProduct
_TEXT   SEGMENT
_i$1 = -32                      ; size = 4
_j$ = -20                       ; size = 4
_result$ = -8                       ; size = 4
_x$ = 8                         ; size = 4
_y$ = 12                        ; size = 4
_xsize$ = 16                        ; size = 4
_ysize$ = 20                        ; size = 4

?DotProduct@@YAHQAH0HH@Z PROC               ; DotProduct, COMDAT
; Line 2
    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
    mov DWORD PTR _result$[ebp], 0          ;result=0
    mov DWORD PTR _j$[ebp], 0              ; j=0
    mov DWORD PTR _i$1[ebp], 0             ; i=0

```

Continue…

```
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;for condition (i<size)
    $Loop:
    cmp eax, DWORD PTR _xsize$[ebp]         ; compare index i with the size of the array
    jge SHORT $Exit

    ;; increase i by one
    mov eax, DWORD PTR _i$1[ebp]            ; i is saved to eax
    add eax, 1                              ; eax increases i by 1
    mov DWORD PTR _i$1[ebp], eax             ; saves it back to i

    ;;;;; result= X[i]*Y[i] + result;
    mov eax, DWORD PTR _i$1[ebp]     ; move  i to eax
    mov ecx, DWORD PTR _x$[ebp]       ;  move addres of x to ecx
    mov edx, DWORD PTR _j$[ebp]       ; move j to edx
    mov esi, DWORD PTR _y$[ebp]        ; move address of y to esi
    mov eax, DWORD PTR [ecx+eax*4]       ; increment i and move it toeax
    imul eax, DWORD PTR [esi+edx*4]  ; y[j]
    add eax, DWORD PTR _result$[ebp]     ; add result to eax
    mov DWORD PTR _result$[ebp], eax     ; save eax back to results
    ;;;;; increase j
    mov eax, DWORD PTR _j$[ebp]          ; move j value to eax
    add eax, 1                            ; add 1 to eax
    mov DWORD PTR _j$[ebp], eax            ; save eax back to j
    ;;; Save i to eax
    mov eax, DWORD PTR _i$1[ebp]            ; move i to eax
    jmp SHORT $Loop

    $Exit
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Line 13
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?DotProduct@@YAHQAH0HH@Z ENDP               ; DotProduct
_TEXT   ENDS
END
```

In the last two pages there is a generated compiler assembly code and a corresponding optimized compiler generated assembly code for dot product computation using indexing arithmetic. The compiler generated code optimize code contains the variables i, j and result outside of the loop. The optimize code contains less number of loops that are in the compiler generated assembly code. They both follow the same functionality. Initialize the first three variables to zero. Check condition. The condition is whether or not the current index has reached to the size of the array. If it has not reach the end of the array, then the dot product computation continues to happen. If the index has reached the last index the function begins to exit and return the value result.

Dot Product using Pointer Technique:

main.cpp

```
#include<iostream>
using namespace std;

int DotProductPointers(const int* x,const int* y, int size);

static int A[5] = { 1, 2, 3,4,5};
static int B[5] = { 1, 2, 3,4,5};
int main()
{
    int e = DotProductPointers(A, B, 5);
    cout << e << endl;

    system("Pause");
    return 0;
}
```

dotproduct.cpp

```
int DotProductPointers(const int *a, const int *b, int size) {

    const int *p, *q;
    int result = 0;
    for (p = a, q = b; p < a + size; p++, q++) {
        result += *p * *q;
    }
    return result;
}
```

Compiler Assembly listing code for Dot Product function using Pointer arithmetic:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1

    TITLE   c:\Users\Demetri\documents\visual studio 2013\Projects\Project5\Project5\DotProductPointer.cpp
    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC  ?DotProductPointers@@YAHPBH0H@Z         ; DotProductPointers
EXTRN   __RTC_InitBase:PROC
EXTRN   __RTC_Shutdown:PROC
;   COMDAT rtc$TMZ
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;   COMDAT ?DotProductPointers@@YAHPBH0H@Z
_TEXT   SEGMENT
_result$ = -32                      ; size = 4
_q$ = -20                       ; size = 4
_p$ = -8                        ; size = 4
_a$ = 8                         ; size = 4
_b$ = 12                        ; size = 4
_size$ = 16                     ; size = 4
?DotProductPointers@@YAHPBH0H@Z PROC            ; DotProductPointers, COMDAT
; File c:\users\demetri\documents\visual studio 2013\projects\project5\project5\dotproductpointer.cpp
; Line 1
    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
; Line 4
    mov DWORD PTR _result$[ebp], 0
; Line 5
    mov eax, DWORD PTR _a$[ebp]
    mov DWORD PTR _p$[ebp], eax
    mov ecx, DWORD PTR _b$[ebp]
    mov DWORD PTR _q$[ebp], ecx
    jmp SHORT $LN3@DotProduct
$LN2@DotProduct:
    mov eax, DWORD PTR _p$[ebp]
    add eax, 4
    mov DWORD PTR _p$[ebp], eax
    mov ecx, DWORD PTR _q$[ebp]
    add ecx, 4
    mov DWORD PTR _q$[ebp], ecx
$LN3@DotProduct:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _a$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _p$[ebp], edx
    jae SHORT $LN1@DotProduct
; Line 6
    mov eax, DWORD PTR _p$[ebp]
    mov ecx, DWORD PTR _q$[ebp]
    mov edx, DWORD PTR [eax]
    imul    edx, DWORD PTR [ecx]
    add edx, DWORD PTR _result$[ebp]
    mov DWORD PTR _result$[ebp], edx
; Line 7
    jmp SHORT $LN2@DotProduct
$LN1@DotProduct:
; Line 8
    mov eax, DWORD PTR _result$[ebp]
; Line 9
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
?DotProductPointers@@YAHPBH0H@Z ENDP            ; DotProductPointers
_TEXT   ENDS
END
```

Optimizing compiler Assembly listing code for Dot Product function using Pointer arithmetic:

```
	.686P
	.XMM
	include listing.inc
	.model	flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC	?DotProductPointers@@YAHPBH0H@Z		; DotProductPointers
EXTRN	__RTC_InitBase:PROC
EXTRN	__RTC_Shutdown:PROC

_TEXT	SEGMENT
_result$ = -32					; size = 4
_q$ = -20					; size = 4
_p$ = -8					; size = 4
_a$ = 8						; size = 4
_b$ = 12					; size = 4
_size$ = 16					; size = 4
?DotProductPointers@@YAHPBH0H@Z PROC			; DotProductPointers, COMDAT

; Line 1
	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

	mov DWORD PTR _result$[ebp], 0	; results =0
	mov eax, DWORD PTR _a$[ebp]	;
	mov DWORD PTR _p$[ebp], eax	; p points a
	mov ecx, DWORD PTR _b$[ebp]	;
	mov DWORD PTR _q$[ebp], ecx	; q points b

	;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
	; calculate the end of the array and compare current pointer to the end pointer

	$Loop:
	mov eax, DWORD PTR _size$[ebp]
	mov ecx, DWORD PTR _a$[ebp]
	lea edx, DWORD PTR [ecx+eax*4]
	cmp DWORD PTR _p$[ebp], edx
	jae SHORT $Exit
	; Muiltply result= (x[i] +y[i])  + result;
	mov eax, DWORD PTR _p$[ebp]
	mov ecx, DWORD PTR _q$[ebp]
	mov edx, DWORD PTR [eax]
	imul	edx, DWORD PTR [ecx]
	add edx, DWORD PTR _result$[ebp]
	mov DWORD PTR _result$[ebp], edx
	; increment pointer p and q by 4 bytes
	mov eax, DWORD PTR _p$[ebp]
	add eax, 4
	mov DWORD PTR _p$[ebp], eax
	mov ecx, DWORD PTR _q$[ebp]
	add ecx, 4
	mov DWORD PTR _q$[ebp], ecx
	jmp SHORT $Loop

	;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
$Exit:
; Line 8
	mov eax, DWORD PTR _result$[ebp]
; Line 9
	pop edi
	pop esi
	pop ebx
	mov esp, ebp
	pop ebp
	ret 0
?DotProductPointers@@YAHPBH0H@Z ENDP		; DotProductPointers
_TEXT	ENDS
END
```

In the last two pages is the implementation of the dot product using pointer arithmetic. On page 24, is the compiler generated assembly code. On page 25, is the optimized compiler generated assembly code. Both codes are similar to each other. There were no other ways to optimize the code. There are less jumps in the optimize code then the compiler generated code. They have the same functionality. They both have pointer p and q point to a and b respectively. Then the size of the array is calculated. The size is needed to compute the end of the array. The memory location of the end is being compared to the current pointer in p. If the pointer p has not reach the end, then the dot product computation is continued.

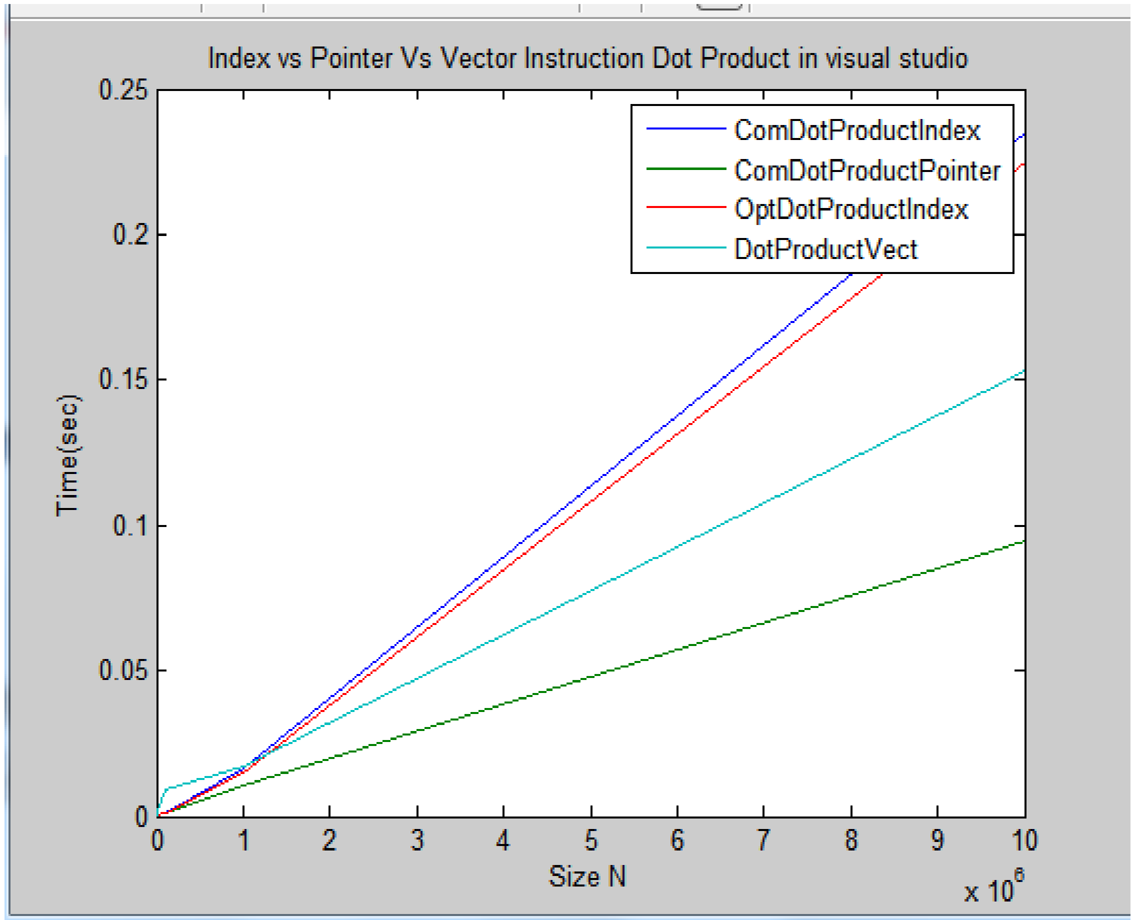
Dot Product Computation Using Vector Instruction:

```
// VectorInstructionDotProduct.cpp : main project file.
#include "stdafx.h"
extern "C" {
#include<xmmintrin.h>
}

#include<iostream>
#include<fstream>
#include<windows.h>

using namespace std;
using namespace System;

void DotProduct(float x[], float y[], int size);

_declspec(align(16)) static float A[10000000];
_declspec(align(16)) static float B[10000000];
// 8 25 250 2500
int N[7] = { 8, 100, 1000, 10000, 100000, 1000000, 10000000 };

int main()
{
    int size = 10;
    _int64 ctrl = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;

    double Avg = 0.0;
    ofstream myfile;
    myfile.open("VectorDotProductdata.txt");


    double k = 0;
    for (int p = 0; p < 10000000; p++)
    {
        A[p] = 1;
        B[p] = 1;
    }


    for (int u = 0; u < 7; u++){
         for (int i = 0; i < 5; i++)
            {

            if (QueryPerformanceCounter((LARGE_INTEGER*)&ctrl) != 0){
                DotProduct(A, B, N[u]);
                QueryPerformanceCounter((LARGE_INTEGER*)&ctr2);
                QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

                k = ("{0}", (((ctr2 - ctrl)* 1.0) / freq)) + k;
            }

          }
        Avg = k / 5;
        myfile << Avg << "\t" << N[u] << "\n";

```

```
        }

    Console::WriteLine(Avg);
    myfile.close();
    Console::WriteLine();

    Console::Read();


    return 0;
}


void DotProduct(float x[], float y[], int size)
{
    __m128 temp1, temp2, res;
    float sum = 0.0;
    _declspec(align(16)) float temp4[4];

    for (int e = 0; e < size; e += 4)
    {
        temp1 = _mm_load_ps(x + e);
        temp2 = _mm_load_ps(y + e);
        res = _mm_mul_ps(temp1, temp2);
        res = _mm_hadd_ps(res, temp2);
        res = _mm_hadd_ps(res, temp2);
        _mm_store_ps(temp4, res);
        sum += temp4[0];

    }

    cout << sum << endl;

}
```

In the code above is the implementation of dot product using vector instructions. The inputs to the dot product functions are two arrays. The two arrays are turned into vector of size N. The loop reads every four values of each vector until size N in groups of four. The _mm_mul_ps command multiples each group of four from both vectors and inputs them into result (res). The result is added horizontally twice to perform the sum of the dot product.   The next instruction _mm_store_ps stores the value of result to a temporary array of four. The result of the dot-product of every four groups is stored at the first index of temp four. Then variable sum acts like an accumulator and adds all the result from each group of four to produce a single sum. The code above iterates different values of N (the size of the two arrays). It calculates the time five times and computes the avg. The data is then stored to an external text file.

Performance Measurements:



The performances of the dot product with different implementation are shown above. The fastest implementation of the dot product is the compiler generated assembly code using pointer arithmetic. The second fastest is the dot-product computation using vector instructions. Ideally the vector instruction implementation should be the fastest in terms of performance. One reason that it is not the fastest because the loop to multiple and adds in groups of four makes it sequential statement and not in parallel with the other elements of the array. This is not the ideal parallel performance that is expected with vector instructions. The loop makes the calculation sequential in parallel groups of four instead of size multiply groups of four computations in parallel.

Conclusion:

The compiler does not always generate the best assembly code. The fastest way to clear an array by initialize each element to zero and dot product computation is using pointer arithmetic. The second fastest performance for calculating the dot product is the implementation of vector instructions. The compiler in Linux operating system appears to generated assembly code better than the visual studio compiler. However, the assembly code generated by the visual studio compiler generates assembly code that it is easier to understand.

I learned a lot with this take home test. I learned to generated assembly compiler code and run it side by side with a c++ file for both windows and Linux operating systems. I learn that you can optimize compiler generated code. I also learned about vector instructions and how to implement it. The most importantly I learned to calculate time in visual studio by query-performance counter and in Linux get-time.

Appendix:

Generated Measurement plots (Matlab):

```
%% Take home test 3
%% Plot Index vs Pointer Clearing Array in visual studio
x=[10, 100, 1000, 10000, 100000, 1000000, 10000000];% Size N
compilerIndexCode=[3.56331e-005,3.75785e-005,4.40294e-005,8.23247e-005,0.000503266,0.00450646,0.044986 ];% time seconds
compilerPointerCode=[0.0000290799,0.0000309229,0.0000378857, 0.0000876491,0.000560606,0.0114575,0.110457];
optIndexCompCode=[3.39E-05,3.56E-05,4.36E-05,8.67E-05,0.000507771,0.00635917,0.0496016];
optPointerCode=[4.09E-05,4.28E-05,4.90E-05,9.16E-05,0.000516884,0.0083963,0.104164];

plot(x,compilerIndexCode,x,compilerPointerCode,x,optIndexCompCode,x,optPointerCode)
xlabel('Size N')
ylabel('Time(sec)')
legend('compilerIndexCode','compilerPointerCode','optIndexCompCode', 'optPointerCode')
title('Compiler and optimize code Index vs Pointer Clearing Array in Visual Studio')


%% Plot Index vs Pointer Clearing Array in Linux
x=[10, 100, 1000, 10000, 100000, 1000000, 10000000];% Size N
compilerIndexCode=[16020.2,3157,9219.4,50928.4, 461470,5.21314e+06, 4.88297e+07];% time seconds
compilerPointerCode=[1242.8, 2877.2, 9707.6,44502.8,427304, 4.49783e+06, 4.47219e+07];% time seconds
optcompIndexCode=[1257.2, 3324.8,12711.6,87693.4,672992,6.00486e+06, 5.2492e+07];% time seconds
optcompPointerCode=[1326.8,2793.2,7961,103061,535444,4.43619e+06,4.30305e+07];% time seconds
plot(x,compilerIndexCode,x,compilerPointerCode,x,optcompIndexCode,x,optcompPointerCode);
legend('compilerIndexCode','compilerPointerCode','optcompIndexCode','optcompPointerCode')
xlabel('Size N')
ylabel('Time(nsec)')
title('Index vs Pointer Clearing Array in Linux')
%% Plot Index VS Pointer Scaler and vector Dot Product in Visual Studio
x=[10, 100, 1000, 10000, 100000, 1000000, 10000000];% Size N
ComDotProductIndex=[4.9661e-005,5.21185e-005,6.28698e-005,.000148778, .00106705,.0167145,.234933];% time seconds
ComDotProductPointer=[3.82953e-005,4.0548e-005,4.97634e-005,0.000141713,0.00091366,0.0109088,0.0948066];
OptDotProductIndex=[3.6657e-005,3.87049e-005,4.77155e-005,0.000116524,0.00079857,0.0150687,0.225133];
DotProductVect=[0.000529379,0.00091817,0.00128884,0.00175658,0.00911156,0.0168489,0.153381];
plot(x,ComDotProductIndex,x,ComDotProductPointer,x,OptDotProductIndex,x,DotProductVect);
%axis([0 100000000 0 1000000000000]);
xlabel('Size N')
ylabel('Time(sec)')
legend('ComDotProductIndex','ComDotProductPointer','OptDotProductIndex','DotProductVect')
title('Index vs Pointer Vs Vector Instruction Dot Product in visual studio')
```