



# Multiprocessors

Feb-Jun 2021

5

## Vectorization

Alejandro Guajardo Moreno



## From last class...

Understand why Memory Alignment is important. Read the following:

<https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization>

[https://en.wikipedia.org/wiki/Data\\_structure\\_alignment](https://en.wikipedia.org/wiki/Data_structure_alignment)

<https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/aligned-malloc?view=vs-2019>

Look for the file “Single Threaded Array Addition (FLOATS)s” in Canvas.

- 1.- Document the time it took to process the loop BEFORE changing the code. Run 5 times the program. Average the 5 results. This will be your BASELINE for comparisons (The ## in “Results verified!!! (##)”).
- 2.- Rewrite the loop commented “//This loop can be optimized using Intrinsics” using **Intrinsics**. First use SSE (128-bit) Instruction Set Extensions. Run it 5 times and average the runs. Document the time it took to process the loop.
- 3.- Change again the loop and now use AVX (256-bit) Instruction Set Extensions. Run 5 times, average and document.
- 4.- Upload to the platform a Word or PDF Document which includes:
  - 1 Table comparing the times it took to run each case and the % of improvement vs. your BASELINE.
  - Your conclusions.
- 5.- Upload your Visual Studio Solution or GCC's code of both SSE and AVX (You can add a menu to the program to choose which code to execute or leave commented one of the codes).

**Note:** Use what you learned on Memory Alignment. You may have to use `__declspec(align())` when defining your variables, depending on the Intrinsics used.

**Note 2:** You might want to use the Intrinsics: `_load_ps()`, `_loadu_ps()`, `_store_ps()` or `_storeu_ps()`.



## Solution...

```
// No Vector  
for (i = 0; i < elements; i++)  
    C[i] = A[i] + B[i];
```

|     | BASELINE | SSE | AVX |
|-----|----------|-----|-----|
| 1   | 306      |     |     |
| 2   | 302      |     |     |
| 3   | 305      |     |     |
| 4   | 302      |     |     |
| 5   | 303      |     |     |
| AVG | 303.6    |     |     |
| %   | -        |     |     |



## Solution...

```
// Using SSE Extensions
for (i = 0; i < elements; i += 4) {
    __m128 a = _mm_loadu_ps(&A[i]);
    __m128 b = _mm_loadu_ps(&B[i]);
    __m128 sum4 = _mm_add_ps(a, b);
    _mm_storeu_ps(&C[i], sum4);
}
```

|     | BASELINE | SSE  | AVX |
|-----|----------|------|-----|
| 1   | 306      | 94   |     |
| 2   | 302      | 98   |     |
| 3   | 305      | 96   |     |
| 4   | 302      | 96   |     |
| 5   | 303      | 95   |     |
| AVG | 303.6    | 95.8 |     |
| %   | -        | 317% |     |





## Solution...

```
// Using AVX/AVX2 Extensions
for (i = 0; i < elements; i += 8) {
    __m256 a = _mm256_loadu_ps(&A[i]);
    __m256 b = _mm256_loadu_ps(&B[i]);
    __m256 sum4 = _mm256_add_ps(a, b);
    _mm256_storeu_ps(&C[i], sum4);
}
```

|     | BASELINE | SSE  | AVX  |
|-----|----------|------|------|
| 1   | 306      | 94   | 79   |
| 2   | 302      | 98   | 79   |
| 3   | 305      | 96   | 76   |
| 4   | 302      | 96   | 78   |
| 5   | 303      | 95   | 77   |
| AVG | 303.6    | 95.8 | 77.8 |
| %   | -        | 317% | 390% |



## Conclusions and notes...

- Not the best option, but the most simple: Use unaligned loads and stores: `loadu/storeu`
- When using SSE or AVX, change your cycles, depending on your type of data:

`for (i = 0; i < elements; i++)`

`SSE => for (i = 0; i < elements; i += 4) //Valid for floats or ints.`

`AVX => for (i = 0; i < elements; i += 8)`

- The best option: ALIGN your data in memory.
- If we go from scalar to SSE, why don't we see a 4x improvement?
  - Performance of the instruction (CPI)
  - Memory alignment
  - Cache Hits

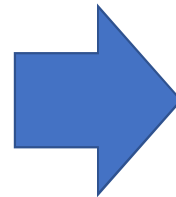


## Memory Alignment (Windows) – `aligned_malloc()`

```
//Array creation
size_t datasize = sizeof(float) * elements;
A = (float*)malloc(datasize);
B = (float*)malloc(datasize);
C = (float*)malloc(datasize);
```

```
//Add loop
for (i = 0; i < elements; i++)
    C[i] = A[i] + B[i];
```

```
//Memory deallocation
free(A);
free(B);
free(C);
```



```
size_t alignment = 16;
```

```
//Array creation
size_t datasize = sizeof(float) * elements;
A = (float*) _aligned_malloc(datasize,alignment);
B = (float*) _aligned_malloc(datasize,alignment);
C = (float*) _aligned_malloc(datasize,alignment);
```

```
//Add loop
for (i=0; i< elements/4 ; i++)
{
    Aint = _mm_load_ps(A + i*4);
    Bint = _mm_load_ps(B + i*4);
    Cint = _mm_add_ps(Aint, Bint);
    _mm_store_ps(C + i*4, Cint);
}
```

```
//Memory deallocation
_aligned_free(A);
_aligned_free(B);
_aligned_free(C);
```

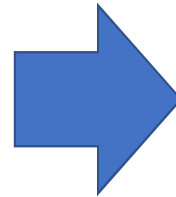


## Memory Alignment (Linux/GCC) – aligned\_alloc()

```
//Array creation
size_t datasize = sizeof(float) * elements;
A = (float*)malloc(datasize);
B = (float*)malloc(datasize);
C = (float*)malloc(datasize);
```

```
//Add loop
for (i = 0; i < elements; i++)
    C[i] = A[i] + B[i];
```

```
//Memory deallocation
free(A);
free(B);
free(C);
```



```
size_t alignment = 16;
```

```
//Array creation
size_t datasize = sizeof(float) * elements;
A = (float*) aligned_alloc(alignment,datasize);
B = (float*) aligned_alloc(alignment,datasize);
C = (float*) aligned_alloc(alignment,datasize);
```

```
//Add loop
for (i=0; i< elements/4 ; i++)
{
    Aint = _mm_load_ps(A + i*4);
    Bint = _mm_load_ps(B + i*4);
    Cint = _mm_add_ps(Aint, Bint);
    _mm_store_ps(C + i*4, Cint);
}
```

```
//Memory deallocation
free(A);
free(B);
free(C);
```





## Remember?



### Other functions

- Logics (AND, OR, XOR, NOT, ...)
- Cache support:
  - `_mm_prefetch(...)`
  - `_mm_stream_ps(...)`
- ... and many others.

Like...

Aligned  
Memory  
Allocation!



## Memory Alignment (Intrinsics Win/Linux) - `_mm_malloc`

```
//Array creation
size_t datasize = sizeof(float) * elements;
A = (float*)malloc(datasize);
B = (float*)malloc(datasize);
C = (float*)malloc(datasize);

//Add loop
for (i = 0; i < elements; i++)
    C[i] = A[i] + B[i];

//Memory deallocation
free(A);
free(B);
free(C);
```



```
size_t datasize = sizeof(float) * elements;
```

```
//Array creation
A = (float*)_mm_malloc(datasize, 16);
B = (float*)_mm_malloc(datasize, 16);
C = (float*)_mm_malloc(datasize, 16);

//Add loop
for (i = 0; i < elements; i += 4)
{
    __m128 a = _mm_load_ps(&A[i]);
    __m128 b = _mm_load_ps(&B[i]);
    __m128 sum4 = _mm_add_ps(a, b);
    _mm_store_ps(&C[i], sum4);
}

//Memory Deallocation
_mm_free(A);
_mm_free(B);
_mm_free(C);
```

## Structure Alignment (Windows) - \_\_declspec(align())

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

int main() {

    int __declspec(align(32)) A[SIZE];
    int __declspec(align(32)) B[SIZE];

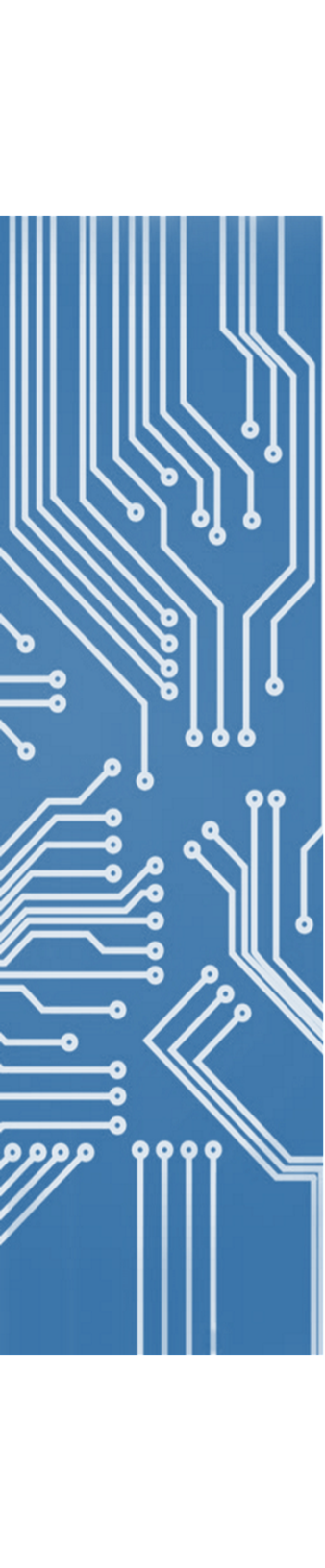
    for (int i = 0; i < SIZE; i++)
        A[i] = B[i] = i;

    for (int i = 0; i < SIZE; i++)
        A[i] = A[i] + B[i];

    for (int i = 0; i < 1024; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);

    return 0;
}
```

Align a structure in memory from its creation...



## Structure Alignment (GCC) - `__attribute__((aligned ()))`

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

int main() {

    int A[SIZE] __attribute__((aligned (32)));
    int B[SIZE] __attribute__((aligned (32)));


    for (int i = 0; i < SIZE; i++)
        A[i] = B[i] = i;

    for (int i = 0; i < SIZE; i++)
        A[i] = A[i] + B[i];

    for (int i = 0; i < 1024; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);

    return 0;
}
```

Align a structure  
in memory from  
its creation...





# Options for vectorization

Assembly language

```
..B8.5
MOVAPS a(,%rdx,4), %xmm0
ADDPA b(,%rdx,4), %xmm0
MOVAPS %xmm0, c(,%rdx,4)
ADDQ $4, %rdx
CMPQ $rdi, %rdx
JL ..B8.5
```

Explicit Vectorization

## Intrinsic Functions

- Middle ground Assembly  $\Leftrightarrow$  C
- Direct Access to processor instructions
- Retain C Syntax

```
void ejemplo(){
    __m128 rA, rB, rC;
    for (int i = 0; i<LEN; i+=4){
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC = _mm_add_ps(rA,rB);
        _mm_store_ps(&c[i], rC);
    }
}
```

Automatic Vectorization

Source code in C language.  
+ Vectorizing compiler (#pragma).  
&/| Vectorizing compiler options (/arch)

```
#pragma xyz
for (i=0; i<LEN; i++)
    c[i] = a[i] + b[i];
```





# Auto-vectorization

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

int main() {

    int __declspec(align(32)) A[SIZE];
    int __declspec(align(32)) B[SIZE];

    for (int i = 0; i < SIZE; i++)
        A[i] = B[i] = i;

    // This loop will be auto-vectorized
    for (int i = 0; i < SIZE; i++)
        A[i] = A[i] + B[i];

    for (int i = 0; i < 1024; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);

    return 0;
}
```

- Normal C code
- Data is aligned.
- Compiler should take care of it.



## Auto-vectorization

```
// This loop will be auto-vectorized
for (int i = 0; i < SIZE; i++)
00151076 xor     ecx,ecx
00151078 jmp     main+30h (0151080h)
0015107A lea     ebx,[ebx]
        A[i] = A[i] + B[i];
00151080 mov     eax,dword ptr [esp+ecx+1040h]
00151087 add     dword ptr [esp+ecx+40h],eax
0015108B mov     eax,dword ptr [esp+ecx+1044h]
00151092 add     dword ptr [esp+ecx+44h],eax
00151096 mov     eax,dword ptr [esp+ecx+1048h]
0015109D add     dword ptr [esp+ecx+48h],eax
        A[i] = A[i] + B[i];
001510A1 mov     eax,dword ptr [esp+ecx+104Ch]
001510A8 add     dword ptr [esp+ecx+4Ch],eax
001510AC add     ecx,10h
001510AF cmp     ecx,1000h
001510B5 jnl     main+30h (0151080h)
```

If it is auto-vectorizing,  
why do we see General  
Registers?

✓ Just some  
configuration needed.

# Auto-vectorization

Let's enable the compiler's ability to generate vectorized code:

The screenshot shows the Visual Studio IDE with the 'Test1.c' file open. The 'Project Properties' dialog is open, showing the 'Configuration: Active(Debug)' and 'Platform: Win32'. The 'C/C++' section is expanded, and the 'Code Generation' tab is selected. The 'Basic Runtime Checks (Cleared)' checkbox is checked. The 'Compiler' and 'Code Generation' sections are highlighted with blue arrows. The 'Basic Runtime Checks' section is also highlighted with a blue arrow.

**Select Project => Properties**

**Basic Runtime Checks (Cleared)**

**Compiler**

**Code Generation**

| Property                         | Value                           |
|----------------------------------|---------------------------------|
| Enable String Pooling            | No (/Gm-)                       |
| Enable Minimal Rebuild           | No (/Gm-)                       |
| Enable C++ Exceptions            | Yes (/EHsc)                     |
| Smaller Type Check               | No                              |
| Basic Runtime Checks             | Checked                         |
| Runtime Library                  | Multi-threaded Debug DLL (/MDd) |
| Struct Member Alignment          | Default                         |
| Security Check                   | Enable Security Check (/GS)     |
| Control Flow Guard               | Not Set                         |
| Enable Function-Level Linking    | Not Set                         |
| Enable Parallel Code Generation  | Not Set                         |
| Enable Enhanced Instruction Set  | Not Set                         |
| Floating Point Model             | Not Set                         |
| Enable Floating Point Exceptions | Not Set                         |
| Create Hotpatchable Image        | Not Set                         |
| Spectre Mitigation               | Disabled                        |

**Basic Runtime Checks**  
Perform basic runtime error checks, incompatible with any optimization type other than debug. (/RTC, /RTCu, /RTC1)

OK Cancel Apply

# Auto-vectorization

Let's enable the compiler's ability to generate vectorized code:

The screenshot shows the Visual Studio IDE with the 'Test1 Property Pages' dialog open. The 'Configuration' is set to 'Active(Relase)' and the 'Platform' is 'Win32'. The 'C/C++' section is expanded, and 'Code Generation' is selected. The 'Enable Enhanced Instruction Set' option is highlighted in the list, showing the value 'No Enhanced Instructions (/arch:IA32)'. A blue arrow points from the 'Compiler' label to the 'C/C++' section, and another blue arrow points from the 'Code Generation' label to the 'Code Generation' section. A third blue arrow points from the 'Enable Instruction Set Extensions' label to the 'Enable Enhanced Instruction Set' option.

**Select Project => Properties**

**Compiler**

**Code Generation**

**Enable Instruction Set Extensions**

| Property                         | Value                                 |
|----------------------------------|---------------------------------------|
| Enable String Pooling            | No (/Gm-)                             |
| Enable Minimal Rebuild           | Yes (/EHsc)                           |
| Enable C++ Exceptions            | No                                    |
| Smaller Type Check               | Default                               |
| Basic Runtime Checks             | Multi-threaded DLL (/MD)              |
| Runtime Library                  | Default                               |
| Struct Member Alignment          | Enable Security Check (/GS)           |
| Security Check                   | Yes (/Gy)                             |
| Control Flow Guard               | Enable Enhanced Instruction Set       |
| Enable Function-Level Linking    | No Enhanced Instructions (/arch:IA32) |
| Enable Parallel Code Generation  |                                       |
| Enable Enhanced Instruction Set  |                                       |
| Floating Point Model             |                                       |
| Enable Floating Point Exceptions |                                       |
| Create Hotpatchable Image        |                                       |
| Spectre Mitigation               | Disabled                              |

**Enable Enhanced Instruction Set**  
Enable use of instructions found on processors that support enhanced instruction sets, e.g., the SSE, SSE2, AVX, and AVX2 enhancements to IA-32; AVX and AVX2 to x64. Currently /arch:SSE and /arch:SSE2 are only available when building for the x86 architecture. If no option is specified, the compiler will use ins...

OK Cancel Apply



## Auto-vectorization

Enable Enhanced Instruction Set:

Enable Enhanced Instruction Set

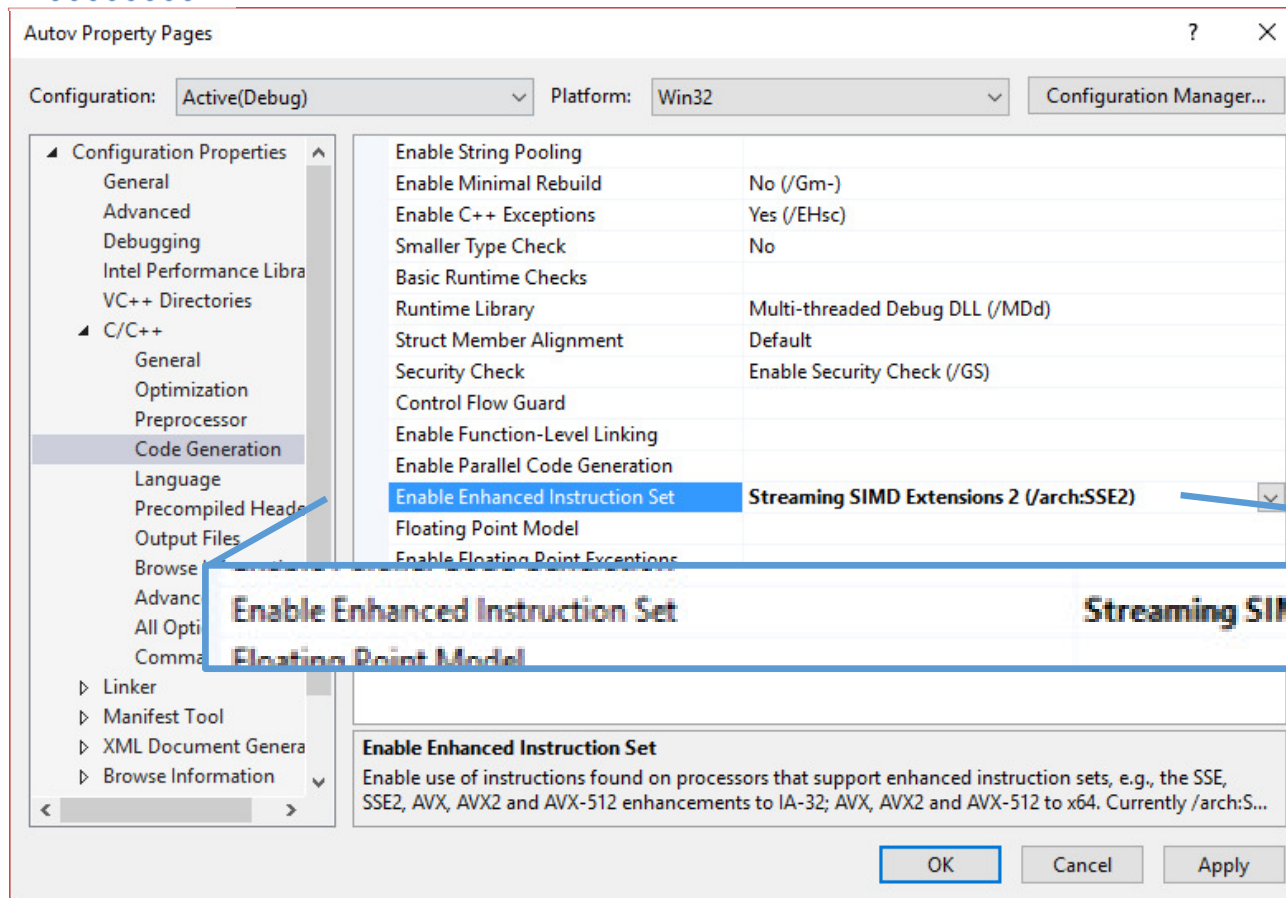
No Enhanced Instructions (/arch:IA32)

This option is adding a compiler flag: `/arch`

- Streaming SIMD Extensions (/arch:SSE)
- Streaming SIMD Extensions 2 (/arch:SSE2)
- Advanced Vector Extensions (/arch:AVX)
- Advanced Vector Extensions 2 (/arch:AVX2)
- No Enhanced Instructions (/arch:IA32)



# Auto-vectorization



# Auto-vectorization

Let's enable the compiler's ability to generate vectorized code:

The screenshot shows the Visual Studio IDE with the 'Test1' project selected in the Server Explorer. The 'Project' menu is open, and the 'Optimization' tab is selected in the Properties window. The 'Compiler Optimization' section is expanded, showing the 'Optimization' tab. The 'Optimization' tab is selected, and the 'Maximum Optimization (Favor Speed) (/O2)' option is chosen. The 'Optimization' section is expanded, showing the 'Optimization' tab. The 'Optimization' tab is selected, and the 'Maximum Optimization (Favor Speed) (/O2)' option is chosen.

| Optimization                    | Maximum Optimization (Favor Speed) (/O2) |
|---------------------------------|--|
| Inline Function Expansion       | Default                                  |
| Enable Intrinsic Functions      | No                                       |
| Favor Size Or Speed             | Neither                                  |
| Omit Frame Pointers             | No (/Oy-)                                |
| Enable Fiber-Safe Optimizations | No                                       |
| Whole Program Optimization      | No                                       |

**Optimization**  
Select option for code optimization; choose Custom to use specific optimization options. /Od, /O1, /O2

OK Cancel Apply



## Auto-vectorization

```
// This loop will be auto-vectorized
for (int i = 0; i < SIZE; i++)
00DE10F0  xor     eax,eax
        A[i] = A[i] + B[i];
00DE10F2  movups  xmm0,xmmword ptr [esp+eax+40h]
00DE10F7  movups  xmm1,xmmword ptr [esp+eax+1040h]
00DE10FF  padd    xmm1,xmm0
00DE1103  movups  xmmword ptr [esp+eax+40h],xmm1
00DE1108  movups  xmm0,xmmword ptr [esp+eax+50h]
00DE110D  movups  xmm1,xmmword ptr [esp+eax+1050h]
00DE1115  padd    xmm1,xmm0
00DE1119  movups  xmmword ptr [esp+eax+50h],xmm1
00DE111E  movups  xmm0,xmmword ptr [esp+eax+60h]
00DE1123  movups  xmm1,xmmword ptr [esp+eax+1060h]
00DE112B  padd    xmm1,xmm0
00DE112F  movups  xmmword ptr [esp+eax+60h],xmm1
00DE1134  movups  xmm0,xmmword ptr [esp+eax+70h]
00DE1139  movups  xmm1,xmmword ptr [esp+eax+1070h]
00DE1141  padd    xmm1,xmm0
00DE1145  movups  xmmword ptr [esp+eax+70h],xmm1
00DE114A  add     eax,40h
00DE114D  cmp     eax,1000h
00DE1152  jl      main+0A2h (00DE10F2h)
```

Is it auto-vectorizing?

✓ Yes

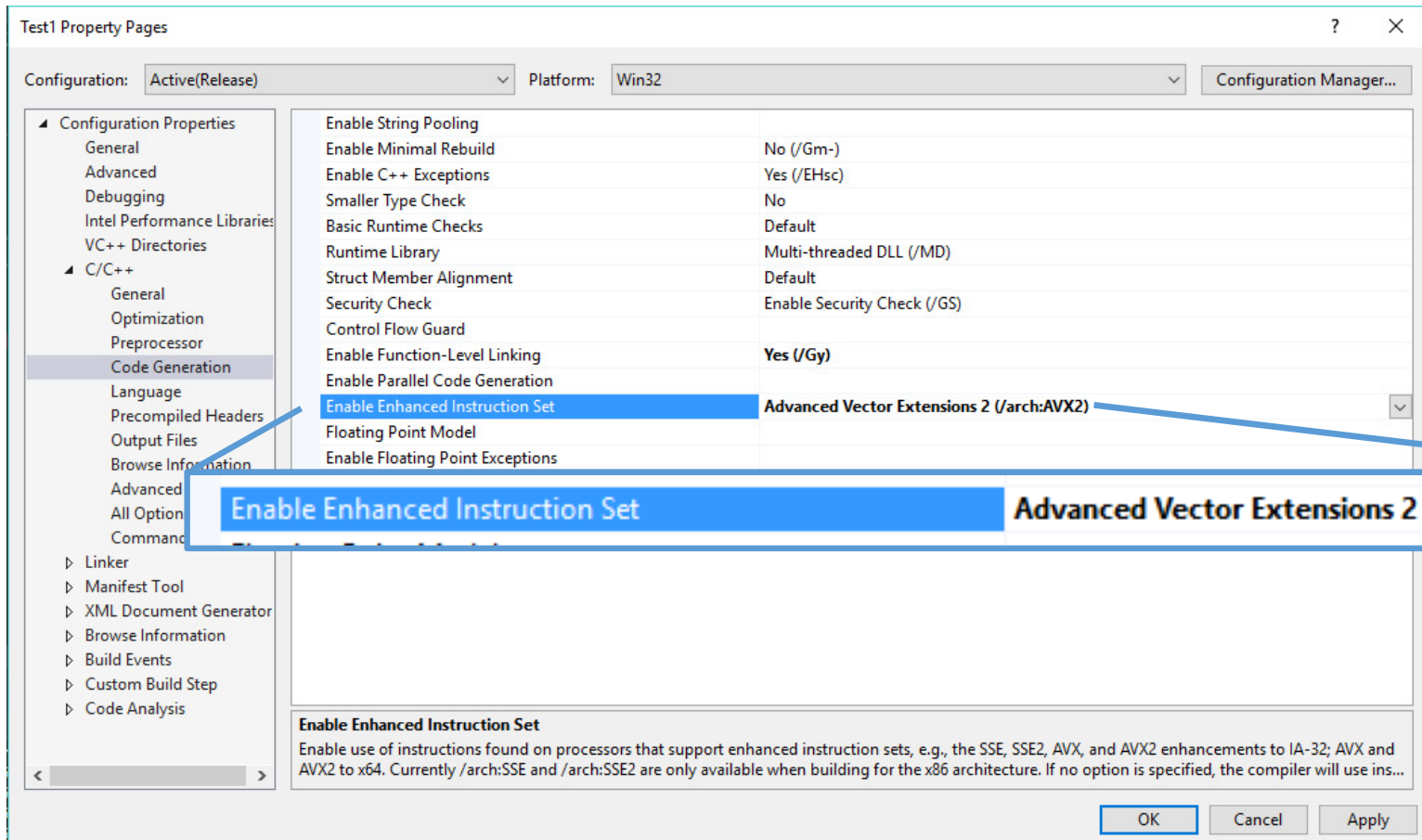
What registers is it using?

✓ SSE 128-bit.

Is it working as expected?

✓ Yes

# Auto-vectorization





## Auto-vectorization

```
// This loop will be auto-vectorized
for (int i = 0; i < SIZE; i++)
01141103 xor     eax,eax
01141105 nop     word ptr [eax+eax]
    A[i] = A[i] + B[i];
01141110 vmovdqu  ymm0,ymmword ptr [esp+eax+1040h]
01141119 vpadd    ymm0,ymm0,ymmword ptr [esp+eax+40h]
0114111F vmovdqu  ymmword ptr [esp+eax+40h],ymm0
01141125 vmovdqu  ymm0,ymmword ptr [esp+eax+1060h]
0114112E vpadd    ymm0,ymm0,ymmword ptr [esp+eax+60h]
01141134 vmovdqu  ymmword ptr [esp+eax+60h],ymm0
0114113A vmovdqu  ymm0,ymmword ptr [esp+eax+1080h]
01141143 vpadd    ymm0,ymm0,ymmword ptr [esp+eax+80h]
0114114C vmovdqu  ymmword ptr [esp+eax+80h],ymm0
01141155 vmovdqu  ymm0,ymmword ptr [esp+eax+10A0h]
0114115E vpadd    ymm0,ymm0,ymmword ptr [esp+eax+0A0h]
01141167 vmovdqu  ymmword ptr [esp+eax+0A0h],ymm0
01141170 sub     eax,0FFFFFF80h
01141173 cmp     eax,1000h
01141178 jl     main+0C0h (01141110h)
```

Is it auto-vectorizing?

✓ Yes

What registers is it using?

✓ AVX 256-bit.

Is it working as expected?

✓ Yes



# Auto-vectorization

```
// This loop will be auto-vectorized
for (int i = 0; i < SIZE; i++)
```

```
00DE10F0 xor     eax,eax
```

```
    A[i] = A[i] + B[i];
```

```
00DE10F2 movups  xmm0,xmmword ptr [esp+eax+40h]
00DE10F7 movups  xmm1,xmmword ptr [esp+eax+1040h]
00DE10FF paddb  xmm1,xmm0
00DE1103 movups  xmmword ptr [esp+eax+40h],xmm1
00DE1108 movups  xmm0,xmmword ptr [esp+eax+50h]
00DE110D movups  xmm1,xmmword ptr [esp+eax+1050h]
00DE1115 paddb  xmm1,xmm0
00DE1119 movups  xmmword ptr [esp+eax+50h],xmm1
00DE111E movups  xmm0,xmmword ptr [esp+eax+60h]
00DE1123 movups  xmm1,xmmword ptr [esp+eax+1060h]
00DE112B paddb  xmm1,xmm0
00DE112F movups  xmmword ptr [esp+eax+60h],xmm1
00DE1134 movups  xmm0,xmmword ptr [esp+eax+70h]
00DE1139 movups  xmm1,xmmword ptr [esp+eax+1070h]
00DE1141 paddb  xmm1,xmm0
00DE1145 movups  xmmword ptr [esp+eax+70h],xmm1
00DE114A add     eax,40h
00DE114D cmp     eax,1000h
00DE1152 jl      main+0A2h (0DE10F2h)
```

```
// This loop will be auto-vectorized
for (int i = 0; i < SIZE; i++)
```

```
01141103 xor     eax,eax
```

```
01141105 nop     word ptr [eax+eax]
```

```
    A[i] = A[i] + B[i];
```

```
01141110 vmovdqu  ymm0,ymmword ptr [esp+eax+1040h]
01141119 vpaddq  ymm0,ymm0,ymmword ptr [esp+eax+40h]
0114111F vmovdqu  ymmword ptr [esp+eax+40h],ymm0
01141125 vmovdqu  ymm0,ymmword ptr [esp+eax+1060h]
0114112E vpaddq  ymm0,ymm0,ymmword ptr [esp+eax+60h]
01141134 vmovdqu  ymmword ptr [esp+eax+60h],ymm0
0114113A vmovdqu  ymm0,ymmword ptr [esp+eax+1080h]
01141143 vpaddq  ymm0,ymm0,ymmword ptr [esp+eax+80h]
0114114C vmovdqu  ymmword ptr [esp+eax+80h],ymm0
01141155 vmovdqu  ymm0,ymmword ptr [esp+eax+10A0h]
0114115E vpaddq  ymm0,ymm0,ymmword ptr [esp+eax+0A0h]
01141167 vmovdqu  ymmword ptr [esp+eax+0A0h],ymm0
01141170 sub     eax,0FFFFFF80h
01141173 cmp     eax,1000h
01141178 jl      main+0C0h (01141110h)
```



## Auto-vectorization in GCC

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

int main() {

    int A[SIZE] __attribute__((aligned(32)));
    int B[SIZE] __attribute__((aligned(32)));

    for (int i = 0; i < SIZE; i++)
        A[i] = B[i] = i;

    // This loop will be auto-vectorized
    for (int i = 0; i < SIZE; i++)
        A[i] = A[i] + B[i];

    for (int i = 0; i < 1024; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);

    return 0;
}
```

- Normal C code
- Data is aligned.
- Compiler should take care of it.



## Auto-vectorization

```
0x0000000000000719 <+111>:  cmpl    $0x3ff,-0x203c(%rbp)
0x0000000000000723 <+121>:  jle     0x6df <main+53>
0x0000000000000725 <+123>:  movl    $0x0,-0x2038(%rbp)
0x000000000000072f <+133>:  jmp     0x769 <main+191>
0x0000000000000731 <+135>:  nop
0x0000000000000732 <+136>:  mov     -0x2038(%rbp),%eax
0x0000000000000738 <+142>:  cltq
- Type <return> to continue, or q <return> to quit--
0x000000000000073a <+144>:  mov     -0x2030(%rbp,%rax,4),%edx
0x0000000000000741 <+151>:  mov     -0x2038(%rbp),%eax
0x0000000000000747 <+157>:  cltq
0x0000000000000749 <+159>:  mov     -0x1030(%rbp,%rax,4),%eax
0x0000000000000750 <+166>:  add     %eax,%edx
0x0000000000000752 <+168>:  mov     -0x2038(%rbp),%eax
0x0000000000000758 <+174>:  cltq
0x000000000000075a <+176>:  mov     %edx,-0x2030(%rbp,%rax,4)
0x0000000000000761 <+183>:  nop
0x0000000000000762 <+184>:  addl    $0x1,-0x2038(%rbp)
0x0000000000000769 <+191>:  cmpl    $0x3ff,-0x2038(%rbp)
0x0000000000000773 <+201>:  jle     0x731 <main+135>
0x0000000000000775 <+203>:  movl    $0x0,-0x2034(%rbp)
0x000000000000077f <+213>:  jmp     0x7bf <main+277>
0x0000000000000781 <+215>:  mov     -0x2034(%rbp),%eax
```

If it is auto-vectorizing,  
why do we see General  
Registers?

✓ Just some  
configuration needed.



# Auto-vectorization

First of all, turn on optimization flag:

`-O`

What optimization do you want?

A: Auto-vectorization!

`-O -ftree-vectorize`

Example: `$ gcc autovec.c -o autovec -O -ftree-vectorize`

```
push    %r10
push    %rbx
sub     $0x2038,%rsp
mov     %fs:0x28,%rax
mov     %rax,-0x38(%rbp)
xor     %eax,%eax
movdqa  0x152(%rip),%xmm0      # 0x860
movdqa  0x15a(%rip),%xmm1      # 0x870
movaps  %xmm0,-0x1050(%rbp,%rax,1)
movaps  %xmm0,-0x2050(%rbp,%rax,1)
add     $0x10,%rax
padd    %xmm1,%xmm0
cmp     $0x1000,%rax
jne     0x716 <main+60>
mov     $0x0,%eax
nop
mov     -0x1050(%rbp,%rax,1),%edx
add     %edx,-0x2050(%rbp,%rax,1)
```



## Auto-vectorization

Want AVX auto-vectorization optimizations?

Simple: `-mavx2`

What about AVX-512?

`-mavx512f`

```
xor    %eax,%eax
vmovdqa 0x152(%rip),%ymm0    # 0x860
vmovdqa 0x16a(%rip),%ymm1    # 0x880
vmovdqa %ymm0,-0x1050(%rbp,%rax,1)
vmovdqa %ymm0,-0x2050(%rbp,%rax,1)
add     $0x20,%rax
vpaddq %ymm1,%ymm0,%ymm0
cmp     $0x1000,%rax
```

```
mov     $0x0,%esi
vmovdqa64 0x1ee(%rip),%zmm1    # 0x980
vmovdqa64 %zmm0,(%r8,%rdx,1)
vmovdqu32 %zmm0,(%rdi,%rdx,1)
add     $0x1,%esi
add     $0x40,%rdx
vpaddq %zmm1,%zmm0,%zmm0
```

Example: `$ gcc autovec.c -o autovec -O -ftree-vectorize -mavx2`





## Auto-vectorization

Ok, so now we know the compiler auto-vectorizes. Is there a way to know without disassembling the code every time?



YES



## Auto-vectorization report (Windows)

`/Qvec-report:{1}{2}`

**`/Qvec-report:1`**

Outputs an informational message for loops that are vectorized.

# Auto-vectorization report (Windows)

The screenshot shows the Visual Studio IDE with the 'Test1 Property Pages' dialog open. The dialog is titled 'Test1 Property Pages' and has a 'Configuration: Active(Relase)' dropdown and a 'Platform: Win32' dropdown. The left pane shows a tree view of property categories. The right pane shows the 'All Options' for the selected category. Annotations with blue arrows point to specific parts of the dialog:

- Select Project => Properties**: Points to the 'Project' menu in the Visual Studio menu bar.
- Compiler**: Points to the 'C/C++' category in the left pane.
- Command Line**: Points to the 'Command Line' sub-category under 'C/C++'.
- Additional Options**: Points to the 'Additional Options' section at the bottom of the dialog, which contains the text '/Qvec-report:1'.

The 'All Options' section displays the following command line options:

```
/permissive- /GS /GL /analyze- /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Fd"Release\vc142.pdb" /Zc:inline /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /arch:AVX2 /Gd /Oy- /Oi /MD /FC /Fa"Release\" /EHsc /nologo /Fo"Release\" /Fp"Release\Test1.pch" /diagnostics:column
```

The 'Additional Options' section contains the text:

```
/Qvec-report:1
```

At the bottom right of the dialog are buttons for 'OK', 'Cancel', and 'Apply'.



## Auto-vectorization report (Windows)

```
133 %  ✓ No issues found

Output
Show output from: Build

1>----- Rebuild All started: Project: Test1, Configuration: Release Win32 -----
1>Test1.c
1>Generating code
1>Previous IPDB not found, fall back to full compilation.
1>
1>--- Analyzing function: main
1>C:\Users\Alex\source\repos\Test1\Test1.c(13) : info C5001: loop vectorized
1>C:\Users\Alex\source\repos\Test1\Test1.c(17) : info C5001: loop vectorized
1>All 4 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>Finished generating code
1>Test1.vcxproj -> C:\Users\Alex\source\repos\Test1\Release\Test1.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
|
```



## Auto-vectorization report (GCC)

`-fopt-info-vec[=file.ext]`

### **-fopt-info-vec**

Outputs an informational message for loops that are vectorized.

```
mp@multiprocesadores:~/multiprocs$ gcc autovec.c -o autovec -O -ftree-vectorize -mavx2 -fopt-info-vec  
autovec.c:11:1: note: loop vectorized
```





Lets try this!

1

Recall Homework code **Single Threaded Array Addition (FLOATs)?**

**Single Threaded Array Addition (FLOATs)**

- 1.- Check your project properties and make sure No Enhanced Instructions (/arch:IA32 or no -O option) is selected.
- 2.- Run the program and take note of the time needed to run.
- 3.- Now change your project's properties and enable Streaming SIMD Extensions 2 (/arch:SSE2 or -O -ftree-vectorize)
- 4.- Run the program and take note of the time.
- 5.- Compare the times and give a conclusion.



Lets try this!

1

Continue with the same code...

- 1.- Now change your project's properties and enable Advanced Vector Extensions 2 (/arch:AVX2)
- 2.- Run the program again and take note of the time.
- 3.- Disable your Enhanced Instruction Sets (/arch:IA32)
- 4.- Replace the code with your 128-bit Intrinsics.
- 5.- Run the program and take note of the time.
- 6.- Replace the code with your 256-bit Intrinsics.
- 7.- Run the program again and take note of the time.
- 8.- Compare the times and give a conclusion.



Lets try this!

1

| No enhancements | SSE2 Auto | SSE Intrin | AVX Auto | AVX Intrin |
|-----------------|-----------|------------|----------|------------|
| 172             | 48        | 55         | 46       | 46         |

- ✓ Auto-vectorized code is better than no vectorization at all (\*not always).
- ✓ Sometimes, using Intrinsics you can get better performance (you, the programmer, have to know what you are doing).
- ✓ Why is SSE Autovectorization better than Intrinsics?
  - ✓ Cache hits, using storeu instead of store or bad alignment.

## Auto-vectorization

So... **Auto-vectorization:**



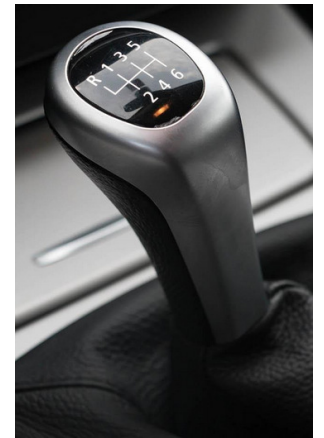
**Auto-vectorization**

**VS**



**Intrinsics**

**VS**



**Assembly Language**

The forced question...  
**Why assembly and  
Intrinsics!?**





Thank you!

See you  
next time!

