Introduction to AVX Vectorization using C

DR. MATTHEW SMITH, MSMITH@ASTRO.SWIN.EDU.AU

Simplest Possible Beginnings

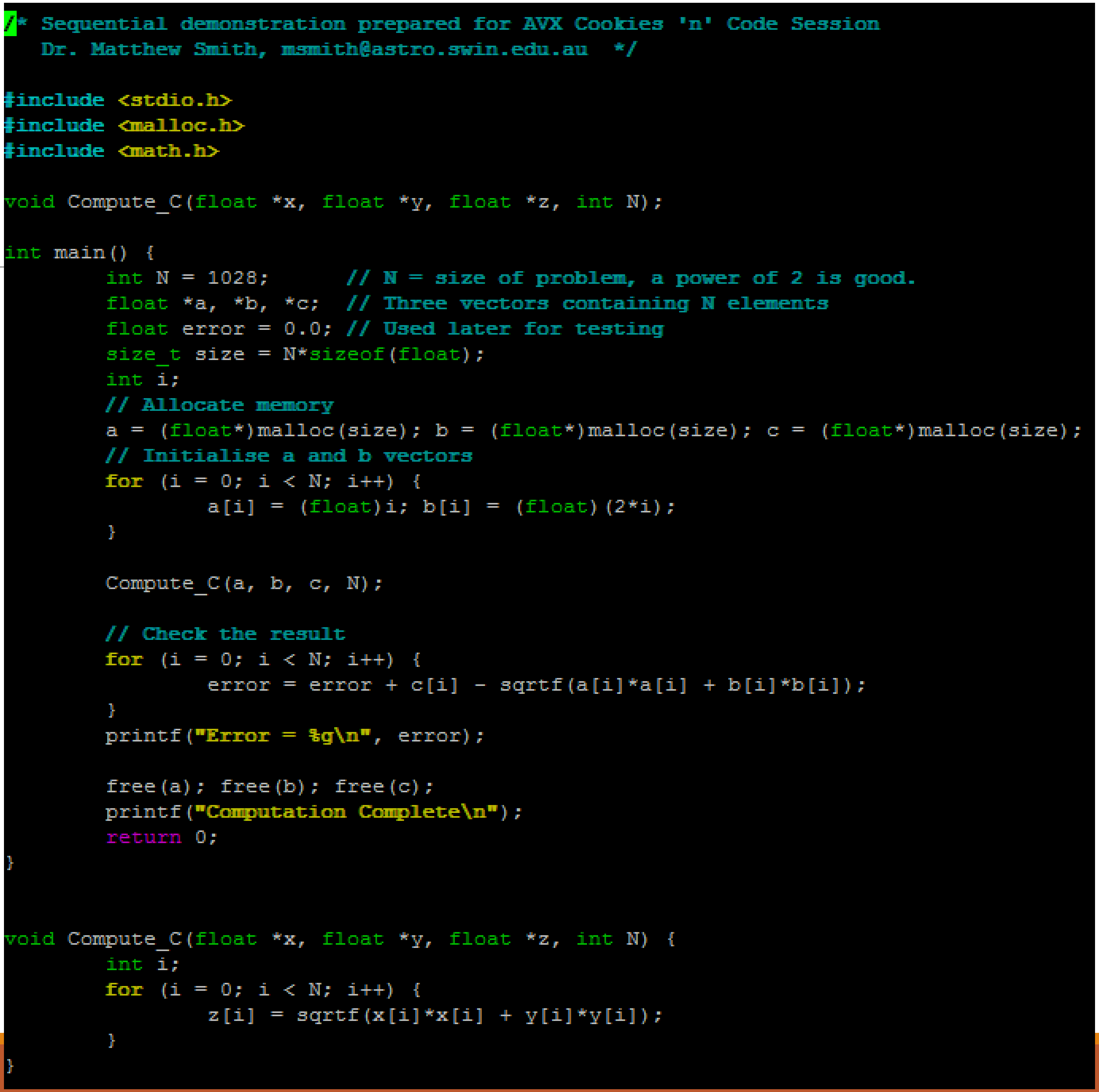
Consider this standard C code performing a simple vector computation:

$$c_i = \sqrt{a_i^2 + b_i^2}$$

for i = 0 to N-1 (Since we are using C)

In this code we use a for loop to go over all N values of a and b to compute C.

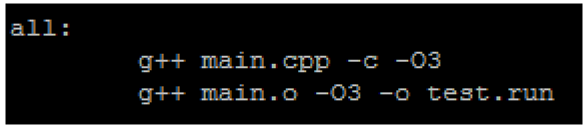
Is this code the best possible code?



```
/* Sequential demonstration prepared for AVX Cookies 'n' Code Session
   Dr. Matthew Smith, msmith@astro.swin.edu.au  */

#include <stdio.h>
#include <malloc.h>
#include <math.h>

void Compute_C(float *x, float *y, float *z, int N);

int main() {
        int N = 1028;       // N = size of problem, a power of 2 is good.
        float *a, *b, *c;   // Three vectors containing N elements
        float error = 0.0;  // Used later for testing
        size_t size = N*sizeof(float);
        int i;
        // Allocate memory
        a = (float*)malloc(size); b = (float*)malloc(size); c = (float*)malloc(size);
        // Initialise a and b vectors
        for (i = 0; i < N; i++) {
                a[i] = (float)i; b[i] = (float)(2*i);
        }

        Compute_C(a, b, c, N);

        // Check the result
        for (i = 0; i < N; i++) {
                error = error + c[i] - sqrtf(a[i]*a[i] + b[i]*b[i]);
        }
        printf("Error = %g\n", error);

        free(a); free(b); free(c);
        printf("Computation Complete\n");
        return 0;
}


void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

Simplest Possible Beginnings

The makefile here:



```
all:
	g++ main.cpp -c -O3
	g++ main.o -O3 -o test.run
```

We might use –O3 to get the fastest possible performance. This level of optimization should include vectorization. Perhaps we better check:

Disassemble the object to view the assembly-like code: objdump –d main.o > outputfile.txt

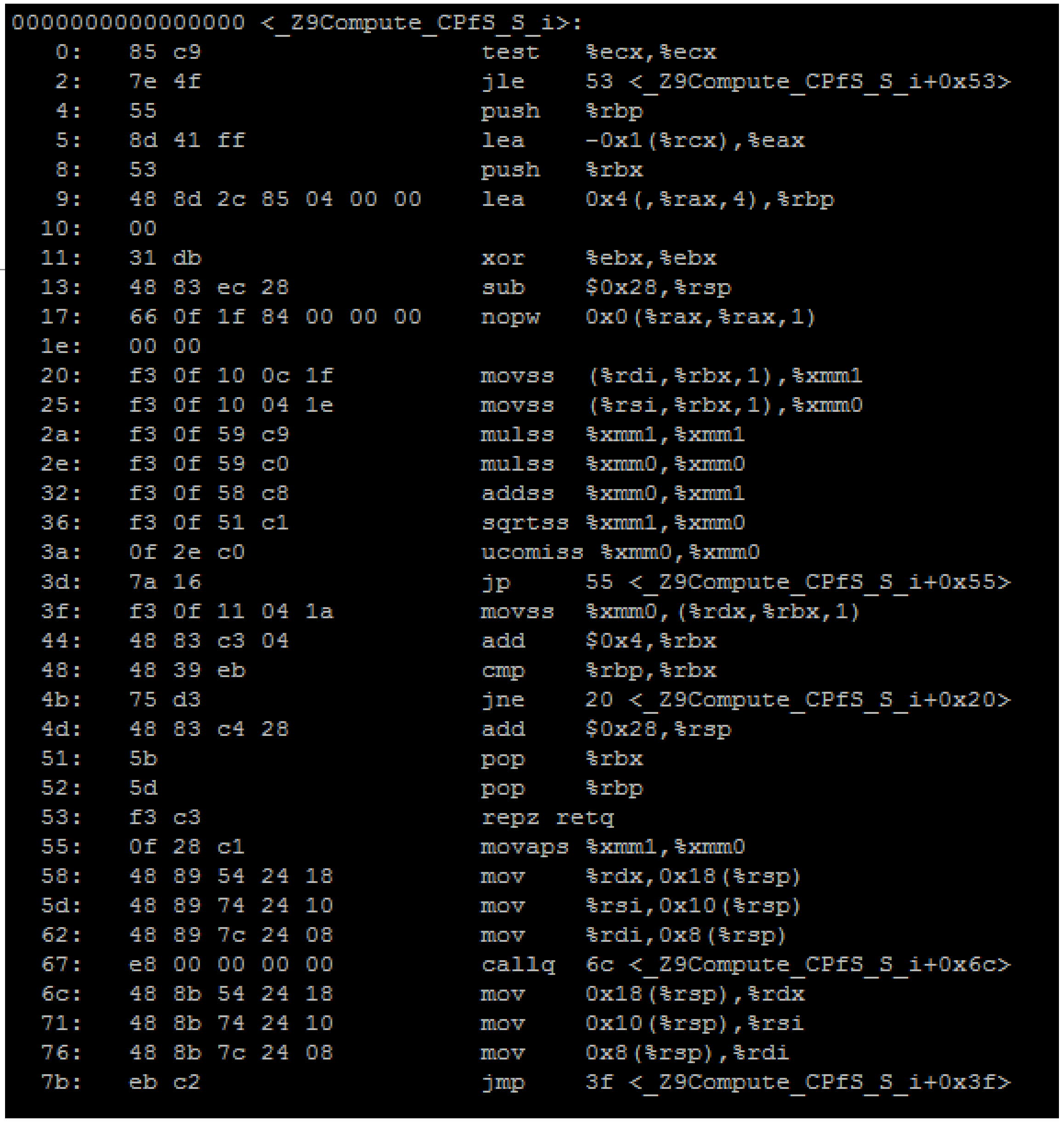
Simplest Possible Beginnings

What exactly is going on?

- The computation is happening in the XMM1 and XMM0 registers.
- These computations are sequential: how do we know?

  mulss, adds, sqrtss → Sequential operations.

Immediately we can conclude that this code is **not** making the most out of the hardware.



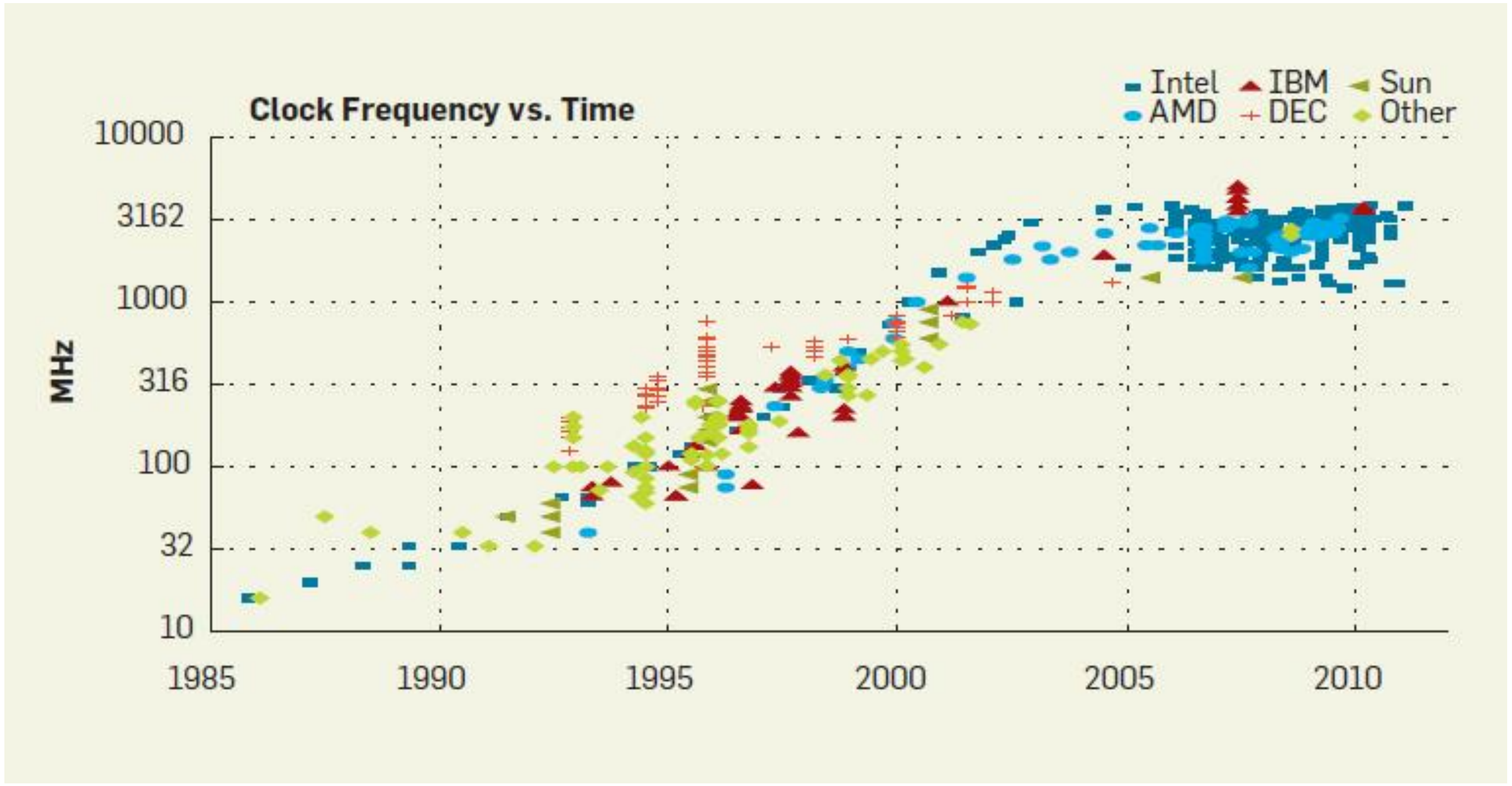
```
0000000000000000 <_Z9Compute_CPfS_S_i>:
   0:   85 c9                   test   %ecx,%ecx
   2:   7e 4f                   jle    53 <_Z9Compute_CPfS_S_i+0x53>
   4:   55                      push   %rbp
   5:   8d 41 ff                lea    -0x1(%rcx),%eax
   8:   53                      push   %rbx
   9:   48 8d 2c 85 04 00 00    lea    0x4(,%rax,4),%rbp
  10:   00
  11:   31 db                   xor    %ebx,%ebx
  13:   48 83 ec 28             sub    $0x28,%rsp
  17:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  1e:   00 00
  20:   f3 0f 10 0c 1f          movss  (%rdi,%rbx,1),%xmm1
  25:   f3 0f 10 04 1e          movss  (%rsi,%rbx,1),%xmm0
  2a:   f3 0f 59 c9             mulss  %xmm1,%xmm1
  2e:   f3 0f 59 c0             mulss  %xmm0,%xmm0
  32:   f3 0f 58 c8             addss  %xmm0,%xmm1
  36:   f3 0f 51 c1             sqrtss %xmm1,%xmm0
  3a:   0f 2e c0                ucomiss %xmm0,%xmm0
  3d:   7a 16                   jp     55 <_Z9Compute_CPfS_S_i+0x55>
  3f:   f3 0f 11 04 1a          movss  %xmm0,(%rdx,%rbx,1)
  44:   48 83 c3 04             add    $0x4,%rbx
  48:   48 39 eb                cmp    %rbp,%rbx
  4b:   75 d3                   jne    20 <_Z9Compute_CPfS_S_i+0x20>
  4d:   48 83 c4 28             add    $0x28,%rsp
  51:   5b                      pop    %rbx
  52:   5d                      pop    %rbp
  53:   f3 c3                   repz retq
  55:   0f 28 c1                movaps %xmm1,%xmm0
  58:   48 89 54 24 18          mov    %rdx,0x18(%rsp)
  5d:   48 89 74 24 10          mov    %rsi,0x10(%rsp)
  62:   48 89 7c 24 08          mov    %rdi,0x8(%rsp)
  67:   e8 00 00 00 00          callq  6c <_Z9Compute_CPfS_S_i+0x6c>
  6c:   48 8b 54 24 18          mov    0x18(%rsp),%rdx
  71:   48 8b 74 24 10          mov    0x10(%rsp),%rsi
  76:   48 8b 7c 24 08          mov    0x8(%rsp),%rdi
  7b:   eb c2                   jmp    3f <_Z9Compute_CPfS_S_i+0x3f>
```

SIMD REGISTERS

SIMD REGISTERS

Computer technology has evolved rapidly over a short period of time.

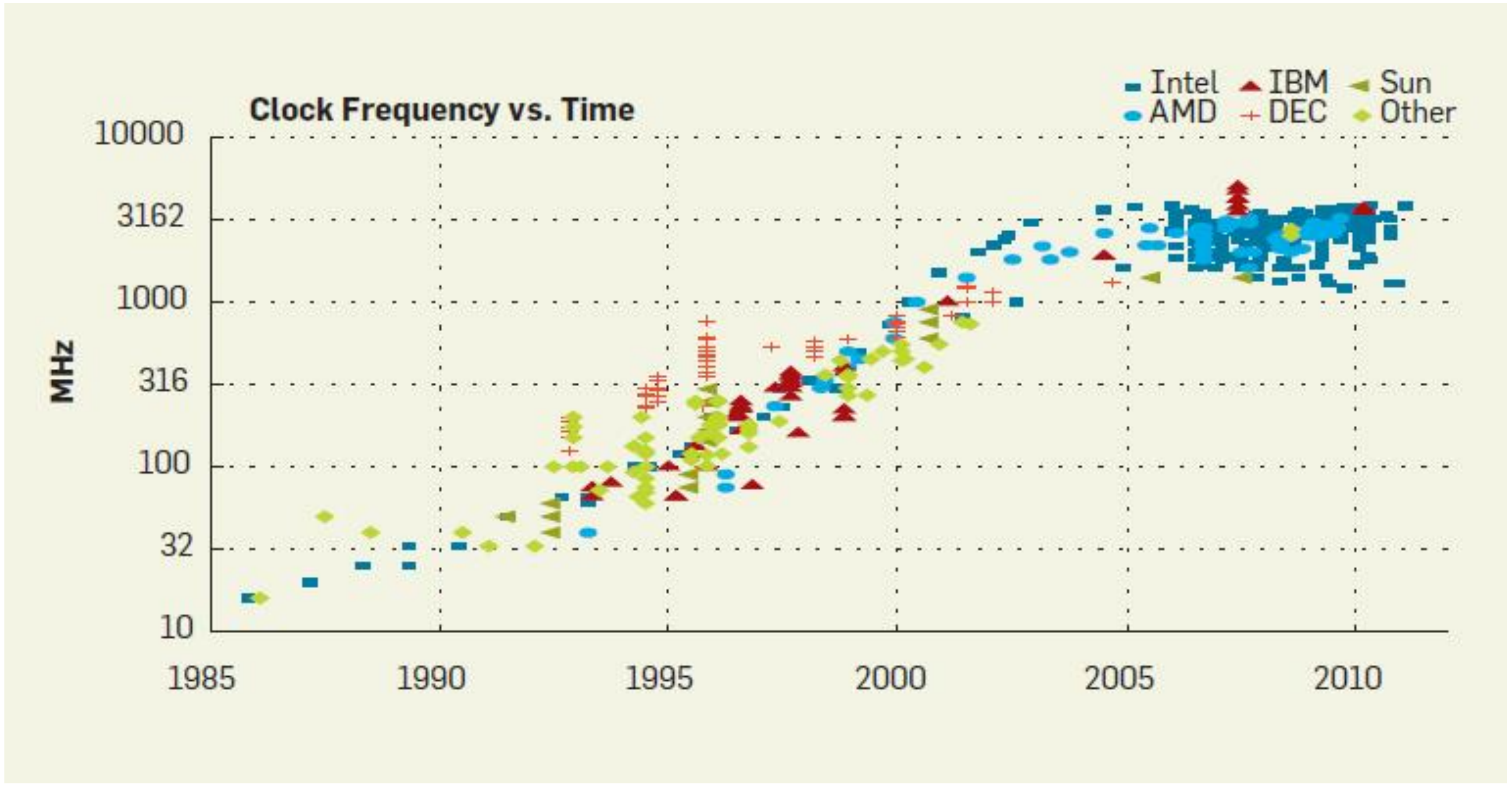
The complexity of the applications, driven by consumers, has also grown rapidly.



SIMD REGISTERS

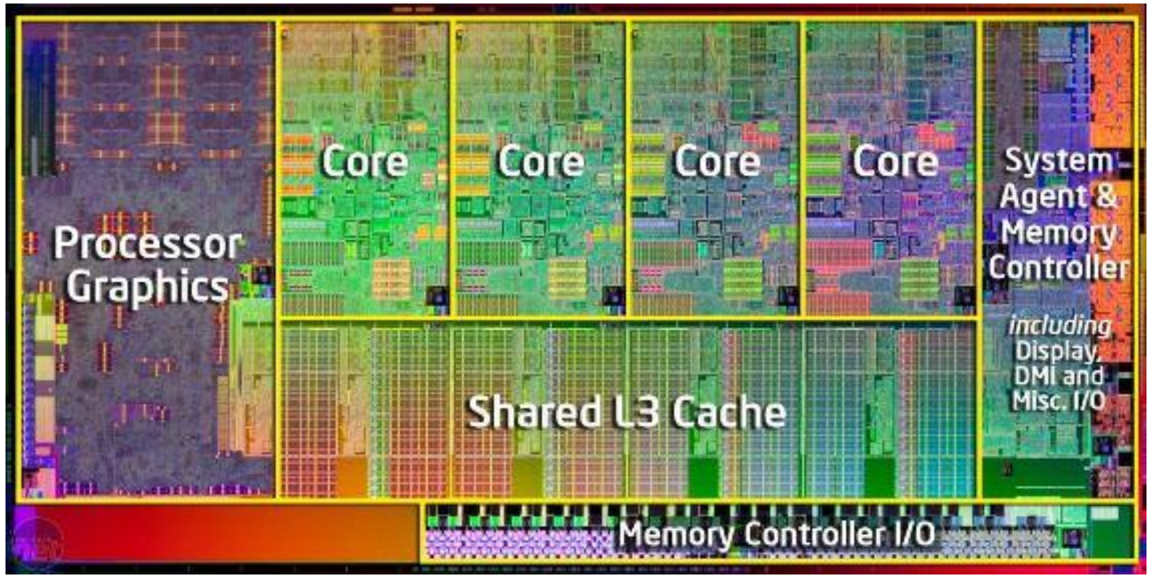
However, in recent years, the clock speed has stagnated – very high frequencies lead to high temperatures.

So what have CPU manufacturers been doing to increase CPU performance?

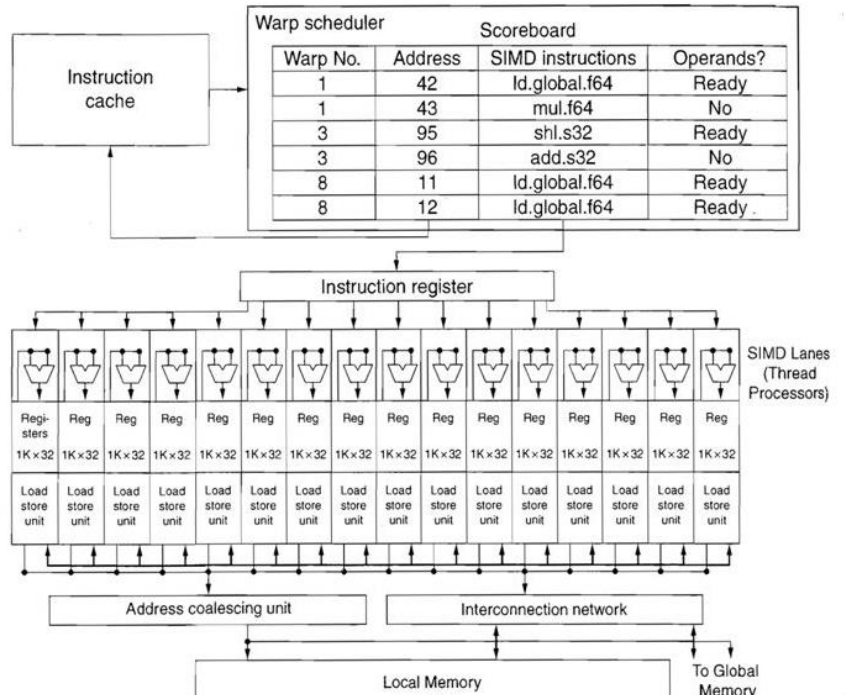


SIMD REGISTERS

Their approach has been multi-pronged:



Each CPU contains numerous cores..

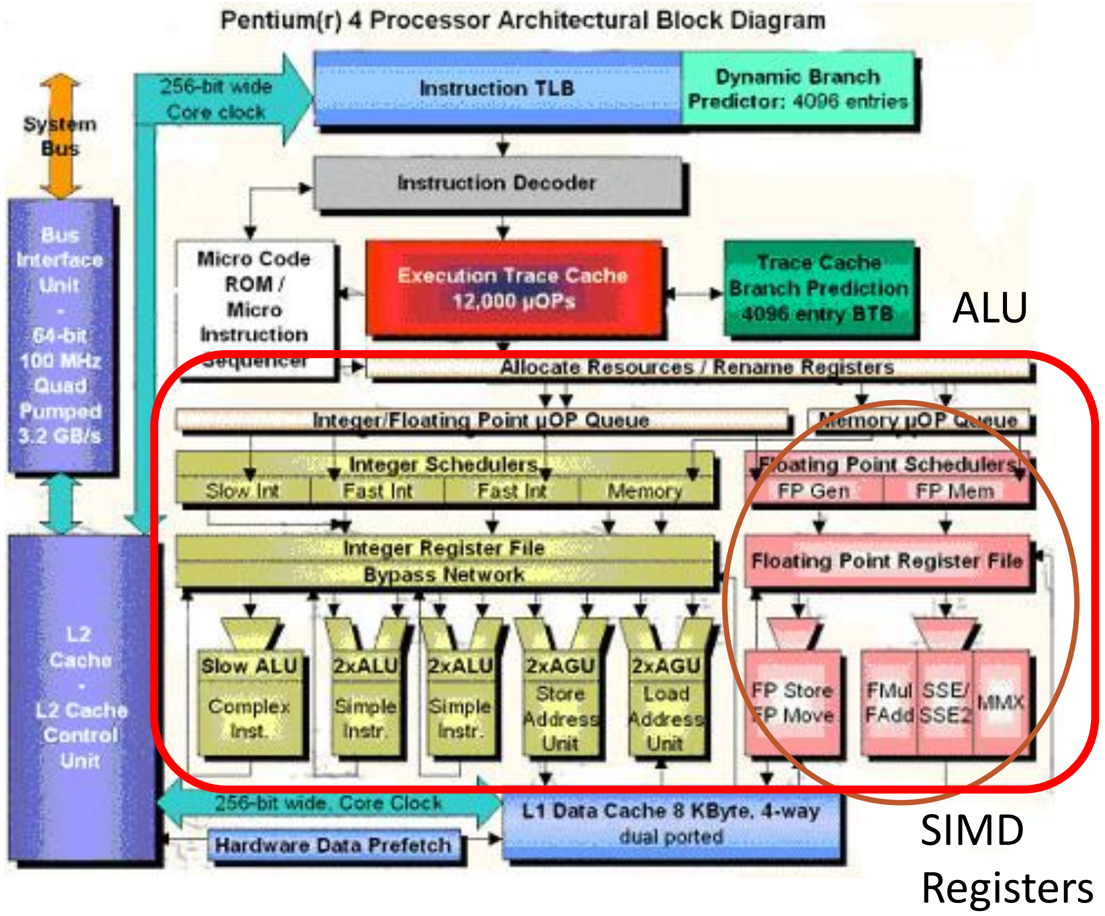


Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

..and each core contains SIMD registers.

MMX

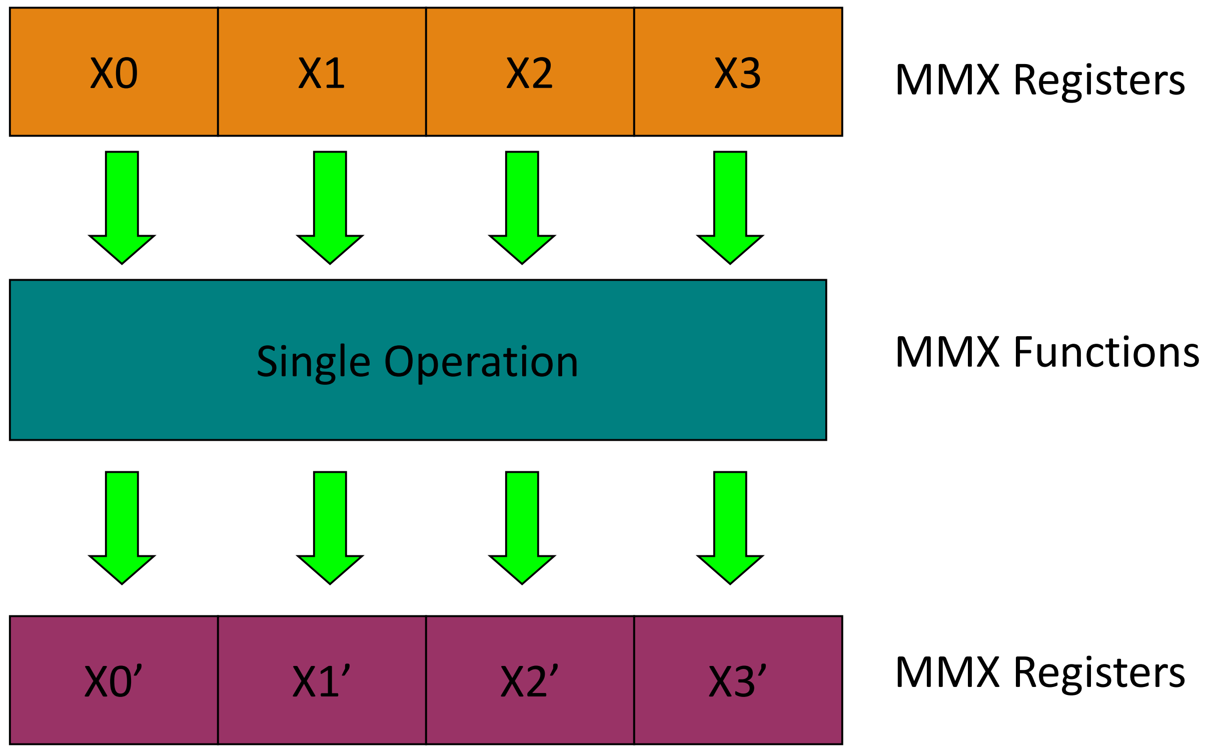
The first SIMD registers were introduced as by Intel in 1997.


Pentium(r) 4 Processor Architectural Block Diagram
System Bus
256-bit wide Core clock
Instruction TLB
Dynamic Branch Predictor: 4096 entries
Instruction Decoder
Bus Interface Unit - 64-bit 100 MHz Quad Pumped 3.2 GB/s
Micro Code ROM / Micro Instruction Sequencer
Execution Trace Cache 12,000 µOPs
Trace Cache Branch Prediction 4096 entry BTB
ALU
Allocate Resources / Rename Registers
Integer/Floating Point µOP Queue
Memory µOP Queue
Integer Schedulers
Slow Int
Fast Int
Fast Int
Memory
Floating Point Schedulers
FP Gen
FP Mem
Integer Register File Bypass Network
Floating Point Register File
L2 Cache - L2 Cache Control Unit
Slow ALU Complex Inst.
2xALU Simple Instr.
2xALU Simple Instr.
2xAGU Store Address Unit
2xAGU Load Address Unit
FP Store FP Move
FMul FAdd
SSE/ SSE2
MMX
256-bit wide, Core Clock
L1 Data Cache 8 KByte, 4-way dual ported
Hardware Data Prefetch
SIMD Registers

MMX Concept

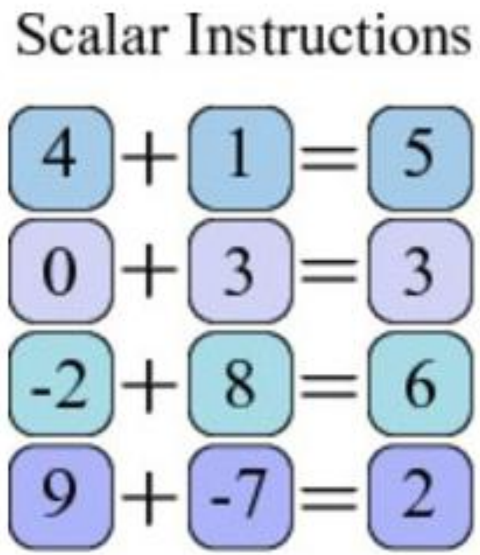
The concept – that data would be packed together into a single MMX type.

A single operation on an MMX type would correspond to that operation being performed on the data packed within in.

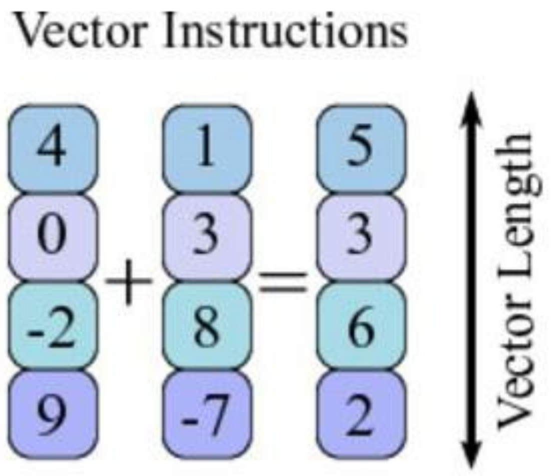


MMX Concept

Explained in another way:



4 separate operations (additions) on integers



1 single operation (addition) performed on packed data.

MMX Packed Vector Length = 64 bits.

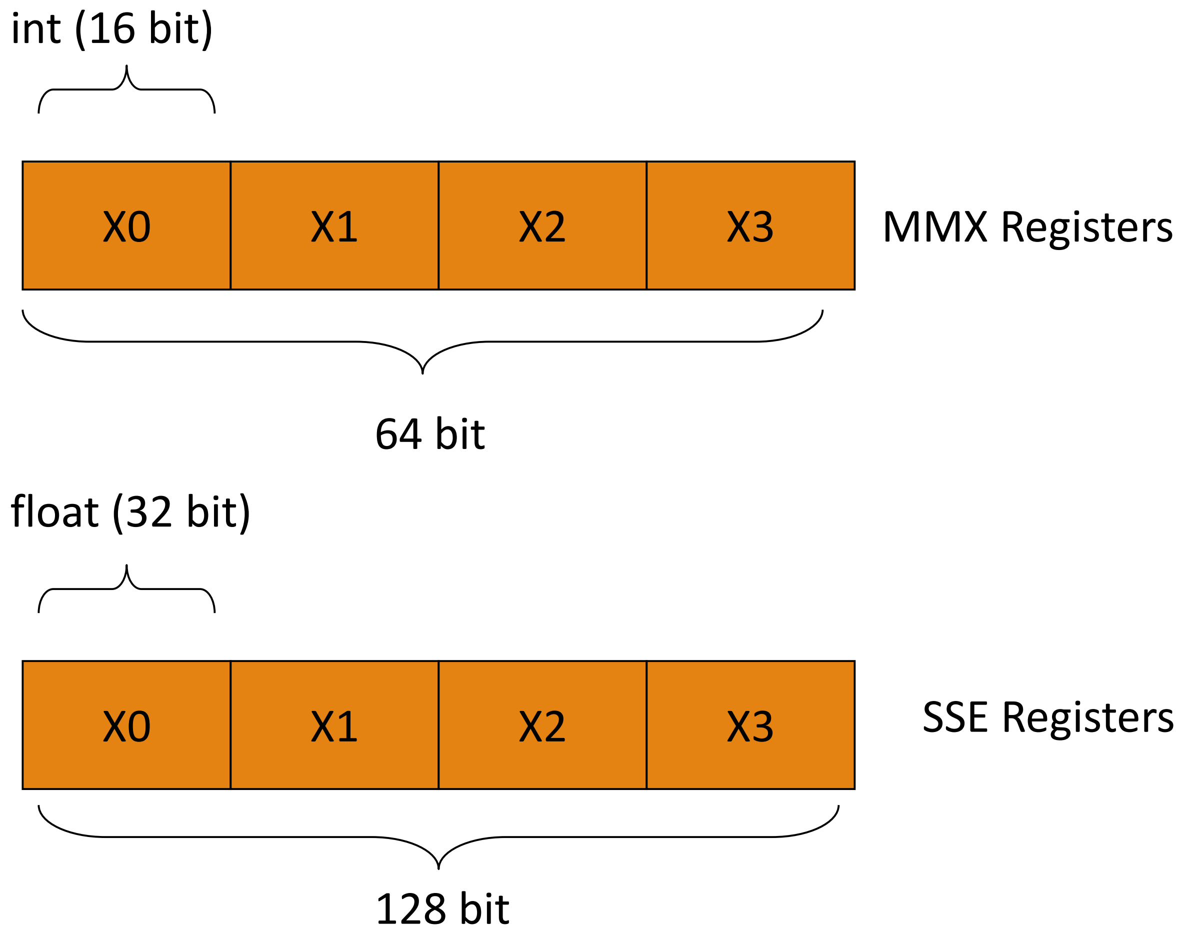
From MMX to SSE

These were very successful – however, 64 bits was too small, and MMX was only useful for integer computations.

These parallel registers were upgraded to:

- Hold more data (128 bits), and
- Perform floating point operations.

and SSE was born.

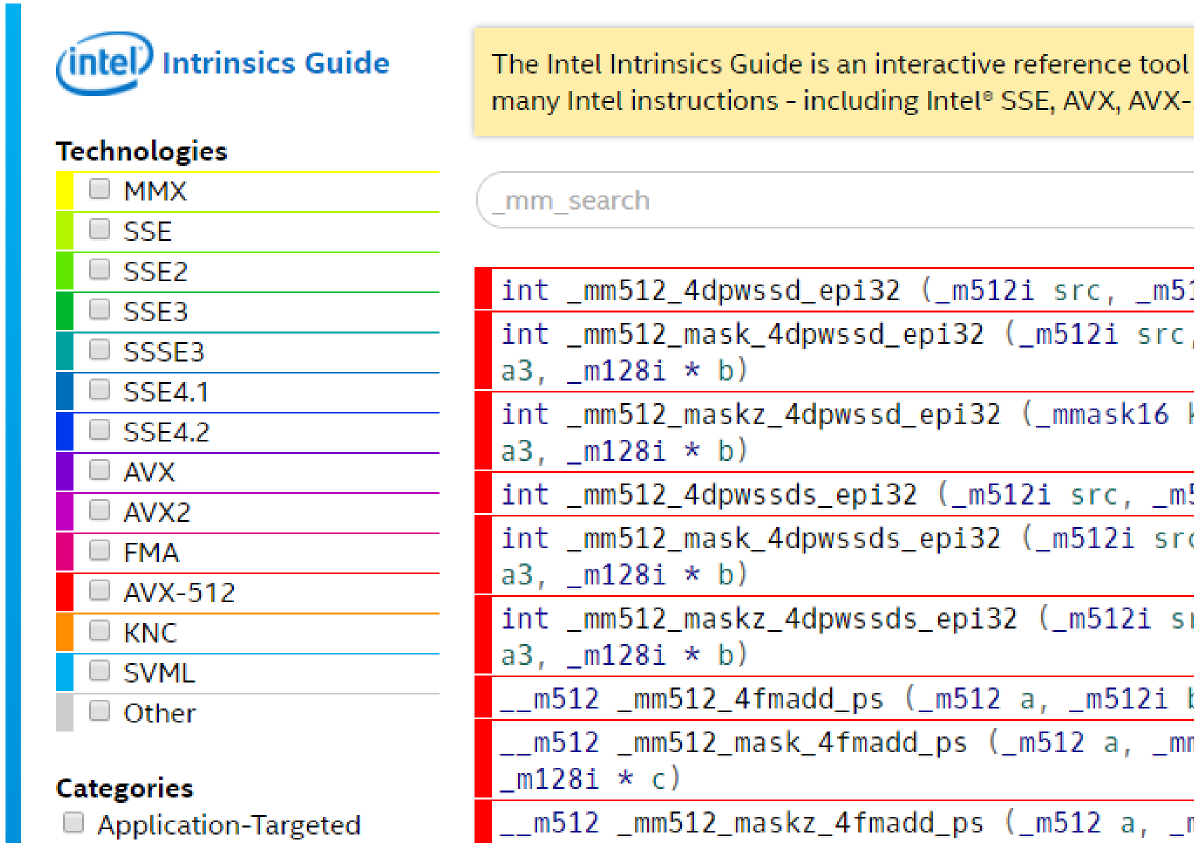


SSE Intrinsic Functions

SSE has been incredibly successful – so successful, unfortunately, that most compilers default to it automatically, even nowadays.

We can get information on the SSE functions from the Intel Intrinsics Guide.

SSE has since been superseded by AVX.



The Intel Intrinsics Guide is an interactive reference tool
many Intel instructions - including Intel® SSE, AVX, AVX-

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

_mm_search

Categories

- ☐ Application-Targeted

```
int _mm512_4dpwssd_epi32 (_m512i src, _m5
int _mm512_mask_4dpwssd_epi32 (_m512i src,
a3, _m128i * b)
int _mm512_maskz_4dpwssd_epi32 (_mmask16 k
a3, _m128i * b)
int _mm512_4dpwssds_epi32 (_m512i src, _m5
int _mm512_mask_4dpwssds_epi32 (_m512i src
a3, _m128i * b)
int _mm512_maskz_4dpwssds_epi32 (_m512i s
a3, _m128i * b)
__m512 _mm512_4fmadd_ps (_m512 a, _m512i b
__m512 _mm512_mask_4fmadd_ps (_m512 a, _mm
_m128i * c)
__m512 _mm512_maskz_4fmadd_ps (_m512 a, _m
```

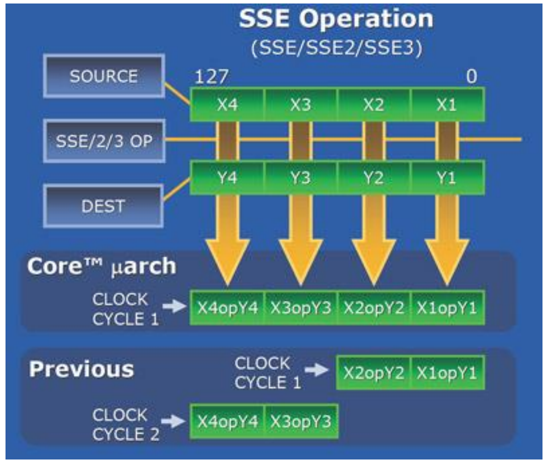
Advanced Vector Extensions

AVX stands for Advanced Vector eXtensions.

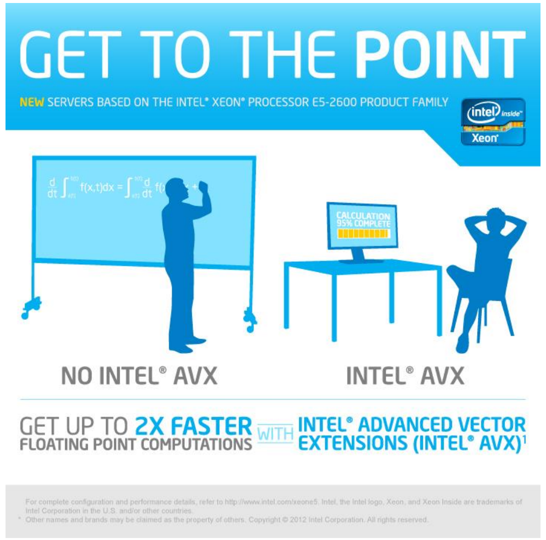
It is basically an extension of the SSE instruction set – the concept is the same.

However, AVX registers contain 256 bits of data.

MMX - 64 bit wide registers for parallel computing on integer types.



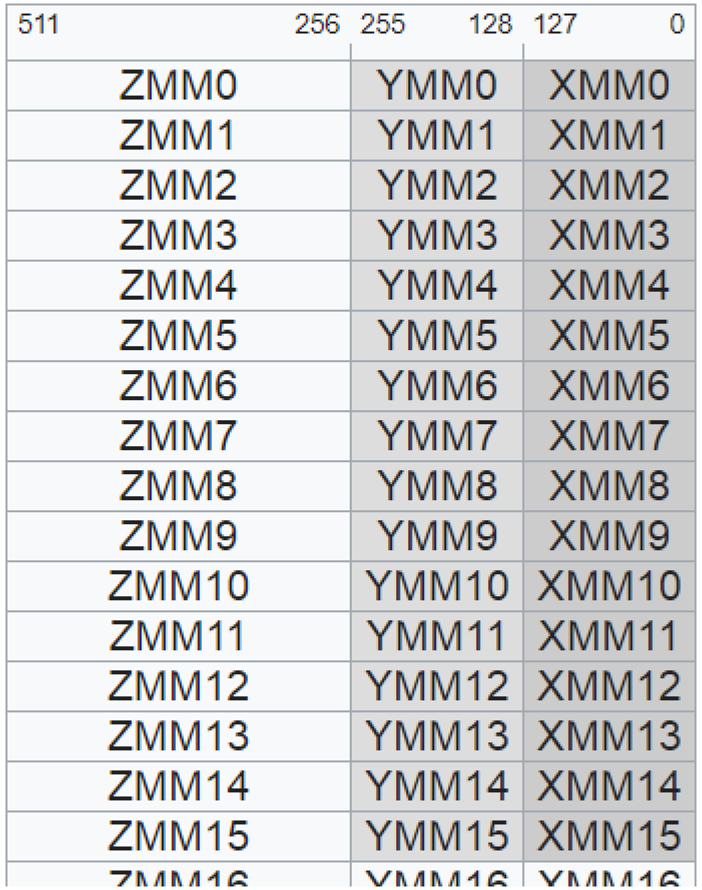
SSE - 128 bit wide registers for parallel computing on integers, floats and (later) doubles.


GET TO THE POINT
NEW SERVERS BASED ON THE INTEL® XEON® PROCESSOR E5-2600 PRODUCT FAMILY
intel inside Xeon
CALCULATION 95% COMPLETE
NO INTEL® AVX
INTEL® AVX
GET UP TO 2X FASTER FLOATING POINT COMPUTATIONS WITH INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)¹

AVX - 256 bit wide registers for parallel computing on integers, floats and doubles.

Quick Guide

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

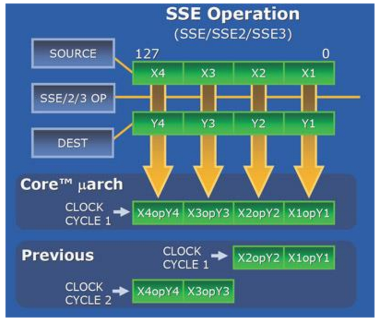


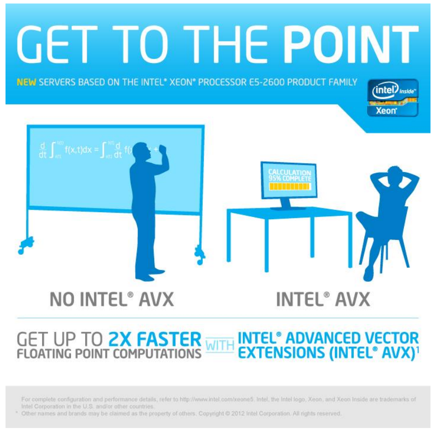
511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16

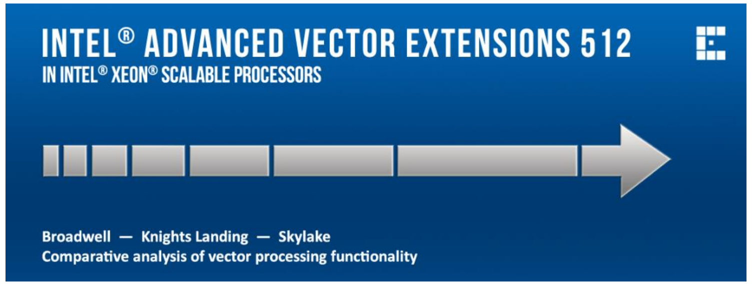
XMM

YMM

ZMM




GET TO THE POINT
NEW SERVERS BASED ON THE INTEL® XEON® PROCESSOR E5-2600 PRODUCT FAMILY
NO INTEL® AVX
INTEL® AVX
GET UP TO 2X FASTER FLOATING POINT COMPUTATIONS WITH INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)¹


INTEL® ADVANCED VECTOR EXTENSIONS 512
IN INTEL® XEON® SCALABLE PROCESSORS
Broadwell — Knights Landing — Skylake
Comparative analysis of vector processing functionality

SSE - 128 bit wide registers for parallel computing on integers, floats and (later) doubles.

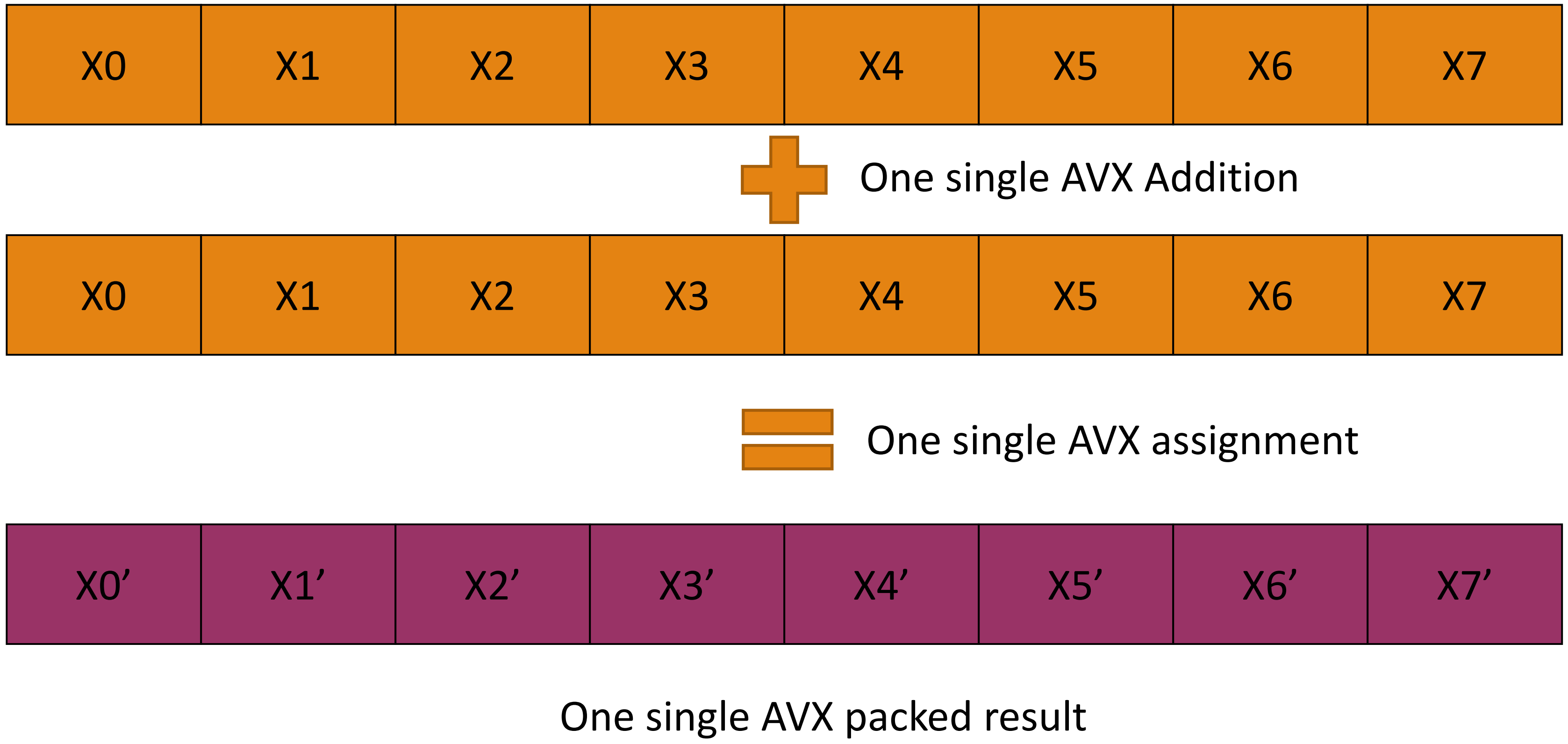
AVX - 256 bit wide registers for parallel computing on integers, floats and doubles.

AVX 512 - 512 bit wide registers for parallel computing on integers, floats and doubles.

Advanced Vector Extensions

The additional space in the registers means that, instead of performing simultaneous operations across **4 floats**, we can now do the same for **8 floats**.

This means that we should see a vast improvement in performance!



Getting Started

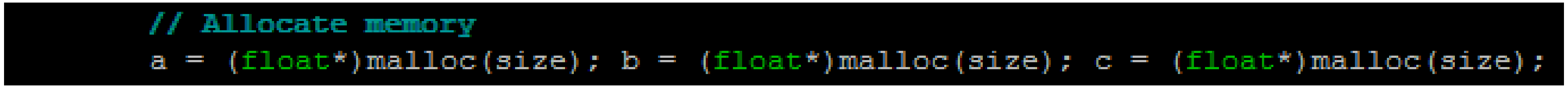
Several things need to happen before we can use AVX to accelerate our computations:

- We need to find a better way to allocate memory.
- We need to pack our arrays of floats into arrays of AVX types.
- We need to use AVX intrinsic functions for our calculation.

Piece of cake. Let’s deal with memory allocation first.

Memory Allocation

The code we started with used malloc:



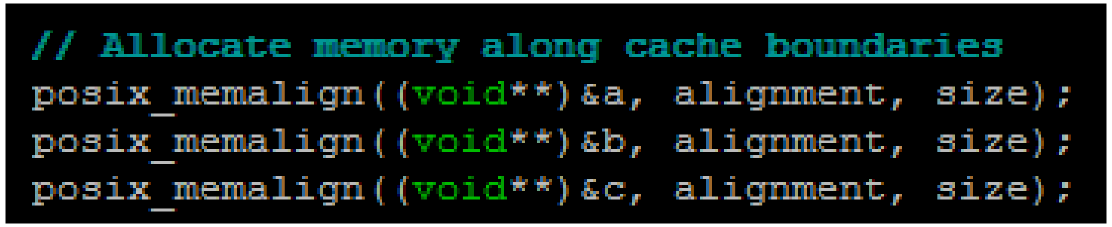
```
        // Allocate memory
        a = (float*)malloc(size); b = (float*)malloc(size); c = (float*)malloc(size);
```

Malloc will align our data in memory based on the type of data being allocated.

However, if we want to pack our data into larger types, we need align our data along cache boundaries based on the larger type.

Memory Allocation

Being old-school, I use posix_memalign for this:



```
// Allocate memory along cache boundaries
posix_memalign((void**)&a, alignment, size);
posix_memalign((void**)&b, alignment, size);
posix_memalign((void**)&c, alignment, size);
```

where size_t size = 32*sizeof(float) (32 byte alignment) – this will be OK for AVX, but we might need to increase it again if the vector length changes. (HINT)

NOTE: posix_memalign is found inside <stdlib.h>, and the variables are freed in the same way [free(a) etc].

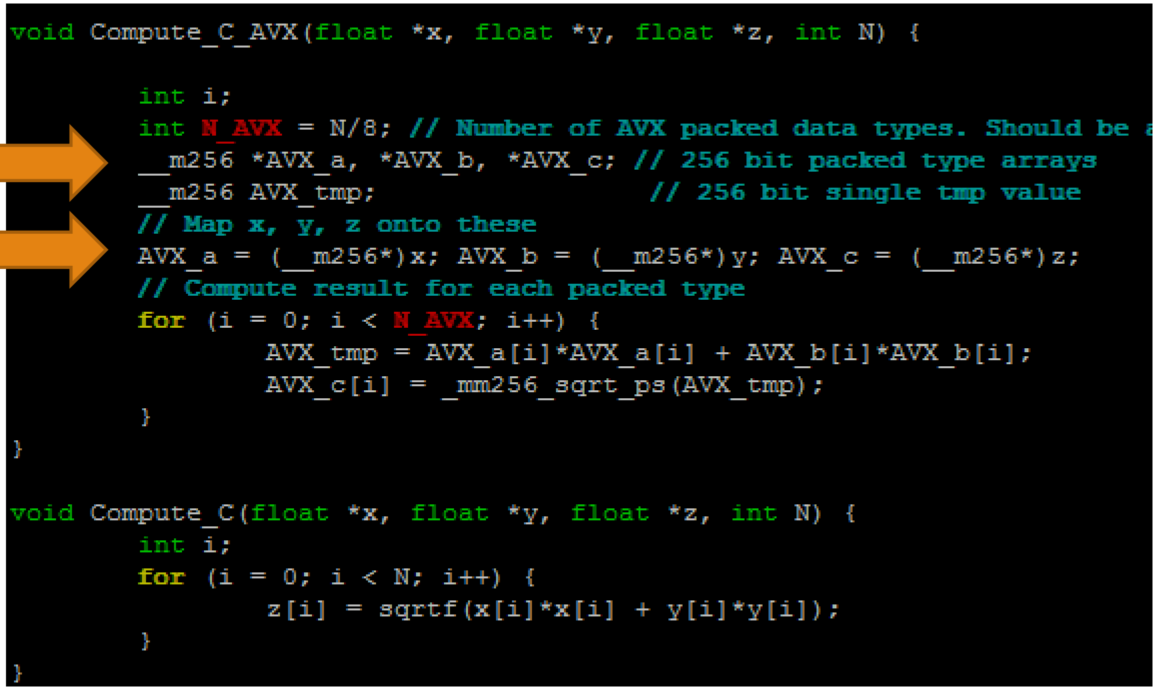
AVX Packing and Computation

Let’s create a new function – Compute_C_AVX() – to perform the same job as our previous code.

First things first, we need to pack our floats into 256 bit wide AVX types – the fastest way to do this is to perform a direct type conversion / casting using an AVX pointer:

__m256 *AVX_a;   // Pointer to AVX type

AVX_a = (__m256*)x;   // Type conversion (casting)



```
void Compute_C_AVX(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/8; // Number of AVX packed data types. Should be a
        __m256 *AVX_a, *AVX_b, *AVX_c; // 256 bit packed type arrays
        __m256 AVX_tmp;                // 256 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m256*)x; AVX_b = (__m256*)y; AVX_c = (__m256*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);
        }
}

void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

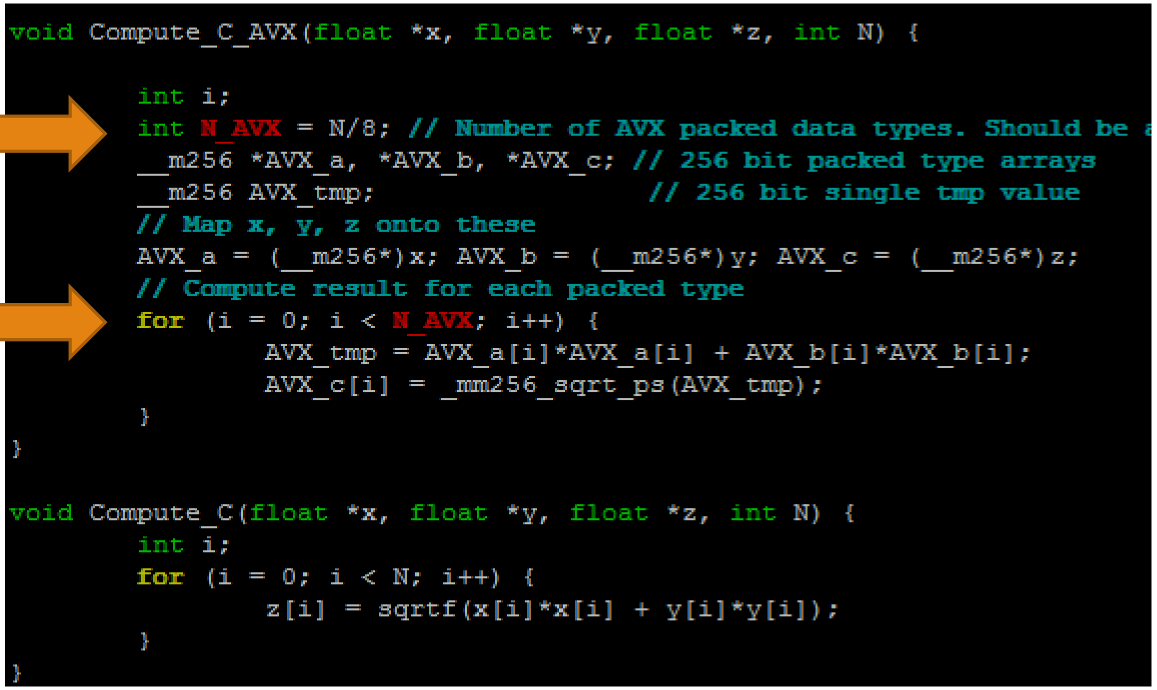
AVX Packing and Computation

Each time we perform a computation on an AVX type, we are performing a computation on 8 packed floats.

Since our previous problem size was 1024, this means we will need to iterate over 1024/8 = 128 AVX types.

Our for loop needs to reflect this:

```
N_AVX = N/8;

for (i = 0; i < N_AVX; i++) {
      <stuff>
}
```



```
void Compute_C_AVX(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/8; // Number of AVX packed data types. Should be a
        __m256 *AVX_a, *AVX_b, *AVX_c; // 256 bit packed type arrays
        __m256 AVX_tmp;                 // 256 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m256*)x; AVX_b = (__m256*)y; AVX_c = (__m256*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);
        }
}

void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

AVX Packing and Computation

Finally, the computation.

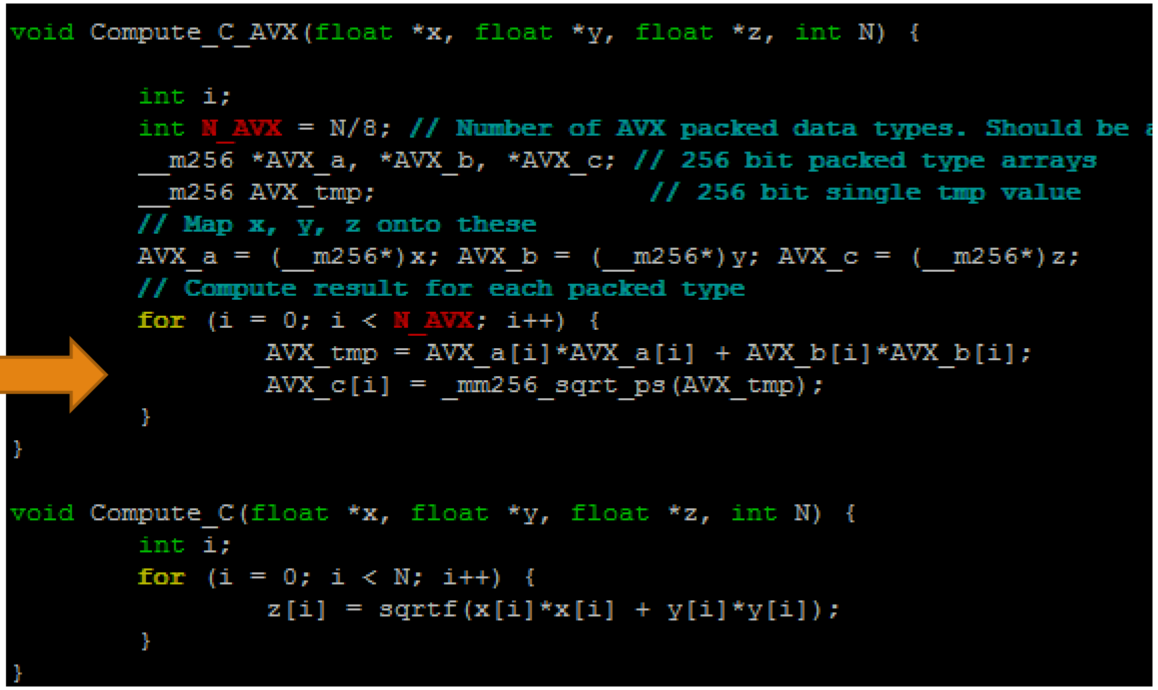
The squaring and addition is straightforward:

AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];

AVX_tmp contains 8 packed floats now. We cannot use the ordinary sqrt function – we need a function to use on an AVX type:

AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);

_mm256_sqrt_ps is a vector intrinsic function



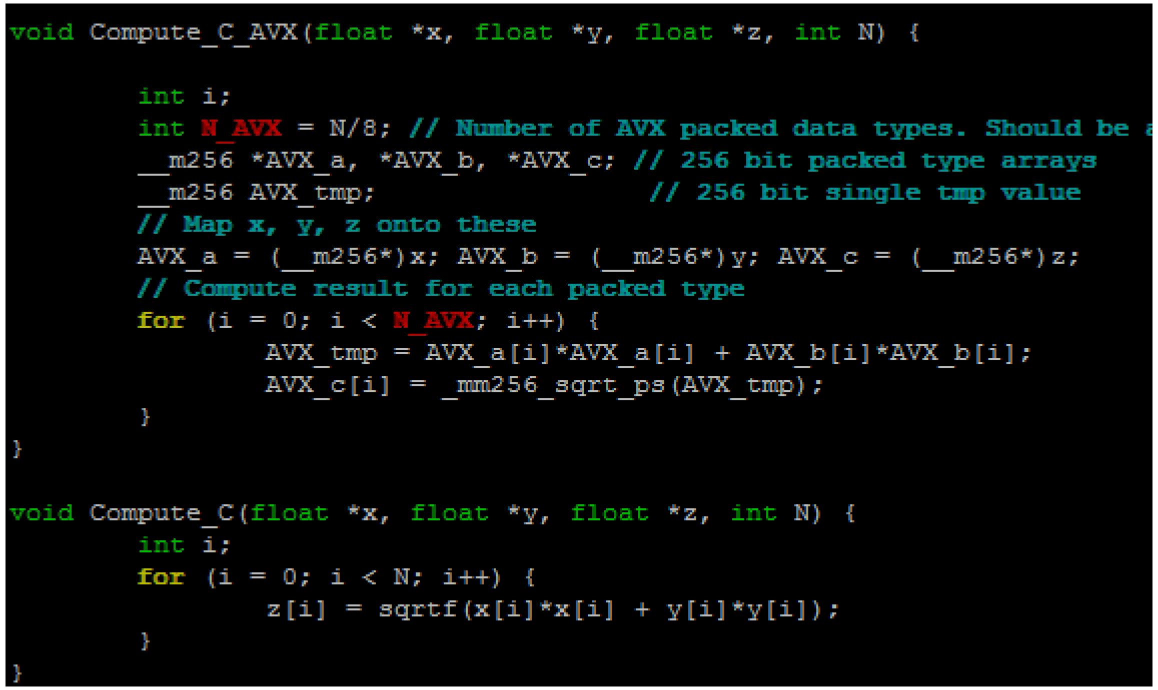
```
void Compute_C_AVX(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/8; // Number of AVX packed data types. Should be a
        __m256 *AVX_a, *AVX_b, *AVX_c; // 256 bit packed type arrays
        __m256 AVX_tmp;                // 256 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m256*)x; AVX_b = (__m256*)y; AVX_c = (__m256*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);
        }
}

void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

AVX Packing and Computation

Quick review of our new function using AVX types and intrinsic functions:

- Our loop goes from 1 to N_AVX, performing 1/8th the iterations than normal since we are performing operations on 8 packed floats each iteration.

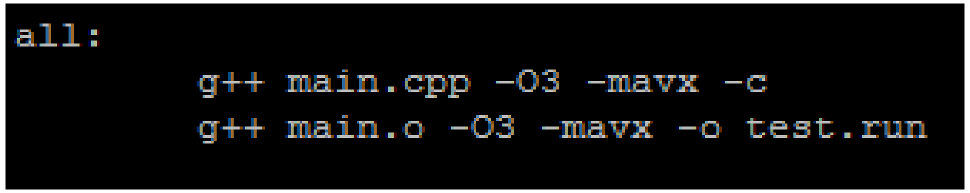
- We use the special vector intrinsic function _mm256_sqrt_ps to compute the sqrt on each packed float in AVX_tmp.



```
void Compute_C_AVX(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/8; // Number of AVX packed data types. Should be
        __m256 *AVX_a, *AVX_b, *AVX_c; // 256 bit packed type arrays
        __m256 AVX_tmp;                // 256 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m256*)x; AVX_b = (__m256*)y; AVX_c = (__m256*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);
        }
}

void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

AVX Packing and Computation

To make this code, we need to modify our makefile by adding a flag:



```
all:
        g++ main.cpp -O3 -mavx -c
        g++ main.o -O3 -mavx -o test.run
```

Let’s rebuild and see what happens.

AVX Code & Result

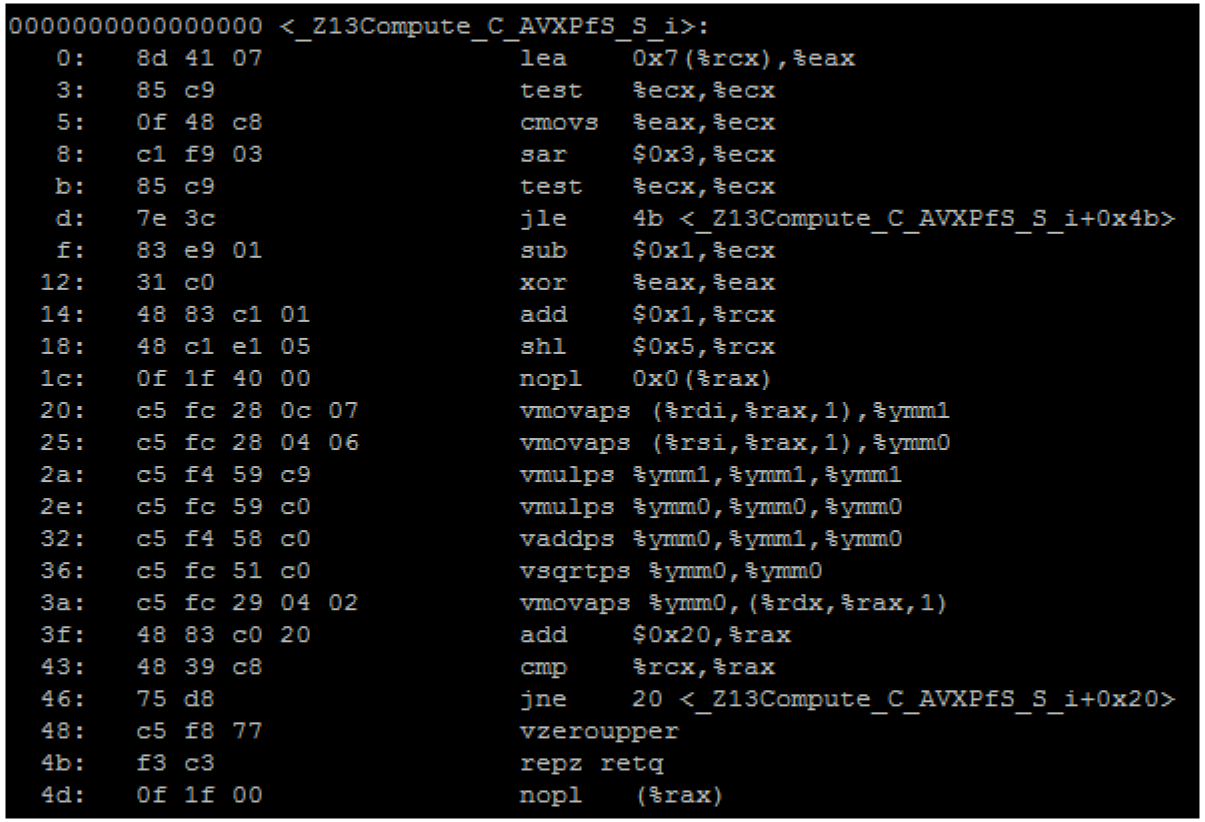
Perform a quick object dump → let's see what the computer is doing now:

- We see the computer is now using the larger (256 bit) YMM0 and YMM1 registers.

- We also see that the instructions we are performing are all parallel instructions:

(vmulps, vaddps, vsqrtps etc)

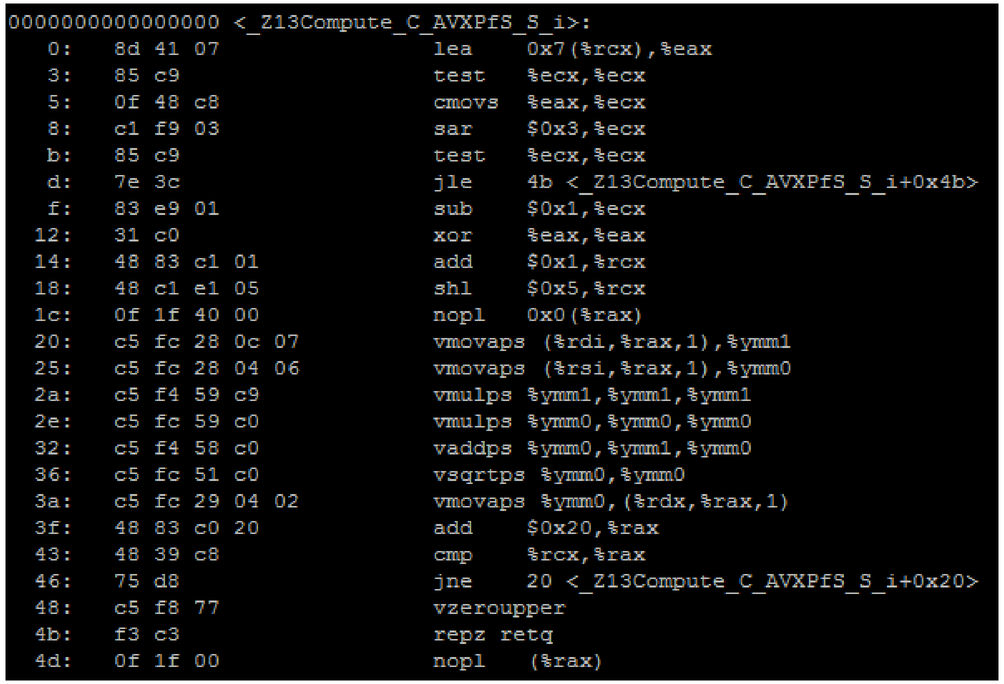
This tells us we are making proper use of AVX vectorization.



```
0000000000000000 <_Z13Compute_C_AVXPfS_S_i>:
   0:   8d 41 07                lea    0x7(%rcx),%eax
   3:   85 c9                   test   %ecx,%ecx
   5:   0f 48 c8                cmovs  %eax,%ecx
   8:   c1 f9 03                sar    $0x3,%ecx
   b:   85 c9                   test   %ecx,%ecx
   d:   7e 3c                   jle    4b <_Z13Compute_C_AVXPfS_S_i+0x4b>
   f:   83 e9 01                sub    $0x1,%ecx
  12:   31 c0                   xor    %eax,%eax
  14:   48 83 c1 01             add    $0x1,%rcx
  18:   48 c1 e1 05             shl    $0x5,%rcx
  1c:   0f 1f 40 00             nopl   0x0(%rax)
  20:   c5 fc 28 0c 07          vmovaps (%rdi,%rax,1),%ymm1
  25:   c5 fc 28 04 06          vmovaps (%rsi,%rax,1),%ymm0
  2a:   c5 f4 59 c9             vmulps %ymm1,%ymm1,%ymm1
  2e:   c5 fc 59 c0             vmulps %ymm0,%ymm0,%ymm0
  32:   c5 f4 58 c0             vaddps %ymm0,%ymm1,%ymm0
  36:   c5 fc 51 c0             vsqrtps %ymm0,%ymm0
  3a:   c5 fc 29 04 02          vmovaps %ymm0,(%rdx,%rax,1)
  3f:   48 83 c0 20             add    $0x20,%rax
  43:   48 39 c8                cmp    %rcx,%rax
  46:   75 d8                   jne    20 <_Z13Compute_C_AVXPfS_S_i+0x20>
  48:   c5 f8 77                vzeroupper
  4b:   f3 c3                   repz retq
  4d:   0f 1f 00                nopl   (%rax)
```

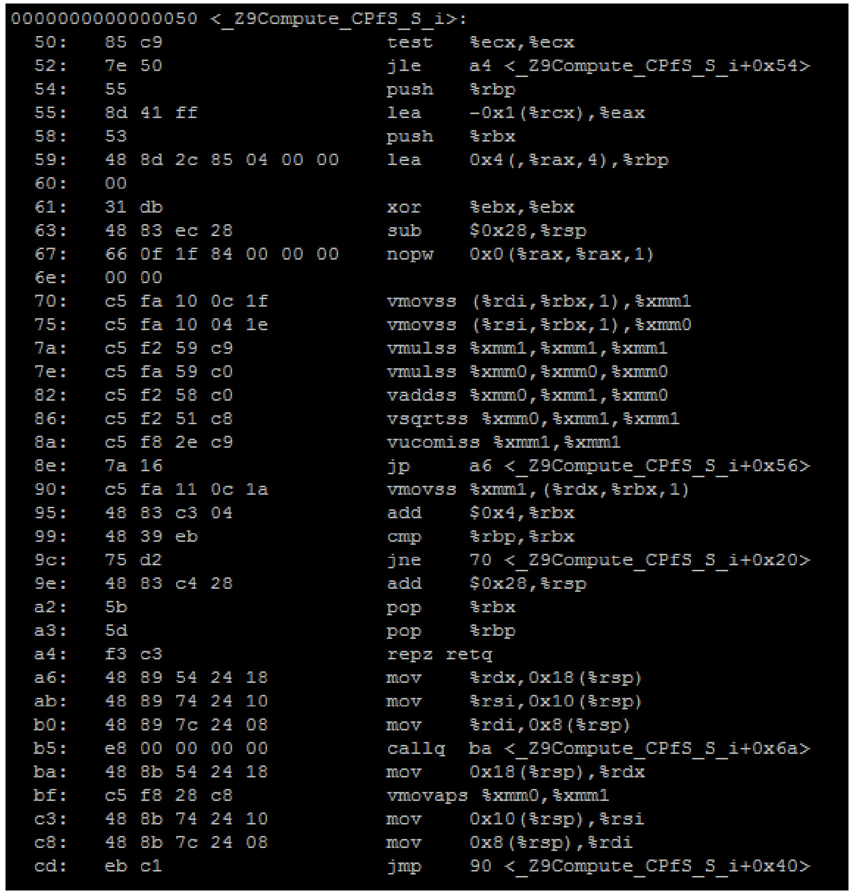
AVX Code & Result

Compare the two assembly-like codes:



```
0000000000000000 <_Z13Compute_C_AVXPfS_S_i>:
   0:   8d 41 07                lea    0x7(%rcx),%eax
   3:   85 c9                   test   %ecx,%ecx
   5:   0f 48 c8                cmovs  %eax,%ecx
   8:   c1 f9 03                sar    $0x3,%ecx
   b:   85 c9                   test   %ecx,%ecx
   d:   7e 3c                   jle    4b <_Z13Compute_C_AVXPfS_S_i+0x4b>
   f:   83 e9 01                sub    $0x1,%ecx
  12:   31 c0                   xor    %eax,%eax
  14:   48 83 c1 01             add    $0x1,%rcx
  18:   48 c1 e1 05             shl    $0x5,%rcx
  1c:   0f 1f 40 00             nopl   0x0(%rax)
  20:   c5 fc 28 0c 07          vmovaps (%rdi,%rax,1),%ymm1
  25:   c5 fc 28 04 06          vmovaps (%rsi,%rax,1),%ymm0
  2a:   c5 f4 59 c9             vmulps %ymm1,%ymm1,%ymm1
  2e:   c5 fc 59 c0             vmulps %ymm0,%ymm0,%ymm0
  32:   c5 f4 58 c0             vaddps %ymm0,%ymm1,%ymm0
  36:   c5 fc 51 c0             vsqrtps %ymm0,%ymm0
  3a:   c5 fc 29 04 02          vmovaps %ymm0,(%rdx,%rax,1)
  3f:   48 83 c0 20             add    $0x20,%rax
  43:   48 39 c8                cmp    %rcx,%rax
  46:   75 d8                   jne    20 <_Z13Compute_C_AVXPfS_S_i+0x20>
  48:   c5 f8 77                vzeroupper
  4b:   f3 c3                   repz retq
  4d:   0f 1f 00                nopl   (%rax)
```

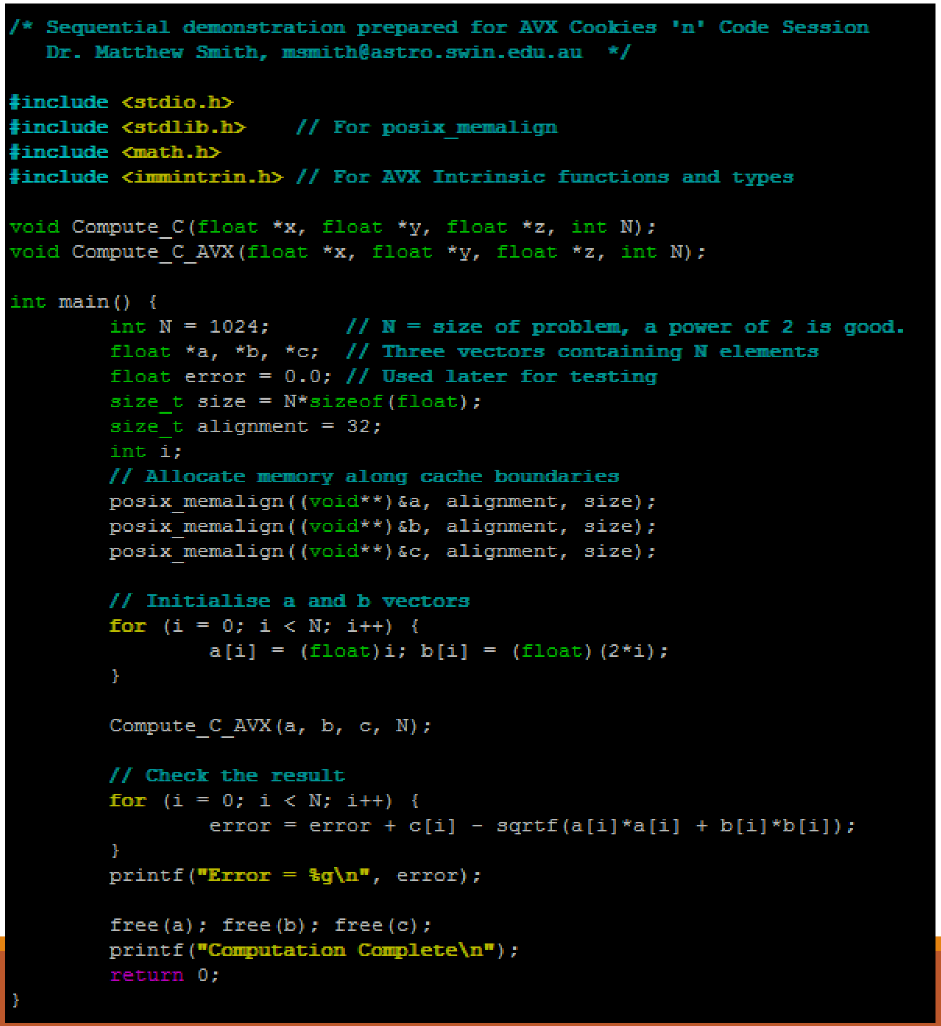
avx function



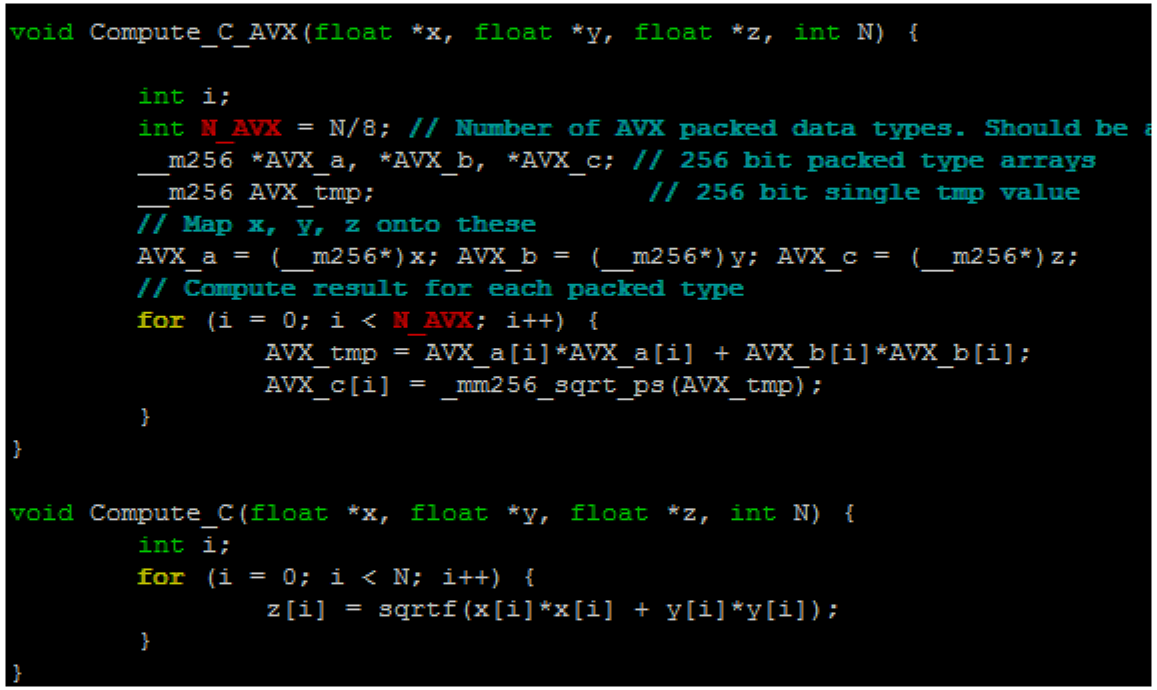
```
0000000000000050 <_Z9Compute_CPfS_S_i>:
  50:   85 c9                   test   %ecx,%ecx
  52:   7e 50                   jle    a4 <_Z9Compute_CPfS_S_i+0x54>
  54:   55                      push   %rbp
  55:   8d 41 ff                lea    -0x1(%rcx),%eax
  58:   53                      push   %rbx
  59:   48 8d 2c 85 04 00 00    lea    0x4(,%rax,4),%rbp
  60:   00
  61:   31 db                   xor    %ebx,%ebx
  63:   48 83 ec 28             sub    $0x28,%rsp
  67:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  6e:   00 00
  70:   c5 fa 10 0c 1f          vmovss (%rdi,%rbx,1),%xmm1
  75:   c5 fa 10 04 1e          vmovss (%rsi,%rbx,1),%xmm0
  7a:   c5 f2 59 c9             vmulss %xmm1,%xmm1,%xmm1
  7e:   c5 fa 59 c0             vmulss %xmm0,%xmm0,%xmm0
  82:   c5 f2 58 c0             vaddss %xmm0,%xmm1,%xmm0
  86:   c5 f2 51 c8             vsqrtss %xmm0,%xmm1,%xmm1
  8a:   c5 f8 2e c9             vucomiss %xmm1,%xmm1
  8e:   7a 16                   jp     a6 <_Z9Compute_CPfS_S_i+0x56>
  90:   c5 fa 11 0c 1a          vmovss %xmm1,(%rdx,%rbx,1)
  95:   48 83 c3 04             add    $0x4,%rbx
  99:   48 39 eb                cmp    %rbp,%rbx
  9c:   75 d2                   jne    70 <_Z9Compute_CPfS_S_i+0x20>
  9e:   48 83 c4 28             add    $0x28,%rsp
  a2:   5b                      pop    %rbx
  a3:   5d                      pop    %rbp
  a4:   f3 c3                   repz retq
  a6:   48 89 54 24 18          mov    %rdx,0x18(%rsp)
  ab:   48 89 74 24 10          mov    %rsi,0x10(%rsp)
  b0:   48 89 7c 24 08          mov    %rdi,0x8(%rsp)
  b5:   e8 00 00 00 00          callq  ba <_Z9Compute_CPfS_S_i+0x6a>
  ba:   48 8b 54 24 18          mov    0x18(%rsp),%rdx
  bf:   c5 f8 28 c8             vmovaps %xmm0,%xmm1
  c3:   48 8b 74 24 10          mov    0x10(%rsp),%rsi
  c8:   48 8b 7c 24 08          mov    0x8(%rsp),%rdi
  cd:   eb c1                   jmp    90 <_Z9Compute_CPfS_S_i+0x40>
```

normal function

Whole Code



```c
/* Sequential demonstration prepared for AVX Cookies 'n' Code Session
   Dr. Matthew Smith, msmith@astro.swin.edu.au  */

#include <stdio.h>
#include <stdlib.h>    // For posix_memalign
#include <math.h>
#include <immintrin.h> // For AVX Intrinsic functions and types

void Compute_C(float *x, float *y, float *z, int N);
void Compute_C_AVX(float *x, float *y, float *z, int N);

int main() {
        int N = 1024;      // N = size of problem, a power of 2 is good.
        float *a, *b, *c;  // Three vectors containing N elements
        float error = 0.0; // Used later for testing
        size_t size = N*sizeof(float);
        size_t alignment = 32;
        int i;
        // Allocate memory along cache boundaries
        posix_memalign((void**)&a, alignment, size);
        posix_memalign((void**)&b, alignment, size);
        posix_memalign((void**)&c, alignment, size);

        // Initialise a and b vectors
        for (i = 0; i < N; i++) {
                a[i] = (float)i; b[i] = (float)(2*i);
        }

        Compute_C_AVX(a, b, c, N);

        // Check the result
        for (i = 0; i < N; i++) {
                error = error + c[i] - sqrtf(a[i]*a[i] + b[i]*b[i]);
        }
        printf("Error = %g\n", error);

        free(a); free(b); free(c);
        printf("Computation Complete\n");
        return 0;
}
```



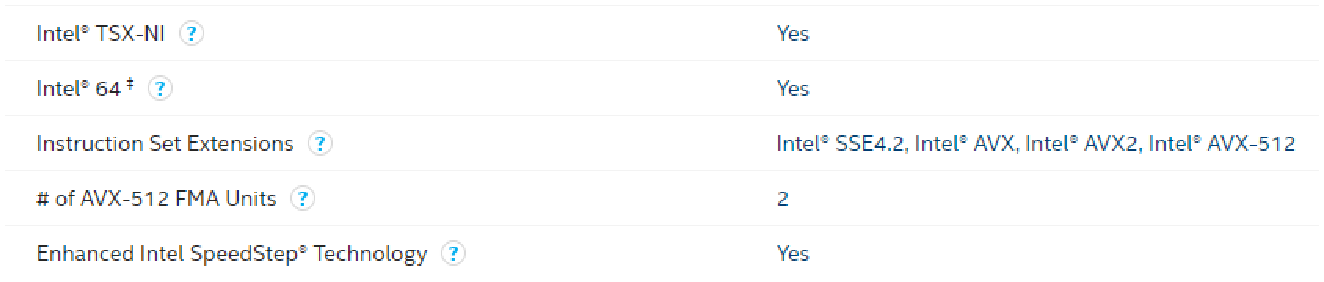
```c
void Compute_C_AVX(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/8; // Number of AVX packed data types. Should be
        __m256 *AVX_a, *AVX_b, *AVX_c; // 256 bit packed type arrays
        __m256 AVX_tmp;                 // 256 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m256*)x; AVX_b = (__m256*)y; AVX_c = (__m256*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm256_sqrt_ps(AVX_tmp);
        }
}

void Compute_C(float *x, float *y, float *z, int N) {
        int i;
        for (i = 0; i < N; i++) {
                z[i] = sqrtf(x[i]*x[i] + y[i]*y[i]);
        }
}
```

AVX512

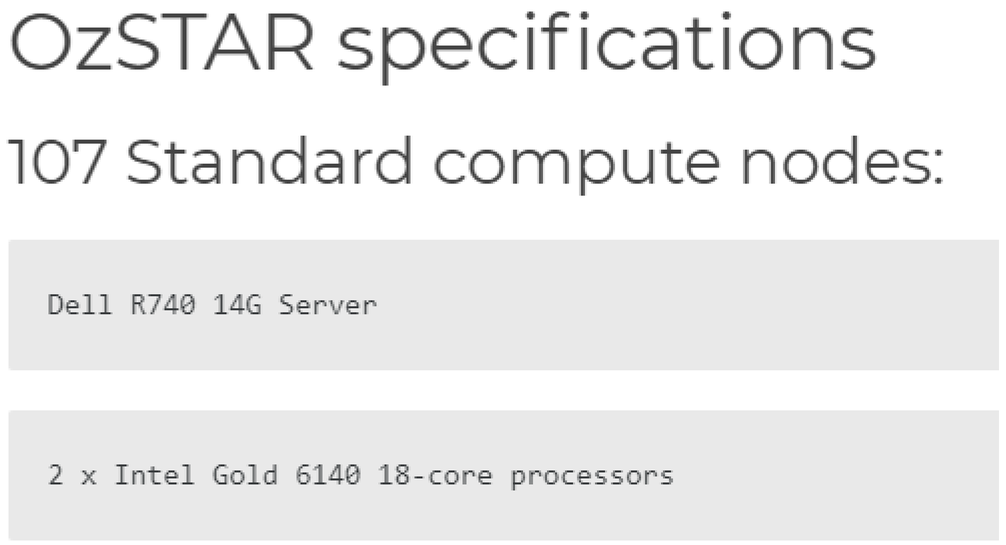
AVX512 on Ozstar

In Ozstar we are fortunate enough to be using a skylake architecture.

This means that the Intel Gold 6140 supports AVX512 – meaning the registers inside its ALU have a length of 512 bits!



Intel® TSX-NI ?	Yes
Intel® 64 ‡ ?	Yes
Instruction Set Extensions ?	Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX-512
# of AVX-512 FMA Units ?	2
Enhanced Intel SpeedStep® Technology ?	Yes



OzSTAR specifications

107 Standard compute nodes:

```
Dell R740 14G Server
```

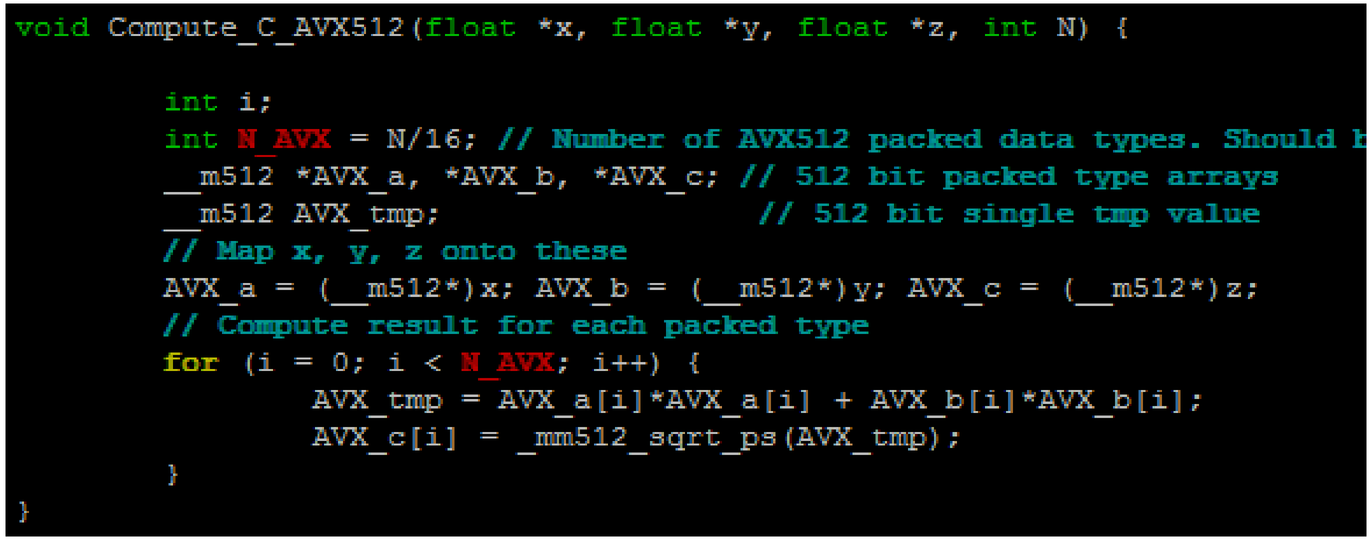
```
2 x Intel Gold 6140 18-core processors
```

AVX-512 Code

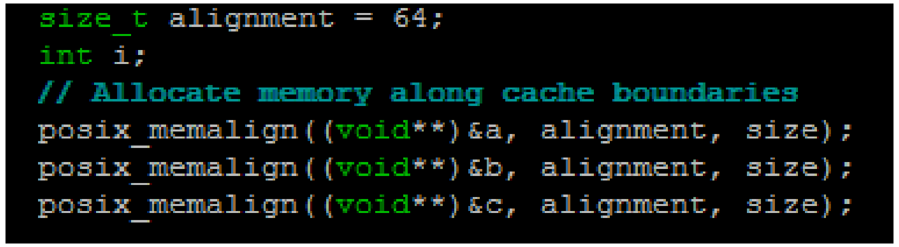
Altering code from AVX to AVX-512 is pretty easy (easier than from SSE to AVX):

- Our N_AVX variable is now N/16 since we pack 16 singles into 512 bits.
- Where we once used 256, we now use 512. Find and replace job, its easy enough.

We also need to make sure our memory is aligned to 64 bytes and **not** 32.



```
void Compute_C_AVX512(float *x, float *y, float *z, int N) {

        int i;
        int N_AVX = N/16; // Number of AVX512 packed data types. Should b
        __m512 *AVX_a, *AVX_b, *AVX_c; // 512 bit packed type arrays
        __m512 AVX_tmp;                // 512 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m512*)x; AVX_b = (__m512*)y; AVX_c = (__m512*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm512_sqrt_ps(AVX_tmp);
        }
}
```



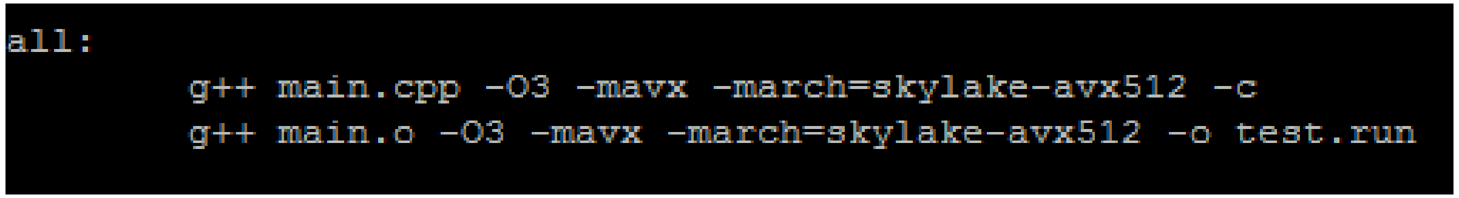
```
size_t alignment = 64;
int i;
// Allocate memory along cache boundaries
posix_memalign((void**)&a, alignment, size);
posix_memalign((void**)&b, alignment, size);
posix_memalign((void**)&c, alignment, size);
```

Requirements

The version of g++ which is loaded by default when you log into Ozstar won’t cut it for AVX512.

You need to load the newer version: **module load gcc/7.3.0**

We also need to update our makefile to let g++ know we wanna use skylake’s AVX 512:



```
all:
	g++ main.cpp -O3 -mavx -march=skylake-avx512 -c
	g++ main.o -O3 -mavx -march=skylake-avx512 -o test.run
```

Let’s rebuild it and see what happens now.

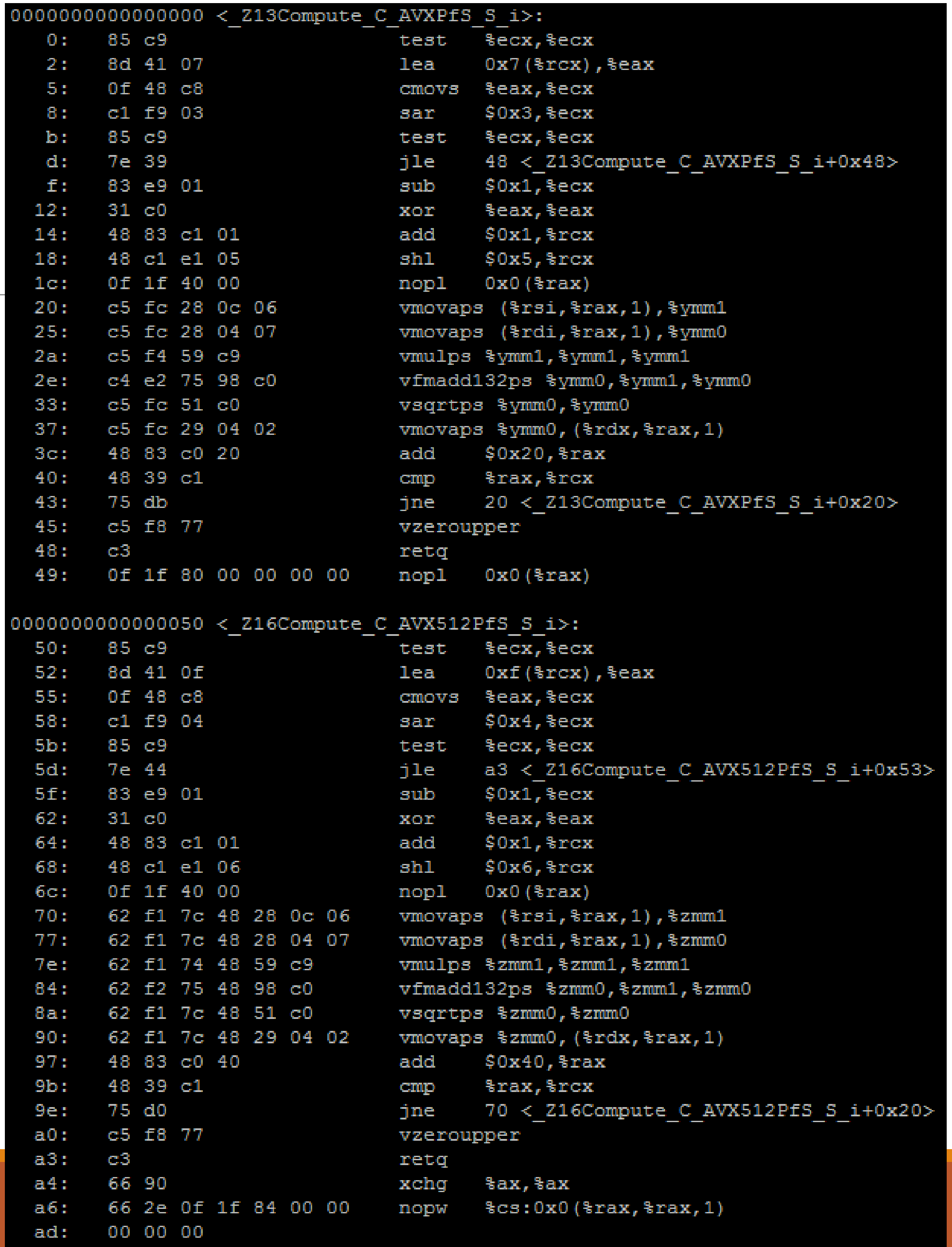
AVX512 Review

Let’s examine the object disassembly:

- The AVX512 code is using the ZMM0 and ZMM1 registers.
- All operations are still being performed in parallel.

Looks like we nailed it.

This code will outperform any attempt at auto-vectorization the compiler will attempt, especially for complex computations (not covered here).



```
0000000000000000 <_Z13Compute_C_AVXPfS_S_i>:
   0:   85 c9                   test   %ecx,%ecx
   2:   8d 41 07                lea    0x7(%rcx),%eax
   5:   0f 48 c8                cmovs  %eax,%ecx
   8:   c1 f9 03                sar    $0x3,%ecx
   b:   85 c9                   test   %ecx,%ecx
   d:   7e 39                   jle    48 <_Z13Compute_C_AVXPfS_S_i+0x48>
   f:   83 e9 01                sub    $0x1,%ecx
  12:   31 c0                   xor    %eax,%eax
  14:   48 83 c1 01             add    $0x1,%rcx
  18:   48 c1 e1 05             shl    $0x5,%rcx
  1c:   0f 1f 40 00             nopl   0x0(%rax)
  20:   c5 fc 28 0c 06          vmovaps (%rsi,%rax,1),%ymm1
  25:   c5 fc 28 04 07          vmovaps (%rdi,%rax,1),%ymm0
  2a:   c5 f4 59 c9             vmulps %ymm1,%ymm1,%ymm1
  2e:   c4 e2 75 98 c0          vfmadd132ps %ymm0,%ymm1,%ymm0
  33:   c5 fc 51 c0             vsqrtps %ymm0,%ymm0
  37:   c5 fc 29 04 02          vmovaps %ymm0,(%rdx,%rax,1)
  3c:   48 83 c0 20             add    $0x20,%rax
  40:   48 39 c1                cmp    %rax,%rcx
  43:   75 db                   jne    20 <_Z13Compute_C_AVXPfS_S_i+0x20>
  45:   c5 f8 77                vzeroupper
  48:   c3                      retq
  49:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000000050 <_Z16Compute_C_AVX512PfS_S_i>:
  50:   85 c9                   test   %ecx,%ecx
  52:   8d 41 0f                lea    0xf(%rcx),%eax
  55:   0f 48 c8                cmovs  %eax,%ecx
  58:   c1 f9 04                sar    $0x4,%ecx
  5b:   85 c9                   test   %ecx,%ecx
  5d:   7e 44                   jle    a3 <_Z16Compute_C_AVX512PfS_S_i+0x53>
  5f:   83 e9 01                sub    $0x1,%ecx
  62:   31 c0                   xor    %eax,%eax
  64:   48 83 c1 01             add    $0x1,%rcx
  68:   48 c1 e1 06             shl    $0x6,%rcx
  6c:   0f 1f 40 00             nopl   0x0(%rax)
  70:   62 f1 7c 48 28 0c 06    vmovaps (%rsi,%rax,1),%zmm1
  77:   62 f1 7c 48 28 04 07    vmovaps (%rdi,%rax,1),%zmm0
  7e:   62 f1 74 48 59 c9       vmulps %zmm1,%zmm1,%zmm1
  84:   62 f2 75 48 98 c0       vfmadd132ps %zmm0,%zmm1,%zmm0
  8a:   62 f1 7c 48 51 c0       vsqrtps %zmm0,%zmm0
  90:   62 f1 7c 48 29 04 02    vmovaps %zmm0,(%rdx,%rax,1)
  97:   48 83 c0 40             add    $0x40,%rax
  9b:   48 39 c1                cmp    %rax,%rcx
  9e:   75 d0                   jne    70 <_Z16Compute_C_AVX512PfS_S_i+0x20>
  a0:   c5 f8 77                vzeroupper
  a3:   c3                      retq
  a4:   66 90                   xchg   %ax,%ax
  a6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  ad:   00 00 00
```

Padding Length

One problem which has loomed over our heads is the requirement that N – our problem size – be evenly divisible by 16 (when using floats).

The solution is to pad the memory length – allocate more memory than you need so that the memory you do have is a multiple of 16 in length.

The computation is quite trivial: $NT = 16\left[(int)\left(\frac{N - 16 + 1}{16}\right)\right]$

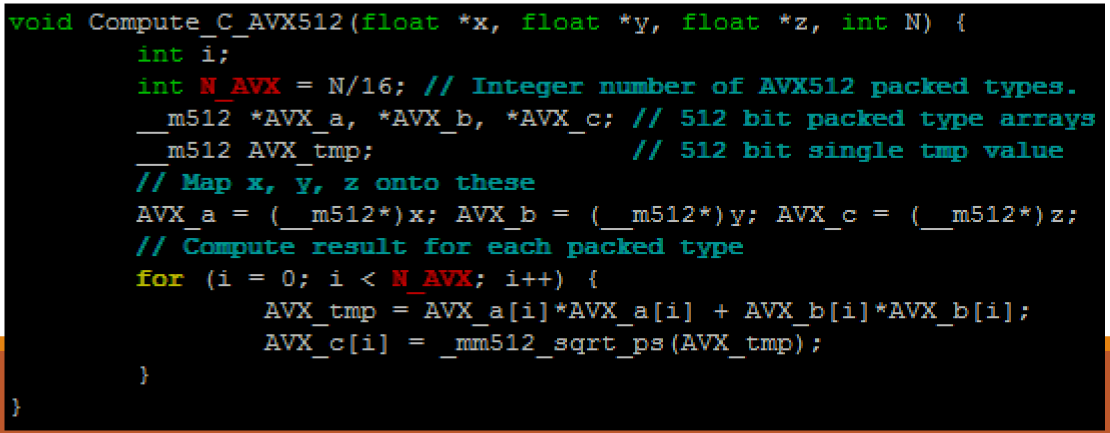
Example: if N = 1025, then NT will be 1040 (1024+16) → giving us 15*4 = 60 wasted bytes, but no segmentation error.

Final Product

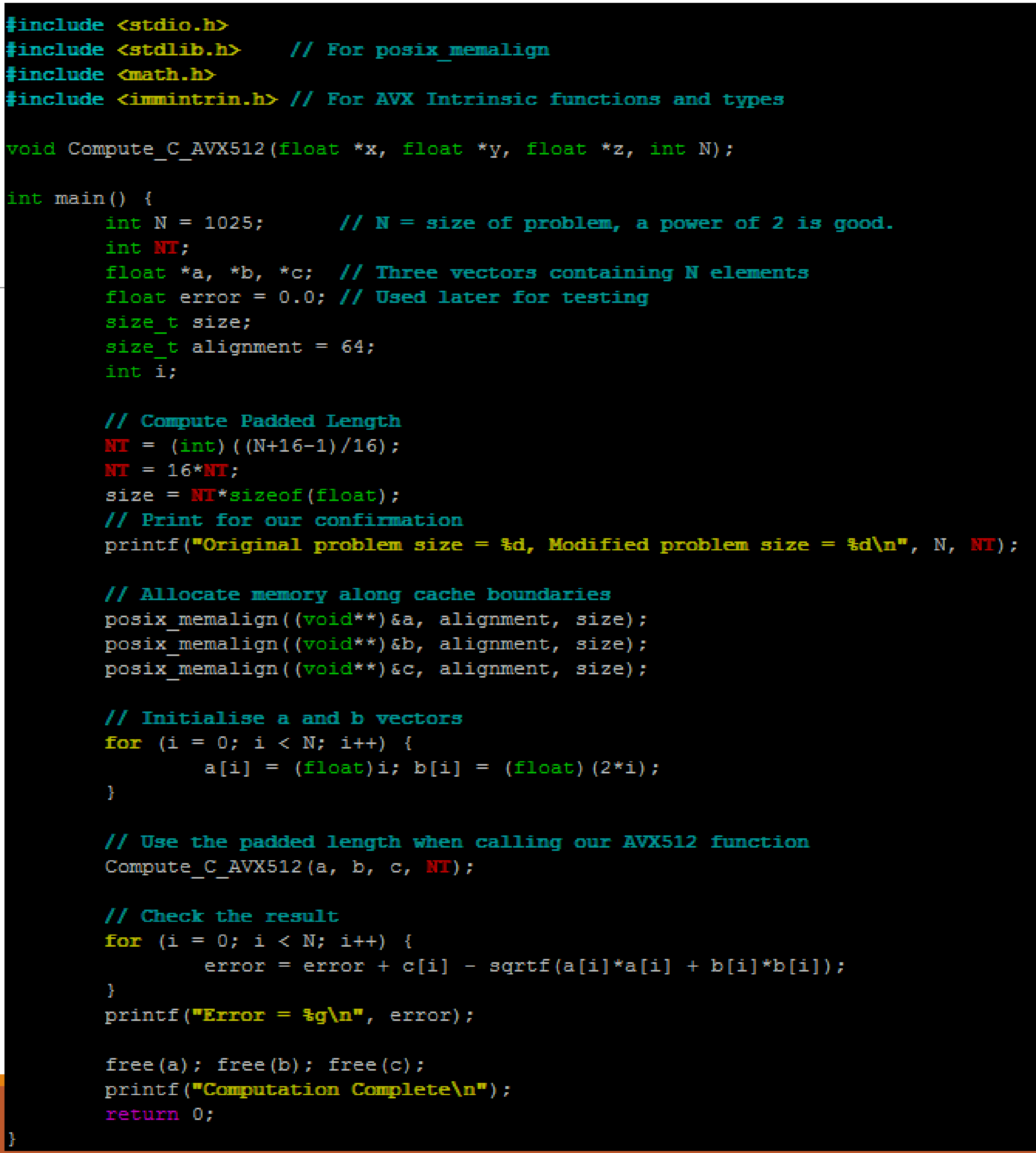
Final code using:

- Cache boundary aligned memory
- Padded memory up to multiple of 16
- AVX512 packed types
- AVX512 Intrinsic Functions

This code will run much faster than the code we started out with.



```c
void Compute_C_AVX512(float *x, float *y, float *z, int N) {
        int i;
        int N_AVX = N/16; // Integer number of AVX512 packed types.
        __m512 *AVX_a, *AVX_b, *AVX_c; // 512 bit packed type arrays
        __m512 AVX_tmp;                // 512 bit single tmp value
        // Map x, y, z onto these
        AVX_a = (__m512*)x; AVX_b = (__m512*)y; AVX_c = (__m512*)z;
        // Compute result for each packed type
        for (i = 0; i < N_AVX; i++) {
                AVX_tmp = AVX_a[i]*AVX_a[i] + AVX_b[i]*AVX_b[i];
                AVX_c[i] = _mm512_sqrt_ps(AVX_tmp);
        }
}
```



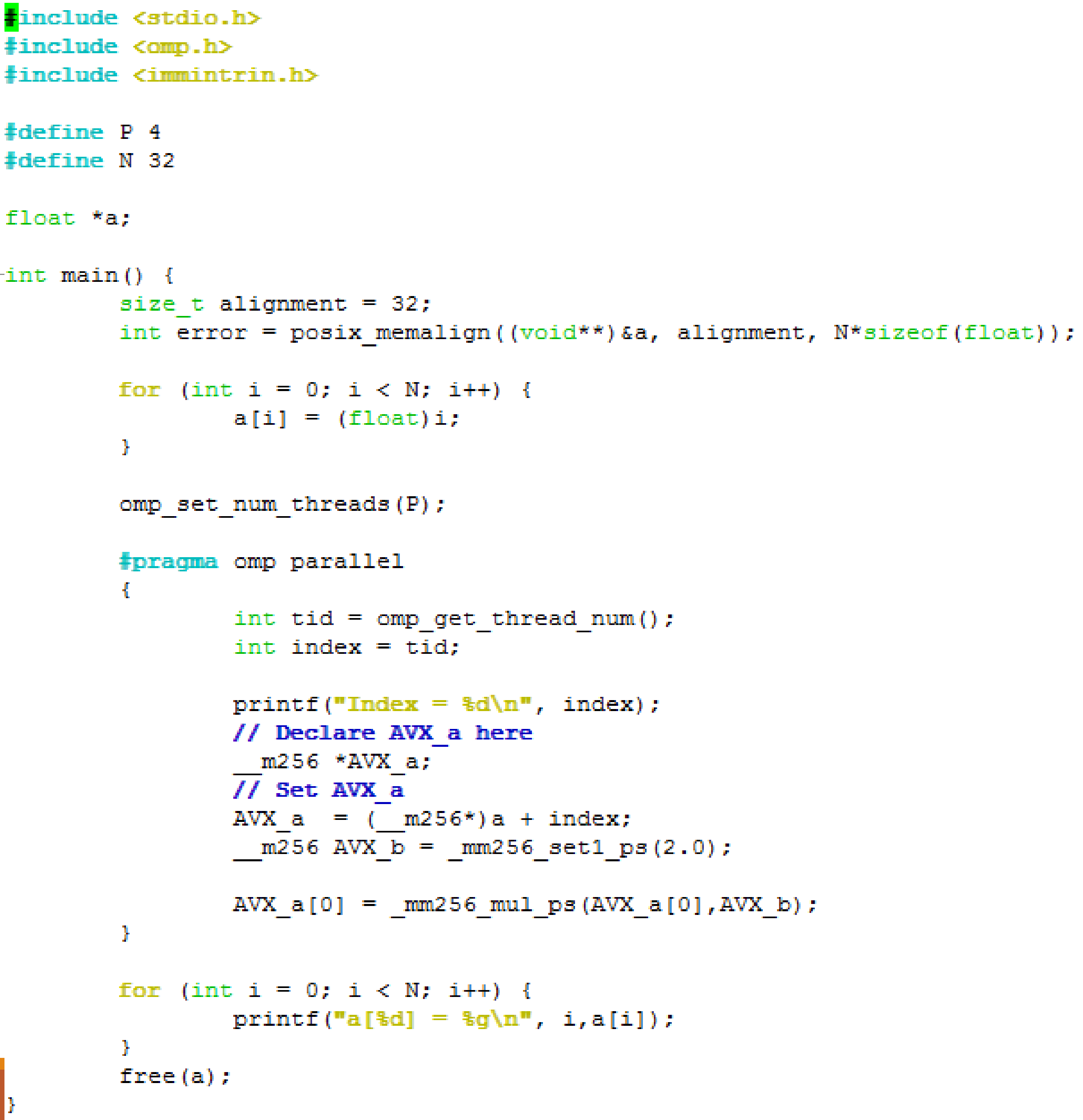
```c
#include <stdio.h>
#include <stdlib.h>    // For posix_memalign
#include <math.h>
#include <immintrin.h> // For AVX Intrinsic functions and types

void Compute_C_AVX512(float *x, float *y, float *z, int N);

int main() {
        int N = 1025;      // N = size of problem, a power of 2 is good.
        int NT;
        float *a, *b, *c;  // Three vectors containing N elements
        float error = 0.0; // Used later for testing
        size_t size;
        size_t alignment = 64;
        int i;

        // Compute Padded Length
        NT = (int)((N+16-1)/16);
        NT = 16*NT;
        size = NT*sizeof(float);
        // Print for our confirmation
        printf("Original problem size = %d, Modified problem size = %d\n", N, NT);

        // Allocate memory along cache boundaries
        posix_memalign((void**)&a, alignment, size);
        posix_memalign((void**)&b, alignment, size);
        posix_memalign((void**)&c, alignment, size);

        // Initialise a and b vectors
        for (i = 0; i < N; i++) {
                a[i] = (float)i; b[i] = (float)(2*i);
        }

        // Use the padded length when calling our AVX512 function
        Compute_C_AVX512(a, b, c, NT);

        // Check the result
        for (i = 0; i < N; i++) {
                error = error + c[i] - sqrtf(a[i]*a[i] + b[i]*b[i]);
        }
        printf("Error = %g\n", error);

        free(a); free(b); free(c);
        printf("Computation Complete\n");
        return 0;
}
```

Homework

Check out the following possibilities:

- Rewrite the example to employ OpenMP parallelization in addition to AVX vectorization.
- Repeat the above step with MPI.

And that's it. Hopefully by now I have had enough cookies to tide me over until next time.



```
#include <stdio.h>
#include <omp.h>
#include <immintrin.h>

#define P 4
#define N 32

float *a;

int main() {
        size_t alignment = 32;
        int error = posix_memalign((void**)&a, alignment, N*sizeof(float));

        for (int i = 0; i < N; i++) {
                a[i] = (float)i;
        }

        omp_set_num_threads(P);

        #pragma omp parallel
        {
                int tid = omp_get_thread_num();
                int index = tid;

                printf("Index = %d\n", index);
                // Declare AVX_a here
                __m256 *AVX_a;
                // Set AVX_a
                AVX_a  = (__m256*)a + index;
                __m256 AVX_b = _mm256_set1_ps(2.0);

                AVX_a[0] = _mm256_mul_ps(AVX_a[0],AVX_b);
        }

        for (int i = 0; i < N; i++) {
                printf("a[%d] = %g\n", i,a[i]);
        }
        free(a);
}
```