# Introduction to AVX

Intel Advanced Vector Extensions (Intel AVX) is a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on Intel architecture CPUs. It isn’t the first try for Intel to enhance performance of operations done by its CPUs using SIMD instructions. The legacy instruction sets were MMX and SSE.  
The main differences between the AVX and the legacy sets are the following:

* The 128-bit SIMD registers have been expanded to 256 bits. AVX is designed to support 512 bits as in AVX 512 that was already launched in 2016 under name of Xeon x200 massively-parallel multicore processors that was targeted super-computers.
* Adding three operands nondestructive operations instead of the Accumulator based operations.
* A few instructions take four-register operands, allowing smaller and faster code by removing unnecessary instructions.
* Three-operand Fused Multiply Add operations support (FMA3). Supported by AVX2.
* Gather support, enabling vector elements to be loaded from non-contiguous memory locations. Supported by AVX2
* Memory alignment requirements for operands are relaxed.

AVX is supported by various Intel CPUs, Compilers, and operating systems that all can be found in [[0]](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions).

# Performance Improvement

To prove that the AVX instruction set has performance higher than the legacy SSE, and conventional non-vectored (serial) codes, the following result is for running Mandelbrot set example codes (a Mandelbrot set is a computationally intensive operation on complex numbers) which were implemented using AVX, SSE, and conventional float and complex data types.

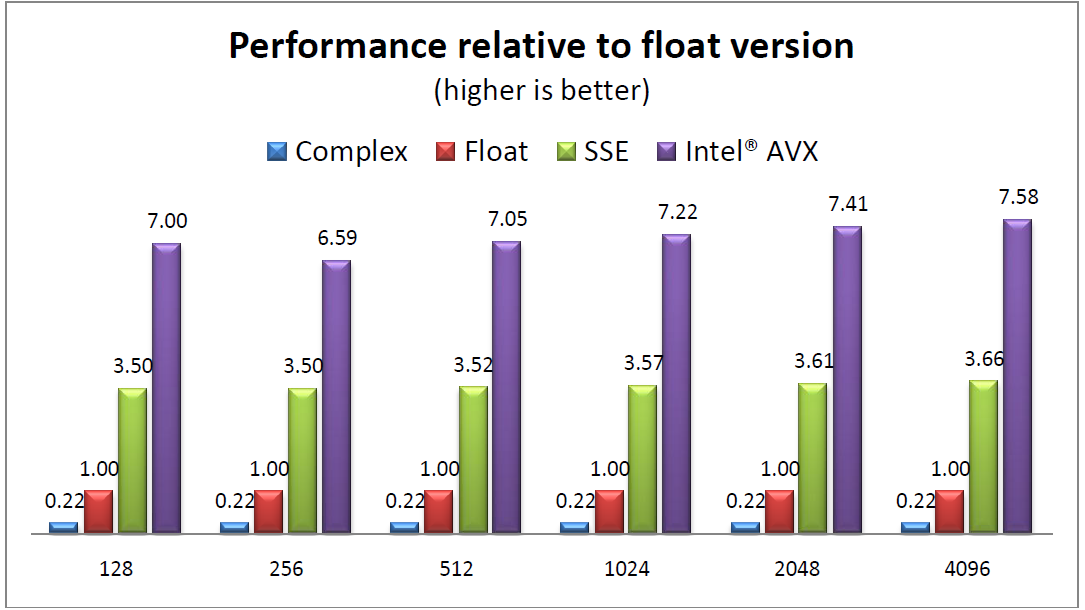


Figure : Performance across sizes

# Programming with AVX

There are two ways of programming using AVX:

* Using the Assembly instructions.
* Using C/C++ intrinsic Functions.

For developers who want to write assembly, there is a complete reference guide for AVX/AVX2 instruction sets provided by Intel. [[1]](https://www.google.com.eg/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiXlPGT3avSAhXrIpoKHY1fASUQFggYMAA&url=https%3A%2F%2Fsoftware.intel.com%2Fsites%2Fdefault%2Ffiles%2F4f%2F5b%2F36945&usg=AFQjCNFwRmucXHdk2aNdPCpNl4YE1_7HiA&sig2=oPPM46d6vwzhwWVaME9BfA&bvm=bv.148073327,d.bGs)

The easier way is to use the 2nd method or the intrinsic functions. Intrinsic functions are assembly-coded functions that allow you to use C++ function calls and variables in place of assembly instructions.

## Intrinsic Data Types

Intrinsic functions use new C data types as operands, representing the new registers that are used as the operands to these intrinsic functions.

The following figure details for which instructions each of the new data types are available. A 'Yes' indicates that the data type is available for that group of intrinsic functions; an 'NA' indicates that the data type is not available for that group of intrinsic functions.

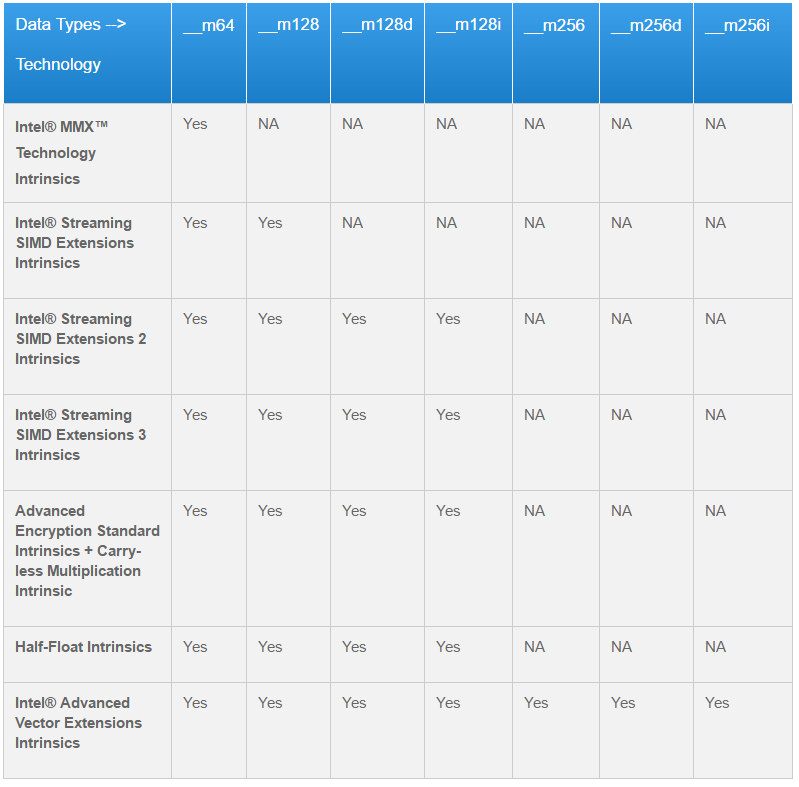


Figure : Data Types supported in various Technologies

## Naming and Usage

Most intrinsic names use the following notational convention:

\_mm\_<intrin\_op>\_<suffix>

The following figure explains each item in the syntax.

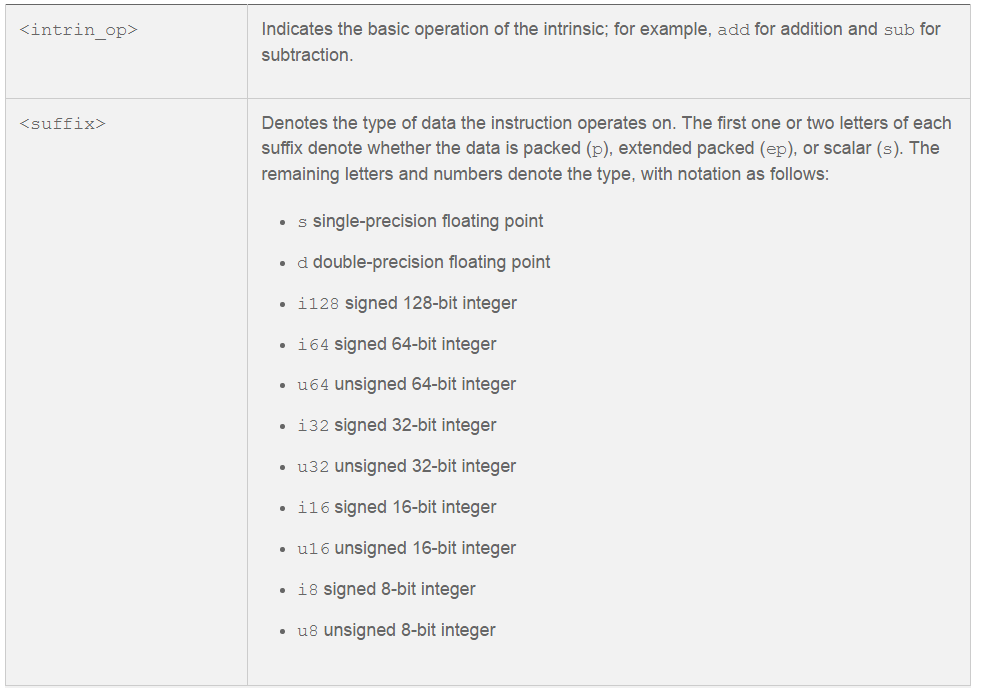


Figure : Explanation of intrinsic notation convention

One can see Intel Intrinsic Guide which is an interactive reference tool for Intel Intrinsic instructions, which are C style functions that provides access to many Intel instructions – including Intel SSE, AVX, AVX-512, and more.[[2]](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX2&expand=1365)