# Quick guide to AVX optimization

## 1. Intro

This page will first provide some background introduction on AVX instruction, and then the AVX support in DI and MBuild, finally there's a guide about how to optimize your code with AVX instruction set.

## 2. AVX instruction quick overview

AVX, or Advanced Vector Extensions, are SIMD extensions to x86 instruction set that was first introduced in 2nd gen Core processors (Sandy Bridge), that features 256 bit SIMD instructions.

So currently most of existing x86 processors would have AVX support, but not all of them. We could take advantage of the 8 element single precision floating point instructions, that could be 2x faster than SSE in general, but we have to take care of the fact that pre-Sandy Bridge processors that still exist today don't have this extension.

To achieve this, we basically have 2 approaches.

a. runtime detection

The AVX optimization would be built into the same code targeting amd64 processors. a runtime check have to be performed prior to calls into AVX code, if AVX is not supported, we'll fall back to SSE code, which we assume all amd64 processors should support.

The advantage of this approach is that only DI will have to maintain a separate AVX optimization branch, and all other technologies could simply use generic_float32 backend on x86 / amd64, and get the performance gain, without any additional development & test effort.

The down side, would be that we'll have nearly doubled code / table size for generic_float32 backend on x86 / amd64, which is not a limitation, at present. For another, the code of other technologies would be still using SSE, so a large part of the algorithm could not benefit from AVX, unless users have separate AVX code branch, and do runtime AVX support detection on their own (We'll cover how to do this later in this page).

b. install time / load time detection

This approach is similar to fat binary found on iOS / OSX, basically we compile the same code with different options, one time with -mavx, one time without it. And we combine it together, and at install time / load time, with certain mechanism the proper version will be chosen.

Although there's obvious benefit like that every technology could take advantage of AVX by simply use scalar implementation of SIMDLib, and compile the project with -mavx enabled, the adoption of this approach is restricted by the fact that there's no cross platform fat binary support. On Windows there's no fat binary, and we have to build separate binaries, and pack them together with separate tools, that means at least at CIDK level, we'll need separate build targets - one is amd64, that uses SSE for legacy compatibility, and amd64_avx, that take advantage of AVX. Through talks with DI users it is found that the extra test effort required by amd64_avx is not favorable even if it could provide remarkable performance gain, most likely because performance is not an issue on x86, in general, and in cases where it does, teams could simply add -mavx to all amd64 builds.

Given this, the runtime detection approach is used in DI, and also made available to users that want to have AVX optimization in their own code.

## 3. DI & M-Build support for AVX optimization

### 3.1. runtime AVX support check

To detect AVX support at runtime, DI provided a new API

**DLB_support_avx()**

That will provide cross-platform runtime check for AVX support. It will return !0 if AVX is supported.

Call this function at runtime before calling AVX optimized code on amd64 (or before assigning function pointers), e.g.,

```
if(DLB_support_avx())
    {
      fp = fn_avx;
    }
    else
    {
      fp = fn_sse;
    }
```

It will always return 0, when called on a platform that doesn't support AVX, e.g., Windows platforms that use old versions of Visual Studio.

## 3.2. compile time helper macro

Although DLB_support_avx() will return a constant zero when called on a platform that doesn't support AVX, compilers sometimes still refer to the symbols in AVX branch, causing unresolved symbols at link time. To avoid this, DI provided a macro **DLB_TOOLCHAIN_SUPPORT_AVX** that will only be defined on toolchains that have AVX support, example usage would be

```
#if defined (__SSE2__) && defined(DLB_BACKEND_GENERIC_FLOAT32) &&
defined(DLB_TOOLCHAIN_SUPPORT_AVX)
    if(DLB_support_avx())
    {
      fp = fn_avx;
    }
    else

#endif
    {
      fp = fn_sse;
    }
```

## 3.3. M-Build support

The AVX optimized code will have to be compiled with -mavx option (or other variants), while the rest of the code base will be compiled without it. That means certain files will be compiled with different options than others, in the same project. This is not previously supported in M-Build.

To achieve this, M-Build introduced a new file tag **feature("avx")** the example usage would be like this

```
[foo_objects]
bar.c

[foo_objects.toolchain_support_avx]
feature("avx"):bar_avx.c
```

So by default only bar.c will be compiled. And on toolchains that have AVX support, both bar.c and bar_avx.c will be compiled. Moreover, bar_avx.c will be compiled with AVX option (e.g., -mavx for Linux gcc).

## 4. How to optimize your code with AVX instruction set

As mentioned earlier, AVX optimization is a two-fold problem. First we have to deal with the fact that AVX is only available on part of existing x86 processors, after that we could discuss how to optimize our code with AVX.

### 4.1. Dealing with AVX availability

#### 4.1.1. AVX support guaranteed

If you're sure that for your product, all x86 / amd64 processors will have AVX support, simply use something like

```
[os_linux.processor_amd64.toolchain_gnu]
@att MAKE_CFLAGS += '-mavx'
```

And regenerate Makefiles for all projects, so amd64 targets will be compiled with the assumption that AVX is supported, and the performance should already benefit from AVX since compilers should be able to auto-vectorize your code to some extent, and generate AVX instructions for those parts.

If your code is already extensively optimized with SIMDLib, there might be some build / runtime errors, since the VEC_N is now increased to 8, for AVX. We'll cover this later in section 4.2.2.

#### 4.1.2. Check AVX availability at runtime

If you have to target all existing x86 / amd64 processors, which is the common case, you could build both AVX version and legacy version into the same binary, and do runtime AVX check to select between versions, this approach is supported by DI / M-build. The steps are as follows.

1. For the module of interest, identify the functions to be optimized with AVX, create a separate source file for the AVX optimized version. And since the AVX and non-AVX version will live together in the same binary, the AVX optimized functions need a different name. e.g.,

```
/* in fft_generc.c */
void fft_64(float* out, float* in)
{
   ...
}

void fft_128(float* out, float* in)
...
```

```
/* in fft_avx.c */
void fft_64_avx(float* out, float* in)
{
   ...
}

void fft_128_avx(float* out, float* in)
...
```

2. And in the code that initialize function pointers, do the runtime check and select between AVX and legacy versions.

```
/* Could also in fft_generc.c */
FFT_HANDLE fft_open(int size)
{
#if defined (__SSE2__) && defined(DLB_BACKEND_GENERIC_FLOAT32) &&
defined(DLB_TOOLCHAIN_SUPPORT_AVX)
    if(DLB_support_avx())
    {
      switch(size)
        case 64:
          return fft_64_avx;
        case 128:
          return fft_128_avx;
        ...
    }
    else
#endif
    {
      switch(size)
        case 64:
          return fft_64;
        case 128:
          return fft_128;
        ...
    }
}
```

The use of DLB_support_avx() function and DLB_TOOLCHAIN_SUPPORT_AVX has already been covered previously. Here the above example just shows the recommended approach that do the runtime AVX support check at initial time.

3. Finally, since the fft_avx.c need to be compiled with avx option, while fft_generic.c without, the manifest file should look like, similar to the example in section 3.3.

```
[fft_objects]
fft_generic.c

[fft_objects.toolchain_support_avx]
feature("avx"):fft_avx.c
```

When all done, regenerate Makefiles / VS project files, and build the executable, it should now be able choose from AVX optimized fft, and legacy fft at runtime, according to the AVX support check results.

## 4.2. Optimize code with AVX

### 4.2.1. use auto-vectorization

Contemporary compilers could do auto-vectorization for simple loops, when avx options specified, AVX instructions will be generated. This is the easiest way to take advantage of AVX, though of course there would be certain limitations.

1. compilers will not do a good job for "complex" loops, even if it's just that the data being processed is interleaved complex arrays.
2. It is found that compilers would still generate data load / store instructions with 128-bit width.
3. certain optimizations would require a re-order of tables, which cannot be done with compilers automatically.

On top of these limitations, if the code is already SIMDLib optimized, things could be a little bit complicated. Lots of SIMDLib optimizations require different table / different code branches, for different VEC_N size, so simply compile those source files with avx option might cause build / runtime errors. To avoid errors, one option would be defining the macro DLB_SIMD_USE_SCALAR to make SIMDLib implementation fallback to the scalar implementation, and rely on auto-vectorization to generate AVX code, where possible. Auto-vectorization is limited to some simple cases, so it is possible that the performance could become worse. Therefore, the recommended approach would be writing AVX specific code with SIMDLib, which is covered in next section.

### 4.2.2. Use SIMDLib

When the source file is compiled with avx option, on generic float32 backend, the SIMDLib would select AVX implementation, and the VEC_N will be set to 8. The same set of operators like DLB_VVld_vec_L, DLB_VaddVV, DLB_vec_Lst_zipVV... is supported.

The usage is similar, but there're a few things to take care about.

1. Since the VEC_N is increased to 8, vectors whose size is not a multiple of 8 need to be handled correctly.
2. The permutation between upper and lower 4 elements is much heavier than the permutation within the upper and lower half
    a. So the zip/unzip operators become expensive and should be avoided when possible.
    b. Sometimes it would be useful to use the permutation within upper and lower half, with the ymmintrinsics **_mm256_shuffle_ps**(). Since this is specific to AVX, it is not added to SIMDLib, we may wait and see what ARM will do for its 256 bit SIMD instruction set.
3. The switch between legacy SSE code and AVX code would introduce performance penalties. Compilers will insert VZEROUPPER instruction to avoid penalty, but still, it is recommended to avoid frequent switch between AVX and non-AVX code.
    a. for more details, refer to https://software.intel.com/en-us/forums/intel-isa-extensions/topic/301853
4. The use of AVX might lower CPU frequency. So it is recommended to avoid frequent switch between AVX and non-AVX code.
    a. This whitepaper from Intel has more details: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf
    b. In this whitepaper Intel seems deliberately mixed the term AVX (1st gen AVX) and AVX (generally term, means both AVX 1st gen, and AVX2), but from our tests on Intel NUC, no frequency downscale is found, as we only use AVX but not AVX2.
5. So performance gain compared to SSE is less than 50%, in general.