Application Note

C7000 Optimization Starting Guide

Abstract

The process of migrating from another DSP or processor to the C7000 requires some optimizations to take advantage of the benefits of the C7x architecture. This document covers the general order of optimization from the easiest to more difficult steps.

As a summary, there are a few basic steps when starting.

1. Run the algorithm on the C7x core and benchmark the initial performance to get a starting point.
2. Turn on compiler optimizations.
3. Set the correct memory cache settings.
4. Use existing optimized libraries that are provided in the SDK.
5. Hand-optimize any custom modules that do not exist.

Table of Contents

1 Profiling Existing Algorithms 2
1.1 Measuring Cycles With the CPU Cycle Counter 2
2 Compiler Optimizations 3
2.1 Verifying Compiler Optimization with Generated Assembly 3
3 Double Check Memory Cache Settings 5
4 Use Existing Optimized Libraries 7
5 Optimizing Custom Kernels/Modules 7
6 Example 8
6.1 Example System Results Summary 8

Trademarks

All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

1 Profiling Existing Algorithms

It is recommended that the algorithm in question be profiled to determine if optimization is necessary, and to help determine the improvements made throughout each step. Profiling an algorithm is essentially determining how many clock cycles are used for an iteration of the loop. To do this, the algorithm or code must be able to run on a C7x core. Inside of the FreeRTOS SDK / MCU Plus SDK, there are several examples given. The quickest approach is to start from the empty or hello world example project for a C7x core, and then copy in the algorithm that exists. Refer to the provided software example code if you want an example. It's possible to use arrays with static data to simply profile the algorithm and then add the external peripherals/real data later. The primary goal is to get the starting point for performance metrics.

The C7000 C/C++ Optimization Guide contains a section about how to profile code, and provides 4 different methods. The simplest way to do profile is to use a C7x compiler intrinsic instruction to get the current value of the CPU counter (TSC register). The CPU cycle counter is a simple up counter that increments by one each clock cycle. It is a 64-bit register, so the risk of overflow is quite small, as this allows for about 585 years of runtime at ns resolution without an overflow when the DSP core runs at 1GHz. Utilizing the CPU cycle counter (TSC register) can capture how many clock cycles are used for various parts of the algorithm, and provide helpful insights into optimization improvements.

It is recommended to iterate the algorithm multiple times (at least 3 times) to allow the cache to warm up, since cold cache hits are the slowest/worst case performance, and in a running system the cache is warm after a few iterations.

1.1 Measuring Cycles With the CPU Cycle Counter

The C7x provides a 64-bit CPU cycle counter that increments by one each clock cycle that can be read using the `__TSC` intrinsic. To measure the cycle count of a function, record the counter value before and after execution and take the difference.

```
#include <c7x.h>
#include <stdio.h>
uint64_t start, end, elapsed;
/* Warm up cache */
for (int i = 0; i < 3; i++)
{
    myAlgorithm(input, output);
}
/* Measure one iteration */
start = __TSC;
myAlgorithm(input, output);
end = __TSC;
elapsed = end - start;
```

To convert cycle counts to real time, divide by the C7x core frequency. At 1GHz, one cycle equals one nanosecond. The actual frequency depends on the device grade and the clock configuration — refer to the SBL Logs and device datasheet to confirm.

```
time_us = elapsed_cycles / C7x_freq_MHz;
time_ms = elapsed_cycles / (C7x_freq_MHz × 1000);
```

Note that `C7x_freq_MHz` is not an actual variable, and you'll need to supply the actual value. This is the most simple way to measure the execution time of an algorithm and profile the code. It is possible to get more granular and measure time of individual functions within the algorithm to further break-down where the algorithm spends most computational cycles.

Copyright © 2026 Texas Instruments Incorporated

2 Compiler Optimizations

The first step to improve performance is also the least amount of effort required, which is to let the C7x compiler optimize the code where it can. Within the code composer studio (CCS) project, open the project settings and check the optimization level used under Tools, C7000 Compiler, Optimization.

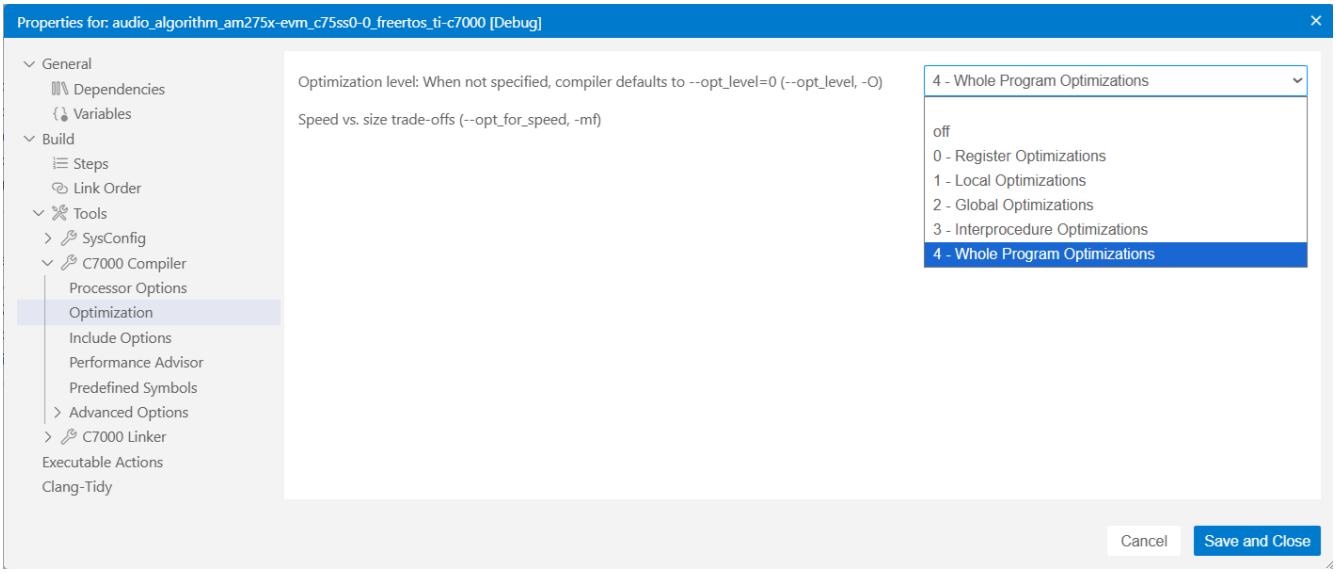


Figure 2-1. Setting Optimization Level

For the optimization level, the higher the number (0 to 4), the more compiler optimizations are applied, at the expense of ease of debug. It is typically recommended to have the optimization level at 3 or higher to provide the best opportunity for optimizing existing code without much hand-optimization. The speed versus size tradeoff is used to tell the compiler whether to prioritize application code memory footprint (size or 0) or fastest run time (speed or 5) or somewhere in between. During debug, these optimizations are typically off to allow the user to easily debug the program, but the performance is reduced as a result. As the compilier more heavily optimizes the code, it becomes more difficult to step through code or look at variables, as sometimes these are completely removed. For detailed information about what these options do, refer to the *C7000 C/C++ Optimization guide*.

2.1 Verifying Compiler Optimization with Generated Assembly

After setting the optimization level, it is worth confirming the compiler is actually pipelining and vectorizing the inner loops of your code. Setting a high optimization level usually tells the compiler to optimize the code but there can be times when it fails.

Software pipelining is a technique the C7x compiler uses to overlap successive iterations of a loop in order to keep the DSP's multiple functional units busy simultaneously. Without pipelining, the processor executes one full loop iteration before starting the next, leaving most functional units idle at any given cycle. With pipelining, the compiler overlaps iteration N+1 with iteration N so that different parts of different iterations are executing at the same time.

Vectorization takes advantage of the C7x's wide vector data path — 256-bit on the AM27x family and AM62D. Rather than operating on one value per instruction, a vectorized loop processes multiple values simultaneously. For example, a loop operating on 32-bit floating-point values can process 8 values per cycle on the AM27x family. Vector instructions in the assembly are identifiable by the V prefix — for example `VFADDSP` and `VFMPYSP` — whereas scalar equivalents have no prefix.

Two compiler options provide visibility into what the compiler did:

1. `--gen_opt_info=2` (or -on2) generates a .nfo file alongside the compiled object. This file contains a human-readable summary of the high-level optimizations applied to each loop, along with advice when optimizations could not be applied. This is the quickest starting point for diagnosing issues.

Copyright © 2026 Texas Instruments Incorporated

2. `--keep_assembly` or `--src_interlist` retains the generated .asm file, which contains a detailed software pipeline information comment block for each loop showing the achieved II, resource usage, and reason if pipelining was not applied. Adding `--debug_software_pipeline` provides further detail in these blocks.

Both options can be set in CCS under Project Properties > Tools > C7000 Compiler > Advanced Options (Runtime Model for `--debug_software_pipeline` and Supplemental Information for `--gen_opt_info`) or directly adding the compiler flag in the makefile. A typical workflow is to start with the .nfo for a high-level view, then open the .asm for deeper analysis when needed. The `--keep_asm` option is used automatically when `--debug_software_pipeline` is enabled.

When a loop fails to pipeline or vectorize, the most common causes and fixes are:

1. Function Call in the Loop

   A function call inside a loop prevents software pipelining if the compiler cannot inline it. To resolve this, define the called function in one of the following ways:

   - In the same source file as the call, using the inline keyword
   - In a .h header file included via #include, using static inline

   Note: The compiler performs automatic inlining at `--opt_level=3` and `--opt_level=4`, so some calls may be inlined without manual changes at higher optimization levels.
2. Unknown Trip Count

   Providing the compiler with information about how many times a loop will execute helps it determine whether vectorization and loop optimizations are profitable, and whether a redundant loop is necessary.

   Software pipelined loops require a minimum number of iterations to execute legally. If the compiler cannot guarantee the actual iteration count will always meet this threshold, it generates two versions of the loop — a fast pipelined version and a slower redundant fallback. The assembly comment block will state `;* Redundant loop generated` when this occurs, and specifies the minimum safe iteration count.

   Use `#pragma MUST_ITERATE` to provide the known interation bounds:

   ```
   #pragma MUST_ITERATE(128, 128, 8)
   for (int i = 0; i < n; i++) {
       ...
   }
   ```

   The min value comes from the application itself: if the algorithm always processes a fixed block size, that is the guaranteed minimum. For example, an audio algorithm processing 128 samples per block will always execute the loop at least 128 times. After adding the pragma, check the assembly to confirm `;* Redundant loop generated` is no longer present. Note that providing incorrect values will cause the compiler to generate code that can produce incorrect behavior.
3. Pointer Aliasing / Loop-Carried Dependency

   If the compiler cannot determine that the input and output arrays occupy separate memory areas, it must conservatively assume that the store in iteration i may affect the load in iteration i+1. This creates a loop-carried dependency that prevents software pipelining. To resolve this, use the restrict keyword on pointer parameters that are guaranteed not to overlap.

   ```
   void weighted_sum(int * restrict a, int * restrict b, int * restrict out, int weight_a, int
   weight_b, int n)
   ```

   Note

   Using restrict incorrectly (on pointers that do overlap) can cause the compiler to produce code with undefined behavior and incorrect results.
4. Loop Not Vectorizing

   The compiler may fail to vectorize a loop entirely, or may vectorize only some operations within it, leading to an inefficient mixed loop where some instructions are vector and others are scalar.

   - If the loop is not vectorizing at all, consider using C7x vector types directly in the source

Copyright © 2026 Texas Instruments Incorporated

- If partial vectorization is the issue, it may be necessary to vectorize the full loop by hand using vector types and intrinsics.

Note: See the C7000 C/C++ Optimization Guide (SPRUIG8) for more information.

3 Double Check Memory Cache Settings

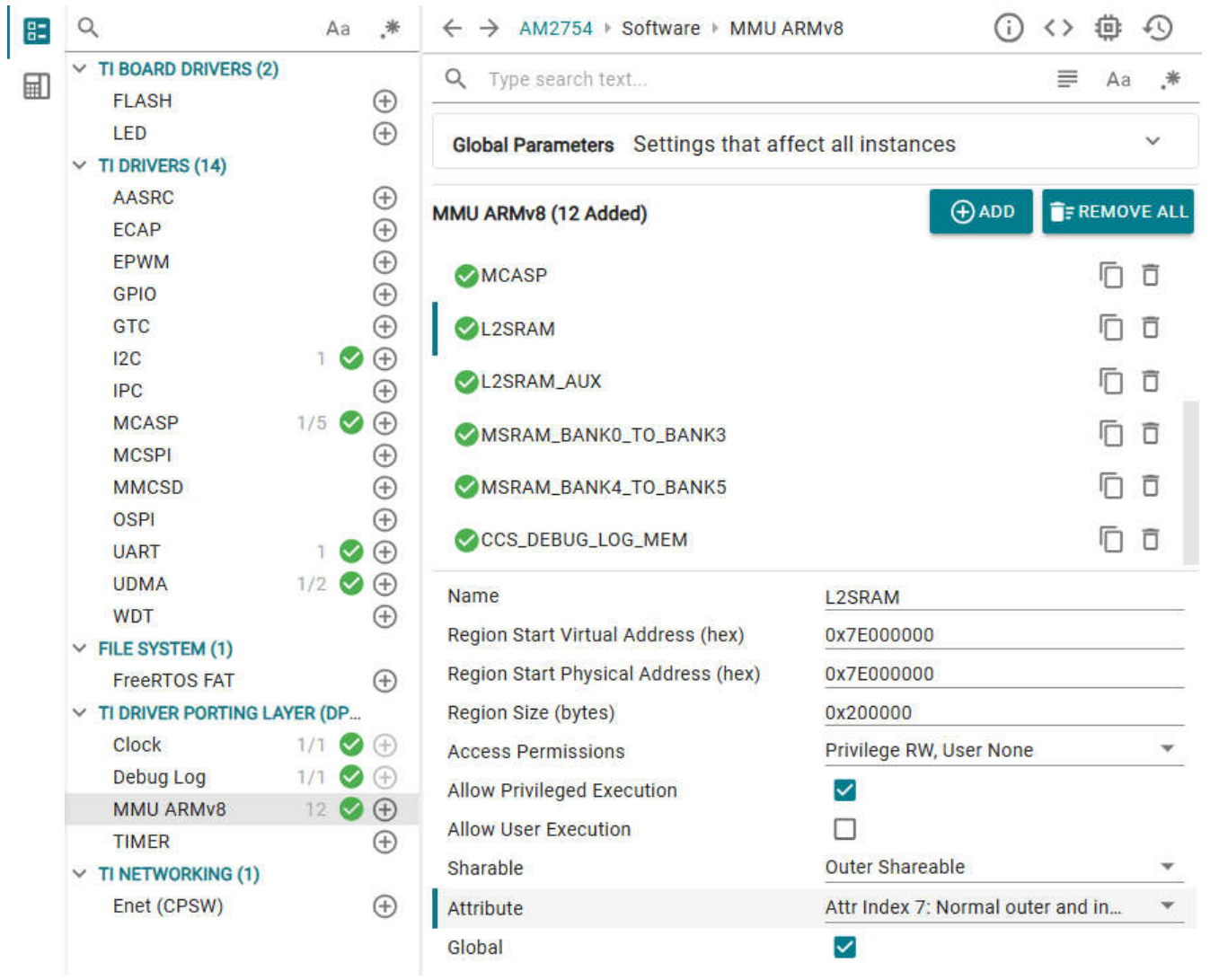
The best performance is achieved when utilizing write-back cache settings instead of write-through. The example projects are configured this way out of the box, but there can be a profound change in performance if the cache is mis-configured.

To summarize the difference, a cache write-back mode (sometimes referred to as write-deferred) is a mode where writes to an address write to the cache only (leaving the original memory location out-of-sync) until it is evicted from the cache, at which point the cache contents are copied back to the original memory location. Since the cache is typically faster than other forms of memory, the DSP is able to write and read this data faster. The risk with this situation is if the memory is shared across multiple cores, the data has to be written back to the memory location so that another core can see the valid data. A cache write back call can be used to force this only when needed so that the improved performance of the cache is still used.

Cache write-through mode is different, where any writes to a memory location are updated in both the cache and the original memory location at the same time. This has a performance downside on writes because the application must wait for the slower memory to be written to. Reads still happen from the cache, but writes are slower because the cache and the memory location are kept in-sync at every write.

Inside of the CCS project, there is a sysconfig file (ending in .syscfg) that contains the hardware settings used. Inside the sysconfig file, there is a section called *MMU* or *MMU ARMv8*. This section contains all the memory locations mapped out.

Copyright © 2026 Texas Instruments Incorporated



Click to select a section (typically the L2SRAM or MSRAM sections are where data is stored), and check that the attribute field is set to write-back cacheable (Attr Index 7). In some special applications, other settings might be required. For example, when write-back is used, memory used by a single core can be cached in the processor cache. With write-back cacheable, when the data is updated, it is only updated in the cache and not the original memory location until the line is purged from the cache. This can create a scenario where the proccessor has done some modifications to a buffer, and the changes exist in the cache only, and not the original memory location. This can cause stale data being pulled from the original memory location (for example for MCASP transactions after an algorithm updates the data). When the updates to the memory location are completed and MUST be synced back to the original location for another hardware peripheral or processor, the application needs to call the `CacheP_inv()` function to force the cache writeback to occur so that the original memory location is updated with the values from the cache.

Copyright © 2026 Texas Instruments Incorporated

4 Use Existing Optimized Libraries

As part of the FreeRTOS SDK / MCU Plus SDK, there are several highly optimized libraries provided which can be used to further improve performance. These libraries provide common functions with code that is already optimized for the C7x architecture. There are libraries for math, digital signal processing, vision processing, and audio processing.

Utilizing these optimized functions can provide further improved performance for the algorithm. Note that the C7x architecture is vector-based, which allows for operations on multiple blocks of data at the same time. The performance generally improves as the size of data (block size) increases to take advantage of the vector architecture.

The location of the documentation for these various libraries lives inside of the SDK folder. For example `<SDK base directory>/source/dsplib/docs/`. Example code is also provided for the provided kernels. It is generally recommended to utilize these optimized DSP functions where possible since they are optimized to take advantage of several features of the C7x architecture such as the streaming engines and vectors, which can provide significant improvements on performance for the sections of the algorithm that are doing the actual computation.

One of the key things to note with these optimized functions is that they are typically used with an init() and an exec() function. The init function is used to set up one-time things such as the streaming engine or address generators. The exec function is used to do the computation and should be called in the loop. The init function should be called only once or when parameters change. Ideally the init function is called once and the exec() is looped until the block of data has been fully processed. This provides the largest performance lift.

On top of utilizing the provided libraries, they must also be used correctly to get a performance improvement. As an example, if an *add* function that is used to add two arrays together is only dealing with one element at a time, there is no performance improvement. In the case of the AM275 with a 256 bit vector width architecture, it would be able to add up to eight 32-bit values together in a single clock cycle, but the function must be called correctly to deal with vector-width data instead of a single-scalar data. The examples provided in the libraries show how to correctly call these functions.

5 Optimizing Custom Kernels/Modules

One of the last steps for optimization after all of the above steps have been taken is to migrate any custom modules that do not currently exist in any of the provided libraries. To do this requires knowledge of the C7x architecture and the additional peripherals to improve the performance. More information is provided in detail inside of the *C7000 C/C++ Code Optimization Guide: Basic Code Optimization Strategies and Techniques* followed with the *Advanced Techniques*. There are several resources available online to help ramp up on the C7x architecture, such as the *video training series*.

Copyright © 2026 Texas Instruments Incorporated

6 Example

Using the steps outlined earlier, we can look at the provided example algorithm (see example code provided on github) and go through the process. We will use a simple example algorithm, which utilizes a few kernels.

The example algorithm is a simple audio algorithm. The following are the parameters of the system:

1. 8 channels of audio (48kHz), 16-bit samples (Q15), interleaved data
2. Block size: 128 samples per channel (1,024 samples total)
3. Input and Output samples are to be signed 16-bit samples (Q15)

In this imaginary system example, there will be a simple algorithm used, which will do the following

1. Convert the 16-bit samples to a float
2. 3 stage cascade biquad low pass filter
3. Global gain of 0.75 is applied
4. Convert back to a 16-bit sample for use with DAC/amplifiers

This example system will start with the most basic "Natural C" code, which does not have any optimized code for a DSP, and we will go through each of the steps outlined above and see what the performance lift is on a C7000, using the AM275 EVM as the test platform.

6.1 Example System Results Summary

Reviewing the performance lift throughout the steps shows how spending some time to optimize the code for the C7000 DSP can significantly improve the performance.

Table 6-1. Effects Compiler Optimization Flag

Code Type	-O Flag	Total Cycles	Q15 to Float Cycles	Cascade Biquad Cycles	Gain Cycles	Float to Q15 Cycles	Cycles per Sample	Comments
Natrual C	0	522,142	18,459	446,010	14,364	43,238	509.9	
	1	249,780	16,412	178,789	13,337	41,171	243.9	
	2	180,145	2,443	150,692	284	26,670	175.9	
	3	180,146	2,443	150,592	284	26,671	175.9	
	4	177,840	241	150,694	176	26,667	173.7	
C7 DSPLib	0	3,602	185	2,981	184	137	3.5	
	1	3,591	185	2,981	184	137	3.5	
	4	3,579	185	2,981	184	137	3.5	

Summary: Looking at the data above, the effects of the compiler's optimization flag (-O) provides a significant performance lift of all of the modules of the algorithm, and in some cases is able to match the performance of the optimized DSPLib kernel. This is because the compiler is able to look for patterns and set up vector-width calculations, providing significant speed up.

There is a significant performance improvement beyond the best natural C code once all of the kernel modules were replaced with the hand-optimized DSPLib kernels. The performance improves from 173.7 clock cycles per sample to 3.5 clock cycle per sample when using the optimized C7000 libraries.

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025