参考資料

Application Note

C7000 最適化入門ガイド

概要

別の DSP またはプロセッサから C7000 に移行するプロセスでは、C7x アーキテクチャの利点を利用するために、ある程度の最適化が必要です。このドキュメントでは、最適化の一般的な順序を最も簡単なステップからより困難なステップまで説明します。

要約すると、起動時にいくつかの基本的な手順があります。

1. C7x コアでアルゴリズムを実行し、初期性能のベンチマークを確立して、出発点を決定します。
2. コンパイラの最適化をオンにします。
3. 正しいメモリ キャッシュ設定を設定します。
4. SDK で提供されている既存の最適化済みライブラリを使用します。
5. 存在しないカスタム モジュールを手動で最適化します。

目次

1 既存のアルゴリズムのプロファイリング 2
1.1 CPU サイクル カウンタでのサイクルの測定 2
2 コンパイラの最適化 3
2.1 生成されたアセンブリでのコンパイラ最適化の検証 3
3 メモリ キャッシュ設定をダブル チェックします 5
4 既存の最適化ライブラリを使用します 7
5 カスタム カーネル/モジュールの最適化 7
6 例 8
6.1 システム結果の概要の例 8

商標

すべての商標は、それぞれの所有者に帰属します。

English Document: SDAA248

Copyright © 2026 Texas Instruments Incorporated

1 既存のアルゴリズムのプロファイリング

問題のアルゴリズムをプロファイルして、最適化が必要かどうかを判断し、各ステップで行われた改善を決定することをお勧めします。アルゴリズムのプロファイリングとは、基本的に、ループの繰り返しに使用されるクロック サイクル数を決定することです。これを行うには、アルゴリズムまたはコードを C7x コアで実行できる必要があります。FreeRTOS SDK/MCU Plus SDK の中には、いくつかの例があります。最も迅速なアプローチは、C7x コアを対象にして空または hello world のサンプル プロジェクトから開始し、存在するアルゴリズムにコピーすることです。サンプルが必要な場合は、提供されているソフトウェア サンプル コードを参照してください。静的データを持つ配列を使用して、アルゴリズムを単純にプロファイルし、後で外部ペリフェラルや実データを追加することも可能です。主な目標は、パフォーマンス メトリックの出発点を取得することです。

『C7000 C/C++ 最適化ガイド』には、コードのプロファイル作成方法に関するセクションが含まれ、4 種類の方法があります。プロファイルを行う最も簡単な方法は、C7x コンパイラの組み込み命令を使用して、CPU カウンタ (TSC レジスタ) の現在の値を取得することです。CPU サイクル カウンタは、クロック サイクルごとに 1 ずつインクリメントするシンプルなアップカウンタです。これは 64 ビット レジスタなので、オーバーフローのリスクは非常に小さくなります。これにより、DSP コアが 1GHz で動作するときにオーバーフローなしに、ns の分解能で約 585 年間の実行時間を実現できます。CPU サイクル カウンタ (TSC レジスタ) を利用すると、アルゴリズムのさまざまな部分で使用されているクロック サイクル数をキャプチャし、最適化の改善に役立つ情報を得ることができます。

キャッシュのウォームアップを可能にするために、アルゴリズムを複数回 (少なくとも 3 回) 繰り返すことをお勧めします。これは、コールド キャッシュ ヒットは最も遅い / 最悪の場合のパフォーマンスであり、実行中のシステムでは、数回の反復後にキャッシュがウォームになるためです。

1.1 CPU サイクル カウンタでのサイクルの測定

C7x には 64 ビットの CPU サイクル カウンタがあり、クロック サイクルごとに 1 ずつインクリメントし、__TSC 固有の機能を使用して読み取ることができます。関数のサイクル カウントを測定するには、実行前と実行後のカウンタ値を記録し、その差を取ります。

```
#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;
```

サイクル カウントをリアルタイムに変換するには、C7x コア周波数で除算します。1GHz では、1 サイクルは 1 ナノ秒に相当します。実際の周波数は、デバイスのグレードとクロック構成によって異なります。確認するには、SBL ログとデバイスのデータ シートを参照してください。

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

C7x_freq_MHz は実際の変数ではないため、実際の値を指定する必要があることに注意してください。これは、アルゴリズムの実行時間を測定し、コードをプロファイルする最も簡単な方法です。アルゴリズム内の個々の関数の時間をより細かく測定し、アルゴリズムがほとんどの計算サイクルを費やす場所をさらに細かく分析することが可能です。

Copyright © 2026 Texas Instruments Incorporated

2 コンパイラの最適化

性能を向上させる最初のステップは、必要な労力が最小化されます。これは、C7x コンパイラがコードを最適化することです。Code Composer Studio (CCS) プロジェクト内でプロジェクト設定を開き、「ツー」、「C7000 コンパイラ」、「最適化」の下で使用する最適化レベルを確認します。

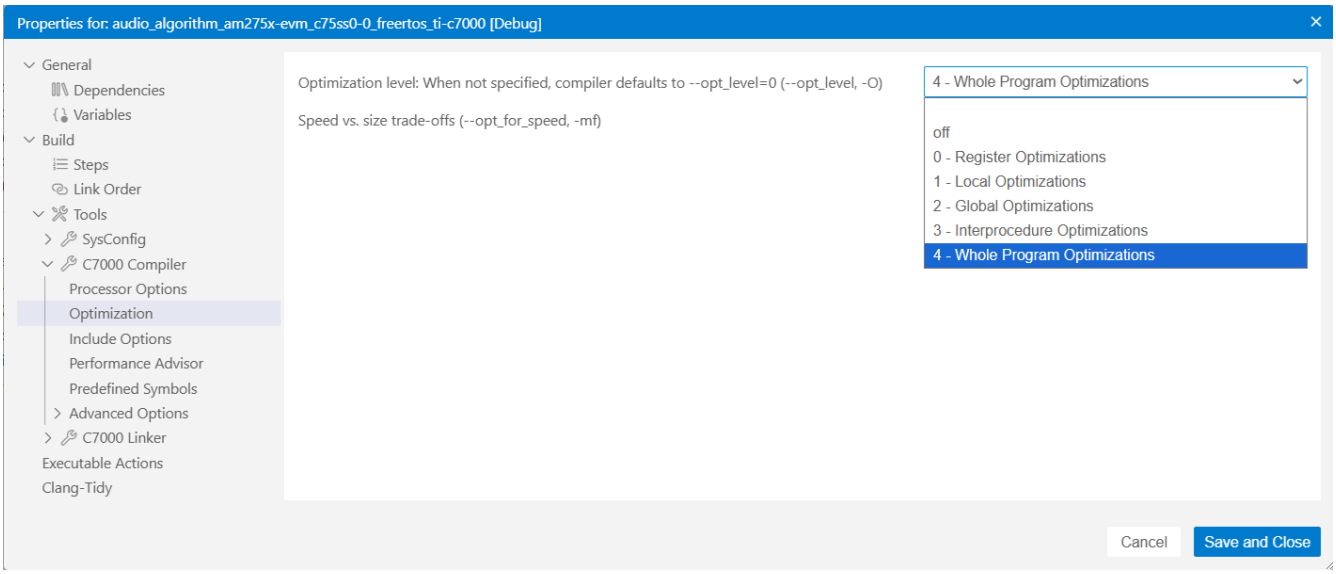


図 2-1. 最適化レベルの設定

最適化レベルでは、0 から 4 までの数が大きいほどコンパイラ最適化が適用されますが、デバッグが容易になります。通常、手作業による最適化を行わずに既存のコードを最適化する最良の機会を提供するために、最適化レベルを 3 以上にすることを推奨します。速度とサイズのトレードオフを利用して、アプリケーションコードのメモリ フットプリント (サイズまたは 0)、最速の実行時間 (速度または 5)、またはその中間のどちらに優先順位を付けるかをコンパイラに指示します。デバッグ中は、ユーザーがプログラムを簡単にデバッグできるようにするために、これらの最適化は通常オフになりますが、結果としてパフォーマンスが低下します。コンパイラがコードを大幅に最適化するにつれて、コードをステップスルーしたり変数を見たりすることが難しくなります。これらのオプションの詳細については、『C7000 C/C++ 最適化ガイド』を参照してください。

2.1 生成されたアセンブリでのコンパイラ最適化の検証

最適化レベルを設定した後、コンパイラが実際にパイプライン化され、コードの内部ループをベクトル化していることを確認する価値があります。最適化レベルを高く設定すると、通常はコンパイラにコードを最適化するように指示しますが、失敗する場合があります。

ソフトウェア パイプライン化は、C7x コンパイラがループの連続した反復をオーバーラップさせる手法であり、DSP の複数の機能ユニットを同時にビジー状態に維持します。パイプラインを使用しない場合、プロセッサは次のループを開始する前に 1 回フルループを繰り返し実行し、ほとんどの機能ユニットは任意のサイクルでアイドル状態のままにします。パイプライン方式では、コンパイラはイテレーション N+1 をイテレーション N と重ね合わせることで、イテレーションの異なる部分が同時に実行されるようになります。

ベクトル化では、C7x のワイド ベクトル データ パス (AM27x ファミリと AM62D が 256 ビット) を利用しています。ベクトル化ループは、命令ごとに 1 つの値で動作するのではなく、複数の値を同時に処理します。たとえば、32 ビットの浮動小数点値で動作するループは、AM27x ファミリでサイクルあたり 8 つの値を処理することができます。アセンブリ内のベクター命令は、V 接頭辞 (VFADDSP や VFMPYSP など) で識別できますが、スカラ命令には接頭辞はありません。

2 つのコンパイラ オプションにより、コンパイラの動作を確認できます。

1. `--gen_opt_info=2` (または -on2) コンパイルされたオブジェクトと一緒に .nfo ファイルを生成します。このファイルには、各ループに適用される高度な最適化の概要と、最適化を適用できなかった場合のアドバイスが、人間が読める要約が含まれています。これは、問題を診断するための最も迅速な出発点です。

Copyright © 2026 Texas Instruments Incorporated

2. `--keep_assembly` または `--src_interlist` オプションを指定すると、生成された .asm ファイルが保持されます。このファイルには、各ループごとに、達成された II、リソース使用量、およびパイプライン化が適用されなかった場合の理由を示す、詳細なソフトウェア パイプライン情報のコメント ブロックが含まれています。`--debug_software_pipeline` を追加すると、これらのブロックで詳細に説明します。

どちらのオプションも CCS で「プロジェクト プロパティ」>「ツール」>「C7000 コンパイラ」>「詳細オプション」(`--debug_software_pipeline` のランタイム モデルおよび `--gen_opt_info` の補助情報) で設定するか、makefile にコンパイラ フラグを直接追加することができます。一般的なワークフローでは、.nfo を使用してハイレベル ビューを作成し、必要に応じて .asm を開いて詳細な分析を行うことができます。`--keep_asm` オプションは`--debug_software_pipeline` が有効な場合に自動的に使用されます。

ループがパイプラインまたはベクトル化に失敗した場合、最も一般的な原因と修正は次のとおりです。

1. 関数ループ内で呼び出します

   ループ内の関数呼び出しは、コンパイラがインライン化できない場合、ソフトウェア パイプラインを防止します。この問題を解決するには、次のいずれかの方法で呼び出された関数を定義します。

   - inline キーワードを使用して、コールと同じソース ファイルに格納します
   - #include を介して含まれる .h ヘッダ ファイルで、静的インラインを使用します

   注: コンパイラは `--opt_level=3` と `--opt_level=4` で自動インライン化を実行するため、高い最適化レベルで手動で変更しなくても、呼び出しによってはインライン化されることがあります。
2. トリップ数が不明です

   ループの実行回数に関する情報をコンパイラに提供することで、ベクトル化とループ最適化が有益かどうか、および冗長ループが必要かどうかを判断できます。

   ソフトウェア パイプライン ループは、合法的に実行するために最小限の反復回数を必要とします。コンパイラが実際のイテレーション カウントが常にこのしきい値を満たすことを保証できない場合、ループの 2 つのバージョン (高速パイプライン バージョンと低速冗長フォールバック) が生成されます。アセンブリのコメント ブロックは、これが発生すると、`;* Redundant loop generated` とステートし、安全な反復回数の最小値を指定します。

   `#pragma MUST_ITERATE` を使用して、既知の干渉境界を指定します。

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

   最小値はアプリケーション自体から得られます。アルゴリズムが常に固定ブロック サイズを処理する場合、それは保証された最小値です。たとえば、オーディオ アルゴリズムはブロックごとに 128 サンプルを処理し、常に少なくとも 128 回ループを実行します。プラグマを追加した後、アセンブリをチェックして`;* Redundant loop generated` が存在しないことを確認します。誤った値を指定すると、コンパイラは誤った動作を発生させるコードを生成することに注意してください。
3. ポインタのエイリアス/ループ搬送依存関係

   もしコンパイラが入出力配列が別々のメモリ領域を占有していると判断できない場合、I+1 の負荷に影響を与える可能性があることを前提にしなければなりません。これにより、ソフトウェア パイプラインを妨げるループ搬送依存性が生成されます。これを解決するには、オーバーラップしないことが保証されているポインタ パラメータに restrict キーワードを使用します。

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

   注

   restrict を誤って使用すると (重複するポインタ上で)、コンパイラは未定義の動作と誤った結果を持つコードを生成することがあります。
4. ループがベクトル化されていません

Copyright © 2026 Texas Instruments Incorporated

コンパイラはループを完全にベクトル化できない場合や、その中の一部の演算だけをベクトル化してしまう場合があり、一部の命令がベクトル化され、他の命令がスカラー化された非効率的なミックスループになります。

- ループがまったくベクトル化されていない場合は、ソースで直接 C7x ベクトル型を使用することを検討してください
- 部分ベクトル化が問題である場合は、ベクトル型と組み込み関数を使用して、完全なループを手動でベクトル化する必要がある場合があります。

注：詳細については、『C7000 C/C++ 最適化ガイド』(SPRUIG8) を参照してください。

3 メモリ キャッシュ設定をダブル チェックします

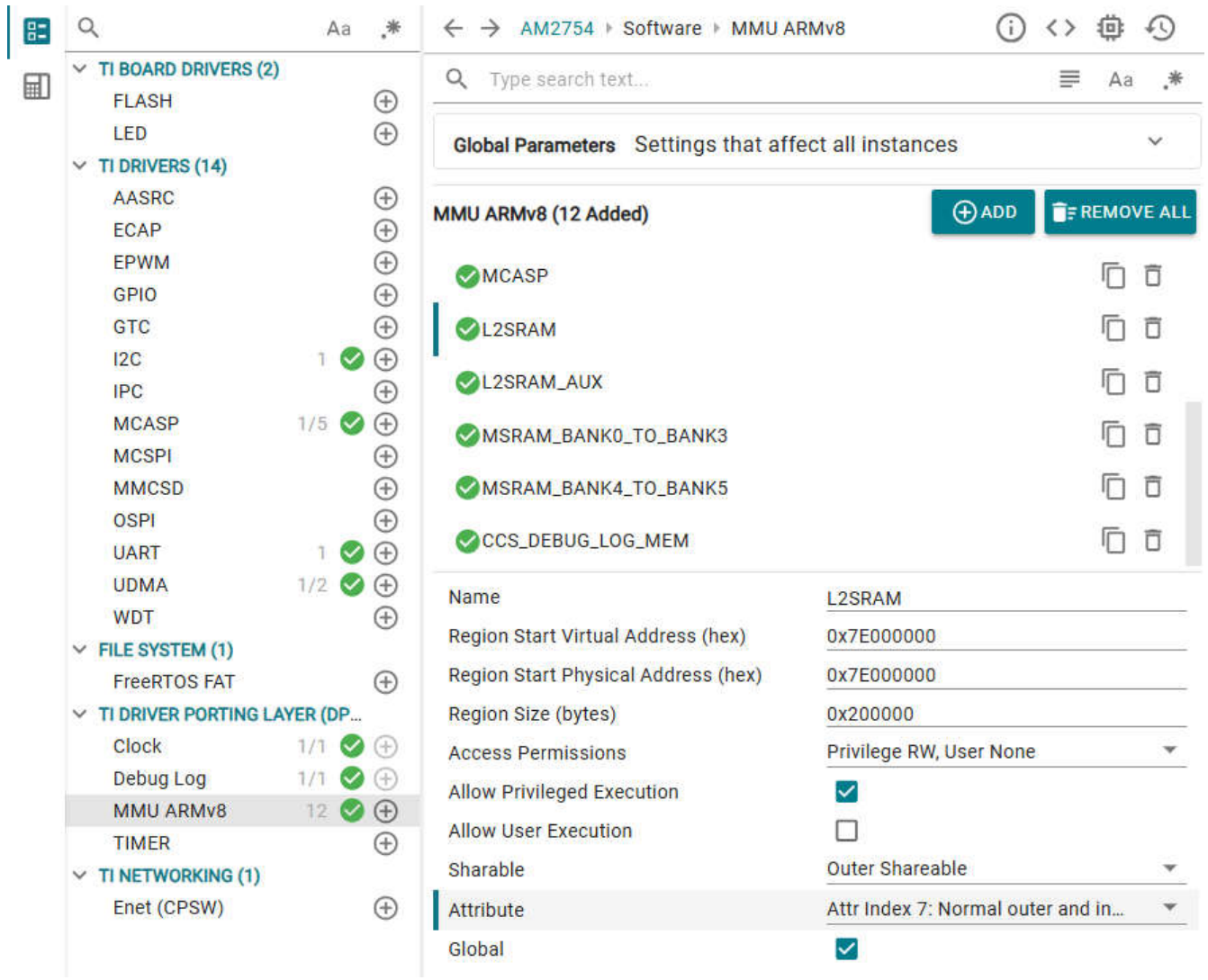
ライトスルーの代わりにライトバック キャッシュ設定を使用すると、最高のパフォーマンスが得られます。サンプル プロジェクトはそのままで構成されますが、キャッシュが正しく構成されていない場合、パフォーマンスに大きな変化が生じる可能性があります。

この違いを要約すると、キャッシュ ライトバック モード (write-deferred とも呼ばれます) は、キャッシュから削除されるまで、アドレスへの書き込みのみをキャッシュに行うモードです (元のメモリロケーションは同期しないままにします)。キャッシュの内容は、元のメモリ ロケーションにコピーされます。キャッシュは通常、他の形式のメモリよりも高速であるため、DSP はこのデータの書き込みと読み取りをより高速に行うことができます。この状況のリスクは、メモリが複数のコアで共有されている場合、別のコアが有効なデータを認識できるように、データをメモリ位置に書き戻す必要があることです。キャッシュ ライトバック コールを使用すると、キャッシュのパフォーマンスを向上させたまま、必要な場合にのみこれを強制的に実行できます。

キャッシュ ライトスルー モードは異なり、メモリ ロケーションへの書き込みはキャッシュと元のメモリ ロケーションの両方で同時に更新されます。アプリケーションは低速のメモリが書き込まれるのを待つ必要があるため、書き込み時のパフォーマンスが低下します。読み取りはキャッシュから行われますが、キャッシュとメモリの場所はすべての書き込みで同期されているため、書き込み速度は遅くなります。

CCS プロジェクトの内部には、使用するハードウェア設定を含む SysConfig ファイル (最後は .syscfg) があります。SysConfig ファイルの中には、MMU または MMU ARMv8 というセクションがあります。このセクションでは、マップされたすべてのメモリ位置を示します。

Copyright © 2026 Texas Instruments Incorporated



セクションをクリックして選択します (通常、L2SRAM セクションまたは MSRAM セクションはデータが格納されています)。属性フィールドがライトバック キャッシュ可能に設定されていることを確認します (Attr Index 7)。一部の特殊なアプリケーションでは、他の設定が必要になる場合があります。たとえば、ライトバックを使用する場合、1 つのコアが使用するメモリをプロセッサ キャッシュにキャッシュできます。ライトバック キャッシュ可能では、データが更新されると、キャッシュ内でのみ更新され、行がキャッシュから消去されるまで元のメモリ位置では更新されません。これにより、プロセス担当者がバッファにいくつかの変更を加え、変更がキャッシュにのみ存在し、元のメモリの場所には存在しないというシナリオが作成される可能性があります。これにより、元のメモリ位置から古いデータがプルされる可能性があります (たとえば、アルゴリズムがデータを更新した後で MCASP トランザクション)。メモリ位置の更新が完了し、別のハードウェア ペリフェラルまたはプロセッサがある元の場所に同期する必要がある場合、アプリケーションは `cacheP_inv()` 関数を呼び出して、キャッシュの書き込みを強制的に実行し、元のメモリ位置がキャッシュの値で更新されるようにする必要があります。

Copyright © 2026 Texas Instruments Incorporated

4 既存の最適化ライブラリを使用します

FreeRTOS SDK/MCU Plus SDK の一部として、高度に最適化されたライブラリがいくつか提供されており、それらを使用して性能をさらに向上させることができます。これらのライブラリは、C7x アーキテクチャ向けにすでに最適化済みのコードと、一般的な機能を提供します。数学、デジタル信号処理、ビジョン処理、オーディオ処理に適したライブラリがあります。

これらの最適化された関数を利用することで、アルゴリズムの性能をさらに向上させることができます。C7x アーキテクチャはベクトルベースであり、複数のデータ ブロックで同時に動作できることに注意してください。一般に、ベクトル アーキテクチャを利用するためにデータのサイズ (ブロック サイズ) が増加すると、パフォーマンスが向上します。

これらの各種ライブラリに関する資料の場所は、SDK フォルダ内にあります。たとえば、`<SDK base directory>/source/dsplib/docs/`。提供されたカーネル用のサンプル コードも提供されています。これらの最適化された DSP 関数は、ストリーミング エンジンやベクトルなどの C7x アーキテクチャのいくつかの機能を活用するよう最適化されているため、可能な限り使用することが一般的に推奨されます。これらの関数は、実際の計算を実行するアルゴリズムのセクションで性能を大幅に向上させることができます。

これらの最適化された関数で注意すべき重要な点の 1 つは、通常、init() 関数と exec() 関数で使用されることです。init 関数は、ストリーミング エンジンやアドレス ジェネレータなどの 1 回限りの設定に使用されます。exec 関数は計算に使用され、ループ内で呼び出される必要があります。init 関数は、一度だけ、またはパラメータが変更されたときにのみ呼び出す必要があります。理想的には、init 関数が 1 回呼び出され、exec() はデータ ブロックが完全に処理されるまでループされます。これは最大のパフォーマンス リフトを提供します。

提供されたライブラリを利用する上で、パフォーマンスを向上させるためにも、それらを正しく使用する必要があります。例えば、2 つの配列を一緒に追加する [追加] 関数が一度に 1 つの要素しか処理しない場合、パフォーマンスの向上はありません。256 ビット ベクトル幅アーキテクチャを採用した AM275 の場合、1 クロック サイクルで最大 8 つの 32 ビット値を加算できますが、この関数を正しく呼び出すと、単一のスカラ データではなくベクトル幅データを処理できます。ライブラリに提供されている例は、これらの関数を正しく呼び出す方法を示しています。

5 カスタム カーネル/モジュールの最適化

上記のすべての手順を実行した後、最適化の最後のステップの 1 つは、提供されているどのライブラリにも現在存在しないカスタムモジュールを移行することです。これを行うには、C7x アーキテクチャと、性能を向上させるために追加のペリフェラルに関する知識が必要です。詳細については、『C7000 C/C++ コード最適化ガイド』内で詳しく解説されています。まず「基本的なコード最適化戦略と手法」が紹介され、続いて「高度な手法」が解説されています。C7x アーキテクチャの開発開始に役立つ、ビデオ トレーニング シリーズなど、オンラインでいくつかのリソースをご用意しています。

Copyright © 2026 Texas Instruments Incorporated

6 例

前述した手順を使用すると、提供されているサンプル アルゴリズムを確認し (github で提供されているサンプル コードを参照)、そのプロセスを進めることができます。ここでは、いくつかのカーネルを利用した簡単なサンプル アルゴリズムを使用します。

この例のアルゴリズムは単純なオーディオ アルゴリズムです。システムのパラメータは次のとおりです。

1. 8 チャンネルのオーディオ (48kHz)、16 ビット サンプル (Q15)、インターリーブ データ
2. ブロックサイズ:チャネルごとに 128 サンプル (1,024 サンプル合計)
3. 入力および出力サンプルは符号付き 16 ビット サンプル (Q15)

この仮想システムの例では、以下のような単純なアルゴリズムが使用されます

1. 16 ビット サンプルを浮動小数点に変換します
2. 3 段カスケード バイクワッド ロー パス フィルタ
3. グローバル ゲイン 0.75 が適用されます
4. DAC/アンプ で使用するため、16 ビット サンプルに戻ります

このサンプル システムは、DSP 用に最適化されたコードを持たない、最も基本的な「自然 C」コードから開始します。ここでは、前述の各ステップを実行し、AM275 EVM をテスト プラットフォームとして使用して、C7000 での性能向上状況を確認します。

6.1 システム結果の概要の例

ステップ全体を通じて性能の向上を確認すると、C7000 DSP 向けにコードを最適化するためにある程度の時間を費やして性能を大幅に向上させる方法を示しています。

表 6-1. エフェクト コンパイラ最適化フラグ

コードタイプ	-O フラグ	合計サイクル数	Q15 からフロート サイクル	カスケード バイクワッド サイクル	ゲイン サイクル	Q15 サイクルまでのフロート	サンプルあたりのサイクル数	コメント
自然 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	

概要:上記のデータを見ると、コンパイラの最適化フラグ (-O) の効果により、アルゴリズムのすべてのモジュールのパフォーマンスが大幅に向上し、場合によっては最適化された DSPLib カーネルのパフォーマンスと一致することができます。これは、コンパイラがパターンを探してベクトル幅計算を設定できるため、高速化が可能です。

すべてのカーネル モジュールを手動で最適化した DSPLib カーネルに置き換えると、最適な自然な C コードを超える性能が大幅に向上します。最適化された C7000 ライブラリを使用すると、パフォーマンスがサンプルあたり 173.7 クロック サイクルからサンプルあたり 3.5 クロック サイクルに向上します。

Copyright © 2026 Texas Instruments Incorporated

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

Copyright © 2026, Texas Instruments Incorporated

最終更新日：2025 年 10 月