XAPP1206 (v1.1) 2014 年 6 月 12 日

NEON を使用して Zynq-7000 AP SoC でのソフトウェア性能を向上

著者 : Haoliang Qin

概要

ザイリンクス Zynq®-7000 All Programmable SoC は、エンベデッド製品で広く使用されているデュアルコア ARM® Cortex™-A9 プロセッサを統合するアーキテクチャです。ARM Cortex-A9 コアはどちらも NEON とも呼ばれる、高度な単一命令複数データ (SIMD) エンジンを備えています。これは、大型データセット上のパラレル データ演算に特化されています。このアプリケーション ノートでは、NEON を使用してソフトウェア性能とキャッシュ効率の向上を図り、性能全体を向上させる方法について説明します。

はじめに

一般的に、CPU は命令を実行してデータを処理する動作を 1 つずつ実行します。通常、高い性能は高クロック周波数を使用することで達成しますが、この点で半導体技術には限界があります。このため、CPU データ処理能力を向上する次の手段として採用されるのが、並列演算です。SIMD テクニックを取り入れることにより、CPU の 1 サイクルまたはわずかなサイクル数で、複数のデータが処理できるようになります。この SIMD を ARM v7A プロセッサに実装しているのが NEON です。NEON の有効な活用により、ソフトウェア性能の大幅な向上を期待できます。

内容

このアプリケーション ノートでは、次の技術情報を提供します。

「導入 : 重要なコンセプト」

コードの効率的な最適化に必要な次について説明します。

- 「ソフトウェア最適化の基本」
- 「NEON の基礎知識」

「ソフトウェア性能最適化方法」

NEON を使用してソフトウェア性能を最適化する 4 つの方法を説明します。

- 「NEON 最適化ライブラリを使用する方法」

  Cortex-A9 がエンベデッド デザインで広く採用されていることから、多数のソフトウェア ライブラリが NEON 向けに最適化されており、性能が向上しています。ここでは、このように頻繁に使用されるライブラリを取りあげます。
- 「コンパイラの自動ベクトル化を使用する方法」

  オープン ソース コンパイラとして多用されている GCC は、適切なコンパイル オプションを設定すると NEON 命令を生成できます。ただし、C 言語は、並列演算の記述に優れているとはいえません。C コードを変更してコンパイラ ヒントの追加が必要となる場合があります。「ラボ 1」に実践例を示しています。
- 「NEON イントリンシクスを使用する方法」

  通常、単純な最適化 (レジスタ割り当て、命令スケジューリングなど) はコンパイラが問題なく処理します。複雑なアルゴリズムの解析や最適化をコンパイラが処理できない場合、NEON イントリンシクスの使用が必要となる場合があります。さらに、NEON 命令の一部は同等な C 演算式がなく、イントリンシクスまたはアセンブリが唯一の選択肢となります。このため、状況によっては、性能を向上するために、NEON イントリンシクスのタイム クリティカルなコードを変更することが必要な場合もあります。「ラボ 2」に実践例を示しています。

- 「NEON アセンブラ コード最適化する方法」

  状況によっては、コンパイラが最高性能のバイナリを生成できない場合があります。このような場合、NEON アセンブラ コードでファンクションを記述し、制御する必要があります。「ラボ 3」に実践例を示しています。

「メモリ アクセス効率向上による NEON 性能の向上」

NEON の性能について検討する場合、通常、メモリ サブシステムが CPU の速度に対応できることが前提です。つまり、データや命令すべてが L1 キャッシュを使用すると想定していますが、データセットが非常に大きい場合は、必ずしもそうとは限りません。ここでは、キャッシュの効率を向上する 3 つの方法を検討します。

- 「複数データのバースト ロード/格納」
- 「キャッシュ ヒット率向上にプリロード エンジンを使用」
- 「タイルを使用してキャッシュ スラッシングを防止」

ここに示す方法は、NEON コードだけではなく、ソフトウェア コード全般にメリットをもたらします。

前提条件

このアプリケーション ノートでは、ザイリンクス ソフトウェア開発キット (SDK) の使用方法 (新規プロジェクトの作成、ソースファイルのインポート、デバッグ、ハードウェアでスタンドアロン アプリケーションを実行) を理解していることを前提としています。

サンプル ソフトウェアとラボ

このアプリケーション ノートには、サンプル ソフトウェアおよびラボが含まれます。次のリンクから、ZIP ファイルをダウンロードしてください。
https://secure.xilinx.com/webreg/clickthrough.do?cid=359072

表 1 に、デザインの詳細を示します。

表 1 : デザインの詳細

パラメーター	説明
一般	
ターゲット デバイス	全 Zynq-7000 デバイス
ソース コードの提供	あり
ソース コードの形式	C およびアセンブラ
既存のザイリンクス アプリケーション ノート/リファレンス デザイン、CORE Generator™ ツール、サード パーティからデザインへのコード/IP の使用	なし
検証	
ハードウェア検証	あり
ソフトウェア開発ツール	SDK 2013.4
使用したハードウェア プラットフォーム	ZC702 評価キット

導入：重要なコンセプト

最適化について検討する前に、理解しておくべき重要な関連コンセプトについて説明します。

ソフトウェア最適化の基本

エンベデッド システムでは、最適化の目的として多いのは高速化ですが、その他にバッテリの寿命、コードの密度、メモリのフットプリントを最適化する場合もあります。一般に、エンベデッド システムは、特定の目的で高度にカスタマイズされたハードウェアやソフトウェア システムを使用します。ほとんどの場合、コード密度、スピード、デバッグ可視性などといった要素の間に存在するトレードオフを考慮する必要があります。このアプリケーション ノートでは、ソフトウェアを高速化する方法について説明します。高速化には、性能向上というメリットが付随し、消費電力削減やバッテリ寿命の面からも重視されます。1 つのタスクが数回のサイクルで終了すると、電力が不要な時間が長くなります。

ソフトウェアを最適化して性能を向上する方法として、たとえば、次のような方法があります。

- アルゴリズムの演算順序を変更して、キャッシュ スラッシングを回避し、キャッシュ効率を改善します。この方法は、通常、行列演算に必要です。このアプリケーション ノートに、その実施例を示しています。
- 下位のプロセッサ アーキテクチャの使用効率を向上するアルゴリズムを開発します。対象は、命令に限定されません。ここでは、コードを最適化する上で CPU ハードウェアを理解しておくことの重要性について説明します。このような知識は、デザイン アルゴリズムの最適化や効率の向上につながります。

Zynq-7000 AP SoC に使用されている ARM Cortex-A9 プロセッサには、ソフトウェア性能を向上する高度な機能が多数装備されています。

- スーパースカラー、アウトオブオーダー /投機的実行
- データを事前に読み込むプリロード エンジン
- データ処理を向上するメディア処理エンジンおよび VFPv3
- パイプライン処理のストールを抑制するヒットアンダーミス動作
- コンテキスト スイッチを高速化する PIPT (物理インデックス物理タグ) 付きのデータ キャッシュ

図 1 に、シングル コア Cortex-A9 プロセッサのブロック図を示します。

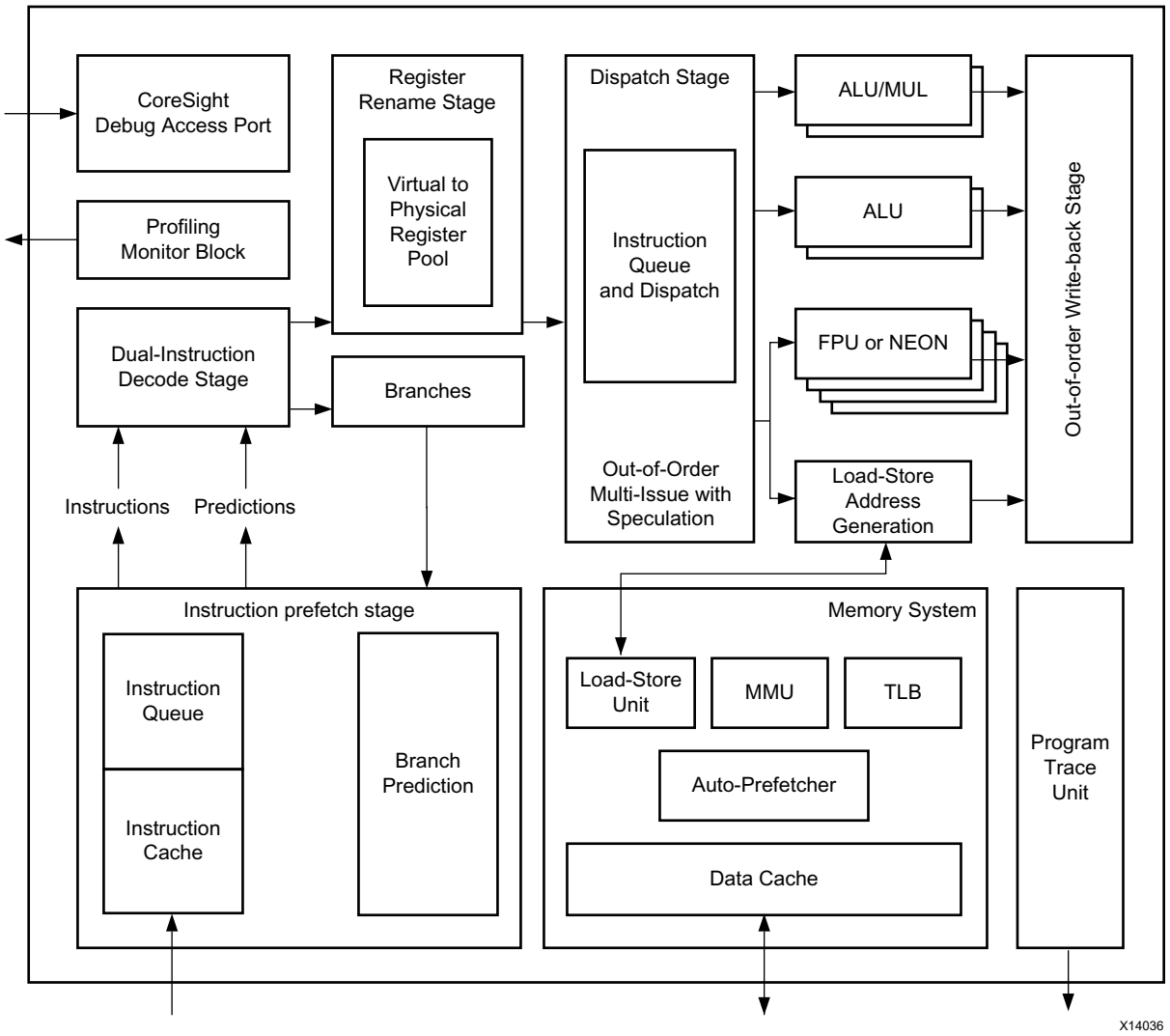

図 1 : シングル コア Cortex-A9 プロセッサのブロック図

Cortex-A9 プロセッサの潜在的な効果は、メモリ アクセスの順序を変更することにあります。ロードや格納命令が実行される順序は、必ずしも逆アセンブリで確認される順序と同じではありません。キャッシュ内のヒットアンダーミス動作は、キャッシュに入る (ヒットする) ロードが、その前にキャッシュをミスしたプログラムのロードよりも先に完了する可能性があることを示します。ロードや格納命令以外の命令でも、依存性がない限り、アウトオブオーダーで実行される可能性があります。このため、コード シーケンスを統計的に解析しても実用的ではありません。

NEON の場合、サイクル タイミング情報が ARM 資料に記載されていますが、どんなに小さいコードであっても必要なサイクル数を決定することは困難です。命令がパイプライン処理をどのように移動するかは、コンパイラが生成する命令シーケンスだけでなく、周囲の命令のパターンにも依存します。また、パイプライン処理の命令の移動に対して、メモリ システムも大きく影響を及ぼす可能性があります。

保留中のメモリ アクセス (データのロード/格納命令および命令フェッチなど) はキャッシュ ミスとして、これに依存する命令を何十回というサイクルにわたってストールさせる可能性があります。極端な例では、動的 RAM 競合またはリフレッシュ状態が存在すると、レイテンシは何百回ものサイクルまで増大する可能性があります。標準のデータ処理命令は通常、1、2 回のサイクルしかかかりませんが、多数の要因の影響を受ける可能性があり、理論的な数字は全体的な性能とは異なる場合があります。複雑なアプリケーション コードの性能を最適に評価するには、実システムで性能を測定することが必要です。

性能データの収集に十分な時間だけ高速でソフトウェアを実行するのがソフトウェア性能を向上する標準的な方法です。その一環として、プロファイリング ツールまたは Cortex-A9 プロセッサに内蔵されたシステム パフォーマンス モニターを使用します。収集されたデータを使用して、実行時間を長引かせているボトルネックやホットスポットを検出します。通常、このようなコードの量は限定されています。このように比較的少量のコードに重点を置き、全体的な性能を短時間かつ最低限の作業で向上させることができます。

ボトルネックを検出するため、通常、プロファイリング ツールが必要になります。プロファイリング ツールの高度な使用方法は、このアプリケーション ノートの範疇ではありませんが、有効なオプションとして、ソリューションをいくつか示します。

Gprof

Gprof は、C/C++ アプリケーションを簡単にプロファイルできる GNU ツールです。使用するには、`-pg` フラグを指定した GCC を用いてソース ファイルをコンパイルする必要があります。行ごとのプロファイルには、`-g` オプションも必要です。これらのオプションは、実行時に関数の入口と出口でデータを収集するプロファイラーを追加します。次に、コンパイルされたプログラムを実行し、プロファイリング データを生成します。この後、gprof がデータを解析し、有用情報を生成します。

Gprof を使用してプロファイルできるのは、`-pg` オプションを使用して再ビルドされているコードに限定されます。プロファイリングを目的としてビルドされていないコードには適用できません (たとえば、`libc` またはカーネル)。オペレーティング システム カーネルの制限または I/O のボトルネック (メモリ フラグメンテーションまたはファイル アクセスなど) はプロファイリング結果に影響を及ぼす可能性があることに注意してください。

OProfile

OProfile は、Linux システム全体をプロファイルするツールであり、カーネルがメトリックに含まれています。統計的なサンプリング方法を使用して、定間隔でシステムを試験しながら実行中のコードを判断し、適切なカウンターを更新します。割り込みを使用するため、割り込みを無効にするコードでは結果が不正確になる可能性があります。

また、OProfile は、ハードウェア イベントでトリガーし、カーネルやライブラリ コードの実行など、システム アクティビティをすべて記録するように設定可能です。OProfile は、特定のフラグを指定してコードをコンパイルし直す必要はありません。Cortex-A9 パフォーマンス モニター ユニット (PMU) を使用して、クロック サイクルやキャッシュ ミスなど、重要なハードウェア情報を提供します。

ARM DS-5 Streamline

ARM DS-5 Streamline は、Linux または Android システムで性能を解析する GUI ベースのツールです。ARM DS-5 の一部で、Linux カーネル ドライバー、ターゲット デーモン、Eclipse ベースの UI から構成されます。

DS-5 Streamline は、周期的にシステムをサンプルし、統計手法でデータをグラフ表示します。プロセッサ イベントのハードウェア カウンターを備えるハードウェア パフォーマンス モニター ユニットと、アプリケーション情報をトレースする Linux カーネル メトリックの両方を使用します。

プロファイリングの問題として、オペレーティング システムが準備状態であることを要求する場合があります。最適化の対象が特定のアルゴリズムのみであったり、ボトルネックの場所が明確であったりする場合は、タイム クリティカルなコードを抽出し、スタンドアロン モードで実行できます。コード シーケンスをいくつか試すことで、最適なコードを特定できます。通常、この方法には高精度のタイマーが必要です。

コードの最適化にスタンドアロン モードを使用することには、次のようなメリットがあります。

- 簡単で便利
- オペレーティング システムからの干渉を排除
- 所要時間を短縮

ARM DS-5 Streamline ツールを使用する際は、時間を正しく計測することが重要です。通常、これには高精度なタイマーが必要です。Cortex-A9 プロセッサには、コアすべてに対応する 1 つのグローバル タイマー (64 ビット) と各コアに 1 つのプライベート タイマー (32 ビット) があります。タイマーのプリスケーラーは、タイマーの精度と引き換えに、タイマー クロック レートを低下させ、オーバーフロー時間を増大させます。実際のタイマー クロックは、次の式で計算できます。

$$\frac{PERIPHCLK}{PRESCALERvalue + 1}$$

Zynq-7000 AP SoC では、PERIPHCLK が CPU クロック周波数の半分であることから、タイマーの最高精度は 2ns になります。ソフトウェア開発を容易にするよう、ザイリンクスはこれらのタイマーの API も提供しています。

NEON の基礎知識

ソフトウェア性能の向上に NEON を使用する理由と方法について説明します。

NEON 技術は、ソフトウェア面で ARMv7 プロセッサの単一命令複数データ (SIMD) 演算を基本とし、高度な SIMD アーキテクチャ拡張を実装します。このため、新しい関数を含む新しい命令セット、さらに新しい開発方法論が求められます。ハードウェア面では、Cortex-A シリーズ プロセッサ上の独立したハードウェア ユニットで、ベクター浮動小数点 (VFP) 処理ユニットと組み合わされています。専用ハードウェアの利用が可能になるようにアルゴリズムを設計すると、最大限の性能が発揮されます。

SIMD の概要

SIMD は、単一命令を使用して並列で多数のデータ値 (一般に 2 の累乗) を処理する演算技法であり、オペランドのデータが特殊なビット幅の広いレジスタにパックされています。したがって 1 つの命令が、単一命令単一データ (SID) アーキテクチャ上で多数の独立した命令を実行できます。並列化が可能なコードでは、性能の大幅な向上を達成できます。

ソフトウェア プログラムの多くは、大型データセットで演算を実行します。データセットの各要素は、32 ビット未満です。8 ビットのデータは、ビデオ、グラフィックス、画像処理で、16 ビットのデータはオーディオ コーデックがそれぞれ一般的です。したがって、実行される演算は単純で、何回も繰り返され、制御コードの必要性はほとんどありません。SIMD は、この種のデータ処理で性能の大幅な向上を実現できます。特に、次のようなDSP デジタル信号処理またはマルチメディア アルゴリズムに役立ちます。

- FFT、行列乗算など、ブロックベースのデータ処理
- MPEG-4、H.264、On2 VP6/7/8、AVS など、オーディオ、ビデオ、画像処理コーデック
- 矩形ブロックのピクセルをベースとする 2 D グラフィックス
- 3 D グラフィックス
- 色空間変換
- 物理シミュレーション
- リード ソロモン コーデック、CRC、楕円曲線暗号など、誤り訂正

Cortex-A シリーズ プロセッサのような 32 ビット マイクロプロセッサでは、大量の 8 ビットまたは 16 ビット演算は比較的非効率です。プロセッサ ALU、レジスタ、データパスは 32 ビット演算用に設計されています。8/16 ビット演算に使用する場合、オーバーフローの処理に命令を追加する必要があります。SIMD は、単一命令が複数のデータ要素として 1 つのレジスタ値を処理し、これらの要素上で複数の同一演算を実施できるようにします。

図 2 に、SIMD 並列加算と 32 ビット スカラー加算の比較を示します。

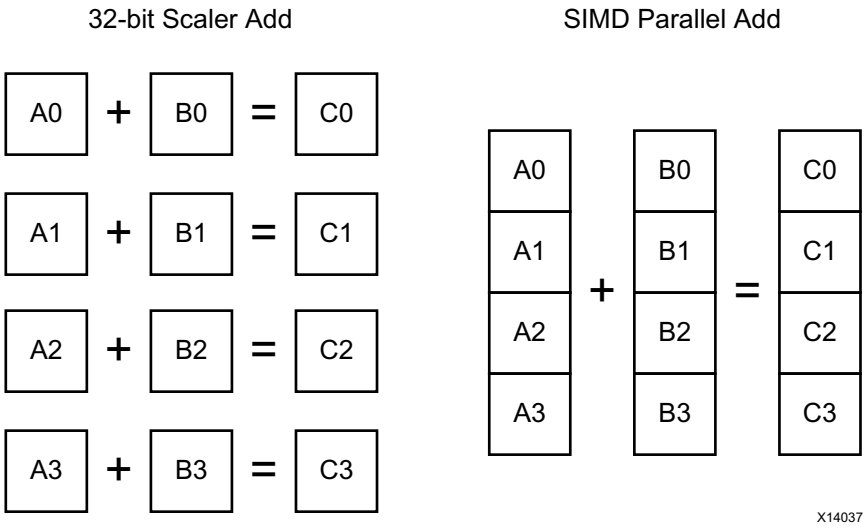

図 2 : SIMD 並列加算と 32 ビット スカラー加算の比較

スカラー演算を使用して 4 つの独立した加算を実行するには、図 2 に示すように 4 つの加算命令を使用し、さらに 1 つの結果が隣接するバイトにオーバーフローしないようにする命令を追加する必要があります。SIMD では、命令を 1 つしか必要とせず、オーバーフローを管理する必要もありません。さらに、専用 ALU を備えているため、SIMD 命令が必要とするサイクル数は、一般に同じ関数に対して ARM 命令よりも少なくなります。

レジスタ

NEON アーキテクチャでは、64 ビットまたは 128 ビットの並列処理が可能です。レジスタ バンクは、16 個の 128 ビット レジスタ (Q0 ～ Q15) または 32 個 の 64 ビット レジスタ (D0 ～ D31) と考えることができます。Q0 ～ Q15 のレジスタはそれぞれ、D レジスタのペアにマップされます。

図 3 に、NEON レジスタ バンクを示します。

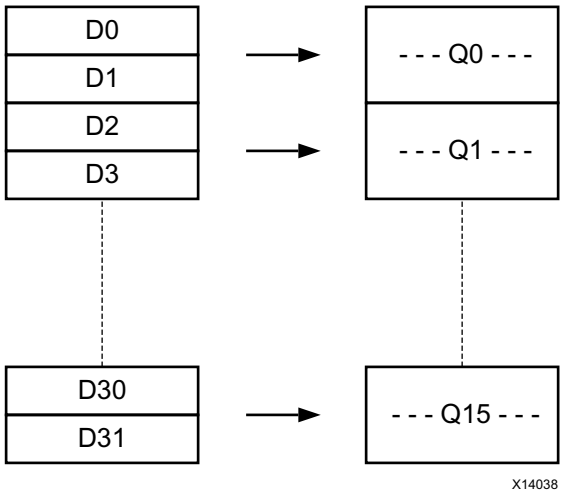

図 3 : NEON レジスタ バンク

NEON と VFP

Zynq-7000 プラットフォームには、NEON と VFP の両方が統合されています。NEON はベクターのみを処理しますが、VFP は「ベクター」という名前が付けられているにもかかわらずベクターを処理しないところが NEON と VFP の主な違いです。実際、Cortex-A9 プロセッサでは浮動小数点処理ユニット (FPU) と呼ぶほうがふさわしいでしょう。浮動小数点の演算の場合、VFP は、単精度と倍精度の両方を

サポートできますが、NEON は単精度しかサポートしません。また、VFP は、平方根や除算などの複雑な関数に対応できますが、NEON は対応しません。

NEON と VFP は、ハードウェアで 32 個 の 64 ビット レジスタを共有します。つまり、VFP は VFPv3-D32 と表され、32 個の倍精度浮動小数点レジスタがあります。このため、コンテキスト スイッチのサポートが簡単になります。VFP コンテキストを保存および復元するコードは、NEON コンテキストも保存し、復元します。

データ型

NEON 命令のデータ型指定子は、データの型を示す文字と幅を示す数字から構成されます。ピリオドで命令ニモニックと区切られています。次のオプションがあります。

- 符号なし整数 U8 U16 U32 U64
- 符号付き整数 S8 S16 S32 S64
- 型指定のない整数 I8 I16 I32 I64
- 浮動小数点数 F16 F32
- {0,1} を超える多項式 P8

NEON 命令

NEON 命令のニモニックはすべて (VFP と同様に) 文字 V で始まります。この文字によって、ARM/Thumb 命令と区別されます。このインジケーターを使用すると、コンパイラの効率を確認する際に逆アセンブリ コードで NEON 命令を検出できます。次の例に、NEON 命令の一般的な形式を示します。

```
V{<mod>}<op>{<shape>}{<cond>}{.<dt>}(<dest>}, src1, src2
```

説明：

`<mod>`	修飾子 (Q、H、D、R) の 1 つ
`<op>`	演算 (たとえば、ADD、SUB、MUL)
`<shape>`	形状 (L、W、N) [参照 4]
`<cond>`	条件、IT 命令で使用
`<.dt>`	データ型
`<dest>`	デスティネーション
`<src1>`	ソース オペランド 1
`<src2>`	ソース オペランド 2

NEON 命令の詳細は、『NEON Programmers Guide』[参照 3] を参照してください。NEON 命令に関する知識が重要になる理由は次のとおりです。

- NEON 命令は、アルゴリズムの設計時に最大領域となるように使用する必要があります。命令シーケンスで関数をエミュレートすると、性能が大幅に低下する可能性があります。
- コンパイラが最適コードを生成できないことがあるため、逆アセンブリを理解し、生成されたコードが最適であるかの判断が必要な場合があります。
- 飽和算術、インターリーブされたメモリ アクセス、テーブル ルックアップ、ビット単位のマルチプレックス演算など、C 言語では表現できない演算子があります。このため、これらの命令では、イントリンシクスまたはアセンブラ コードの使用が必要な場合があります。
- タイム クリティカルなアプリケーションでは、最高の性能を実現するために NEON アセンブラ コードの作成が必要な場合があります。

詳細は、『ARM Architecture Reference Manual』[参照 2] および『Cortex-A Series Programmer's Guide』[参照 7] を参照してください。

NEON 性能の限界

NEON を使用してソフトウェア アルゴリズムを最適化する場合、期待できる性能向上の程度を判断することが重要です。

ARM 資料、『Cortex-A9 NEON Media Processing Engine Technical Reference Manual』[参照 4] には、VFP および NEON 命令それぞれが詳細に説明されています。表 2 および表 3 に、この資料で提供されている情報の一部をまとめました。

表 2 : VFP 命令のタイミング

名前	形式	サイクル	ソース	結果	ライトバック
VADD VSUB	.F Sd,Sn,Sm .D Dd,Dn,Dm	1	-,1,1	4	4
VMUL	.F Sd,Sn,Sm	1	-,1,1	5	5
VNMUL	.D Dd,Dn,Dm	2	-,1,1	6	6
VMLA	.F Sd,Sn,Sm	1	-,1,1	8	8
VMLS VNMLS VNMLA	.D Dd,Dn,Dm	2	-,1,1	9	9

表 3 : 高度な SIMD (NEON) 浮動小数点命令

名前	形式	サイクル	ソース	結果	ライトバック
VADD VSUB VABD VMUL	Dd,Dn,Dm	1	-,2,2	5	6
	Qd,Qn,Qm	2	-,2,2 -,3,3	5 6	6 7
VMLA VMLS	Dd,Dn,Dm	1	3,2,2	9	10
	Qd,Qn,Qm	2	3,2,2 4,3,3	9 10	10 11

「サイクル」の項目には、それぞれの命令に必要な発行済みのサイクル数が示されます。オペランド インターロックがない場合、これが絶対最小サイクル数になり、NEON 性能の上限を示します。たとえば、表 3 から、命令 VMUL/VMLA は、Q レジスタの演算をわずか 2 サイクルで完了できることがわかります。NEON は、4 つの単精度浮動乗算を 2 サイクルで実行できることになります。このため、NEON が 1GH z で稼動する場合に最大 2GFLOPS の単精度浮動小数点演算が可能です。表 2 から、VFP が 1 つの倍精度累積乗算を完了するには、2 サイクル必要であることもわかります。

NEON サイクルのタイミング情報は、ARM 社の資料に記載されていますが、わずかなコードであっても、実際のアプリケーションで必要なサイクル数を判断することは困難です。実際に必要な時間は、命令シーケンスだけでなく、キャッシュやメモリ システムにも依存します。

注記：アプリケーションで最も正確なデータを得るには、プロファイリング ツール (前述) を使用します。

NEON のメリット

エンベデッド システムで実現される NEON のメリットは、次のとおりです。

- 単純な DSP アルゴリズムで性能向上率が大きくなります (4～8倍)
- 一般に、複雑なビデオ コーデックで約 60～150% の性能向上率を期待できます
- ビット幅が広いレジスタでメモリのアクセス効率が向上します
- プロセッサがタスクを短時間で終了し、スリープ モードに早く入るために消費電力が削減されます

次に、NEON のプログラム方法を詳細に説明します。

ソフトウェア性能最適化方法

最適化する方法はいくつあります。

- 「NEON 最適化ライブラリを使用する方法」
- 「コンパイラの自動ベクトル化を使用する方法」
- 「NEON イントリンシクスを使用する方法」

NEON 最適化ライブラリを使用する方法

ARM Cortex A9 プロセッサは使用率が拡大したため、エンベデッド デザインで最も一般的なプラットフォームになっています。このプロセッサは、モバイル、タブレット、STB、DTV、テレコミュニケーション、産業用制御など、さまざまなアプリケーションで幅広く利用されています。ARM Cortex A9 プロセッサの使用が普及したことから、大規模なユーザー コミュニティが形成されており、ソフトウェア アルゴリズム設計者は NEON 最適化ソフトウェア ライブラリに関連するさまざまなエコシステムを活用できます。表 4 は、関連プロジェクトのいくつかを示しています。

表 4 : NEON 最適化オープン ソース ライブラリ

製品名	プロジェクトと内容
Project Ne10	ARM Expert 最適化ベクター、マトリックス、DSP 関数
OpenMAX DL サンプル ソフトウェア ライブラリ (ARM)	OpenMAX DL (開発層) のARM サンプル インプリメンテーション ソフトウェア ライブラリ。広範囲の高速化コーデックおよび演算アルゴリズムに使用可能
Google WebM	NEON アセンブラ最適化を含むマルチメディア コーデック
FFmpeg	オーディオやビデオの録音や録画、変換、ストリーミングの総合クロス プラットフォーム ソリューション
x264	ビデオ ストリームを H.264/MPEG-4 AVC 圧縮フォーマットにエンコードする無償のソフトウェアおよびアプリケーション
Android	Skia ライブラリ、S32A_D565_Opaque などのコンポーネントで s NEON を使用して 5 倍に高速化
OpenCV	リアルタイム コンピューター ビジョン向けのライブラリ
BlueZ	Linux 向けの Bluetooth スタック)
Pixman	ピクセル操作用の下位ソフトウェア ライブラリ。画像合成、台形のラスタライズなどをサポート
Theorarm	ARM プロセッサでの使用を目的として最適化された Ogg Theora/Vorbis デコード ライブラリ。Theora デコーダ (xiph.org が提供) と Tremolo ライブラリをベースとしている
Eigen	線形代数 (行列、ベクター、数値解法などを含む) 用 C++ テンプレート ライブラリ
FFTW	1 次元または多次元での拡散フーリエ変換 (DFT) の演算用 C ライブラリ。実データおよび複素数データを共にサポート

ここに挙げた以外のプロジェクトも多数進行中です。表 4 には、次の理由のため一部の一般的なプロジェクトのみ記載しています。

- 一部のプロジェクトは目的が特殊で、一般的な関心に欠けている。
- ますます多くのソフトウェア プロジェクトが NEON 最適化に着手している。OpenCV がよい例で、2.3 (2011 年 4 月 7 日) から NEON をサポート。
- ソフトウェア コミュニティが急速に展開し、新しいソフトウェア プロジェクトが頻繁にリリースされている。

このようなオープン ソース ライブラリだけでなく、表 5 にリストされている商用プロバイダーも NEON の開発を進めています。

表 5 : NEON サード パーティの開発プロジェクト

プロバイダー	プロジェクトと内容
Sasken Communication Technologies	H.264、VC1、MPEG-4
Skype	On2 VP6 ビデオ、SILK オーディオ (v1.08+)
Ittiam Systems	MPEG-4、MPEG-2、H.263、H.264、WMV9、VC1、DD
Aricent	MPEG-4、H.263、H.264、WMV9、オーディオ
Tata Elxsi	H.264、VC1
SPIRIT DSP	TEAMSpirit® ボイスおよびビデオ エンジン
VisualOn	H.264、H.263、S.263、WMV、RealVideo、VC-1
Dolby	マルチチャネル オーディオ処理、MS10/11
Adobe	Adobe Flash 製品
Techno Mathematical Co., Ltd. (TMC)	MPEG-4
drawElements	2D GUI ライブラリ
ESPICO	オーディオ：低ビットレートおよびデジタル シアター、コンサルティング
CoreCodec	CoreAVC、CoreMVC、CoreAAC、x264
DSP Concepts	NEON 最適化オーディオおよび信号処理ライブラリ
Ace Thought Technologies	NEON ビデオおよびオーディオ

コンパイラの自動ベクトル化を使用する方法

ここでは、次について説明します。

- GCC を使用して自動ベクトル化を有効にする方法 (ザイリンクス SDK および Linux 開発で使用)
- ソース コードのわずかな修正で、ソフトウェア性能を大幅に向上する方法

はじめに

NEON を最適化する最も簡単な方法は、コンパイラを使用する方法です。GCC には複数の最適化レベルがあり、さまざまなオプションを使用して特定の最適化を有効または無効に設定できます。

コンパイラの最適化レベルを設定するには、コマンド ライン オプション `-On` を次のように使用します。

- `-O0` (デフォルト)。最適化は実行されません。ソース コードの各行は、実行ファイルの対応する命令に直接マップされます。これはソース レベルのデバッグに最も適していますが、性能のレベルは最も低くなります。
- `-O1`。サイズやスピードに関する指定が不要な、最も一般的な形式の最適化を可能にします。ほとんどの場合、`-O0` よりも高速な実行ファイルが生成されます。
- `-O2`。命令スケジューリングなど、さらに高度な最適化を有効にします。この場合も最適化に関してスピードやサイズを考慮しません。
- `-O3`。関数のインライン展開など、アグレッシブな最適化を有効にします。一般に、イメージ サイズは大きくなりますが、高速化が実現されます。さらに、このオプションは `-ftree-vectorize` を有効にして、標準 C または C++ から NEON コードを自動的に生成させます。ただし、現実的にこの最適化レベルは必ずしも `-O2` よりも高速なバイナリを生成するとは限りません。ソフトウェア性能はそれぞれの状況で確認してください。
- `-Os`。このオプションでは、スピードよりもイメージ サイズの縮小を優先します。(これは、このノートの主旨ではありません。)

- `-Ofast`。厳密な標準コンプライアンスを無視します。`-Ofast` では、`-O3` の最適化オプションすべてが有効になります。また、一部の標準コンプライアンス プログラムで有効ではない最適化を有効にします。`-ffast-math` も有効にします。

最適化レベルに加え、コンパイラが NEON 命令を生成するようにコンパイラのオプションを設定する必要があります。

- `-std=c99`。C99 規格には NEON 最適化に使用できる新しい機能が追加されました。
- `-mcpu=cortex-a9`。ターゲットの ARM プロセッサの名前を指定します。GCC は、この名前を使用して、アセンブリ コードの生成時に発行できる命令の種類を判断します。
- `-mfpu=neon`。ターゲットで利用可能な浮動小数点ハードウェア (またはハードウェア エミュレーション) を指定します。Zynq-7000 デバイスには NEON ハードウェア ユニットが統合されており、これをソフトウェアの高速化に使用するため、`neon` という名前を用いて、コンパイラに明確に指示する必要があります。
- `-ftree-vectorize`。ツリー上でループのベクトル化を実行します。このオプションは、`-O3` ではデフォルトで有効になっています。
- `-mvectorize-with-neon-quad`。デフォルトでは、GCC 4.4 はダブルワードのみベクトル化します。ほとんどの場合、クワッド ワードの方がコードの性能や密度の点で優れていますが、使用可能なレジスタ多少減ります。
- `-mfloat-abi=name`。使用する浮動小数点 ABI を指定します。有効な値は、`soft`、`softfp`、`hard` です。
  - `soft` に設定すると、GCC は、浮動小数点演算のライブラリ呼び出しを含む出力を生成します。この値は、システム内にハードウェア浮動小数点ユニットがない場合に使用されます。
  - `softfp` に設定すると、ハードウェア浮動小数点ユニットを使用して命令を生成できますが、ソフト浮動小数点呼び出し表記規則が使用されます。これにより、互換性が向上します。
  - `hard` に設定すると、浮動小数点命令が生成でき、FPU 固有の呼び出し表記規則が使用されます。このオプションを使用する場合、同じ設定でソース コード全体をコンパイルし、リンクする必要があります。
- `-ffast-math`。このオプションは、`-Ofast` 以外、どの `-O` オプションを使用しても有効になりません。算術関数に関して、IEEE または ISO 規則/規格の正確な実装に依存するプログラムでは、不正な出力になる恐れがあるためです。ただし、これらの規格の保証を必要としないプログラムでは、コードの生成にかかる時間が短縮される場合があります。

実際には、最適化レベルを `-O2` または `-O3` に設定し、[Other optimization flags] (図 4 を参照) オプションを使用できます。ザイリンクス SDK で、プロジェクトを右クリックして、[C/C++ Build Settings] → [Optimization] をクリックすると、最適化関連に関連したフィールドが表示されます。

```
-mcpu=cortex-a9 -mfpu=neon -ftree-vectorize -mvectorize-with-neon-quad
-mfloat-abi=softfp -ffast-math
```

図 4 に、最適化フラグの設定を示します。

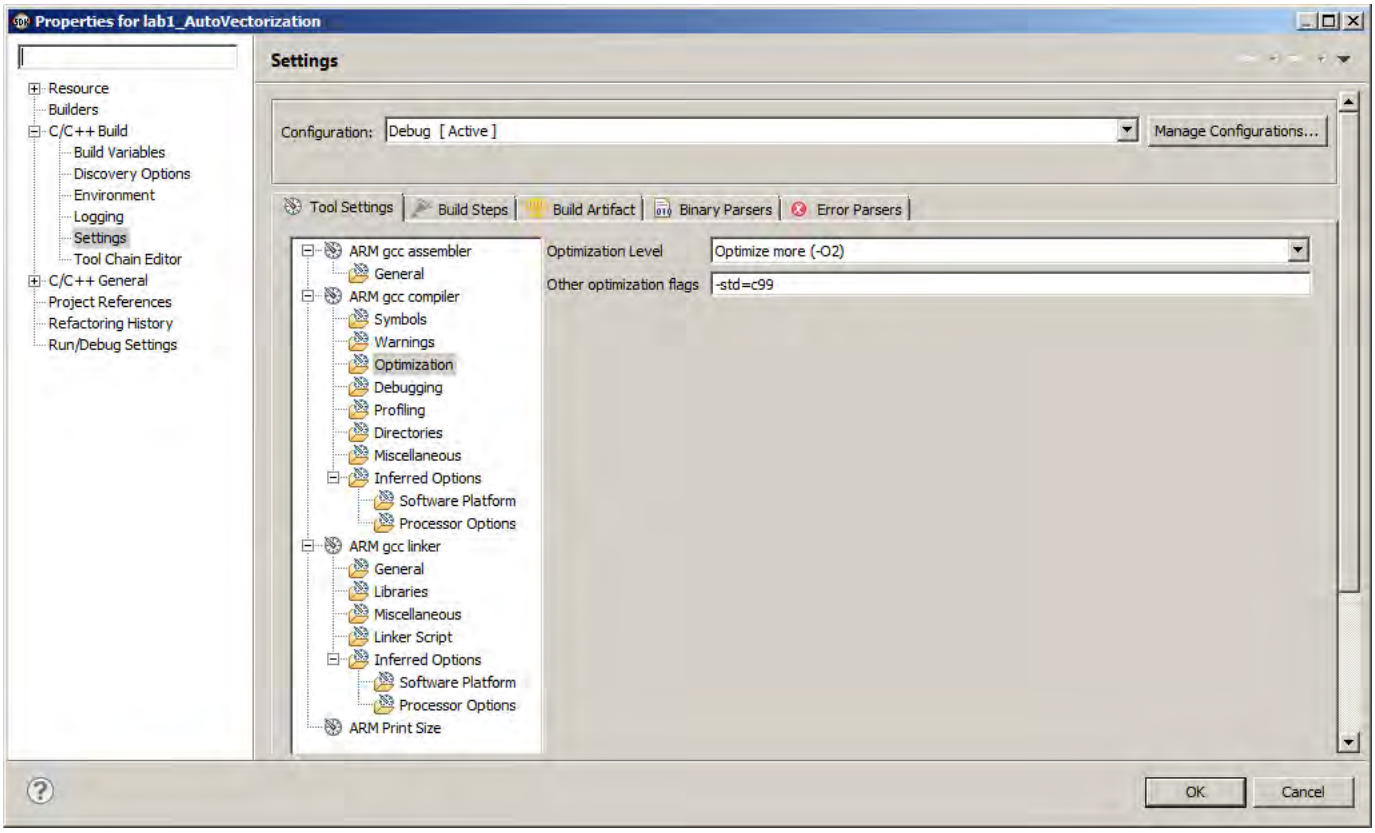

図 4 : 最適化フラグの設定

コンパイラは必ずしも意図したように C 言語をベクトル化しないため、適切な命令が生成されたことを確認する必要があります。

- 逆アセンブリを読み出します。これは最も直接的な方法ですが、NEON 命令の知識が必要です。
- コンパイル オプション `-ftree-vectorizer-verbose=n` を使用します。このオプションは、ベクトル化機能が出力するデバッグ出力量を制御します。この出力情報は、`-fdump-tree-all` または `-fdump-tree-vect` を指定していない限り、標準エラーに書き込まれます。これらのオプションを指定していると、通常のダンプ リスト ファイル、`.vect` に出力されます。n=0 の場合、診断情報はレポートされません。n=9 の場合、解析および変換中にベクトル化機能が生成した情報がレポートされます。これは、`-fdump-tree-vect-details` が使用するのと同じ冗長レベルです。

C コードの修正

C/C++ 規格は並列動作を指定する構文を提供しないため、コンパイラが NEON コードを生成するタイミングを判断することは困難です。十分な根拠がなければ、コンパイラはコードをベクトル化しないため、コンパイラにヒントを与えるようにコードを修正する必要があります。このようなソース コードの修正は、標準言語仕様の範囲であるため、プラットフォームやアーキテクチャのコード移植性に影響を及ぼすことはありません。

NEON に合わせてコードを修正する場合、次の方法を推奨します。

- ループの反復回数を示す
- ループの実行依存を避ける
- ループ内部の条件を避ける
- 制限キーワードを使用する
- 適切なデータ型を使用する

ここでは、標準ドット積アルゴリズムの例を使用します。この関数は、2 つの浮動小数点ベクターのドット積を計算します。ベクターにはそれぞれ浮動小数点型要素が len 個あります。

```
float dot_product(float * pa, float *  pb, unsigned int len)
{
    float sum=0.0;
    unsigned int i;

    for( i = 0; i < len; i++ )
      sum += pa[i] *pb[i];
    return sum;
}
```

ループの反復回数を示す

次の条件下ではコンパイラが安全に演算できるコードを作成可能です。

- ループの反復回数が固定されている
- 反復回数は、コーディング段階で N (レジスタ長/データ型サイズ) の倍数と特定できる

上記の例では、len の値が必ず 4 の倍数であるとわかっている場合、ループ カウンターを len と比較する際に下位 2 ビットをマスクして、4 の倍数であることをコンパイラに示すことができます。これにより、ループが必ず 4 の倍数回実行されるため、コンパイラはコードをベクトル化しても安全であると認識します。

> **注記**：反復回数が 4 の倍数というのは、一例に過ぎません。実際には、ベクターのレーン数の倍数になるはずです。たとえば、NEON クワッド ワード レジスタを使用し、データ型が 32 ビット 浮動小数点である場合には 4 の倍数の反復回数が理想的です。これは、次に示すコードの一部分の例 (14 ページ) にあるように、下位 2 ビットをマスクすることで示されます。
>
> 反復回数を固定、または反復回数をコーディング段階で特定するという要件は、必須ではありません。反復回数が実行時にしか判断できない場合は、ループを 2 つに分離します。一方のループでは、反復回数をレーン数の倍数とし、他方で残りの反復を処理するようにします。

ループの実行依存を避ける

コードに含まれたループで、ある反復の結果がその前の反復の結果の影響を受ける場合、コンパイラはベクトル化を実行できません。可能な場合、ループの実行に依存関係が生じないようにコードを再構築し、コンパイラがベクトル化を実行できるようにします。

ループ内部の条件を避ける

可能であれば、データはループ内部のみで処理します。一般に、コンパイラは、条件シーケンスを含むループのベクトル化には不向きです。ベスト ケースではループを複製しますが、多くの場合、このようなコードはまったくベク トル化されません。

制限キーワードを使用する

C99 には、`restrict` という新しいキーワードが追加されています。これによって、特定のポインターからアクセスされる場所は現在のスコープのほかのポインターからはアクセスされないと、コンパイラに伝えることができます。つまり、現在のスコープ内のポインターがターゲットとするメモリ領域が重複することはありません。

このキーワードを使用しなければ、コンパイラは、ポインター `pa` がポインター `pb` と同じ場所を参照していると想定する場合があります。これは、ループ実行依存の可能性を示唆しており、コンパイラがこのシーケンスをベクトル化できなくなります。`restrict` キーワードを使用することで、`pa` と `pb` が指しているメモリが重複していないと、コンパイラに示すことができます。コンパイラは、エイリアシングの可能性を無視し、エラーを発生させずにシーケンスをベクトル化できると想定します。

前述の方法を使用し、C ソース言語を次のようなスタイルに変更して、コンパイラが自動でベクトルかできるようにします。

```
float dot_product(float * restrict pa, float *  restrict pb, unsigned int
len)
{
  float sum=0.0;
```

```
  unsigned int i;

  for( i = 0; i < ( len & ~3); i++ )
    sum += pa[i] *pb[i];
  return sum;
}
```

GCC は、C99 でコンパイルしない場合、代替オプションとして __restrict__ および __restrict の形式をサポートします。コーディングに使用する規格を指定するには、オプション -std=C99 を使用します。その他の規格として、c90、gnu99 などがあります。

文献の中には、次の例のようにループを手動で展開すると、コンパイラの自動ベクトル化が簡単になると記載しているものがありますが、最新の GCC コンパイラは、手動で展開するよりもコードの認識や自動ベクトル化で優れています。実際、コンパイラは手動で展開されたループをベクトル化しない場合があります。

```
float dot_product(float * restrict pa, float *  restrict pb, unsigned int
len)
{
  float sum[4]={0.0,0.0,0.0,0.0};
  unsigned int i;
  for(i = 0; i < ( len & ~3); i+=4)
  {
    sum[0] += pa[i] *pb[i];
     sum[1] += pa[i+1] *pb[i+1];
    sum[2] += pa[i+2] *pb[i+2];
    sum[3] += pa[i+3] *pb[i+3];
  }
  return sum[0]+sum[1]+sum[2]+sum[3];
}
```

適切なデータ型を使用する

SIMD を使用せず、16 ビットまたは 8 ビット データでアルゴリズム演算を最適化する場合、データを 32 ビット変数のように処理すると、性能が向上する場合があります。これは、結果がハーフ ワードまたはバイト分オーバーフローしないように、コンパイラが追加の命令を生成する必要があるためです。

ただし、NEON を使用した自動ベクトル化を目的とする場合は、必要な値を保持できる最小のデータ型を使用するようにします。NEON エンジンは、一定時間内に 16 ビットの値の 2 倍の 8 ビット値を処理できます。また、NEON 命令がサポートしないデータ型や演算もあります。たとえば、NEON は倍精度浮動小数点データ型をサポートしないため、単精度浮動小数点で十分な場合に倍精度浮動小数点を使用すると、コンパイラがコードのベクトル化を妨げる可能性があります。NEON は、所定の演算しか 64 ビット整数をサポートしないため、可能な場合はロング変数を使用しないでください。

NEON には、構造化されたロード演算と格納演算を実行できる命令があります。これらの命令は、すべてが同じサイズであるデータ構造にベクトル化アクセスする場合にのみ使用可能です。これらの命令で、2/3/4 チャネルのインターリーブされたデータにアクセスすると、NEON メモリのアクセス性能を高速化できます。

NEON 演算結果の偏差

整数の場合、演算順序は重要ではありません。たとえば、整数の配列の和は、左右どちからでも必ず同じ結果が得られます。しかし、浮動小数点数では、コーディング精度のために同じ結果が得られるとは限りません。つまり、浮動小数点数の場合、NEON 最適化コードは最適化されていないコードとは結果が異なることがあります。ただし、通常、この差はわずかです。演算結果を比較してコードを検証する必要がある場合、浮動小数点またはダブルのデータ型の「等しい」は、差が許容されるものの、正確には同じものではないことに留意してください。

ラボ 1

1. SDK に新しいプロジェクトを作成します。
2. ラボ 1 のソース ファイルをインポートします。
3. ハードウェアでアプリケーションを実行し、コンソールで出力を確認します。
4. 生成された ELF ファイルを開いて、命令がどのように生成されているかを確認します。
5. 次の点に注意します。
   - 手動で展開したループは、ベクトル化され、さらに時間がかかります。
   - 自動的にベクトル化されたコードの実行時間は、最適化レベルが `-O3` に設定され、NEON 自動ベクトル化が有効な場合でも約 10.8µs です。
   - 最適化レベルが `-O3` に設定されていると、PLD 命令が挿入されます。(これについては後述します。)

NEON イントリンシクスを使用する方法

NEON C/C++ イントリンシクスは、armcc、GCC/g++、llvm で利用可能です。これらのコンパイラは同じ構文を使用するため、イントリンシクスを用いるソース コードは、どのコンパイラでもコンパイルでき、優れたコード移植性を提供します。

基本的に、NEON イントリンシクスは、NEON アセンブラ命令の C 関数ラッパーです。生成された NEON 命令の細粒度の制御を保ちながら、NEON アセンブラ コードよりも保守が簡単な NEON コードを作成する方法を提供します。さらに、NEON レジスタ (D レジスタおよび Q レジスタ両方) に対応する新しいデータ型定義があり、異なるサイズの要素を含んでいるため、NEON レジスタに直接マップされる C 変数を作成できます。これらの変数は、NEON イントリンシクス関数へ直接渡すことができます。その後、コンパイラは、実際のサブルーチン呼び出しを発生させる代わりに、NEON 命令を生成します。

NEON イントリンシクスは、NEON 命令に対する下位アクセスを提供しますが、アセンブリ言語の作成に通常関連する次のような作業はコンパイラが実行します。

- レジスタの割り当て
- 最高性能を達成するためのコード スケジューリング、または命令の順序変更。C コンパイラは、ターゲットとなるプロセッサの指定を受け、コードの順序を変更して、CPU パイプラインが最適な方法で実行されるようにします。

イントリンシクスの主な欠点は、コンパイラに必要なコードだけを出力させることができない点です。したがって、NEON アセンブラ コードを使うことで、さらに改善できる場合があります。

NEON イントリンシクスの詳細は、次の資料を参照してください。

- 『RealView Compilation Tools Compiler Reference Guide』[参照 9]
- GCC 資料 [参照 10]

C のNEON 型

『ARM C Language Extensions』[参照 9] にはすべての NEON 型が記載されています。形式は次のとおりです。

```
<basic type>x<number of elements>_t
```

NEON 型およびイントリンシクスを使用するには、ヘッダー ファイル `arm_neon.h` を含める必要があります。

表 6 に、開発者が参照できるNEON 型の基本情報を示します。

表 6 : NEON 型定義

64 ビット型 (D レジスタ)	128 ビット型 (Q レジスタ)
`int8x8_t`	`int8x16_t`
`int16x4_t`	`int16x8_t`
`int32x2_t`	`int32x4_t`

表 6 : NEON 型定義

64 ビット型 (D レジスタ)	128 ビット型 (Q レジスタ)
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

大型の struct 型に上記のそれぞれ 2 つ、3 つ、4 つを含めた複合型もあります。これらは、NEON のロード/格納演算がアクセスするレジスタをマップするために使用します。これらの演算では、単一命令に最大 4 つまでレジスタをロード/格納できます。次に例を示します。

```
struct int16x4x2_t
{
int16x4_t val[2];
}<var_name>;
```

これらを使用できるのは、ロード、格納、トランスポーズ、インターリーブ、逆インターリーブ命令のみです。実際のデータで演算を実行するには、次の構文を使用して個別のレジスタを選択します。

```
<var_name>.val[0] and <var_name>.val[1]
```

NEON イントリンシクス特有の方法

変数の宣言

例 :

```
uint32x2_t vec64a, vec64b; // create two D-register variables
```

定数の使用

次のコードは、ベクターの各要素に定数を複製します。

```
uint8x8 start_value = vdup_n_u8(0);
```

汎用の 64 ビット定数をベクターにロードするコードは次のようになります。

```
uint8x8 start_value =
  vreinterpret_u8_u64(vcreate_u64(0x123456789ABCDEFULL));
```

結果を正規 C 変数に戻す

NEON レジスタからの結果にアクセスするには、VST を使用してメモリに格納するか、レーン取得型演算を使用して ARM に戻します。

```
result = vget_lane_u32(vec64a, 0); // extract lane 0
```

Q レジスタの 2 つの D レジスタにアクセスする

次のように、vget_low および vget_high を用いて実行します。

```
vec64a = vget_low_u32(vec128); // split 128 bit vector
vec64b = vget_high_u32(vec128); // into 2x 64 bit vectors
```

異なる型の間で NEON 変数をキャストする

NEON 命令は型の規定が厳格であるため、C 言語のように自由に型キャストを実行できません。異なる型のベクター間でキャストする必要がある場合は vreinterpret を使用します。これは、実際にはコードを生成しませんが、NEON 型のキャストを有効にします。

```
uint8x8_t  byteval;
uint32x2_t wordval;
byteval = vreinterpret_u8_u32(wordval);
```

デスティネーション型 u8 は、vreinterpret の後に初めてリストされます。

2 つのベクターからドット積を計算する例を紹介します。これは NEON イントリンシクスの使用方法について知識を深めるのに役立ちます。難易度は中程度です。

```
float dot_product_intrinsic(float * __restrict vec1,
float * __restrict vec2, int n)
{
  float32x4_t vec1_q, vec2_q;
  float32x4_t sum_q = {0.0, 0.0, 0.0, 0.0};
  float32x2_t tmp[2];
  float result;

  for( int i=0; i<( n & ~3); i+=4 )
  {
      vec1_q=vld1q_f32(&vec1[i]);
      vec2_q=vld1q_f32(&vec2[i]);

      sum_q = vmlaq_f32(sum_q, vec1_q, vec2_q);
  }
  tmp[0] = vget_high_f32(sum_q);
  tmp[1] = vget_low_f32 (sum_q);
  tmp[0] = vpadd_f32(tmp[0], tmp[1]);
  tmp[0] = vpadd_f32(tmp[0], tmp[0]);
  result = vget_lane_f32(tmp[0], 0);
  return result;
}
```

注記：前述したように、NEON 型およびイントリンシクスを使用するには、ヘッダー ファイル arm_neon.h を含める必要があります。

GCC で NEON イントリンシクスをコンパイルする

自動ベクトル化を設定して C コードをコンパイルする複雑なオプションとは異なり、NEON 命令のコンパイルは非常に単純で、2、3 のコンパイラ オプションしか必要ありません。

- -On (デフォルト)。最適化レベルを設定します。
- -mcpu=cortex-a9。Zynq-7000 AP SoC のプロセッサ タイプを cortex-a9 に設定します。
- -mfpu=neon。コンパイラに対して、Zynq-7000 AP SoC の NEON 命令を生成するように指示します。

NEON アセンブラ コード最適化する方法

状況によっては、NEON アセンブラ コードが最適な性能を達成する唯一の方法となります。NEON イントリンシクスを調べると、コンパイラが最高速のバイナリを生成できないことが明らかな場合があり

ます。このような場合、特にパフォーマンス クリティカルなアプリケーションでは、アセンブラ コードを手動で慎重に作成して NEON の最良結果を達成できます。

欠点は明白です。まず、アセンブラ コードの維持は簡単ではありません。Cortex-A シリーズ プロセッサはすべて NEON 命令をサポートするものの、ハードウェアの実装が異なるため、命令のタイミングやパイプラインでの移動も異なります。つまり、NEON の最適化はプロセッサに依存します。ある Cortex-A シリーズ プロセッサで高速に実行するコードが、別の Cortex-A シリーズ プロセッサでは良好に機能しない場合があります。次に、アセンブラ コードの作成も簡単ではありません。適切に作成するには、パイプライン処理、スケジューリング問題、メモリ アクセス動作、スケジューリング ハザードなど、下位ハードウェアに機能ついて詳細な知識が必要です。これらの要素を簡単に説明します。

メモリ アクセスの最適化

一般に、NEON は大量のデータ処理に使用します。最適化で最も重要な点は、アルゴリズムが可能な限り効率的な方法でキャッシュを使用できるようにすることです。また、アクティブなメモリの場所の数を考慮することも重要です。一般的な最適化では、タイルと呼ばれる小さいメモリ領域を 1 つずつ処理するアルゴリズムを設計して、キャッシュおよび変換ルックアサイド バッファー (TLB) ヒット率を最大にし、外部のダイナミック RAM へのメモリ アクセスを抑制します。

NEON には、インターリーブおよび逆インターリーブをサポートする命令があり、正しく使用すれば、状況によっては大幅な性能向上を実現できます。`VLD1/VST1` は、逆インターリーブせず、複数のレジスタをメモリにロード、または複数のレジスタをメモリから格納します。その他の `VLDn/VSTn` 命令は、サイズが等しい 要素を 2 つ、3 つ、または 4 つ含む構造体のインターリーブおよび逆インターリーブを実行できます。

アライメント

NEON アーキテクチャは、NEON データ アクセスの完全にアライメントされていないサポートを提供しますが、命令オペコードには、アドレスがアライメントされてヒントが指定されている場合にインプリメンテーションの高速化を実現するアライメント ヒントが含まれています。

ベース アドレスは `[<Rn>:<align>]` として指定されます。

実際、アライメントされたキャッシュ ラインとしてデータを配置することも有効です。そうしなければ、データがキャッシュ ライン境界をまたぐ場合、追加のキャッシュ ライン フィルが発生する場合があり、システム全体の性能が低下します。

命令スケジューリング

NEON の高速なコードを作成するには、特定の ARM プロセッサのコードのスケジュール方法について理解しておく必要があります。Zynq-7000 AP SoC の場合、Cortex-A9 です。

結果と使用のスケジューリングは、NEON コードを作成する場合、性能を最適化する重要な方法となります。NEON 命令は一般に、1 または 2 サイクルで発行されますが、結果は必ずしも次のサイクルになるとは限りません (`VADD` および `VMOV` など、最も簡単な NEON 命令の発行時を除く)。命令の中には、たとえば、`VMLA` 乗算累積命令 (整数に 5 サイクル、浮動小数点に 7 サイクル) など、著しいレイテンシが発生するものがあります。ストールを避けるために、現在の命令とその結果を使用する次の命令の間の時間を考慮します。結果のレイテンシが少数のサイクルであっても、これらの命令は完全にパイプライン処理されるため、いくつかの演算が同時に実行される可能性があります。

スケジューリングに関するもう 1 つの一般的な問題はインターロックです。適切なハードウェア知識がなければ、データをメモリからレジスタにロードした直後に処理する可能性があります。メモリ アクセスがキャッシュ ヒットとなった場合は問題ありませんが、キャッシュ ヒットがミスになると、CPU は、データを外部メモリからキャッシュにロードするため、処理を開始するまで数 10 サイクル間待機することになります。したがって、通常 VLD 命令に依存しない命令は、VLD とその結果を使用する命令の間に配置する必要があります。Cortex-A9 プリロード エンジンを使用すると、キャッシュ ヒット 率を向上できます。これについては後述します。

また、外部メモリはオンチップ メモリに比べて低速で、レイテンシが長いことにも注意します。CPU は、キャッシュを使用してバッファに書き込み、この問題を緩和します。長いメモリ書き込みバーストがあると、書き込みバッファーがフルになり、次の VST 命令がストールする場合もあります。したがって、アセンブリ命令を書き込む場合は、メモリ アクセス命令をデータ処理命令にあわせて分散させることが重要です。

メモリ アクセス効率向上による NEON 性能の向上

プロセッサ コア性能について論じる際、開発者は一般に、現実のアプリケーションでは必ずしも正確とはいえない次のことを前提としています。

- プロセッサ パイプラインが最適で、インターロックは存在しない。
- メモリ サブシステムが理想的 (待機状態がゼロ) で、プロセッサはメモリがデータまたは命令を返すまで待機する必要がない。

オンチップ スタティック RAM は非常に高速ですが、コストがかかりすぎて大型の RAM を SoC に統合することはできません。BOM (Bill of Materials) のコストを削減するため、メイン メモリとしてダイナミック RAM が使用される場合も多くあります。過去 10 年間、プロセッサのクロック周波数は、ダイナミック RAM よりをはるかに上回るペース高速化してきました。低速なダイナミック RAM は、プロセッサの高クロック周波数のメリットを生かすことができません。ダイナミック RAM に関する別の問題は、一般にデータを返す際のレイテンシが長いことです。CPU の場合、スループットよりもレイテンシによって左右されるため高いため、重大な問題となります。

メモリ サブシステムがプロセッサ コアに必要なデータや命令を短時間で返すことができなければ、プロセッサは待機する以外、何もできません。ARM Cortex-A9 コアのアウトオブオーダー実行は、この問題を軽減できますが、解消はできません。

プロセッサとメモリ サブシステムのこのギャップをなくすため、エンジニアは最新の SoC にキャッシュを導入しました。キャッシュは、高速なオンチップ スタティック RAM とデータ移動のタイミングを決定するキャッシュ コントローラーから構成されます。しかし、キャッシュはエンベデッド システムの解決策にはなりません。効率的に使用するには、下位のハードウェア システムに関する深い知識が必要です。

キャッシュは、時間的局所性と空間的局所性によってシステム性能を向上させます。時間的局所性とは、アクセスされるリソースが近い将来再びアクセスされることを意味します。空間的局所性とは、近隣のリソースがちょうど参照されたのであれば、そのリソースにアクセスする可能性が高いことを意味します。CPU が必要とするデータがキャッシュ内で検出されると、キャッシュ ヒットと呼ばれます。キャッシュは、短いレイテンシでデータを返すことができます。データがキャッシュ内にない場合はキャッシュ ミスと呼ばれ、キャッシュ コントローラーがまずキャッシュにデータをフェッチしてから、CPU に返す必要があります。この場合、レイテンシがはるかに長くなるために実際の CPU 性能が低下します。実際に使用する前にキャッシュにデータをロードしておくと、キャッシュ ヒット率が高くなり、システム性能が向上します。ARM Cortex-A9 は、プリロード エンジンを実装するほか、実装に必要な命令を提供します。

実際の SoC では、性能と複雑性のトレードオフとしてキャッシュが実装されます。直接マップされたキャッシュは単純ですが、低効率です。フル アソシエーティブキャッシュが最も高効率ですが、ハードウェア デザインは非常に複雑になります。実際は、N ウェイ セット アソシエーティブ キャッシュが頻繁に使用されます。ただし潜在的な問題が 1 つあります。コードが適切に作成されていない場合、キャッシュ スラッシングが発生する可能性があり、システム性能を大幅に低下させる原因となります。この場合、キャッシュ ヒット率を向上するために、タイル方式でデータにアクセスする必要があります。

次に、メモリ効率を向上させる方法をいくつか紹介します。

- 「複数データのバースト ロード/格納」
- 「キャッシュ ヒット率向上にプリロード エンジンを使用」
- 「タイルを使用してキャッシュ スラッシングを防止」

複数データのバースト ロード/格納

複数の命令をロード/格納すると、連続したワードがメモリから読み出される/書き込まれるようになります。これは、スタック プッシュ /ポップやメモリ コピーの場合に非常に有効です。ワード配置アドレスでは、この方法で演算できるのはワード値のみです。オペランドは、ベース レジスタ (ベース レジスタのライトバックがオプションを指定されている場合) で、ブレース間レジスタのリストが含まれています。

一般に複数の命令をロードして格納すると、特にキャッシュが有効でない場合、またはメモリ境域が変換テーブルでキャッシュ不可とマークされている場合、同じロード/格納命令を複数回実行するよりも性能が向上します。これを理解するには、AMBA® 仕様を詳細に研究する必要があります。各メモリ アクセスには、AXI バス上のオーバーヘッドがあります。バスの効率を向上させるには、AXI サポート

バーストを使用します。これは、グループ N がまとめて連続アクセスするため、オーバーヘッドを一度しか必要としません。シングル ビート方式で N 個のワードにアクセスする場合、N 回のオーバーヘッドが必要です。これは、内部バスのスループットを低下させるだけでなく、レイテンシも長くなります。

通常、コンパイラはスタック演算に複数の命令のロード/格納のみを使用します。ルーチンがメモリ コピーのようにメモリ アクセスに大きい負荷をかけるときは、LDM/STM を手動で実行することが必要な場合があります。

このような命令の例 :

```
LDMIA R10!, { R0-R3, R12 }
```

この命令は、レジスタ (R10) が示すアドレスから 5 つのレジスタを読み出し、ライトバック指定子のために、最後に R10 を 20 (5 x 4 バイト) 増加させます。

レジスタ リストはカンマで区切られ、ハイフンは範囲を示します。このリストで指定された順序は重要ではありません。ARM プロセッサは必ず一定方向に進み、最小番号のレジスタが最小のアドレスにマップされます。

命令は、次の 4 つのモードのいずれか 1 つを使用して、ベース レジスタから進む方法を指定する必要があります。IA (ポストインクリメント)、IB (プリインクリメント)、DA (ポストデクリメント)、and DB (プリデクリメント)。これらの指定子には、スタックで機能する 4 つのエイリアス (FD、FA、ED、EA) もあります。これらは、スタック ポインターがフル スタックまたは空スタックのいずれを示すかと、スタックがメモリ内で昇順または降順のいずれで進むかを指定します。

したがって、NEON は同様の方法で複数のロード/格納をサポートします。次に例を示します。

```
VLDMmode{cond} Rn{!}, Registers
VSTMmode{cond} Rn{!}, Registers
```

モードは次のいずれかです。

- IA - 毎転送後にアドレスをインクリメント。これはデフォルト モードであり、省略できます。
- DB - 毎転送後にアドレスをデクリメント。
- EA - 空の昇順スタック演算。これは、ロードの DB および保存の IA と同じです。
- FD - フルの降順スタック演算。これは、ロードの IA、保存の DB と同じです。

NEONには、インターリーブおよび逆インターリーブに特殊な命令があることに注目してください。

- VLDn (ベクターは複数の n 要素構造体をロード) は、メモリからの複数の n 要素構造体を 1 つ以上の NEON レジスタへロードし、逆インターリーブします (n == 1 の場合を除く)。各レジスタのあらゆる要素がロードされます。
- VSTn (ベクターは複数の n 要素構造体を格納) は、複数の n 要素構造体を 1 つ以上の NEON レジスタからメモリへ書き込み、インターリーブします (n == 1 の場合を除く)。各レジスタのあらゆる要素が格納されます。

図 5 に、VST3 のグラフ デモを示します。

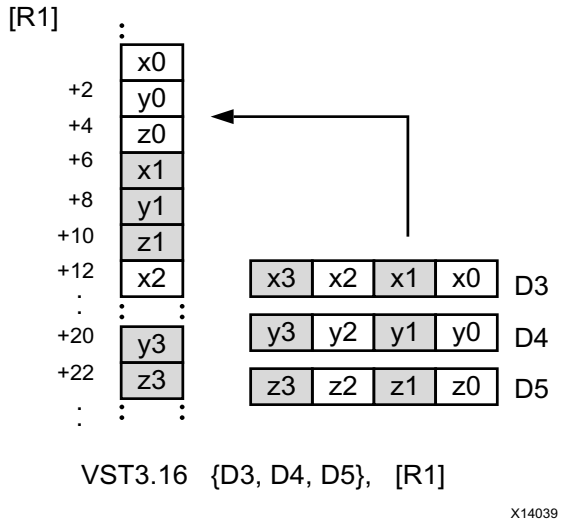

図 5 : VST3 演算のデモ

VLD2 は、2 つまたは 4 つのレジスタをロードし、偶数と奇数の要素を逆インターリーブします。これは、たとえば、左右のチャネル ステレオ オーディオ データの分離に使用できます。同様に、VLD3 は RGB ピクセルまたは XYZ 座標を別々のチャネルに分離するために使用できます。したがって、VLD4 は ARGB または CMYK 画像で利用可能です。

注記：これらの特殊な NEON 命令は、純粋な C 言語では表現できません。NEON イントリンシクスまたはアセンブラ コードを使用して、コンパイラに機械命令を生成させる必要があります。

キャッシュ ヒット率向上にプリロード エンジンを使用

ARM Cortex-A9 プロセッサは、投機的/アウトオブオーダー実行をサポートし、メモリ アクセスに関連するレイテンシを隠すことができます。それでも、外部メモリへのアクセスは非常に低速で、通常ペナルティが発生します。必要になる前に命令やデータをプリフェッチできると、CPU のストール時間を最小限に抑制し、CPU 性能を最大限に活用できます。

ハードウェアの面では、プリロード命令はすべて、専用リソースを備えた Cortex-A9 プロセッサの専用ユニットで処理されます。これによって、整数コアまたはロード格納ユニットのリソースを使用する必要がなくなります。

ソフトウェアの観点からは、キャッシュへのプリロードには、PLD (データ キャッシュ プリロード)、PLI (命令キャッシュ プリロード)、PLD (書き込み予定のプリロード データ) の 3 つの命令が関与します。PLD 命令は、データ キャッシュ ミスでキャッシュ ラインフィルを生成する場合がありますが、プロセッサは、その他の命令を引き続き実行します。PLD は、正しく使用する限り、メモリ アクセス レイテンシを隠すことで性能を大幅に向上できます。また、PLI 命令は、特定のアドレスからの命令ロードが程なく発生しそうだということをプロセッサに示唆できます。これにより、プロセッサは命令キャッシュに命令をプリロードできます。

ラボ 2

1. ザイリンクス SDK で新しいプロジェクトを作成します。
2. ラボ 2 のソース ファイルをインポートします。
3. ハードウェアでソース ファイルを実行します。
4. 達成される性能向上を確認します。

使用されたアルゴリズムは、ここでも 2 つの浮動小数点ベクター (長さ = 1024) のドット積計算で、アセンブラ コードで作成されています。プリロードで最適化されたバージョンと、PLD 命令をすべてコメント アウトした、プリロードを使用せずに最適化されたバージョンの 2 つがあります。

```
        .align  4
        .global neon_dot_product_vec16_pld
        .arm

neon_dot_product_vec16_pld:
        pld [r0, #0]
        pld [r1, #0]
        pld [r0, #32]
        pld [r1, #32]

        vmov.i32          q10, #0
        vmov.i32          q11, #0
        vmov.i32          q12, #0
        vmov.i32          q13, #0

.L_mainloop_vec_16_pld:
        @ load current set of values
        vldm  r0!, {d0, d1, d2, d3, d4, d5, d6, d7}
        vldm  r1!, {d10, d11, d12, d13, d14, d15, d16, d17}

        pld [r0]
        pld [r1]
        pld [r0, #32]
        pld [r1, #32]

        @ calculate values for current set
        vmla.f32        q10, q0, q5
        vmla.f32        q11, q1, q6
        vmla.f32        q12, q2, q7
        vmla.f32        q13, q3, q8
        @ loop control
        subs            r2, r2, #16
       bgt            .L_mainloop_vec_16_pld @ loop if r2 > 0, if we have
more elements to process

.L_return_vec_16_pld:
        @ calculate the final result
        vadd.f32        q15, q10, q11
        vadd.f32        q15, q15, q12
        vadd.f32        q15, q15, q13

        vpadd.f32       d30, d30, d31
        vpadd.f32       d30, d30, d30
        vmov.32        r0, d30[0]
        @ return
        bx                lr
```

コンソール上に実行時間が表示されます。次の点を確認してください。

- プリロードなしの最適化バージョンでは、アセンブラ関数の実行時間は約 11.8µs です。コンパイラ最適化方法をやや下回っていますが、上記の例がデモを目的としたもので、その他の下位最適化手法を使用していないためです。
- プリロードの最適化バージョンでは、アセンブラ関数の実行時間は 9.5µs です。コンパイラ最適化方法を上回る結果が得られています。

ホット キャッシュ上でソフトウェア性能を確認します。ラボ 2 では、コールド キャッシュを使用すると想定し、保守的な方法で性能をテストしています。コールド キャッシュは、アルゴリズムが開始する際にデータがキャッシュにないことを意味します。これに関するコーディングは、ソース ファイル、`benchmarking.c` の 67 行目にあります。各呼び出し前に、L1 と L2 キャッシュは、関数呼び出し `Xil_DCacheFlush()` によってフラッシュされていることがわかります。

この行をコメントアウトして、ホットキャッシュの性能を確認します。実行時間が約 2.67µs にまで短縮され、キャッシュが性能を大幅に向上できることを示します。この例では、PLD 命令のレイテンシが、演算時間よりもはるかに長いため、すべての PLD 命令の効果が表れているわけではありません。

キャッシュ ヒット率とシステム性能を向上する方法をその他に 2 つ紹介します。

- アルゴリズムが前もってデータをキャッシュにロードするプリロード ルーチンを作成し、実際のアルゴリズム演算ルーチン前に実行します。
- 実際のアルゴリズム演算のプリロードを進めるステップを増加して、プリロードを連続的に実行します。

適切に調整すると、ホット キャッシュの性能にほぼ等しい性能を実現できます。

ただし、データのプリロードを効果的に使用するには、リード タイムを考慮する必要があります。プリロードが早すぎると、プリロードされたデータがその他のコードのために削除される場合があります。遅すぎると、データが必要な時にキャッシュになく、システム性能を低下させる場合があります。重要となる要素はメイン メモリ アクセスのレイテンシです。幸いなことに、それをテストするためのコードを作成する必要はありません。オープン ソース プロジェクト `lmbench` には、エンベデッド システムでこのパラメーターを識別するテスト コードが用意されています。一般的な Zynq デバイスのコンフィギュレーション (CPU が 667MHz で稼動、DDR3 が 533MHz で稼動) では、レイテンシは、約 60 ～ 80 CPU サイクルです。実際の計算ルーチンの前にプリロード ルーチンを挿入すべき場所について十分な情報が提供されています。

また、データをプリロードする、C で記述された `memcpy()` を最適化する方法もあります。性能向上率は約 25% です。データ プリロード レイテンシを補正する演算がないため、上記の方法ほど効果はありません。

タイルを使用してキャッシュ スラッシングを防止

Zynq-7000 デバイスの場合、2 つの Cortex-A9 プロセッサそれぞれに個別の 32KB レベル 1 命令とデータ キャッシュがあり、どちらのキャッシュも 4 ウェイ セット アソシエーティブ方式となっています。L2 キャッシュは、デュアル Cortex-A9 コアの 8 ウェイ セット アソシエーティブ方式の 512KB キャッシュとして設計されています。これらのパラメーターは、キャッシュ スラッシングが発生するタイミングを予測するカギとなります。

ソリューションを特定しようとする前に、問題発生の原因を究明する必要があります。最初に、最も簡単なキャッシュ インプリメンテーション、直接マップされたキャッシュから開始します。

直接マップされたキャッシュ (図 6 を参照) では、メイン メモリの各場所が、キャッシュの単一場所にマップされています。次の図は、ラインあたり 4 ワード、4 ラインを含む小さいキャッシュ (64 バイト) を簡素化した例を示しています。この例では、アドレス ビット [3:2] はキャッシュ ライン内部のワードを選択するオフセットとして機能し、アドレス ビット [5:4] は利用可能な 4 つのキャッシュ ラインの 1 つを選択するインデックスとして機能しています。アドレス ビット [31:6] は、各キャッシュ ラインのタグ値として使用されます。

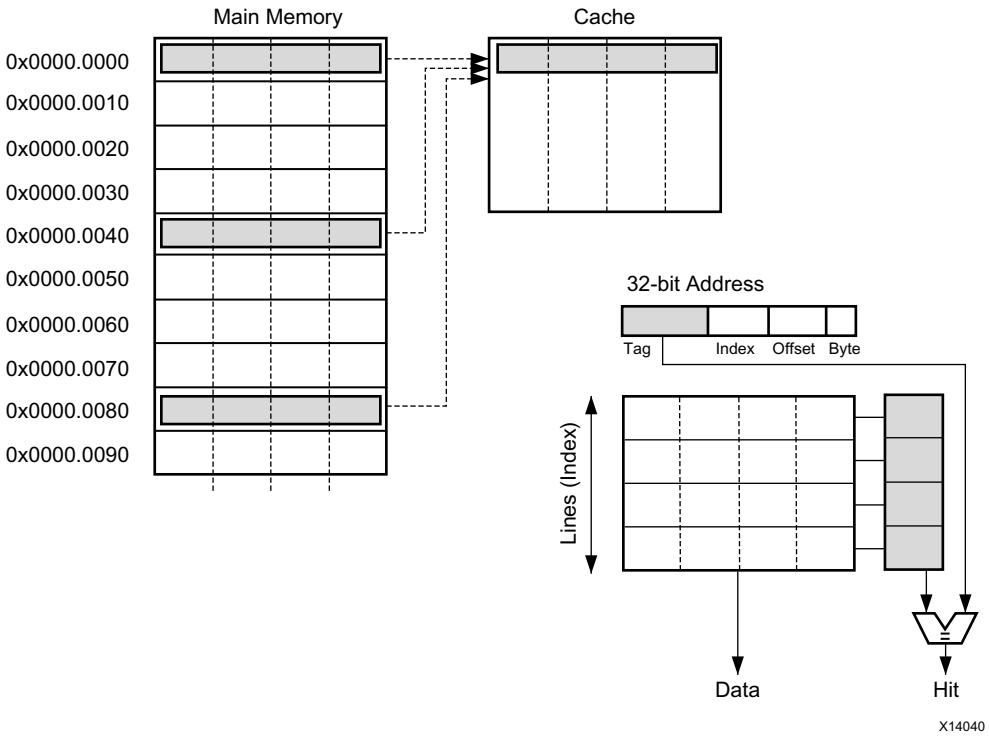

図 6 : 直接マップされたキャッシュ

図 6 は、16 バイトの配列を示しており、開始アドレス 0x0、0x40、0x80 が同じキャッシュ ラインを共有していることがわかります。キャッシュ内に存在できるのは、常にこれらのラインの 1 つのみです。

次のようなループで、ポインター result、data1、data2 がそれぞれ、0x00、0x40、0x80 を示しているとします。

```
void add_array(int *data1, int *data2, int *result, int size)
{
int i;
for (i=0 ; i<size ; i++) {
result[i] = data1[i] + data2[i];
}
```

コードが開始すると、次の順で実行されます。

- 最初に、アドレス 0x40 が読み出されます。ラインフィルはキャッシュ内には存在しないため、0x40 ～ 0x4F のデータをキャッシュに入れます。
- 次に、アドレス 0x80 が読み出されます。アドレスはキャッシュ内には存在しないため、ラインフィルが実行され、0x80 ～ 0x8F のデータをキャッシュに入れ、アドレス 0x40 ～ 0x4F のデータをキャッシュから削除します。
- 結果は 0x00 に書き込まれます。キャッシュ割り当て規則によっては、別のラインフィルが発生する場合があります。0x80 ～ 0x8F のデータはキャッシュから除去される場合もあります。
- ループの反復ごとに同じ動作が繰り返し発生する可能性があります。キャッシュ内容の再利用はほぼないに等しく、ソフトウェア性能が非常に低いことがわかります。

この問題は、キャッシュ スラッシングと呼ばれます。キャッシュ スラッシングは、直接マップされたキャッシュ上で非常に発生しやすく、このために実際のデザインではほとんど使用されません。

この問題を解決できるのが、フル アソシエーティブ キャッシュです。このソリューションでは、メイン メモリ内の任意の特定の場所が、すべてのキャッシュ ラインに格納可能です。ただし、このようなキャッシュを構築するのは、非常に小さい場合を除き、現実的ではありません。制御ハードウェアが複雑になり、消費電力が増加します。

実際には、複雑性と性能のバランスをとるために、N ウェイ セット アソシエーティブ キャッシュが広範に使用されています。このソリューションでは、キャッシュは N ページに分割され、メイン メモリの各場所は、キャッシュ内の N 個の可能な場所に格納されます。

事実、N をキャッシュ ライン サイズによって分割されたキャッシュ サイズの割合として、フル アソシエーティブ キャッシュを N ウェイ セット アソシエーティブと考えることができます。N の選択に関する研究によると、4 ウェイ アソシエーティブを超えるレベル 1 キャッシュでは性能向上が最も低く、8 ウェイまたは 16 ウェイ アソシエーティブは、さらに大型のレベル 2 キャッシュに適していることが明らかになっています。図 7 に、4 ウェイ セット アソシエーティブ キャッシュの構造を示します。

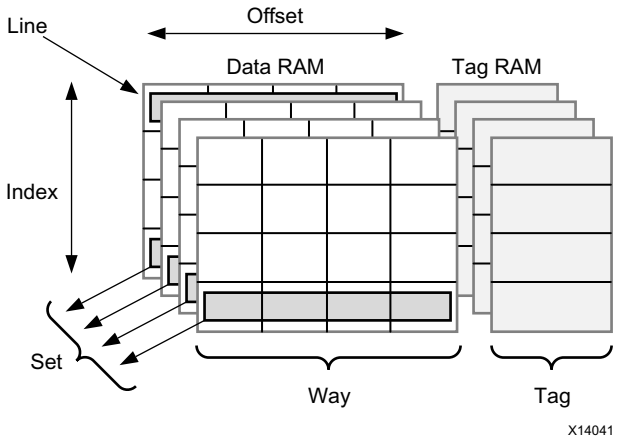

図 7 : 4 ウェイ セット アソシエーティブ キャッシュの構造

フル アソシエーティブ キャッシュであっても、厳しい条件では、キャッシュ スラッシングの問題は緩和されるものの依然として発生します。たとえば、4 ウェイ セット アソシエーティブ キャッシュ上で上記のコード例を実行する場合、ソース ベクターの数が 4 (キャッシュ割り当て規則に応じて 3 の場合でも) を超えると、キャッシュ スラッシングが発生します。

このような問題が発生する典型的な事例は、2 次元配列内部の同じ列内の要素に順次アクセスする場合です。『Cortex-A Series Programmer's Guide』[参照 7] に、行列乗算例があります。次のコード例は単純な行列乗算を実行します。

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
          result[i][j] += a[i][k]*b[k][j];
```

この例では、行列 a の内容は順次アクセスされていますが、行列 b は行別にアクセスされています。このため、行列のサイズが非常に大きく、キャッシュに格納できない場合、行列 b の要素にアクセスする間にキャッシュ ミスが発生する可能性が高くなります。

この問題を解決するには、大型の行列を小さいパーティションに分割し、この小さいパーティション内で演算を行います。パーティションは、タイルとも呼ばれます。ここでは、行列のデータ型は `int` と想定しています。Cortex-A9 プロセッサの場合、L1 キャッシュ ラインは 32 バイトで、L1 キャッシュ ラインはそれぞれ 8 つの要素を保持できます。この場合、8*8 タイルを使用してコードを変更し、キャッシュ ヒット率を向上できます。次の例は、最適化されたコードです。

```
for (io = 0; io < N; io += 8)
    for (jo = 0; jo < N; jo += 8)
        for (ko = 0; ko < N; ko += 8)
            for (ii = 0, rresult = &result[io][jo], ra = &a[io][ko];
ii < 8; ii++, rresult += N, ra += N)
```

```
for (ki = 0, rb = &b[ko][jo];ki < 8; ki++, rb += N)
    for (ji = 0; ji < 8; ji++)
        rresult[ji] += ra[ki] * rb[ji];
```

ラボ 3

1. 新しいプロジェクトを作成し、ラボ 3 のソース ファイルをインポートします。
2. ボード上でスタンドアロン アプリケーションを実行します。
3. 提供されたコードで、行列サイズは、実行時間が長引かないように 512*512 に設定します。

   注記：タイルによる性能向上に注目しているため、このラボでは、NEON のコンパイラ自動ベクトル化を有効にしません。

最適化レベルは -O2 に設定し、最適化フラグは -std=c99 に設定します。

4. ラボ 3 を実行した後、次の点を確認します。
   - タイルを使用しないインプリメンテーションの実行時間は、約 7.9 秒です。
   - タイルを使用したインプリメンテーションの実行時間はわずか 2.1 秒です。
   - 性能の向上が顕著であることがわかります。

NEON は、大型データセットを処理するために使用されることが多いため、タイル方法でアルゴリズムを適切に変更すると、キャッシュ ヒット率が向上し、はるかに優れた性能が実現されます。また、プロジェクト内でコンパイラ自動ベクトル化を使用すると、さらなる向上 (中程度) を達成できます。
ラボ 1 に示すように、コンパイラは、複雑なループ上では自動ベクトル化の効果はあまりありません。さらに大きい性能向上が必要な場合、タイル内部の演算を動で最適化する必要があります。

まとめ

このアプリケーション ノートでは、Cortex-A9 プロセッサ コアで NEON を使用してソフトウェア性能を向上する 4 つの方法を紹介しました。NEON は一般に大型データセットで使用されるため、システム性能にはキャッシュ性能が必要不可欠です。また、CPU、キャッシュ、メイン メモリ間のデータ交換を向上する 3 つの方法についても考察しました。ソフトウェアの最適化は複雑なトピックです。ハードウェアから最適な性能を実現するには、これらの技法をすべて適用し、総合して適切なバランスをとる必要があります。

参考資料

このアプリケーション ノートの参考資料は次のとおりです。

1. 『Zynq-7000 All Programmable SoC : コンセプト、ツール、テクニック ガイド』(UG873) japan.xilinx.com/support/documentation/sw_manuals_j/xilinx14_7/ug873-zynq-ctt.pdf
2. 『ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition』 silver.arm.com/download/download.tm?pv=1299246
3. 『NEON Programmer’s Guide』 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html
4. 『Cortex™-A9 NEON™ Media Processing Engine Technical Reference Manual』 infocenter.arm.com/help/topic/com.arm.doc.ddi0409g/DDI0409G_cortex_a9_neon_mpe_r3p0_trm.pdf
5. 『Cortex™-A9 Technical Reference Manual』 infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf
6. 『Cortex™-A9 MPCore® Technical Reference Manual』 infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf
7. 『Cortex™-A Series Programmer's Guide』 silver.arm.com/download/download.tm?pv=1296010
8. 『Zynq-7000 All Programmable SoC テクニカル リファレンス マニュアル』(UG585) japan.xilinx.com/support/documentation/user_guides/j_ug585-Zynq-7000-TRM.pdf
9. 『RealView Compilation Tools Compiler Reference Guide』は infocenter.arm.com から入手できます。

10. GCC 資料は、gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html から入手できます。

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	内容
2014 年 5 月 28 日	1.0	初版
2014 年 6 月 12 日	1.1	表 4 および表 5 を更新。

Notice of Disclaimer