Alveo 開発の基礎

UG1352 (v1.1) 2019 年10 月23 日

XILINX®

XILINX®

第 1 章

Alveo を使用したアクセラレーション入門

はじめに

ザイリンクス FPGA および Versal ACAP デバイスは、高パフォーマンスでワークロードの大きいアルゴリズムのレイテンシの短いアクセラレーションに適しています。ムーアの法則の限界が指摘されるなか、機能性、消費電力、レイテンシ、柔軟性の最適なバランスを求める開発者は、デザイン特化アーキ テクチャ(DSA) をツールとして選択するようになってきています。それでも、ソフトウェアのバック グラウンドしかない開発者にとっては、FPGA および ACAP の開発は非常に困難であるように思われるかもしれません。

この資料およびチュートリアルでは、ザイリンクスのカスタマーおよびパートナーを対象に、ザイリンクスカードを使用したアプリケーションのアクセラレーション方法をわかりやすく紹介します。まず、アクセラレーションの基礎から開始し、基本的なアーキテクチャ上の手法、アクセラレーションに適したコードの特定、メモリの管理および Alveoカードとの通信を最適に実行するためのソフトウェア API の使用について説明します。

この資料はソフトウェア開発者を対象にしており、ハードウェア開発者用ではありません。RTL コード記述、下位 FPGA アーキテクチャ、高位合成最適化などのトピックは、ザイリンクスから提供されているほかの資料を参照してください。この資料の目的は、ユーザーが Alveo の使用方法を短期間で習得し、アクセラレーションの目標を達成できるよう、ザイリンクスデバイスに関する知識を深め、使いこなせるようになってもらうことです。

デザインファイル

このディレクトリツリーには、**doc** および **examples** という 2 つのディレクトリがあります。この 2 つのディレクトリにはそれぞれ、この資料のソースファイルと、この資料で使用するサンプルデザインが含まれています。サンプルデザインは、この資料の演習セクションで使用します。この資料のコード例は、できる限り簡潔に、ポイントをつかみやすいものにしています。

XILINX®

第 2 章

アクセラレーションの基礎

アクセラレーションの概念

ここでは、すぐに API についての説明を始めるのではなく、アクセラレーションの開発経験があまりない開発者のために、簡単な比喩を使用して、アクセラレーションシステムで何が起こる必要があるのかを説明しましょう。

たとえば、自分の住んでいる街のツアーガイドを任されたとします。その街は、人口密度の高い大都市かもしれませんし、人がまばらに暮らしている小さな町かもしれませんが、どんな街であろうと、その地域の交通ルールに従う必要があります。この「街」が、アクセラレーションシステムにおけるアプリケーション空間に相当します。ツアーでは、街の歴史や地理などの地域情報を説明したり、観光名所や商店に参加者を案内したりします。ツアーガイドとしてやらなければならない仕事が、アルゴリズムにあたります。

アプリケーション空間とアルゴリズムを考慮して、小規模で開始します。ツアーの参加者は、最初は 1 台の車に収まる程度の人数ですが、毎年より大型で高速の車を購入していきます。ツアーの人気はしだいに高まり、参加申込者は増える一方です。ツアー用にスポーツカーを買ったとしても、街を案内するスピードには限界があります。さらに事業を拡大するにはどうしたらいいでしょうか。これが CPU アクセラレーションの問題です。

その答えはツアーバスです。スポーツカーに比べれば、バスの最高速度は低いですし、乗客の乗り降りにも時間がかかりますが、スポーツカーよりずっと多くの人を乗せられます。つまり、アルゴリズムでより多くのデータを処理できるということです。ツアーバスの台数を増やしていけば、さらに多くのツアー客に対応できるようになります。これが、GPU アクセラレーションのモデルです。このモデルでツアー事業は順調に拡大していきますが、ガソリン代がかさみ始め、観光名所の1 つである噴水の前では交通渋滞が慢性的に起こるようになり、しかもこの街でワールドカップが開催されることが決まりました。

モノレールでも建設しなければ、ツアー事業は頭打ちです。そこで街の税金が投入され、モノレール建設が許可されました。モノレールは、どの車両も満席で、すべての観光名所に停車します。しかも、必要に応じてブルドーザーを使用して路線を変更できる柔軟性もあります。この比喩では説明が厳しくなってきましたが、ツアーガイドとしてはこれは夢の世界です。でも、これこそが Alveo アクセラレーションのモデルなのです。FPGA は、GPU の並列処理と、ドメイン特化アーキテクチャのレイテンシの短いストリーミングを組み合わせ、これまでにないパフォーマンスを実現します。

ところが、古い冗談にもあるように、いくらフェラーリでも、運転手がギアの変え方を知らなければスピードは出せません。比喩を使った説明はここまでにして、ここからは実際のアクセラレーションについて学んでいきましょう。

アクセラレーションを利用可能な部分の特定

アクセラレーションシステムでは通常、パフォーマンスターゲットを設定します。そのターゲットは、全体的なエンドトゥエンドのレイテンシ、1 秒ごとのフレーム数、スループットなどで設定されます。一般的に、アクセラレーションの候補となるのは、確定的な方法で大量データを処理するアルゴリズム処理ブロックです。
1967 年、ジーン・アムダールという人が「アムダールの法則」として知られる理論を提唱しました。この法則では、システム内で達成可能な高速化は次の式で表されます。

$$S_{latency}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

この式で、$S_{latency}$ はタスクの理論上の高速化、s はアクセラレーションで効果が得られるアルゴリズムの部分、p はアクセラレーションを適用する前にそのタスクの実行にかかった時間を表わします。

アムダールの法則は、アプリケーションにおけるアクセラレーションの効果には明らかな限界があることを示しています。これは、アクセラレーションできない部分のタスク(意思決定タスク、I/O などのシステムオーバーヘッドになるタスク) がシステムのボトルネックになるからです。

$$\lim_{s \to \infty} S_{latency}(s) = \frac{1}{1 - p}$$

アムダールの法則が正しいのであれば、現代のシステムの多くに汎用またはドメイン特化アクセラレーションが使用されているのはなぜでしょうか。現代のシステムでは、処理されるデータ量はしだいに増えていますが、アムダールの法則はサイズが決まっている場合にしか適用できません。つまり、限界が存在するのは、p が全体的な実行時間に対して一定であるからです。

1988 年に、ジョン・グスタフソンとエドウィン・バルシスがアムダールの法則を改良した法則(グスタフソンの法則) を提唱しました。この法則では、式は次のようになります。

$$S_{latency}(s) = 1 - p + sp$$

$S_{latency}$ はタスクの理論上の高速化、s は並列処理の効果を得ることができるタスクの高速化(レイテンシ)、p はアプリケーションに改良を加える前の全体的なタスクレイテンシに占める割合を表わします。
グスタフソンの法則は、より多くの演算リソースを使用して1　つのタスクの実行時間を高速化しようとするアムダールの法則とは異なり、より多くの演算リソースを使用して同じ時間内により多くの演算を実行できるようにします。

どちらの法則も、アプリケーションを高速化させるためには、並列処理が必要であることを示しています。つまり、並列処理によって、同じ時間内でより多くのデータを処理しようとするか、同じデータ量をより短い時間で処理しようとしているわけです。どちらの方法にも数学的な限界がありますが、より多くのリソースを使用することで、程度の違いはありますが、どちらも利益が得られます。一般的には、Alveo カードは、大きなデータセットで大量の計算を実行するアルゴリズム(ビデオトランスコーディング、ファイナンス分析、ゲノム解析、機械学習などの大型データブロックを並列処理可能なアプリケーション) で効果を発揮します。アクセラレーションに取り組む際は、ソフトウェア

のアルゴリズムに外部アクセラレーションをどのように、いつ適用するのかを理解することが重要です。Alveo カードを使用する場合でも、それ以外のカードを使用する場合でも、アクセラレータにランダムなコードを挿入しただけでは通常は最適な結果は得られません。その理由は 2 つあります。1 つは、アクセラレーションを活用して並列処理部分を増やすには、順次アルゴリズムの構成を変える必要がある場合があること、もう1　つは、Alveo アーキテクチャでは並列データを高速に処理できますが、PCIe 上での DDR メモリとのデータ転送には追加レイテンシがあることです。この追加レイテンシは、外部アクセラレータとデータを共有するために支払う必要のある一種の「アクセラレーション税」だと考えることができます。

これを考慮して、次の条件を満たすコード部分を見つけます。

- 確定的な方法で大きなデータブロックを処理する。
- データの依存関係が明確に定義されており、順次処理またはストリームベースの処理を使用する。範囲を制限できる場合以外は、ランダムアクセスは回避することをお勧めします。
- 処理に時間がかかり、CPU と Alveoカード間のデータ転送のオーバーヘッドがアクセラレータの実行時間を支配しない。

Alveo の概要

ソフトウェアの説明に入る前に、Alveoカード自体の機能について説明しておきましょう。各 Alveo カードは、アクセラレーション用のFPGA またはACAP、広帯域幅 DDR4 メモリバンク、高帯域幅の PCIe Gen3x16 リンクを介するホストサーバーへの接続の 3 つの基本エレメントで構成されています。このリンクは、Alveo カードとホストの間で毎秒約 16 GiB のデータを転送できます。

CPU、GPU、またはASIC　とは異なり、FPGA は実質的にブランク状態のチップです。FPGA は、フリップフロップ、ゲート、SRAM などの下位ロジックのリソースをまとめたものですが、固定の機能はほとんどありません。デバイスのあらゆるインターフェイスは、PCIe や外部メモリリンクも含め、これらのリソースを少なくとも一部使用してインプリメントされます。
PCIe　リンク、システムのモニター機能、ボードの状態をチェックするインターフェイスを常にホストプロセッサで使用できるようにするため、Alveo デザインは概念的にシェルと役割の 2 つの部分に分けられます。シェル部分には、外部リンク、コンフィギュレーション、クロッキングなどのスタティック機能がすべて含まれます。役割部分には、デザイン特定のアルゴリズムをインプリメントするカスタムロジックが含まれます。このトポロジを図 2.1 に示します。

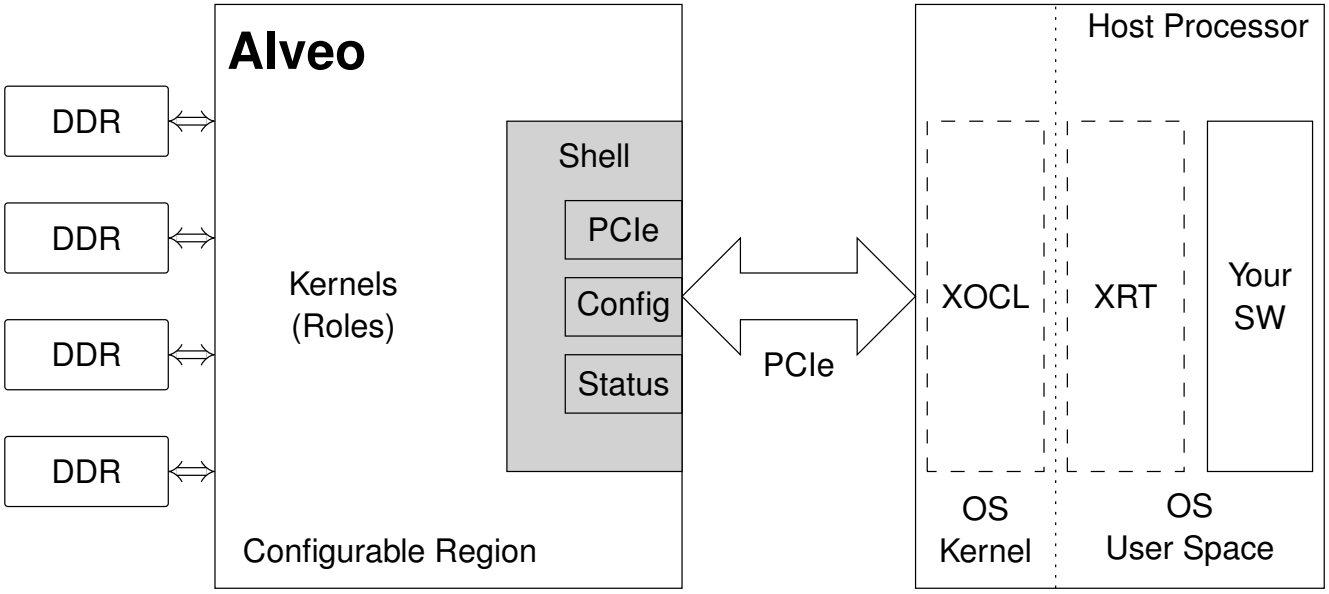

図 2.1: Alveo の概念的なとトポロジ

Alveo FPGA の内部は、さらに複数の SLR (Super Logic Region) に分かれており、高パフォーマンスデザインのアーキテクチャで有益です。これは上級トピックであり、Alveo の開発に初めて着手する場合は気にする必要はありません。

Alveo カードには複数のオンカード DDR4 メモリがあります。これらのメモリは、広帯域幅で Alveo デバイスと通信し、OpenCL のコンテクストでは「デバイスグローバルメモリ」と呼ばれます。各メモリバンクのデータ容量は 16 GiB で、2400 MHz DDR で動作します。このメモリは非常に広帯域幅なので、カーネルを簡単に飽和させることができます。ただし、このメモリの読み出しまたは書き込み操作を実行する際にレイテンシが発生し、特に不連続のアドレスにアクセスする場合や、短いデータビートを読み出したり書き込んだりする場合に、影響があります。

一方、PCIe レーンの帯域幅は十分に広いのですが、Alveo カード自体の DDR メモリの帯域幅ほどではありません。また、PCIe 上でのデータ転送のレイテンシは非常に長くなります。PCIe 上でのデータ転送は、最小限に抑えることをお勧めします。連続データを処理する場合は、カーネルが別の処理を実行している間にデータを転送するようなシステムを構築してみてください。たとえば、Alveo がビデオの1 フレームを処理している間に、次のフレームを CPU からグローバルメモリに転送するなどです。

FPGA アーキテクチャおよび Alveo カード自体についてさらに詳細に説明することもできますが、この資料では概要を紹介することが目的なので、基本的な情報はここまでにしておきましょう。アクセラレーションアーキテクチャを設計するという観点から言えば、重要な点は次のとおりです。

- PCIe 上でのデータ転送は、Gen3x16 であってもレイテンシが長くなります。大量のデータ転送では、帯域幅がシステムのボトルネックになる可能性があります。
- DDR4 と FPGA 間の転送は、PCIe 上での転送と比較して帯域幅およびレイテンシの面ではかなり良いですが、外部メモリを使用すると、全体的なシステムパフォーマンスに影響します。
- FPGA ファブリック内では、1 つの操作から次の操作へのストリーミングに実質的なコストはかかりません。これが先ほどのモノレールの比喩にあたります。これについては、後ほど詳しく説明します。

ザイリンクスランタイム(XRT) およびAPI

ハードウェアアクセラレーションシステムは、大まかには、ハードウェアアーキテクチャおよびインプリメンテーションと、そのハードウェアと通信するソフトウェアの 2 つの部分で説明できます。FFmpeg や Gstreamer など、アプリケーションにどんな上位ソフトウェアフレームワークを使用していても、Alveo カードでは、Alveo ハードウェアと通信するソフトウェアライブラリはザイリンクスランタイム (XRT) です。

XRT は多くのコンポーネントで構成されていますが、その主な役割は次の 3 つです。

- **Alveo** カードカーネルのプログラムとハードウェアのライフサイクルの管理
- メモリ割り当てと、ホストのCPU とAlveo カード間でのそのメモリの移動
- ハードウェア操作の管理: カーネル実行のシーケンス、カーネル引数の設定など

上記の操作は、Alveo アクセラレーションカード上でのコストが高い順にリストしています。さらに詳しく見ていきましょう。

Alveo カードカーネルのプログラムには、ある程度時間がかかります。コンフィギュレーションイメージを転送する PCIe 帯域幅など、Alveo カードの FPGA の性能にもよりますが、プログラムにかかる時間はおおよそ数十ミリ秒から数百ミリ秒です。これは通常はアプリケーションを起動したときに 1 回だけ実行される操作なので、コンフィギュレーション時間はセットアップレイテンシとして吸収されますが、これを認識しておくことは重要です。アプリケーションによっては、異なる大型のカーネルを提供するため、Alveo が複数回再プログラムされるものもあります。そのようなアーキテクチャを構築する場合は、コンフィギュレーション時間をアプリケーションにできる限りシームレスに組み込む必要があります。また、多くのアプリケーションで同時に Alveo カードを使用することは可能ですが、1 回にプログラムできるのは1 つのイメージのみであることに注意してください。

XRT が真価を発揮できるのは、メモリの割り当てとメモリの移動です。メモリを効率よく割り当てて管理することは、アクセラレーションアーキテクチャを開発する上で非常に重要です。メモリおよびメモリの移動を効率よく管理しないと、アプリケーションの全体的なパフォーマンスに大きく影響します。XRT には、メモリと通信するための機能が多数あります。これらの機能については、後ほど説明します。

最後に、XRT は、カーネル引数を設定し、カーネル実行フローを管理することにより、ハードウェアの操作を管理します。カーネルは、1 つまたは多数のプロセスから、順次または並列に、ブロッキング方式またはノンブロッキング方式で実行できます。ソフトウェアとカーネルの通信方法はユーザーが制御できます。この資料の後の方で、その制御方法をいくつか説明します。

XRT は下位 API です。アドバンスユースモデル、または一般的でないユースモデルを使用している場合は、XRT と直接通信する方がよいかもしれませんが、OpenCL、ザイリンクスメディアアクセラレータ (XMA) フレームワークなどの上位 API が使用されるのが一般的です。図 2.2 に、使用可能な API の概要を示します。この資料では、OpenCL API を中心に説明します。ザイリンクスでは FPGA に特化したタスク用の拡張機能を提供していますが、OpenCL を使用したことがあれば、ほぼ同様です。

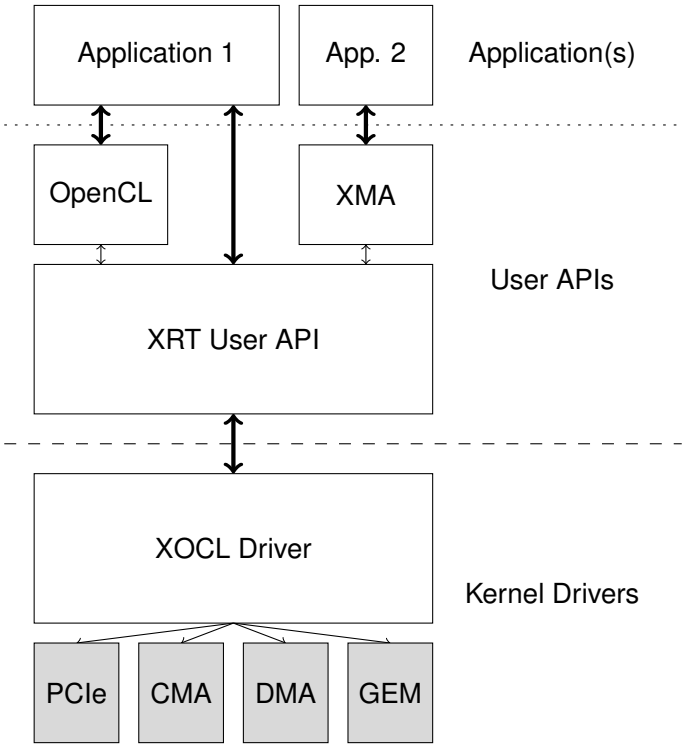

図 2.2: XRT ソフトウェア スタック

XILINX

第 3章

ランタイムソフトウェアデザイン

メモリ割り当ての概念

CPU 上でプログラムを実行する際は、ハードウェアでメモリがどのように管理されているかを気にすることは通常ありません。一部のプロセッサアーキテクチャではアライメントなどの問題が発生することがありますが、現代の OS およびコンパイラではこれらが抽象化されているので、ユーザーが下位ドライバーにかかわる作業 (またはアクセラレーション) を実行するのでない限り、ユーザーには見えないようになっています。

基本的には、メモリには 6 つの属性があると考えることができます。まずデータバッファーへのポインターでは、データポインターは仮想または物理です。そのポインターが指定するメモリは、ページング方式を使用しているか、物理的に連続しています。最後に、プロセッサの視点から見ると、メモリはキャッシュ可能またはキャッシュ不可能です。

現代のほとんどの OS では、仮想メモリが使用されています。これには多くの理由がありますが、この資料では説明ません。ここでは、XRT を実行するのに通常に使用される Linux では、仮想メモリが使用されるということを知っておくだけで十分です。そのため、標準 C または C++ のユーザー空間の API 関数 (malloc()、new など) を使用すると、ポインターが指定するのは物理メモリアドレスではなく仮想メモリアドレスです。

また、ほとんどの場合、ページング方式を使用するメモリのアドレスブロックを指定するポインターを使用することになります。Linux を含む現代のほとんどの OS では、アドレス範囲はページという単位に分割され、各ページのサイズは 4 KiB です(サイズは OS によって異なる可能性あり)。各ページは、物理メモリの対応するページにマップされます。この説明は、不正確に単純化されていると指摘する声もあるかもしれませんが、この資料はコンピューターアーキテクチャの教科書ではないので、ここでは詳細な説明は省きます。
ただし、重要なことを 2 点指摘しておきます。1 点目は、標準C のAPI を使用してヒープからバッファーを割り当てても、物理メモリアドレスは取得できないということ、2 点目は、1 つのバッファーではなく、長さが 4 KiB のメモリページN 個が取得されるということです。これを具体的に説明しましょう。たとえば、6 MiB のバッファーを割り当てるとします。ページ数は次のようになります。

$$\frac{6\,\text{MiB}}{4\,\text{KiB}} = 1536\ \text{ページ}$$

6 MiB バッファー全体をホストから Alveo にコピーする場合は、1536 個の仮想ページ アドレスを物理メモリ アドレスに変換する必要があります。その後、これらの物理アドレスをスキャッター ギャザー リストにまとめ、スキャッター ギャザー機能を持つ DMA エンジンにエンキューし、これらのページを 1

つずつそれぞれのデスティネーションにコピーします。Alveo からバッファーをホストメモリの仮想、ページング方式アドレス範囲にコピーする場合は、この逆になります。ホストプロセッサは通常高速なので、このリストを作成するのにそれほど時間はかかりません。ただし、バッファーのサイズがそれなりに大きいため、システムの全体的なレイテンシに影響する可能性があるので、システムの全体的なパフォーマンスへの影響を理解しておくことが重要です。
このようなシステムの簡略図を図 3.1 に示します。この例には、A および B という2 つの仮想バッファーがあります。この例では、バッファーA のデータを Alveo に転送して処理し、バッファー B をアップデートして、結果をバッファーA に戻します。この図から、仮想メモリから物理メモリにどのようにマップされるのかがわかります。Alveo カード内では、アクセラレータは物理メモリアドレスでのみ操作を実行するので、データは常に連続して格納されます。これは、これが一般的に最高のパフォーマンスを得ることのできる構成だからです。

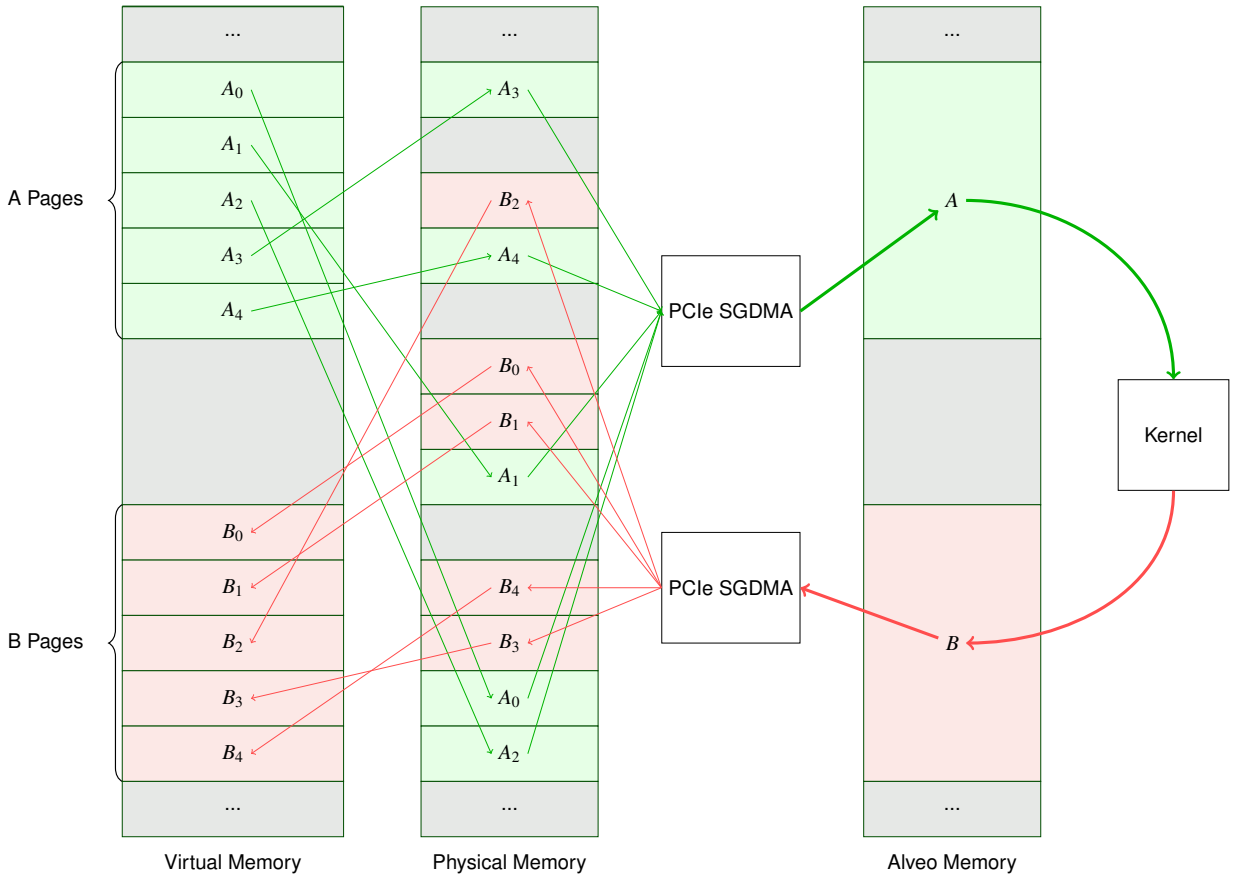

図 3.1: Alveo との仮想メモリ転送

この単純化されたデータフローであっても、既に少し複雑です。ここで、何メガバイト (またはギガバイト) にも及び、何千ページもある非常に大きなバッファーを考えてみてください。ホストプロセッサがいくら高速でも、これらのスキャッターギャザーリストを作成して管理し、ページテーブルも管理するのにどれだけ時間がかかるか想像がつきます。実際のメモリは通常この例ほど細分割されていませんが、ページの物理アドレスは事前にわからないので、これらのアドレスはそれぞれ固有なアドレスとして処理する必要があります。

物理メモリでページがすべて連続しているとわかっていれば、スキャッターギャザーリストを作成す

るのもずっと簡単になります。データバイトが順に並んでいれば、物理アドレスをインクリメントするとデータ [n+1] が得られるので、バッファーの開始アドレスとサイズがわかればスキャッターギャザーリストを作成できます。

これは、DMA と Alveo の間の操作だけでなく、多くの DMA 操作で有益です。現代のオペレーティングシステムでは、この目的のためだけに、メモリアロケーター(通常はカーネルサービスを介して提供)があります。Linux では、これは連続メモリアロケーター サブシステムを使用して実行されます。これはカーネル空間の機能ですが、dmabuf、XRT API、グラフィックドライバーなど、さまざまなメカニズムを介してユーザーが確認できます。先ほどのバッファーを連続で割り当てると、図 3.2 に示すように、すっきりとした図になります。

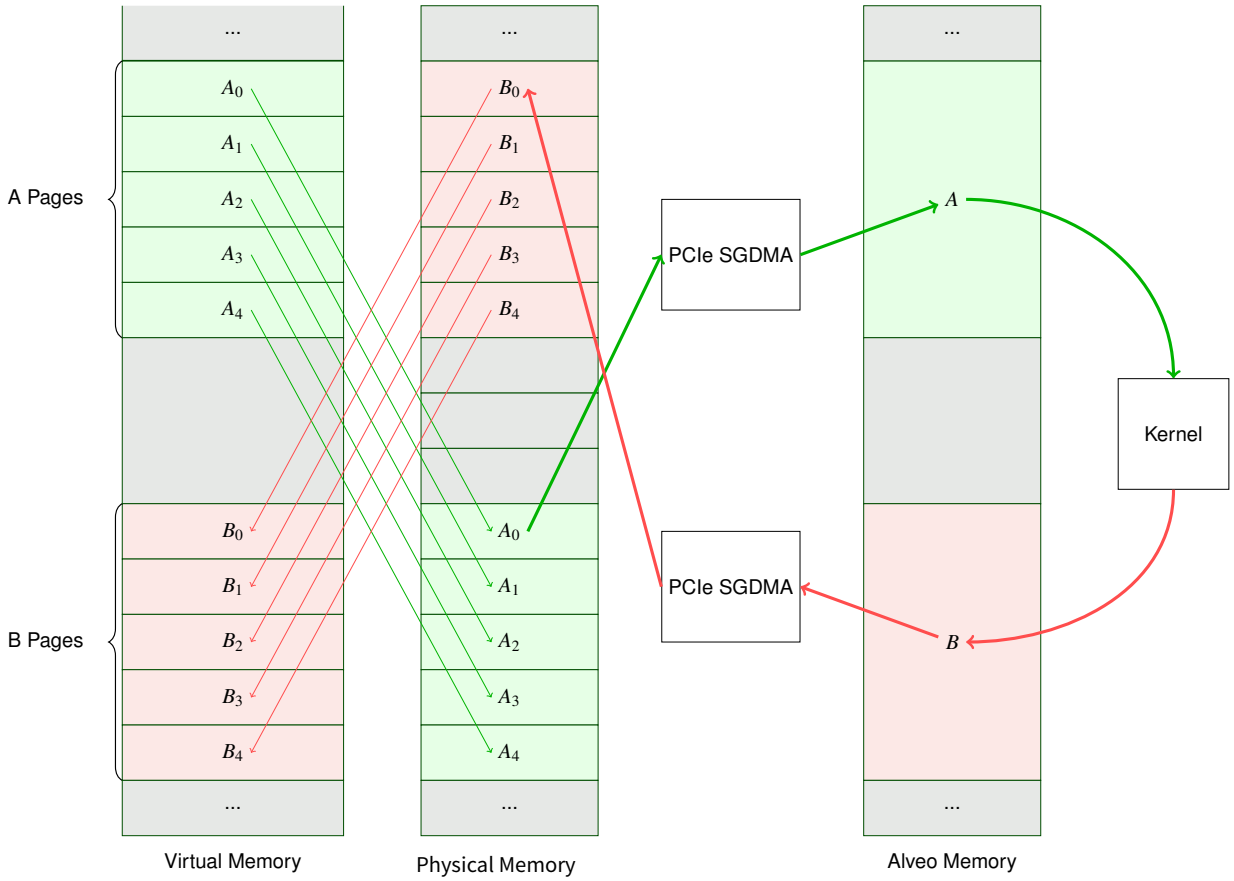

図 3.2: Alveo との仮想メモリ転送

バッファーが4k ページの境界で開始しない場合はどうなるでしょうか。DMA エンジンではある程度のアライメントが必要ですが、Alveo の DMA も例外ではありません。割り当てられたメモリがページの境界に揃えられていない場合、ランタイムにより自動的に揃えられます。そのために **memcpy()** 演算が追加されるので、演算コストが高くなります。これはできるだけ早期に解決すべき問題なので、警告メッセージを非表示に設定していなければ、ランタイムから警告メッセージも表示されます。この問題の特定方法および修正方法は、後ほど説明します。

最後に、メモリがキャッシュ可能なのか、キャッシュ不可能なのかを知る必要があります。外部メモリへのアクセスはコストが高いので、現代のほぼすべてのプロセッサにレイテンシが非常に短い内部データキャッシュがあります。プロセッサのアーキテクチャによって、キャッシュのサイズは数十キ

ロバイトから数メガバイトまでさまざまです。この内部キャッシュメモリは、必要に応じて外部の物理メモリと同期化されます。通常、このキャッシュ管理はプロセッサ上で実行されているソフトウェアには見えません。キャッシュを使用する利点は実行時間の短縮により確認できますが、開発者がキャッシュに手を加える必要は通常ありません。

ただし、DMA を使用する場合は、CCIX などのキャッシュコヒーレンシを共有する技術がなく、プロセッサとアクセラレータが共有する必要のあるデータはすべて、転送前に外部メモリと同期化させる必要があります。DMA 転送を開始する前に、キャッシュにあるデータを外部メモリにフラッシュする必要があります。同様に、外部メモリからデータが戻されると、キャッシュにあるデータを無効化して、外部メモリから更新する必要があります。これらの操作は、x86 型のプロセッサでは非常に高速に実行され、ランタイムにより透過的に実行されます。ただし、ほかのアーキテクチャでは、キャッシュ管理がパフォーマンスに影響する可能性があるので、API にキャッシュ不可能なバッファーを割り当て、操作する機能があります。キャッシュ不可能なバッファーのデータにアクセスするプロセッサは、キャッシュ管理操作のみを実行する場合よりも処理速度がかなり遅くなります。これは通常、プロセッサがバッファーに順次にアクセスしているものの、バッファーに含まれるデータにはアクセスしないモデルで使用されます。

Alveo ソフトウェア入門

ここまでで、多くの背景情報を学びました。ソフトウェアの例を実行してみる前に、ここまでで学んだ重要点を復習しておきましょう。

- アクセラレーションは通常、同じタスクを高速に実行する方法 (アムダールの法則) と、同じ時間内により多くのタスクを実行する方法 (グスタフソンの法則) を組み合わせて実行されます。これら 2 つ方法では、最適化方法および最適化の効果が異なります。
- アクセラレーションには「アクセラレーション税」(コスト) がかかります。実際のシステムでアセラレーションを達成するには、それで得られる利益が、アクセラレーションのために発生する追加レイテンシを上回る必要があります。つまり、アルゴリズムで「コストに見合う価値」が最大限になるような方法を選択する必要があります。
- Alveo カードとの通信には、XRT および OpenCL などの上位 API を使用します。ソフトウェア側の最適化は、ハードウェアカーネルの最適化とは別に、ライブラリを使用して実行されます。
- メモリの割り当てと管理は、アプリケーションの全体的なパフォーマンスに大きな影響を与えます。

これらのトピックについて、例を使用して詳しく説明します。

XILINX®

第 4 章

ソフトウェアの例

デザインファイル

これらの例のインストールパッケージには、**doc** および **examples** という2 つのディレクトリがあります。**doc** ディレクトリにはこの資料のソースファイルが含まれており、**examples** ディレクトリには例をビルドして実行するのに必要なソースファイルがすべて含まれています (SDAccel や XRT などのビルドツール、Alveo 開発シェルは除く)。

examples ディレクトリには、**hw_src** および **sw_src** というディレクトリがあります。名前からわかるように、これらのディレクトリにはアプリケーションのハードウェアおよびソフトウェアのソースファイルが含まれています。ハードウェアソースはザイリンクスの **XOCC** コンパイラを使用して FPGA 上で実行するアルゴリズムに合成され、ソフトウェアソースはホストプロセッサ上で実行するため標準の GCC を使用してコンパイルされます。

このチュートリアルでは、ハードウェアよりもソフトウェアに焦点を置いているので、ファイルを管理しやすいようにソースファイルを分けています。実際のプロジェクトでは、ディレクトリ構造などはプロジェクトによって異なるので、チームまたは組織のベストプラクティスに従ってディレクトリを管理してください。
一部の例は外部ライブラリを使用するので、環境設定を簡単にするため、ソフトウェア例に **Cmake** ビルドシステムが使用されています。ハードウェアでは標準の **make** が使用されています。これは、XOCC に渡されるコマンドライン引数を簡単に確認できるようにするためです。

ハードウェアデザインセットアップ

このセクションでは、Alveo カードをターゲットとするアクセラレーションの概念を説明します。まずホストコードを記述する作業から始め、FPGA をプログラムし、メモリを割り当てて移動させます。最初の方の例では、非常に単純なアクセラレータを使用します。実際、アクセラレーションハードウェアが非常にささいなものなので、少なくともはじめのうちは、アルゴリズムは CPU 上での方が高速に実行されるはずです。

また、ハードウェアデザインの構築にも時間がかかる可能性があります。ナノ秒未満のタイミングで何十億個もあるトランジスタにまたがるカスタムロジックを合成し、配置配線するのは、機械コードにコンパイルするよりも複雑です。それでも価値のある結果が得られます。FPGA を不必要に再構築せずにすむようにするため、この資料では多数のカーネルインスタンスを含む 1 つの FPGA デザインを

提供しており、サンプルデザインよってそれらのカーネルを組み合わせて使用します。カーネル最適化にも少し触れますが、基本概念を説明する程度なので、それ以上の内容は、ザイリンクスのほかの資料を参照してください。
このオンボーディングサンプルには、Alveo U200 アクセラレータカード (シェルバージョン **201830.1**) をターゲットとするビルド済みハードウェアイメージが含まれています。このカードとこのバージョンのシェルがあれば、ハードウェアのセットアップは不要で、直接ソフトウェアチュートリアルに進むことができます。
この 2 つのいずれか、または両方ともない場合は、まずハードウェアデザインをビルドする作業から始める必要があります。でも心配は無用です。ソフトウェアはその場合でも機能します。この作業を実行するには、次のディレクトリに移動します。

```
onboarding/examples/hw_src
```

ボードにインストールされているシェルのバージョンが上記のものとは異なる場合は、この手順をスキップできます。Alveo U200 以外のプラットフォームをターゲットにしている場合は、makefile を開き、1 行目を変更してご使用のプラットフォームの XPFM ファイルを指定します。

注記: 次の例では、見やすくするために改行が追加されていますが、実際の makefile には改行を追加しないでください。

```
PLATFORM := /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
            xilinx_u200_xdma_201830_1.xpfm
```

makefile を変更したら、SDAccel および XRT の環境が正しく設定されていることを確認します。実行していない場合は、次のコマンドを実行します。

```
source /opt/Xilinx/SDx/2018.3/settings.sh
source /opt/xilinx/xrt/setup.sh
```

XRT または SDAccel のインストールパスがデフォルトとは異なる場合は、コマンドラインをそれに合わせて変更します。そして、次のコマンドを実行します。

```
make
```

このビルドプロセスには時間がかかりますが、最終的には **alveo_examples.xclbin** というディレクトリにファイルが生成されます。このファイルには、演習に使用するカーネルすべてが含まれています。使用するボードとシェルの組み合わせでこのファイルをコンパイルしたら、次のセクションに進みます。

ソフトウェアデザインの構築

システム特定の依存性を避けるため、これらの例ではテストアプリケーションをビルドするために CMake を使用します。例で使用されるコードはすべて **swsrc_src** ディレクトリにあります。各ソースファイルには、対応する例に関連付けられた名前が付いています。たとえば、例 0 のものには **00_load_kernels.cpp** という名前が付いています。アプリケーション間で共有される「ヘルパー」ファイルもいくつかあり、これらはそれぞれのライブラリにコンパイルされます。
ソースをビルドするには、ハードウェアの場合と同様、ご使用の環境で XRT が正しく設定されていることを確認してください。環境変数 $XILINX_XRT は、XRT のインストールディレクトリをポイント

している必要があります。ポイントしていない場合は、次を実行します (XRT が /opt/xilinx/xrt にインストールされていると想定):

```
source /opt/xilinx/xrt/setup.sh
```

XRT を設定したら、ビルドディレクトリを作成して、そこに移動します。

```
cd onboarding/examples
mkdir build
cd build
```

そのビルドディレクトリから CMake を実行してビルド環境を設定し、make を実行してすべての例をビルドします。

```
cmake ..
make
```

異なる番号が付いた例に対応する実行ファイルと、前の手順で生成した **alveo_examples.xclbin** ファイルのコピーが生成されます。

例 0: Alveo イメージの読み込み

概要

最初の例では、Alveo カードにイメージを読み込みます。システムに電源を投入すると、Alveo カードによりシェルが初期化されます (図 2.1 を参照)。このシェルによりホスト PC との接続がインプリメントされますが、ロジックのほとんどの部分はデザインをビルドできるようにブランクのキャンバスとなっています。そのロジックをアプリケーションで使用できるようにする前に、まずコンフィギュレーションが必要です。

また、先ほど説明したように、一部の操作にはレイテンシの点でコストが高くなります。FPGA のコンフィギュレーションは、アプリケーションフローの中で最も時間のかかる作業です。コストがどの程度高いかを知るため、イメージを読み込んでみましょう。

キーコード

この例では、XRT 用に OpenCL ランタイム API を初期化し、コマンドキューを作成して、Alveo カードの FPGA をコンフィギュレーションします。これは通常、1 回だけの操作です。カードがコンフィギュレーションされると、電源がオフになるまで、または別のアプリケーションでリコンフィギュレーションされるまで、コンフィギュレーションされたままになります。1 つのカードに複数の独立したアプリケーションがハードウェアを読み込もうとすると、最初のアプリケーションが制御を解放するまでは次のアプリケーションはブロックされます。複数の独立したアプリケーションで、カード上で実行中の同じイメージを共有することはできます。
まず、コード 4.1 に示すようにヘッダーを含める必要があります。この資料に示す行番号は **00_load_kernels.cpp** ファイルの行番号に対応しています。

コード 4.1: XRT および OpenCL ヘッダー

```
// Xilinx OpenCL and XRT includes
#include "xcl2.hpp"
```

```
#include <CL/cl.h>
```

この2 つのうち、CL/cl.h のみが必要です。xcl2.hpp は、必要な初期化関数のラッパーとしてザイリンクスから提供されているヘルパー関数のライブラリです。
適切なヘッダーを含めたら、コマンドキューを初期化し、バイナリファイルを読み込んで、FPGA にプログラムする必要があります (コード 4.2 を参照)。これは、どのプログラムにもある時点で含める必要のある定型コードです。

コード 4.2: XRT および OpenCL ヘッダー

```
// This application will use the first Xilinx device found in the system
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device               = devices[0];

cl::Context context(device);
cl::CommandQueue q(context, device);


std::string device_name    = device.getInfo<CL_DEVICE_NAME>();
std::string binaryFile     = xcl::find_binary_file(device_name, argv[1]);
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);

devices.resize(1);
cl::Program program(context, devices, bins);
```

このワークフローをまとめると、次のようになります。

1. (33 ～ 34 行目): システムでザイリンクスデバイスを検出し、番号を付けます。ここではデバイス 0 をターゲットカードと想定できますが、マルチアクセラレータカードシステムでは、デバイスを指定する必要があります。
2. (36 ～ 37 行目): OpenCL コンテキストおよびコマンドキューを初期化します。
3. (39 ～ 41 行目): Alveo ボードをターゲットとするバイナリファイルを読み込みます。これらの例ではコマンドラインでファイル名を渡しますが、ファイル名はコードに記述するか、アプリケーションごとに処理することもできます。
4. (44 行目) FPGA をプログラムします。

44 行目で、プログラム操作が実際にトリガーされます。プログラム中に、ランタイムにより Alveo カードの現在の設定がチェックされます。既にプログラムされている場合は、デバイスのメタデータを xclbin から読み込んだ後に戻ることができますが、プログラムされていない場合は、ここでデバイスをプログラムします。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./00_load_kernels alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 0: Loading the FPGA Binary --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'

FPGA programmed, example complete!

-- Key execution times --
OpenCL Initialization : 1624.634 ms
```

この例ではFPGA の初期化に 1.6 秒かかりました。このカーネルの読み込み時間には、ディスク I/O、PCIe レイテンシ、コンフィギュレーションオーバーヘッドなど、多数の操作が含まれます。通常は、アプリケーションの起動時にFPGA をコンフィギュレーションするか、あらかじめコンフィギュレーションしておきます。既に読み込まれているビットストリームを使用して、このアプリケーションをもう一度実行しましょう。

```
-- Key execution times --
OpenCL Initialization :  262.374 ms
```

.26 秒は、1.6 秒よりもかなり良い結果です。ディスクからファイルを読み出して解析し、FPGA に読み込まれた xclbin ファイルが正しいことを確認する必要がありますが、全体的な初期化時間は大幅に短縮されています。

追加演習

この演習に、追加で次のことを試してみてください。

- **xbutil** ユーティリティを使用してボードをクエリします。どの xclbin ファイルが読み込まれましたか。
- もう一度 **xbutil** ユーティリティを使用し、FPGA にどのカーネルが存在するかを確認します。FPGA のプログラム前後で違いがあるかを調べます。

学習ポイント

- FPGA コンフィギュレーションはコストのかかる操作です。必要になる前に余裕をもって FPGA を初期化しておくのが理想的です。これは、アプリケーションのほかの初期化タスクを実行しているときに別のスレッドで実行するか、専用システムのシステムブート時に実行することも可能です。
- FPGA が読み込まれると、その後の読み込み時間は大幅に短縮されます。

これで、FPGA にイメージを読み込めるようになったので、何かを実行してみましょう。

例 1: 単純なメモリ割り当て

概要

読み込んだ FPGA イメージには、非常に単純なベクター加算コアが含まれています。任意の長さのバッファー 2 つを入力として使用し、同じの長さバッファーを出力として生成します。「ベクター加算コア」という名前からわかるように、このコアは 2 つの値を加算します。

提供されているコードは、FPGA 用に最適化されていません。コード 4.3 に示すアルゴリズムを直接 FPGA デバイスに読み込んだのとほぼ同じです。特に効率が良いわけではありません。1 クロックごとに加算を1 回ずつ処理することは可能ですが、1 回に 32 ビット出力を1 つしか処理できません。

コード 4.3: ベクター加算アルゴリズム

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

この時点では、このコードがプロセッサに勝ることはあり得ません。FPGA ファブリックのクロックは、CPU クロックよりもかなり低速です。この資料のはじめに紹介した比喩に戻ってたとえると、モノレールの各車両に 1 人しか乗客を乗せていないのと同じ状況です。また、PCIe 上でのデータ転送、DMA のセットアップなどのオーバーヘッドもあります。次のいくつかの例では、この関数の入力および出力のバッファーを効率よく管理する方法を見ていきます。Alveo カードでのアクセラレーションの効果が見られるのはその後になります。

キーコード

この例では、実際に FPGA 上で演算を実行してみます。カード上で演算を実行するには、次の操作を実行する必要があります。

1. カードとのデータの送受信に使用するバッファーを割り当て、データを挿入します。
2. ホストメモリ空間と Alveo グローバルメモリとの間でこれらのバッファーを転送します。
3. カーネルを実行してこれらのバッファーのデータを処理します。
4. カーネルでの処理結果をホストメモリ空間に戻し、プロセッサからアクセスできるようにします。

これらの操作のうちカード上で実行されるのは1 つだけです。メモリ管理はアプリケーションのパフォーマンスに大きく影響するので、まずメモリ管理から見てみます。
アクセラレーションに関する作業を実行したことがない場合、メモリの割り当てに malloc() または new への標準呼び出しを使用してしまいがちです。この例では、これらの標準呼び出しを使用して、ホストと Alveo カードとの間でデータを転送する一連のバッファーを割り当てます。4 つのバッファーを割り当てます。加算の入力に使用する2 つの入力バッファー、Alveo で使用する出力バッファー、および vadd 関数のソフトウェアインプリメンテーションに使用する追加のバッファーです。これにより、Alveo 用のメモリの割り当て方法と、プロセッサの実行効率に与える影響を見ることができます。

バッファーはコード 4.4 に示すように割り当てられています。この例では、BUFSIZE は24 MiB (uint32_t 型の6 × 1024 × 1024) です。ここに示していないコードは、前の例とまったく同じか、機能的に同等のコードです。

コード 4.4: 単純なバッファー割り当て

```
uint32_t *a = new uint32_t[BUFSIZE];
uint32_t *b = new uint32_t[BUFSIZE];
uint32_t *c = new uint32_t[BUFSIZE];
uint32_t *d = new uint32_t[BUFSIZE];
```

これにより、ページング方式を使用する境界に揃えられていない、仮想メモリが割り当てられます。特に、「境界に揃えられていない」ことが後で問題になります。
バッファーを割り当て、初期テストベクター値を挿入したら、Alveo グローバルメモリに転送します。これには、**CL_MEM_USE_HOST_PTR** フラグを使用して OpenCL バッファーオブジェクトを作成します。これにより、API に対して、バッファーを割り当てる代わりにポインターが提供されます。これ自体は悪くありませんが、ポインターを割り当てる設定はしていないので、パフォーマンスの劣化を招きます。

コード 4.5 には、割り当て済みのバッファーを OpenCL バッファーオブジェクトにマップするコードが含まれています。

コード 4.5: ホスト メモリ ポインターを使用した OCL バッファーのマップ

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       a,
                       NULL);
cl::Buffer b_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       b,
                       NULL);
cl::Buffer c_from_device(context,
                         static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                                   CL_MEM_USE_HOST_PTR),
                         BUFSIZE * sizeof(uint32_t),
                         c,
                         NULL);
inBufVec.push_back(a_to_device);
inBufVec.push_back(b_to_device);
outBufVec.push_back(c_from_device);
```

このコードでは、API で認識される cl::Buffer オブジェクトを割り当て、前の段階で割り当てたバッファーからポインター a、b、c を渡します。追加フラグである CL_MEM_READ_ONLY および CL_MEM_WRITE_ONLY は、ランタイムでこれらのバッファーのカーネルからのアクセス権を指定します。つまり、a および b はホストからカードに書き込まれますが、カーネルから見るとこれらは読み取り専用です。その後、c がカードからホストにリードバックされます。カーネルから見ると、これは書き込み専用です。さらに、これらのバッファーオブジェクトをベクターに追加して、一度に複数のバッファーを転送できるようにします。実質的にはベクターにポインターを追加しているだけで、データバッファーそのものを追加しているわけではありません。

次に、コード 4.6 を使用して、入力バッファーを Alveo カードに転送します。

コード 4.6: ホスト メモリを Alveo に移動

```
cl::Event event_sp;
q.enqueueMigrateMemObjects(inBufVec, 0, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

このコード抜粋では、主要なイベントは108　行目のenqueueMigrateMemObjects() への呼び出しです。バッファーのベクターに 0 を渡しており、これがホストからデバイスへの転送であることを示します。cl::Event オブジェクトも渡します。

ここで、同期化について少し説明しておきましょう。転送をエンキューするということは、その転送をランタイムの「やるべきことリスト」に追加することですが、ランタイムはその転送が完了するのを待つわけではありません。cl::Event オブジェクトを登録することにより、今後いつでもそのイベントが発生したときに待機するようにできます。通常ここは待機するところではありませんが、さまざまな操作にかかった時間を簡単に表示できるようにするため、コード内のさまざまな時点に挿入しています。これによりアプリケーションにわずかなオーバーヘッドが追加されますが、これは演習であって、最大パフォーマンスを得るために最適化する例ではありません。
ここで、カーネルに何を渡すのかをランタイムに指示する必要があるので、コード 4.7 に示すようにします。引数リストは次のとおりです。

```
  (uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
```

この場合、a は引数0、b は引数1、のようになります。

コード 4.7: カーネル引数の設定

```
krnl.setArg(0, a_to_device);
krnl.setArg(1, b_to_device);
krnl.setArg(2, c_from_device);
krnl.setArg(3, BUFSIZE);
```

次に、コマンドキューにカーネル自体を追加し、実行が開始するようにします。通常は、転送とカーネルの間に同期化するのではなく、連続実行されるようにエンキューします。カーネルの実行をコマンドキューに追加する行は、コード 4.8 に示すようになります。

コード 4.8: カーネル実行のエンキュー

```
q.enqueueTask(krnl, NULL, &event_sp);
```

この時点で待機しない場合は、cl::Event オブジェクトではなくNULL を渡します。

最後に、カーネルが処理を完了したら、メモリをホストに戻し、CPU から新しい値にアクセスできるようにします。これは、コード 4.9 で実行します。

コード 4.9: データをホストに戻す

```
q.enqueueMigrateMemObjects(outBufVec, CL_MIGRATE_MEM_OBJECT_HOST, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

この例では、同期化を待機します。これらのエンキュー関数を呼び出すときに、エントリをノンブロッキング方式でコマンドキューに配置するということに注意してください。転送をエンキューした直後にバッファーにアクセスしようとすると、リードバックは完了しています。

例 0 からFPGA コンフィギュレーションを除外すると、カーネルを実行するために新しく次を追加します。

1. (60 ～ 63 行目): 通常の方法でバッファーを割り当てます。これより良い方法がありますが、アクセラレーションを初めて試す多くの人がこの方法を使用します。
2. (81 ～ 102 行目): 割り当て済みバッファーを cl::Buffer オブジェクトにマップします。
3. (108 行目): 入力バッファー (a および b) のAlveo デバイスグローバルメモリへの移動をエンキューします。
4. (113 ～ 116 行目): カーネル引数、バッファーおよびスカラー値の両方を設定します。
5. (120 行目): カーネルを実行します。
6. (126 ～127 行目): カーネルの結果を CPU のホストメモリにリードバックし、読み出しの完了を同期化します。

これが実際のアプリケーションであれば、同期化は 1 回のみ必要です。前述のように、この例では、ワークフローのさまざまな操作のタイミングををより良くレポートするため、いくつかの同期化を使用しています。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./01_simple_malloc alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 1: Vector Add with Malloc() --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: ’./alveo_examples.xclbin’
Running kernel test with malloc()ed buffers
WARNING: unaligned host pointer ’0x154f7909e010’ detected,
this leads to extra memcpy
WARNING: unaligned host pointer ’0x154f7789d010’ detected,
this leads to extra memcpy
WARNING: unaligned host pointer ’0x154f7609c010’ detected,
this leads to extra memcpy

Simple malloc vadd example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  247.371 ms
Allocating memory buffer           :    0.030 ms
Populating buffer inputs           :   47.955 ms
Software VADD run                  :   35.706 ms
Map host buffers to OpenCL buffers :   64.656 ms
Memory object migration enqueue    :   24.829 ms
Set kernel arguments               :    0.009 ms
OCL Enqueue task                   :    0.064 ms
Wait for kernel to complete        :   92.118 ms
Read back computation results      :   24.887 ms
```

境界に揃えられていないポインターに関する警告がいくつか表示されています。割り当てに関して何もしていないので、Alveo カードとの間で転送されるバッファーはどれも、Alveo DMA エンジンで必要なように、4 KiB 境界に揃えられていません。そのため、転送の前にバッファーの内容を境界に揃うようにコピーする必要がありますが、この操作には時間がかかります。

ここからは、これらの数値に注意します。複数の実行間でレイテンシに多少の変動はありますが、各エリアの差に注目します。表 4.1 にベースラインを示します。

表 4.1: タイミング サマリ - 例 1

操作	例 **1**
OCL 初期化	247.371 ms
バッファーの割り当て	30 µs
バッファーへの値の挿入	47.955 ms
ソフトウェア VADD	35.706 ms
バッファーのマップ	64.656 ms
バッファーの書き出し	24.829 ms
カーネル引数の設定	9 µs
カーネルの実行時間	92.118 ms
バッファーの読み込み	24.887 ms
$\Delta_{Alveo \rightarrow CPU}$	-418.228 ms
$\Delta_{Alveo \rightarrow CPU}$(アルゴリズムのみ)	-170.857 ms

この結果は、それほど良いものではありませんが、これが得られる最高の結果ではありません。ザイリンクスがこれを開発したのにはもちろん理由があります。それでは、改善できないか見てみましょう。

追加演習

この演習に、追加で次のことを試してみてください。

- 割り当てるバッファーのサイズを変えてみます。バッファーサイズと個々の操作のタイミングとの間にどのような関係がありますか。どれも同じレートでスケーリングしますか。
- 各ステップ間の同期化を削除すると、実行時間にどのように影響しますか。
- Alveo からホストに最後にバッファーをコピーした後の同期化を削除すると、どうなりますか。

学習ポイント

- ここでもまた、FPGA コンフィギュレーション「税」を支払う必要があります。これを補うため、実行時間を CPU よりも 250 ms 以上短縮する必要があります。1 つのバッファーを処理するだけのこの単純な例では、CPU を上回るスピードを達成するのは無理です。
- 単純に割り当てられたメモリは、メモリのコピーに時間がかかるので、アクセラレータに移動するには適しません。この後の例で、この影響を調べてみます。
- OpenCL はコマンドキューで機能します。同期化の方法およびタイミングは開発者しだいですが、CPU がバッファーのデータにアクセスする前に同期しているようにするため、Alveo グローバルメモリからバッファーをリードバックするときに注意が必要です。

例2: 境界に揃えられたメモリ割り当て

概要

前の例では、単純にメモリを割り当てただけですが、DMA エンジンでは、バッファーが 4 KiB のページ境界に揃えられていることが必要です。バッファーが境界に揃えられていない場合、ランタイムによりバッファーがその内容が境界に揃うようにコピーされます。

これは時間のかかる操作ですが、具体的にどの程度の時間がかかるのでしょうか。そして、境界に揃えられたメモリはどのように割り当てることができるでしょうか。

キーコード

この例は、バッファーを割り当てた例 1 のコードを4 行変更するだけなので、比較的短いです。境界に揃えられたメモリを割り当てるにはさまざまな方法がありますが、ここでは POSIX 関数 posix_memalign() を使用します。コード 4.4 に示す前の例のメモリ割り当てを、コード 4.10 に置き換えます。また、メモリのコード例には示されていない追加ヘッダーを含める必要もあります。

コード 4.10: 境界に揃えられたバッファーの割り当て

```
uint32_t *a, *b, *c, *d = NULL;
posix_memalign((void **)&a, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&b, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&c, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&d, 4096, BUFSIZE * sizeof(uint32_t));
```

posix_memalign() への呼び出しでは、前述のとおり、要求されたアライメント (4 KiB) を渡します。変更はこれだけです。CPU ベースラインの VADD 関数によってのみ使用されるバッファー d も含め、すべてのバッファーの割り当てが変更されています。これがアクセラレータと CPU の両方のランタイムパフォーマンスに与える影響を確認してみましょう。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./02_aligned_malloc alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 2: Vector Add with Aligned Allocation --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test with aligned virtual buffers

Simple malloc vadd example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  256.254 ms
Allocating memory buffer           :    0.055 ms
Populating buffer inputs           :   47.884 ms
Software VADD run                  :   35.808 ms
Map host buffers to OpenCL buffers :    9.103 ms
Memory object migration enqueue    :    6.615 ms
Set kernel arguments               :    0.014 ms
OCL Enqueue task                   :    0.116 ms
Wait for kernel to complete        :   92.110 ms
Read back computation results      :    2.479 ms
```

一見したところでは、パフォーマンスがかなり向上しているようです。これらの結果を例 **1** の結果と比較して、どのように変化したかを確認してみましょう。表 4.2 に詳細を示します。わかりやすくするため、実行ごとの変動は含めていません。

表 4.2: タイミング サマリ - 例 2

操作	例 **1**	例 **2**	$\Delta_{1\rightarrow2}$
OCL 初期化	247.371 ms	256.254 ms	-
バッファー割り当て	30 µs	55 µs	25 µs
バッファーへの値の挿入	47.955 ms	47.884 ms	-
ソフトウェア VADD	35.706 ms	35.808 ms	
バッファーのマップ	64.656 ms	9.103 ms	-55.553 ms
バッファーの書き出し	24.829 ms	6.615 ms	-18.214 ms
カーネル引数の設定	9 µs	14 µs	-
カーネルの実行時間	92.118 ms	92.110 ms	-
バッファーの読み込み	24.887 ms	2.479 ms	-22.408 ms
$\Delta_{Alveo\rightarrow CPU}$	-418.228 ms	-330.889 ms	87.339 ms
$\Delta_{Alveo\rightarrow CPU}$ (アルゴリズムのみ)	-170.857 ms	-74.269 ms	96.588 ms

良い結果です。たった 4 行のコードを変更しただけで、実行時間を約 100 ms 短縮できました。それでもまだ CPU のほうが高速ですが、メモリの割り当て方法を少し変えただけで、非常に大きな改善が見られました。これで、アライメントに必要なのはメモリコピーだけになりました。バッファーを割り当てるとき、数マイクロ秒余分に時間をかけてバッファーが境界に揃えられるようすると、後でこれらのバッファーが使用されるときにその何十倍もの時間を節約できます。

このユースケースでは、予想どおり、ソフトウェアの実行時間は同じです。割り当てられたメモリのアライメントを変更しているだけで、ユーザー空間のメモリ割り当ては通常のものです。

追加演習

この演習に、追加で次のことを試してみてください。

- 割り当てるバッファーのサイズを変えてみます。前の例から導きかれた関係は、この例でも有効ですか。
- 境界が揃えられたメモリを割り当てる OCL API ではない別の方法でも試してみます。実行ごとの変動以上の違いはありますか。

学習ポイント

- 境界に揃えられていないメモリを使用すると、パフォーマンスが低下します。Alveo カードと共有するバッファーは、常に境界に揃えるようにします。

改善が見られたので、今度は OpenCL API を使用してメモリを割り当ててみましょう。

例 3: OpenCL を使用したメモリ割り当て

概要

メモリがページ境界に揃うように割り当てることで、パフォーマンスが最初のコンフィギュレーションよりも大幅に改善できました。別のワークフローとして、OpenCL と XRT によりバッファーが割り当てられるようにし、そのバッファーをユーザー空間のポインターにマップして、アプリケーションで使用できるようにする方法があります。その方法を使用して、タイミングにどのように影響するかを見てみましょう。

キーコード

概念的にはわずかな変更ですが、この例では、例 **2** より大きなコード変更が必要になります。その理由は主に、標準ユーザー空間のメモリ割り当てを使用する代わりに、OpenCL ランタイムによりバッファーが割り当てられるようにするからです。バッファーが割り当てられたら、バッファー内のデータにアクセスできるように、バッファーをユーザー空間にマップする必要があります。
ここでは、コード 4.10 をコード 4.11 のように変更します。

コード 4.11: OpenCL での境界に揃えられたバッファーの割り当て

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer d_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
inBufVec.push_back(a_buf);
inBufVec.push_back(b_buf);
outBufVec.push_back(c_buf);
```

プログラムの早い段階で OpenCL バッファーオブジェクトを割り当てており、ユーザー空間のポインターはまだありません。 それでも、これらのバッファー オブジェクトを enqueueMigrateMemObjects() などの OpenCL 関数に渡すことは可能です。この時点で補助ストレージが割り当てられますが、そのユーザー空間ポインターはまだありません。
cl::Buffer コンストラクターへの呼び出しは、前とほぼ同じように見えます。変更されているのは 2 点のみで、既存のバッファーを使用する代わりに新しいバッファーを割り当てることをランタイムに指示するため、CL_MEM_USE_HOST_PTR フラグではなくて、CL_MEM_ALLOC_HOST_PTR フラグを渡します。また、新しいバッファーを割り当てるためユーザーバッファーにポインターを渡す必要はもうないので、代わりに NULL を渡します。

次に、ソフトウェアですぐに使用するバッファー a、b、d へのユーザー空間ポインターに OpenCL バッファーをマップする必要があります。この時点では、c へのポインターをマップする必要はありません。後で、カーネル実行後にそのバッファーから読み出す必要があるときにマップできます。これには、コード 4.12 を使用します。

コード 4.12: 境界に揃えられたバッファーのユーザー空間ポインターへのマップ

```
uint32_t *a = (uint32_t *)q.enqueueMapBuffer(a_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
uint32_t *b = (uint32_t *)q.enqueueMapBuffer(b_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
```

```
                                                BUFSIZE * sizeof(uint32_t));
uint32_t *d = (uint32_t *)q.enqueueMapBuffer(d_buf,
                                                CL_TRUE,
                                                CL_MAP_WRITE | CL_MAP_READ,
                                                0,
                                                BUFSIZE * sizeof(uint32_t));
```

マップしたら、通常どおりユーザー空間ポインターを使用してバッファーの内容にアクセスできます。ただし、OpenCL ランタイムではオープンになっているバッファーのカウントが参照されるので、マップするバッファーごとに enqueueUnmapMemObject() を呼び出す必要があります。

カーネルの実行フローは同じですが、入力バッファーをデバイスに戻すときに違いが現れます。移動を手動でエンキューする代わりに、バッファーをマップするだけです。OpenCL ランタイムにより、バッファーの内容が現在 Alveo デバイスのグローバルメモリ内にあることが認識され、バッファーがホストに戻されます。コーディングスタイルは開発者しだいですが、基本的には、バッファー c をホストに戻すには、コード4.13 で十分です。

コード 4.13: カーネル出力のユーザー空間ポインターへのマップ

```
uint32_t *c = (uint32_t *)q.enqueueMapBuffer(c_buf,
                                                CL_TRUE,
                                                CL_MAP_READ,
                                                0,
                                                BUFSIZE * sizeof(uint32_t));
```

最後に、ランタイムでメモリオブジェクトが完全に消去されるようにするため、これらのメモリオブジェクトのマップを解除する必要があります。これは、前のようにバッファーに free() を設定するのではなく、プログラムの最後に実行します。この操作は、4.14 に示すように、コマンドキューが終了する前に実行する必要があります。

コード 4.14: OpenCL での割り当てバッファーのマップ解除

```
q.enqueueUnmapMemObject(a_buf, a);
q.enqueueUnmapMemObject(b_buf, b);
q.enqueueUnmapMemObject(c_buf, c);
q.enqueueUnmapMemObject(d_buf, d);
q.finish();
```

このユースモデルの主なワークフローをまとめると、次のタスクを実行する必要があります。

1. (63 ～90 行目): CL_MEM_ALLOC_HOST_PTR フラグを使用してバッファーを割り当てます。
2. (94 ～108 行目): 入力バッファーをユーザー空間ポインターにマップします。
3. カーネルを通常どおりに実行します。
4. (144 ～148 行目): 出力バッファーをマップしてホストメモリに戻します。
5. (181 ～185 行目): すべてのバッファーを使用し終えたら、正しく消去されるようにすべてのバッファーのマップを解除します。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./03_buffer_map alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 3: Allocate and Map Contiguous Buffers --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test with XRT-allocated contiguous buffers

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization               :  247.460 ms
Allocate contiguous OpenCL buffers :   30.365 ms
Map buffers to userspace pointers  :    0.222 ms
Populating buffer inputs           :   22.527 ms
Software VADD run                  :   24.852 ms
Memory object migration enqueue    :    6.739 ms
Set kernel arguments               :    0.014 ms
OCL Enqueue task                   :    0.102 ms
Wait for kernel to complete        :   92.068 ms
Read back computation results      :    2.243 ms
```

表 4.3: タイミング サマリ - 例 3

操作	例 2	例 3	$\Delta_{2\rightarrow3}$
OCL 初期化	256.254 ms	247.460 ms	-
バッファーの割り当て	55 µs	30.365 ms	30.310 ms
バッファーへの値の挿入	47.884 ms	22.527 ms	-25.357 ms
ソフトウェア VADD	35.808 ms	24.852 ms	-10.956 ms
バッファーのマップ	9.103 ms	222 µs	-8.881 ms
バッファーの書き出し	6.615 ms	6.739 ms	-
カーネル引数の設定	14 µs	14 µs	-
カーネルの実行時間	92.110 ms	92.068 ms	-
バッファーの読み込み	2.479 ms	2.243 ms	-
$\Delta_{Alveo\rightarrow CPU}$	-330.889 ms	-323.996 ms	-6.893 ms
$\Delta_{FPGA\rightarrow CPU}$(アルゴリズムのみ)	-74.269 ms	-76.536 ms	-

ここでの高速化を期待していたかもしれませんが、特定の操作で高速化は見られず、システム内でレイテンシが移動しただけです。つまり、別の銀行口座から税金を支払ったようなもので、税金から逃れることはできません。プロセッサとカーネルのメモリマップが 1 つに統合されているエンベデッドシステムであれば、大きな違いが見られますが、サーバークラスの CPU では違いは見られません。

この方法でバッファーをあらかじめ割り当てておくのは時間がかかりましたが、通常はアプリケーションのクリティカルパスにバッファーを割り当てるのは好ましくありません。ただし、この方法では、ランタイムでのバッファーの使用はかなり高速になります。
このメモリにアクセスするのになぜCPU のほうが高速なのでしょうか。ここまで説明してきませんでしたが、この API を介してメモリを割り当てると、仮想アドレスが物理アドレスに固定されます。これにより、CPU とDMA の両方でより効率よくメモリにアクセスできるようになります。ただし、これにもコストがかかります。割り当てには時間がかかり、小型のバッファーを多数割り当てると使用可能なメモリがフラグメント化されてしまうリスクがあります。

通常は、バッファーはアプリケーションのクリティカルパス以外に割り当てるべきで、この方法を正しく使用すれば、負荷を高パフォーマンス部分から別の部分にシフトできます。

追加演習

この演習に、追加で次のことを試してみてください。

- 割り当てるバッファーのサイズを変えてみます。前の例から導きかれた関係は、この例でも有効ですか。
- メモリの割り当てと転送のエンキューに、ほかのシーケンスを試してみます。
- 入力バッファーを変更し、カーネルを再実行すると、どうなりますか。

学習ポイント

- OpenCL および XRT の API を使用すると、領域によってパフォーマンスが向上しますが、根本的にはアクセラレーション税から逃れることはできません。

カーネルの実行に 1 番時間がかかっていますが、それは簡単にスピードアップできます。

例 4: データパスの並列処理

概要

全体的なシステムスループットは大きく改善しましたが、ボトルネックがアクセラレータ自体であることが明らかになりました。デザインの最適化には2 つの方法があることを思い出してください。アムダールの法則に従って演算をより高速に実行するためにリソースを投入する方法と、グスタフソンの法則に従って基本的な操作を並列処理しする方法です。
ここでは、アムダールの法則に従って最適化してみましょう。ここまでは、アクセラレータは CPU と同じアルゴリズムに従って、クロックごとに 32 ビットの加算を1 回実行していました。ただし、CPUクロックははるかに高速なので (しかも PCIe 上でデータを転送する必要がない)、CPU のほうがスピードでは勝っていました。これを逆転させてみましょう。

DDR コントローラーのには 512 ビット幅のインターフェイスが含まれています。アクセラレータでデータフローを並列処理すると、1 クロックごとに1 個の配列要素ではなく、16 個の配列要素を処理できます。つまり、入力をベクター化するだけで、スピードを16 倍にすることが可能です。

キーコード

この例で、引き続きホストソフトウェアに焦点を置き、カーネルをブラックボックスとして処理すると、例 3 のコードと実質的に同じコードになります。カーネル名を vadd からwide_vadd に変更するだけです。
コードが同じなので、XRT および OpenCL のメモリを操作する別の概念を紹介しましょう。Alveo カードには、使用可能なメモリバンクが 4 つあります。これらの単純なアクセラレータでは必ずしも必要ではありませんが、各インターフェイスで使用可能な帯域幅を最大にするため、操作をこれらのアクセラレータに均等に分散するのが望ましい場合があります。この単純なベクター加算例では、図 4.1 に示すトポロジを使用しています。

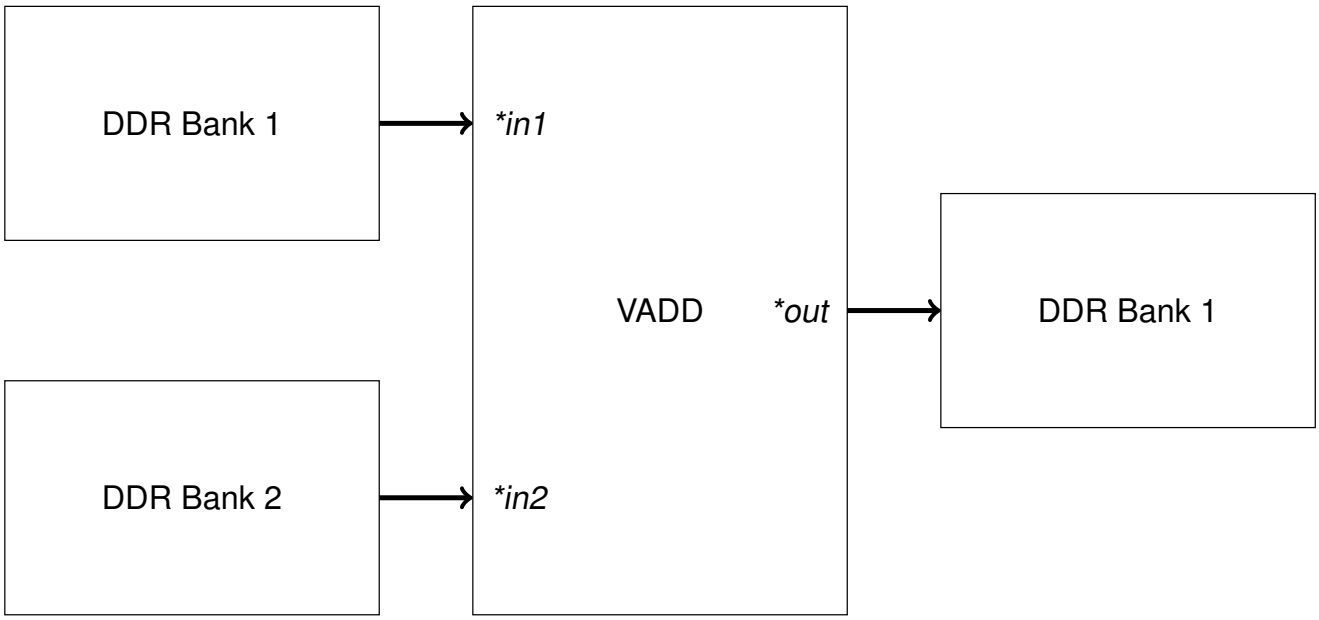

図 4.1: Wide VADD のメモリ接続

このようにすると、異なる外部メモリバンクを使用して、広帯域幅のトランザクション同時に実行できるようになります。短い読み出しおよび書き込みを何回も実行するより、長いバーストを実行する方が良いパフォーマンスが得られますが、同じメモリで 2 つの操作を同時に実行することは根本的には不可能です。

コマンド ライン オプションを使用してハードウェアの接続性を指定することは簡単ですが、実際にハードウェアにバッファーを転送するには、XRT にどのメモリを使用するかを指定する必要があります。ザイリンクスでは、標準 OpenCL ライブラリを拡張することにより、バッファー割り当てのフラグ CL_MEM_EXT_PTR_XILINX を構造体 cl_mem_ext_ptr_t と組み合わせて使用します。これはコード 4.15 のようになります。

コード 4.15: XRT での外部メモリ バンクの指定

```
cl_mem_ext_ptr_t bank1_ext, bank2_ext;
```

```
bank2_ext.flags = 2 | XCL_MEM_TOPOLOGY;
bank2_ext.obj   = NULL;
bank2_ext.param = 0;
bank1_ext.flags = 1 | XCL_MEM_TOPOLOGY;
bank1_ext.obj   = NULL;
bank1_ext.param = 0;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank1_ext,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank2_ext,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank1_ext,
                 NULL);
```

このコードは 4.11 に非常によく似ていますが、違いは cl_mem_ext_ptr_t オブジェクトを cl::Buffer コンストラクターに渡している点と、flags フィールドを使用して特定バッファーにどのメモリバンクを使用するのかを指定している点です。このような単純な例でこのようにする必要はありませんが、この例は前の例と構造的に非常によく似ているので、この方法を学ぶのによい機会です。これは、ワークロードの大きいシステムでパフォーマンスを最適化するのに非常に有益な手法となることがあります。

wide_vadd カーネルに変更したことと、バッファーのサイズを 1 GiB に増やしたことを除けば、コードは同じです。この変更は、ハードウェアとソフトウェアの違いを強調するために加えられています。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./04_wide_vadd alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 4: Parallelizing the Data Path --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test XRT-allocated buffers and wide data path:


OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization               :  244.463 ms
Allocate contiguous OpenCL buffers :   37.903 ms
Map buffers to userspace pointers  :    0.333 ms
Populating buffer inputs           :   30.033 ms
Software VADD run                  :   21.489 ms
Memory object migration enqueue    :    4.639 ms
Set kernel arguments               :    0.012 ms
OCL Enqueue task                   :    0.090 ms
Wait for kernel to complete        :    9.003 ms
Read back computation results      :    2.197 ms
```

表 4.4: タイミングサマリ- 例 3

操作	例 3	例 4	$\Delta_{3\rightarrow4}$
OCL 初期化	247.460 ms	244.463 ms	-
バッファーの割り当て	30.365 ms	37.903 ms	7.538 ms
バッファーへの値の挿入	22.527 ms	30.033 ms	7.506 ms
ソフトウェアVADD	24.852 ms	21.489 ms	-3.363 ms
バッファーのマップ	222 µs	333 µs	-
バッファーの書き出し	6.739 ms	4.639 ms	-
カーネル引数の設定	14 µs	12 µs	-
カーネルの実行時間	92.068 ms	9.003 ms	-83.065 ms
バッファーの読み込み	2.243 ms	2.197 ms	-
$\Delta_{Alveo\rightarrow CPU}$	-323.996 ms	-247.892 ms	-76.104 ms
$\Delta_{FPGA\rightarrow CPU}$(アルゴリズムのみ)	-76.536 ms	5.548 ms	-82.084 ms

CPU よりも速いスピードを達成できました。

ただし、注意すべき点がいくつかあります。まず、データを FPGA との間で転送するのに必要な時間は変わっていません。これまでのセクションで説明したように、この時間は、メモリトポロジ、データ量、システム全体のメモリ帯域幅の使用に基づいて一定になります。特にクラウドデータセンターなどの仮想化された環境では、実行ごとに若干の変動が見られますが、実行時間が固定された操作とみなすことができます。

興味深いのは、達成された高速化の量です。クロックごとのデータパス幅を 16 ワードに広げても、16 倍の高速化が得られるわけではありません。カーネルではクロックごとに 16 ワード処理されていますが、タイミング結果を見ると、外部DDR レイテンシが累積されているのがわかります。各インターフェイスはデータをバースト入力して処理し、それをバースト出力しますが、DDR とのやりとりでレイテンシが発生するため、実際には1 ワードのインプリメンテーションと比較して 10 倍の高速化しか達成できていません。

ベクター加算は簡単すぎるので、根本的な問題に直面してしまっています。vadd の演算は $O(N)$ ですが、非常に単純な $O(N)$ です。その結果、演算ではなく I/O の帯域幅が制限となります。$O(N^x)$ などの入れ子のループや、$O(N)$ アルゴリズムであっても計算量の多いフィルターなどであれば、メモリにそれほど頻繁にアクセスせずに、FPGA ファブリック内で大量の計算が実行されるので、大幅なアクセラレーションを達成できます。アクセラレーションに最適なアルゴリズムは、データ転送がほとんどなく、計算量が非常に多いものです。

ここで、より大型のバッファーを使用してみましょう。バッファーサイズをコード 4.16 のように変更します。

コード 4.16: 大きいメモリ サイズ

```
#define BUFSIZE (1024 * 1024 * 256) // 256*sizeof(uint32_t) = 1 GiB
```

これを再ビルドして実行し、その結果をメトリクスを少なくした表 4.5 に示します。これまでの演習で OpenCL 初期化については理解できていると思うので、ここではアルゴリズムのパフォーマンスのみを比較します。この初期化時間が重要ではないということではありませんが、これが初期化操作中に 1 回だけ実行されるようにアプリケーションを構築するべきです。

バッファーの割り当ておよび初期化にかかる時間も示しません。これらも、アプリケーションの設定中に実行すべき操作です。アプリケーションのクリティカルパスに大型のバッファーを割り当てるのが習慣となっているのであれば、ハードウェアアクセラレーションを適用しようとするよりは、そのアプリケーションの構造を考え直した方が効率が良いでしょう。

表 4.5: 1 GiB バッファーのタイミング サマリ - 例 4

操作	例 **4**
ソフトウェア VADD	820.596 ms
バッファー PCIe TX	383.907 ms
VADD カーネル	484.050 ms
バッファー PCIe RX	316.825 ms
ハードウェア VADD (合計)	1184.897 ms
$\Delta_{Alveo \rightarrow CPU}$	364.186 ms

これまでうまくいっていたのに、それが台無しになってしまいました。この大型バッファーで

は、CPU のスピードを上回ることがまったくできていません。何が起きたのでしょうか。

結果の値をよく見てみると、ハードウェアカーネルはかなり効率よくデータを処理していますが、FPGA とのデータ転送に時間が取られています。全体的な実行のタイムラインを描くと、図 4.2 のようになります。

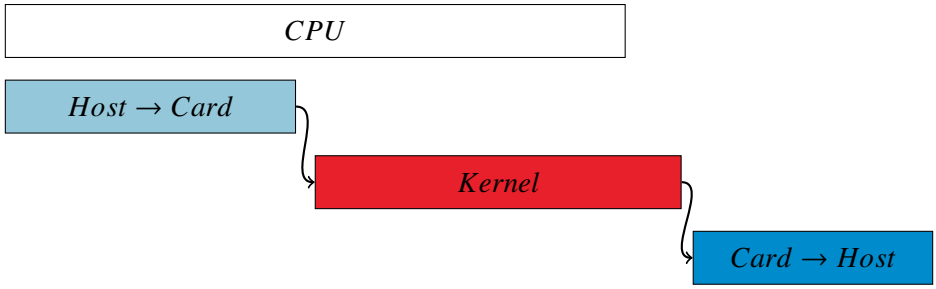

図 4.2: カーネル実行のライムライン

この図は厳密なものではありませんが、相対的な関係を示しています。結果を詳しく見ると、CPU のスピードを上回るには、カーネル全体を約 120 ms で完了するよう実行する必要があります。それを実現するには、4 倍のスピードで実行してクロックごとに処理するデータ量を 4 倍に増やすか、4 つのアクセラレータを並列実行する必要があります。どちらを選択すべきでしょうか。それを次の例で見てみましょう。

追加演習

この演習に、追加で次のことを試してみてください。

- バッファーのサイズを変えてみます。CPU のほうが速くなる点を見つけられますか。
- SDAccel タイムライントレースを記録して確認してみます。この手順は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207: 英語版、日本語版) を参照してください。

学習ポイント

- カーネルで並列処理を使用すると非常に良い結果が得られる可能性はありますが、データ転送が実行時間の大部分を占めないようにする必要があります。データ転送に時間がかっている場合は、目標の実行時間を超えないようにします。
- 単純な計算は、ほかのタスクを実行するためにプロセッサを解放しようとしているのでなければ、必ずしもアクセラレーションに適しているとは言えません。アクセラレーションに適しているのは、特に入れ子のループのような $O(N^x)$ アルゴリズムなど、複雑でワークロードの大きいアゴリズムです。
- 処理を開始する前にすべてのデータが転送されるまで待機する必要がある場合は、カーネルが非常に最適化されていても、全体的なパフォーマンスターゲットを達成できない可能性があります。

データパスを並列処理するだけでは十分ではないことがわかったので、データ転送にかかる時間をどうすればよいかを見ていきましょう。

例 5: 計算および転送の最適化

概要

前の例では、データがあるサイズを超えると、アプリケーションとのデータ転送がボトルネックになりました。PCIe 上での転送には通常一定の時間がかかるので、wide_vadd カーネルのインスタンスを 4 つ作成し、それらすべてをエンキューして、大型のバッファーを並列処理すれば良いと考えるかもしれません。
これは、PCIe 上で大量のデータを広帯域幅のメモリに転送し、配列された複数のエンベデッドプロセッサを使用してデータを非常に幅広いパターンで処理するという従来の GPU モデルです。

FPGA では通常、この問題を別の方法で解決する必要があります。各クロックで 512 ビットのデータをカーネルがフルに処理できる場合、一番の問題は並列処理ではなく、その処理能力に見合うようにデータを転送できるかどうかです。DDR とのデータ転送にバーストを使用すると、帯域幅の上限にすぐに達してしまいます。複数のコアを並列に配置して同じバッファーで継続的に並行処理すると、帯域幅の競合が発生し、これらのコアが低速になります。これでは逆効果です。
どうすればよいでしょうか。wide_vadd カーネルはアクセラレーションには適していないことを思い出してください。計算が単純すぎるので、FPGA の並列処理機能を活用できません。*A* + *B* だけのアルゴリズムではできることは限られています。このような単純なアルゴリズムであっても、最適化手法をいくつか紹介することはできます。
そのため、wide_vadd カーネルのハードウェアデザインに少し工夫を加えています。バスからの生のデータを使用する代わりに、FPGA のブロックRAM (非常に高速な SRAM) を使用してデータをバッファリングし、演算を実行しています。これにより連続するバースト間に少し時間を吸収できるので、それを利用します。

図 4.1 に示すように、インターフェイスを複数の DDR バンクに分割したことを思い出してください。PCIe の帯域幅は、Alveo カード上の DDR メモリの帯域幅の合計よりもかなり狭く、2 つの異なる DDR バンクにデータを転送しているため、これらの DDR バンク間でインターリーブが実行されます。その隙間に処理を挿入することが可能なので、データ転送が完全に完了するのを待つのではなく、データが到達したらすぐに処理を開始し、データを戻すことができます。その結果、タイムラインは図 4.3 に示すようになります。

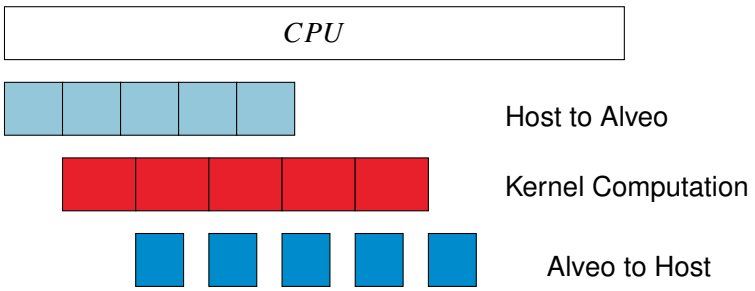

図 4.3: Kカーネル実行のタイムライン (最適化済み)

バッファーをこのように分割し、適切な分割数を選択することにより、アプリケーションの実行時間と転送時間のバランスをとることができ、スループットが大幅に向上します。同じハードウェアカーネルを使用して、これをホストコードで設定するのに何が必要かを見てみましょう。

キーコード

コードのアルゴリズムフローはほぼ同じです。転送をエンキューする前に、バッファーをループ処理し、それを分割します。ただし、いくつかの一般的な規則に従う必要があります。まず、効率よく転送するため、バッファーを境界に揃えて分割するようにします。次に、分割しても意味がないような小さなバッファーは分割しないようにします。バッファーの数を設定するため定数 **NUM_BUFS** を定義し、コード 4.17 に示すように、バッファーを分割するための新しい関数を記述します。

コード 4.17: XRT バッファーの分割

```
int subdivide_buffer(std::vector<cl::Buffer> &divided,
                     cl::Buffer buf_in,
                     cl_mem_flags flags,
                     int num_divisions)
{
    // Get the size of the buffer
    size_t size;
    size = buf_in.getInfo<CL_MEM_SIZE>();

    if (size / num_divisions <= 4096) {
        return -1;
    }

    cl_buffer_region region;

    int err;
    region.origin = 0;
    region.size   = size / num_divisions;

    // Round region size up to nearest 4k for efficient burst behavior
    if (region.size % 4096 != 0) {
        region.size += (4096 - (region.size % 4096));
    }

    for (int i = 0; i < num_divisions; i++) {
        if (i == num_divisions - 1) {
            if ((region.origin + region.size) > size) {
                region.size = size - region.origin;
            }
        }
        cl::Buffer buf = buf_in.createSubBuffer(flags,
                                                CL_BUFFER_CREATE_TYPE_REGION,
                                                &region,
                                                &err);
        if (err != CL_SUCCESS) {
            return err;
        }
        divided.push_back(buf);
        region.origin += region.size;
    }

    return 0;
}
```

ここでは、バッファーを NUM_BUFS で設定した回数だけループ処理し、作成する各サブバッファーに対して cl::Buffer.createSubBuffer() を呼び出しています。cl_buffer_region は、作成するサブバッファーの開始アドレスおよびサイズを定義します。サブバッファーはオーバーラップさせることは可能ですが、ここではオーバーラップさせません。

コード 4.18 に示すように、複数の操作をエンキューするために使用できるcl::Buffer オブジェクトのベクターを戻します。

コード 4.18: 分割した XRT バッファーのエンキュー

```
int enqueue_subbuf_vadd(cl::CommandQueue &q,
                        cl::Kernel &krnl,
                        cl::Event &event,
                        cl::Buffer a,
                        cl::Buffer b,
                        cl::Buffer c)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;

    static std::vector<cl::Event> tx_events, rx_events;

    std::vector<cl::Memory> c_vec;
    size_t size;
    size = a.getInfo<CL_MEM_SIZE>();

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(a);
    in_vec.push_back(b);
    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);
    if (tx_events.size() > 1) {
        tx_events[0] = tx_events[1];
        tx_events.pop_back();
    }

    krnl.setArg(0, a);
    krnl.setArg(1, b);
    krnl.setArg(2, c);
    krnl.setArg(3, (uint32_t)(size / sizeof(uint32_t)));

    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);
    if (rx_events.size() == 1) {
        krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }
    c_vec.push_back(c);
    q.enqueueMigrateMemObjects(c_vec,
                               CL_MIGRATE_MEM_OBJECT_HOST,
                               &krnl_events,
                               &event);
    rx_events.push_back(event);

    return 0;
}
```

コード 4.18 は、基本的に以前の実行と同じイベントシーケンスになります。

1. ホストメモリから Alveo メモリへのバッファーの移動をエンキューします。
2. カーネル引数を現在のバッファーに設定します。

3. カーネルの実行をエンキューします。
4. 結果を戻す転送操作をエンキューします。

ただし、今回は実際にキュー方式で順次実行している点が違います。これまでの例とは異なり、イベントが完全に終了するまで待機しません。待機すると、せっかくのパイプライン処理が無駄になるからです。そこでこの例では、イベントベースの依存関係を利用します。まず、cl::Event オブジェクトを使用して、イベントを実行する前に完了する必要のあるイベントを定義するイベントのチェーンを構築します。リンクされていないイベントは、いつでもスケジュールできます。
カーネルの複数実行をエンキューし、すべてが完了するのを待ちます。これにより、さらに効率の良いスケジューリングになります。このキュー方式を使用して例 4と同じ構造を構築すると、そのときと同じ結果になります。それは、データをすべて転送する前に、処理を安全に開始できるかどうかはランタイムにはわかり得ないからです。何が実行可能で実行不可能かを設計者がスケジューラに指示する必要があります。

そして最後に、もう 1 つ重要な設定しておかないと、上記の操作は正しいシーケンスで実行されません。コマンドキューを作成するときに、
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE フラグを渡してアウトオブオーダーコマンドキューを使用することを指定する必要があります。
それ以外は、例 5 のコードは前と似ています。main() から直接 API を読み出す代わりに、これらの関数を呼び出すようになりますが、それ以外は変わっていません。
ここで興味深いのが、バッファー C をユーザー空間にマップし直す際に、個々のサブバッファーを処理する必要がないということです。これらのサブバッファーは既にホストメモリに戻されており、サブバッファーを作成したときにその基になるポインターは変更されないので、サブバッファーがあっても、元の親バッファーを使用すればよいのです。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーショ

```
./05_pipelined_vadd alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 5: Pipelining Kernel Execution --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous buffers
    and wide VADD (16 values/clock)

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  263.001 ms
Allocate contiguous OpenCL buffers :  915.048 ms
Map buffers to userspace pointers  :    0.282 ms
Populating buffer inputs           : 1166.471 ms
Software VADD run                  : 1195.575 ms
Memory object migration enqueue    :    0.441 ms
Wait for kernels to complete       :  692.173 ms
```

表 4.6 に、結果を前回の実行と比較します。

表 4.6: 1 GiB バッファー タイミング サマリ - パイプライン処理 vs. 順次処理

操作	例 **4**	例 **5**	$\Delta_{4\rightarrow5}$
ソフトウェア VADD	820.596 ms	1166.471 ms	345.875 ms
ハードウェア VADD (合計)	1184.897 ms	692.172 ms	-492.725 ms
$\Delta_{Alveo\rightarrow CPU}$	364.186 ms	-503.402 ms	867.588 ms

今回は目的を果たすことができました。このマージンを見てください。

これが逆転することはあり得ません。これで演習は終わりにしてもよいでしょう。もうほかに問題が隠れていることはないでしょう。

追加演習

この演習に、追加で次のことを試してみてください。

- バッファーのサイズを変えてみます。この追加演習でもCPU の方が速くなる点はありますか。
- もう一度トレースを記録してみましょう。違いはありますか。サブバッファーの数は実行時間にどのように影響しますか。

学習ポイント

- データ転送およびコマンドキューを合理的に管理すると、大幅な高速化につながる可能性があります。

例 6: OpenMP の使用

概要

例 5 で演習は終わったと思ったかもしれませんが、そうはいきません。前述のように、vadd がプロセッサよりも高速になることはありません。vadd は単純すぎるからです。CPU はデータを転送する必要がなく、ローカルキャッシュを使用できるので、スピードでは常に勝ります。

前のセッションでは良い結果が得られましたが、それはあくまで机上の結果にすぎません。マーケティングの面ではここで終わりにしたいところですが、エンジニアリングの面では、お伝えすべきことがまだあります。

キーコード

アルゴリズムが単純な場合、アクセラレータが CPU を上回ることはありません。OpenMP を使用してプロセッサループを並列処理してみましょう。CPU コードにヘッダー omp.h を含め、コード 4.19 に示すように OpenMP プラグマを適用します。

コード 4.19: OpenMP を使用してソフトウェアをパラメーター化

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

やることはそれだけです。GCC に渡す必要のあるコマンドラインフラグがいくつかありますが、それらは CMake で自動的に処理されるので (OpenMP がインストールされている場合)、開発者は直接ビルドして実行できます。この例のコードは、それ以外は例 5 のコードと同じです。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./06_wide_processor alveo_examples
```

プログラムにより、次のようなメッセージが表示されます。

```
-- Example 6: VADD with OpenMP --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous buffers
    and wide VADD (16 values/clock), with software OpenMP

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  253.898 ms
Allocate contiguous OpenCL buffers :  907.183 ms
Map buffers to userspace pointers  :    0.307 ms
Populating buffer inputs           : 1188.315 ms
Software VADD run                  :  157.226 ms
Memory object migration enqueue    :    1.429 ms
Wait for kernels to complete       :  618.231 ms
-- Example 5: Pipelining Kernel Execution --
```

表 4.7 に、結果を前回の実行と比較します。

表 4.7: 1 GiB バッファーのタイミング サマリ - 順次処理 vs. OpenMP

操作	例 5	例 6	$\Delta_{5\rightarrow6}$
ソフトウェア VADD	1166.471 ms	157.226 ms	-1009.245 ms
ハードウェア VADD (合計)	692.172 ms	618.231 ms	-73.94 ms
$\Delta_{Alveo\rightarrow CPU}$	-503.402 ms	461.005 ms	964.407 ms

CJK

アクセラレータの実行時間が変動しているのは、これらのテストが仮想クラウド環境で実行されていることが主な理由ですが、それはこの演習の焦点ではありません。

追加演習

この演習に、追加で次のことを試してみてください。

- ベクター加算でプロセッサのスピードを上回ることができるかどうか試してみます。

- OpenMP のさまざまなプラグマを試してみます。ハードウェアアクセラレータを上回るには、CPU コアがいくつ必要ですか。

学習ポイント

- 繰り返しになりますが、単純な *O*(*N*) では CPU を上回ることはできません。

それでも、落胆する必要はありません。それでは、実際の例を見てみましょう。

例 7: OpenCV を使用したイメージサイズの変更

概要

画像処理は、FPGA でのアクセラレーションを活用するのに適した分野です。それにはいくつか理由があります。まず、画像をピクセルレベルで処理する場合、画像のサイズが大きいほど、計算量も増加します。この資料の冒頭で登場したモノレールの比喩がこれによく当てはまります。

ここで、バイラテラルサイズ変更アルゴリズムの簡単な例を見てみましょう。このアルゴリズムは、入力画像の解像度を変更します。この操作手順は次のようになります。

1. メモリから画像のピクセルデータを読み出します。
2. 必要に応じて、ピクセルデータを適切なフォーマットに変換します。ここでは、OpenCV ライブラリで使用されるデフォルトフォーマットである BGR に変換します。さまざまなストリームやカメラなどからデータを受信する実際のシステムでは、ソフトウェアまたはアクセラレータ(次の例で説明するように、ここでは基本的に「コストのかからない」操作) でフォーマットを変換する必要があります。
3. カラー画像の場合は、各チャネルを抽出します。
4. 個々のチャネルに対してバイラテラルサイズ変更アルゴリズムを使用します。
5. チャネルを再結合し、メモリに戻します。

これを図で表すと、図 4.4 のようになります。

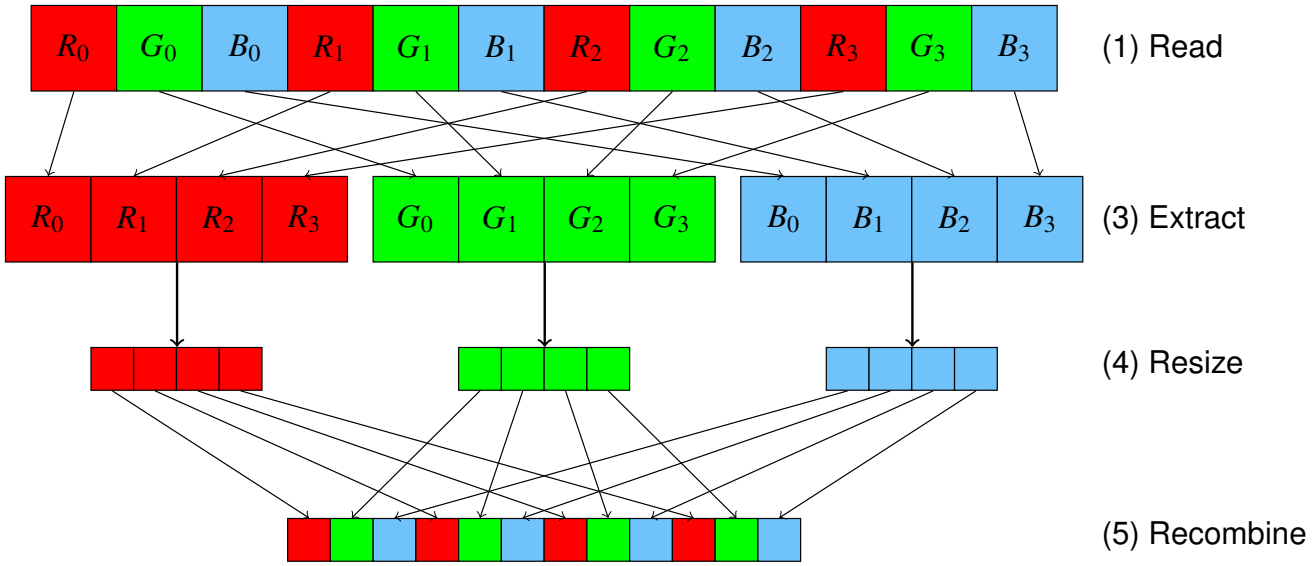

図 4.4: サイズ変更アルゴリズムのデータフロー

この図では、オプションの色変換手順は示していません。
これらの操作は、1 つずつ順に実行されます。前にも説明しましたが、メモリに何度もアクセスするのはコストがかかります。メモリからデータをストリーミングしながら、FPGA でこれらの計算すべてを実行できれば、通常は各操作および計算の追加レイテンシが非常に短くなります。これを利用して、インプリメントする特定の関数用にハードウェアを最適に使用するカスタムドメイン特化アーキテクチャを構築できます。
そして、クロックごとにより多くのピクセルを処理するためにデータパス幅を広くするか (アムダールの法則)、または複数のパイプラインを使用して多くの画像を並行処理するか (グスタフソンの法則) を選択できます。理想的には、この両方を使用して最適化していきます。アルゴリズムをできるだけ効率よく処理できるカーネルを構築し、FPGA ファブリックを最大限に活用してできるだけ多くのカーネルを配置します。

キーコード

このアルゴリズムには、ザイリンクス **xf::OpenCV** ライブラリを利用します。これはハードウェアに最適化されたライブラリで、アプリケーションで直接使用可能なよく使用される OpenCL 関数の多くをインプリメントします。このライブラリは、必要に応じてソフトウェアの OpenCV 関数やほかのライブラリ呼び出しと組み合わせて使用することもできます。

また、ほかのカーネル用に、画像の事前および事後処理用にこれらのライブラリを使用することも可能です。たとえば、カメラやネットワークストリームから生のデータを入力して、それを事前処理し、その結果をニューラルネットワークに入力して処理するなどです。これらの操作は、ホストメモリに戻らずにすべて FPGA で実行でき、すべて並列処理できます。発生する可能性のある唯一の競合は、レジスタ空間の競合ではなく、帯域幅の競合である、パイプライン処理された機能のストリームを構築すると考えてください。
xf::OpenCV ライブラリで、テンプレートを使用して、クロックごとに処理するピクセルの数などを設定します。ここでは詳しく説明しませんが、詳細は『ザイリンクス OpenCV ユーザーガイド』(UG1233: 英語版、日本語版) または次のチュートリアルを参照してください。

SDSoC 開発環境チュートリアル: OpenCV からxfOpenCV への移行

このチュートリアルで、ザイリンクスのエンベデッドアクセラレーションツールである SDSoC を使用していますが、その概念および手法は、Alveo ボード用のカーネルを記述するために OpenCV ライブラリを使用する場合にも適用できます。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./07_opencv_resize alveo_examples <path_to_image>
```

この例でのハードウェアのコンフィギュレーション方法のため、画像が特定の要件を満たす必要があります。1 クロックごとに 8 ピクセル処理するので、入力幅は 8 の倍数である必要があります。また、512 ビット幅のバスでデータをバースト転送するので、入力画像は次の要件を満たす必要があります。

$$in\_width \times out\_width \times 24 \,\%\, 512 = 0$$

要件を満たしていない場合は、どの条件が満たされていないかを示すエラーメッセージが表示されます。これはライブラリの基本的な要件ではありません。どんな解像度の画像でも、1 クロックごとに何ピクセルでも処理できます。ただし、入力画像が特定の要件を満たしていれば、さらに高速処理が可能となり、最適なパフォーマンスが得られます。
ハードウェアおよびソフトウェア OpenCV 両方からのサイズ変更後の画像に加え、プログラムにより次のようなメッセージが表示されます。

```
-- Example 7: OpenCV Image Resize --

OpenCV conversion done! Image resized 1920x1080 to 640x360
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :    5.145 ms
OpenCL initialization            :  292.673 ms
OCL input buffer initialization  :    4.629 ms
OCL output buffer initialization :    0.171 ms
FPGA Kernel resize operation     :    4.951 ms
```

この例では根本的に違う操作を実行しているので、前の実行と結果を比較することはしませんが、画像を処理するのにかかる時間を Alveo カードとCPU で比較することは可能です。

Alveo カードと CPU の結果はほぼ同じです。使用しているバイリニア補間アルゴリズムは $O(N)$ ですが、計算は前ほど単純ではないので、前ほど I/O で制限されません。CPU を上回ることはできますが、ほんの少しだけです。

ここで興味深いのは、多くの計算を入力の解像度ではなく出力の解像度に基づいて実行しているということです。同じ量のデータを転送する必要がありますが、画像を入力の 1/3 のサイズに変換するのではなく、解像度を倍にしたらどうなるかを見てみましょう。例のコードを 4.20 に示すように変更し、再コンパイルします。

コード 4.20: 倍の解像度

```
uint32_t out_width  = image.cols * 2;
uint32_t out_height = image.rows * 2;
```

例を再実行すると、おもしろい結果が得られます。

```
-- Example 7: OpenCV Image Resize --

OpenCV conversion done! Image resized 1920x1080 to 3840x2160
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :   11.692 ms
OpenCL initialization            :  256.933 ms
OCL input buffer initialization  :    3.536 ms
OCL output buffer initialization :    7.911 ms
FPGA Kernel resize operation     :    6.844 ms
```

ここではひとまず、バッファーの初期化は除外して考えます。適切に設計されたシステムでは、メモリ割り当てはアプリケーションのクリティカルパスでは実行されません。データ転送も含めたサイズ変更処理を見ると、CPU よりほぼ 60% 高速です。

CPU と Alveo カードは両方ともこの画像を比較的高速に処理できますが、システム全体のスループット目標が 1 秒ごとに 60 フレームを処理することである場合を考えてみてください。その場合、各フレームを処理するのに 16.66 ms しかありません。たとえば、サイズ変更したフレームをニューラルネットワークに入力する場合、ほぼすべての時間を使い切ってしまいます。

パイプラインを実行するためバッファーのチェインをあらかじめ割り当てれば、アクセラレータを使用することによりフレームごとの時間をほぼ 30% 回復できます。

入力画像のサイズが 1920x1200 の場合、結果は表 4.8 に示すようになります。

表 4.8: 画像のサイズ変更のタイミング サマリ - Alveo vs. OpenCV

操作	拡大	縮小
ソフトウェアでのサイズ変更	5.145 ms	11.692 ms
ハードウェアでのサイズ変更	4.951 ms	6.684 ms
$\Delta_{Alveo \rightarrow CPU}$	-194 µs	-5.008 ms

さらに、色変換は含めていなかったことを思い出してください。FPGA ではこれはコストのかからな

い操作です。追加で数クロックサイクルのレイテンシがありますが、基本的にはピクセル値に対していくつかの乗算および加算を実行する単純な *O*(*N*) 演算です。

追加演習

この演習に、追加で次のことを試してみてください。

- ホストコードを編集して画像サイズを変えてみます。画像サイズを大きくすると、実行時間はどうなりますか。画像サイズを小さくするとどうなりますか。アクセラレータを使用する意味がなくなる地点はどこですか。
- FPGA アクセラレータに色変換を追加します(ハードウェアを再ビルドする必要あり)。処理時間が長くなるかどうかを確認します。

学習ポイント

- 計算がより複雑な*O*(*N*) 演算はアクセラレーションの良い候補となりますが、CPU と比較して大きな利点は見られません。
- xf::OpenCV などの FPGA に最適化されたライブラリを使用すると、一般的なアルゴリズムを再インプリメントする必要なく、処理速度とリソースのバランスを取ることができますき、アプリケーションのより重要な部分に焦点を置くことができます。
- 選択するライブラリの最適化によって、デザインが制限されることがあります。ハードウェアをインプリメントする前に、使用するライブラリ関数の資料を参照してください。

先ほど、FPGA ファブリックでの追加処理は非常に短時間で実行できると述べました。次のセクションで、これが本当であるかを見てみましょう。

例 8: OpenCV を使用した演算のパイプライン処理

概要

前の例では単純なバイラテラルサイズ変更アルゴリズムを使用しました。これはアクセラレーションで大きな利益が得られる候補ではありませんでしたが、異なる機械学習用にバッファーを同時に多数の解像度に変換する場合もあるでしょう。または、フレームウィンドウ中に CPU はほかの処理用に確保しておくため処理をオフロードする場合もあります。
それでは、FPGA ストリーミングの本当の実力を見てみましょう。メモリとの間でデータを転送するのには時間がかかるので、その代わりに画像の各ピクセルを1 つの操作から次の操作にストリーミングし、メモリに戻さずに別の画像処理パイプライン段に送信するようにします。

この例では、先ほどのイベントシーケンスを変更してガウシアンフィルターを追加します。これは、エッジ検出、コーナー検出などの操作の前に画像のノイズを除去するのによく使用されるパイプライン段です。その後に、2D フィルターやその他のアルゴリズムを追加することも可能です。

前のワークフローを次のように変更します。

1. メモリから画像のピクセルを読み込みます。

2. 必要に応じて、ピクセルデータを適切なフォーマットに変換します。ここでは、OpenCV ライブラリで使用されるデフォルトフォーマットであるBGR に変換します。さまざまなストリームやカメラなどからデータを受信する実際のシステムでは、ソフトウェアまたはアクセラレータ (次の例で説明するように、ここでは基本的に「コストのかからない」操作) でフォーマットを変換する必要があります。
3. カラー画像の場合は、各チャネルを抽出します。
4. 個々のチャネルに対してバイラテラルサイズ変更アルゴリズムを使用します。
5. 各チャネルにガウシアンぼかしを実行します。
6. チャネルを再結合し、メモリに戻します。

この例には、バイラテラルサイズ変更とガウシアンぼかしの2 つの大きなアルゴリズムがあります。$w_{out} \times h_{out}$ のサイズ変更された画像と、幅 k の正方ガウシアンウィンドウに対しては、パイプライン全体の計算時間はおよそ次のようになります。

$$O(w_{out} \cdot h_{out}) + O(w_{out} \cdot h_{out} \cdot k^2)$$

試しに、k を比較的大きくしてみましょう。7 × 7 ウィンドウを選択します。

キーコード

このアルゴリズムでは、引き続きザイリンクス **xf::OpenCV** ライブラリを使用します。
先ほどと同様、ライブラリ(ハードウェアソースファイルのテンプレートを使用してハードウェアに配置) をクロックサイクルごとに 8 ピクセルを処理するよう設定します。このハードウェアアルゴリズムは、機能的には標準 OpenCV のコード 4.21 と同等です。

コード 4.21: 例 8: バイラテラル サイズ変更およびガウシアンぼかし

```
cv::resize(image, resize_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_LINEAR);
cv::GaussianBlur(resize_ocv, result_ocv, cv::Size(7, 7), 3.0f, 3.0f, cv::BORDER_CONSTANT);
```

例 7 と同様にサイズ変更を実行し、ガウシアンぼかし関数に 7 × 7 ウィンドウを適用します。また、ランダムに $\sigma_x = \sigma_y = 3.0$ を選択しました。

アプリケーションの実行

XRT ランタイムが初期化されたら、ビルドディレクトリから次のコマンドを実行してアプリケーションを実行します。

```
./08_opencv_resize alveo_examples <path_to_image>
```

前と同様、この例でのハードウェアのコンフィギュレーション方法のため、画像が特定の要件を満たす必要があります。1 クロックごとに 8 ピクセル処理するので、入力幅は 8 の倍数である必要があり、また 512 ビット幅のバスでデータをバースト転送するので、入力画像は次の要件を満たす必要があります。

$$in\_width \times out\_width \times 24 \,\%\, 512 = 0$$

要件を満たしていない場合は、どの条件が満たされていないかを示すエラーメッセージが表示されます。先ほども述べましたが、これはライブラリの基本的な要件ではありません。どんな解像度の画像でも、1 クロックごとに何ピクセルでも処理できます。ただし、入力画像が特定の要件を満たしていれば、さらに高速処理が可能となり、最適なパフォーマンスが得られます。
ハードウェアおよびソフトウェア OpenCV 両方からのサイズ変更後の画像に加え、プログラムにより次のようなメッセージが表示されます。

```
-- Example 8: OpenCV Image Resize and Blur --

OpenCV conversion done! Image resized 1920x1080 to 640x360 and blurred 7x7!
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :    7.170 ms
OpenCL initialization            :  275.349 ms
OCL input buffer initialization  :    4.347 ms
OCL output buffer initialization :    0.131 ms
FPGA Kernel resize operation     :    4.788 ms
```

前の例では、CPU と FPGA のパフォーマンスはほぼ同じでした。CPU 関数では処理時間が大幅に長くなっていますが、FPGA の実行時間はほとんど増加していません。

ここで入力のサイズを倍にし、1080p 画像を4k 画像に変更してみましょう。例 7 でコード 4.20 と同様の変更をこの例にも加え、再コンパイルします。

例を再び実行すると、非常に興味深い結果が得られます。

```
-- Example 8: OpenCV Image Resize and Blur --

OOpenCV conversion done! Image resized 1920x1080 to 3840x2160 and blurred 7x7!
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :  102.977 ms
OpenCL initialization            :  250.000 ms
OCL input buffer initialization  :    3.473 ms
OCL output buffer initialization :    7.827 ms
FPGA Kernel resize operation     :    7.069 ms
```

驚くべき結果です。CPU の実行時間はほぼ 10 倍になりましたが、FPGA の実行時間はほとんど変わっていません。
FPGA は、処理を並列実行するのに非常に優れています。このアルゴリズムは、I/O で制限されるので

はなく、プロセッサで制限されます。さらに分解してクロックごとに複数のピクセルを計算することによりデータをより高速に処理し(アムダールの法則)、1 つの演算から次の演算にストリーミングすることにより並列実行を増やす(グスタフソンの法則) ことができます。ガウシアンぼかしをさらに個々の要素の計算に分解し、それらを並列に実行することもできます (xf::OpenCV ライブラリで既に実行済み)。

帯域幅ではなく計算が制限になったので、アクセラレーションの利点がはっきりわかります。これを FPS で考えると、x86 クラスの CPU インスタンスでは毎秒 9 フレーム処理できますが、FPGA カードでは 141 フレーム処理できます。さらに演算を追加すると CPU での処理時間がどんどん長くなりますが、FPGA ではリソースさえあればいくらでも演算を追加できます。実際、Alveo U200 カードで使用可能なリソースと比較すると、この例のカーネルはまだ非常に小型です。
これを前の例と比較すると、1920x1200 の入力画像では結果は表 4.9 に示すようになります。差を示す列は、例 7 の拡大とこの例の拡大を比較したものです。

表 4.9: イメージのサイズ変更とガウシアンぼかしのタイミング サマリ - Alveo vs. OpenCV

操作	縮小	拡大	$\Delta_{7\rightarrow8}$
ソフトウェアでのサイズ変更	7.170 ms	102.977 ms	91.285 ms
ハードウェアでのサイズ変更	4.788 ms	7.069 ms	385 µs
$\Delta_{Alveo\rightarrow CPU}$	-2.382 ms	-95.908 ms	-90.9 ms

アクセラレーションの利点は明らかです。

追加演習

この演習に、追加で次のことを試してみてください。

- ホストコードを編集して画像サイズを変えてみます。画像サイズを大きくすると、実行時間はどうなりますか。画像サイズを小さくするとどうなりますか。アクセラレータを使用する意味がなくなる地点はどこですか。
- 追加のハードウェアレイテンシはありますか。

学習ポイント

- ファブリックで1 つの演算から次の演算にパイプライン処理してストリーミングすると、有益です。
- xf::OpenCV などの FPGA に最適化されたライブラリを使用すると、一般的なアルゴリズムを再インプリメントする必要なく、処理速度とリソースのバランスを取ることができますき、アプリケーションのより重要な部分に焦点を置くことができます。
- 選択するライブラリの最適化によって、デザインが制限されることがあります。ハードウェアをインプリメントする前に、使用するライブラリ関数の資料を参照してください。

まとめ

これらのチュートリアルが、Alveo カードの利用方法を学ぶのに役立ったことを願います。もちろん、これらのチュートリアルでこのトピックのすべてを伝えることは不可能です。実際、表面をほんの少しなぞっただけです。Alveo 用にカーネルをビルドする方法の詳細は、次のサイトからハードウェアカーネルに焦点を置いたチュートリアルを参照することをお勧めします。

- SDAccel 開発環境チュートリアル
- 『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207: 英語版、日本語版)

ライブラリは、次の資料を参照してください。

- 『ザイリンクス OpenCV ユーザーガイド』(UG1233: 英語版、日本語版)
- SDSoC 開発環境チュートリアル: OpenCV から xfOpenCV への移行

ザイリンクスフォーラムもご活用ください。フォーラムには、高度な知識を持つザイリンクスチームおよび世界中の SDAccel 上級者が積極的に参加し、活発に意見交換しています。

Alveo カードに関するハードウェアの質問:
Alveo データセンターアクセラレータカードフォーラム

開発ツールチェーンに関する質問:
SDAccel フォーラム

機械学習に関するフォーラムもあります。次のフォーラムにログインし、深層学習プロセッシングユニットおよび関連のツールチェーン、ディープニューラルネットワーク開発キットに関する議論にご参加ください。

DPU および DNNDK フォーラム

これらのチュートリアルをご利用いただき、ありがとうございました。これらが役に立ったことを願うと共に、今後ユーザーのみなさんと協力できることを期待します。

付録 A

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、ザイリンクス サポート サイトを参照してください。

ソリューション センター

デバイス、ツール、IP のサポートについては、ザイリンクス ソリューション センターを参照してください。デザイン アシスタント、デザイン アドバイザリ、トラブルシューティングのヒントなどが含まれます。

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとします。また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、 https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。https://japan.xilinx.com/legal.htm#tos で見られるザイリンクスの販売条件を参照してください。

XILINX

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。