オープンソース FPGA CNN ライブラリ

まえがき：

畳み込みニューラルネットワーク（CNN）は、画像および視覚認識タスクで使用される機械学習アルゴリズムの中心的な構成要素となっています。このネットワークの低ビット幅 FPGA 実装は、高スループットおよび低消費電力の機械学習推論ソリューションへの潜在的な道筋を提供します。

このツールボックスの主な目的は、ネットワークを構成する各層で浮動小数点定義ネットワーク重みから低ビット幅固定小数点重みへ切り替えることの精度への影響を調べるための開発者向けの簡単なパスを提供することです。このツールボックスの 2 つ目の目的は、さまざまなネットワーク層の構造的な影響を調べることができるため、教育目的です。これは、ニューロンを実装するのに必要な単純な乗算累積演算でさえも要件が変わる算術上の固定幅ビット幅の意味と、重み、入力および中間データをオン、オフおよびオフにする構造的およびチップ周辺の並列化の要件の両方に当てはまります。3 つ目の目的は、ユーザーがこれらの構造を探索して自分の要件を満たすようにコードを変更できるように、オープンソースのリファレンスコードを提供することです。このコードは、FPGA ハードウェアや FPGA ツールを事前に購入しなくても、CPU ベースのシステムでデザインを探索できるように、オープンソースコンパイラと連携するように設計されています。最後の目的は、x86 CPU PCIe プラグインカード構成と IBM Power 8 CAPI プラグインカード構成の両方で動作する、ザイリンクス FPGA ベースの Alpha Data 社のハードウェア上でこれらのアルゴリズムを実装するためのリファレンスデザインを提供することです。

開発者がハードウェアやソフトウェアを事前に購入することなく FPGA CNN の実装を検討できるようにすることで、このパッケージは、機械学習推論に FPGA を使用することの利点を理解し、機械学習に FPGA を使用する自信を与えます。FPGA ネットワーク回路自体を設計するのではなく、ザイリンクスまたは IBM から現在入手可能な市販の機械学習用 FPGA ライブラリを選択する場合。ツールボックスのコードはから以下のリンクからダウンロードできます。

ftp://ftp.alpha-data.com/pub/appnotes/cnn/adcnnlib-v1_0_0.zip

ftp://ftp.alpha-data.com/pub/appnotes/cnn/ad-an-0055_v1_0.pdf

ハードウェア記述言語を用いた構造的ニューラルネットワーク記述：

ツールボックスのサンプルコードは、ADA から派生したハードウェア記述言語 VHDL で提供されています。

なぜ *VHDL* なのか、なぜ *OpenCL* ではないのか？

株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

機械学習推論で HDL を使用することは従来のソフトウェア記述よりも多くの利点があります。まず、ニューラルネットワークの最も基本的な記述は並列ハードウェア記述であるため、行列やテンソル積表現などの低レベルのシリアル実行の実装の詳細を見なくても、これらのネットワークを並列ニューロンとしてモデル化することは非常に簡単です。また、このツールボックスの主な目的は教育的なものであり、基礎となる FPGA ハードウェアの構造を明確にすることは、タイミングや構造の詳細を抽象化する言語を後に使用したとしても、開発者がニューラルネットワークを FPGA 実装回路に適合させることに関わるタスクを理解するのに役立ちます。FPGA、GPU、CPU、その他の並列ハードウェアアクセラレータ間のコードの移植性を可能にすることを主な目的とする言語を使用すると、ユーザーがそれらを調査して活用を可能にするだけでなく、プラットフォーム間の違いが取り除かれます。クロックタイミングと FPGA リソースの完全並列動作を理解しておくと、ソリューションがハードウェアに対して完全に最適化されているかどうかをユーザーが理解するのに役立ちます。ユーザーがより高レベルのフレームワークを使用して開発可能なアプリケーションを構築することを選択した場合でも、あるいは事前に最適化されたライブラリソリューションでも、チップへのデータ移動、キャッシング要件、比較的単純な計算アルゴリズムに対するこれの影響を詳しく調べることができます。
次のコードセクションは、Figure 1 に示す構造の VHDL のニューロンについて記述しています。実データ型は、入力、出力、重みに使用されます。このコードは、入力と重みの並列乗算を示しています。シグナル（製品など）とエンティティポートはそれぞれの値を保持しており、並列プロセス間の通信に使用されます。 変数（例：sum）はプロセス内の範囲のみを有します。ただし、コードはすべての商品の合計と同じタイムステップ内の偏りを指定します。並列加算を説明する別の方法も可能です。ここで信号 timestep で指定されたタイムステップは通常、クロック信号を使用して指定され実装されています。

株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

```
...
    type real_array is array(natural range <>) of real;
...
    entity neuron is
     generic (
       number_of_synapses : integer := 32);
     port (
       next_timestep : in boolean;
       synapses : in real_array(1 to number_of_synapses);
       output : out real);
    end entity;
     architecture behave of neuron is
       constant bias : real := BIAS_VALUE;
       constant weights : real_array(1 to number_of_synapses) := (1=>WEIGHT1, 2=>WEIGHT2 ...
       signal products : real_array(1 to number_of_synapses);

      function ReLU(x : real) return real is
       begin
         if x<0 then
           return 0;
         else
           return x;
         end if;
       end function;
     begin

       -- Parallel Instantiation of multiplier
       gen mults: for i to 1 to number_of_synapses generate
         products(i) <= weights(i) * synapses(i);
       end generate;

       p neuron: process(next_timestep)
         variable sum : real;
       begin
         -- Note, everything computed in same timestep
         if (next_timestep) then

             sum := -bias;
             for i in 1 to number_of_synapses loop
               sum :=  sum + products(i);
             end loop;
             output <= ReLU(sum);
           end if;
         end if;
      end architecture;
```


株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

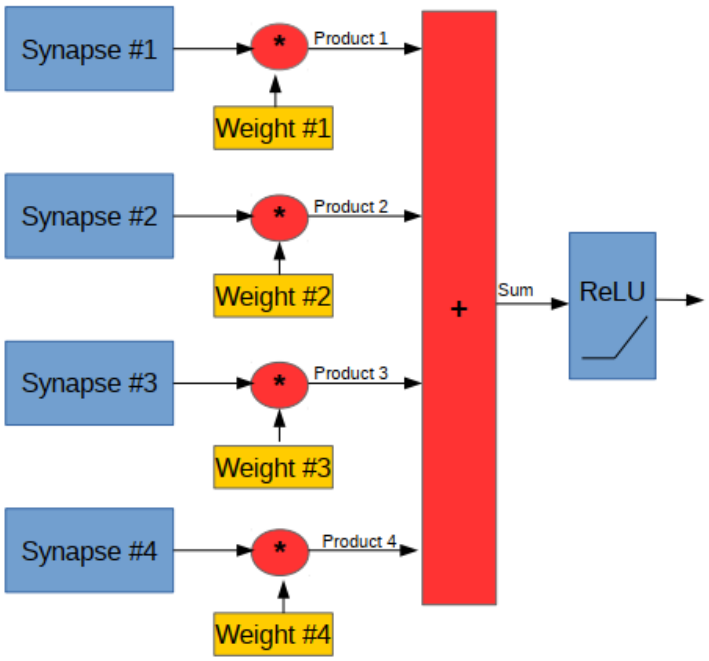

Figure 1 : Neuron Structure

VHDL（およびその他の HDL）は高度にモジュール化されており階層的です。ニューラルネットワークを実装するために、それらの並列ニューロンの多くを別のモジュールで並列にインスタンス化することができます。これにより、コードは Figure 2 に示す構造のように記述することができます。

```
-- Generate Neural Network layer
gen neurons: for i to 1 to number_of_neurons generate
  neu_i: neuron
    generic map(
       number_of_synapses => number_of_synapses)
    port map(
       next_timestep => next_timestep,
       synapses => layer_inputs,
       output => layer_outputs(i));
end generate;
```


株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

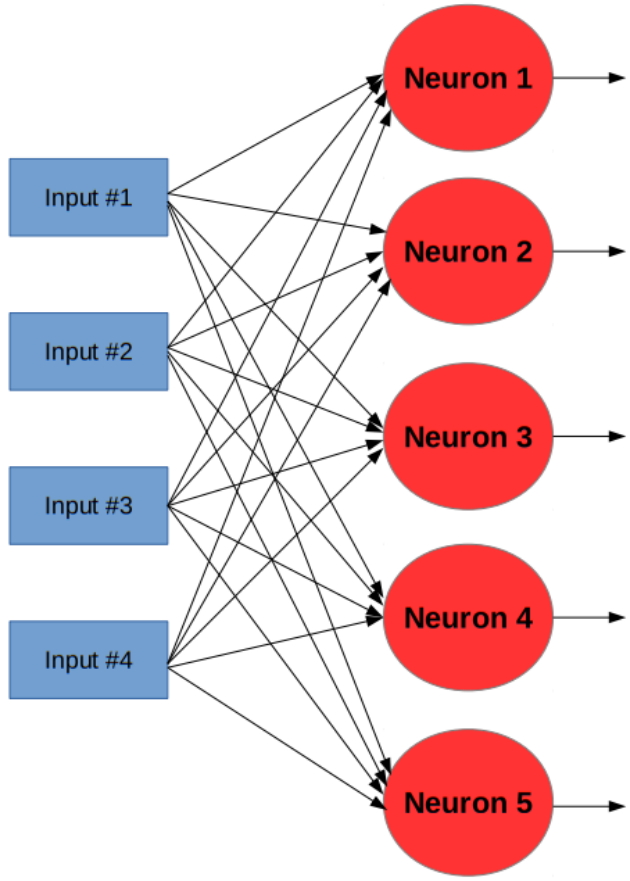

Figure 2 : Neural Network Layer Structure

前述の例は、VHDL 言語のいくつかの基本的な特性とその記述がニューラルネットワークで使用される構造をモデル化する容易さを説明するのに役立ちます。これまでに示したこれらのモデルは、FPGA ハードウェアを記述するのに十分ではなく、FPGA 上で実行するために合成することはできません。ただし、シミュレーションやモデリングの目的で、CPU 上で実行するようにコンパイルすることはできます。このモデルの大きな利点の 1 つは、実際の（浮動小数点）指定重みを持つ事前学習済みの畳み込みニューラルネットワークを、同じ精度を達成できる最小のビット幅を持つ低エネルギー、電力効率の固定小数点ソリューションに減らすことです。VHDL は、あらゆるビット幅で固定小数点符号付きおよび符号なし演算をネイティブにサポートしているため、ニューロン演算のさまざまな部分のビット幅を最小限に抑えることができるため、全体的なリソースの最適化が可能になります。

たとえば、ニューロンが 8 ビットの符号付き重みと 8 ビットの符号なしデータを使用する場合、符号付き乗算はデータに符号ビットを追加するために 8×9 ビットである必要があります。この積は 17 ビットになります。入力数に応じて、アキュムレータの合計は、潜在的なオーバーフローを回避するために追加のビット log2（入力数）を必要とします。出力はフル解像度である必要はなくビット数は少なくてもかまいませんが、リファレンス浮動小数点の動作に近い動作を実現するには、ある程度のスケーリングと飽和が必要になります。

株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

GHDL：オープンソース VHDL コンパイラ：

HDL コードのコンパイルは 2 つの目的で行うことができます。まず、ハードウェア ASIC または FPGA デザインを生成するために合成をします。次に、ハードウェアの動作をシミュレートするために CPU ベースのシステムで実行するためのハードウェアモデルとしてコンパイルします。最初のケースでは言語のどの部分を使用できるかについていくつかの制限があります。後者の場合は、OS からファイルにアクセスしたり、コマンドラインに情報を出力したりするなど、FPGA ハードウェアには同等の機能を持たない追加の機能やライブラリを使用できます。

実行可能な回路を作成するための HDL（または他のデザイン入力方法）からの FPGA 合成は、基本的に回路の合成、配置・配線は巡回セールスマン問題（トラベリングセールスマンタイプ問題）であるため、簡単な解決策はありません。そのため、シミュレーションによって動作が検証されるまで、合成フローは開発フローで回避する必要があります。

デザインのシミュレーションでは、VHDL コードをソフトウェア言語として効果的に扱い、それをコンパイルして CPU 上で実行します。FPGA 設計環境に含まれているものを含む、市販のシミュレーションパッケージが多数あります。これらは VHDL コードを解釈またはコンパイルすることができます。通常、これらのツールはデザイン内のすべての信号を表示用に記録することを優先しているため、効率的に実行されません。オープンソースの代替シミュレーション GHDL もあります。これは GCC 上に構築された VHDL コンパイラであり（最初に VHDL から C へのクロスコンパイルを行います）、コードをインタープリタではなくコンパイルするため、一部の商用シミュレーションパッケージよりも高速に実行できます。

この Alpha Data ライブラリのすべてのテストベンチベースのデザインは、開発フローのようなソフトウェアを提供するので、このパッケージを使用してテストされています。コンパイル、リンク、実行、動作分析、バグ修正、繰り返し...

このパッケージは http://ghdl.free.fr からダウンロードできます。

これにより、開発者はハードウェアやソフトウェアを購入することなく、FPGA ベースの機械学習推論の調査を開始できます。FPGA が特定の推論アプリケーションに適したハードウェアの選択であると確信できたら、このコードを再利用する、他の言語のコードを利用する、IBM、ザイリンクスなどから事前に最適化された機械学習スタックを利用するなど、さまざまな方法で実装できます。

ビット幅調査のための非タイミング VHDL モデル：

このセクションでは、CNN レイヤの非タイミング VHDL モデルについて説明します。単一プロセスとクロックなしのスティミュラスを使用することで、VHDL コードは ADA シーケンシャルプロセスのように複雑になりません。これはソフトウェア開発者にとって容易に理解できるはずです。

このコードは、圧縮ファイルの sim_only/sim_zfnet_layer0.vhd ファイルにあります。シミュレーションは ZFNet CNN [1]の入力層をモデル化します。テンソル型は、ネットワーク層で使用されるさまざまな固定小数点精度データ型に対して定義されます。

異なる算術演算に対する異なるデータビット幅を理解することは、計算あたりのエネルギー使用量を最小限に抑え、スループットを最大にするために不可欠です。浮動小数点の記述から来る入力データ、重み、バイアス、積、アキュムレータ値、そして整流後の線形出力はすべて同じ 32 ビット型で表されます。入力は通常、符号なし 8 ビット画像デー

タまたは 0 から 1 の範囲の符号なし値であるため、これは比較的非効率的です。重みは符号付きの値ですが、範囲と精度も限られています。これらの積はより高い分解能と算術演算のオーバーフローを避けるためにこれらの追加の範囲の累積を必要とします。出力の前に、整流された線形演算は範囲を縮小し、その段階で必要な分解能を下げ、また累積値の符号を取り除き、符号なしの値を次の層に出力します。Figure 3 は、さまざまなビット幅を示しています。

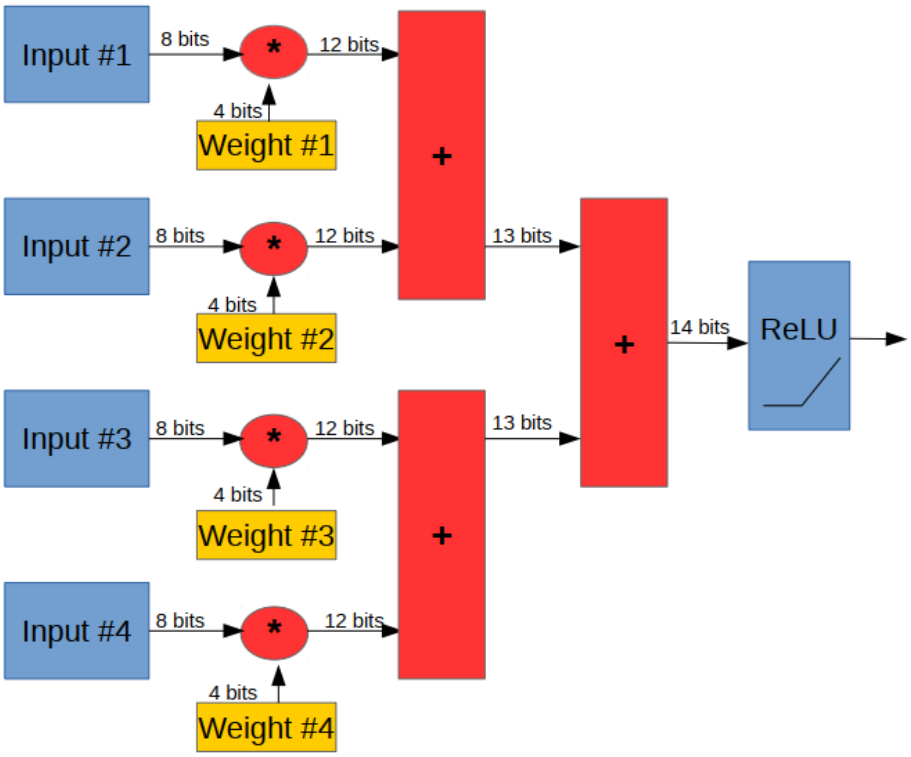

Figure 3 : Fixed Point Neuron Implementation

したがって、ニューロン計算全体で単一の算術型を使用することは、エネルギーを無駄に消費することになります。固定小数点演算を使用して最適な解を生成するには、演算内のこれらの各変数に必要な最小ビット幅を特定する必要があります。たとえば、32 ビットの浮動小数点ではなく 8 ビットの入力画像データを使用すると、メモリの帯域幅要件、オンチップデータによるメモリキャッシュの再利用、および入力データのルーティング要件が 4 分の 1 になります。8 ビット（または 32 ビット）ではなく 4 ビットの重みを使用すると、ウェイトメモリキャッシュの要件が緩和され、乗算およびアキュムレータのロジックフットプリントが減少し（または DSP タイルで固定サイズの乗算を共有できるようになります）、より多くのニューロン計算を FPGA に並列に実装できます。この例では、バイアスビット幅をウェイトと同じ幅に設定していますが、いくらかのシフトでダイナミックレンジを広げることができます。オプションで、バイアスはアキュムレータと同じビット幅の結果を有することができます。出力ビット分解能は次の層の要件、および整流された線形ユニットが線形領域の勾配を変えることができるようになる前の 2 倍の倍率（ビットシフト）に依存するので、生成された重要な大きな値が誤って飽和することはありません。

非タイミングモデルでは、さまざまなビット幅を変更した場合の影響を詳しく調べることができます。これらは、定数パラメータを書き換えることで簡単に変更できます。このコードは、入力データのファイルを読み込むように構成されており、単一のイメージをバッチで表します（複数のイメージを含むファイルを処理するようにコードを変更するのは

簡単です）。サンプルコードは現在、すべてのニューロンの重みをランダム化しています。予め学習された重みのネットワークが利用可能である場合、画像データと同じ方法でシミュレーションに読み込まれる可能性があります。
重みとデータがテンソル変数に読み込まれると、コードは各ピクセルにわたって順次ループしながら実行され、各フィルタの関心領域が抽出されます。これは領域マスクテンソルに読み込まれます。次に、領域マスクテンソル要素を各ニューロンの重みで畳み込み（積算）しその和を線形に修正します。
その後、ポストフィルタテンソルは、最大プーリングループを通過します。最大プーリングループは、小さい 2 次元のピクセル範囲で最大値を選択することによって出力データサイズを縮小します。この出力は分析のためにファイルに書き出されます。このコードは単層の例のみを提供しています。ネットワーク内の他の層をシミュレートし、それらの固定小数点性能の解析を可能にするために、同じコードを修正することができます。このコードは、FPGA 実装用の実用的なテンプレートを提供するものではありません。これは、FPGA では避けた方が良いいくつかの依存関係とデータアクセスの順序付けを強制します。ただし、比較的わかりやすい高水準形式で、ビット幅を固定した固定小数点の実装内で算術データ型の選択を検証する非常に有益なメカニズムを提供します。
GHDL を使用してこのコードをコンパイルして実行するのは非常に簡単です。単に-a でコンパイルし、-e でリンクしてからコードを実行します。

```
[user@machine sim_only]$ ghdl -a sim_zfnet_layer0.vhd
[user@machine sim_only]$ ghdl -e sim_zfnet_layer0
[user@machine sim_only]$ ./sim_zfnet_layer0
```

コードは、カラー画像として表示できる整数のテキストファイルである input_data.txt を読み込みます。データは、チャンネル（RGB）、列、行の順です。出力データもこのフォーマットで書き出され、チャンネル数は 96 ですがチャンネルは特定のカラーフィールドに関係しません。
コードは、ザイリンクス Vivado に組み込まれている XSim シミュレータ、Mentor Graphics Modelsim、または Cadence NCSIM などの標準的な EDA シミュレーションツールにインポートすることもできます。
Caffe Model Zoo http://caffe.berkeleyvision.org/model_zoo.html から、事前に学習された重みを使用した 2 番目の例です。このレイヤシミュレーションは、AlexNet[2]の入力レイヤに基づいています。重みは Caffe から整数形式でエクスポートされ、範囲を-1：1 を超えて拡大するために 2^30 でスケーリングされ、テキストファイルとして保存されています。シミュレーション VHDL はこれらをインポートし、シミュレーションで指定されたビット幅に収まるように縮小します。Figure 4 は入力画像、Figure 5 はレイヤー0 以降の 96 個のニューロン出力を示しています。

Figure 4 : Input Image

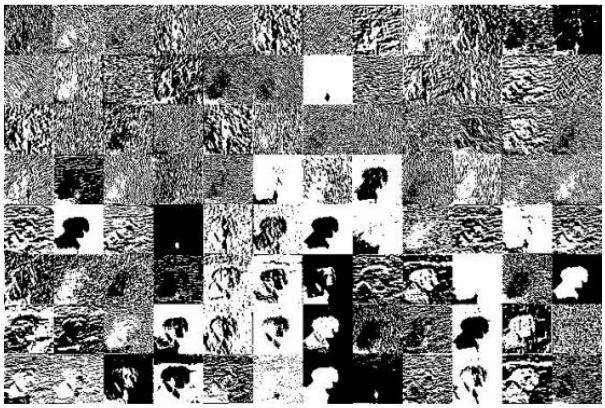

Figure 5 : Layer 0 Output

ハードウェア合成可能ニューロン：

前のセクションで完全並列のニューロン構造とシーケンシャル指定されたネットワークは、一般的に FPGA 実装には有用ではありません。完全なシーケンシャルソリューションは明らかに FPGA 論理の並列性を利用しませんが、完全に並列な構造も FPGA チップ内で利用可能な積和演算器の数によって規模が制限されます。5500 以上の DSP ブロックが利用可能であっても、完全に並列なネットワークは前述の ZFNet ネットワーク層の約 37 のニューロンを実装できるだけです。スループットは非常に高速ですが、ネットワーク全体を多数のチップにわたって実装する必要があります。

より実用的な並列化は、 1 積和演算器対 1 ニューロン比で達成することができます。スループットを向上させるためのニューロン内での並列化のように計算リソースが限られている場合は、多数のニューロン間で単一の積和演算器を共有することを検討することができ、これは並列化オプションの真ん中にあります。

エネルギー効率の良い実装を作成する際の重要な優先事項は、乗算ユニットへの入力データの移動を最小限に抑えることです。例えば、ザイリンクスの KU115 デバイスでは 5500 個の DSP タイルがあり、控えめに見ても 250MHz で 16 ビット幅を仮定すると、すべてのタイルを個別に動作させて両方の入力をフルに保つためのメモリ帯域幅は 2.75TB/s になります。そのため、データの再利用とキャッシュが不可欠です。同じ層のすべてのニューロンは同じ入力データを使用するので、並列のニューロンの層を持つことは計算のその半分のための入力帯域幅要件をニューロンの数に関連した係数で減らします。乗算ユニットへの入力帯域幅の残りの半分は重みになります。これらのキャッシングは、実用的な入力帯域幅を達成するために必要なデータの再利用を達成するために必要になります。畳み込みネットワークの入力端では、典型的に各重み値は、バッチ内のあらゆるフレームと同様に、あらゆるピクセル位置出力に対して何度も使用されます。例の ZFNet 入力レイヤでは、これはバッチ内のフレームあたり 12100（110×110）の再利用で行われます。したがって、このレイヤでは、ウェイト値をできる限り乗数に近づけることが望ましいです。例えば ZFNet の後の層、特に完全に接続された層の場合、この構造的な議論は重み値の再利用率が異なるために変わります。

株式会社ミッシュインターナショナル 〒190-0004 東京都立川市柏町 4-56-1
TEL: 042-538-7650 https://www.mish.co.jp E-mail: sales@mish.co.jp

FPGA には通常これらのブロックが多数あり、それぞれ独立して約 2kB のスペースでアドレス指定できるため、合成可能なサンプルコード onv_neuron.vhd では、ウェイトメモリはデュアルポートメモリとして実装されています。それらはまた、より小さな、そしてより大きな利用可能メモリを持っています。ただし、ウェイトに最も有用なメモリは、約 2kB の FPGA ブロックメモリです。通常 FPGA では、各乗算器ユニットにこれらの 2kB メモリブロックの 1 つがあります。

推論モデルでは、ウェイトは通常 1 回書き込まれ、次に処理するデータのバッチサイズが大きくなる可能性があるためストリーミングされ、ウェイト入力帯域幅は狭くなり、一度に 1 つのウェイトが書き込まれます。 最初と最後のウェイトストローブでストリーミングします。最初のウェイトは実際にはバイアスとして使用されます。不必要な減算または符号の変更を取り除くために、重みが初期化されるときにバイアスの負の値がメモリに書き込まれると仮定されます。ウェイトが完全に静的に事前定義されている場合、ウェイトメモリは理論的にはさらなるリソースを節約する FPGA 内の ROM として実装することができるが、柔軟性を犠牲にすることに留意してください。

この処理は特徴入力によってドライブされます。これは、最初と最後の機能が独立してシグナリングされる、連続したデータの流れとして到着すると想定されます。これにより、アキュムレータをバイアスにリセットし、メモリ読み出しアドレスを使用して最初の重みにアクセスすることで最初の要素を使用可能にします。最後の入力サンプルが最後によって示されるまで、乗算および累積は重みおよび特徴を通って流すことができ、累積値は線形に整流されて出力されます。特徴データは、間に無駄時間なしでこれらのニューロンに連続的に入力することができます。出力はこのレートを重みの数で割った値で生成されます。コード例の出力スケーリングは、コード例のレイヤ全体で同じ値に固定されています。静的に定義されているか、または実行時設定パラメータとして異なるニューロンで異なるスケーリングを可能にすることによって、ネットワークのダイナミックレンジを拡張することは有用でしょう。

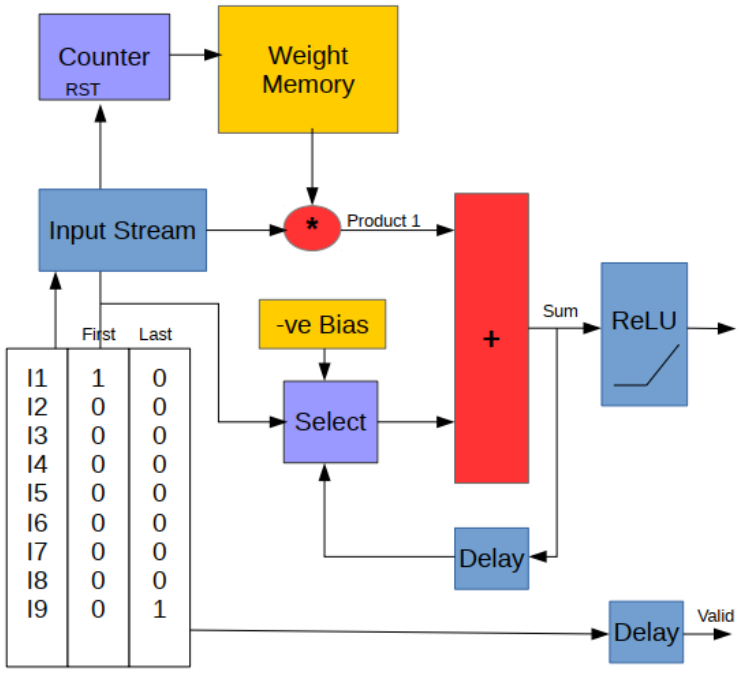

Figure 6 : Example Hardware Synthesizable Neuron Implementation

Figure 6 は、乗算累積ベースのハードウェア合成可能ニューロン実装の一般的構造を示しています。サンプルコードでは、スループットと最大クロックレートはレイテンシとして扱っています。ブロック RAM と DSP タイルの基本は、データがそれらの入力と出力に登録されている場合、最大のクロック性能を発揮します。したがって、最初のストローブおよび最後のストローブなどの制御信号は遅延され、データの蓄積および出力は、BRAM および DSP タイルプリミティブの内蔵レジスタ構造と最もよく一致するようにタイミングがとられます。

サンプルコードのテストは、conv_neuron.vhd および tb_conv_neuron.vhd というテストベンチファイルを使用して、conv_neuron フォルダから実行することができます。このテストベンチファイルは、ニューロンモデルを実行して結果をコマンドラインに出力します。

```
[user@machine sim_only]$ ghdl -a conv_neuron.vhd
[user@machine sim_only]$ ghdl -a tb_conv_neuron.vhd
[user@machine sim_only]$ ghdl -e tb_conv_neuron
[user@machine sim_only]$ ./tb_conv_neuron
```

理論上、conv_neuron ブロックは永遠に実行するように指定されているので、Ctrl+C を使用してシミュレーションを終了する必要があります。または、停止時間を指定することもできます。

```
[user@machine sim_only]$ ./tb_conv_neuron --stop-time=100us
```

ハードウェア合成可能なニューラルネットワーク層：

ネットワーク層は、多数の前述のニューロンを並列に指定することによって容易に構築することができます。すべてのニューロンはそれぞれ同じ入力特徴データストリームを受け取ります。非常に多くのニューロンにとって、これは潜在的にルーティング混雑問題を引き起こす可能性があります。ニューロン層の最初の例（conv_neuron_layer.vhd）は、シフトレジスタを介してデータを各ニューロンに広げることでこれを回避します。これにより、非常に高速にクロック供給できる FPGA デザインが生成され、配線上の問題はほとんど発生しません。ニューロンは、それぞれ 1 クロックサイクル遅れて順番に出力を生成します。その後、これらが出力のシフトレジスタを使用して 1 つのストリームにまとめられると、データは 2 クロックサイクルごとに有効なデータを含むバーストとして到着します。

この制限は、ニューロンの重みの数が、レイヤ内のニューロンの数の 2 倍より大きくなければならないということです。ZFnet の入力層は行いませんが、ほとんどの内部ネットワーク層はこれを行います。最大クロックレートが低くなる可能性がある 2 番目のオプションは、入力側でフラットファンアウトを使用することです（conv_neuron_layer_ffanout.vhd）。これにより、同じクロックサイクルですべてのニューロン出力が生成されます。これは、シフトレジスタを使用して連続したストリームとして出力できます。この層は重みと同数のニューロンを持つことができるので、ZFnet 入力層を実装することができます。この制限は出力帯域幅によるものであり、ウェイト数に対するウェイト数の比率がより高いニューロンを扱うもう 1 つの方法は、この構造を複数回並行して実装し、各構造がニューロンのサブセットを実装し、ストリームを並列に出力することです。

```
[user@machine sim_only]$ ghdl -a ../conv_neuron/conv_neuron.vhd
[user@machine sim_only]$ ghdl -a conv_neuron_layer.vhd
[user@machine sim_only]$ ghdl -a tb_conv_neuron_layer.vhd
```

```
[user@machine sim_only]$ ghdl -e tb_conv_neuron_layer
[user@machine sim_only]$ ./tb_conv_neuron_layer
```

ハードウェア合成可能な畳み込みニューラルネットワーク層：

前の層は完全に接続された層を実装しています。ただし、畳み込みレイヤを実装するには、データパイプラインにいくつかの追加ブロックを追加する必要があります。

畳み込みネットワークでは、入力データストリームは通常、特徴（チャネル）の 2D「画像」からなるバッチ内の各フレーム、すなわち 3D テンソルに対するものです。入力レイヤの場合、入力は多くの場合、赤、緑、青のカラーデータ用の 3 チャンネルの 2D 画像です。より深い層では、チャネルはより多くなることがあり、必ずしも容易に説明できるものに対応するとは限りません。

この画像データをメモリから効率的に読み取るには、前のセクションでシミュレートしたものとはかなり異なるアプローチが必要です。そのコードは、メモリアドレスへのバイト幅のアクセス、連続したもの、および次の行のものを指定します。内部データバス幅が 512 ビットで、72 ビット幅の ECC DDR4 DIMM でバイト幅の読み取りを使用する FPGA での単純な実装では、1.6%の効率が得られます。各ピクセルは約 10 回再利用されるため、組み合わせて使用すると 19.2GB/s のメモリインターフェイスが 30MB/s に低下し、実際にメモリ帯域幅の問題が生じる可能性があります。CPU や GPU では、内蔵のハードウェアキャッシュ階層がこれを大幅に軽減し、一度にキャッシュラインを読み取るので、メモリの帯域幅はそれほど重要ではありません。FPGA の選択肢は、組み込みキャッシュはなく CPU のような汎用キャッシュまたはアプリケーション固有のキャッシュを使用することです。CNN を実装する場合、データアクセスパターンは予測可能であるため、アプリケーション固有のキャッシュを実装するのは比較的簡単です。

バッチ内のフレームの入力データは通常連続したバッファに格納されるため、最大幅で DDR4 から（またはホストから PCIe を介して）このデータをバーストリードする必要があります。これにより、フィルタマスクサイズよりもわずかに大きいサイズのローカルオンチップキャッシュメモリにバッファできるよりも多くのピクセルのストリームが作成され、必要とされるテンソルマスク領域を入力画像上の 12100 個の位置を 1 つ毎に抽出することができます。

このキャッシュのコードは feature_buffer.vhd に盛り込まれています。キャッシュはフィルタ領域データが他のラインから必要な速度で読み取られている間に、フロー制御入力ストリームがデータをラインに書き込むことができるようにするために必要なライン数プラス 2 を保持するのに十分な深さです。

ネットワークやレイヤによっては、データを画像の端からはみ出してフィルタに読み込むことを指定するものがあります。外枠を追加するこのゼロパディングは、機能バッファの読み取りロジックに組み込むことができます。しかしながら、これはコードをかなり複雑にし、理解することをより困難にし、タイミングに関して最良の解決策ではないかもしれません。したがって、このコード例では、必要に応じてゼロパディングがバッファの前の入力ストリームに適用されます。これは余分なメモリリソースを必要とする際に少し無駄になります。処理速度を制御するのはバッファの読み取り側であるため、これはパフォーマンスには影響しません。

この例では、バッファから 3 画素ずつデータが出力されます。ただし、読み取られた各領域の終わりと次の領域の間にはギャップがあります。この出力にはストリーム絞り込みバッファが配置されています。これにより、データ幅は 1 クロックサイクルあたり 1 フィーチャに縮小されます。これは、クロックサイクルごとに演算を行いながら乗算器が完全

に利用されることを保証する連続ストリームとして、クロックサイクルごとに 1 つの特徴を畳み込みニューラルネットワーク層に供給されます。
ニューラルネットワーク層の後で、データはより管理しやすいデータストリームを提供するために任意に拡大されます。いくつかの層の後には、MaxPool 層が続くことがよくあります。これは、ニューロン出力面の小さな領域から最大値を選択することによってデータ出力の量を減らす単純な非線形フィルタリング手法です。これはゼロパディングと機能バッファキャッシュモジュールを再利用して同様の機能を提供します。ただし、maxpool フィルタは非常に単純で、積和演算よりも比較演算を実行します。maxpool ステージの後、データはストリームとして出力され、メモリに書き戻すことも次のレイヤに渡すこともできます。

データストリームと並列化：

前のセクションではデータストリームについて詳しく説明しました。これは、FPGA の実装を正しく理解しソリューションが効率的かどうかを判断するために、データをメモリ内の静的オブジェクトと見なすのではなく、データがチップ内をどのように移動するかを理解する必要があるためです。
ストリームにはいくつかの異なる定義と意味があります。最も単純な場合、ストリームは、クロックサイクルごとに変化するデータを含む信号になります。データ有効信号を追加し、いつデータが存在するのかまたいつ信号を無視できるのかを示すことで、最初のそしてほぼ不可欠なレベルの制御が追加されます。データのパケット（オブジェクト、機能、フレーム）の最初と最後をマークするために最初と最後のストローブを追加すると、より多くの制御が可能になります。しかしながら、これらの信号はフロー制御されていないタイプのものです。データを受け取るロジックは常にこのデータを受け付けることができなければなりません。ニューロンの積和演算ユニットのような単純な算術演算ユニットはこの仮定で容易に設計することができます。パケット内で連続していないデータを想定すると、ロジックが複雑になる可能性があります。
場合によっては、フローに制御を加える必要があります。PCIe を介してまたはメモリから到来する（または PCIe またはメモリに送信される）データは通常この場合に該当します。これらのストリームには有効なシグナルがあるだけでなく、受け入れる能力を示す準備完了シグナルもあります。feature_buffer キャッシュメモリのように、準備ができるまでデータを遅らせる設計内のモジュールは、フロー制御ストリームを使用するロジックの例です。フロー制御されたストリームは多くの状況で有用であり、ARM/Xilinx AXI4-Stream プロトコルのようなスタンダードがあります。
これらの設計例で使用される 3 番目のタイプのストリームは、パケットレベルのフロー制御ストリームです。狭帯域化接続をストリーミングするための機能バッファはその一例です。レディフラグは、データの各パケットの開始前にのみチェックされます。これにより、2 つのモジュール間の接続ロジックが大幅に簡素化され、クロックサイクルごとにデータのフロー制御が不要になりタイミングクロージャの困難につながる可能性があります。
処理されるデータをストリームにパッケージ化することによって、それを必要とする異なるモジュールへチップ周りのフローを可能にします。しかしフロー制御されていない場合、受信側は常にデータを受け入れることができなければなりません。これは特定の幅では問題になるかもしれませんが、ストリームを広げ受信ハードウェアの並列インスタンスを作成することによって、受信側が受信データでオーバーフローしないようにする必要があります。

CNN レイヤの PCIe FPGA 実装：

このセクションでは、PCIe フォームファクタボードにおける FPGA 上の CNN レイヤの実用的な実装について説明します。実用的な FPGA 実装の 1 つの主要なキーは、ストレージから、ネットワークから、あるいはホストＣＰＵ上で実行されているアプリケーションからのデータの入出力を効率的に処理することです。この例では、PCIe バスを介して効率的なレートで DMA データの単純なストリーミングアプローチを使用します。ストリーミングアプローチは一般的に大きな連続したデータセットで最良の動作をします。それは幸いにも機械学習推論、画像のバッチ処理のための要件にマッチします。したがって、このアプローチは複雑さを犠牲にしてより高い柔軟性を提供する、より複雑なメモリマップ構造の代わりに使用されます。

Figure 7 は、デザインの最上位構造を示しています。このデザインは、Alpha Data ADM-PCIe-8K5 ボードをターゲットにしています。ボード固有の PCIe IP コアは PCIe とユーザーデザインの間のインターフェイスに使用されます。このコアは、ホスト API 機能に応答して、250MHz のクロックで幅 256bit の AXI4 DMA ストリームを提供および消費するようにコンフィギュレーションできます。チャネル 0 は入力データを提供するために使用されます。これは、レイヤ 0 モジュールのフィーチャバッファキャッシュの入力幅に合わせて、幅が 3 バイトに縮小されています。重みは、異なる DMA チャネル上のストリームを使用して初期化されます。重みを初期化するタスクに固有のストリーム絞り込みモジュールが使用されます。このモジュールはストリームデータをバッファし、各ニューロンの重みに対して連続的にバーストし、予想される入力動作と一致するように最初と最後のストローブを生成します。この狭いストリーム構造は、極端な FPGA ファブリック配線を必要とせずにチップ上のすべてのニューロンにウェイトデータを送信するための低帯域幅の手段を提供します。レイヤ出力データは、別の DMA チャネルを使用して PCIe 経由で送り返されます。メモリマップされたダイレクトスレーブポートは、ザイリンクスの AXI4 幅およびプロトコル変換 IP コアを介して使用され、ソフトウェアがパフォーマンスカウンタ値をレポートするために読み込むことができるレジスタバンクにアクセスするために使用されます。

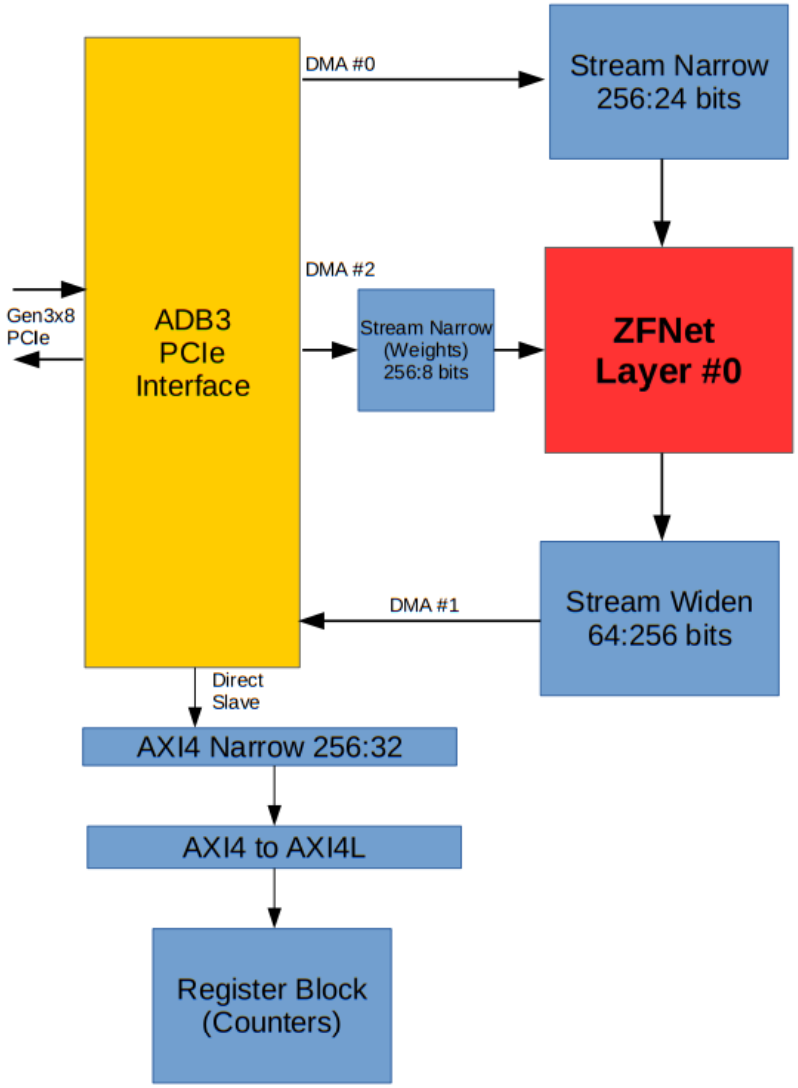

Figure 7 : Example PCIe FPGA Implementation

サンプルをビルドするには、ザイリンクス Vivado 合成インプリメンテーションツールが必要です。バージョン 2017.1 がデフォルトのターゲットですが古いバージョンが使用されることもあります。古いバージョンでは、conv_neuron VHDL コードで use_dsp 属性を use_dsp48 に変更する必要があるかもしれません。この例では、RD-8K5 サポートパッケージを購入するときに ADM-PCIE-8K5 の顧客が利用できる ip_repo/adb3_admpcie8k5_x8_axi4_v1_2.zip ファイルも必要です。

ビルドスクリプトを実行する前に、このファイルを ip_repo の場所にコピーする必要があります。プロジェクトは TCL スクリプト build_cnn_fpga_pcie.tcl を使用して構築されています。これは Vivado のコマンドラインモードを使用して実行できます。

vivado -mode batch -source build_cnn_fpg_pcie.tcl

これによりプロジェクトが作成され、GUI が開きます。その後、GUI を使用して合成とインプリメンテーションを実行できます。ソースコードでは、いくつかの最適化を盛り込んでいます。ザイリンクスデバイスでは、2 つの 8 ビット乗算演算を 1 つの DSP48 タイルに簡単に収めることができます[3]。

しかしながら、これは入力の一方が両方の乗算に対して同じであると仮定します：

P1+2^16*P2 = A*B1+A*B2*2^16

この状況は、同じデータがすべてのニューロンに渡されるため、ニューラルネットワークレイヤで発生します。したがって、これを利用する最も簡単な方法は 2 つの隣接するニューロン間で乗算（DSP タイル）を共有することです。ファイル conv_neuron_shmu.vhd および conv_neuron_layer_ffanout_shmu.vhd はこの最適化を実装し、必要な乗数の半分になります。この最適化は必ずしもアキュムレータの使用を改善するわけではありませんが、パフォーマンスへの影響がそれほど重要ではないため、ファブリック内で追加の実装をすることもできます。

Table 1 に単一層のリソース利用率を示します。

Resource	Utilization	Available	Utilization %
LUT	36290	663360	5.47
LUTRAM	6051	293760	2.06
FF	64557	1326720	4.87
BRAM	169.5	2160	7.85
DSP	96	5520	1.74
IO	48	624	7.69
GT	8	48	16.67
BUFG	8	1248	0.64
MMCM	1	24	4.17
PCIe	1	6	16.67

Table 1 : ZFNet Layer 0 Implementation in KU115

推定チップ電力は 7.24W です。

これらの結果は多くのことを示しています。 第一に、96 個のニューロンは、48 個の共有乗算のみを使用していますが、それでも 96 個の独立した累積を使用しています。しかし、乗算が不足している場合、加算はほぼ同じ性能で LUT に実装できます。もう 1 つの重要な洞察は、レイヤが KU115 チップで利用可能なリソースのごく一部を使用しているということです。これは、ネットワーク全体をこのデバイスに適合させる、またはネットワーク層を小型の低電力デバイスに適合させる余地があることを意味します。これは、おそらくカメラ入力ストリームに直接接続される可能性があります。DSP と BRAM の比率は約 1.75%から約 8%です。これは、スケールアップすると BRAM リソースがサイズを制限する要因になることを意味します。故に、この場合にはより高い DSP 対 BRAM（乗数対重み）比を有するように層を再構成することが望ましいです。

CNN レイヤの CAPI FPGA 実装：

このセクションでは、CAPI ベース FPGA デザインの同じ CNN レイヤへのターゲティングについて説明します。CAPI は IBM Power 8 サーバーのプロトコルで PCIe バス上で動作しますが、高レベルのメモリ一貫性とセキュリティを提供します。これは FPGA が、同じユーザー空間の仮想アドレスを使用してホストアプリケーションのユーザー空間メモリにアクセスすることができ、PCIe ドライバのオーバーヘッドでホストメモリにアクセスできることを可能にします。そして理論的には物理的なホストアドレスにアクセスしシステムを破壊する可能性がある、基本的な PCIe 設計では利用できないセキュリティレベルが最も重要です。CAPI はまた、完全な再構成を可能にします。したがって、CAPI および Power 8 実装はクラウド FPGA 実装においてセキュリティ上の大きな利点があります。

既存の PCIe デザインを CAPI に移植するために、デザインは同じストリームベースのデザインを使用します。ネイティブ CAPI PSL インターフェイスから AXI-Stream インターフェイスに変換する最上位の既製の AFU テンプレートデザインを使用して、同様の DMA ライクインターフェイスをユーザーコードに提供します。バルクデータストリームは、CAPI ではデフォルトで 512 ビット幅であるため、わずかに異なるストリーム変換ロジックが必要ですが、中央の CNN コアは変更せずに使用できます。

このデザインを構築するには、ユーザーは ADM-PCIE-KU3 または ADM-PCIE-8K5 CAPI カード付きの CAPI 開発キットを購入している必要があります。圧縮ファイルには、xilinx_fpga_capi / Sources / prj にあるプロジェクトファイルを含むフォルダ xilinx_fpga_capi/Sources/afu に afu を構築するためのすべてのソースコードが含まれています。これらのファイルを CAPI ビルドディレクトリにプラグインして、Vivado を通常の TCL ビルドスクリプトで実行し、実装可能な CAPI ビットストリームを生成することができます。

最近発売された CAPI-SNAP フレームワークでも同じ設計を再ターゲット化することができます。CAPI SNAP アクションで ZFNet レイヤをカプセル化するために使用できるアクションファイルの例が SNAP フォルダにあります。同様の 512 ビット幅のストリーミングインターフェースがデータと重みをホストメモリとの間で転送するために使用されます。

複数の CNN レイヤの最適化された PCIe FPGA 実装：

このセクションでは、ZFNet ネットワーク全体のより最適化された実装について考察します。コーディングの説明をする前に、ネットワークのスプレッドシート分析を行う必要があります。さまざまな CNN のスプレッドシートがフォルダスプレッドシートに含まれています。ZFNet の主なパラメータは Table 2 の通りです。

このネットワークでは、5 つの畳み込み層と 3 つの完全接続層の 8 つの層が考慮されます。スプレッドシートの主な目的の 1 つは、各データ要素（入力または重み）を計算に使用する必要がある回数を計算すること、およびさまざまなネットワーク層の計算要件をどのようにバランスさせるかを計算することです。 各レイヤが並行して動作する場合はスループットを最大化し、それらのパフォーマンスを一致させる必要があります。

入力サイズは実際の入力サイズ（ゼロ埋め）に依存しますが、乗算は実際には"有効な"行と列の値に依存します。これはストライドとマスクサイズを考慮して、1 フレームあたりの各ニューロン計算に使用される実際の入力点数を返します。

フレームごとの計算入力画素の再利用率は、畳み込みフィルタリングのために入力画素をキャッシュすることによって得られる利益のレベルを特定します。これら画素の実際の再利用はこれにニューロンの数を掛けたものです。これは、実際のレベルでバランスをとる必要があるレイヤ乗算性能に対する入力データ帯域幅の指標を与えます。
ニューロンごとおよび層ごとの乗算は、各層に必要な性能の指標を与えます。パイプラインで 1 つの層を次の層に供給することが望ましい場合、これらの性能は層の並列化を増減することによって一致しなければなりません。レイヤ間のデータをレイヤ間で外部メモリに格納し、レイヤを順番に実行してウェイト値を切り替えた場合（代替の実装戦略、データとウェイト値をオフチップでバッファリングする必要があるため電力効率が低い） バランスをとることはそれほど重要ではありません。

Layer	0	1	2	3	4	5	6	7
Type	Convolution Layers					FC	FC	FC Softmax
Input Rows	224	55	13	13	13	1	1	1
Input Cols	224	55	13	13	13	1	1	1
ZP Rows	227	55	15	15	15	1	1	1
ZP Cols	227	55	15	15	15	1	1	1
Skip	2	2	1	1	1	1	1	1
Mask Rows	7	5	3	3	3	1	1	1
Mask Size	7	5	3	3	3	1	1	1
Channels	3	96	256	384	384	9216	4096	4096
Input Size	150528	290400	43264	64896	64896	9216	4096	4096
Neurons	96	256	384	384	256	4096	4096	1000
Effective Rows	110	26	13	13	13	1	1	1
Effective Cols	110	26	13	13	13	1	1	1
Mults per neuron-per pixel	147	2400	2304	3456	3456	9216	4096	4096
Mults per neuron	1778700	1622400	389376	584064	584064	9216	4096	4096
kMults for layer	170755	415334	149520	224280	149520	37748	16777	4096
Input Pixel re-use ratio	12.25	6.25	9	9	9	1	1	1
Layer Performance Requirement	14.62%	35.56%	12.80%	19.20%	12.80%	3.23%	1.44%	0.35%
Weight re-use per frame	12100	676	169	169	169	1	1	1
Weight Memory Size (kB)	14	614	884	1327	884	37748	16777	4096

Table 2 : ZFNet Resource User Spreadsheet

ウェイトメモリとメモリ帯域幅の要件も計算されます。畳み込みネットワークのためのフレーム当たりの高い重みの再使用は、これらの値をチップ上に記憶することにおける大きな利点を示しています。畳み込み層におけるこれらのパラメータの比較的小さいメモリフットプリントは、それらが計算のために内部メモリに完全に適合することを可能にします。
畳み込みネットワーク層の総ウェイトメモリサイズは 3.7MB になります（8 ビット係数、係数あたり 1 バイトと仮定）。完全接続層の総重量メモリサイズは 58MB になります。

完全に接続された層に対するこのはるかに大きいメモリ要件は、畳み込み層構造およびコードを単純に再利用することができないことを示しています。レイヤのパフォーマンス要件を見ると、完全に接続されたレイヤは計算負荷の 5%しか占めていません。そのため、これらのレイヤを別々に扱い、別の回路を設計し、場合によってはこれらを処理するために別の FPGA 上で実行することは非常に理にかなっています。他のワークロードがほとんどない場合は、このタスクをホストシステムの CPU で処理することもできます。この段階でのデータ出力のメモリ帯域幅は比較的小さいため、それを別のリソースに転送したり、後でバッチ処理するために外部 RAM に格納したりすると、エネルギーが比較的低くなります。

前の表は、元のネットワークを指定して学習したユーザーによって定義された一般的なネットワーク構造を示しています。レイヤ入力サイズ、ニューロン数、ウェイトメモリサイズなどのパラメータの多くは固定されます。ただし、レイヤ構造と並列化は、データを最大限に活用して通過させるように最適化できます。

次の表は、並列化の調査を可能にするスプレッドシートの一部を示しています。並列化係数を大きくすることで、デバイスで利用可能な数の DSP を使用することを目的として、DSP 対ウェイト比を大きくすることができます。レイヤを介してフレームをプッシュするためのクロックサイクル数が計算されます。すべてのレイヤがパイプラインとして実装されているため、最も遅いレイヤがスループットを決定します。異なる層で異なる並列化係数を使用すると、すべての層のスループットのサイクル数をできる限り近づけることによって、可能な限り最高の負荷率で DSP を使用するように指定できます。

Layer	0	1	2	3	4
Parallelization Factor	4	4	1	2	2
DSPs Required	384	1024	384	768	512
Cycles	447700	405600	389376	292032	292032
Utilization	100%	90.60%	86.97%	65.23%	65.23%
DSP Utilization	384	927.7	333.9	500.9	333.9

Table 3 : ZFNet Implementation

この設定のための総 DSP ハードウェア要件は 3072 DSP 乗算器です。これらは 80%の効率で利用されます。クロックレートを 250MHz と仮定すると、ネットワークは 80MB/s のレートで処理し 1.79ms ごとに新しいフレームを受け取ることができます。これは KU115 デバイスではかなり控えめな使用方法です。DSP タイルの半分以上を使用し、比較的低速でクロックを供給し、1 つの乗算ブロックで 2 つのニューロンを共有する手法を利用しない。しかし、デザインは比較的迅速に配置配線されるため、何が可能かを理解するのに役立ちます。

このプロジェクトは TCL スクリプト build_cnn_fpga_pcie.tcl を使用して構築されています。これは Vivado のコマンドラインモードを使用して実行できます。

vivado -mode batch -source build_zfnet_fpga_pcie_opt.tcl

これによりプロジェクトが作成され、GUI が開きます。その後、GUI を使用して合成と実装を実行し、FPGA のビットストリームを作成できます。

実施結果を下表に示します。DSP タイルの数が予想より多くなっています。これは、par2 および par4 ニューロンで使用されている追加の加算回路によるものです。LUT はあまり使用されていないため、これらの追加をスライスロジック

で実行してリソースを節約できます。代わりに、それらは ReLU 回路と共に層内の全てのニューロンの間で共有されることができ、付加的なロジック/DSP 資源を節約しますが、層出力シフトレジスタ内の経路指定およびレジスタ使用を増加します。

Resource	Utilization	Available	Utilization %
LUT	128784	663360	19.41
LUTRAM	6825	293760	2.32
FF	267608	1326720	20.17
BRAM	1228.5	2160	56.87
DSP	3760	5520	68.12
IO	48	624	7.69
GT	8	48	16.67
BUFG	8	1248	0.64
MMCM	1	24	4.17
PCIe	1	6	16.67

Table 4 : ZFNet Implementation in KU115

推定消費電力は 29W と表示されています。

このネットワークは 5 つの畳み込み層を実装しただけです。完全に接続されたレイヤを同じ FPGA に実装するには、これらのレイヤの重みをチップ外に、おそらく DDR4 メモリに格納する必要があります。これは ADM-PCIE-8K5 ボード上で 19.2GB/s の 2 バンクでかなり高い帯域幅を持っていますが、このデータをすべてのフレームにロードするには 30GB/s を必要とします。より低い帯域幅のアプローチは、処理のためにフレームをまとめます。各レイヤ 5 フレームは 9kB の入力メモリと約 8kB のアキュムレータメモリを必要とします。レイヤ 6 では、4kB の入力メモリと 8kB のアキュムレータ、レイヤ 7 の 4kB の入力と 2kB のアキュムレータメモリが必要となるため、1 フレームあたり約 35kB のメモリ（約 8x36kB の BRAM）が必要になります。したがって、バッチあたり最大 32 フレームを処理するバッチ処理では 256 個の BRAM が必要になりますが、DDR4 の帯域幅要件は 1GB/s 未満に減少します。

スプレッドシートは、他のいくつかの一般的なネットワークにも提供されています。同じ一般的な構造を使用し、ADM-PCIE-8K5 ボード上の KU115 デバイスをターゲットにすると、AlexNet のフレームあたりのスループット時間は、控えめな 250MHz クロックで約 584us になり、乗算器を共有しないことがわかります。14MB のオンチップメモリ要件のため、VGG Net を 2 つの FPGA に分割して完全にパイプラインで並列実行する必要がありますが、これはスループット時間 5.4ms を達成することができます。2 つの畳み込み層を持つ基本的な CIFAR10 Tensor Flow の例は、非常に高度の並列化でチップに簡単に収まり、完全に接続された層の重みをチップ上に維持するためのスペースがあります。

まとめ：

このホワイトペーパーでは、Machine Learning Inference の実装における FPGA テクノロジの使用を評価するためのオープンソースリソースについて説明しました。オープンソースツールと互換性のあるオープンソースコードは、事前に学習されたネットワークの構造評価および算術評価に使用して、FPGA の導入に対する適合性を見積もることができ

ます。主な目的の 1 つは、ネットワーク精度に対する固定精度算術の影響をテストする簡単で直接的な CPU 互換の方法を提供し、各層を希望の精度に最適化できるようにすることです。固定精度の実装サイズと構造が選択されると、これをインプリメントするためのサンプルコードが提供されますが、他の利用可能な FPGA CNN ライブラリソリューションとの比較参照に使用することが可能になります。お客様が必要に応じてレイヤを最適化できるように、実際の FPGA ハードウェアをターゲットとしたこれらのネットワークの例を示しました。おそらく提供されている最も強力なツールはコードではなく、分析用のスプレッドシートの例です。これは、ネットワークのパフォーマンスとメモリ要件を分割して適切な FPGA リソースの使用を見積もるためのものです。これらはサンプルの HDL オープンソースコードと一緒に使用できますが、サードパーティの CNN ライブラリを使用してハードウェア上の潜在的なパフォーマンスの有用な指標を提供することもできます。これらは、さまざまな FPGA デバイスを比較して配置のための適合性や各デバイスを判断するのに特に役立ちます。

参考資料：

[1] Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," Lecture Notes in Computer Science, vol. 8689, pp. 818?833, 2014, Springer
[2] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Advances in Neural Information Processing Systems 25, pp. 1106?1114, 2012
[3] Yao Fu, Ephrem Wu, Varun Santhase elan, Kristof Denolf, Kamran Khan, and Vinod Kathail, WP490: Embedded Vision with INT8 Optimization on Xilinx Devices, Xilinx, April 19, 2017

Alpha Data 社について

Alpha Data は、1993 年に設立され、計算集約型アプリケーションをターゲットとした最先端の FPGA ソリューションを提供しており、FPGA アクセラレータのマーケットリーダとして市場を牽引しています。主な製品は VPX、XMC、PMC、PCI、CompactPCI、PCIExpress、VXS、VME などの A/D, D/A, FPGA ボードや CameraLink 等のデジタル I/F を搭載した信号処理ボードです。ボーイング、ロックウェルコリンズ、JPL、ロッキードマーチン、モトローラ、BAE などのミリタリ向けやデータセンター向けに広く採用されています。Alpha Data 社の詳細については、https://www.alpha-data.com/を参照してください。