第 7 章
INT8 ディープラーニング推論

この章では、インテル® テクノロジー上のディープラーニング推論向けの INT8 データ型について説明します。ここでは、インテル® AVX-512 実装と新しいインテル® Deep Learning Boost (インテル® DL Boost) 命令を使用した実装の両方についてカバーします。

この章はいくつかの節に分けて説明します。最初の節では、INT8、具体的にはインテル® DL Boost 命令を主なデータタイプとして説明し、ML ワークロードで使用する命令を紹介します。2 番目の節では、効率良い推論計算向けの一般的な方法論とガイドラインでについて説明します。3 番目の節では CNN 向けの最適化について説明し、最後の節で LSTM/RNN 固有の最適化について説明します。

関連がある場合、インテル® DL Boost 命令の有無にかかわらず例を示します。多くの場合 (量子化やメモリーレイアウトなど)、オフラインで実行可能なステップと、実行時に行うステップがあります。各ステップについてその都度明確に説明していきます。

7.1 ディープラーニング推測向けの INT8 データ型について

伝統的にディープラーニングでは、単精度浮動小数点 (F32) データ型が使用されてきました (https://itpeernetwork.intel.com/myth-busted-general-purpose-cpus-cant-tackle-deep-neural-networktraining/#gs.rGp9lgWH を参照)。INT8 は精度をほとんど低下させることなく大幅なパフォーマンス向上をもたらし、ディープラーニングの推論で採用されはじめています。単一の F32 FMA 命令に対し、INT8 MAC 操作に必要な 4 つのナローデータ型と 3 つのインテル® AVX-512 命令は、ほぼ 1.33 倍のゲインをもたらします。Skylake Server† マイクロアーキテクチャー・ベースのインテル® Xeon® スケーラブル・プロセッサーで、ResNet-50、Inception-Resnet V2、SRGAN および NMT を評価した結果、INT8 の少ないメモリー・フットプリントにより、1.5 倍以上のスピードアップを観測できました。7.2 節では、Cascade Lake† 製品ベースのプロセッサーで導入されたインテル® Deep Learning Boost 命令について説明します。これにより、DL 推論のパフォーマンスがさらに向上します。

ここでは、DL 推論の 2 つの利用ケースを検討します。最初のケースはスループット・モデルです。ここでは、要素 (イメージや文章) が単一の要素にかかる時間に関係なく処理されます。通常、この使用モデルは、大量のイメージを処理して分類したり、特定のユーザーに合わせた推奨事項をオフラインで準備するサーバーに適しています。次の使用モデルは、単一要素の計算時間に制限があるスループット・レイテンシー・モデルです。この使用モデルは、オンライン計算 (言語翻訳、リアルタイム・オブジェクト検出など) に適しています。

7.2 インテル® DL BOOST について

インテル® DL Boost 命令はインテル® AVX-512 命令に含まれ、ニューラル・ネットワークのワークロードをスピードアップするように設計されています。ここでは、この新しい命令について説明し、以前のインテル® AVX-512 コードとの簡単な比較を示します。命令の完全な定義 (VPDP プリフィクスと命令) については、『インテル® アーキテクチャー命令セット拡張プログラミング・リファレンス』または『インテル® 64 および IA-32 アーキテクチャー・ソフトウェア開発マニュアル』を参照してください。

7.2.1 符号なしおよび符号ありバイトの積和演算 (VPDPBUSD 命令)

VPDPBUSD は、32 ビット整数アキュムレーターで 8 ビット整数積和ベクトル演算を行います。2 つのソース・ベクトル・オペランドと 1 つのソース/デスティネーション・ベクトル・オペランドから成ります。2 つのソースオペランドは、8 ビット整数データ要素を含みます。ソース/デスティネーション・オペランドは、32 ビット・データ要素を含みます。

例えば、各ソースオペランドが 64 x 8 ビット要素を含み、ソース/デスティネーション・オペランドが 16 x 32 ビット要素を含む 512 ビットのベクトルオペランドについて考えます。

この命令は、各ソースオペランドの 64 個の要素を 16 個の要素から成る 4 つのグループに分割し、それぞれのソースオペランドから 1 グループずつ、そのグループの 4 要素を垂直方向に乗算して、4 つの 32 ビット中間結果を生成します。次に、4 つの 32 ビット中間結果と 3 番目のベクトルオペランド (ソースオペランド) の垂直方向に対応する 32 ビット整数要素の 5 方向加算を実行し、その結果を 3 番目のベクトルオペランド (デスティネーション・オペランド) に 32 ビット・データ要素として格納します。インテル® AVX-512 は 16 ビットの中間結果を飽和させるため、同じ機能をより高い精度で実現する VPDPBUSD でインテル® AVX-512 の 3 つの命令シーケンス (VPMADDUBSW + VPMADDWD + VPADD) を置き換えます。図 7-1 を参照してください。

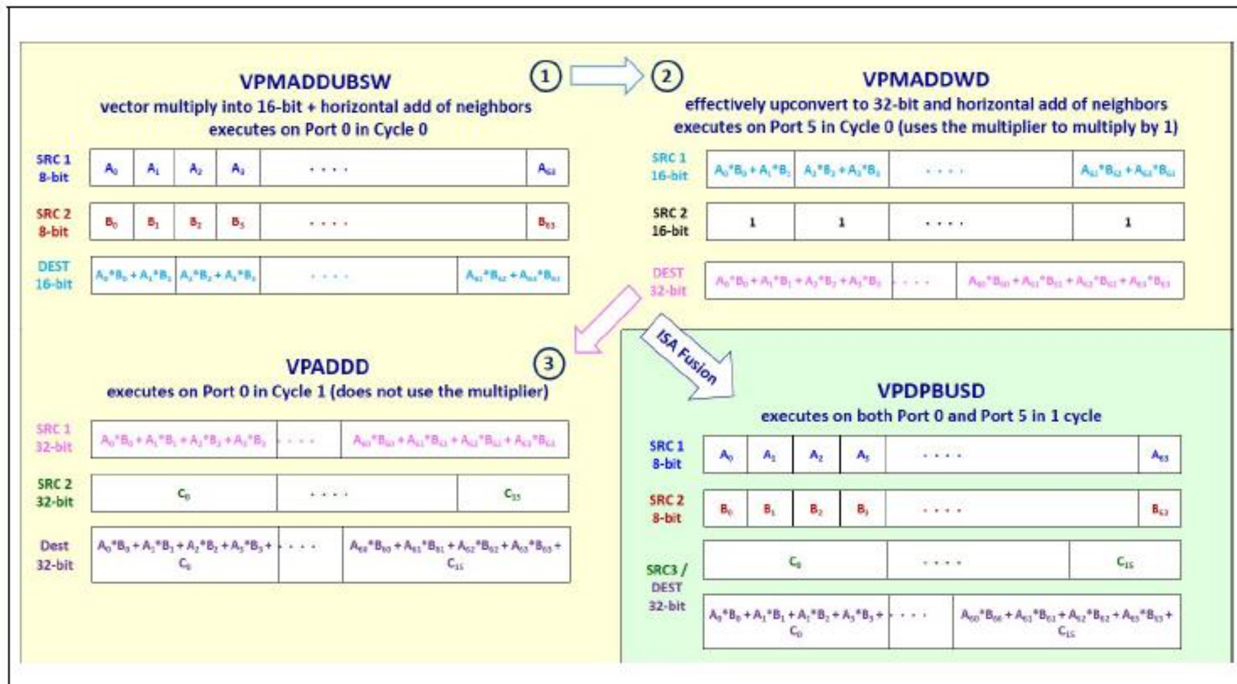

図 7-1. VPMADDUBSW + VPMADDWD + VPADDD を VPDPBUSD に融合 (3x ピーク Ops)

例 7-1. VPDPBUSD 実装

インテル® DL Boost 以前のベクトル実装 (インテル® AVX-512)	**インテル® DL Boost の VPDPBUSD 実装**
// アンロールされたドット積の内部ループ	// アンロールされたドット積の内部ループ
vpbroadcastd zmm31, dword ptr [onew]	vpbroadcastd zmm24, signal
vpbroadcastd zmm24, signal	vpbroadcastd zmm25, signal + 64
vmovups zmm25, weight	vpbroadcastd zmm26, signal + 128
vmovups zmm26, weight + 64	vpbroadcastd zmm27, signal + 192
vmovups zmm27, weight + 128	vmovups zmm28, weight
vmovups zmm28, weight + 192	vmovups zmm29, weight + 64
vpmaddubsw zmm29, zmm24, zmm25	vmovups zmm30, weight + 128
vpmaddwd zmm29, zmm29, zmm31	vmovups zmm31, weight + 192
vpaddd zmm0 , zmm0 , zmm29	vpdpbusd zmm0 , zmm24, zmm28
vpmaddubsw zmm30, zmm24, zmm26	vpdpbusd zmm6 , zmm24, zmm29
vpmaddwd zmm30, zmm30, zmm31	vpdpbusd zmm12, zmm24, zmm30
vpaddd zmm6 , zmm6 , zmm30	vpdpbusd zmm18, zmm24, zmm31
vpmaddubsw zmm29, zmm24, zmm27	vpdpbusd zmm1 , zmm25, zmm28
vpmaddwd zmm29, zmm29, zmm31	vpdpbusd zmm7 , zmm25, zmm29
vpaddd zmm12, zmm12, zmm29	vpdpbusd zmm13, zmm25, zmm30
vpmaddubsw zmm30, zmm24, zmm28	vpdpbusd zmm19, zmm25, zmm31
vpmaddwd zmm30, zmm30, zmm31	vpdpbusd zmm2 , zmm26, zmm28
vpaddd zmm18, zmm18, zmm30	vpdpbusd zmm8 , zmm26, zmm29

vpbroadcastd zmm24, signal + 64 vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm1 , zmm1 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm7 , zmm7 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm13, zmm13, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm19, zmm19, zmm30 vpbroadcastd zmm24, signal + 128 vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm2 , zmm2 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm8 , zmm8 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm14, zmm14, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm20, zmm20, zmm30 vpbroadcastd zmm24, signal + 192 vpmaddubsw zmm29, zmm24, zmm25 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm3 , zmm3 , zmm29 vpmaddubsw zmm30, zmm24, zmm26 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm9 , zmm9 , zmm30 vpmaddubsw zmm29, zmm24, zmm27 vpmaddwd zmm29, zmm29, zmm31 vpaddd zmm15, zmm15, zmm29 vpmaddubsw zmm30, zmm24, zmm28 vpmaddwd zmm30, zmm30, zmm31 vpaddd zmm21, zmm21, zmm30	vpdpbusd zmm14, zmm26, zmm30 vpdpbusd zmm20, zmm26, zmm31 vpdpbusd zmm3 , zmm27, zmm28 vpdpbusd zmm9 , zmm27, zmm29 vpdpbusd zmm15, zmm27, zmm30 vpdpbusd zmm21, zmm27, zmm31
ベースライン	スピードアップ: 2.75x

7.2.2 符号ありワード整数の積和演算 (VPDPWSSD 命令)

VPDPWSSD は、32 ビット整数アキュムレーターで 16 ビット整数積和ベクトル演算を行います。2 つのソース・ベクトル・オペランドと 1 つのソース/デスティネーション・ベクトル・オペランドから成ります。2 つのソースオペランドは、16 ビット整数データ要素を含みます。ソース/デスティネーション・オペランドは、32 ビット・データ要素を含みます。

8 ビットの VPDPBUSD 命令で推論の精度に損失の問題が生じる場合、代わりに 16 ビットの VPDPWSSD 命令を使用できます。新しい BFLOAT16 データ型とそれに関連する命令がインテル® プロセッサーでサポートされるまで、このようなシナリオでは FP32 命令に戻すことを推奨します。

7.3 一般的な最適化

7.3.1 メモリーレイアウト

NHWC メモリーレイアウトが TensorFlow* パフォーマンス・ガイドの記載に従っていると仮定します (詳細に関しては、このドキュメントのデータ形式の節を参照)。入力がネイティブ形式 (行または列のどちらか) である場合、データは計算開始時にスカラーコードによって最適化されたレイアウトに変換されます。詳細については、インテル® AI Academy を参照してください。

7.3.2 量子化

量子化は、活性化と重みにおいてデータ型のサイズを減らす (通常 float から int8/uint8 に) プロセスであり、インテル® MKL-DNN ドキュメントやインテル® AI Academy などさまざまなリソースで詳しく説明されています。

7.3.2.1 重みの量子化

重みは、量子化係数 127/max_range を使用して量子化されます。これは、精度を高めるため OFM ごとに実行できます。重みは事前に判明しているため、プロセスはオフラインで実行できます。

7.3.2.2 活性化の量子化

次のコード例に、量子化係数を指定してデータをスカラーまたはベクトル形式で量子化する方法を示します。

例 7-2. 活性化の量子化

指定された係数で入力を量子化 (スカラー)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
{
    int quant_min = 0;
    int quant_max = (1 << bits) - 1;
    #pragma unroll (4)
    for (int i = 0; i < count; i++) {
        int int32_val = offset + round(data[i] * factor);
        int32_val = std::max(std::min(int32_val, quant_max), quant_min);
        u8 quant_val = (u8)int32_val;
        quantized_data[i] = quant_val;
    }
}
```

指定された係数で入力を量子化 (ベクトル)

```
void quantize_activations(const float* data, u8* quantized_data, int count, Dtype factor, int bits, int offset = 0)
{
    int quant_min = 0;
    int quant_max = (1 << bits) - 1;
    int count_aligned = ALIGN(count, INTR_VECTOR_LENGTH_32_bit);
    __m512i offset_broadcast = _mm512_set1_epi32(offset);
    __m512 factor_broadcast = _mm512_set1_ps(factor);
    __m512i quant_min_broadcast = _mm512_set1_epi32(quant_min);
    __m512i quant_max_broadcast = _mm512_set1_epi32(quant_max);
```

```
    #pragma unroll (4)
        for (int i = 0; i < count_aligned; i += INTR_VECTOR_LENGTH_32_bit) {
        __m512 data_m512 = _mm512_load_ps(&data[i]);
        data_m512 = _mm512_mul_ps(data_m512, factor_broadcast);
        __m512i data_i32 = _mm512_cvt_roundps_epi32
                        (data_m512,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
        data_i32 = _mm512_add_epi32(data_i32, offset_broadcast);
        data_i32 = _mm512_max_epi32(data_i32, quant_min_broadcast);
        data_i32 = _mm512_min_epi32(data_i32, quant_max_broadcast);
        __m128i q = _mm512_cvtusepi32_epi8(data_i32);
        _mm_store_si128((__m128i*)(&quantized_data[i]), q);
    }
}
```

7.3.2.3 負の活性化を量子化

VPMADDUBSW と VPDPBUSD は、最初のパラメーターで符号なしの値と 2 番目のパラメーターで符号付きの値の組み合わせのみをサポートします。2 番目のパラメーターでは符号付きの重みを容易にサポートできますが、最初のパラメーターで符号付きの活性化をサポートするには何らかの操作が必要です。
負の活性化の可能性がある場合 (例えば、現在のレイヤーの前に ReLU がないなど)、最初に -128、127 の値に量子化して、次に結果に 128 を加算して負ではない活性化を行います。OFM バイアスから 128* (OFM フィルターのすべての重みの合計) を引くことで、このオフセットを補正します。詳細は、インテル® AI Academy を参照してください。

7.3.3 マルチコアに関する考慮事項

7.3.3.1 大規模バッチ (スループット・ワークロード)

大規模バッチ計算では、ワークを複数のコアに分割することでマルチコア処理から大きな恩恵を受けることができますが、キャッシュの局所性の観点から同じオブジェクト (イメージや文など) を同じ物理コアで処理することが最善です。また、活性化はオブジェクトごとに異なりますが、重みは通常共有できます。マルチスレッド・モデルでは、コア間で重みを容易に共有できます。マルチコアシステムで大規模バッチ入力を処理するガイドラインを次に示します。

1. スレッドモデルを使用して、複数のコア間で重みを共有します。しかし、マルチソケット・マシンでは、ソケット/NUMA ドメインごとに専用のプロセスが必要です。
2. スレッド・アフィニティーとオブジェクト・アフィニティーを定義して同一物理コア上で単一のオブジェクトを処理し、コアのキャッシュ内で活性化を維持します (キャッシュサイズよりも小さな場合)。
3. ワークがコア間で均等にロードされるように、オブジェクトをコア数の倍数でバッチ処理します。
4. すべての物理コアの兄弟スレッド (論理プロセッサー) がアイドル状態であることを確認します。
5. 例えば、次のレイヤーに移行する前にコアに割り当てられているすべてのイメージで同じレイヤーが実行される BFS モードで、コアごとのミニバッチの実行を検討します。これにより、複数の活性化によってコアキャッシュが汚染されますが、重みの再利用が改善し、(場合によっては) パフォーマンスが向上します。ミニバッチは、行列(テンソル) が非常に細く、積和演算ユニットの使用率が低い場合に有用です。

7.3.3.2 小規模バッチ (レイテンシー・ワークロードのスループット)

小規模バッチ処理には、通常、シングルコアでは解決できないレイテンシー要件があります。この場合、単一オブジェクトの処理を複数コアに分割する必要があります。

単一オブジェクトを処理するコア数を増やすと利点が減少することが多いため、最適なコア数を特定します。ときには、バッチ数を若干増やしてもレイテンシーの要件を満足することができます (例えば、1 から 2 または 3 など)。単一のバッチ・インスタンスに最適なスレッド数を特定すると、システムを完全に活用して複数のインスタンスを実行できるようになります。

7.3.3.3 NUMA

高いパフォーマンスを提供する Cascade Lake† 2 ソケットサーバーには、2 つの Cascade Lake† Advanced Performance パッケージが含まれており、各パッケージはインテル® ウルトラ・パス・インターコネクト (インテル® UPI) リンクで接続された 2 つのプロセッサー・ダイで構成され、合計 4 つの NUMA ドメインを持っています。このような構成では、NUMA ドメイン/ダイごとに個別の DL プロセスを維持することが重要です。これは、複数の NUMA ドメインを持つ以前の世代の製品による 2 ソケットの構成でも同様です。

7.4 CNN

7.4.1 畳み込みレイヤー

7.4.1.1 直接畳み込み

インテル® DL Boost ベクトル演算を利用するため、直接畳み込み演算を一連の行列加算と乗算に置き換えるようにします。次の説明では、入力、重み、出力に代わる行列をそれぞれ A、B、C としています。

メモリーレイアウト

入力を行列形式で表現するため、空間次元を A の行 (以降 M 次元と称します)、チャネル (IFM) の次元を A の列 (以降 K 次元と称します) となるように平坦化します。同様に、出力の空間次元は C の行となり、チャネル (OFM) 次元は C の列になります (以降 N 次元と称します)。図 7-2 では、サイズ 5x5 の 6 つのチャネルが畳み込みレイヤーによってサイズ 3x3 の 4 つのチャネルに変換されています。

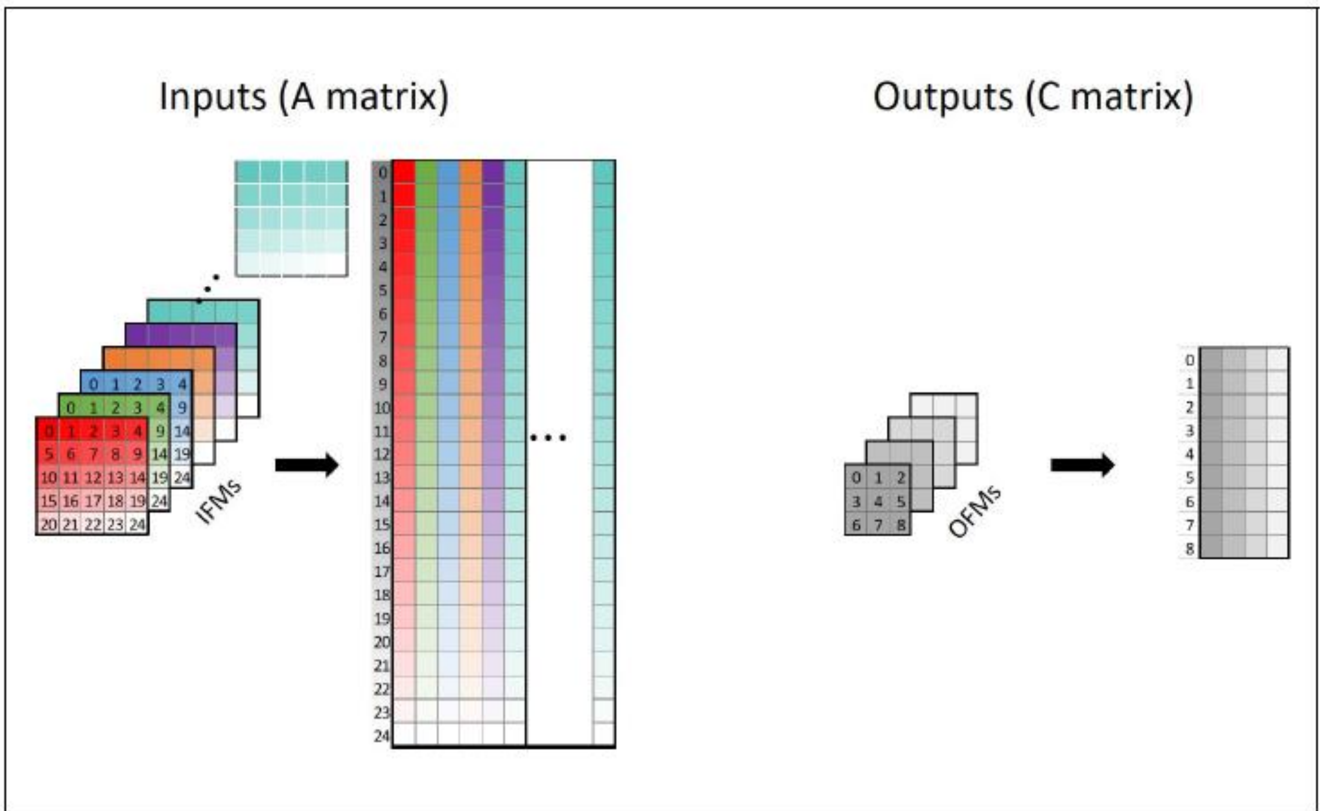

図 7-2. 行列レイアウト、入力および出力

標準の 2D 畳み込みでは、各ターゲット OFM に対してサイズ KH x KW x #IFM の 3D カーネルの差 #OFM を使用しますが、ここで KH、KW、#IFM、および #OFM は畳み込みカーネルの高さと幅、入力チャネル数、および出力チャネル数です。

重みは、サイズ #IFM x #OFM の KH x KW 行列に変換されます (図 7-3 を参照)。

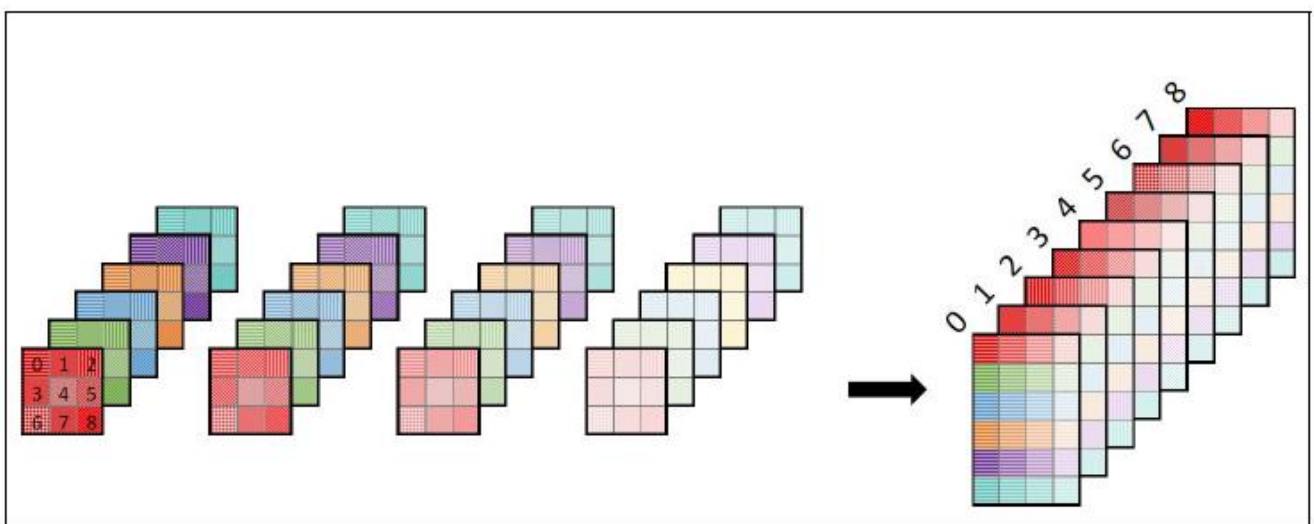

図 7-3. 変換された重み

その結果、畳み込み操作 (図 7-4 を参照) は、

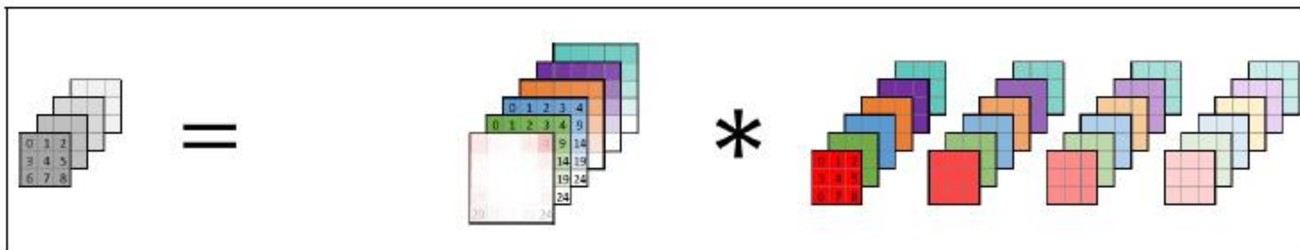

図 7-4. 畳み込み操作

一連の行列の乗算と加算になります (図 7-5 を参照)。

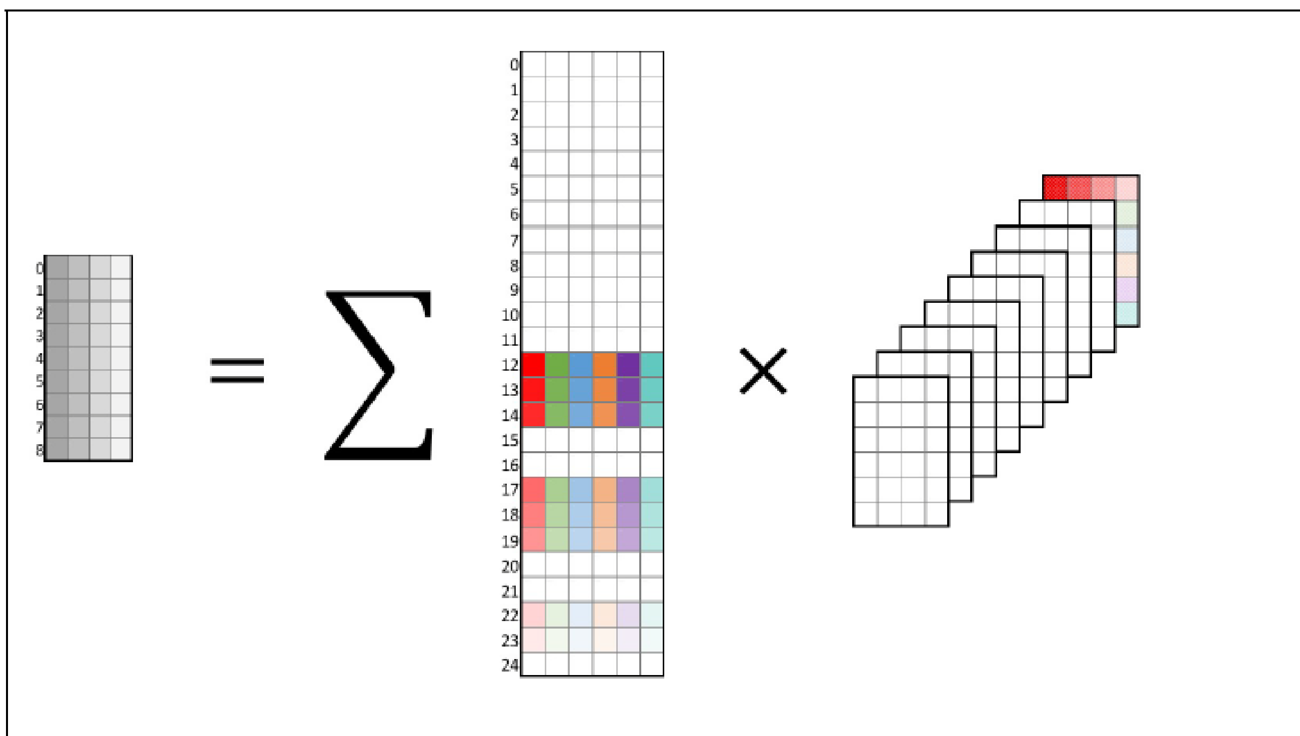

図 7-5. 行列の乗算と総和

行列乗算

行列乗算は標準的な方法で行われます (第 17 章、「インテル® AVX-512 向けの Skylake Server† マイクロアーキテクチャーと最適化」を参照)。

ブロック化

一般に扱う行列は大規模であるため、乗算の結果を累積しながら行列全体を反復的に横断 (トラバース) する必要があります。そのため、結果の累積にいくつかのレジスター (アキュムレーター) を割り当てて、必要に応じて再利用するため A および B 行列を一時的にキャッシュするいくつかのレジスターを用意します。

行列を横断する順番は、全体のパフォーマンスに影響する可能性があります。一般に、M または N 次元で次の要素に移動する前に、K 次元全体を移動することが推奨されます。K 次元の処理が完了すると、アキュムレーター内の結果は最終結果となります (部分的な結果については以下の説明を参照してください)。それらは、畳み込みの後処理ステージ (「畳み込みの後処理」を参照) に送られ、出力場所に保存されます。しかし、M または N 次元で移動する前に K 次元の処理が終わらないと、アキュムレーター内の結果は部分的であり、それらはいくつかの A の列と B の行の乗算結果となります。この結果は、新しいデータチャンク向けにアキュムレーターを解放するため、補助バッファーに保存する必要があります。これらの M、N 座標に戻る場合、結果を補助バッ

ファーからロードする必要があります。そのため、K 次元の余りが生じないシナリオと比較すると、追加のストアとロードが行われることになります。さらに、M または N 次元の移動を制限することを推奨します。一般的に、行列 B のアキュムレーター K キャッシュレベル (図 7-6) が DCU にあり、キャッシュブロックの蓄積サイズ (図 7-7) が MLC に収まり可能な限り大きな場合に最良の結果が得られました。ただし、蓄積サイズがかなり大きい場合でも (最大 MLC の 3 倍) 最良の結果が得られることがあります。これらのハイパーパラメーターは、通常試行錯誤から得られます。

「K 次元の余りが生じない」ガイドラインは、すべてのケースで最適な結果をもたらすわけではありません。また、M、N の局所性の最適な範囲は状況に応じて決定されるべきです。現代の CNN におけるさまざまなシナリオに対応するため、畳み込みプロセスの制御フローを構築する簡単で効果的な方法を概説します。

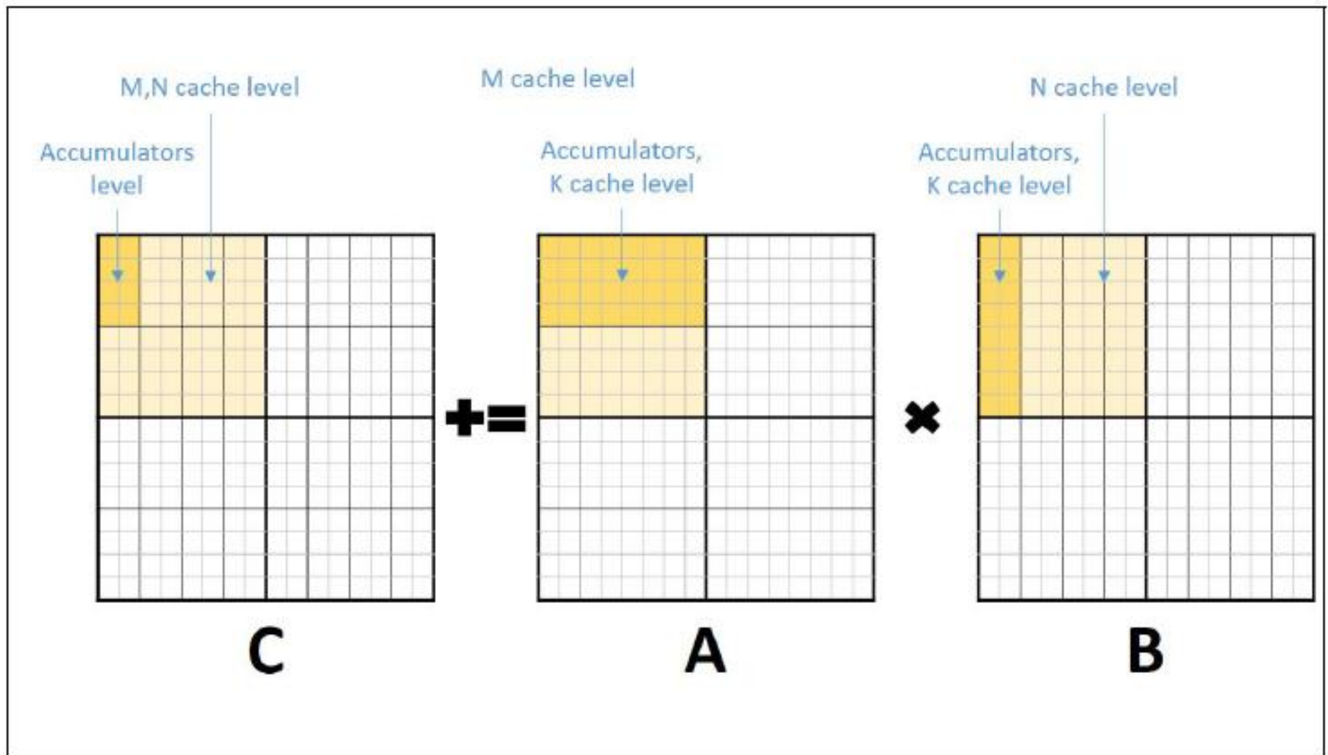

図 7-6. 3 層のフレキシブルな 2D ブロック化

```
n = 0;                              // Rows of A,C (pixels) iterator
m = 0;                              // Cols of B,C (OFMs) iterator
k = 0;                              // Cols of A, rows of B (IFMs)
aC[N_ACCUMS][M_ACCUMS] = 0;         // Accumulators of C

Matrices tiling by "cache blocks"
while(n < N_END)                                                        Not unrolled
  while(m < M_END)
    while(k < K_END)
Partial convolution residing in caches
      for(ni = 0; ni < N_CACHE; ni += N_ACCUMS, n += N_ACCUMS)          Unrolling left to compiler's discretion
        for(mi = 0; mi < M_CACHE; mi += M_ACCUMS, m += M_ACCUMS)
          for(ki = 0; ki < K_CACHE; ki++, k++)
Summation over kernel pixels
            for(kh = 0; kh < KERNEL_H; kh++)                            Always unrolled
              for(kw = 0; kw < KERNEL_W; kw++)
Partial convolution filling accumulators
                for(n_acc = 0; n_acc < N_ACCUMS; n_acc++)
                  for(m_acc = 0; m_acc < M_ACCUMS; m_acc++)
                    aC[n_acc][m_acc] += A(m,k) * B(k,n);

          store_partial_results(C(m,m_acc,n,n_acc), aC);
    store_final_results(C(m,m_acc,n,n_acc), aC)
```

図 7-7. 3 層のフレキシブルな 2D ブロック化ループ

直接畳み込みの例

次のパラメーターを持つ直接畳み込みについて考えます。
IFM サイズ = 34x34、KH = KW = 3、畳み込みストライド = 1、#IFMs = 32、#OFMs = 32、および IFM のパディングなし。つまり、OFM のサイズは 32 x 32 です。

選択されたブロック化は、M_ACCUMS = 4、N_ACCUMS = 2、M_CACHE = 1024、N_CACHE = 2、K_CACHE = 8 です。これらのブロック化パラメーターは、K 次元が完全に横断され、一時的な結果をストアする必要がないことを保証します。次に、OFM のピクセル (0,0)、(0,1)、(0,2)、および (0,3) に対して 32 出力チャネルを計算するため、以下のコードを使用できます。

例 7-3. 直接畳み込み

直接畳み込み

vpxord zmm0 , zmm0 , zmm0 // ゼロ·アキュムレーター·タイル [m,n] = [0,0]
vpxord zmm1 , zmm1 , zmm1 // ゼロ·アキュムレーター·タイル [m,n] = [1,0]
vpxord zmm2 , zmm2 , zmm2 // ゼロ·アキュムレーター·タイル [m,n] = [2,0]
vpxord zmm3 , zmm3 , zmm3 // ゼロ·アキュムレーター·タイル [m,n] = [3,0]
vpxord zmm4 , zmm4 , zmm4 // ゼロ·アキュムレーター·タイル [m,n] = [0,1]
vpxord zmm5 , zmm5 , zmm5 // ゼロ·アキュムレーター·タイル [m,n] = [1,1]
vpxord zmm6 , zmm6 , zmm6 // ゼロ·アキュムレーター·タイル [m,n] = [2,1]
vpxord zmm7 , zmm7 , zmm7 // ゼロ·アキュムレーター·タイル [m,n] = [3,1]
 	
for (int k = 0; k < 32 / 4; ++k) {
    vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=0,k,n=0)] // 重みのロード [kh,kw,n] = [0,0,+0]
    vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=0,k,n=1)] // 重みのロード [kh,kw,n] = [0,0,+1]
    vpbroadcastd zmm8 , dword ptr [IFM_ADDR(0)] // IFM ピクセルのロード +0
    vpbroadcastd zmm9 , dword ptr [IFM_ADDR(1)] // IFM ピクセルのロード +1
    vpbroadcastd zmm10 , dword ptr [IFM_ADDR(2)] // IFM ピクセルのロード +2
    vpbroadcastd zmm11 , dword ptr [IFM_ADDR(3)] // IFM ピクセルのロード +3

```
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=1,k,n=0)] // 重みのロード [kh,kw,n] =
[0,1,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(4)] // IFM ピクセルのロード +4
vpdpbusd zmm5 , zmm9 , zmm13
vpdpbusd zmm6 , zmm10, zmm13
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=1,k,n=1)] // 重みのロード [kh,kw,n] =
[0,1,+1]
vpdpbusd zmm0 , zmm9 , zmm12
vpdpbusd zmm1 , zmm10, zmm12
vpdpbusd zmm2 , zmm11, zmm12
vpdpbusd zmm3 , zmm8 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=0,kw=2,k,n=0)] // 重みのロード [kh,kw,n] =
[0,2,+0]
vpdpbusd zmm4 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(5)] // IFM ピクセルのロード +5
vpdpbusd zmm5 , zmm10, zmm13
vpdpbusd zmm6 , zmm11, zmm13
vpdpbusd zmm7 , zmm8 , zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=0,kw=2,k,n=1)] // 重みのロード [kh,kw,n] =
[0,2,+1]
vpdpbusd zmm0 , zmm10, zmm12
vpdpbusd zmm1 , zmm11, zmm12
vpdpbusd zmm2 , zmm8 , zmm12
vpdpbusd zmm3 , zmm9 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=0,k,n=0)] // 重みのロード [kh,kw,n] =
[1,0,+0]
vpdpbusd zmm4 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(34)] // IFM ピクセルのロード +34
vpdpbusd zmm5 , zmm11, zmm13
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(35)] // IFM ピクセルのロード +35
vpdpbusd zmm6 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(36)] // IFM ピクセルのロード +36
vpdpbusd zmm7 , zmm9 , zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=0,k,n=1)] // 重みのロード [kh,kw,n] =
[1,0,+1]
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(37)] // IFM ピクセルのロード +37
vpdpbusd zmm0 , zmm10, zmm12
vpdpbusd zmm1 , zmm11, zmm12
vpdpbusd zmm2 , zmm8 , zmm12
vpdpbusd zmm3 , zmm9 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=1,k,n=0)] // 重みのロード [kh,kw,n] =
[1,1,+0]
vpdpbusd zmm4 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(38)] // IFM ピクセルのロード +38
vpdpbusd zmm5 , zmm11, zmm13
vpdpbusd zmm6 , zmm8 , zmm13
vpdpbusd zmm7 , zmm9 , zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=1,k,n=1)] // 重みのロード [kh,kw,n] =
```

```
[1,1,+1]
vpdpbusd zmm0 , zmm11, zmm12
vpdpbusd zmm1 , zmm8 , zmm12
vpdpbusd zmm2 , zmm9 , zmm12
vpdpbusd zmm3 , zmm10, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=1,kw=2,k,n=0)] // 重みのロード [kh,kw,n] =
[1,2,+0]
vpdpbusd zmm4 , zmm11, zmm13
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(39)] // IFM ピクセルのロード +39
vpdpbusd zmm5 , zmm8 , zmm13
vpdpbusd zmm6 , zmm9 , zmm13
vpdpbusd zmm7 , zmm10, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=1,kw=2,k,n=1)] // 重みのロード [kh,kw,n] =
[1,2,+1]
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=0,k,n=0)] // 重みのロード [kh,kw,n] =
[2,0,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(68)] // IFM ピクセルのロード +68
vpdpbusd zmm5 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(69)] // IFM ピクセルのロード +69
vpdpbusd zmm6 , zmm10, zmm13
vpbroadcastd zmm10 , dword ptr [IFM_ADDR(70)] // IFM ピクセルのロード +70
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=0,k,n=1)] // 重みのロード [kh,kw,n] =
[2,0,+1]
vpbroadcastd zmm11 , dword ptr [IFM_ADDR(71)] // IFM ピクセルのロード +71
vpdpbusd zmm0 , zmm8 , zmm12
vpdpbusd zmm1 , zmm9 , zmm12
vpdpbusd zmm2 , zmm10, zmm12
vpdpbusd zmm3 , zmm11, zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=1,k,n=0)] // 重みのロード [kh,kw,n] =
[2,1,+0]
vpdpbusd zmm4 , zmm8 , zmm13
vpbroadcastd zmm8 , dword ptr [IFM_ADDR(72)] // IFM ピクセルのロード +72
vpdpbusd zmm5 , zmm9 , zmm13
vpdpbusd zmm6 , zmm10, zmm13
vpdpbusd zmm7 , zmm11, zmm13
vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=1,k,n=1)] // 重みのロード [kh,kw,n] =
[2,1,+1]
vpdpbusd zmm0 , zmm9 , zmm12
vpdpbusd zmm1 , zmm10, zmm12
vpdpbusd zmm2 , zmm11, zmm12
vpdpbusd zmm3 , zmm8 , zmm12
vmovups zmm12, zmmword ptr [WEIGHT_ADDR(kh=2,kw=2,k,n=0)] // 重みのロード [kh,kw,n] =
[2,2,+0]
vpdpbusd zmm4 , zmm9 , zmm13
vpbroadcastd zmm9 , dword ptr [IFM_ADDR(73)] // IFM ピクセルのロード +73
vpdpbusd zmm5 , zmm10, zmm13
vpdpbusd zmm6 , zmm11, zmm13
vpdpbusd zmm7 , zmm8 , zmm13
```

```
    vmovups zmm13, zmmword ptr [WEIGHT_ADDR(kh=2,kw=2,k,n=1)] // 重みのロード [kh,kw,n] =
    [2,2,+1]
    vpdpbusd zmm0 , zmm10, zmm12
    vpdpbusd zmm1 , zmm11, zmm12
    vpdpbusd zmm2 , zmm8 , zmm12
    vpdpbusd zmm3 , zmm9 , zmm12
    vpdpbusd zmm4 , zmm10, zmm13
    vpdpbusd zmm5 , zmm11, zmm13
    vpdpbusd zmm6 , zmm8 , zmm13
    vpdpbusd zmm7 , zmm9 , zmm13
}
// zmm0-zmm7 の結果で後処理を行います。
```

このコードでは、IFM 値に M_ACCUM (4) zmm レジスター zmm8-zmm11 を、重み付けに N_ACCUM (2) zmm レジスター zmm12-zmm13 を割り当てています。

最初にアキュムレーターをゼロにクリアする必要があります。次に K 次元全体 (#IFM=32) を横断する必要があります。それぞれの反復は 4 つの連続した IFM を操作します。畳み込みは、一連の 4 バイト IFM データのブロードキャスト、64 バイト重みデータのロード、および乗算と累積演算で構成されます。大規模 IFM データは異なる kh、kw 値でオーバーラップするため、IFM データを効率良く再利用でき、データロード数は大幅に減少します。

7.4.1.2 低 OFM カウントによる畳み込みレイヤー

チャネルの次元に沿ったベクトル化は、ベクトルレジスターを満たすのに十分なチャネル (入力と出力の両方) がある場合 (通常、分類トポロジーの場合) に上手く動作します。しかし、敵対的生成ネットワーク (GAN) のようなケースでは、最終結果はイメージです。つまり、最後の畳み込みレイヤーには 3 つのチャネルしかありません。このレイヤーでは、空間の次元に沿ってベクトル化する方法が理にかなっています。これには、異なるデータレイアウトが必要になります。大きな中間バッファーを使用しなくてもいいように、1 つの 4x16 ブロックに計算をオンザフライでレイアウトし直し、部分的な畳み込みを行ってブロックを破棄します (このメカニズムは 1x1 カーネルに限定され、新しいレイアウトに対応して重みが並べ替えられていると仮定します)。

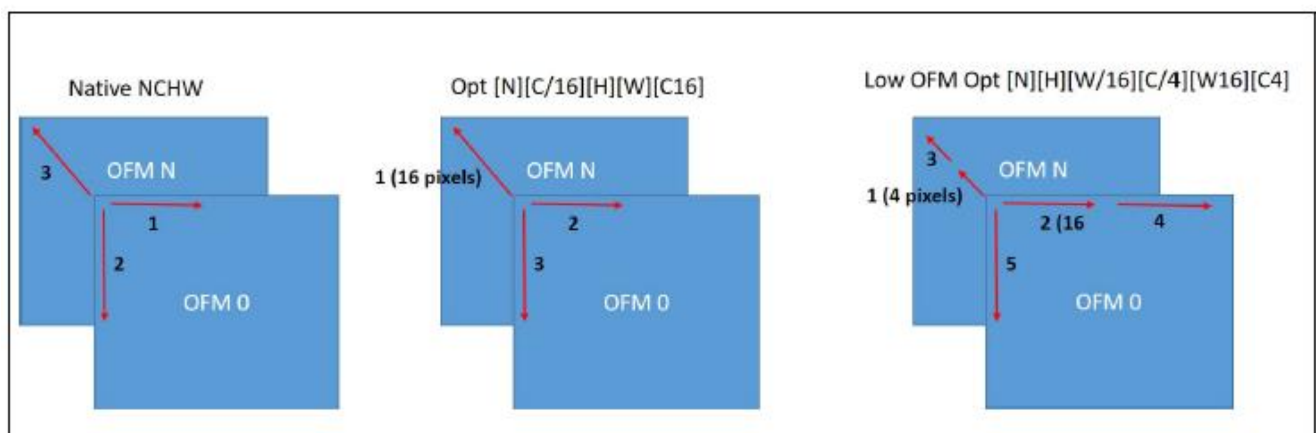

図 7-8. 標準、最適化済み、および低 OFM 最適化データレイアウト[1]

注意:

1. 低 OFM 最適化の 4x16 ブロックは、オンザフライで作成され一度だけ使用されます。

例 7-4. 低 OFM カウントによるレイヤーの畳み込み

低 OFM カウントによるレイヤーの畳み込み

```
# IFM_W % 16 == 0
# NUM_OFMS = 3
# NUM_IFMS = 64
# dqfs - ダウンコンバート向けの逆量子化係数の配列

int src_ifm_size = IFM_H * IFM_W * IFMBlock;
int ofm_size = IFM_W * IFM_H;

__m512i gather_indices = _mm512_setr_epi32(0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60);

        __m512 dqf_broadcast[NUM_OFMS];
        #pragma unroll(NUM_OFMS)
        for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
                dqf_broadcast[ofm] = _mm512_set1_ps(dqfs[ofm]);
        }

        for (int h = 0 ; h < IFM_H ; h++) {
            int src_line_offset = h * IFM_W * IFMBlock;
            int w = 0;
            int src_w_offset = src_line_offset;
            for ( ; w < IFM_W ; w += 16) {
            __m512i res_i32[NUM_OFMS] = { 0 };
            // オンザフライで再編成して 4x16 の OFM を畳み込み
                    for (int ifm = 0 ; ifm < NUM_IFMS ; ifm += 4) {
                    int src_block_offset = ifm & 0xf;
                    int src_ifm_index = ifm >> 4;
                    size_t src_ifm_offset = src_w_offset + src_ifm_index * src_ifm_size + src_block_offset;
                    __m512i ivec = _mm512_i32gather_epi32(gather_indices, input + src_ifm_offset, 4);
                    #pragma unroll(NUM_OFMS)
                        for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
                        int weight_offset = (ofm * NUM_IFMS + ifm) * 16;
                        __m512i wvec = _mm512_load_si512(weights_reorged + weight_offset);
                        res_i32[ofm] = _mm512_dpbusd_epi32(res_i32[ofm], ivec, wvec);
                        }
                    }
                    // ダウンコンバートして結果をネイティブレイアウトで格納
                    #pragma unroll(NUM_OFMS)
                    for (int ofm = 0 ; ofm < NUM_OFMS ; ofm++) {
                        __m512 res_f32 = _mm512_cvtepi32_ps(res_i32[ofm]);
                        res_f32 = _mm512_mul_ps(res_f32, dqf_broadcast[ofm]);
                        size_t output_offset = ofm * ofm_size + h * IFM_W + w;
                        _mm512_store_ps(output + output_offset, res_f32);
                    }
                    src_w_offset += 16 * IFMBlock;
            }
        }
}
```

7.4.2 畳み込みの後処理

畳み込みが行われると、レイヤーデータに対し多くの変換が実行されます。これらには、ReLU のような古典的な畳み

込み後の操作、プーリングや EltWise のような個別のレイヤーとして考慮される操作、および量子化/逆量子化操作が含まれます。メモリー階層のスラッシングを減らすため、畳み込み中にこれらのステップの実行を試みます (つまり、畳み込み操作に融合します)。量子化ステップを融合することで、4 倍の計算帯域幅が得られ、メモリー帯域幅は 4 分の 1 に減少するため試みる価値があります。

7.4.2.1 融合量子化/融合逆量子化

7.3.2.1 節はオフラインでの量子化の方法を示しています。多くの場合、現在のレイヤーの逆量子化と次のレイヤーの量子化は、畳み込みステップに融合することができます。次のコードは、 OFM ブロックの畳み込み後に開始される後処理の基本操作を示しています。前述のように、この手順は同じピクセルに属する 16 個の int8 OFM を操作します。この例では、単一の係数が現在のレイヤーの逆量子化を表し、次のレイヤーへの再量子化を表すように、逆量子化係数 (存在する場合はバイアスも) が準備されていると仮定します。一般に、次のレイヤーに渡すいくつかの乗算係数 (例えば、逆量子化、レイヤー定数乗算値、および量子化) を単一の係数として示すことによってオンラインでの計算数を減らします。さらに、元の OFM が負の値 (ReLU なし) である可能性がある場合、7.3.2.1 節の手順に従ってすべての値に 127 を加算します。

例 7-5. 基本 PostConv

基本 PostConv

```
// dest は同じピクセルに属する 16 個の OFM のベクトルを指します。
uint8_t* dest = (uint8_t*) (outputFeatureMaps) + offset;

// in は操作する int32 の 16 個のアキュムレーターです。
__m512 resf=_mm512_cvtepi32_ps(in); // float へ変換

// bias がある場合は bias を追加してから、シングルステップで逆量子化と再量子化を行います
if(bias) {
    resf = _mm512_fmadd_ps(resf,
                _mm512_load_ps(dqfs+OFMChannelOffset),
                _mm512_load_ps((__m512*) (bias + OFMChannelOffset)));
} else {
    resf= _mm512_mul_ps(resf,
                _mm512_load_ps(dqfs+ OFMChannelOffset));
}

#if RELU
    resf = _mm512_max_ps(resf, broadcast_zero);
#endif

// この時点では uint8 範囲内
__m512i res = _mm512_cvt_roundps_epi32
(resf,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
__m128i res8;

#if ELTWISE
    /* 融合 Eltwise 操作 */
#else
# if !RELU
    res = _mm_add_epi8(res,_mm_set1_epi8(-128)); //hack to add 128
# endif
    res8 = _mm512_cvtusepi32_epi8(res);
#endif // ELTWISE

#if POOLING
    /* 融合プーリング操作 */
```

```
#endif
_mm_store_si128((__m128i*) dest, res8);
```

7.4.2.2 ReLu

ReLu は、畳み込みと融合した際に無視できるオーバーヘッドが生じる、ゼロレジスターを有する最大値として実装されます。

7.4.2.3 EltWise

要素ごとの操作は通常、最終結果が保存される直前にアキュムレーターで直接操作されるため、畳み込みステップに容易に融合できます。ただし、異なる入力レイヤーの量子化係数は通常同じではないため、入力を f32 に逆量子化し操作した後に再度量子化する必要があります (ベクトル化されたコードでこのステップの最適化を示します)。

次に、入力と出力のデータ型を想定した eltwise 操作の例を示します。すべての例で、畳み込み操作からのデータは VPDPBUSD 命令が返す INT32 データであり、量子化された出力は uint8 であるため、量子化されない出力は負の値になる可能性があります。負の値と uint8 への量子化をどのように行うかは、7.3.2 節の「量子化」を参照してください。

次の最適化されたコードは、「dest」が出力の同じピクセルに属する 16 個の OFM ベクトルを指すと仮定する、いくつかの eltwise の使用例に対する実装を示します。原則として、次の式のように、eltwise データと畳み込みデータを逆量子化し、加算してから逆量子化します。

$$result = \left(eltwise_{f32} \times eltwiseDQfactor + conv_{i32} \times convDQfactor\right) \times NextQfactor$$

ただし、係数 ([ ] 内の演算) はオフラインで前処理できるため、オンラインでの乗算は 2 回だけになります。

$$result = \left(eltwise_{f32} + conv_{i32}\left[\frac{convDQfactor}{eltwiseDQfactor}\right]\right) \times ([NextQfactor \times eltwiseDQfactor])$$

例 7-6. Uint8 残差入力

Uint8 残差入力

```
__m128i eltwise_u8 = _mm_load_si128((const __m128i*) (eltwise_data + ew_offset));
__m512i eltwise_i32 = _mm512_cvtepu8_epi32(eltwise_u8);
if signed_residual {
eltwise_i32 = _mm512_sub_epi32(eltwise_i32, broadcast_128);
}

__m512 eltwise_f32 = _mm512_cvtepi32_ps(eltwise_i32);
resf = _mm512_add_ps( eltwise_f32,resf); /* 畳み込みの結果と加算 */

/* 一度の操作で逆量子化して次のレイヤーに再量子化 */
resf = _mm512_mul_ps(resf,broadcast_fused_eltwise_out_qfactor);
if (relu)
    resf = _mm512_max_ps(resf, broadcast_zero);
    __m512i data_i32 = _mm512_cvt_roundps_epi32(resf,
                                    (_MM_FROUND_TO_NEAREST_INT|
                                    _MM_FROUND_NO_EXC));
res8= _mm512_cvtusepi32_epi8(data_i32);
```

```
if (!relu) {
    res8 = _mm_add_epi8(res8,_mm_set1_epi8(-128)); // 128 加算に戻る
}
```

7.4.2.4 プーリング

プーリングレイヤーを畳み込みステップに融合するのは困難かもしれませんが、場合によっては容易なこともあります。例えば、Inception ResNet 50 の最後の畳み込み平均プーリングは、各 OFM チャネルの 8x8 ピクセルすべてを単一の値に平均化するため OFM ごとに単一の値を出力します。このような操作は、すべてのピクセルで同じように動作するため、容易に融合できます。

例 7-7. 8x8 レイヤーのストライド 1 の 8x8 平均プーリング

8x8 レイヤーのストライド 1 の 8x8 平均プーリング

```
__m512 pool_factor = _mm512_set1_ps((float)1.0/64);

// resf は基本 PostConv コード例で計算された 16 個の float 値です

resf = _mm512_mul_ps(resf,pool_factor ); // 64 で除算

// pool_offset は、現在の OFM (OFMltr) にのみ依存します。
int pool_offset = (BlockOffsetOFM + OFMltr);
float *pool_dest = (float *) (outputFeatureMaps) + pool_offset;
__m512 prev_val = _mm512_load_ps((const __m512 *) (pool_dest));
__m512 res_tmp_ps = _mm512_add_ps(resf, prev_val);
_mm512_store_ps((__m512 *) pool_dest, res_tmp_ps);
```

次の融合されていないベクトル化コードにより、最大および平均プーリングを実行できます。また、プーリングステップでは入力量子化範囲を出力量子化範囲に調整できます。これは、通常 NOP として実装され連結レイヤーの前に必要です。すべての連結レイヤーの出力量子化範囲は同じでなければなりません。

例 7-8. 融合されていないベクトル化プーリング

融合されていないベクトル化プーリング

```
// 次の IFM セットに移動する前に、16 個の IFM を同時にプーリング
for (int ifm = 0; ifm < no_ifm; ifm+=16) {
    // 入力と出力で、プーリングするブロック位置を検出
    int block_idx = (ifm >> 4) ;
    size_t block_offset = (spatial_size_in * block_idx) << 4;
    size_t block_offset_out = (spatial_size_out * block_idx) << 4;

    for (int y = -pad_h_; y < top_y + pad_h_; y++) {
        int y_offset_out = (top_x * 16 * (y + pad_h_));
        for (int x = -pad_w_; x < top_x + pad_w_; x++) {
        __m256i res_pixel = _mm256_set1_epi16(0); // u_int である必要がありますが、0 は 0 です
            for (int i = 0; i < kernel_w; i++) {
            int y_offset_in = (bottom_x * (i + (y * stride)))<<4;
            for (int j = 0; j < kernel_h; j++) {
                int x_offset = (j + (x * stride)) << 4;
                // pad 内にあるピクセルをスキップ
                if (pad && ((j + (x * stride)) < 0 || (i + (y * stride)) < 0))
                continue;
                // ピクセルデータをロード
                __m128i data_px = _mm_load_si128((const __m128i *) (bottom_data+ ifm_image_offset +
```

```
                    block_offset + y_ofset_in + x_ofsset ));
                    // 平均化して u16 に変換
                    __m256i data_px_u16 = _mm256_cvtepu8_epi16(data_px);
                    if (MAX) {
                        res_pixel = _mm256_max_epu16(res_pixel,data_px_u16);
                    } else if (AVERAGE) {
                        res_pixel = _mm256_adds_epu16(res_pixel,data_px_u16);
                    }
                } // kernel_h
            } // kernl_w

            // 入力データの処理は完了ですが、調整が必要な場合があります。
            int x_offset_out = (x + pad_w_) << 4;
            if (SHOOUL_ADJUST_QUANTIZATION_RANGE) { // 通常、NOP 連結の前
                float factor = layer_param().quantization_param().bottom_range()
                / this->layer_param().quantization_param().top_range();

                __m512 broadcast_factor = _mm512_set1_ps(factor);

                __m512i res_pixel_i32 = _mm512_cvtepu16_epi32(res_pixel);
                __m512 res_pixel_f32 = _mm512_cvtepi32_ps(res_pixel_i32);
                res_pixel_f32 = _mm512_mul_ps(res_pixel_f32,broadcast_factor);

                __m512i data_i32 = _mm512_cvt_roundps_epi32 (res_pixel_f32
                                ,_MM_FROUND_TO_NEAREST_INT|_MM_FROUND_NO_EXC);
                res_pixel= _mm512_cvtusepi32_epi16(data_i32);
            }
            if (AVERAGE) {
                uint8_t kernel_size_u8 = kernel_h_ * kernel_w_;
                __m256i broadcast_kernel_size = _mm256_set1_epi16(kernel_size_u8);

                res_pixel=_mm256_div_epu16(res_pixel,broadcast_kernel_size);
            }
            // 最後のオフセットを計算して保存
            uint8_t * total_offset =
                top_data + output_image_offset + layer_offset + block_offset_out +
                        y_ofsset_out + x_ofsset_out ;//+ vect_idx;
            _mm_store_si128((__m128i*) total_offset, _mm256_cvtusepi16_epi8(res_pixel));
            }
        }
}
```

7.4.2.5 ピクセル・シャッフラー

SRGAN トポロジーには、次のような入力を変形するレイヤーが含まれます。

1. フィーチャー・マップの幅と高さは 2 倍になります。
2. 出力フィーチャー・マップ数は 4 で割られます。

出力フィーチャー・マップのそれぞれの 2x2 クワッドは、(c mod K/4) の条件を満たす入力の同じ空間次元からの 4 つのピクセルで占められます。ここで、c は入力チャネルであり、K は入力フィーチャー・マップ数です (参考文献 SRGAN)。

例 7-9. ピクセル・シャッフラーの Caffe スカラーコード

ピクセル・シャッフラーの Caffe スカラーコード

```
typedef pstype int8_t // すべてのデータ型で動作.
vector<int> bottom_shape = bottom[0]->shape();
const int bn = bottom_shape[0];
const int bc = bottom_shape[1];
const int bh = bottom_shape[2];
const int bw = bottom_shape[3];
vector<int> top_shape = top[0]->shape();
const int tc = top_shape[1];
const int th = top_shape[2];
const int tw = top_shape[3];

int test_r1 = bc/tc;
const int r = th / bh;
int test_r2 = r * r;
const pstype* bottom_data = (const pstype*)bottom[0]->cpu_data();
pstype* top_data = (pstype*)top[0]->mutable_cpu_data();
pstype* cur_channel;
int bottom_ch_size = bw * bh;
for(int n = 0; n < bn; n++){
    for(int c = 0; c < tc; c++){
        cur_channel = top_data + n*(tc*th*tw)+ c*(th*tw);
        for(int h = 0; h < bh; h++){
            for(int w = 0; w < bw; w++){
                int bottom_offset = h * bw + w;
                int bottom_index = c * bottom_ch_size + bottom_offset;
                int top_index = h * r * tw + w * r;
                cur_channel[top_index] = bottom_data[bottom_index]; // 左上
                bottom_index = (c + tc) * bottom_ch_size + bottom_offset;
                top_index = h * r * tw + w * r + 1;
                cur_channel[top_index] = bottom_data[bottom_index]; // 右上
                bottom_index = (c + 2 * tc) * bottom_ch_size + bottom_offset;
                top_index = (h * r + 1) * tw + w * r;
                cur_channel[top_index] = bottom_data[bottom_index]; // 左下
                bottom_index = (c + 3 * tc) * bottom_ch_size + bottom_offset;
                top_index = (h *r + 1) * tw + w * r + 1;
                cur_channel[top_index] = bottom_data[bottom_index]; // 右下
                }
            }
        }
    }
}
```

ベクトル化された directConv のメモリーレイアウトにより、ピクセル・シャッフラー・レイヤーを畳み込みに融合するのは容易です。唯一の変更点は、畳み込みの結果を出力の正しい位置に保存することです。

例 7-10. 融合ピクセル・シャッフラーの出力オフセット計算

融合ピクセル・シャッフラーの出力オフセット計算

```
// base_ofm - 出力ターゲットの位置 (base_ofm % 16 == 0)
// SubTileX - クワッドの X 位置 (0 または 1)
// SubTileY - クワッドの Y 位置 (0 または 1)
// ConvOutputX - 畳み込み出力の X 位置
// ConvOutputX - 畳み込み出力の Y 位置
```

```
int PostPSNoOutMs =(NoOutFMs / 4)
int PostPSOptOfmIndex = (base_ofm % PostPSNoOutMs) / 16;
int QuarterIndex = base_ofm / PostPSNoOutMs;
int SubTileX = QuarterIndex & 0x1;
int SubTileY = (QuarterIndex & 0x2) >> 1;
int PostPSX = ConvOutputX * 2 + SubTileX;
int PostPSY = ConvOutPutY * 2 + SubTileY;
size_t offset = (OFM_H * OFM_W * PostPSOptOfmIndex + PostPSY * OFM_W + PostPSX) * 16;
```

7.5 LSTM ネットワーク

LSTM (Long short-term memory) ユニットは、音声やテキスト翻訳などのタスク向けに RNN (Recurrent Neural Network) を作成するために使用されます。LSTM セルの基本計算は、CNN などの直接畳み込みではなく行列乗算 (GEMM) です。

7.5.1 LSTM 組込み融合

LSTM セルは、入力データを入力カーネルで乗算することから開始します。ここで、入力データ (埋め込みとも言う) はすべての辞書ワードに対して 512 要素であり、入力カーネルはオフライン分かっており変化することがありません。最後に、それぞれの埋め込みベクトルに同じ行列を掛けます。各ベクトルにオフラインでカーネルを乗算して保存し、実行時に GEMM の結果をワード・インデックスで検索してセルのアキュムレーター領域にコピーすることが推奨されます[1]。この最適化により、およそ 20% のパフォーマンス・ブーストがもたらされます。

7.5.2 GEMM 後処理融合

既存の LSTM セルの多くの異形は、活性化関数としてシグモイドおよび双曲線正弦などの超越操作を含みます。

$$sigmoid(x) = \frac{1}{e^{-x} + 1}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

活性化を完全精度スカラーまたは SVML ベースのベクトル化コードとして実装すると遅くなることがあります。代替方法としては、高いパフォーマンスをもたらす近似を使用することです。超越関数を近似する方法の 1 つは、区分的多項式近似を使用することです。

例 7-11. ミニマックス多項式によるシグモイド近似

ミニマックス多項式によるシグモイド近似

```
// シグモイド近似のための 2 次ミニマックス多項式の係数
__declspec( align(64) ) const float sigmoid_poly2_coeffs[3][16];

inline void sigmoid_poly_2(const __m512& arg, __m512& func)
{
    // 多項式係数をレジスターにロード (一度の操作)
    const __m512 sigmoid_coeff0 = _mm512_load_ps( sigmoid_poly2_coeffs[0] );
```

[1] Jacob Devlin、Rabih Zbib、Zhongqiang Huang、Thomas Lamar、Richard M Schwartz、および John Makhoul。2014。統計的機械翻訳向けの高速で堅牢なニューラル・ネットワーク結合モデル。ACL (1)。Citeseer、ページ 1370-1380。

```
    const __m512 sigmoid_coeff1 = _mm512_load_ps( sigmoid_poly2_coeffs[1] );
    const __m512 sigmoid_coeff2 = _mm512_load_ps( sigmoid_poly2_coeffs[2] );

    // 引数の符号を抽出
    const __m512 ps_sign_filter = _mm512_castsi512_ps(_mm512_set1_epi32( 0x7FFFFFFF ));

    __mmask16 signs = _mm512_movepi32_mask(_mm512_castps_si512(arg));
    __m512 abs_arg = _mm512_and_ps(arg, ps_sign_filter);

    // 引数の指数と MSB から近似間隔を計算し、
    // 間隔数を 16 に制限
    const __m512i lut_low = _mm512_set1_epi32( 246 );
    const __m512i lut_high = _mm512_set1_epi32( 261 );

    __m512i indices = _mm512_srli_epi32(_mm512_castps_si512(abs_arg), 22);
    indices = _mm512_max_epi32(indices, lut_low);
    indices = _mm512_min_epi32(indices, lut_high);

    /*
    * 近似
    */
    __m512 func_p0 = _mm512_permutexvar_ps(indices, sigmoid_coeff0);
    __m512 func_p1 = _mm512_permutexvar_ps(indices, sigmoid_coeff1);
    __m512 func_p2 = _mm512_permutexvar_ps(indices, sigmoid_coeff2);
    func = _mm512_fmadd_ps(abs_arg, func_p2, func_p1);
    func = _mm512_fmadd_ps(abs_arg, func, func_p0);

    // 引数の符号を考慮
    func = _mm512_mask_sub_ps(func, signs, ps_ones, func);
}
```

ミニマックス多項式近似は、レイヤーごとで最も高い精度をもたらしますが、トポロジー (特に NMT) によってはエンドツーエンドの精度が損なわれる可能性があります。その場合、別の方法を検討すべきです。次の近似において、

$$e^x = 2^{x \log_2 e} = 2^{n+y}$$

ここで、

$$n = \mathrm{round}(x \log_2 e)$$

$$y = x \log_2 e - n$$

2^n は scalef 命令で計算されます

$$2^n = \mathrm{scalef}(x \log_2 e)$$

2^y は次数 2 のテイラー多項式で近似できます。

例 7-12. scalef を使用したシグモイド近似

scalef を使用したシグモイド近似

```
static const __m512 ps_ones = _mm512_set1_ps( 1.0 );
const __m512 half = _mm512_set1_ps( 0.5f );
const __m512 sixteen = _mm512_set1_ps( 16.0 );
const __m512 log2_e = _mm512_set1_ps( 1.442695f );
const __m512 minus_log2_e = _mm512_set1_ps( -1.442695f );
const __m512 twice_log2_e = _mm512_set1_ps(1.442695f*2 );
const __m512 ln_2 = _mm512_set1_ps( 0.6931471f );
const __m512 ln2sq_over_2 = _mm512_set1_ps( 0.240226507f );
const __m512 ln2__ln2sq_over_2 = _mm512_set1_ps( 0.452920674f );
const __m512 one__ln2sq_over_8 = _mm512_set1_ps( 0.713483036f );

inline void sigmoid_scalef(const __m512& arg, __m512& func)
{
    __m512 x = _mm512_fmadd_ps(arg, minus_log2_e, half);
    __m512 y = _mm512_reduce_ps(x, 1);
    __m512 _2y = _mm512_fmadd_ps(_mm512_fmadd_ps(y, ln2sq_over_2, ln2__ln2sq_over_2), y,
                                                                          one__ln2sq_over_8);
    __m512 exp = _mm512_scalef_ps(_2y, x);
    func = _mm512_rcp14_ps(_mm512_add_ps(exp, ps_ones));
}
```

7.5.3 動的バッチサイズ

- Encoder:
  - Sort the sentences by length
  - Execute LSTM with dynamic batch size

4 3 1 Dynamic batch size

- Decoder:
  - In general, target length correlates to source length
  - Reduce the LSTM batch size when a sentence has ended
    - Currently done only on the last active sentence in the batch (Finished in opt #13)

Wasted decoder executions:

Batch Size	1	2	4	8	16
Dynamic	0.0%	0.2%	1.3%	1.9%	4.3%
Static (TF)	0.0%	32.2%	66.3%	92.3%	115%

図 7-9. 動的バッチサイズ[1]

注意:

1. NMT は、各反復の計算をアクティブな文の数に合わせることで大幅に向上します。

異なる RNN オブジェクト (例えば文) には、大きく異なる計算量が必要です (例えば、短い文と長い文向けに)。複数のオブジェクトを一括してバッチ処理する場合、このことを考慮して無駄な計算を避けることが重要です。NMT の例では (図 7-9 を参照)、文が長い順に並べられていることを確認すれば、各反復を実際にアクティブな文の数に合わせるのは容易です。

7.5.4 NMT の例: 上位 K を取得するビーム検索デコーダー

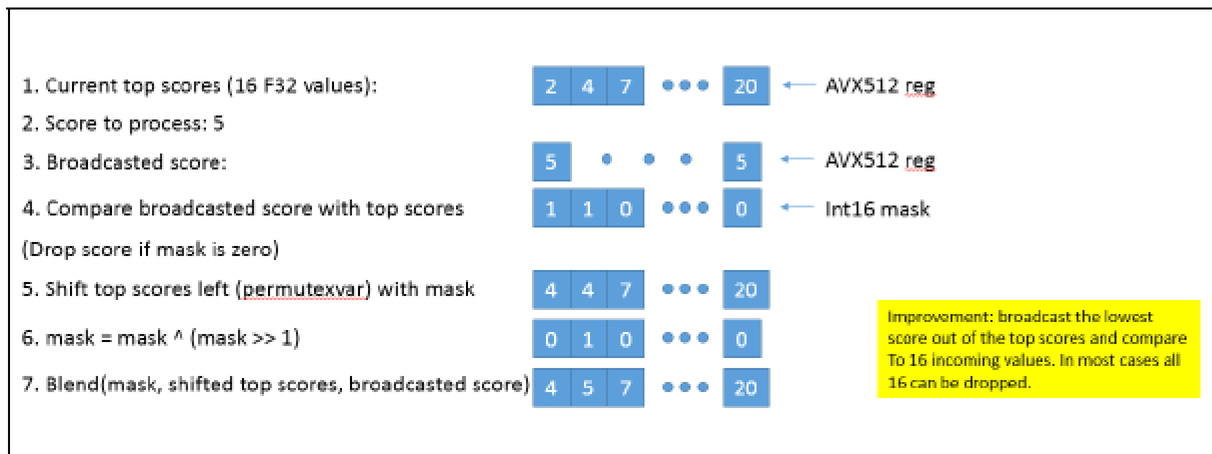

図 7-10. 入力から上位 16 の値を検出

https://papers.nips.cc/paper/5346-sequence-tosequence-learning-with-neural-networks.pdf (英語) と https://github.com/tensorflow/nmt (英語) に示すように、ニューラル機械翻訳では、BEAM_WIDTH*VOCAB_SIZE 値のうち、上位 BEAM_WIDTH 算出スコアの検索にかなり時間がかかります。このステップを最適化するため次のアルゴリズムを使用することを推奨します。このプロセスの要点は、新しい値を一度の操作ですべての上位値と比較できることです (図 7-10 の 4 行目を参照)。op によって返されるマスクは、1 のシーケンスとそれに続くゼロ (1* 0 *) で構成される必要があるため、上位スコアをソートしたままにしておくことに注意してください。

例 7-13. 上位 K を検出する疑似コード

上位 K を検出する疑似コード

```
// ZMM0 - ベストスコア、-MAX_FLOAT に初期化
// ZMM1 - ベストスコアのインデックス
// ZMM4 - 現在のスコアのインデックス

index = 0
pxor ZMM4
while index < MAX_DATA
    vbroadcastss ZMM2, array[index]
    VPCMPPS K1,ZMM0,ZMM2, _CMP_LT_OQ
    KTESTW K1,K1
    JZ … // K1 == 0 の場合、新しいスコアを配置します
    // K1!=0 の場合
    VPERMPS ZMM0(k1),ZMM0
    VPERMPS ZMM1(k1),ZMM0
    KSHIFT k2,k1,1
    KXOR k3,k2,k1
    VPBLENDMPS k3, ZMM0,ZMM2
    VPBLENDMD k3, ZMM1,ZMM4
    VPADD ZMM4, 1
    add index, 1
```

† 開発コード名