インテル® プロセッサーにおける Transformer モデル推論の最適化

oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (oneDNN) の利点を生かした融合操作

Xiaoming（Jason）Cui インテル コーポレーション AI フレームワーク・エンジニア
Ashraf Bhuiyan インテル コーポレーション AI ソフトウェア・エンジニアリング・マネージャー

Transformer モデルは、自然言語処理（NLP）で最も人気のあるモデルの 1 つです（**図 1**）。2017 年に Google によって公開（英語）されて以来、ほかの多くの NLP モデルで採用されており、非 NLP モデルにも拡張されています。言語翻訳は Transformer の一般的な用途であり、その優れた精度と並列処理により、従来の LSTM モデルに大きく取って代わるものです。BERT（英語）モデルは Transformer に基づいているため、Transformer を最適化すると、多くの NLP モデルおよび非 NLP モデルを改善できます。

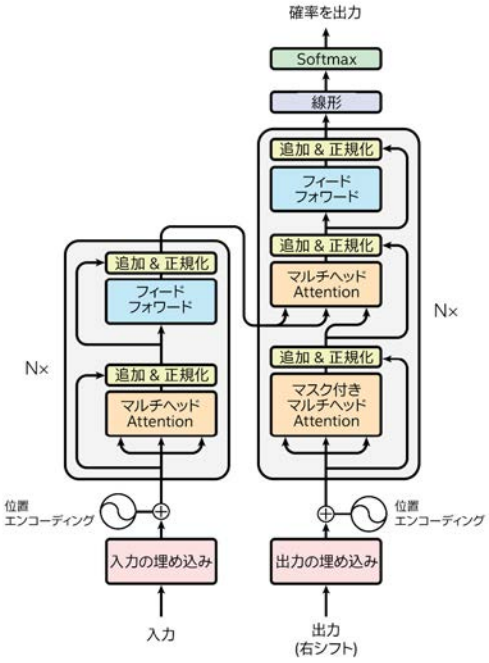

図 1. Transformer モデル・アーキテクチャー（出典：Vaswani ほか、Attention Is All You Need（英語））

インテルは、MLCommons v0.5（英語）の Transformer のトレーニングおよび評価コードを始め、インテル® Xeon® スケーラブル・プロセッサーでの推論向けにモデルを最適化しています。これらの最適化により、推論のスループットとレイテンシーのどちらもパフォーマンスが大幅に向上しました。最適化されたモデルは、インテル® アーキテクチャー向け Model Zoo（英語）に追加されています。

モデルのトレーニング

推論の最適化を試みる前に、トレーニング済みモデルを入手する必要があります。MLPerf* Transformer モデル（v0.5）を使用し、MLPerf* が推奨する精度でトレーニングしました。このモデルにはチェックポイントがあり、必要に応じて、さらにトレーニングすることで精度を少し向上できます。トレーニング済みのチェックポイントから推論を行うこともできますが、チェックポイントには推論に必要のないトレーニング・ノードが含まれているためサイズが大きく、通常、モデルをデプロイする方法としては最適ではありません。また、チェックポイントから推論を実行すると、重みをロードするオーバーヘッドが発生します。

推論パフォーマンスを最適化するため、トレーニング済みチェックポイントを使用する代わりに、トレーニング済みモデルのチェックポイントを、推論グラフとモデルの重みだけを含むグラフに固定しました。その結果、MLPerf* Transformer モデルのサイズは、チェックポイントを使用した場合の 1GB 以上から約 800MB に減りました。固定したグラフは、モデルを int8 に量子化してさらにパフォーマンスを向上する場合にも必要になります。グラフを固定するスクリプトは、インテル® アーキテクチャー向け Model Zoo にあります。

グラフを固定する過程で、定数の畳み込み、単位元ノードの削除など、いくつかの最適化が適用されます。グラフを固定したら、既知の入力ノードと出力ノードを持つ protobuf（`.pb`）ファイルを取得します。固定したグラフから推論を実行するのは容易です。入力データをモデルの入力ノードに供給するだけです。実行後、モデルの出力ノードから結果を得ることができます。後でモデルをさらに最適化するアイデアを思いついた場合、モデルの重みが変わらない限り、再度トレーニングすることなく、最適化されたコードでモデルを再固定できます。

Transformer モデルには、エンコーダーとデコーダーの 2 つの主要なモジュールがあります。各モジュールには、フィード・フォワード・ネットワーク（FFN）、エンコーダー・デコーダーの Attention、ビーム検索、融合操作、埋め込み、Self-Attention など、複数のレイヤーがあります。ここでは、これらのレイヤーのいくつかに適用した最適化を紹介します。

FFN レイヤーのパディング / アンパディングの最適化

ここで最も重要な最適化は、FFN レイヤーからパディング / アンパディング・アルゴリズムを削除することです。このレイヤーへの入力は、形状 `[batch_size, input_length, Hidden_size]` の 3 次元パディングテンソルです。**図 2** に示すように、このモデルには 2 番目の次元にパディングがあり、空白ブロックはパディング値ゼロを表します。モデルは、テンソルを密レイヤーに送る前にパディングを検出して削除し、**図 3** に示すように、形状 `[batch_size, number_tokens*Hidden_size]` の 2 次元テンソルに再整形します。パディングの値はゼロのままなので、計算する必要はありません。しかし、密レイヤーの後、出力テンソルにパディングを追加して、元の形状に戻す必要があります（次のレイヤーの入力テンソルになるため）。

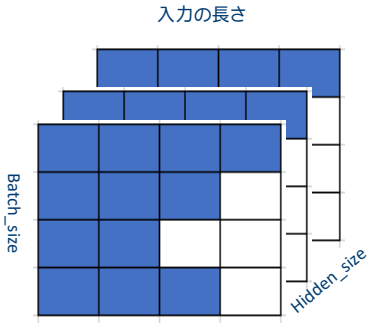

図 2. FFN 入力テンソルのデータレイアウト

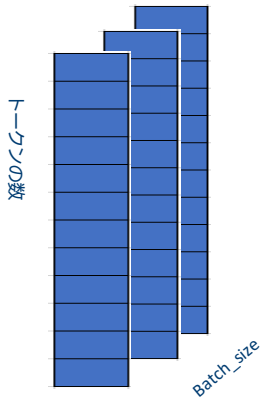

図 3. FFN 入力テンソルの最適化されたデータレイアウト

パディング / アンパディング・アルゴリズムは、より多くのメモリー操作を使用することで、計算量を減らします。計算がメモリー操作より遅い場合、パフォーマンスは向上します。しかし、最近のインテル® Xeon® スケーラブル・プロセッサーでは、このアルゴリズムの計算はメモリー操作よりも高速です。したがって、FFN レイヤーからパディング / アンパディングを削除することで、パフォーマンスが向上します。

エンコーダー・デコーダーの Attention キャッシュの最適化

Transformer モデルでは、入力テンソルはエンコーダー・モジュールを通過して、エンコーダー・メモリー・テンソルを出力します。これは、デコーダーモジュールでエンコーダー・デコーダーの Attention の計算に使用されます（**図 4**）。

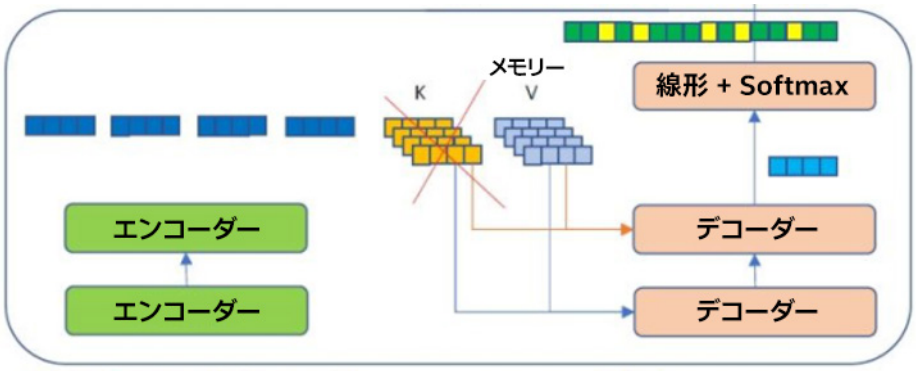

図 4. エンコーダー・デコーダーの Attention

Attention は、Transformer モデルの最も重要な部分です。次の式で計算されます。ここで、Q、K、および V は密レイヤーの後のテンソルです。

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Q は、デコーダーの入力を使用した密レイヤーの結果です。K と V は、エンコーダー・メモリー・テンソルの入力を使用した密レイヤーの結果です。デコーダーの反復のすべてのステップで、デコーダーの入力テンソルの値は変化しますが、エンコーダーからのメモリーテンソルの値は同じです。つまり、K テンソルと V テンソルはデコーダーのすべてのステップで同じですが、元の Transformer モデルはこれらのテンソルを再計算します。

この冗長な計算を排除する必要があることは明らかです。そのため、K と V をキャッシュするメカニズムを追加しました。エンコーダー・デコーダーの Attention の開始時にのみ K と V テンソルを計算し、残りのデコーダーの反復で再利用するため値をキャッシュします。

ビーム検索キャッシュの最適化

ビーム検索も時間のかかる Transformer モジュールです。プロファイル結果は、このモジュールの最大のボトルネックが、特定の条件を満たすテンソル値を収集する `Gather_ND` 操作であることを示しています（**図 5**）。メモリーアクセスがランダムであるため、この操作のベクトル化と並列化は困難です。

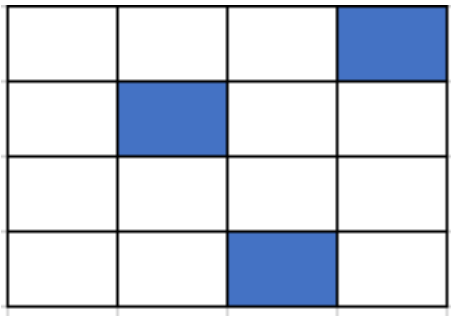

図 5. Gather_ND パターン

しかし、モデルで収集する必要がある値は、多次元テンソルのベクトルであることに気付きました。TensorFlow* フレームワークには、テンソルからベクトルを収集する `Gather_V2` 操作が含まれています（**図 6**）。この操作は容易にベクトル化および並列化でき、結果は同じになります。ビーム検索アルゴリズムで `Gather_ND` を `Gather_V2` に置き換えると、パフォーマンスが向上します。

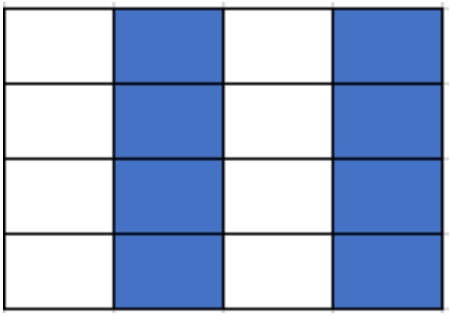

図 6. Gather_V2 パターン

融合操作の最適化

融合操作は、ディープラーニング・モデルにおける一般的な最適化です。Transformer では、`MatMul`、`Reshape`、`BiasAdd`、および要素ごとの操作が FFN レイヤーのホットスポットであることに気付きました。インテルによる TensorFlow*（英語）の最適化は、これらの操作の特定のパターンを新しい操作に融合します。しかし、Transformer モデルの操作パターン（**図 7**）は、必要な融合パターン（**図 8**）と一致しません。モデルのソースコードを調査した結果、エラーを発生させたり、計算やメモリー使用量を増加させることなく、`Reshape` 操作を `MatMul` の前に移動できることが分かりました。これは、ソースコードを数行変更するだけで済みます。

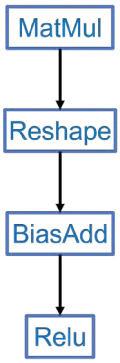

図 7. 最適化前の操作

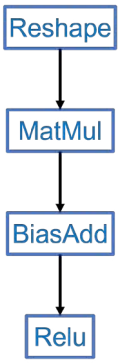

図 8. 変更後のパターン

再生成されたモデルで推論を実行すると、TensorFlow* 向けインテル® オプティマイゼーション・ランタイムは、高度に最適化された oneAPI ディープ・ニューラル・ネットワーク・ライブラリー（oneDNN）（英語）を呼び出す **MklFusedMatMul** という名前の新しい操作に操作を自動的に融合します（**図 9**）。これにより、メモリー操作が減り、元のモデルよりもはるかに優れたパフォーマンスを実現します。

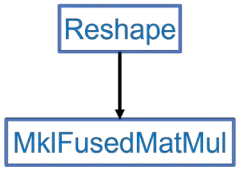

図 9. 融合および最適化された最終パターン

Transformer の Attention レイヤーにおける推論パフォーマンスのホットスポットは、`Mul`、`BatchMatMul`、`AddV2`、および `Softmax` であることが分かりました（**図 10**）。`Mul`、`AddV2`、および `Softmax` は、要素単位のメモリー依存の操作です。

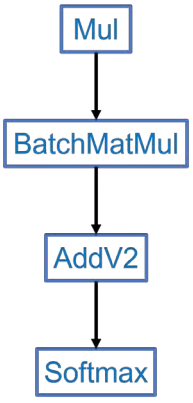

図 10. オリジナルの操作シーケンス

TensorFlow* 向けのインテルによる最適化は、シーケンシャルな `BatchMatMul`、`Mul`、および `AddV2` 操作を融合します。ここでも、パターンを再配置することで（**図 11**）、最適化が可能になります（**図 12**）。これらの融合は、効率良く計算するだけでなく、メモリー・トラフィックを軽減するため、モデルにとって重要です。メモリー・トラフィックの軽減は、メモリー依存の操作で特に重要です。

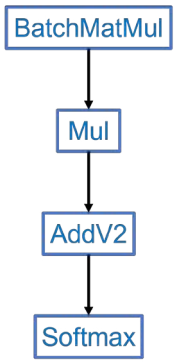

図 11. 変更後の操作シーケンス

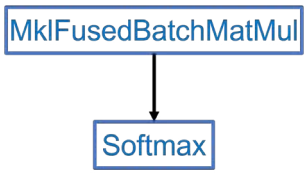

図 12. 融合および最適化された最終パターン

次の最適化は、Transformer モデルでレイヤーの正規化を融合します。レイヤーの正規化は、エンコーダーおよびデコーダーモジュールのすべてのレイヤーで使用され、メモリー移動または要素単位の操作のいずれかを行います。これらの操作は、メモリー依存のパフォーマンス・ボトルネックであるため、メモリー・トラフィックを軽減するレイヤー正規化の融合を実装しました（**図 13** と**図 14**）。

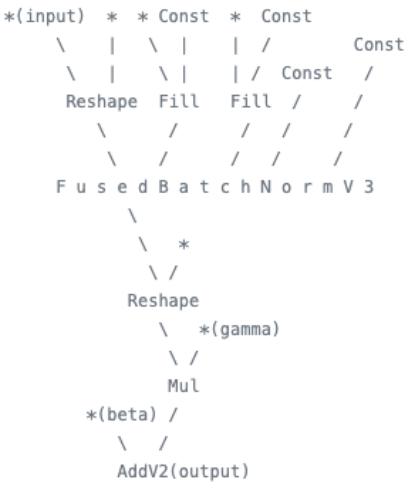

図 13. 最適化前の操作

図 14: 最適化後のフレームワーク

まとめ

Transformer は強力ですが、複雑なディープラーニング・モデルです。MLPerf* リポジトリーの Transformer 推論モデルは、oneDNN ライブラリーを利用するインテルの最適化バージョンほどハイパフォーマンスではありません。ここで紹介した最適化は、ベースラインの推論パフォーマンスを取得したデフォルトの FP32 モデルに基づいています。これらの最適化により、モデルの精度を損なうことなく、パフォーマンスが大幅に向上しました。サポートしているハードウェアでは、bfloat16 および int8 に変換することで、モデルをさらに最適化できます。例えば、第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、インテル® アドバンスト・マトリクス・エクステンション（インテル® AMX）命令で、bfloat16（BF16）（英語）と 8 ビット量子化（int8）の両方をサポートしています。

- **TensorFlow* と Bfloat16 により第 3 世代インテル® Xeon® スケーラブル・プロセッサー上で AI パフォーマンスを高速化（英語）**
- **第 4 世代インテル® Xeon® スケーラブル・プロセッサー上で TensorFlow* を最適化（英語）**

これらの低精度のサポートを使用するように変換されたモデルでは、さらに優れたパフォーマンスを実現できます。詳細は、インテル® アーキテクチャー向け Model Zoo（英語）を参照してください。この記事で使用したすべてのトレーニング済みチェックポイント、グラフの固定、およびソースコードは、皆さんが参照したり、使用できるようにリポジトリーで公開しています。