ホワイトペーパー

MATLABからのCUDAコード生成：GPUによる画像処理・ディープラーニングアルゴリズムの高速化

MathWorks®

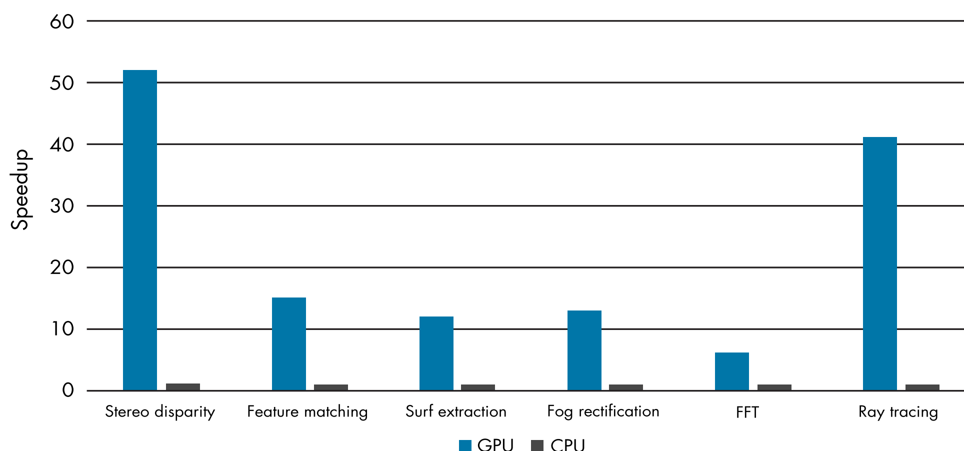
MATLABを利用したGPUコード生成について

膨大なデータの並列計算を得意とするGPUは、ディープラーニングと関連するアプリケーションの普及に大きく貢献しています。画像処理やディープラーニングでは畳み込み等の計算コストの高い演算が必要とされますが、それらのアルゴリズムには並列実行可能な演算も多く、GPUコンピューティングによって高速化を実現できます。

GPUによる高速処理を実現するためには、ヘテロジニアス・プログラミングが必要となります。並列処理によって高速化が期待できる部分については、GPU上の数百～数千のコアで処理を実行するためにカーネルにマッピングし、一方で逐次処理に該当する部分についてはCPU上で実行します。

GPU Coder™ はMATLAB®のコードからCUDA®コードを自動生成することができ、NVIDIA® GPU上で処理を高速化できます。GPU Coderには様々な最適化エンジンが搭載されており、MATLABコードに対する並列性の解析や、GPU↔CPU間のデータ転送を最小化するためのデータ依存性解析等が行われ、結果として最適化されたCUDAコードが生成されます。GPU↔CPU間のデータ転送の部分はGPUコンピューティングにおけるパフォーマンスのボトルネックになる場合も多く、データ転送を最小化するための依存性解析は非常に重要です。CUDAを手書きする場合はこれらの解析をエンジニアが行う必要があり、個人のスキルへの依存度が高くなりますが、自動コード生成であればこのような作業に要する工数を削減でき、さらに手書きによるコーディングエラーを回避できます。

GPU Coderによって生成したコードはデスクトップ、クラウド、NVIDIA® Jetson やNVIDIA Drive®プラットフォーム等の組み込みデバイス等様々なGPU環境で利用できます。図1および図2はGPU Coderを利用することでどの程度パフォーマンスが改善されたのかを表しています。SURF特徴量抽出、ディスパリティの計算、霧除去アルゴリズムやFFTといった画像処理・信号処理において、数十倍の高速化が実現できていることが確認できます。

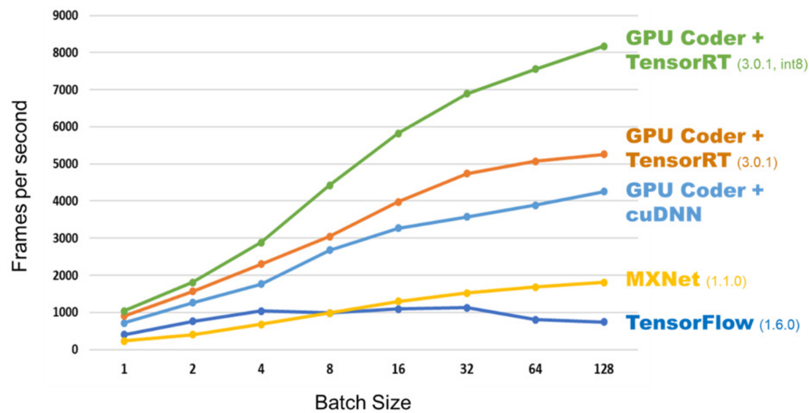


Testing Platform	MATLAB 2018a
CPU	Intel Xeon CPU E5-1650 v3 @ 3.50 GHz
GPU	NVIDIA Pascal TITAN V (Volta architecture)
CUDA	Version 9.0

図1. 代表的な画像処理・信号処理におけるパフォーマンス ベンチマーク。

図2ではディープラーニングの推論におけるベンチマークを示しています。GPU Coderで生成したコードと、様々なディープラーニングフレームワークの比較となります。GPU Coderで生成されたコードはTensorFlow™より5倍、MXNetより2倍高速に動作することが示されています。

MathWorks®



Testing Platform	MATLAB 2018a
CPU	Intel Xeon CPU E5-1650 v4 @ 3.60 GHz
GPU	NVIDIA Pascal TITAN Xp
cuDNN	v7

図 2. ディープラーニング パフォーマンス ベンチマーク: AlexNetの推論実行時のパフォーマンスをGPU Coder, TensorFlow, MxNetで比較(NVIDIA Titan® Xp使用) 。

ここで、ディープラーニングのネットワークを利用するためにはほとんどのケースで前処理や後処理が必要となることに注目してください。例えば、非常にシンプルな画像分類のタスクにおいて、画像データをネットワークに通す前に色空間の変換や画像のリサイズ等が必要になる場合があります。また、後処理として分類結果を画像として表示させるようなケースも考えられますが、GPU Coderはこのような前処理・後処理を含むアプリケーション全体に対してコード生成が可能です。

図3には工業製品の不良品検出を行うためのアルゴリズム例が示されています。この例では六角ナットのキズの有無を判定していますが、六角ナットが写っている箇所をROIとして抽出するための前処理、キズの有無を判定するためのCNN、後処理として判定結果の可視化する部分の3要素で構成されています。GPU Coderはこれらの処理部分全てをコード生成対象として指定することが可能です。

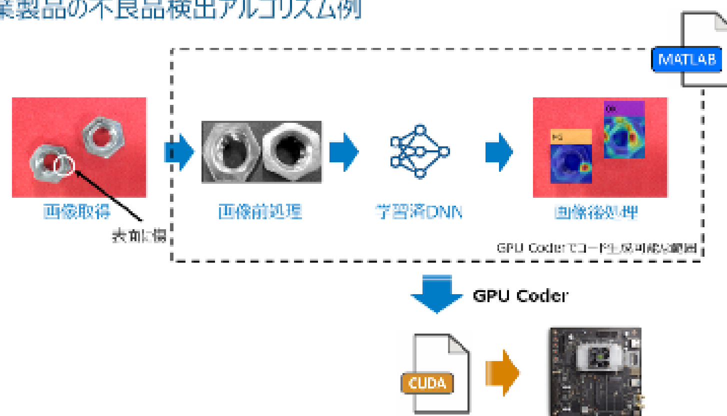


図3. 前処理・後処理を含むアプリケーション全体に対するコード生成。

GPU Coderを利用した実装ワークフロー

アルゴリズム開発から自動コード生成による実装までの一般的なワークフローを図4に示します。手順は以下の通りです。

#1, コード生成の対象となるMATLABコードの準備

#2, 生成されたコードがコード生成元のMATLABコードと機能的に同じ振る舞いをするか確認

#3, MEXファイルの生成

(シミュレーション高速化の目的で、コード生成元のMATLABコードと置換)

もしくは既存のCUDAプロジェクトと統合するために、CUDAソースコード or 静的ライブラリ or 動的ライブラリを生成

#4, [任意]パフォーマンスを最適化するためのコード解析やMATLABコードの書き換え

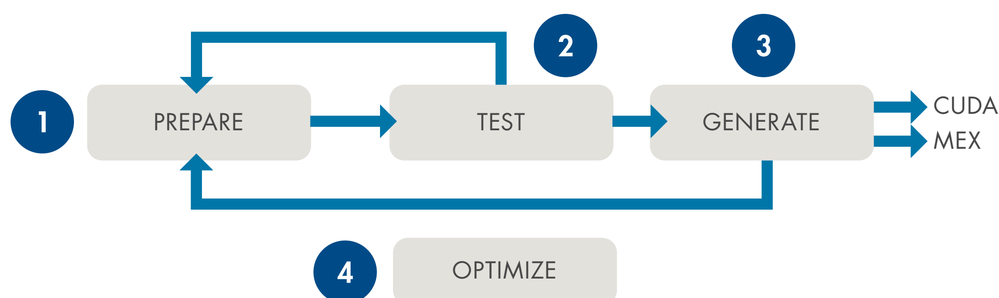


図4. MATLABコードからCUDAコードを生成するためのワークフロー。

道路標識検出アプリケーション

さて、より解りやすくワークフローをご紹介するために、ディープラーニングを利用した道路標識の検出例を具体例として用います。この例題では、大別して"標識の検出"、"NMS(検出された領域の統合)"、"標識の認識"の3ステップで最終的に標識の検出および認識を行っています。例題のMATLABのコードでは以下のような流れで処理が行われます。

- テスト用画像の読み込みと前処理
- 標識を検出するための、物体検出用ネットワークの実行
- NVSアルゴリズムを利用して検出された領域を統合
- 標識を認識するためのネットワークの実行
- 標識が検出された領域に注釈を挿入し、最終的な出力として表示

これらの前処理・後処理を含むアルゴリズムからCUDAコードを生成するために、MATLABコードをコード生成が可能な状態に修正していきます。

MathWorks®

CUDAコード生成のためのMATLABコードの準備

コード生成を行うために、以下の手順に沿って既存のMATLABコードを修正していきます。

1. **アルゴリズムとその検証用テストベンチを分離** コード生成の対象としたいアルゴリズム部分をユーザー定義関数として分離し、その関数の検証用としてテストベンチを準備します。前処理や後処理、ディープラーニングの推論部分、といった形で処理毎に関数を作成するのが良いかもしれません。

2. **アルゴリズムの中で、計算コストの高い部分を特定** MATLAB プロファイラ を利用してプロファイリングを行い、アルゴリズムの中で処理に時間が掛かっている部分(計算コストの高い部分)を特定します。そして、その部分を別関数として分離しておきます。

3. **CUDA カーネルを生成するための下準備** `coder.gpu.kernelfun`プラグマを当該関数の冒頭に挿入します。このことにより、GPU CoderはCUDAカーネルを生成するための解析をどの部分に対して実行すればよいのかを認識できるようになります。

4. **コード生成可能な文法や関数で構成されているか確認** GPU Coder App(図5)を利用します。このAppはMATLABコードがCUDAコード生成可能な状態になっているかを確認するための機能を内蔵しており、コード生成を妨げるような問題、コード生成サポート外の関数や文法を検出できます。
   - **Unsupported functions:** コード生成の対象となる関数に unsupported functions が含まれている場合、削除するか別関数で置き換える等の作業を行います。
   - **Unbounded variable-size data:** コード生成時に配列のサイズが一意に決まらない場合やサイズが動的に変わると判断された場合、可変サイズの変数として認識されます(図6)。GPU Coderは上限指定のない変数や可変サイズの配列に対してワーニングを出力します。この場合、GPU Coderがメモリを静的に割り当てることができるように、サイズの上限を指定する必要があります。

     Â 可変サイズ入力の上限を指定します: `coder.typeof` を利用して入力サイズの上限を指定します。

     Â ローカル変数のサイズの上限を指定します: `assert`関数を関係演算子と組み合わせて利用することで可変サイズ配列の上限を指定するか`coder.varsize`を利用します。

5. **実行時の問題確認** コード生成を妨げる要因がないかどうかを確認した後、ランタイムエラーチェックを実施してください。メモリの整合性、配列上限や次元の検証を行います。また、GPU固有の問題の確認を行うために、GPUオプションを選択します。この場合はRegister Spill(カーネルの使用するレジスタ数がハードウェア上限を超えていないか)の確認や、スタックサイズの確認を行います。

図5. GPU Coder App.

MathWorks®

Order	Variable	Type	Size	Class	Complex
1	C	Output	1 x :100	double	No
2	A	Input	1 x :100	double	No
3	B	Input	1 x :?	double	No

(Tabs: Summary | All Messages (0) | Variables | Target Build Log)

図6. Sizeと書かれた列は計算された配列の最大サイズを示します。コロン(:)は配列が可変サイズであることを示しており、クエスチョンマーク(?)はサイズの上限が決まっていない事を示しています。

CUDAコード生成のためのMATLABコードの準備：道路標識検出アルゴリズムの例

前のセクションで示した通り、まず初めにコード生成の対象とする部分(アルゴリズム部分)を関数として分離します。ここではコード生成の対象とする部分を`tsdr_predict`という名前の関数とし、テストベンチを`tsdr_testVideo.m`として保存しています。次に、MATLABプロファイラを利用して実行時間を計測します。12行目~22行目をコメントアウトし、34行目の`tsdr_predict_mex`を`tsdr_predict`に書き換えてから実行します。プロファイル概要では目立って時間を要している関数は見当たりませんので、この結果からさらに特定部分を分離する必要はなく、`tsdr_predict`関数全体に対してコード生成が可能と判断できます。ここで、関数に `coder.gpu.kernelfun` プラグマを挿入しておきます。次に、コード生成を妨げる要因があるかどうかの確認ですが、上限の指定がない可変サイズ配列がありますので、`coder.varsize`を使用して上限を指定します(図7)。また、コード生成でサポートされていない関数は使用されていませんので、関数の削除や置き換え等の必要はありません。最後にランタイムエラーチェックですが、CPU、GPUどちらの環境でもエラーが検出されることなく完了することを確認します。

```
%% Run Non-Maximal Suppresion on the detected bounding boxes
coder.varsize('selectedBbox',[98, 4],[1 0]);
[selectedBbox,~] = selectStrongestBbox(round(boxes),probs);

%% Recognition

persistent recognitionnet;
if isempty(recognitionnet)
    recognitionnet = coder.loadDeepLearningNetwork('RecognitionNet.mat','Recognition');
end

idx = zeros(size(selectedBbox,1),1);
inpImg = coder.nullcopy(zeros(48,48,3,size(selectedBbox,1)));
```

図7. `coder.varsize`を利用し、可変サイズ配列の上限を指定

MathWorks®

MEXを利用した検証

ここまでの作業によってMATLABコードはコード生成可能な状態になっていますが、ターゲットのハードウェア向けにコード生成を行う前に検証を行います。GPU Coderはビルドタイプとして MEXを選択できますので、MEXを利用してコード生成の元になった関数と生成されたコードが機能的に同じ振る舞いをするかどうかの確認を行います。

MEXは生成されたコードを再びMATLAB上で利用するためのWrapperインタフェースで、コード生成元のMATLAB関数と直接置き換える形で利用することができます。

MEXを生成するためにはGPU Coder AppでビルドタイプとしてMEXを指定し(図8)、NVIDIAのnvccコンパイラを利用してMEX関数としてコンパイルを行います。元のMATLAB関数との比較はGPU Coder App上の"コードの検証"機能を利用して行うことができます。

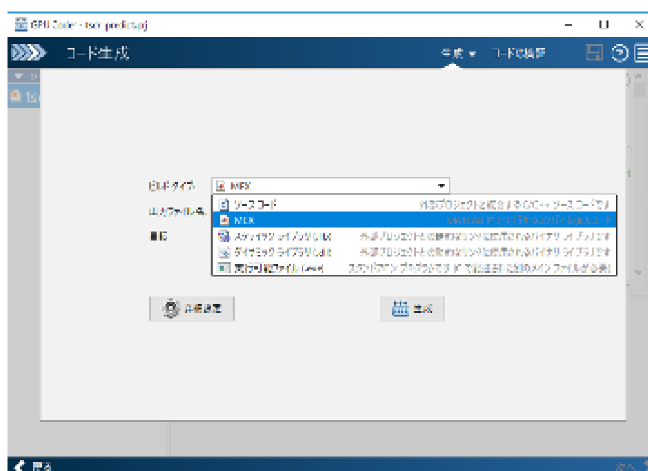


図8. GPU Coder Appにおけるビルドタイプの選択とコードの検証

さて、道路標識検出アルゴリズムの例題に戻ります。**tsdr _ testVideo.m**の12行目~22行目をコメントアウトして使用していましたが、MEXを生成するためにこのコメントを解除します。コード生成はGPU Coder Appだけではなくコマンドラインからも可能で、12行目~22行目にはMEX生成に必要な設定や関数が記述されています。そして、34行目の**tsdr _ predict**を**tsdr _ predict _ mex**に戻し、テストベンチを実行してみます。MEX生成と生成されたMEXによる道路標識検出のアルゴリズムが実行され、元のMATLAB関数と同じ結果が得られることを確認できます。

さらに、GPU Coderと Embedded Coder® を組み合わせて使うことで、software-in-the-loop (SIL) execution を利用した検証が可能になります。SILを利用することで、ターゲットハードウェア上で実行した結果とコード生成の元になった関数の振る舞いを比較することが可能です。

ソースコードもしくはMEXとしてのコード生成:
デスクトップ、クラウド環境、組み込みGPUへの実装

MEXを利用した検証の結果生成されたコードの振る舞いに問題がなければ、ビルドタイプを変更してC++ソースコードもしくは静的ライブラリを生成し、既存のアプリケーションに統合してターゲットデバイスに実装します。ターゲットのプラットフォームがNVIDIA JetsonやDriveといった組み込み系のハードウェアである場合は、生成されたコード類をターゲットハードウェアにコピーし、ターゲット上でビルド&実行します。ターゲットがNVIDIA JetsonのTX2, TX1もしくはTK1である場合は、ホストマシン上でクロスコンパイルを行い、生成された実行形式のファイルをターゲットにコピー&実行することで代替することもできます。

また、GPU Coderの利用用途として、MATLAB上でのシミュレーションの高速化も考えられます。GPU Coderを利用して作成したMEXはGPU上で計算されることになるため、MATLAB関数としての実行に比べて高速化されることが期待できます。

コード生成及び実装: 道路標識検出アルゴリズムの例

道路標識検出の例において、最終的なゴールはアルゴリズムをデスクトップ環境に配布することです。この場合、GPU Coder App上でビルドタイプとしてソースコードを選択し、生成されたコードをメインファイルと統合して実行ファイルをビルドします。また、NVIDIA Jetsonのような組み込みGPU向けにクロスコンパイル可能な静的ライブラリを生成し、配布することも可能です。

最適化されたコードを生成するためのTips

生成されるコードのパフォーマンスをより改善するために、追加で実行・設定できる機能やオプションがあります:

- **MEX関数のプロファイリング** 生成されたコードのパフォーマンスにおけるボトルネックを特定するために、MATLABプロファイラを利用して生成されたMEX関数の実行時間をプロファイリングできます。生成されたコードに対するプロファイルには、MATLAB関数の各行の呼び出し回数や要した時間が表示されます。特定されたボトルネックに対しては、MATLABのアルゴリズムに若干の修正を加えるか、コード生成時のオプション設定を調整することで改善される場合があります。幾つかの手法を以下に示します。
- **`coder.gpu.kernel` プラグマの利用** アルゴリズムに並列ループが含まれているにもかかわらず生成されたコードでは並列化がされていなかった(カーネルが生成されなかった)場合、`coder.gpu.kernel` プラグマを当該並列ループの直前に挿入してください。これによりGPU Coderは強制的にカーネルを生成します。但し、これはアドバンストな使い方になりますので、本プラグマは慎重に利用してください。
- **デザインパタンの利用** 複数の隣接する出力配列を計算するために入力配列に繰り返しアクセスするような、配列演算において良く用いられるステンシル演算 (`gpucoder.stencilKernel`) や行列x行列演算 (`gpucoder.matrixMatrixKernel`) については専用のデザインパタンが準備されており、演算効率を改善することができます。これらのデザインパタンではGPUの共有メモリを活用することでパフォーマンスの向上が図られています。詳細や使い方についてはこちら design pattern documentation を参考にしてください。
- **カスタムCUDAコードの利用** 特定のアルゴリズムを実現するための高度に最適化されたCUDAコードが既に手元にあり、GPU Coderで生成されるコードに統合したい場合は、`coder.ceval`を利用することで外部関数として既存のコードを呼び出すことができます。詳細はこちらのヘルプドキュメントdocumentation on legacy code integration を参考にしてください。
- **コード生成オプションのカスタマイズ** コード生成用の様々なオプションを調整することで、目的に応じて最適化されたコードを生成できる場合があります:
  - メモリ割り当てモードの設定 : ユニファイドメモリ選択が可能
  - cuBLAS や cuSOLVERといったCUDAライブラリの有効か
  - ターゲットに応じたCompute Capability の設定
  - GPU コンパイラに渡すコンパイラフラグの追加

Next Steps

- GPU Coder を利用するための例題集 : MATLABからGPUコード生成
- GPU Coder評価版のリクエスト : GPU Coder 評価版

© 2019 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.
Other product or brand names may be trademarks or registered trademarks of their respective holders.