# DirectX* 11 のイメージ処理における高速フーリエ変換

この記事は、インテル® デベロッパー・ゾーンに公開されている「Fast Fourier Transform for Image Processing in DirectX* 11」の日本語参考訳です。

---

FFT コードサンプルのダウンロード

**はじめに**

高速フーリエ変換 (FFT) は、分割統治法を利用した離散フーリエ変換 (DFT) の実装です。DFT は、イメージなどの離散信号を周波数領域へ、あるいは周波数領域から変換します。周波数領域変換では、一般に画像領域での高コストな多くの効果を低コストにできます。画像領域と周波数領域間の変換では、著しいオーバーヘッドが発生します。このサンプルは、計算シェーダーと共有ローカルメモリー (SLM) を使用して、メモリー帯域幅を減らすことでパフォーマンスを向上する、最適化された FFT を紹介します。

サンプルでは、FFT 実行の 2 つの手法を説明します。1 つ目の UAV 手法は、アンオーダード・アクセス・ビュー (UAV) 間で繰り返しデータをやり取りして処理を行います。2 つ目の SLM (共有ローカルメモリー) 手法は、メモリー帯域幅を効率良く使用します。そのため、メモリー帯域幅がボトルネックの場合、パフォーマンスが大幅に向上します。

以前紹介したサンプル「Implementation of Fast Fourier Transform for Image Processing in DirectX 10」(英語) は、ピクセルシェーダーを利用して UAV 手法を実装します。UAV 手法と FFT アルゴリズムについては、以前のサンプルを参照してください。

## UAV 手法

UAV 手法は、以前のサンプルで説明されているように、バタフライパスと呼ばれるパスを実行します。各パスでは、実数および虚数の入力テクスチャーごとに 2 回サンプリングし、実数および複素数の出力テクスチャーへ 1 回書き込みます。キャッシュ・アーキテクチャーによっては、キャッシュ・スラッシングが発生してパフォーマンスが低下することがあります。図 1 は、UAV 手法で必要なディスパッチ呼び出しです。

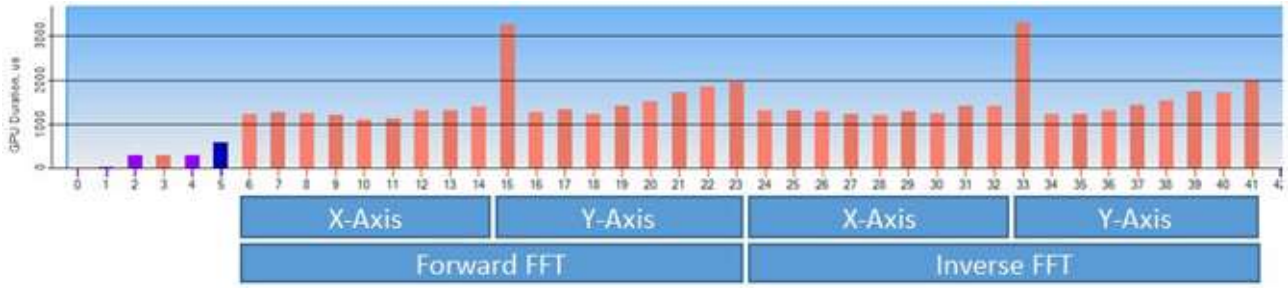


**図 1**: インテル® Graphics Performance Analyzer (インテル® GPA) でキャプチャーされた UAV 手法によるシェーダーのディスパッチ呼び出しの計算

## SLM 手法

SLM 手法は、各 FFT 軸のパスごとに 1 つのディスパッチを行います。テクスチャーの各行ごとにスレッドグループをディスパッチし、各スレッドグループはテクスチャーの列ごとにスレッドを生成します。各行は 1 スレッドグループによって表現され、行内の各ピクセルは 1 つのスレッドによって処理されます。各スレッドは、そのピクセルのすべてのバタフライパスを実行します。パス間には依存性があるため、スレッドグループ内のすべてのスレッドは、ほかのスレッドが次へ進む前に各パスを完了しなければなりません。図 2 にスレッドグループのワークフローを示します。


**図 2**: スレッドグループのワークフロー 入力テクスチャーの各行ごとにスレッドグループが作成されます。

各スレッドがバタフライパスを実行した後に GroupMemoryBarrierWithGroupSync 関数を使用して、確実に同期が行われるようにすることができます。

各パスの結果は SLM 配列の特定のインデックスに格納されます。最初にソース・テクスチャーの入力値が配列の前半部分にロードされ、各後続パスが後半部分に書き込みます。基本的に、UAV 手法で行われるテクスチャー間のやり取りは、SLM でも行われます。

各スレッドで各パスに対して次の処理が行われます。

1. 前のパスの出力へのインデックスを 2 つ計算します。
2. 各インデックスで値をスケーリングするための重み付けを 2 つ計算します。
3. インデックスと重み付けを使用して基本的な演算を実行し、結果を SLM 配列の後半部分に書き込みます。
4. GroupMemoryBarrierWithGroupSync を呼び出して、ほかのスレッドと同期します。
5. 上記のステップを繰り返します。

計算シェーダーで上記の処理を行うためのコードが図 3 です。図 4 は、SLM 手法で必要なディスパッチ呼び出しです。

```
groupshared float3 pingPongArray[4][LENGTH];
void ButterflyPass(int passIndex, uint x, uint t0, uint t1, out float3 resultR,
    out float3 resultI)
{
    uint2 Indices;
    float2 Weights;
    GetButterflyValues( passIndex, x, Indices, Weights );

    float3 inputR1 = pingPongArray[t0][Indices.x];
    float3 inputI1 = pingPongArray[t1][Indices.x];

    float3 inputR2 = pingPongArray[t0][Indices.y];
    float3 inputI2 = pingPongArray[t1][Indices.y];

    resultR = inputR1 + Weights.x * inputR2 - Weights.y * inputI2;
    resultI = inputI1 + Weights.y * inputR2 + Weights.x * inputI2;
}

[numthreads( WIDTH, 1, 1 )]
void ButterflySLM(uint3 position : SV_DispatchThreadID)
{
    // 行全体または列全体を配列にロードする
    pingPongArray[0][position.x].xyz = TextureSourceR[position];
    pingPongArray[1][position.x].xyz = TextureSourceI[position];

    uint4 textureIndices = uint4(0, 1, 2, 3);

    for( int i = 0; i < BUTTERFLY_COUNT-1; i++ )
    {
        GroupMemoryBarrierWithGroupSync();
        ButterflyPass( i, position.x, textureIndices.x, textureIndices.y,
        pingPongArray[textureIndices.z][position.x].xyz,
```

```
        pingPongArray[textureIndices.w][position.x].xyz );
        textureIndices.xyzw = textureIndices.zwxy;
    }

    // 最後のバタフライパスは直接ターゲット・テクスチャーへ書き込む
    GroupMemoryBarrierWithGroupSync();
    ButterflyPass(BUTTERFLY_COUNT - 1, position.x, textureIndices.x,
        textureIndices.y, TextureTargetR[position], TextureTargetI[position]);
}
```

**図 3**: SLM 手法の実装

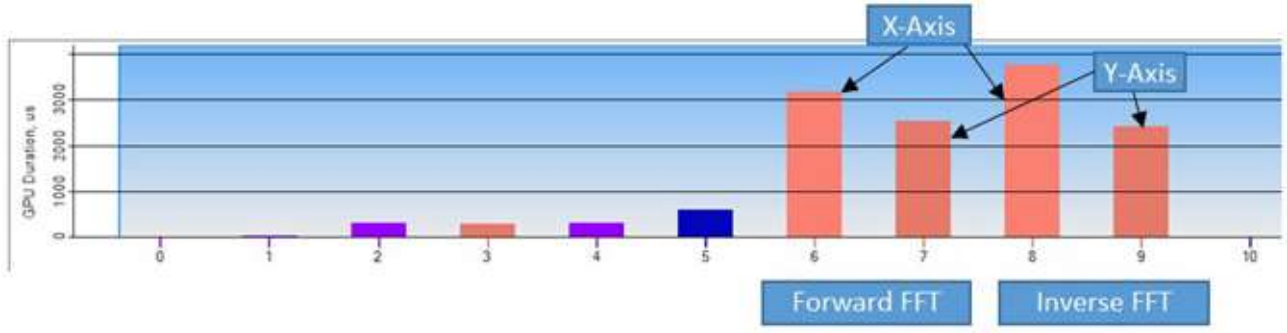


**図 4**: インテル® Graphics Performance Analyzer (インテル® GPA) でキャプチャーされた SLM 手法による計算シェーダーのディスパッチ呼び出しの計算

多少変更を加えることで、y 軸を処理するようにアルゴリズムを変換できます。

## メモリー帯域幅解析

周波数領域の画像の各色成分は、実数または複素数 (虚数) 平面を使用して表現できます。これらの平面は高精度でなければならないため、128 ビットの R32G32B32A32 を使用します。

図 5 の式は、UAV 手法を使用した x 軸に対する FFT のメモリー帯域幅を計算します。各バタフライパスには 4 つのテクスチャー・サンプル (2 つの実数と 2 つの虚数) が必要なため、BytesRead に 2 を 2 回乗算します。

```
TextureSize = Width * Height * 16
PassCount = log2(Width)
BytesRead = TextureSize * PassCount * 2 * 2
BytesWritten = TextureSize * PassCount * 2
```

**図 5**: UAV 手法のメモリー帯域幅

この式を使用して 512x256 テクスチャーのメモリー帯域幅を計算します (図 6)。

```
TextureSize = 512 * 256 * 16 = 2097152
PassCount = log2(512) = 9
BytesRead = 2097152 * 9 * 2 * 2 = 75497472
BytesWritten = 2097152 * 9 * 2 = 37748736
```

**図 6**: UAV 手法による 512x256 テクスチャーのメモリー帯域幅

x 軸のパスだけで、72MB のテクスチャーの読み取りと、32MB のテクスチャーの書き込みが必要なことが分かります。y 軸の変換と逆変換には、さらに多くのメモリー帯域幅が必要です。

SLM 手法は、各行は個別に処理できるという利点を活かして、メモリー帯域幅を減らします。計算シェーダーを使用して、行全体を共有ローカルメモリーに読み込み、すべてのバタフライパスを実行して、結果を出力テクスチャーに 1 回書き込みます。

図 7 は、SLM 手法のメモリー帯域幅を計算する単純な式です。各テクスチャーを 1 回読み書きします。パス回数はメモリー帯域幅に影響しません。

```
TextureSize = Width * Height * 4
BytesRead = TextureSize * 2
BytesWritten = TextureSize * 2
```

**図 7**: SLM 手法のメモリー帯域幅

```
TextureSize = 512 * 256 * 16 = 2097152
BytesRead = 2097152 * 2 = 4194304
BytesWritten = 2097152 * 2 = 4194304
```

**図 8**: SLM 手法による 512x256 テクスチャーのメモリー帯域幅

図 8 で計算した読み書き帯域幅 (4MB) は、最初の虚数テクスチャー (常に 0) の読み取りを回避することでさらに向上できます。その場合、読み取り帯域幅は 2MB、書き込み帯域幅は 4MB になります。x 軸の FFT 変換の場合、SLM 手法では読み取り帯域幅が 1/36、書き込み帯域幅が 1/8 になります。

事前に計算された重み付けとインデックスを用いるバタフライ・ルックアップ・テクスチャーを使用すると、帯域幅が増えます。ルックアップ・テクスチャーは 2 つのインデックスと 2 つの重み付けを R32G32B32A32 形式で格納します。各スレッドグループは、すべてのバタフライパスを通してテクスチャー全体を使用します。そして、ソース・テクスチャーの各行ごとにスレッドグループがあります。図 9 は、512x256 テクスチャーの変換でバタフライ・ルックアップを使用する場合の式です。約 18MB の追加のテクスチャー読み取りが必要になることが分かります (図 10)。

```
PassCount = log2(Width)
ButterflyTextureSize = Width * PassCount * 16
BytesRead = ButterflyTextureSize * Height
```

**図 9**: バタフライ・ルックアップ・テクスチャーを使用する場合のメモリー帯域幅を計算する一般的な式

```
PassCount = log2(512) = 9
ButterflyTextureSize = 512 * 9 * 16 = 73728
BytesRead = 73728 * 256 = 18874368
```

**図 10**: 512x256 ソース・テクスチャーでバタフライ・ルックアップ・テクスチャーを使用する場合に追加で必要なメモリー帯域幅

ここで計算した結果は、部分書き込みが許可される場合に発生する読み取りを考慮しない、理想的なメモリー帯域幅の予測です。そのような追加の読み取りは、メモリー帯域幅に著しく影響する可能性があり、ハードウェア実装に依存します。

## バタフライ・ルックアップ・テクスチャー

各ピクセルのパスで使用するインデックスと重み付けは、事前に計算し、テクスチャーに保存しておくことができます。バタフライ・テクスチャーを使用すると計算を軽減できますが、メモリー帯域幅が著しく増えます。各バタフライを適用すると、各行は基本的に 128 ビットのバタフライ・テクスチャー全体をサンプリングしたことになります。GPU アーキテクチャーによっては、図 11 に示すように、バタフライ・テクスチャーを使用することで実際にパフォーマンスが低下することがあります。

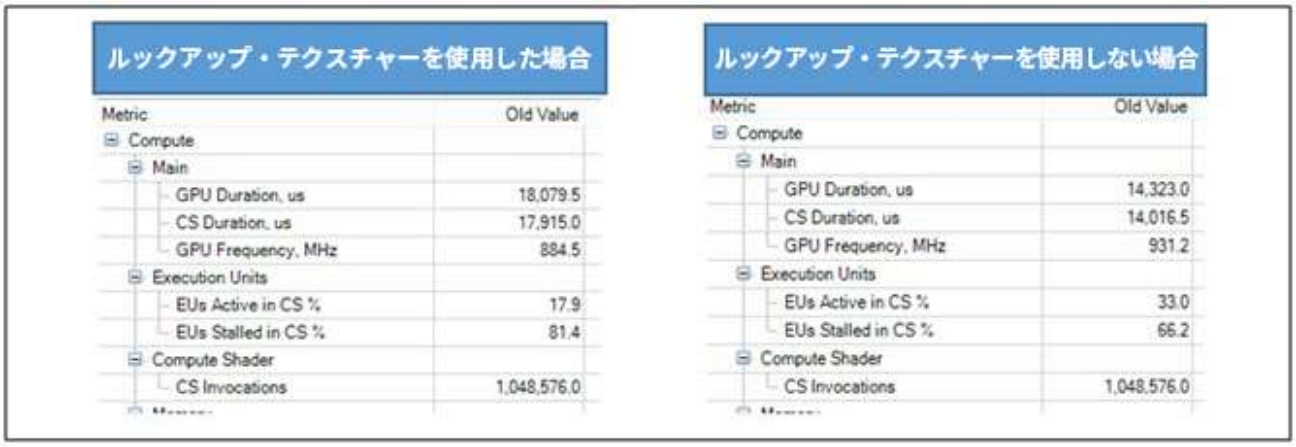


**図 11**: インテル® Core™ i5-4300U プロセッサーでのインテル® Graphics Performance Analyzer (インテル® GPA) の実行結果 - 計算シェーダーのストール、メモリー読み取り、実行時間が減少していることが分かります。

図 12 は、パス・インデックスとオフセットを受け取り、バタフライ・インデックスと重み付けを計算する HLSL 関数です。

```
void GetButterflyValues( uint passIndex, uint x, out uint2 indices, out float2 weights )
{
    int sectionWidth = 2 << passIndex;
    int halfSectionWidth = sectionWidth / 2;

    int sectionStartOffset = x & ~(sectionWidth - 1);
    int halfSectionOffset = x & (halfSectionWidth - 1);
    int sectionOffset = x & (sectionWidth - 1);

    sincos( 2.0*PI*sectionOffset / (float)sectionWidth, weights.y, weights.x );
    weights.y = -weights.y;

    indices.x = sectionStartOffset + halfSectionOffset;
    indices.y = sectionStartOffset + halfSectionOffset + halfSectionWidth;

    if( passIndex == 0 )
    {
        indices = reversebits(indices) >> (32 - BUTTERFLY_COUNT) & (LENGTH - 1);
    }
}
```

**図 12**: 計算シェーダーのバタフライ・インデックスと重み付けの計算

## パフォーマンス

SLM 手法は、UAV 手法と比べて、インテルおよび AMD* 製のさまざまな GPU で優れたパフォーマンスをもたらします。図 13 は、さまざまな設定で 512x512 テクスチャーを周波数領域へ、あるいは周波数領域から変換した場合の変換にかかった時間です。

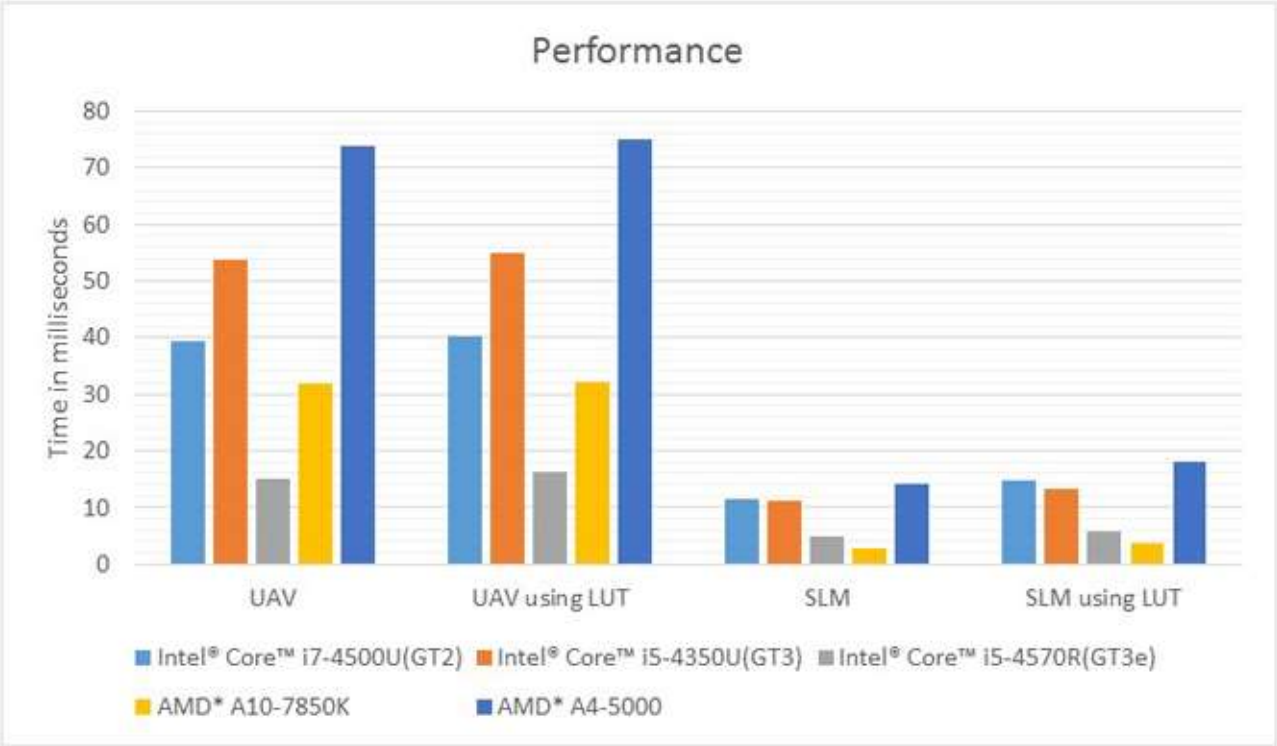


**図 13**: x 軸と y 軸の変換と逆変換の合計実行時間 (ミリ秒)[1]

## その他の最適化

この記事で紹介したサンプルを作成中に、いくつかのほかの最適化の可能性が明らかになりましたが、説明を簡潔にするためここでは取り上げませんでした。

### 成分のパック

ほかの最適化の 1 つは、UAV データをパックして使用していないアルファチャネルを削除することです。残念ながら、UAV テクスチャー・フォーマットは、3 成分の 32 ビット・テクスチャー (R32G32B32) をサポートしていません。ただし、実数および虚数平面は、1 つの R32G32B32A32 テクスチャーと 1 つの R32B32 テクスチャーにパックすることができます。最初のテクスチャーの RGB の実数のすべての成分を格納し、虚数成分を切り分けることで、1 つの成分は ARGB テクスチャーのアルファ成分に、残りの 2 つの成分は R32B32 にすることができます。

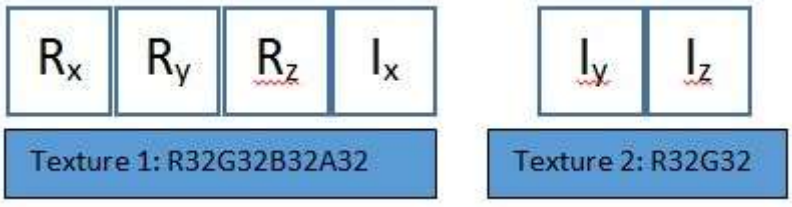


**図 14**: 実数値と虚数値を 1 つの R32G32B32A32 テクスチャーと 1 つの R32G32 テクスチャーにパックする例

### 不要な高精度テクスチャーの回避

サンプルを単純にするため、変換前に入力イメージを実数 R32G32B32A32 テクスチャーにコピーしました。逆変換も同じテクスチャー・フォーマットに出力します。コピーしないで直接ソース・テクスチャーから読み取ると、メモリー帯域幅が減ります。逆変換の結果も低い精度のテクスチャー・フォーマットに書き込まれます。

## まとめ

FFT アルゴリズムで SLM を使用することで、解析した統合 GPU のパフォーマンスが大幅に向上しました。テクスチャーに一時データを保持し、メモリーによって制約を受けるマルチパス・アルゴリズムには、SLM の使用を検討すべきです。ここで紹介したサンプルコードを使用して、ほかの GPU でパフォーマンスを解析し、どちらの手法が最適か判断することができます。2 つの手法は似ているため、わずかなコード変更でどちらの手法にも対応できるでしょう。

[1] 性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

システム構成:
**詳細については、http://www.intel.co.jp/jp/performance/resources/benchmark_limitations.htm を参照してください。**

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください