ArrayFire* と oneAPI による 2 次元フーリエ相関アルゴリズムの高速化

ヘテロジニアス並列処理について ArrayFire* と oneAPI の抽象化を徹底比較

Henry A Gabb インテル コーポレーション シニア主席エンジニア兼 The Parallel Universe 編集長
Umar Arshad ArrayFire ソフトウェア・エンジニア

「oneAPI によるフーリエ相関アルゴリズムの実装」（The Parallel Universe 44 号）では、SYCL* とインテル® oneMKL 関数の組み合わせを使用して 2 つの信号の 1 次元相互相関を計算する方法を紹介しました。この記事では、2 次元相互相関を調べて、2 つの類似した画像の最適な重なりを見つけます（**図 1**）。2 次元のユースケースの説明に加えて、1 つのコードでどこでも実行できるヘテロジニアス並列処理である oneAPI と ArrayFire* の 2 つのアプローチを比較して、以前のワークを拡張します。[**編集者注：**「ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性」（The Parallel Universe 47 号）も参考になります。]

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。

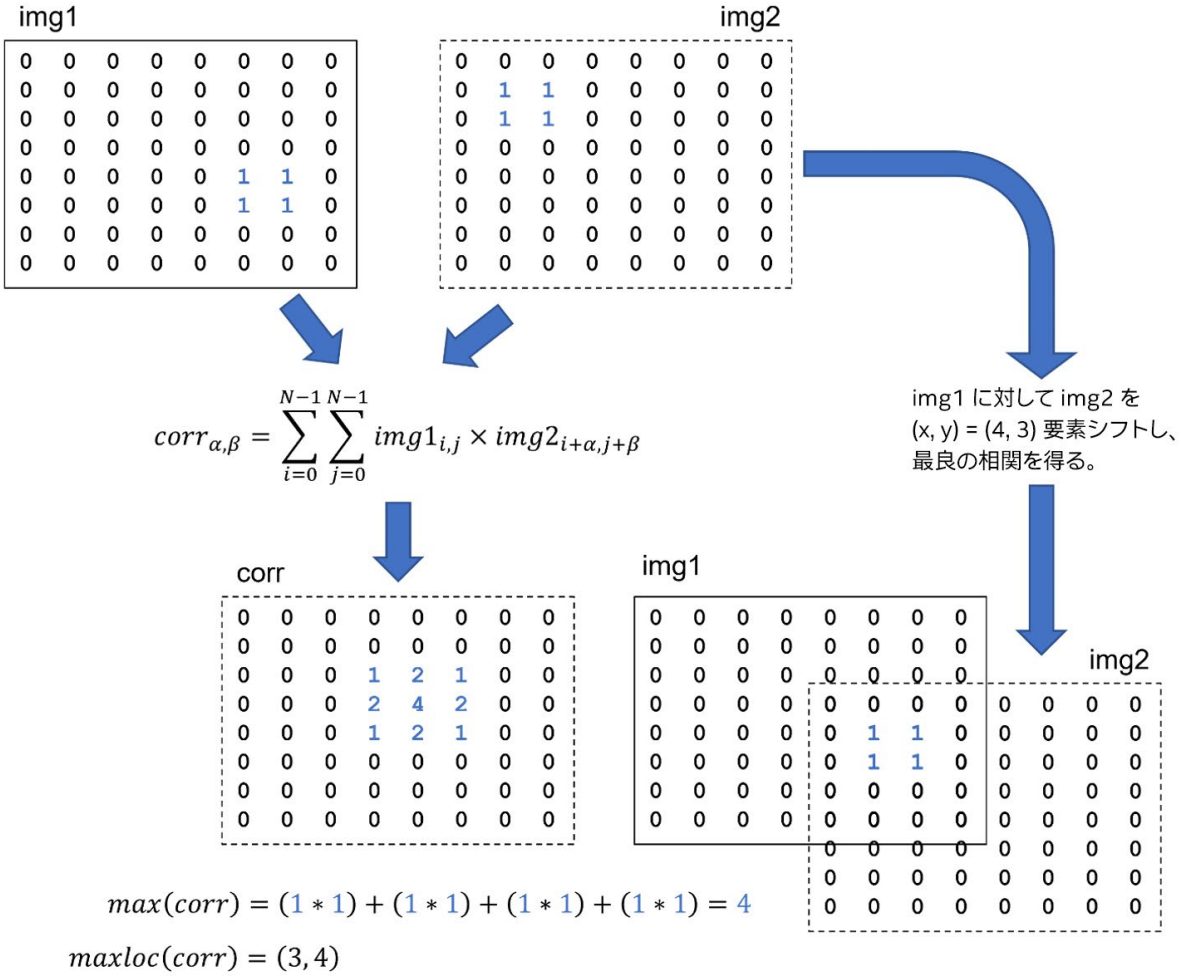

図 1.（2 進数の正方行列として表された）2 つの画像の最適な配置を見つける。ここで、(α , β) は img1 に対する img2 の変位。相関を計算するとき、2 番目の画像は循環シフトされることに注意。

図 1 の総当たり法の総和は非効率で、大規模な問題では実行できない可能性があるため、以前の 1 次元の例と同様に、2 次元相互相関でもフーリエ相関アルゴリズムを利用して計算の複雑さを大幅に軽減します（**図 2**）。両方のアプローチを使用して 2 次元フーリエ相関を実装し、コードを並べて表示して違いと相対的な強みを示します。

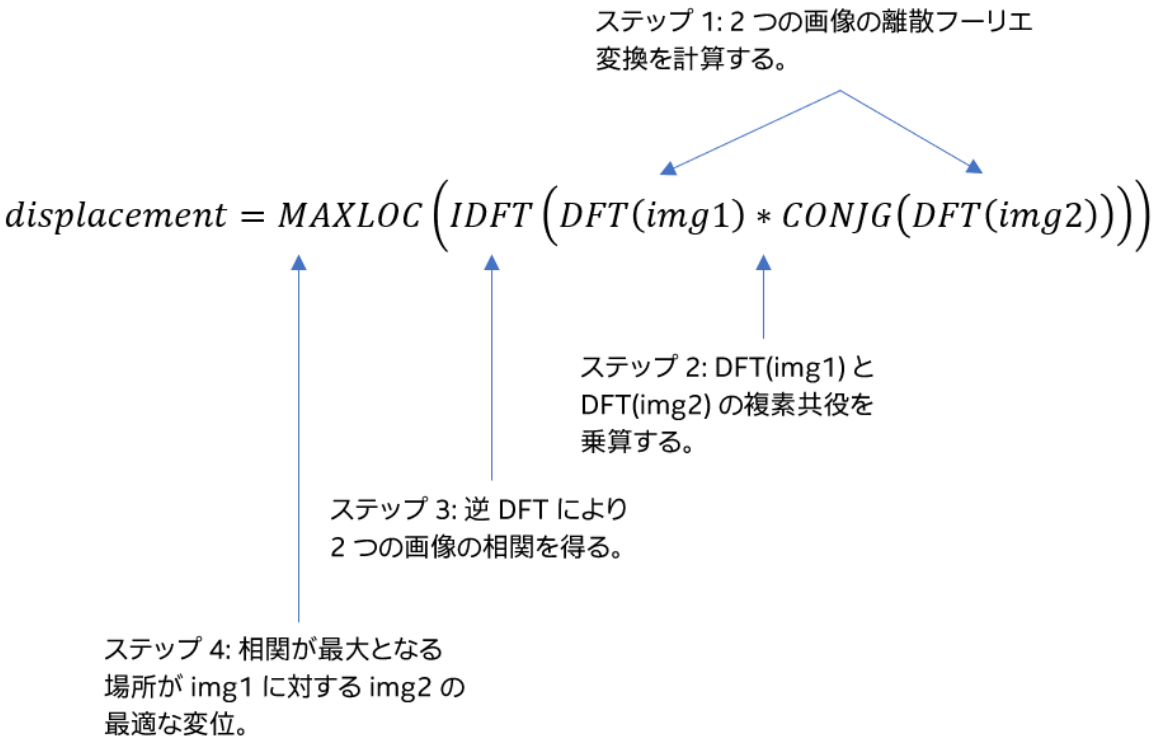

図 2. 4 ステップのフーリエ相関アルゴリズム。DFT は離散フーリエ変換、IDFT は逆 DFT、CONJG は複素共役、MAXLOC は最大相関スコアの場所。

2 次元の oneAPI 実装では、以前の 1 次元実装との違いと改善点も示します。まず、oneMKL の多次元フーリエ変換では、実数領域と複素数領域のデータレイアウトを考慮する必要があります。以降の例で示します。次に、1 次元実装では、SYCL* で記述された `MAXLOC` リダクション演算子を使用しました。2 次元実装では、このコードを oneAPI DPC++ ライブラリー (oneDPL (英語)) の標準関数に変更します。この変更により、コードが明確で簡潔になります。

フーリエ相関アルゴリズムの実装

理想的には、画像がデバイスのメモリーにロードされた後に、すべての計算がデバイスで実行されるべきです。ホストで必要なのは、最終的な変位(2 次元相関の 2 つのスカラー)のみです。その他のホストとデバイス間のデータ転送は不要であり、特に画像が大きい場合、パフォーマンスが低下します。

ArrayFire* によるアクセラレーター・オフロード

図 2 の各ステップは実際には 1 つの文にマップされるため（**図 3**）、ArrayFire* でフーリエ相関アルゴリズムを実装するのは簡単です。C++ 、MATLAB（英語）、Fortran、NumPy* の配列表記を知っているプログラマーなら、コメントがなくても、このコードが理解できるでしょう。このコードには明示的なループはなく、明示的なホストとデバイス間のデータ転送やデバイスのオフロードもありません。コーディングの抽象化は暗黙的にデータ並列であり、これらの詳細は ArrayFire* ランタイム・ライブラリーによって処理されます。

```
#include <iostream>
#include <arrayfire.h>

int main(int argc, char **argv)
{
    // デバイスを選択して ArrayFire* 情報を表示
    af::setDevice(0);
    af::info();

    // 2 次元画像配列を割り当てて初期化（注： ArrayFire* は列優先）
    unsigned int n_rows = 8, n_cols = 8;

    auto img1 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto img2 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto corr = af::constant(0.0, n_rows, n_cols, af::dtype::f32);

    img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;  // 最初の画像の右下に要素を設定
    img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;  // 2 番目の画像の左上に要素を設定

    // ステップ 1: DFT(img1) と DFT(img2) を計算
    img1 = af::fftR2C<2>(img1, 0.0);
    img2 = af::fftR2C<2>(img2, 0.0);

    // ステップ 2: DFT(img1) * CONJG(DFT(img2)) を計算
    corr = img1 * af::conjg(img2);

    // ステップ 3: 逆 DFT を実行
    corr = af::fftC2R<2>(corr, 0.0);

    // ステップ 4: img1 に対する img2 の最適な変位を見つける
    af::array max_score, shift;
    af::max(max_score, shift, af::flat(corr));

    auto max_corr = max_score.scalar<float>();
    auto s = shift.scalar<unsigned>();
    int x_shift = s / n_cols;
    int y_shift = s % n_rows;

    std::cout << std::endl << "Shift the second image (x, y) = (" << x_shift << ", " << y_shift
              << ") elements relative to the first image to get a maximum," << std::endl
              << "normalized correlation score of " << max_corr
              << ". Treat the images as circularly shifted versions of each other." << std::endl;
}
```

図 3. ArrayFire* を使用して実装したフーリエ相関アルゴリズム

ほとんどの ArrayFire* 関数は非同期であるため、呼び出し元は関数がリターンするのを待たずに続行します。関数呼び出しは、内部のインオーダー・デバイス・キューに追加されます。しかし、明示的な同期を使用して、キューが空になるまで待機することもできます。次に例を示します。

```
// ステップ 1: DFT(img1) と DFT(img2) を計算
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
af::sync();
```

`sync` 文は、2 つの順変換が完了するまで呼び出し元を待機させます。

ArrayFire* は、一部の計算の遅延評価も行います。この場合、命令の順序は保存されますが、結果が必要になるまでデバイスキューにワークが送信されません。その後にカーネルが生成され、キューに追加されます。共役乗算文は、遅延評価の例です。

```
// ステップ 2: DFT(img1) * CONJG(DFT(img2)) を計算
corr = img1 * af::conjg(img2);
corr.eval();
af::sync();
```

`eval` 文は、カーネルを強制的に作成してキューに追加します。`sync` 文は、共役乗算操作が完了するまで呼び出し元を待機させます。

oneAPI によるアクセラレーター・オフロード

oneAPI を使用した 1 次元フーリエ相関の実装は、以前の記事およびウェビナー（英語）で紹介しました。その原理は 2 次元の場合も同じであるため、このセクションでは、oneAPI 実装の各ステップを説明するのではなく、oneAPI のコードと ArrayFire* のコードを比較して、ヘテロジニアス並列処理に対する 2 つのアプローチの違いを説明します。

デバイスの画像の初期化

ホスト CPU とさまざまなアクセラレーター・デバイス間のデータ移動は、ヘテロジニアス並列プログラミングにおける重要な考慮事項です。アルゴリズムの一部のステップをホストで実行し、ほかのステップをデバイスで実行すると、ホストとデバイス間のデータ転送により、アクセラレーター・オフロードのパフォーマンスの利点が制限されます。幸いなことに、フーリエ相関アルゴリズムの各ステップはデバイスで実行できます。デバイスメモリーに転送した画像を、ホストに戻す必要はありません。

デバイスのデータを初期化する ArrayFire* コードと SYCL* コードを**図 4** に示します。簡単にするため、**図 1** と同じ人工の画像を使用しています。オフロードデバイスを設定した後、ArrayFire* コードは便利な関数と配列構文を使用してデータを初期化します（**図 4**、左）。同様に、oneAPI コードはデフォルトデバイスの SYCL* キューを初期化し、統合共有メモリーにインプレースの実数から複素数の変換に十分な空間を割り当てて、データレイアウトを定義します（**図 4**、右）[多次元 DFT のデータレイアウトの説明は本題から外れるためここでは取り上げません。

FFTW のドキュメントに優れた説明（英語）が記載されています。])。SYCL* コードは、SYCL* キューにワークを送信して（つまり、parallel_for と single_task カーネルにより)、デバイスで初期化を実行します。SYCL* キューは非同期であるため、これらのカーネルは、ワークが完了するまで待機してから続行するように明示的に指示されています。

```
af::setDevice(0);

auto img1 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto img2 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto corr = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);

img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;
img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;
```

```
// SYCL* キューを初期化
sycl::queue Q(sycl::default_selector{});

// 2 次元画像と相関配列を割り当て
auto img1 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto img2 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto corr = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);

// 行優先アドレス指定の一般化されたストライドを設定
int r_stride = 1;
int c_stride = (n_cols / 2 + 1) * 2;
int c_stride_h = (n_cols / 2 + 1);

// 人工データで入力画像を初期化
// デバイスで初期化を行う
Q.parallel_for<>(sycl::range<2>{n_rows, n_cols},
                 [=](sycl::id<2> idx)
{
   unsigned int r = idx[0];
   unsigned int c = idx[1];
   img1[r * c_stride + c * r_stride] = 0.0;
   img2[r * c_stride + c * r_stride] = 0.0;
   corr[r * c_stride + c * r_stride] = 0.0;
}).wait();

Q.single_task<>([=]()
{
   // 最初の画像の右下に要素を設定
   img1[4 * c_stride + 5 * r_stride] = 1.0;
   img1[4 * c_stride + 6 * r_stride] = 1.0;
   img1[5 * c_stride + 5 * r_stride] = 1.0;
   img1[5 * c_stride + 6 * r_stride] = 1.0;

   // 2 番目の画像の左上に要素を設定
   img2[1 * c_stride + 1 * r_stride] = 1.0;
   img2[1 * c_stride + 2 * r_stride] = 1.0;
   img2[2 * c_stride + 1 * r_stride] = 1.0;
   img2[2 * c_stride + 2 * r_stride] = 1.0;
}).wait();
```

図 4. ArrayFire*（左）と SYCL*（右）を使用したデバイスのデータの初期化。

ホストとデバイス間の不要な転送を防ぐため、ホストはデバイスメモリーのデータを変更しないようにします。画像の要素は `single_task` カーネルを使用して設定されています。このカーネルがないと、要素がホストで設定されるため、oneAPI ランタイムはホストとデバイスのデータの整合性を取る必要があります。データが大きい場合（医療用画像処理アプリケーションのボリューム画像など）、この処理によりパフォーマンスが低下します。

ArrayFire* コードは、よりコンパクトで直感的です。oneAPI コードは、データが割り当てられる場所、計算が実行される場所とタイミングが、より明確です。また、人気の高いオープンソースの高速フーリエ変換パッケージである FFTW（英語）との整合性も高くなっています。

ステップ 1：順変換を実行

ArrayFire* コードはシンプルで直感的です（**図 5**、左）。関数呼び出しは、入力データにより定義された精度で、2 次元の実数から複素数(R2C)の非正規化 FFT を指定します。oneMKL DFT 記述子のアプローチ(**図 5**、右)は、それほどコンパクトではありませんが、以前のインテル® MKL DFTI ユーザーと FFTW ユーザーには馴染み深いものです。oneMKL コードは、必要なサイズと次元の、単精度の実数から複素数の変換の記述子を初期化し、この記述子を SYCL* キューにコミットした後、順変換を計算します。`compute_forward` 関数は、後で同期に使用する SYCL* イベントを返します。

```
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
```

```
// FFT 記述子を初期化
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                             oneapi::mkl::dft::domain::REAL>
                                       forward_plan({n_rows, n_cols});

// 実数領域のデータレイアウト
std::int64_t real_layout[4] = {0, c_stride, 1};

// 共役偶数領域のデータレイアウト
std::int64_t complex_layout[4] = {0, c_stride_h, 1};

forward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,
                       real_layout);
forward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_STRIDES,
                       complex_layout);
forward_plan.commit(Q);

auto evt1 = oneapi::mkl::dft::compute_forward(forward_plan, img1);
auto evt2 = oneapi::mkl::dft::compute_forward(forward_plan, img2);
```

図 5. ArrayFire*（左）と oneMKL（右）を使用したデバイスでの順変換（実数から複素数）の実行。

ステップ 2：複素共役乗算

ArrayFire* と oneMKL は、異なるアプローチを使用して共役乗算操作を実行します。ArrayFire* は、単純な配列表記を使用します（**図 6**、左）。プログラマーは、デバイスやデータのレイアウトを指定する必要はありません。oneMKL は、便利な `mulbyconj` 関数を提供します（**図 6**、右）。`mulbyconj` 関数はより複雑ですが、プログラマーは、データレイアウト、計算を実行する場所とタイミング（つまり、SYCL* キューとイベント）を明示的に制御できます。

`corr = img1 * af::conjg(img2);`	`oneapi::mkl::vm::mulbyconj(Q, n_rows * c_stride_h,`
	`reinterpret_cast<std::complex<float>*>(img1),`
	`reinterpret_cast<std::complex<float>*>(img2),`
	`reinterpret_cast<std::complex<float>*>(corr),`
	`{evt1, evt2}).wait();`

図 6. ArrayFire*（左）と oneMKL（右）を使用した複素共役乗算。

ステップ 3：逆変換を実行

ステップ 1 とステップ 3 のコードは似ていますが、1 つの複素数から実数の変換のみ実行される点が異なります。ArrayFire* コード（**図 7**、左）は、`fftR2C` 関数ではなく `fftC2R` 関数を呼び出しています。oneMKL コード（**図 7**、右）は、複素数から実数のデータレイアウトを指定する新しい DFT 記述子を初期化します。

`corr = af::fftC2R<2>(corr, 0.0);`	`oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,`
	`oneapi::mkl::dft::domain::REAL>`
	`backward_plan({n_rows, n_`
	`cols});`
	
	`// 共役偶数領域のデータレイアウト`
	`backward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,`
	`complex_layout);`
	
	`// 実数領域のデータレイアウト`
	`backward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_`
	`STRIDES,`
	`real_layout);`
	`backward_plan.commit(Q);`
	
	`auto bwd = oneapi::mkl::dft::compute_backward(backward_plan, corr);`
	`bwd.wait();`

図 7. ArrayFire*（左）と oneMKL（右）を使用したデバイスでの逆変換（複素数から実数）の実行。

ステップ 4 : MAXLOC リダクション

以前の実験で使用した SYCL* MAXLOC リダクション演算子を使用すると、コードは複雑になります (fcorr_1d_usm.cpp（英語）を参照)。oneDPL は、C++ プログラマーに馴染み深い 2 つの関数、max_element（英語）および distance（英語）と同じ機能を提供します。**図 8**（右）の MAXLOC リダクションは、これらの関数の oneDPL 実装を使用しています。SYCL* キューは、max_element に計算を実行する場所を指示します。SYCL* カーネルとは異なり、oneDPL アルゴリズムは同期的であるため、明示的な wait 文はありません。

ArrayFire* コード（**図 8**、左）は以前のステップほど単純ではありません。af::flat 関数で 2 次元 corr 配列を平坦化した後、af::max 関数で最大相関スコアと平坦化された配列内での場所を見つけます。この場所は、ArrayFire* が列優先（**図 3** を参照）であることを考慮して、2 つの画像の最適な配置を提供する x および y シフトに変換されます。oneDPL コード（**図 8**、右）も同様の操作を実行しますが、oneMKL のデータレイアウトを考慮する必要があります。

```
af::array max_score, shift;
af::max(max_score, shift, af::flat(corr));

auto max_corr = max_score.
scalar<float>();
auto s = shift.scalar<unsigned>();
int x_shift = s / n_cols;
int y_shift = s % n_rows;
```

```
auto policy = oneapi::dpl::execution::make_device_policy(Q);
auto maxloc = oneapi::dpl::max_element(policy,
                                       corr,
                                       corr + (n_rows * n_cols * 2 + 2));

auto s = oneapi::dpl::distance(corr, maxloc);
float max_corr = corr[s];
int x_shift = s % (n_cols + 2);
int y_shift = s / (n_rows + 2);
```

図 8. ArrayFire*（左）と oneDPL（右）を使用した MAXLOC リダクションの実行

まとめ

oneAPI と ArrayFire* のアプローチはどちらも、1 つのコードでどこでも実行できるヘテロジニアス並列処理を実現できます。すべての 2 次元フーリエ相関計算が oneAPI または ArrayFire* のライブラリーで行われるため、パフォーマンスについては説明しません。The Parallel Universe では、アプリケーション開発者とコンパイラー / ライブラリー開発者の間の関心の分離（英語）が繰り返しテーマになっています。後者のグループは、主にパフォーマンスと計算効率に関心があります。前者のグループは、主に問題を可能な限り生産的に解決することに関心があります。このグループに属している場合は、優れたパフォーマンスを発揮する高レベルのプログラミング抽象化を好むでしょう。ライブラリーが優れたパフォーマンスを発揮することは当然のことです。

ArrayFire* は高レベルの抽象化を提供するため、ArrayFire* のフーリエ相関の実装は簡潔です。その配列表記は、Fortran、MATLAB、Python* NumPy* プログラマーには馴染み深いものです。ArrayFire* には Python* API も用意されています。遅延評価は計算を開始するときのランタイム制御を意味しますが、上記のように、プログラマーは必要に応じて制御を行うことができます。

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。

oneAPI の実装は、プログラマーがホストとデバイス間のデータ転送、計算を実行する場所とタイミングを細かく制御できるため、より複雑になります。oneMKL の DFT 記述子とデータレイアウトは、以前のインテル® MKL DFTI ユーザーと FFTW ユーザーには馴染み深いものです。実際に、インテル® oneAPI マス・カーネル・ライブラリーは FFTW インターフェイスをサポートしています。最後に、oneDPL の関数は C++ プログラマーには受け入れやすいものです。

最終的には、プロジェクトの要件と個人的な好みで oneAPI と ArrayFire* のどちらかを選択することになるでしょう。

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。