ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性

oneAPI を活用してコードの書き換えを回避

Stefan Yurkevitch ArrayFire ソフトウェア・エンジニア

oneAPI は、ヘテロジニアス・アクセラレーター向けの開発を大幅に簡素化します。一度コードを記述したらどこでも実行できるようにコードを開発する強力な方法を提供します。ArrayFire*（英語）は、多くの計算ドメインで有用な関数の膨大なコレクションを提供している GPU ライブラリーです。ArrayFire* は、oneAPI がソフトウェア開発の世界にもたらす先見性を共有します。この記事では、oneAPI ディープ・ニューラル・ネットワーク (oneDNN)（英語）ライブラリーと SYCL* ベースのデータ並列 C++（DPC++）プログラミング言語を、既存のコードベースに統合する方法を探ります。そして、プログラマーが oneAPI を活用することで、新しいプログラミング・モデルへの移行時にしばしば必要となるコードの書き換えを回避できるようにします。

SYCL* との相互運用性

oneAPI は、クロスアーキテクチャーの並列プログラミングを簡素化する DPC++ とライブラリーの組み合わせです。ライブラリーは、DPC++ 言語と緊密に統合されています。どちらも、さまざまな方法で OpenCL* 実装との相互運用性を提供します。ベースとなる言語では、ほとんどのユースケースをカバーする 3 つの主要な OpenCL* との相互運用性が提供されています（**図 1**）。既存のコードから SYCL* へ、またはその逆の相互運用関数のフローがあります。

1. カーネル文字列からカーネル・オブジェクトを作成して、DPC++ コード内で既存の OpenCL* カーネルを使用します。
2. 既存の SYCL* オブジェクトから OpenCL* オブジェクトを抽出します。
3. 既存の OpenCL* オブジェクトから SYCL* オブジェクトを作成します。

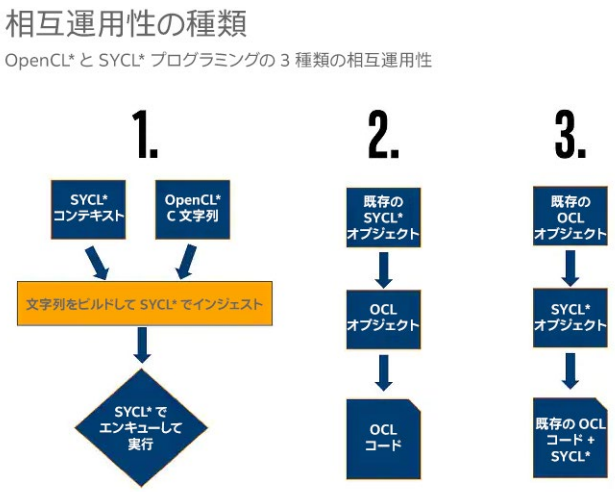

図 1. SYCL* と OpenCL* の相互運用性

これらの相互運用オプションによって既存の ArrayFire* コードベースを統合する方法を考えてみます。最初のケースでは、ArrayFire* カーネルを直接再利用することができます（**図 1** の左）。

```
queue q{gpu_selector()};        // GPU をターゲットとするコマンド キューを作成
program p(q.get_context());     // q と同じコンテキストからプログラムを作成

// R" で示される C++ 生文字列として表現された OpenCL* vecAdd カーネルをコンパイル
p.build_with_source(R"( __kernel void existingArrayFireVecAdd(__global int *a,
                                                              __global int *b,
                                                              __global int *c)
                        {
                                int i = get_global_id(0);
                                c[i] = a[i] + b[i];
                        } )");
  // ここにバッファーを記述 ...
  q.submit([&](handler& h) {
          // ここにアクセサーを記述...
          // カーネルへの引数としてバッファーを設定
          h.set_args(A, B, C);
          // p プログラム・オブジェクトから N 要素に対して vecAdd カーネルを起動
          h.parallel_for(range<1> (N), p.get_kernel("vecAdd"));
  });
```

実際には、ArrayFire* カーネルは単純なバッファーよりも複雑なデータ構造を利用するため、CL 文字列をコピー & ペーストするだけでカーネルを再利用できるわけではありません。ほかの 2 つの方法のいずれかで、データ交換を処理する必要があります。

SYCL* オブジェクトから OpenCL* コンポーネントを抽出する 2 つ目の方法（**図 1** の中央）は、既存の SYCL* オブジェクトに `.get()` を使用するだけの単純なものです。SYCL* オブジェクトの各呼び出しは、対応する OpenCL* オブジェクトを返します。例えば、`cl::sycl::queue::get()` は OpenCL* の `cl_command_queue` を返します。

3 つ目の方法（**図 1** の右）は、既存の OpenCL* オブジェクトを取り込み、それを使用して SYCL* オブジェクトを作成します。`sycl::queue::queue(cl_command_queue,…)` などの SYCL* オブジェクトのコンストラクターを使用します。この場合、コンストラクターは、構築時に OpenCL* リソースの参照カウントを増やすため OpenCL* インスタンスを維持し、SYCL* オブジェクトを破棄する際にインスタンスを解放します。

oneAPI ライブラリーとの相互運用性

oneAPI ライブラリーにも同様の相互運用規則があります。oneMKL のように、DPC++ の相互運用性に直接依存しているライブラリーもあります。これらの関数は、統合共有メモリー（USM）ポインターを受け付けます。この例で使用する oneDNN のように、同様の `.get()` や `constructor()` メカニズムを提供するライブラリーもあります。

oneDNN のデータ構造は、DPC++ と似ていますが、少し異なります。`sycl::device` と `sycl::context` は単一の `dnnl::engine` オブジェクトにまとめられ、`sycl::queue` の代わりに `dnnl::stream` が使用されます。これらの違いはありますが、OpenCL* の相互運用（英語）メカニズムは同じです。OpenCL* オブジェクトは `get` 関数で取得でき、新しい oneDNN オブジェクトは既存の OpenCL* オブジェクトからコンストラクターで作成できます。oneDNN は、同じ機能を持つ明示的な相互運用ヘッダーも提供します。

oneDNN は、サポートするランタイム・バックエンドに柔軟性があります。ハードウェアと対話するため、CPU および GPU エンジンに OpenCL* ランタイムまたは DPC++ ランタイムのいずれかを使用できます。開発者は、OpenCL*（英語）または DPC++（英語）のいずれかを使用するほかのコードで oneDNN を必要とすることがあります。そのため、oneDNN は、対応するオブジェクトと相互運用性がある API 拡張を提供しています。ターゲットに応じて、相互運用 API は `dnnl_ocl.hpp` または `dnnl_sycl.hpp` ヘッダーのいずれかで定義されています。ここで紹介するユースケースは、oneDNN の推論エンジンの機能を ArrayFire* ライブラリーが提供する既存の前処理機能で補完することを目的としています。現時点では、これは OpenCL* の相互運用関数を介して行われます。

ArrayFire* と oneDNN : 詳細

cnn_inference_f32.cpp（英語）サンプルを使用して、OpenCL* と oneDNN の相互運用性を詳しく調査します。このサンプルは、oneDNN を使用して推論用の AlexNet ネットワークをセットアップします。ここでは、ArrayFire* の多くの OpenCL* 画像処理関数を使用して、既存の推論エンジンにデータを供給する前にユーザー入力画像の前処理を行います。ワークフローには、以下のステップが含まれます。

- 関連する相互運用ヘッダーをインクルードします。
- ArrayFire* と `cl_context` を共有し、GPU エンジンを作成します。
- OpenCL* 相互運用インターフェイスを利用して GPU コマンドキューを作成します。
- ArrayFire* で前処理とデータの準備を行います。
- GPU メモリー記述子 / オブジェクトを作成します。
- OpenCL* 相互運用インターフェイスを介して GPU メモリーにアクセスして、入力を取得します。
- oneDNN のプリミティブ / 記述子 / メモリーを作成し、ネットワークを構築します。
- oneDNN で通常通りネットワークを実行します。
- GPU メモリーを解放します。

最初に、ArrayFire* と oneDNN の両方の相互運用ヘッダーをインクルードします。OpenCL* ヘッダーもインクルードします。

```
#include "oneapi/dnnl/dnnl.hpp"     // oneDNN ヘッダー
#include "oneapi/dnnl/dnnl_ocl.hpp" // oneDNN OpenCL* 相互運用ヘッダー

#include <CL/cl.h>       // OpenCL* ヘッダー

#include <arrayfire.h>   // ArrayFire* ヘッダー
#include <af/opencl.h>   // ArrayFire* OpenCL* 相互運用ヘッダー
```

次に、ArrayFire* から OpenCL* のコンテキストとキューを取得し、oneDNN と共有します。

```
cl_device_id     af_device_id = afcl::getDeviceId();
cl_context       af_context   = afcl::getContext();
cl_command_queue af_queue     = afcl::getQueue();
```

OpenCL* オブジェクトを使用して、対応する oneDNN オブジェクトを作成します。相互運用ヘッダー（英語）で定義された相互運用関数を使用します。これらの関数は、追加の `ocl_interop` 名前空間に存在します。オブジェクトは oneDNN スコープのライフタイムを通して保持されることを忘れないでください。

```
dnnl::engine eng = dnnl::ocl_interop::make_engine(af_device_id, af_context);
dnnl::stream s   = dnnl::ocl_interop::make_stream(eng, af_queue);
```

そして、ArrayFire* ライブラリーの GPU アクセラレーション機能を利用して、画像の読み込みや前処理を行います。

```
// oneDNN と同じコンテキストに空の配列を作成
af::array images = af::constant(0.f, h, w, 3, batch);
images = read_images(directory);
images = af::resize(images, 227, 227) / 255.f; // AlexNet の入力サイズに変更して
                                               // 正規化 [0-1]
     images = af::reorder(images, 3, 2, 0, 1); // hwcn -> nchw
...                                            // その他の前処理
```

oneDNN は、dnnl::memory オブジェクトを必要とします。これは生のメモリーではなく、メモリーと dnnl::descriptor のような追加のメタデータを一緒にしたものです。oneDNN はバッファーと USM メモリーの両方のモデルをサポートしており、バッファーがデフォルトです。相互運用性をサポートする oneDNN メモリー・オブジェクトを構築するには、以下の相互運用関数（英語）を使用します。

```
ocl::interop make_memory(
   const memory::desc& memory_desc, // メモリーの形状とレイアウトを示す記述子
   const engine& aengine,           // 相互運用エンジン
   memory_kind kind,                // バッファーまたは USM
   void* handle = DNNL_MEMORY_ALLOCATE // ストレージへのハンドル
   )
```

記述子は、AlexNet の入力が 227 × 227 の NCHW 画像であることを想定したサンプルの記述子と同じです。engine は、ArrayFire* と oneDNN で共有している実行エンジンです。memory_kind は、USM またはバッファーのどちらのインターフェイスを使用するかを指定します。ハンドルポインターを渡す場合、メモリーの種類に応じて処理する必要があります。ハンドルが USM ポインターまたは OpenCL* バッファーの場合、oneDNN ライブラリーはバッファーを所有しないため、ユーザーがメモリーを管理する責任があります。特別な DNNL_MEMORY_ALLOCATE 値を指定すると、ライブラリーがユーザーに代わって新しいバッファーを割り当てます。

oneDNN はバッファーと USM の両方のメモリーモデルをサポートしているので、エンジンとキューを ArrayFire* と共有するオブジェクトに置き換えると、互換性のないメモリー作成モードになります。dnnl::memory オブジェクトの生成時に、以下のエラーが発生します。

```
oneDNN error caught:
        Status: invalid_arguments
        Message: could not create a memory object
```

デフォルトの dnnl::memory 生成の代わりに、以下のように相互運用関数を使用する必要があります。

```
cl_mem *src_mem = images.device<cl_mem>(); // ArrayFire*  から cl_mem を取得
dnnl::memory user_src_memory = ocl_interop::make_memory( // 相互運用メモリー関数
       {{conv1_src_tz}, dt::f32, tag::nchw},             // 記述子を作成
       eng,                                              // エンジンを指定
       ocl_interop::memory_kind::buffer,                 // メモリーの種類を指定
       *src_mem);                                        // cl_mem ハンドルを渡す
```

これは、デフォルトの dnnl::memory 割り当てのすべてのインスタンスに適用されます。ocl_interop::memory_kind::buffer を指定する相互運用関数を使用する必要があります。

```
ocl_interop::make_memory(descriptor, engine, ocl_interop::memory_kind::buffer);
```

最後に、すべての重みをロードしたら、推論プリミティブを作成して、通常通り呼び出します。ネットワークの実行が完了したら、リソースを解放します。

```
// 追加の AlexNet ネットワークのセットアップ
// cnn_inference_f32.cpp の後に重みをロード
...

// 入力を使用して完全な推論を行うためすべてのプリミティブ・ステップを実行
for (size_t i = 0; i < net.size(); ++i) {
  net.at(i).execute(s, net_args.at(i));
}

s.wait();   // ストリームがメモリーへの書き込みを終了するのを待機

images.unlock();  // メモリーの所有権を ArrayFire* に返してリソースを解放
```

DPC++ ランタイムではなく、OpenCL* ランタイムで oneDNN を実行していることを確認します。これは、SYCL_DEVICE_FILTER=opencl 環境変数を指定することで達成できます。修正後の動作する cnn_inference_f32.cpp は、Gist（英語）で参照できます。

まとめ

oneAPI は、既存の OpenCL* コードベースを新しいヘテロジニアス・プログラミング・アプローチと統合するために必要なすべてのツールを提供します。DPC++ では、OpenCL* オブジェクトをどちらの方向でも共有できます。oneAPI ライブラリーは、相互運用タスクを処理する独自の方法を備えています。わずかなコード変更で、OpenCL* ライブラリー全体を書き直すのではなく、再利用することができます。oneAPI は、すでに有用なコードの再開発作業を回避することで、将来の開発時間を節約します。