暗黙の同期を使用してインテル® プロセッサー・グラフィックス上で OpenCL* と OpenGL* 4.3 のサーフェスを共有する

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「Sharing Surfaces between OpenCL™ and OpenGL* 4.3 on Intel® Processor Graphics using implicit synchronization」の日本語参考訳です。

はじめに

この記事では、インテル® プロセッサー・グラフィックス (Microsoft* Windows*) 上で動作する OpenCL* C カーネルによってサブ領域を更新する OpenGL* 4.3 を使用したテクスチャーを作成する方法を説明します。この記事で紹介する 1 つの例は、OpenCL* のイメージ上で機能検出器を実行し、マークされた検出器でリアルタイムに画面に最終出力をレンダリングする、リアルタイム・コンピューター・ビジョン・アプリケーション向けのテクニックです。計算には OpenCL* C カーネル言語を利用しますが、既存のパイプラインとの互換性には OpenGL* API のレンダリング機能を利用します。もう 1 つの例は、シーンの 3D オブジェクトをレンダリングするときにテクスチャーとして使用する、OpenCL* で動的に生成されるプロシージャル・テクスチャーです。最後の例は、3D パイプラインを使用してシーンをレンダリングした後、OpenCL* でイメージを後処理します。この例は、カラー変換、再サンプリング、圧縮の実行などに役立ちます。

このサンプルは、OpenGL* で作成されたテクスチャーを OpenCL* を使用して更新します。同じ推奨事項は、非インタラクティブのオフラインイメージ処理パイプラインで使用されるバーテックス・バッファーやオフスクリーン・フレームバッファー・オブジェクトの更新にも適用されます。

サーフェス共有拡張は、OpenCL* 拡張仕様の `cl_khr_gl_sharing` として定義されています。インテル® プロセッサー・グラフィックスでサポートされている `cl_khr_gl_event` 拡張も利用します。

目的

このチュートリアルの目的は、OpenCL* と OpenGL* で共有されるサーフェスを作成する方法を理解することです。また、API、(インテル® プロセッサー・グラフィックス上での) OpenGL* API のテクスチャー作成パスのパフォーマンス、サーフェスを共有している場合にアドイン GPU とどのように異なるかについても理解します。

概要

OpenGL* テクスチャーを作成して OpenCL* イメージとして共有し、インテル® プロセッサー・グラフィックス上で最高のパフォーマンスを得るには、OpenGL* ピクセル・バッファー・オブジェクト (PBO) を作成しないようにします。インテル® プロセッサー・グラフィックス上では PBO にパフォーマンス上の利点はありません。次に、最小限のデータの追加リニアコピーを作成して、レンダリングのため GPU により実際に使用されるタイル形式のテクスチャーにコピーします。さらに、`glFinish()` を使用して OpenCL* と OpenGL* 間を同期する代わりに、`cl_khr_gl_event` 拡張をサポートする、OpenCL* と OpenGL* 間の暗黙の同期手法をインテル® プロセッサー・グラフィックスで使用します。

インテル® プロセッサー・グラフィックスと共有物理メモリー

インテル® プロセッサー・グラフィックスは CPU とメモリーを共有します。図 1 にこの関係を示します。この図には示されていませんが、メモリー・サブシステムを拡張するいくつかのアーキテクチャー機能があります。例えば、メモリー・サブシステムのパフォーマンスを最大限に引き出せるように、キャッシュ階層、サンプラー、アトミックサポート、読み取り/書き込みキューなどが利用されます。

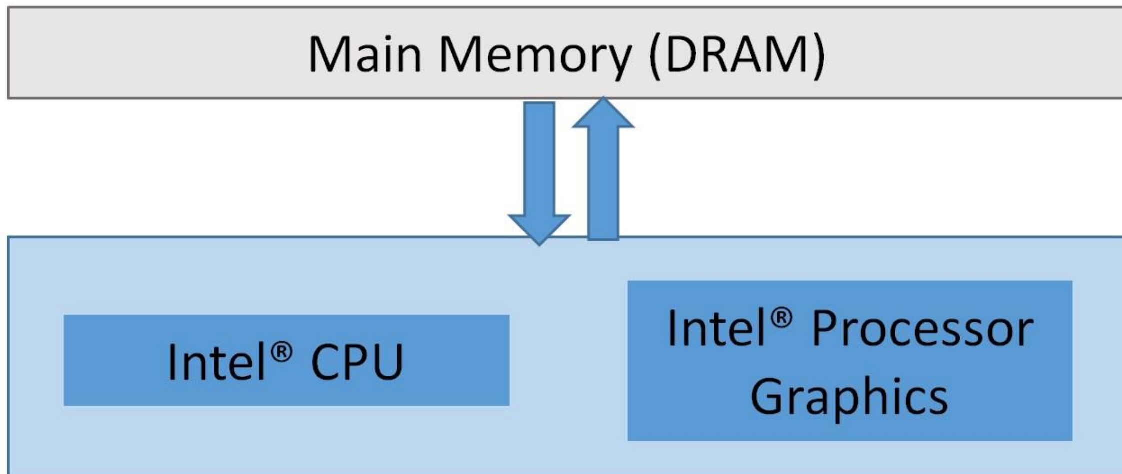

図 1. CPU、インテル® プロセッサー・グラフィックス、メインメモリーの関係。
ドライバーによって管理される専用メモリーがあるアドイン GPU とは違い、
CPU と GPU で 1 つのメモリープールが共有されているのが分かります。

インテル® プロセッサー・グラフィックス上でピクセル・バッファー・オブジェクト (PBO) を使用しない理由

OpenGL* 仕様では、CPU と GPU で共有する場合にピクセル・バッファー・オブジェクトを使用することを奨励しています。

OpenGL* プログラミング・ガイドの第 6 章には、次のように記述されています。

「テクスチャー・データをステージするためにバッファー・オブジェクトを使用する主な利点は、データがシェーダーに要求される時点までに転送を行えばよいため、バッファー・オブジェクトからテクスチャーへの転送を直ちに行う必要がないことです。つまり、アプリケーションの実行と並行して転送を行うことができます。データがアプリケーションのメモリーに位置する場合、**glTexSubImage2D() のセマンティクスにより関数がリターンする前にデータのコピーを作成する必要がある**ため、並列転送は行えません。この方法の利点は、関数がリターンすると直ちにアプリケーションが関数に渡したデータを自由に変更できることです。」

この API 呼び出しは、図 1 で示される同じデバイスと同じ物理メモリーでコマンドストリームを実行する 2 つの API 間の共有**ではなく**、アプリケーション・メモリー (つまり CPU メモリー) と GPU 間の共有に焦点を当てていることに注目してください。

PBO は、共有物理メモリーを利用するデバイスのパフォーマンスを実際に**低下**させます。まず、PBO は、アプリケーションによる追加のメモリー消費が発生する余分なステージング領域です。次に、PBO ではデータは線形的にレイアウトされます。OpenGL* テクスチャーや OpenCL* イメージのようにタイル形式でデータをレイアウトする必要がある場合、データを並び換える必要があります。最後に、API 間で必要なコピーにかかる時間もアプリケーションのパフォーマンスを低下させます。

ただし、外付け GPU と共有する場合、PBO を使用することは適切です (CPU と非同期に実行する DMA 転送を発行できます)。PBO がないと、OpenGL* セマンティクスは同時書き込みを行い、結果が返されるのを待つため、パフォーマンスが低下します。このケースでは、CPU から GPU メモリー・サブシステムへのデータ転送はありません。

サーフェス共有で PBO を使用する場合

PBO を使用するケースはいくつかあります。1 つは、OpenCL* 拡張仕様の表 9.4 で OpenGL* と OpenCL* 間に適したサーフェス形式がない場合です。この場合、PBO を作成し、バッファー共有関連 API で共有します。しかし、前述したパフォーマンスの観点から、このシナリオは避けるようにしてください。必要な場合は、参考文献の Maxim Shevtsov のサンプルを参照してください。

OpenCL* と OpenGL* 間の同期

ランタイム時に OpenCL* と OpenGL* 間で最高のパフォーマンスを得ることは重要です。仕様には次のように記述されています。

「clEnqueueAcquireGLObjects を呼び出す前に、アプリケーションは mem_objects で指定されたオブジェクトにアクセスする保留中の GL 操作が完了していることを保証する必要があります。この保証は、これらのオブジェクトへの保留中の参照を含むすべての GL コンテキストで glFinish コマンドを発行して完了を待つことにより汎用的に達成できます。実装によっては、より効率的な同期手法が用意されていることもあります (一部のプラットフォームでは glFlush の呼び出しで十分である、同期はスレッド内で暗黙的に行われる、GL コマンドストリームでフェンスを配置して CL コマンドキューでフェンスの完了を待つことが可能なベンダー固有の仕様が追加されている、など)。現時点で、glFinish 以外に OpenGL* 実装間で移植可能な同期手法はないことに注意してください。」

最大限の移植性を達成するにはブロッキング呼び出し `glFinish()` を呼び出すと記述されています。インテル® プロセッサー・グラフィックス上では、`cl_khr_gl_events` 拡張を使用して OpenCL* と OpenGL* 間で**暗黙の同期**または同期オブジェクトを使用することにより、優れたパフォーマンスを得ることができます。詳細については後述します。明示的な同期を使用する必要はありません。サンプルコードには、プログラマーが明示的な同期を状況に応じて再利用できるように、コメントアウトされています。

OpenCL* と OpenGL* 間のサーフェス共有の概要

まず、初期化、ランタイム、シャットダウン中にサーフェス共有をサポートする手順を示します。次に、API と言語構文を含む、より詳細な情報を示します。最後に、これらのアイデアをほかのサーフェス形式に拡張する方法を示します。ここでは、ウィンドウ管理に利用可能な `freeglut` と `glew` ライブラリーを使用します。これらのライブラリーの使用は OpenGL* サンプル・アプリケーションの標準的な作業であるため、ここでは詳しく触れません。

初期化

1. OpenCL*:
    1. 拡張がサポートされているかどうか判断し、サポートされていない場合は終了します。
    2. 適切なデバイスオプションを渡すコンテキストを作成します。
    3. OpenGL* と OpenCL* 間の共有をサポートするコンテキストでデバイスにキューを作成します。
2. OpenGL*: OpenCL* と共有する OpenGL* テクスチャーを作成します。

3. OpenCL*: ステップ 2 で作成した OpenGL* ハンドルを使用して、OpenCL* 拡張で共有されるサーフェスを作成します。

ステップ 1 と 2 の順番は入れ換えできます。ステップ 3 はステップ 1 と 2 の後に実行します。

OpenCL* で共有されるサーフェスへの書き込み

1. OpenCL* 排他アクセスのためにサーフェスをロックします。
2. OpenCL* C カーネルでサーフェスに書き込みます。テクスチャー・データの場合、イメージの読み取り/書き込み関数とイメージのパスを適切に使用します。
3. サーフェスのロックを解除します。OpenGL* がサーフェスに読み取り/書き込みできるようになります。

ステップ 1、2、3 を順に実行します。

レンダーループ

この記事では、CPU と GPU 間で共有を行うことに焦点を当てています。レンダーループは、プログラム可能なバーテックス・シェーダーとピクセルシェーダーから結果を得て、それらを表示する四辺形を形成する 2 つの画面向け三角形であるテクスチャー・マップに対する単純なパスを使用します。四辺形は、レンダリングの背景全体にクリアな色を表示するため、全画面を処理しません。

シャットダウン

1. OpenCL* 状態をクリーンアップします。
2. OpenGL* 状態をクリーンアップします。

OpenCL* と OpenGL* 間のサーフェス共有の詳細

このセクションでは、前のセクションで説明した手順を詳細に説明します。

初期化

1. OpenCL*:
    1. 拡張がサポートされているかどうか判断し、サポートされていない場合は終了します。

        OpenCL* のすべての実装が OpenCL* と OpenGL* 間のサーフェス共有をサポートしているとは限らないため、最初のステップは拡張がシステムに存在するかどうか判断することです。ここでは、サーフェス共有をサポートするプラットフォームの拡張ストリングがあるかプラットフォームを検索しています。仕様を注意深く読むと、これはデバイス拡張ではなくプラットフォーム拡張であることが分かります。この後、コンテキストを作成するときに、コンテキストのどのデバイスが OpenGL* コンテキストを共有できるか判断します。

        このサンプルはインテル® プロセッサー・グラフィックスのみをサポートしますが、ほかの GPU をサポートするように拡張できます。検索する拡張ストリングは `cl_khr_gl_sharing` です。以下にサンプルコードを示します。

```
char extension_string[1024];
memset(extension_string, '', 1024);
status = clGetPlatformInfo( platforms[i],
                            CL_PLATFORM_EXTENSIONS,
                            sizeof(extension_string),
                            extension_string,
                            NULL);
char *extStringStart = NULL;
extStringStart = strstr(extension_string,
"cl_khr_gl_sharing");
if(extStringStart != 0){
printf("Platform does support cl_khr_gl_sharingn");
…
}
```

2. サポートしている場合、適切なデバイスオプションを渡すコンテキストを作成します。

   OpenCL* が OpenGL* とのサーフェス共有をサポートしている場合、この機能を含む OpenCL* コンテキストを作成します。Windows* では、現在の GL レンダリング・コンテキストと現在のデバイス・コンテキストにハンドルを渡します。ほかのプラットフォームでは異なるフラグをランタイムに渡す必要があることに注意してください。OpenCL* 拡張仕様の表 4.5 には、`clCreateContext()` API に渡す必要があるレンダリング・コンテキスト・フラグの説明が含まれています。`CL_WGL_HDC_KHR` は Windows* 7 および Windows* 8 用のフラグです。Mac OS X* 用のフラグは `CL_CGL_SHAREGROUP_KHR` です。これらの値を取得する方法はいくつかあり、オペレーティング・システムのドキュメントに含まれているウィンドウ API を使用する必要があります。

   Windows* では、次の例を使用しました。

```
// GL レンダリング・コンテキストを取得
HGLRC hGLRC = wglGetCurrentContext();
// デバイス・コンテキストを取得
HDC hDC = wglGetCurrentDC();
cl_context_properties cps[] =
{
    CL_CONTEXT_PLATFORM, (cl_context_properties)platformToUse,
    CL_GL_CONTEXT_KHR, (cl_context_properties)hGLRC,
    CL_WGL_HDC_KHR, (cl_context_properties)hDC,
    0
};

// コンテキストのプロパティーを使用して OCL コンテキストを作成
g_clContext = clCreateContext(cps, 1, g_clDevices, NULL, NULL,
&status);
```

3. OpenGL* と OpenCL* 間の共有をサポートするコンテキストでデバイスにキューを作成します。

   OpenCL* と OpenGL* 間の共有をサポートする特定のデバイスのコンテキストを調べます。拡張がサポートされることはすでに確認済みであるため、拡張のポインターを取得します。

```
clGetGLContextInfoKHR_fn pclGetGLContextInfoKHR =
(clGetGLContextInfoKHR_fn)
clGetExtensionFunctionAddressForPlatform(g_platformToUse,
"clGetGLContextInfoKHR");
```

次に、OpenCL* と OpenGL* 間の共有をサポートするデバイスの ID を調べます。

```
devID = pclGetGLContextInfoKHR(cps,
CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR, bytes, g_clDevices,
NULL);
```

最後に、このデバイスのアプリケーション用のコマンドキューを作成します。

```
// OpenCL* コマンドキューを作成
g_clCommandQueue = clCreateCommandQueue(g_clContext, devID, 0,
&status);
testStatus(status, "clCreateCommandQueue error");
```

`CL_DEVICES_FOR_GL_CONTEXT_KHR` を渡すテンプレートとして使用する場合、サンプルコードは多少異なることに注意してください。詳細は、参考文献 [Shevtsov 2014]のサンプルを参照してください。

2. OpenGL*: OpenCL* と共有する OpenGL* テクスチャーを作成します。

OpenGL* で、OpenCL* と共有するテクスチャーを作成します。データの更新方法に応じて、OpenGL* でテクスチャーを作成する方法はいくつかあります。詳細は、OpenGL* 仕様のドキュメントに記述されています。ここでは、2D テクスチャーを作成し、テクスチャーのストレージを作成して、データを空のままにするか初期化します。別の方法については後述します。

```
// テクスチャー・オブジェクトを作成してデータを割り当て
GLuint texture_object;
glGenTextures(1, &texture_object);
// テクスチャーをバインド
glBindTexture(GL_TEXTURE_2D, texture_object);
// テクスチャー・データにストレージを割り当て
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, tex_width, tex_height);

// データを指定
glTexSubImage2D(GL_TEXTURE_2D,
                0, // mip マップレベル; ここでは 1 レベル==0 のみ
                0, 0, // サブ領域のオフセット
                tex_width, tex_height, // サブ領域の幅と高さ
                GL_RGBA, GL_UNSIGNED_BYTE,
                texture); // 初期データを指定
```

ここでは、フィルタリング、ラッピング、ミップマッピングを行う OpenGL* 固有の API 呼び出しは省略しています。これらの呼び出しは、添付するサンプルコードで示します。OpenCL* と OpenGL* 間のサーフェス共有は、現在ミップマップをサポートしていません。ネイティブ・ミップマップは別の OpenCL* 拡張 `cl_khr_mipmap_image` で行います。

ここでは、できるだけ汎用性を持つように、フル RGBA サーフェスを指定して OpenGL* と OpenCL* 間を共有しています (API 間の最も一般的な機能を提供するため)。また、テクスチャーの一部または全体の更新には、`glTexSubImage2D()` を使用しています。この API を使用して OpenCL* に初期化したバッファーを渡し、OpenGL* と OpenCL* の両方が書き込んでいるサーフェスのシナリオをシミュレートして、OpenCL* がピクセルの一部のサブセットに書き込んだ後でも OpenGL* の書き込みが保存されるようにします。

一般的なアルゴリズムのメモリー・フットプリントを減らすには、1 つのテクスチャー・チャネル (例えば `GL_R`) のみ使用します。このケースは同じサンプルコードとワークでテストされました。ここで重要なのは、フォーマットが一致することと、高さと幅が OpenCL* カーネルで想定したワークグループのサイズと一致しているのを確認することです。最も簡単な方法は、OpenGL* でテクスチャーを作成する際に、グローバル次元 (高さと幅) と `clEnqueueNDRangeKernel()` 呼び出しのワークグループのサイズを同一にすることです。

```
size_t global_dim[2];
global_dim[0] = CL_GL_SHARED_TEXTURE_HEIGHT;
global_dim[1] = CL_GL_SHARED_TEXTURE_WIDTH;

status = clEnqueueNDRangeKernel(g_clCommandQueue, cl_kernel_drawBox,
2, NULL, global_dim, NULL, 0, NULL, NULL);
```

3. OpenCL*: ステップ 2 で作成した OpenGL* ハンドルを使用し、`clCreateFromGLTexture()` を呼び出して OpenCL* サーフェスを作成します。

```
void ShareGLBufferWithCL()
{
int status = 0;
g_SharedRGBAimageCLMemObject = clCreateFromGLTexture(g_clContext,
                                       CL_MEM_WRITE_ONLY,
                                       GL_TEXTURE_2D,
                                       0,
                                       g_RGBAbufferGLBindName,
                                       &status);
    if(status == 0)
    {
        printf("Successfully shared!\n");
    }
    else
    {
        printf("Sharing failed\n");
    }
}
```

これは OpenGL* と OpenCL* 間でサーフェスを共有するための重要な API 呼び出しです。`glGenTextureName()` に、以前作成した OpenCL* コンテキスト、OpenGL* テクスチャーの読み取り、書き込み、または読み書きの両方を行うことを記述する読み取り/書き込みプロパティー、そして以前の OpenGL* API 呼び出しで作成した名前を渡します。出力は、OpenCL* カーネルのイメージとして扱われる `cl_mem` オブジェクトです。実際に、カーネルが OpenCL* パスのみに存在するイメージで動作すれば、OpenGL* テクスチャーをセットアップするときに指定されたパラメーターと同じパラメーターが指定される限り、同じカーネルは OpenGL* のテクスチャーで動作します。

OpenCL* で共有されるサーフェスへの書き込み

1. OpenCL* 排他アクセスのためにサーフェスをマークします。

   OpenCL* がサーフェスに書き込むとき、OpenCL* へのマップ/アンマップ API 呼び出しでサーフェスをロックする必要があります。このロックにより、OpenCL* がサーフェスに書き込んでいるときに OpenGL* がコンテンツやサーフェスを変更しないことが保証されます。`cl_khr_gl_event` 拡張がサポートされていない場合、OpenGL* オブジェクトを取得する前に、OpenGL* オブジェクトにアクセスする OpenGL* 操作が完了していることを保証しなければなりません。`cl_khr_gl_event` 拡張なしでこれを保証する唯一の移植可能な方法は、`glFinish()` を呼び出すことです。この際、GL サーフェスに影響するほかのコマンドを発行しないことを保証する必要があります。

   しかし、インテル® プロセッサー・グラフィックスは `cl_khr_gl_event` 拡張をサポートしているため、以下の利点を活用できます。

   「さらに、この拡張は、OpenCL* コンテキストと同じスレッドでバインドされた OpenGL コンテキストとの同期を暗黙的に保証するため clEnqueueAcquireGLObjects と clEnqueueReleaseGLObjects の動作を変更します。」

   つまり、同期オブジェクトを作成する必要はありません。`clEnqueueAcquireGLObjects()` と `clEnqueueReleaseGLObjects()` を使用すると、明示的に同期されます。これは非常に便利で、クロス API 同期オブジェクトを作成して管理するよりもコードが簡素化され、`glFinish()` や `clFinish()` の呼び出しを使用するよりも優れたパフォーマンスが得られます。私は、これらのオブジェクトが必要ないことと暗黙的な同期セマンティクスを完全に理解する前は、このサンプルでこれらのオブジェクトを作成して管理していました。明示的な同期オブジェクトを使用する場合、イベントと同期オブジェクトの保持、解放、削除を正しく処理することが重要です。正しく処理しないと、アプリケーションが不安定になったりパフォーマンスが低下します。

   API 呼び出しは `clEnqueueAcquireGLObjects()` で、OpenCL* コマンドキューと上記のステップ 3 で作成した `cl_mem` オブジェクトで渡します。

   ```
   status = clEnqueueAcquireGLObjects(g_clCommandQueue, 1,
   &g_SharedRGBAimageCLMemObject, 0, 0, 0);
   ```

2. OpenCL* C カーネルでサーフェスに書き込みます。テクスチャー・データの場合、イメージの読み取り/書き込み関数とイメージのパスを適切に使用します。

   この例のカーネルは、OpenCL* C カーネル `drawRect()` でピクセルのサブセットのみ更新します。2 つの重要なポイントがあります。1 つ目はカーネル署名です。

   ```
   kernel void drawBox(__write_only image2d_t output)
   ```

   カーネルに渡す 2D イメージを `__write_only` 属性で宣言していることに注意してください。

   2 つ目は出力イメージへの書き込みです。

   ```
   write_imagef(output, coord, color);
   ```

イメージ出力の (u,v) 位置に値の色を書き込みます。イメージの書き込みにはサンプラーは必要ありませんが、イメージから読み取った場合はサンプラーを含むことになります。

3. サーフェスへの排他アクセスのマークを解除します。以降 OpenGL* がコンテンツを利用できるようになります。

   これは、サーフェスのロックを解除または解放して OpenGL* が更新されたコンテンツを利用できるようにする、レンダリングの前の最終的な API 呼び出しです。前述したように、インテル® プロセッサー・グラフィックスは同期オブジェクト拡張をサポートしているため、暗黙の同期保証を活用できます。テクスチャーで OpenGL* コマンドを実行する前に `clFinish()` を呼び出す必要はありません。

```
status = clEnqueueReleaseGLObjects(g_clCommandQueue, 1,
&g_SharedRGBAimageCLMemObject, 0, NULL, NULL);
```

   これらの呼び出しがサーフェスの各フレームをロック、更新、アンロックするランタイム・レンダリング・ループで適切に動作しているか確認するのは容易です。

```
void simulateCL()
{
    cl_int status;
    static float fDimmerSwitch = 0.0f;

    status = clEnqueueAcquireGLObjects(g_clCommandQueue, 1,
&g_SharedRGBAimageCLMemObject, 0, 0, 0);

    status = clSetKernelArg(cl_kernel_drawBox, 0, sizeof(cl_mem),
&g_SharedRGBAimageCLMemObject);
    testStatus(status, "clSetKernelArg");

    size_t global_dim[2];
    global_dim[0] = CL_GL_SHARED_TEXTURE_HEIGHT;
    global_dim[1] = CL_GL_SHARED_TEXTURE_WIDTH;

    status = clEnqueueNDRangeKernel(g_clCommandQueue,
cl_kernel_drawBox, 2, NULL,
global_dim, NULL, 0, NULL, NULL);

    status = clEnqueueReleaseGLObjects(g_clCommandQueue, 1,
&g_SharedRGBAimageCLMemObject, 0, NULL, NULL);
}
```

   OpenCL* C カーネル関数自体は特に重要ではなく、イメージ値のサブセットを更新しているだけです。

```
kernel void drawBox(__write_only image2d_t output, float 
fDimmerSwitch)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int xMin = 0, xMax = 1, yMin = 0, yMax = 1;

    if((x >= xMin) && (x <= xMax) && (y >= yMin) && (y <= yMax))
    {
        write_imagef(output, (int2)(x, y), (float4)(0.f, 0.f, 
fDimmerSwitch, 1.f));
    }
}
```

シャットダウン

1. CL メモリー・オブジェクトのクリーンアップ

   OpenCL* オブジェクトをクリーンアップする必要があります。ここで関連する唯一のオブジェクトは、OpenGL* テクスチャーと関連する `cl_mem` オブジェクトです。

   ```
   // すべての CL キュー、コンテキスト、プログラム、メモリー・オブジェクトをクリーンアップ
   status = clReleaseMemObject(g_SharedRGBAimageCLMemObject);
   ```

2. GL サーフェスのクリーンアップ

   ```
   glDeleteTextures(1, &g_RGBAbufferGLBindName);
   ```

   サンプルコードで示されている OpenGL* と OpenCL* のためにほかのオブジェクトをクリーンアップする必要があることに注意してください。

同期オブジェクトを使用して暗黙の同期を使用しなければコードは速くなるか？

現在のインテル® プロセッサー・グラフィックスの実装では、OpenCL* と OpenGL* 間の Flush でコンテキスト間を切り替えています。このため、明示的な同期オブジェクトを使用してもパフォーマンス上の利点はありません。また、プログラマーは明示的な同期を使用するときに OpenCL* と OpenGL* 間に Flush や Finish を挿入する必要はありません。

追加情報

サンプルコードでバーテックス配列オブジェクトを使用するには、`glewInit()` を呼び出す前に `glewExperimental` フラグを GL_TRUE にセットする必要があります。

今後の課題

このチュートリアルは、サーフェス共有の基本を扱っています。将来、チュートリアルの範囲を拡大し、ここで説明した追加の利用例を扱う予定です。

OpenCL* と OpenGL* 間の明示的な同期イベント

OpenCL* は、GLsync フェンス・オブジェクトと cl_khr_gl_event 拡張を含む OpenCL* イベント・オブジェクトを作成できます。同様に、OpenCL* イベントを OpenGL* と共有する GL_ARB_cl_event 拡張を利用して OpenCL* と共有する OpenGL* 拡張もあります。これらを使用してもパフォーマンス上の利点はありませんが、実際のコードで確認することができます。より複雑な利用例では明示的な同期が必要になることがあります。その場合は、これらのシナリオで何を行うべきか理解することが役立ちます。

共有バッファー、フレームバッファー、デプス、ステンシル、MSAA サーフェス

OpenGL* との OpenCL* サーフェス共有拡張は OpenGL* で作成できる全種類のサーフェス (バッファー、デプス、その他) の共有をサポートしますが、拡張仕様で説明されているサーフェスの形式にはいくつかの制限があります。このチュートリアルは、OpenGL* でテクスチャー・マップとして表示または使用されるテクスチャーを共有することに注目していますが、ほかのサーフェスの共有も同様に動作します。OpenGL* テクスチャーとバッファーを共有するためのガイドラインは、インテル® プロセッサー・グラフィックスでも同様です。API 間でサーフェスを共有する場合、OpenGL* ではバーテックス・バッファー・オブジェクトやピクセル・バッファー・オブジェクトは作成しません。

ダブル・バッファリング

ダブル・バッファリング手法を使用する場合、その複雑さとパフォーマンスのトレードオフを調査すべきです。このチュートリアルでは、暗黙の同期を使用したサーフェス共有の機能と基本に注目しました。

サーフェス共有がサポートされていない場合

このユースケースでは、Maxim Shevtsov が公開している、OpenCL* と OpenGL* でコピーを行うケースをカバーしたサンプルコード (参考文献セクションを参照) を参照することを推奨します。例えば、現在、インテルは Linux* 上でのサーフェス共有をサポートしていません。顧客からのリクエストが増えれば、サポートされる可能性があります。

サーフェス共有の例

サーフェス共有を説明するため、このチュートリアルを作成し、 OpenCL* 2.0 および OpenGL* 4.3 ドライバーを使用してインテル® プロセッサー・グラフィックス上でテストしました。しかし、スタートアップ・コードにわずかな移植用の変更を加えることで、さらに多くのプラットフォームやデバイスで動作するようになります。OpenGL* のプログラム可能なバーテックス・シェーダーとピクセルシェーダーは、OpenGL* の以前のバージョンでも同様に動作するはずです。OpenCL* C カーネルは非常に単純ですが、この記事の本質を明白に示します。

必要な設定

チュートリアルをビルドするには、特定のライブラリーをダウンロードしてインクルード・パスとライブラリー・パスに適切なパスを設定する必要があります。これらのライブラリーの URL は、参考文献に含まれています。

- freeglut.h、freeglut.dll - パスにこの動的リンク・ライブラリーの場所を追加します。freeglut バイナリーのダウンロード先は参照文献に含まれています。ダウンロード後、freeglut.dll を freeglut\bin ディレクトリーに展開します。

- `glew.h`、`glew32s.lib` - GLEW 1.11.0 のファイルで、OpenGL* API と拡張管理をすべて制御します。glew.h をインクルードする前に #define GLEW_STATIC を追加してください。
- `cl.h`、`cl_gl.h`、`openCL.lib` - インテル® OpenCL* SDK のファイルです。
- `gl.h` - \Windows Kits\8.0 ディレクトリーに含まれています。

ここでは以下の設定を利用しましたが、環境により多少異なることがあります。

- (プロジェクト・レベルではなく) ソリューション・レベルでチュートリアルの debug/release ディレクトリーに freeglut.dll をコピーしました。ほかにも .dll を制御する方法はありますが、この方法で問題なく動作しました。
- glew32s.lib ライブラリーのパスを設定します。
    - `C:\src\glew-1.11.0\lib\Release\Win32`
- インクルード・パスに cl.h と cl_gl.h の場所を追加します。例:
    - `C:\Program Files (x86)\Intel\OpenCL SDK\3.0\include\CL`
- ライブラリー・パスに OpenCL* ライブラリーの場所を追加します。例:
    - `C:\Program Files (x86)\Intel\OpenCL SDK\3.0\lib\x86`
- 静的リンク・ライブラリーに OpenCL.lib を追加します。

サンプルファイルとディレクトリー構造

サンプルコードについて説明しましょう。ここでは、製品品質の実装ではなく、単純な C サンプルを作成することを目的としています。

プロジェクトは、`CL_20_GL_43_surface_sharing` ディレクトリーに含まれています。これは、OpenCL* 2.0 で OpenGL* 4.3 ソリューションを使用してテストしたことを示しています。OpenCL* 2.0 固有の機能は特に使用していません。実際、OpenCL* カーネルをコンパイルするときに OpenCL* 2.0 コンパイラーを有効にすることすら行いませんでした。OpenGL* では、非常に単純な、プログラム可能なバーテックス・シェーダーとピクセルシェーダーを使用します。`#version` を使用し、シェーダー用に少なくともバージョン 3.3 をリクエストしました。

ファイル構成は次のとおりです。

- `main.cpp`: コードのメイン・エントリー・ポイントと、OpenCL*、OpenGL*、ウィンドウシステムを初期化する呼び出しを含みます。このファイルには、Esc キーイベントを制御するキーボードハンドラーも含まれています。
- OpenGL* と OpenCL* 固有の API は、それぞれ別のファイル、`OGL.h` と `OGL.cpp` および `OCL.h` と `OCL.cpp` にあり、使用される共通のフラグと変数は `commonCLGL.h` にあります。
- 共有イメージにボックスを描く OpenCL* シェーダーはファイル `OpenCLRGBAFile.cl` にあります。OpenGL* シェーダーは `triangles.frag` および `triangles.vert` にあります。

サンプルのビルドと実行

メインメニューから **[ビルド] > [ソリューションのビルド]** を選択してサンプルコードをビルドすると、すべての実行ファイルが生成されます。実行ファイルは、Visual Studio* ディレクトリーから実行することも、`CL_20_GL_43_surface_sharing` ソリューション・ファイルと同じ場所にある Debug/Release ディレクトリーから実行することもできます。

サンプルを実行するには、Visual Studio IDE で F5 キーを押します。**コマンドラインから実行する場合は、プロジェクト・ディレクトリーから実行ファイルのディレクトリーにシェーダーをコピーする必要があります。**3

つの関連カーネルとシェーダーファイルは、`OpenCLRGBAFile.cl`、`triangles.frag` および `triangles.vert` です。

その他の情報

Maxim Shevtsov が作成したサンプルコードを含むサーフェス共有のチュートリアルは、https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial で紹介されています。PBO の使用と `glMapBuffer()` の使用のトレードオフについて詳しく説明し、サンプルコードでこの機能をサポートしています。また、Linux* OpenCL* 実装のように拡張がサポートされないケースの制御方法についても示しています。サンプルコードは、従来の win32 API でウィンドウを作成し、固定関数 OpenGL* グラフィックス・パイプラインを使用して、サーフェスの色を変更しています。さらに、インテルの CPU を含む追加のプラットフォームで実行することにも労力をかけ、いくつかの優れた追加の情報を提供しています。

このチュートリアルのサンプルコードでは、`freeglut`、`glew`、OpenGL* 4.3 のプログラム可能なバーテックス・シェーダーとピクセルシェーダー、表示用にレンダリングされた画面向けの多角形サーフェスのテクスチャー・マッピングを使用しています。ポイントは、生成されるピクセルカラーが協調して動作する OpenGL* と OpenCL* パイプラインのピクセルごとのすべての操作の組み合わせであることを明確にするため、この記事のサンプルは OpenCL* カーネルのテクスチャー部分にのみ書き込んでいることです。

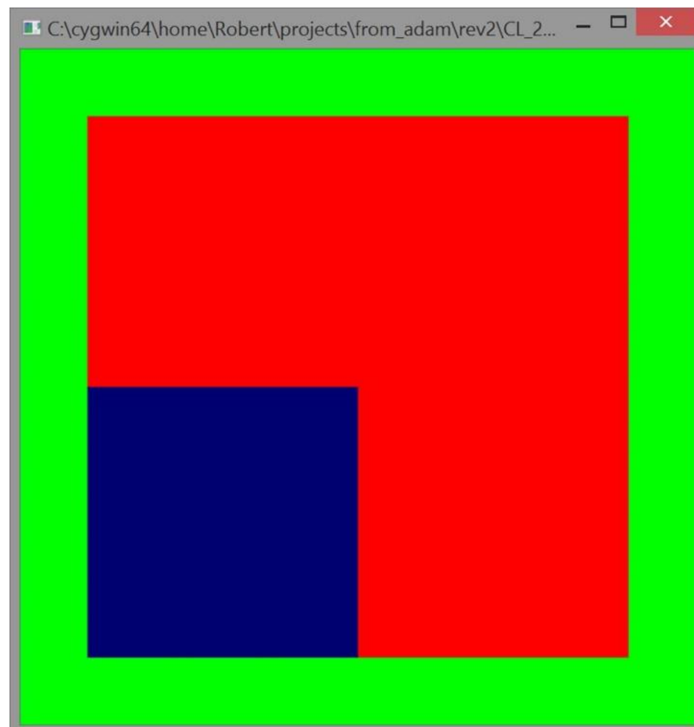

図 2. サンプル実行の予想結果。緑は背景のクリアカラーです。イメージは、2 つのテクスチャーがマップされた三角形で作成された画面向きの四辺形です。テクスチャーは、OpenGL* により生成された後 OpenCL* により書き込まれるテクセルの左下部分を含む小さな赤の 4x4 テクスチャーのマップです。OpenCL* は、黒から青 (青チャネルで 0 から 255) に変化する青のカラーチャネルに値を書き込みます。

謝辞

Murali Sundaresan、Aaron Kunze、Allen Hux、Pavan Lanka、Maxim Shevtsov、Michal Mrozek、Piotr Uminski、Stephen Junkins、Dan Petre、Ben Ashbaugh の皆さんに感謝します。技術的な議論、説明、レビューを行うことができたのは皆さんのおかげです。

著者紹介

Adam Lake – 標準化団体 Khronos* Group のビジュアル・プロダクト・グループのシニア・グラフィックス・アーキテクトで、議決権保有メンバーです。12 年以上の GPGPU プログラミング経験があり、これまで VR、3D、グラフィックス、ストリーム・プログラミング言語コンパイラーに関わってきました。

Robert Ioffe は、インテル コーポレーションのソフトウェア & ソリューション・グループのテクニカル・コンサルティング・エンジニアです。Khronos* の標準化作業に深くかかわっており、これまでに最新機能のプロトタイプの作成やインテル® アーキテクチャーでの動作検証を行ってきました。

参考文献 (英語)

1. OpenCL* 1.2 仕様: https://www.khronos.org/registry/cl/
2. OpenCL* 2.0 仕様 (OpenCL* C Language Specification、OpenCL* Runtime API Specification、OpenCL* Extensions Specification の 3 つで構成される): https://www.khronos.org/registry/cl/specs/
3. Stephen Junkins のホワイトペーパー「The Compute Architecture of Intel® Processor Graphics Gen7.5」: https://software.intel.com/sites/default/files/managed/f3/13/Compute_Architecture_of_Intel_Processor_Graphics_Gen7dot5_Aug2014.pdf。インテル® プロセッサー・グラフィックスで OpenCL* を使用する場合は必読。
4. Adam Lake の「ゼロコピー」サーフェス共有チュートリアル: https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics
5. Maxim Shevtsov の相互運用性チュートリアル: https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial
6. freeglut のソース: http://freeglut.sourceforge.net/
7. freeglut.dll: www.transmissionzero.co.uk/software/freeglut-devel/
8. GLEW ライブラリー: http://glew.sourceforge.net/
    1. 次の記事にも注意: http://stackoverflow.com/questions/13558073/program-crash-on-glgenvertexarrays-call

定義

下記に、このチュートリアルで使用されている用語の定義を示します。詳細は、参考文献セクションを参照してください。

- **バッファー**: OpenCL* は、バッファーとイメージを区別します。OpenCL* バッファーはメモリーで**線型的に**レイアウトされます (配列として見なすことができます)。
- **テクスチャー**: **タイル** 形式でレイアウトされたデータのバッファーで、OpenGL* のオンダイサンプラー経由で読み取ります。このメモリーレイアウトは、事前に指定したフィルターカーネル経由でメモリーから読み取った入力ピクセルをフィルタリングするテクスチャー・サンプラーによりパフォーマンスを向上させます。
- **サーフェス**: バッファー、テクスチャー、またはイメージです。レイアウトでタイルまたは線形になるメモリーのデータの一般的な用語です。場合によっては、次元、高さ、幅、データレイアウトの属性など、追加のデータが含まれます。これらの属性は API (OpenCL*、OpenGL*、DirectX*、その他) により管理されます。
- **サンプラー**: OpenCL* のイメージと OpenGL* のテクスチャーからの読み取りに使用されます。フィルタリングするときのパフォーマンスを向上するため、サンプラーは内部キャッシュとメモリーでタイ

ル型レイアウトのイメージまたはテクスチャーを利用します。サンプラーは、いくつかのテクセルと (おそらく) mip マップレベルからサンプリングを同時に実行し、単一のリクエストに対して単一のテクセル値を出力するためのキャッシュとロジックを含んでいます。

- **イメージ**: タイル形式でレイアウトされたデータのバッファーで、OpenCL* のオンダイサンプラー経由で読み取ります。OpenGL* のテクスチャーの同等物です。共有/サポートされるイメージ形式とテクスチャー形式は実装依存です。
- **サーフェス共有**: **クロス API サーフェス共有**の省略形。ある API におけるサーフェスの作成と別の API におけるデータの使用を参照するために使用されます。目的は、同じサーフェスの複数のコピーを作成することを最小限に抑えることですが、デバイスのセットに依存する制限に従わなければ正確ではありません。このチュートリアルは、インテル® プロセッサー・グラフィックスでの制限について記述しています。
- **テクスチャー・マッピング**: メモリーのピクセルとグラフィックス・パイプラインのポリゴン (多角形) との関連付け。この例では、表示用に OpenGL* テクスチャーを 2 つの画面向けポリゴンにテクスチャー・マップします。
- **ゼロコピー**: ホスト (CPU) とデバイス (GPU) に不正確に適用される技術的な用語です。このチュートリアルでは、OpenGL* と OpenCL* のコマンドストリームの実行間でテクスチャー (イメージ、バッファー、その他) のコピーが必要ないことをゼロコピーと表現しています。これは、それらがサーフェスの同じストレージ場所と互換性のあるパラメーター化を共有するためです。ゼロコピーの考慮すべき点は、バッファーのサイズに比例するストレージのリダクションと、サーフェスの実際のストレージが共有されないシステムでのコピーの省略によるパフォーマンスの向上です。
- **共有物理メモリー**: ホストとデバイスが同じ物理 DRAM を共有すること。ホストとデバイスが同じ仮想アドレスを共有する、共有**仮想**メモリーとは異なります (共有仮想メモリーはこの記事では取り上げません)。ゼロコピーを有効にするには、CPU と GPU が共有**物理**メモリーを使用する必要があります。共有物理メモリーと共有仮想メモリーは相互排他ではありません。デバイスは、共有物理メモリーをサポートすると、物理メモリー全体を見ることはできません。
- **インテル® プロセッサー・グラフィックス**: 現在のインテル® グラフィックス・ソリューションを指します。SoC 内蔵 GPU には、インテル® Iris™ グラフィックス、インテル® Iris™ Pro グラフィックス、インテル® HD グラフィックスなどがあります。ハードウェア・アーキテクチャーの詳細は、「参考文献」にある「The Compute Architecture of Intel® Processor Graphics Gen7.5」または http://ark.intel.com/ (英語) 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