OpenCL* 2.0 のジェネリック・アドレス空間

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「The Generic Address Space in OpenCL™ 2.0」の日本語参考訳です。

はじめに

OpenCL* 2.0 の新機能の 1 つに、**ジェネリック・アドレス空間**があります。OpenCL* 2.0 以前は、ポインターが宣言されるときやポインターが関数の引数として渡されるときに、そのポインターが指すアドレス空間をプログラマーが指定する必要がありました。OpenCL* 2.0 では、ポインター自体はプライベート・アドレス空間に残りますが、ポインターが指すデフォルトのアドレス空間が (ジェネリック・アドレス空間内の任意の名前付きアドレス空間を指すことができることを意味する) **ジェネリック**に変更されました。この機能を有効にするにはコンパイルフラグを追加します。OpenCL* C 1.2 向けのプログラムは変更なしでそのままコンパイルできます。

具体的に説明するため、関数と変数宣言の新しい構文の概要を示します。OpenCL* 1.2 では、以下のようにコードを記述します。

```
void foo(global unsigned int *bar)
// bar のグローバルアドレス空間、追加のコンパイルフラグなしで
// OCL 1.2 と OCL 2.0 の両方で動作
{
    local unsigned int *temp = NULL;
    // temp のローカルアドレス空間、追加のコンパイルフラグなしで
    // OCL 1.2 と OCL 2.0 の両方で動作
}
```

OpenCL* 2.0 では、以下のコードはグローバル、ローカル、またはプライベート・アドレス空間を指すポインターで動作します。

```
void foo(unsigned int *bar) // OCL 2.0、bar のアドレス空間なし
{
    unsigned int *temp = NULL; // OCL 2.0、temp のアドレス空間なし
}
```

オプションの clBuildProgram() や clCompileProgram() を利用する場合、"–cl-std=CL2.0" フラグを指定して OpenCL* 2.0 の機能を有効にすることを忘れないようにしてください。フラグを指定しない場合、以下のエラーが表示されます。

```
1:54:24: error: passing '__local unsigned int *' to parameter of type
'unsigned int *' changes address space of pointer
```

これは、下位互換性のために OpenCL* 2.0 ではデフォルトでプログラムを OpenCL* 1.2 プログラムとしてコンパイルするためです。

ジェネリック・アドレス空間とは?

ジェネリック・アドレス空間は、**ローカル**、**グローバル**、**プライベート**・アドレス空間をカプセル化した抽象的なアドレス空間です。実用的な理由により、**定数**アドレス空間は含まれていません。ジェネリック・アドレス空間は、Embedded C 仕様のジェネリック・アドレス空間の流れを汲むものです。

「ジェネリック・アドレス空間に加えて、実装では別の、名前付きアドレス空間もサポートします。これらの代替アドレス空間にはオブジェクトが割り当てられ、これらのアドレスのオブジェクトを指すポインターが定義されます。アドレス空間のポインターは、そのアドレス空間のアドレス範囲をサポートするために必要最低限のものとなります。」

一部のアーキテクチャーではアドレス空間のポインターサイズやメモリーバンクが異なるため、既存の名前付きアドレス空間を排除する必要はありません。しかし、既存の名前付きアドレス空間が必要ないときに関数の特殊化を必要としないプログラムを記述する手段が必要です。これらの利用例のいくつかは後述します。性能上の理由により、それぞれのアドレス空間に対する特殊化を引き続きサポートしますが、プログラマーが異なるアドレス空間を操作するコードの単一セグメントを記述できるようにします。

ジェネリック・アドレス空間に関する重要なポイント:

- ホストから呼び出されるカーネルの引数はアドレス空間で修飾する必要があります。
- 関数とカーネルの引数はプライベート・アドレス空間に残りますが、変更されたアドレスを**指して**います。この項目は重要かつ微妙であるため、3 度は読み返してください! OpenCL* C 2.0 仕様のセクション 6.5 には多くの例が含まれています (「参考文献」を参照)。
- 2 つの異なるアドレス空間からの null ポインターは、これらのポインターの 1 つがジェネリックである限り、等しいと評価されます。
- null ポインターがあるアドレス空間から別のアドレス空間に変換された場合、null ポインターはその型の null ポインターになります。
- アドレス空間は抽象的な意味では分離していると見なされますが、いくつかの実装ではオーバーラップしている、または物理メモリーの一部であるものとして扱います。これは完全に理に適っています。
- グローバル、ローカル、プライベート、定数は、それぞれ __global、__local、__private、__constant と交換可能に使用できます。ジェネリックは名前なしアドレス空間であり、OpenCL* 2.0 にはそのようなキーワードはありません。

ジェネリック・アドレス空間を有効にする

前述したように、OpenCL* C 2.0 のジェネリック・アドレス空間機能を有効にするには、オプション clBuildProgram() または clCompileProgram() とともに "-cl-std=CL2.0" フラグを指定します。フラグを指定しなかった場合、プログラムは OpenCL* 1.2 プログラムとしてコンパイルされます。まだ「動作する」古いプログラムを新しい OpenCL* 1.2 ランタイムへ移行したり、共有仮想メモリーやジェネリック・アドレス空間のような新しい OpenCL* 2.0 機能への移行をより簡単に行えます。

ジェネリック・アドレス空間を使用する理由

プログラマーが入力ポインターのアドレス空間に関係なく同じ関数を使用する場合、アドレス空間**を指す**すべてのポインターを修飾する必要性をなくすことにより、ジェネリック・アドレス空間を使用して OpenCL* プログラムをより簡単に記述できます。例えば、ヒストグラム値のインクリメント、配列への値のセットの追加やソートなどのケースが考えられます。OpenCL* C 1.2 仕様では、これらすべてのケースで、関数が利用すると思われるアドレス空間ごとに関数のバージョンを記述する必要があります。これは、バージョン管理のリスクを高めます。開発者は 1 つのコードセグメントに変更を加える際、同じ機能を持つ複数のバージョンを保守しなければいけません。新しいジェネリック・アドレス空間では、アドレス空間ですべてのポインターを修飾する**必要**がなくなりますが、アドレス空間を削除する**必要**はないため、古い OpenCL* 1.2 コードもすべて記述したとおりに引き続き動作します。アドレス空間を指定することで利益を得ているプログラムは、プログラマーがアドレス空間を宣言することで引き続き利益を得られます。

ジェネリック・アドレス空間を利用することを選択した別の理由は、CPU と GPU の C コードセグメント (例えば、ホストとデバイスの両方で使用するデータ構造やコア関数のセット) のクロスコンパイルをより簡単にするためです。ジェネリック・アドレス空間対応のコードがない場合、プログラマーはアドレス空間を空白などに変換してホストまたはデバイス・コンパイラーでアドレス空間キーワードを処理できるようにする必要があります。すべての問題に対処できるわけではありませんが、異なる C コンパイラーで同じコードをコンパイルするより簡単になります。

ジェネリック・アドレス空間でいくつかの操作を実行する

プログラマーがジェネリックな手法で操作する関数を記述できる場合でも、特定のアドレス空間にいくつかの操作が必要です。ヒストグラムの場合、ローカルメモリーの値のインクリメントではアトミック操作は必要ありませんが、ワークグループ間でグローバルに共有されるヒストグラムの値のインクリメントではグローバルメモリー上のアトミック操作が必要になります。場合によっては、これらの部分に利用できるビルトイン関数が用意されています。関数 to_global()、to_local()、to_private() は、それぞれのアドレス空間のポインターをキャストするために使用できます。なんらかの理由により、これらの関数がそれぞれのアドレス空間のポインターをキャストできない場合、NULL が返されます。プログラマーは、この結果を見て、各アドレス空間を指すようにポインターを利用できるかどうか知ることができます。

これらの関数が、is_local()、is_global()、is_private() の値に基づくブール値を返さないのか不思議に思われるかもしれません。その理由は、いくつかの実装では 1 つのアドレス空間を別のものとして扱うことができ、要求されたアドレス空間に存在するかのように扱うことができる場合は、ポインター値を返すことが理にかなっているためです。例えば、CPU はすべてのアドレス空間にこの操作を行います。

アドレス空間のキャスト

ある名前付きアドレス空間から別の名前付きアドレス空間へのキャストはできません。名前付きアドレス空間のポインターのアドレス空間はジェネリック・アドレス空間に割り当てることが**できます**が、その逆はできま

せん。単一のジェネリック・ポインターは同じコードシーケンスで異なる名前付きアドレス空間に割り当てることができます。定数アドレス空間を指す変数はジェネリック・アドレス空間のメンバーに変換できません。

次の例は許されます。lp はローカルアドレス空間のポインター、g はジェネリック・アドレス空間を指すプライベート・メモリーのポインターでローカルアドレス空間を指すポインターに割り当てられています。

```
local int *lp;

int *g;

g = lp; // 成功!
```

しかし、次の例は許されません。ローカルアドレス空間を指すプライベート・メモリーにあるポインターをジェネリック値に割り当てようとしているため、エラーになります。

```
local *lp;

lp = p; // エラー!
```

OpenCL* C 仕様のセクション 6.5 () には、有効なキャストと無効なキャストに関するさまざまな例が含まれています (「参考文献」を参照)。

パフォーマンスの留意事項と対処方法

いくつかの実装では、コンパイル時に指されたアドレス空間をコンパイラーが解決できない場合、関数のパフォーマンスが低下することがあります。この問題を回避するには、固有のアドレス空間に関連するポインターを修飾して、コンパイラーがジェネリック・アドレス空間を処理する追加のコードを生成しないようにします。コンパイル時にアドレス空間を解決できない場合、ジェネリック・アドレス空間は極めて小さなカーネルでも多少パフォーマンスへの影響があります。コンパイラーがプログラマーに引数のアドレス空間を解決できなかったことを通知するのが理想的ですが、現在のところ、この機能をサポートしているコンパイラーはありません。また、ほとんどのカーネルは、このコストがカーネルの実行時間に比例して償却されるようにアドレス空間を解決するため、実行時に必要な追加の命令よりも多くの命令でなければなりません。

動作サンプル

この機能を説明するため、OpenCL* 1.2 で正しく動作する短いコードシーケンスと OpenCL* 2.0 のジェネリック・アドレス空間を使用する同じコードシーケンスを記述しました。グローバルまたはローカルアドレス空間のいずれかのバッファーセグメントからグローバルバッファーへの memcpy() を行う単純な統合カーネルを使用します。値を要素ごとにグローバル・メモリー・バッファーに戻す前に算術演算を行うカーネルを想像することは簡単です。例えば、RGB イメージから YUV への変換では、入力値を乗算および加算します。データの演算セットは同じですが、唯一の違いは入力バッファーのアドレス空間になります。RGB から YUV に変換する前にグローバルメモリーにバッファーを残すことでパフォーマンスが向上するマシンがあるケースや、追加の演算を行う前にローカルメモリーにイメージをタイルするような実装と同様に、これはジェネリック・アドレス空間の使用に適した候補です。

グローバルメモリーからロードしてグローバルメモリーの別の場所に書き込む OpenCL* 1.2 で記述された単純な memcpy() 関数は、次のように記述できます。

```
void GlobalXToY_internal(__global unsigned char *in, __global unsigned
char *out, unsigned int startFrom, unsigned int startTo, unsigned int
length)
{
    for(int i = startFrom; i < startFrom + length;)
    {
        out[startTo++] = in[i++];
    }
}
```

グローバルまたはローカルアドレス空間の値を入力として受け付ける、OpenCL* 2.0 の同等の機能を実現するには、以下のように記述します。最初の関数の引数からキーワード __global が除かれていることに注意してください。

```
void GenericXToY_internal(unsigned char *in, unsigned char *out, unsigned
int startFrom, unsigned int startTo, unsigned int length)
{
    for(int i = startFrom; i < startFrom + length;)
    {
        out[startTo++] = in[i++];
    }
}
```

サンプルは、OpenCL* 2.0 をサポートするインテル® プロセッサー・グラフィックスを搭載したインテル® プロセッサー上でコンパイル、実行できることを確認しています。現在のところ、ほかの実装ではテストしていません。

今後の課題

今後、この memcpy() の例に、より複雑なワークロード (例えば、タイル型メモリー操作や、より複雑な行列乗算) を追加する予定です。また、さらに多くの OpenCL* 2.0 実装を調査して、それらのプラットフォームに対応する予定です。ジェネリック・アドレス空間を使用するコストを確認する解析を行うと、パフォーマンスに関してより大きな保証が得られるでしょう。

謝辞

最初のジェネリック・アドレス空間案の共著者である Dillon Sharlet、開発中に多大な貢献をしてくれた Ben Ashbaugh、Stephen Junkins、Ben Gaster、その他の方々に感謝します。OpenCL* 仕様に追加する前に機能の適切な解析を行うために尽力してくれた Khronos のほかのベンダーにも感謝します。

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。

以下の記事も参照してください。

Optimizing Simple OpenCL Kernels: Modulate Kernel Optimization

Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization

GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions

Sierpiński Carpet in OpenCL 2.0

参考文献 (英語)

Embedded C: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf

共有仮想メモリーのサンプル: https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-code-sample

その他のサンプル: https://software.intel.com/en-us/intel-opencl-support/product-library

インテル® SDK for OpenCL* Applications: https://software.intel.com/en-us/intel-opencl

OpenCL* API 仕様: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

OpenCL* C 仕様: https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf

著作権と商標について

サンプルのダウンロード

generic address space.zip (15.5 KB)

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