パート 3: Ubuntu* で SYCL* 開発向けに oneAPI、DPC++、Visual Studio* Code を設定

この記事は、Codeplay Blogs に公開されている「Setting up SYCL™ development with oneAPI, DPC++ and Visual Studio® Code on Ubuntu」の日本語参考訳です。原文は更新される可能性があります、原文と翻訳文の内容が異なる場合原文を優先してください。

2023 年 3 月 1 日

本シリーズの記事:

- パート 1: DPC++ と Visual Studio* Code による SYCL* のデバッグ
- パート 2: Ubuntu* で C++* 開発向けに Visual Studio* Code を設定

パート 3 では、Ubuntu* 上の Visual Studio* Code IDE で C++ および SYCL* コードを記述し、DPC++ コンパイラーでコンパイルし、デバッグできるように設定する方法を紹介します。

> **注:**
> DPC++ と **dpcpp** は、インテルの Clang コンパイラーの名称です。インテルの **icpx** コンパイラーは、dpcpp に代わる新しいコンパイラーで、DPC++ コンパイラーとも呼ばれます。**-fsycl** コンパイラー・オプションを指定することで、dpcpp コンパイラーを互換的に置き換えます。
>
> Clang LLVM コンパイラーと LLDB デバッガーは、それぞれ gcc コンパイラーと gdb デバッガーをドロップインで置換します。
>
> DPC++ と **dpcpp** は、Clang コンパイラーの SYCL* バージョンの名前です。
>
> dpcpp コンパイラーは、oneAPI バージョン 2023.0.0 で非推奨となり、icpx に置き換わりました。icpx コンパイラーは、ビルド時に **-fsycl** コンパイラー・オプションを指定して SYCL* を含めるようコンパイラーに通知することで、dpcpp のドロップイン置換として使用できます。icpx コンパイラーは、DPC++ コンパイラーとも呼ばれます。
>
> インテル® クラシック・コンパイラーは icc (Linux*) または icl (Windows*) と呼ばれます。
>
> CMake または Makefile ビルド構成を使用する C/C++ プロジェクト (フォルダー) は、Visual Studio* Code の Microsoft* C/C++ 拡張プロジェクト (フォルダー) と組み合わせて使用でき、競合することはありません。

Visual Studio* Code で [拡張機能] パネルを開き、利用可能なインテル® oneAPI 拡張を検索して、以下の拡張をインストールします。

- Code Sample Browser for Intel oneAPI Toolkits
- Analysis Configurator for Intel oneAPI Toolkits
- Environment Configurator for Intel oneAPI Toolkits

Visual Studio* Code の C/C++ プロジェクトを DPC++ プロジェクトに変換する準備

> **注:**
> Visual Studio* Code 用のインテル® oneAPI サンプルコードの多くは、CMake または Makefile ベースのビルド構成を使用しています。これらのサンプルは、Visual Studio* Code のターミナルウィンドウからコンパイルおよび実行されますが、Visual Studio* Code のビルドおよびデバッグ用の C/C++ 拡張機能は使用しません。

パート 2 の標準 C/C++ **Helloworld** プロジェクトをモデルとして、シンプルなインテル® oneAPI サンプルコードを取得し、Microsoft* C/C++ 拡張機能のビルドおよびデバッグ構成を使用する別の C/C++ プロジェクトを作成します (既存の CMake または同様の構成を削除します)。プロジェクトの各種構成ファイルは、インテルの **icpx** コンパイラーと **gdb-oneapi** デバッガーを代わりに使用するように編集します。このプロジェクトでは、Helloworld プロジェクトで行ったのと同じ方法で、コンパイルとデバッグを行います。

ここでは、**Simple Add** サンプルコードを使用します。このサンプルコードは、Helloworld と同等のシンプルな DPC++ プログラムで、2 つの異なる SYCL* メモリーモデルを使用して同じ結果を出力する方法を示します。プロジェクトには、それぞれのモデルに対して 1 つずつ、2 つの .cpp ファイルが含まれます。

> **注:**
>
> Visual Studio* Code プロジェクトを将来にわたって動作させるためには、すべてのパスのバージョンテキストを「最新」に置き換えます。
>
> この C/C++ プロジェクトで使用されている icpx コンパイラー・オプションのほとんどは、インテルのサンプルの Makefile から取得したものです。

DPC++ と同等の C/C++プロジェクトを作成するには、以下の操作を行います。

1. 新しいプロジェクト・フォルダーを作成します。この例では、**VSCodeDpcppSimpleAdd** という名前を付けます。
2. **bin** と **src** という名前のサブフォルダーを作成します。
3. Simple Add サンプルの .cpp ファイルを両方とも src フォルダーにコピーします。
4. ターミナルウィンドウを開いて、プロジェクト・フォルダーのトップに移動します。
5. ターミナルウィンドウに `code .` を入力して、このフォルダーの新しい Visual Studio* Code C/C++ プロジェクトを作成します。
6. Visual Studio* Code の [エクスプローラー] ペインで、.cpp ファイルのいずれかを選択します。

7. Visual Studio* Code は IntelliSense の設定を求めるポップアップを表示する場合があります。IntelliSense の設定は、**c_cpp_properties.json** ファイルに含まれています。このファイルは Visual Studio* Code によって作成されます。作成されない場合は、コマンドパレット (Ctrl + Shift + p) で **C/C++: 構成の編集**と入力して選択します。
8. c_cpp_properties.json ファイルを編集して、必要に応じて、図 10 に示すオプションに置き換えます。
9. .cpp ファイルのいずれかを選択して、メニューから **[ターミナル] > [タスクの構成…]** を選択します。
10. コンパイラーのドロップダウン・リストから、インテル® コンパイラー (**/opt/…/icpx**) を選択します。Visual Studio* Code によって新しい **task.json** ファイルが作成されます。
11. task.json ファイルを編集して、図 11a、11b、11c、および 11d に示す 4 つのビルド構成を含めます。これらは 4 種類の実行可能なタスク、つまり、リリース構成とデバッグ構成を持つ 2 つのプログラムです。
12. 環境変数を使用して 4 つの実行ファイルの名前を指定します。図 12 に示すように **settings.json** で指定します。
13. 4 つのビルド構成すべてをテストして、正常にコンパイルされることを確認します。**Ctrl + Shift + b** キーを押して、ビルドする構成を選択します。
14. ターミナルウィンドウを開くか、Visual Studio* Code のターミナルウィンドウを使用して、プロジェクトの bin フォルダーにある 4 つのバイナリーを実行します。

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [],
            "compilerPath": "/opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp",
            "cStandard": "gnu17",
            "cppStandard": "gnu++17",
            "intelliSenseMode": "linux-gcc-x64"
        }
    ],
    "version": 4
}
```

図 10: DPC++ 向けに構成された IntelliSense ファイル c_cpp_properties.json

```
{
    "type": "cppbuild",
    "label": "simple-add-usm Debug C/C++: dpcpp build active file",
    "command": "/opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp",
    "args": [
        "-fdiagnostics-color=always",
        "-g",
        "${workspaceFolder}/src/${config:programNameUsm}.cpp",
        "-o",
        "${workspaceFolder}/bin/${config:programNameUsm}_d"
    ],
    "options": {
        "cwd": "${workspaceFolder}"
    },
    "problemMatcher": [
        "$gcc"
    ],
    "group": "build",
    "detail": "compiler: /opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp"
},
```

図 11a: simple-add-usm の DPC++ デバッグ・ビルド・タスク

注:
Clang コンパイラー・オプション **–fno-limit-debug-info** は、デバッガーで文字列変数を確認する際に変数値を表示します。

```
{
    "type": "cppbuild",
    "label": "simple-add-usm Release C/C++: dpcpp build active file",
    "command": "/opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp",
    "args": [
        "-DNDEBUG",
        "${workspaceFolder}/src/${config:programNameUsm}.cpp",
        "-o",
        "${workspaceFolder}/bin/${config:programNameUsm}"
    ],
    "options": {
        "cwd": "${workspaceFolder}"
    },
    "problemMatcher": [
        "$gcc"
    ],
    "group": "build",
    "detail": "compiler: /opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp"
},
```

図 11b: simple-add-usm の DPC++ リリース・ビルド・タスク

```
{
    "type": "cppbuild",
    "label": "simple-add-buffers Debug C/C++: dpcpp build active file",
    "command": "/opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp",
    "args": [
        "-fdiagnostics-color=always",
        "-g",
        "${workspaceFolder}/src/${config:programNameBuffers}.cpp",
        "-o",
        "${workspaceFolder}/bin/${config:programNameBuffers}_d"
    ],
    "options": {
        "cwd": "${workspaceFolder}"
    },
    "problemMatcher": [
        "$gcc"
    ],
    "group": "build",
    "detail": "compiler: /opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp"
},
```

図 11c: simple-add-buffer の DPC++ デバッグ・ビルド・タスク

```
{
    "type": "cppbuild",
    "label": "simple-add-buffers Release C/C++: dpcpp build active file",
    "command": "/opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp",
    "args": [
        "-DNDEBUG",
        "${workspaceFolder}/src/${config:programNameBuffers}.cpp",
        "-o",
        "${workspaceFolder}/bin/${config:programNameBuffers}"
    ],
    "options": {
        "cwd": "${workspaceFolder}"
    },
    "problemMatcher": [
        "$gcc"
    ],
    "group": "build",
    "detail": "compiler: /opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp"
}
```

図 11d: simple-add-buffer の DPC++ リリース・ビルド・タスク

```
{
    "programNameUsm": "simple-add-usm",
    "programNameBuffers": "simple-add-buffers"
}
```

図 12: プロジェクトの環境変数定義ファイル settings.json

プログラムをデバッグするため Visual Studio* Code の DPC++ プロジェクトを準備

サンプルの 4 つのバリエーションがすべてコンパイルされ実行されることを確認したら、デバッグセッションの設定をプロジェクトに追加します。

launch.json ファイルにデバッグ構成を追加するには、次の操作を行います。

1. [エクスプローラー] ペインで **simple-add-usm.cpp** ファイルを選択します。
2. コマンドパレットで **C/C++: デバッグ構成の追加**と入力して選択し、リストから **simple-add-usm debug** を選択します。
3. launch.json ファイルを編集して、図 13a と図 13b に示す 2 つの構成を追加します。

```
"configurations": [
{
    "name": "C/C++: dpc++ build and debug simple-add-usm",
    "type": "cppdbg",
    "request": "launch",
    "program": "${workspaceFolder}/bin/${config:programNameUsm}_d",
    "args": [],
    "stopAtEntry": true,
    "cwd": "${fileDirname}",
    "environment": [],
    "externalConsole": false,
    "MIMode": "gdb",
    "setupCommands": [
        {
            "description": "Enable pretty-printing for gdb",
            "text": "-enable-pretty-printing",
            "ignoreFailures": true
        },
        {
            "description": "Set Disassembly Flavor to Intel",
            "text": "-gdb-set disassembly-flavor intel",
            "ignoreFailures": true
        }
    ],
    "preLaunchTask": "simple-add-usm Debug C/C++: dpcpp build active file",
    "miDebuggerPath": "/opt/intel/oneapi/debugger/latest/gdb/intel64/bin/gdb-oneapi"
},
{
    "name": "C/C++: dpc++ build and debug simple-add-buffers",
```

Add Configuration...

図 13a: simple-add-usm 実行ファイルのデバッグ構成

図 13b: simple-add-buffers 実行ファイルのデバッグ構成

launch.json ファイルの注目すべき変更点は、以下のとおりです。

- 各構成の名前を一意になるように変更しました (`"name": "C/C++: dpc++ build and debug simple-add-usm"`)。
- tasks.json ファイルと同じ環境変数の置換方法を使用しました (`"program": "${workspaceFolder}/bin/${config:programNameUsm}_d"`)。
- prelaunchTask は、tasks.json ファイルのビルド構成の相当するラベルに一致します。
- miDebuggerPath は oneAPI デバッガーを指します (`"miDebuggerPath": "/opt/intel/oneapi/debugger/latest/gdb/intel64/bin/gdb-oneapi"`)。

`queue q(d_selector, dpc_common::exception_handler);` などの SYCL* 関数にステップインまたはステップオーバーした際に Visual Studio* Code のデバッグセッションがストールしたり、ハングするのを防ぐため、launch.json ファイルを編集して以下を追加します。

```
"setupCommands": [
{
  "description": "Needed by Intel oneAPI: Disable target async",
  "text": "set target-async off",
  "ignoreFailures": true
}]
```

ハングすると、コードを実行したり、ステップオーバーするインタラクティブなデバッグパネルがゴーストアウトして応答しなくなります。一般に、この状況では、デバッグセッションを強制的に中止することしかできません。

注:
Visual Studio* Code のデバッグ・コンソール・コマンド・プロンプトから、C/C++ デバッグセッション中に **-exec <a gdb command>** を使用して gdb または gdb-oneapi コマンドをいつでも実行できます。本記事の執筆時点で、Microsoft 社は、これはまだ完全にテストされていないため、予期しない動作が発生する可能性があると述べています。

プロジェクトの変数設定 (環境変数) **${config:programNameUsm}** の **name** タイプオプションへの代入は動作しません。

これで、コマンドパレットで **[デバッグ: デバッグ セッションの選択] > [新しいデバッグ セッションを開始する] を選択して、デバッグする実行ファイルを選択し**、実行するデバッグセッションを選択できます。これがデフォルトのデバッグセッションになります。以降のデバッグセッションは、キーボード・ショートカット (Fn +) F5 または **Ctrl + Shift + d** を使用して行うことができます。ほかの実行ファイルを選択してデバッグするには、コマンドパレットから **[デバッグ: デバッグ セッションの選択]** を再度選択します。

これで、Visual Studio* Code のデバッグモードとデバッグ実行パネルを使って、通常の C++ プログラムと同様に DPC++ プログラムをデバッグできます。

次のステップ

今後公開予定のガイド 2 では、本ガイドをベースに、**Ubuntu* 上の Visual Studio* Code でインテル® DPC++ デバッガーを使用して**、DPC++ プログラムとそのカーネルを実行しながらデバッグする際に利用できる IDE の機能を紹介します。ガイド 2 では、コードを読み進めながらプログラムとカーネルの状態を視覚化する方法、ブレークポイントを設定する方法、変数の内容を表示する方法、そして各カーネルが実行する際に出力バッファーのメモリーが更新されるのを確認する方法を紹介します。
