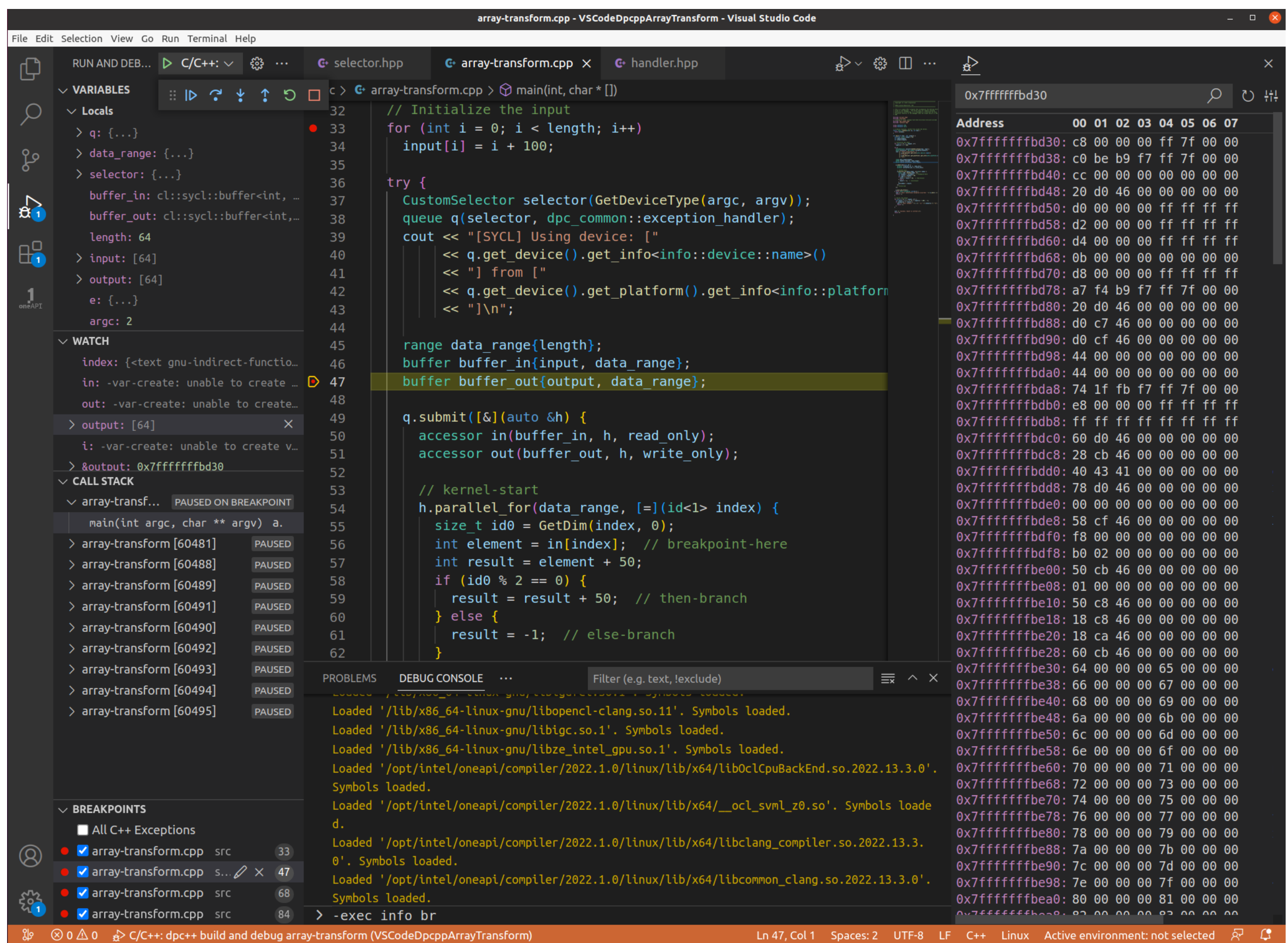
パート 4: Ubuntu* で Visual Studio* Code を使用して DPC++ をデバッグ

この記事は 2023 年 4 月 25 日に Codeplay のウェブサイトで公開された「Debugging the DPC++ debugger using 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は原文を優先してください。



本シリーズのこれまでの記事では、Ubuntu* 20.04 プラットフォーム上で Visual Studio* Code (VSCode) 開発環境をゼロからセットアップする方法を紹介しました。

1. パート 1: DPC++ と Visual Studio* Code で SYCL* コードをデバッグ
2. パート 2: Ubuntu* で C++ 開発向けに Visual Studio* Code を設定
3. パート 3: Ubuntu* で SYCL* 開発向けに oneAPI、DPC++、Visual Studio* Code を設定

はじめに

パート 4 では、VSCode IDE でデータ並列 C++ (DPC++) プログラムをデバッグして詳細を確認する方法を紹介します。ステップ 1 では、DPC++ プログラムの実行中に発生する明示的および暗黙的なアクティビティーについて詳しく見ていきます。

ステップ 2 では、VSCode IDE とさまざまな拡張機能を使用して、デバッグ中の DPC++ プログラムに関する詳細な情報をビューで確認します。VSCode デバッガーと各種拡張を使用することで、ターミナルのコマンドラインからインテルのデバッガー **gdb-oneapi** を使用する際に追加のデバッグ情報が得られます。

インテル® oneAPI ベース・ツールキット 2023.0.0 以降、インテルは dpcpp コンパイラーの代わりに icpx を使用しています。このシリーズでは、DPC++ プログラムは **icpx** コンパイラーでコンパイルされたプログラムを指します。

必要条件

デバッグを開始する前に、以下の必要条件を満たしていることを確認してください。

- Ubunbtu* 20.04 以降。
- インテル® oneAPI ベース・ツールキット 2023.0.0 が設定済みで動作すること。
- VSCode v1.75 が設定済みであること。これらの手順については、パート 1 ～3 を参照してください。
- VSCode 拡張 **NateAGeek Memory Viewer** がインストールされていること。デバッグの演習でこの拡張を参照します。

重要: launch.json ファイルの DPC++ プロジェクトのデバッグ設定に以下を追加します。

```
"setupCommands": [{
   "description": "Intel gdb-oneapi disable target async",
   "text": "set target-async off",
   "ignoreFailures": true
}]
```

このコマンドは、SYCL* 関数 `queue(DeviceSelector.select_device(),...)` にステップインまたはステップオーバーするように要求された場合に、gdb-oneapi デバッグセッションがハングアップするのを防ぎます。

VSCode のドキュメントは、広範囲にわたっており便利です。以下のトピックがこの記事に関連しています。

1. デバッグ:
   https://code.visualstudio.com/docs/editor/debugging (英語)
2. Visual Studio* Code で C++ をデバッグ:
   https://code.visualstudio.com/docs/cpp/cpp-debug (英語)
3. C/C++ デバッグの設定:
   https://code.visualstudio.com/docs/cpp/launch-json-reference (英語)

注:

ベクトル化とは:
https://stackoverflow.com/questions/1422149/what-is-vectorization (英語)

gdb のスレッド処理方法:
https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_24.html (英語)

インテル® DPC++ コンパイラーは、**LLVM clang++** コンパイラーのラッパーです。clang コンパイラーは **gcc** コンパイラーの互換ドライバーであるため、使用されるオプションや引数は同じです。デバッグ用に Linux* では gcc と clang の両方が DWARF/ELF オブジェクト・ファイルを生成します。DWARF/ELF オブジェクト・ファイルは、**gdb**、**lldb**、**gdb-oneapi**、またはその他の Linux* 標準デバッガーでデバッグできます。

gdb-oneAPI デバッガーは gdb 互換で、SIMD レーンをまたがるマルチスレッドのデバッグに対応する拡張を備えています。以下のインテルのドキュメントでは、gdb と gdb-oneapi デバッガーの違いについて説明しています。
https://www.intel.com/content/www/us/en/develop/documentation/debugging-dpcpp-linux/top.html (英語)

C++11 のラムダ関数については、以下の記事に説明があります。
https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170 (英語)

SYCL* では、カーネルで使用する変数の参照によるキャプチャーが許可されていないため、ほとんどの場合、ラムダ関数の定義では値によるキャプチャーが使用されます。値によるキャプチャーでは、関数スコープ全体が、ラムダが呼び出されるすべての呼び出しサイトにコピーされます。したがって、ラムダが並列または非同期操作で実行される SIMD のようなアーキテクチャーでは、このメソッドがよく使用されます。

array-transform プログラムのビルド

インテルの array-transform (英語) サンプルを使用して、VSCode IDE でデバッグを行います。以下の手順に従って、array-transform サンプルを準備します。DPC++ のコンパイルとデバッグ用に Microsoft* C/C++ 拡張機能を使用して VSCode を設定する方法の詳細は、パート 2 を参照してください。

VSCode IDE には、C++ コードを開発およびデバッグするいくつかの方法が用意されています。この記事では、Microsoft* C/C++ 拡張機能を使用して C++ プロジェクトを管理します。この記事で VSCode が単独で言及されている場合、それは通常、Microsoft* C/C++ 拡張パックを使用していることを意味します。

この例では、以下の手順に従って VSCode を設定します。

1. オプション: パート 2 の手順に従って、シンプルな VSCode C++ プロジェクトを作成します。
2. **VSCodeDpCppArrayTransform** という名前の新しい C++ プロジェクト・フォルダーを作成します。
3. 既存の DPC++ VSCode プロジェクトから **.vscode** ディレクトリーの設定ファイルをコピーするか、一連の設定ファイルを新規作成します。
4. **bin** ディレクトリーと **src** ディレクトリーを作成します。
5. array-transform のコードファイルを src ディレクトリーにコピーします。
6. プロジェクトの **.json** ファイル (**settings**、**tasks**、および **launch**) を編集して、コードの **debug** および **release** ビルドの構成を作成し、バイナリーを **bin** ディレクトリーに配置します。それぞれのファイルの例を図 1、2、3、および 4 に示します。

```
{
    "programName": "array-transform",
    "files.associations": {
    "string_view": "cpp",
        "regex": "cpp"
    }
}
```

図 1: プロジェクトの **settings.json** ファイル – プログラム名を設定

```
{
     "version": "2.0.0",
     "tasks": [
         {
             "type": "cppbuild",
             "label": "t_test MKL Debug C/C++: Intel icpx build active file",
             "command": "/opt/intel/oneapi/compiler/latest/linux/bin/icpx",
             "args": [
                 "-fsycl",
                 "-fno-limit-debug-info",
                 "-DMKL_ILP64",
                 "-fdiagnostics-color=always",
                 "-fsycl-device-code-split=per_kernel",
                 "-g",
                 "-I/opt/intel/oneapi/mkl/latest/include",
                 "-L/opt/intel/oneapi/mkl/latest/lib/intel64",
                 "-lmkl_sycl",
                 "-lmkl_intel_ilp64",
                 "-lmkl_sequential",
                 "-lmkl_core",
                 "-O0",
                 "${workspaceFolder}/src/${config:programName}.cpp",
                 "-o",
                 "${workspaceFolder}/bin/${config:programName}_d"
             ],
             "options": {
                 "cwd": "${workspaceFolder}"
             },
             "problemMatcher": [
                 "$gcc"
             ],
             "group": "build",
             "detail": "compiler: /opt/intel/oneapi/compiler/latest/linux/bin/icpx"
         },
```

```
        {
            "type": "cppbuild",
            "label": "t_test MKL Release C/C++: Intel icpx build active file",
            "command": "/opt/intel/oneapi/compiler/latest/linux/bin/icpx",
            "args": [
                "-fsycl",
                "-DNDEBUG",
                "-DMKL_ILP64",
                "-I/opt/intel/oneapi/mkl/latest/include",
                "-L/opt/intel/oneapi/mkl/latest/lib/intel64",
                "-lmkl_sycl",
                "-lmkl_intel_ilp64",
                "-lmkl_sequential",
                "-lmkl_core",
                "${workspaceFolder}/src/${config:programName}.cpp",
                "-o",
                "${workspaceFolder}/bin/${config:programName}"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": "build",
            "detail": "compiler: /opt/intel/oneapi/compiler/latest/linux/bin/icpx"
        }
    ]
}
```

図 2: プロジェクトの **tasks.json** ファイル - ビルド設定ファイル

```
{
    "configurations": [
    {
        "name": "C/C++: Intel icpx build and debug array-transform",
        "type": "cppdbg",
        "request": "launch",
        "program": "${workspaceFolder}/bin/${config:programName}_d",
        "args": [
            "${input:args}"
        ],
        "stopAtEntry": true,
        "cwd": "${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
        },
        {
            "description": "Needed by Intel oneAPI: Disable target async",
            "text": "set target-async off",
```

```
                "ignoreFailures": true
            }
            ],
            "preLaunchTask": "array-transform Debug C/C++: Intel icpx build active file",
            "miDebuggerPath": "/opt/intel/oneapi/debugger/latest/gdb/intel64/bin/gdb-
oneapi"
        }
        ],
        "inputs" : [
        {
            "id": "args",
            "type": "pickString",
            "description": "Program args",
            "default": "cpu",
            "options": [
            "cpu",
            "gpu",
            "accelerator"
            ]
        }
        ]
    }
```

図 3: Project の **launch.json** ファイル - デバッグ設定ファイル

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

図 4: プロジェクトの **c_cpp_properties.json** ファイル - IntelliSense 設定ファイル

7. このサンプルプログラムは引数を受け取るため、**launch.json** ファイルを編集してデバイス・オプション・リストをユーザーに表示します。図 5a と 5b に示すテキストを追加します。

```
"args": [
    "${input:args}"
],
```

図 5a: デバッグセッションに引数を渡すデバッグ設定

```
37    "inputs" : [
38        {
39            "id": "args",
40            "type": "pickString",
41            "description": "Program args",
42            "default": "cpu",
43            "options": [
44                "cpu",
45                "gpu",
46                "accelerator"
47            ]
48        }
49    ]
```

図 5b: デバッグセッションに引数を渡すデバッグ設定

8. **Ctrl + Shift + B** でデバッグビルドを選択してビルドします。Make などのポップアップが表示された場合は無視します。
9. デバッグ・ターミナル・ウィンドウで `./bin/array-transform_d` と入力し、プログラムが実行されることを確認します。

ステップ 1: DPC++ プログラムの検証

以下の詳細は、シンプルなサンプルプログラムのデバッグには必要ありませんが、プログラムの実行中に発生する多くの暗黙的なアクティビティーをより深く理解するのに役立ちます。

コードの構造

DPC++ プログラムは、アプリケーション、コマンドグループ、カーネルスコープの 3 つのスコープのネストされたセットとして見ることができます。この単一ソースのモデルコードは、以下の表 1 の一般的なシーケンスに従います。カーネルスコープ (図 6) は最も内側のスコープを形成し、デバイス上で実行されるカーネルコードを表します。

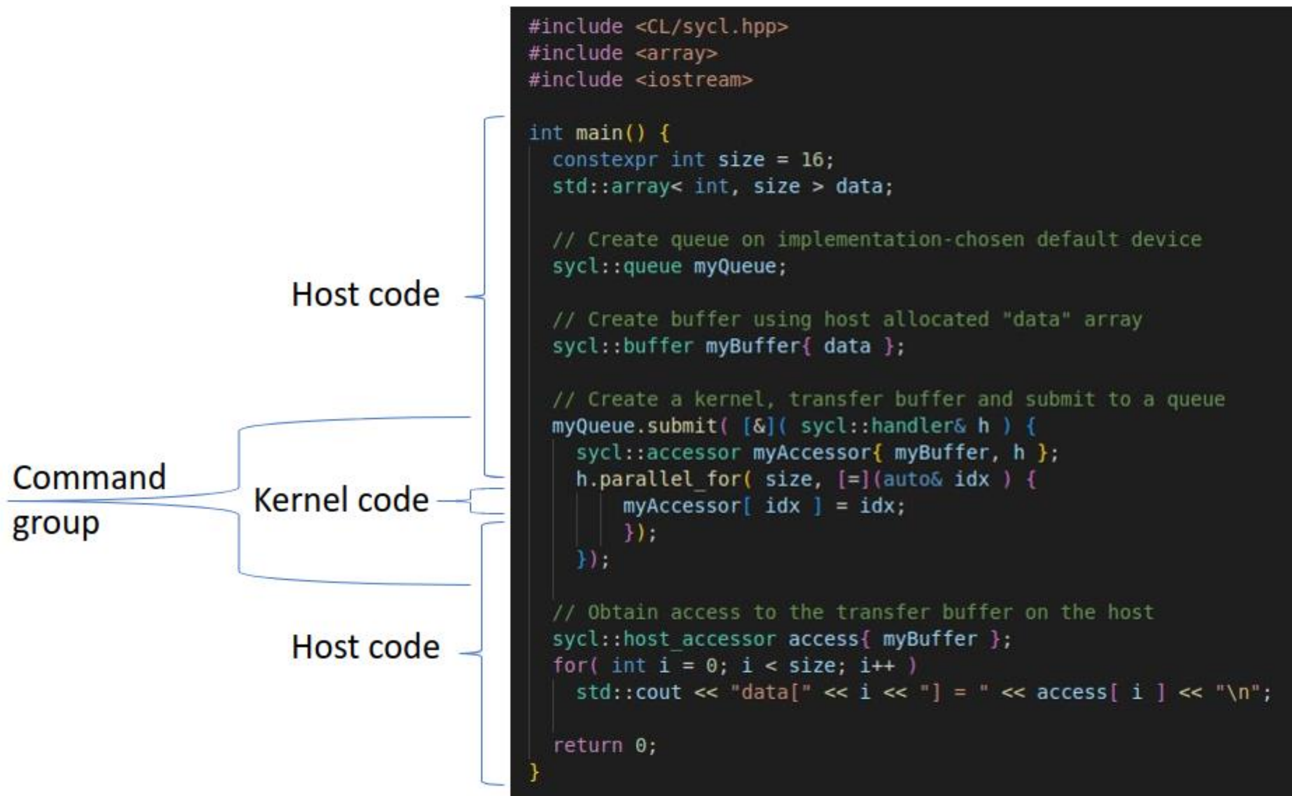


図 6: 典型的な DPC++ プログラムの構造

注:

カーネル･デバイス･コードの定義には、ラムダ式によって生成された匿名の名前のないクロージャー･タイプを使用するのが一般的ですが、代わりに通常の C++ 関数を使用することも可能です。例えば、カーネルコードに含まれる行数が多すぎる場合や、デバッグのため同じカーネルのバリエーションを素早く切り替える場合などに、ラムダの代わりに関数を定義できます。

CPU 以外のデバイスにカーネルコードを送信すると、特定の C++ 機能がサポートされません。これには、カーネル内から呼び出される関数が含まれます。関数ポインターは禁止されており、これには、仮想関数呼び出しやランタイム型情報などの関数ポインターのサポートに依存する機能も含まれます。例外、動的なメモリー割り当て、カーネル内でのランタイム再帰も禁止されています。

明示的および暗黙的なアクティビティー

図 7 に示す一般的な SYCL* プラットフォームは、ホスト上で実行される DPC++ プログラムのデバイスドライバー側または「バックエンド」で発生するアクティビティーを表します。

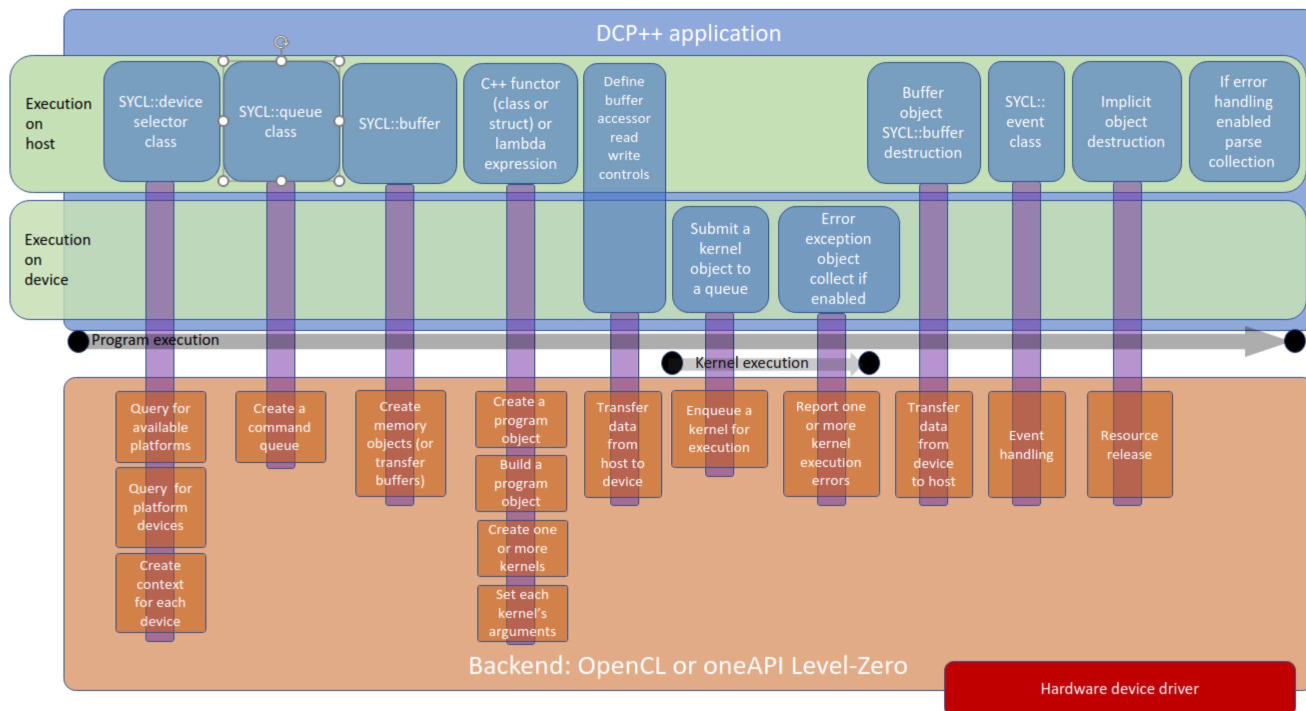


図 7: アクティビティーの順序と、「バックエンド」との相互作用

上記の典型的なプログラムでは、表 1 に示すように、プログラムフローの進行アクティビティーに分割されています。図 7 は、デバイスドライバー層の下位で発生する暗黙的なアクティビティーを表します。

手順	実行場所	DPC++ プログラム	関数	バックエンド (OpenCL* ライブラリー)	用語
1	ホスト	sycl::device selector クラス	カーネル (ラムダ関数) を実行する特定のデバイスを表します。デバイスとは、CPU、GPU、FPGA、またはその他のものです。デバイス・コンテキストを暗黙的にインスタンス化します。	利用可能なプラットフォームを照会します。	デバイス・コンテキストには 1 つ以上のデバイスが含まれます。コンテキストは、コマンドキュー、転送バッファーメモリー、プログラム、カーネル・オブジェクトなどのオブジェクトを管理し、コンテキストで指定された 1 つ以上のデバイスでカーネルを実行するために使用されます。
2	ホスト			デバイスの 1 つ以上のプラットフォームを照会します。	
3	ホスト			デバイスごとのコンテキストを作成します。	

手順	実行場所	DPC++ プログラム	関数	バックエンド (OpenCL* ライブラリー)	用語
4	ホスト	sycl::queue クラス	カーネルや sycl::queue 関数 (memcpy() など) を投入 (enqueued) できるキューを表します。カーネルや関数はホストコードから非同期に実行されます。キューの作業がすぐに開始することは期待できないため、キューの作業が開始される前にホストコードは続行します。カーネルは、実行に必要な依存関係がすべて満たされると作業を開始します。	コンテキストのコマンドキューを作成します。	コマンドキューは、1 つのデバイスと、作業が送信されるデバイス・コンテキストに関連付けられたオブジェクトです。デバイスごとに個別のキューを持つことができます。コマンドキューに送信されたコマンドは順番にキューに投入されますが、順番どおりに実行されるかどうかは、コマンドキューの作成時に指定された属性に応じてます。順番どおりに実行されない場合、同じデータバッファーにアクセスするホストコードとデバイスコードの間で競合状態が発生する可能性があることに注意してください。SYCL* の USM データ・バッファー・モデルは競合状態になりやすいです。SYCL* のバッファーモデルでは、SYCL* スケジューラーがキューのデータ依存関係を検査して競合状態を排除します。
5	ホスト	sycl::buffer クラス	ランタイムがホストとデバイス間のデータ転送に使用できるメモリー割り当てをカプセル化します。	メモリー・オブジェクト (または転送バッファー) を作成します。	
6	ホスト	sycl::handler クラス	カーネルとバッファーを接続するコマンド・グループ・スコープを定義します。		コマンドグループは作業を行います。コマンドグループは 1 つのカーネルとその依存関係をカプセル化し、コマンドキューに送信されると単一のアトミック・エンティティーとして処理されます。コマンドグループはファンクターまたはラムダとして定義されます。
7	ホスト	sycl::accessor クラス	特定のカーネルのバッファーアクセス要件 (読み取り、書き込み、または読み取り-書き込み) を定義します。	ホストからデバイスにデータを転送し、結果を返します。	

手順	実行場所	DPC++ プログラム	関数	バックエンド (OpenCL* ライブラリー)	用語
8	ホスト	sycl::range、sycl::nd_range、sycl::id、sycl::item、sycl::nd_item	実行範囲と範囲内の実行エージェントを表します。		範囲は、ワークグループとそれらのグループ内のワークアイテムを実行する反復スペースを表します。ワークは、より小さなワーク単位に分割したり、複数のワークアイテムを同じサイズのワークグループに分割したりできます。ワークグループは、同じデバイス上に存在することも、異なるデバイスに分散することもできます。カーネルの各インスタンスはワークアイテムと呼ばれます。ワークアイテムは、計算ユニット内の「単一の GPU 実行ユニット」またはプロセシング要素 (PE) 上で実行されます。
9	ホスト	C++ ラムダ表現または関数 (クラスまたは構造体)		プログラム・オブジェクトを作成します。	
10	ホスト			プログラム・オブジェクトをビルドします。	
11	ホスト			1 つ以上のカーネルを作成します。	一般に、カーネルは非同期で実行され、次の特性を備えていると想定されます。• カーネルはデータの依存関係またはデータ要件に従って実行をスケジュールします。•ホスト上のデータを要求するとカーネルはブロックします。•カーネルはメモリー操作を自動的に処理します。「スレッド」と「アイテム」という用語は同じ意味で使用されています。
12	ホスト			各カーネルの引数を設定します。	
13	ターゲットデバイス	カーネル・オブジェクトをキューに送信します。		実行のためカーネルをキューに投入します。	

手順	実行場所	DPC++ プログラム	関数	バックエンド (OpenCL* ライブラリー)	用語
14	ターゲットデバイス	カーネルはデバイスの実行ユニット全体で実行されます (キューを送信したホストスレッドは、カーネルの実行完了を待つことができます)。		1～N 個のカーネルが実行されます。	
15	ターゲットデバイス	オプション: 有効な場合、暗黙の非同期エラー例外を収集します。		0 または N のカーネル実行エラーを報告します。	
16	ホスト	バッファー・オブジェクト sycl::buffer の破棄		デバイスからホストにデータを転送します。	
17	ホスト	オプション: sycl::event クラス		デバイス・コンテキストの生成イベントを処理します。	
18	ホスト	暗黙のオブジェクト破棄		リソースを解放します。	
19	ホスト	オプション: コレクションに対してエラー例外が有効な場合、エラーリストを解析します。			

表 1: DPC++ プログラムによって実行されるホストとデバイス間のアクティビティーの順序

「バックエンド」は、低レベルでハードウェアと直接通信するハードウェア・インターフェイス・ライブラリーです。このようなライブラリーは、ハードウェア・ベンダーによって作成された OpenCL* ドライバーまたは OpenGL* ドライバーである可能性があります。例えば、インテルには oneAPI レベルゼロと呼ばれるドライバーがあります。「バックエンド」には、1 つ以上のアクセラレーション・デバイスが含まれます。一般に、デバイスドライバー API は OpenCL* のようなオープン・スタンダードに基づいています。オープン・スタンダードを採用することで、アプリケーション開発者は多くの異なるベンダーのアクセラレーターをターゲットとしながら、共通 API により同じソフトウェアを使用することができます。

注:

現在、SYCL* 仕様では、少なくとも 1 つのデバイスを持つホスト・プラットフォームは 1 つだけでなければならず、それは CPU デバイスでなければならないとされています。

DPC++ アプリケーションは、利用可能な計算デバイスにアクセスし、各デバイスを順番に使用するか、適切と判断した場合に一緒に使用するかを選択できます。

ステップ 2: array-transform プログラムのビルド

次のインテルのデバッグ情報ページを参照してください。

1. チュートリアル: Linux* ホスト上でのインテル® ディストリビューションの GDB を使用したデバッグ (英語)
2. チュートリアル: CPU 上での DPC++ アプリケーションのデバッグ (英語)
3. チュートリアル: GPU 上での DPC++ アプリケーションのデバッグ (英語)

array-transform プログラムは、入力データバッファーの要素が偶数か奇数かに応じて処理し、カーネルの計算を別の出力バッファーに出力します。

カーネルがいつでも任意の順序で実行されることを実証するには、インテルのサンプルプログラムにいくつかの変更を加えます。図 8 に示すようにプログラムを変更し、プログラムの `try catch` スコープの前にコードを挿入して再コンパイルします。

```
constexpr size_t dataElementSize = sizeof( int );
constexpr size_t fenceSize1 = dataElementSize * 2;
constexpr size_t fenceSize2 = dataElementSize * 2;
constexpr size_t dataBufferSize = 64;
constexpr size_t memBlkDebugLength = dataBufferSize + fenceSize1 + fenceSize2;
// Initialize the input data buffer
int arrayInput[ dataBufferSize ];
for( int i = 0; i < dataBufferSize; i++ )
arrayInput[ i ] = i + 100;

// Initialize the (output data buffer) debug memory block wrapping
// the output buffer. The 0xffs are the debug fence.
int arraysDebugMemBlock[ memBlkDebugLength ];
for( int i = 0; i < memBlkDebugLength; i++ )
arraysDebugMemBlock[ i ] = 0xffffffff;

// Initialize the output data buffer content with something so easy
// to see changes in buffer content as and when written to.
int *ptrArrayOutput = &arraysDebugMemBlock[ 0 ] + fenceSize1;
for( int i = 0; i < dataBufferSize; i++ )
ptrArrayOutput[ i ] = 0xbbbbbbbb;

// Commence program and kernel execution...
```

図 8: 出力バッファーのメモリーフェンスを作成するコード

図 9 は、**[NateAGeek Memory Viewer]** ペインです。これは、プログラムがカーネルを繰り返し実行する際に、データがデバイスからホストにコピーされ、ホスト側のバッファーが更新されるのを視覚的に確認できる非常に便利なツールです。

MEMORY VIEWER

0x7fffffffbd10

Address	00	01	02	03	04	05	06	07	Decoded ASCII Text
0x7fffffffbd10:	ff	ff	ff	ff	ff	ff	ff	ff	. . . .
0x7fffffffbd18:	ff	ff	ff	ff	ff	ff	ff	ff	. . . .
0x7fffffffbd20:	c8	00	00	00	ff	ff	ff	ff	. . . .
0x7fffffffbd28:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd30:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd38:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd40:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd48:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd50:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd58:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd60:	d8	00	00	00	bb	bb	bb	bb	. . . .
0x7fffffffbd68:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd70:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd78:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd80:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd88:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd90:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbd98:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbda0:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbda8:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbdb0:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbdb8:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbdc0:	f0	00	00	00	ff	ff	ff	ff	. . . .
0x7fffffffbdc8:	f2	00	00	00	bb	bb	bb	bb	. . . .
0x7fffffffbdd0:	f4	00	00	00	bb	bb	bb	bb	. . . .
0x7fffffffbdd8:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbde0:	f8	00	00	00	ff	ff	ff	ff	. . . .
0x7fffffffbde8:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbdf0:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbdf8:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbe00:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbe08:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbe10:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbe18:	bb	bb	bb	bb	bb	bb	bb	bb	. . . .
0x7fffffffbe20:	ff	ff	ff	ff	ff	ff	ff	ff	. . . .
0x7fffffffbe28:	ff	ff	ff	ff	ff	ff	ff	ff	. . . .
0x7fffffffbe30:	64	00	00	00	65	00	00	00	d . . . e . . .

図 9: カーネルの実行に伴って出力データバッファーの内容が変化する様子を確認する

ここで使用した単純なメモリーフェンスは、カーネルからホストバッファーへの書き込みがバッファーの境界を越えて、メモリー破壊や致命的なプログラム終了を引き起こすかどうかを検出できる手法の 1 つです。

Visual Studio* Code のデバッグモードに切り替えます。

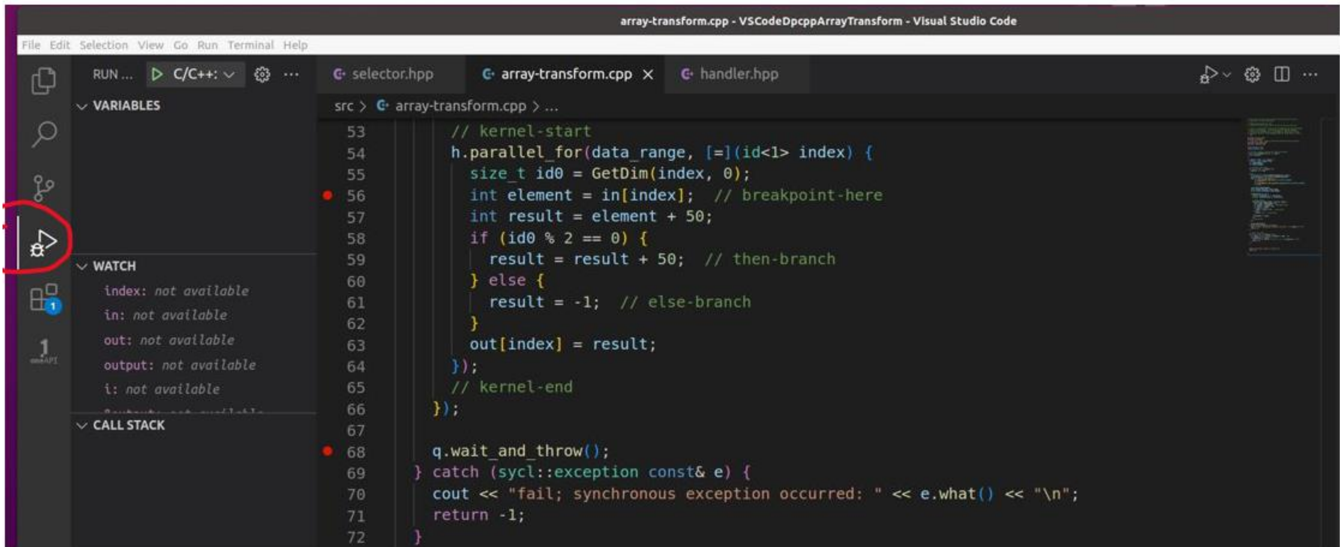


図 10: VSCode のデバッグモード

VSCode デバッガーを使用すると、デバッグセッションが実行中かどうかに関係なく、いつでもブレークポイントと変数ウォッチを追加または削除できます。デバッグセッションで作成されたほとんどの種類のブレークポイントとすべてのウォッチ式は、プロジェクト・ワークスペース (VSCode フォルダー) を閉じたり開いたりしても、将来の異なるセッションで保持されます。

ブレークポイントは、コード行番号の左マージンまたは余白にある赤い点で表されます。行をクリックして **F9** キーを押すか、マウスカーソルを行番号の左側の余白に移動してクリックします。

VSCode は、いくつかの異なる種類のブレークポイントをサポートしています。

- コード行に到達したら停止する
- 条件を満たす場合にコード行に到達したら停止する
- 関数に入ったら停止する
- コード行に N 回アクセスしたら停止する
- メモリー位置でデータが変更されたら停止する

注:

データ変更時の停止は文字列型の変数では機能しません。デバッグエンジンは、このタイプのブレーク・アクティビティーを実行するように要求されると、エラーを報告することがあります。これは、利用可能なプラットフォーム固有のデータ・ハードウェア・ブレークポイントの数、または変数型のサイズに制限があるためです。例えば、x64 では 4 つのハードウェア・データ・ブレークポイントと 8 バイト以下の型がサポートされます。詳細は、https://aka.ms/hardware-data-breakpoints (英語) を参照してください。

データ変更時の停止は、gdb または gdb-oneapi デバッガーでのみサポートされます。

メモリー位置や変数の読み取りを検出するブレークポイントの設定はサポートされていません。書き込み検出には watch、読み取り検出には rwatch、読み取りと書き込みの両方の検出には awatch という gdb-oneapi コマンドを使用します。

CPU 上でのデバッグ

プログラムをデバッグする前に、`main()` の直後の最初のコード行にデバッグ・ブレークポイントを設定します。

デバッグセッションを開始します。初めてサンプルをデバッグする場合は、IDE のコマンドパレット (**Ctrl + Shift + P**) から **[Debug: Select new debug session*]** を選択し、デバッグバージョンのビルドを選択します。**F5** キーを押すだけで新しいデバッグセッションを開始できます。IDE でデバッグするデバイスの選択を促されたら、`CPU` を選択します。

以下は、VSCode でデバッグする際に役立つキーボード・ショートカットです。

- **Ctrl + Shift + D** でデバッガーウィンドウが開きます。
- **F9** で選択したコード行にブレークポイントを設定または設定解除します。
- **F5** でデバッグを開始、またはアプリケーションを実行します。
- **F10** でステップオーバーします。
- **F11** でステップインします。
- **Shift + F11** で現在の関数からステップアウトして、呼び出し元の関数に戻ります。
- **Shift + F5** でデバッグを停止します。
- **Ctrl + Shift + F5** でデバッグを再開します。

プログラムはブレークポイントが設定された行で停止します。**[TERMINAL]** ペインを表示し、**[DEBUG CONSOLE]** を選択します。このペインには、gdb-oneapi デバッガーからのメッセージが表示されます。

インテルの array-transform サンプルの手順を参考に、任意の場所にブレークポイントを設定してコードをステップ実行します。インテルの記事では、カーネルコード行 `const int element = ptrDataIn[ index ]; // <-- set breakpoint here` にブレークポイントを設定することを提案しています。その場合、プログラムを続行すると、プログラムは最初に到達したブレークポイントで停止します。

インテルの記事では、gdb-oneapi コマンドラインで **print index** コマンドを入力し、変数 index の値を確認するよう求めています。VSCode では、次のいずれかを実行することで、この操作を行うことができます。

- **[DEBUG CONSOLE]** の **[TERMINAL]** ペインのプロンプトで exec print index と入力します。
- **[WATCH]** ペインで **+** アイコンをクリックし、index と入力します。
- 変数の上にマウスカーソルを置いて、変数の値を表示します。

  注:

  変数シンボルを選択中に **F12** キーを押すと、変数の定義が表示されます。

変数のアドレスは、**[WATCH]** ペインで「**&**」という接頭辞を付けて変数名を入力することで検索できます。

変数の値を変更するには、**[VARIABLES]** ペインまたは **[WATCH]** ペインで変数を選択して、コンテキスト・メニューから **[Set Value]** を選択します。

VSCode では、メモリー・ダンプ・ビューを表示する方法が制限されています。gdb の **x** コマンドを使用して、特定のメモリーアドレスのメモリーの内容を表示します (例: **-exec x /64ab 0x7fffffffbca0**)。

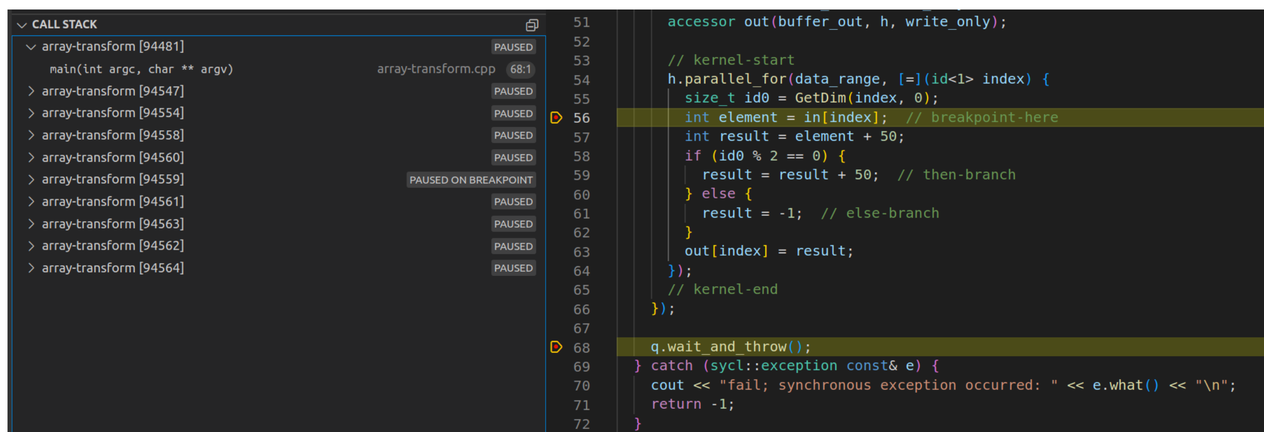


図 11: カーネルコードのある行で停止している間にデータ範囲で動作している
現在のスレッド (カーネル) セットのビュー

変数 index を調べたら、もう一度 **F5** キーを押します。デバッガーは前回と同じ行で、別のスレッドによって停止します。変数の内容が変更されると、**[WATCH]** ペインは自動的に更新されます。**NateAGeek Memory Viewer** を使用している場合、メモリービューは自動的に更新されないため、手動で更新する必要があります。

このプロジェクトのインテルバージョンと同様に、デバッグはステップ実行されません。代わりに、実行中の各スレッドでブレークポイント・イベントが発生し、そのスレッドのコンテキストに切り替わります。シングルスレッドのコードをステップ実行するには、IDE の **[DEBUG CONSOLE]** プロンプトで gdb-oneapi コマンド **set scheduler-locking step** または **on** を使用します。これはメインスレッドではないため、ホスト側のコードのデバッグに戻るときに、コマンド **set scheduler-locking replay** または **off** を使用して、必ずこの設定を元に戻してください。

プログラムのアクセサー `in` の内容を確認するには、**[WATCH]** ペインに追加します。変数 `in` を展開すると、変数のさまざまな側面が表示されます。変数 `in` の **MData** フィールドをクリックすると、最初のメモリーアドレスとその内容が表示されます。変数の HEX Editor をクリックすると、そのメモリー位置の内容が表示されます。拡張機能をインストールするように求められた場合は、**[はい]** を選択してインストールし、再度 MData フィールドを選択すると、新しい HEX メモリー・ビュー・ペインが表示されます。メモリーの内容の上にマウスを移動すると、ポップアップが表示され、ほかの数値形式で 16 進数が表示されます。

デバイスとホスト間のデータ移動の監視

SYCL* プログラムの特徴の 1 つは、ホスト側のコードとデバイス側のカーネルコードで共有可能なデータバッファーを持つことです。カーネルが入力データの処理を実行すると、その結果はさまざまなバッファーに配置されますが、どのバッファーにどの順序で配置されるのでしょうか? 転送は 1 回 (効率的)、あるいは複数回 (非効率的) なのでしょうか? 次のようなバッファーを考えてみます。

- `int output[length]`
- `buffer buffer_out{output, data_range}`
- `accessor out(buffer_out, h, write_only)`

array-transform プログラムの `output` バッファーには、どの段階でカーネルの実行結果が格納されるのでしょうか? プログラムをデバッグして確認していきます。このデバッグセッションでは、プログラム実行のさまざまな場所と時間で、さまざまな種類のブレークポイントを組み合わせて使用します。これは、変数の内容 (変数スコープ) がプログラムの実行時にのみ表示されるためです。発生する可能性がある範囲を調査するため、次の手順を複数回繰り返し、コード内でさまざまなブレークポイントを有効にして実験すると良いでしょう。

1. Visual Studio* Code をデバッグモードに変更します。
2. プログラム内のすべてのブレークポイントを削除します。
3. **[DEBUG CONSOLE]** の **[TERMINAL]** ペインを表示して、デバッガーの出力メッセージを確認します。
4. 行 `int arrayInput[ dataBufferSize ];` にブレークポイントを設定します。
5. デバッグセッションを開始し、アクセラレーション・デバイスとして **[CPU]** を選択します。
6. **[VARIABLES]** ペインの **[Locals]** 以下にある変数 `output` を展開し、`[0]: -16928` を選択します。
7. コンテキスト・メニューから **[Break on Value Change]** を選択します。新しいブレークポイントが **[BREAKPOINTS]** ペインに表示されます。
8. **[WATCH]** ペインで新しい式を追加し、「**&output**」と入力します。
9. `output` 配列のアドレスをコピーし、**NateAGeek Memory Viewer** に貼り付けて、**[Refresh]** をクリックします。
10. このメモリー位置から始まるコンテンツをメモします (画面をキャプチャーします)。
11. 行 `constexpr int dimension = 0;` にブレークポイントを設定します。
12. プログラムの実行を続行します。行 `constexpr int dimension = 0;` のブレークポイントに到達すると、プログラムは実行を停止します。
13. **[VARIABLES]** ペインで、`buffer_out` を選択し、**[Break on Value Change]** を選択します。
14. 新しいブレークポイントが **[BREAKPOINTS]** ペインに表示されますが、有効になっていません。ブレークポイントの上にマウスを移動すると、メッセージが表示されます。
15. このブレークポイントは有効にならないため、`buffer_out` の変更を監視できません。
16. 行 `result = result + 50;` にブレークポイントを配置します。このブレークポイントに到達すると、プログラムは実行を停止します。
17. **[VARIABLES]** ペインで、`out` 変数を選択し、**[Break on Value Change]** を選択します。
18. 新しいブレークポイントが **[BREAKPOINTS]** ペインに表示されますが、有効になっていません。ブレークポイントの上にマウスを移動すると、メッセージが表示されます。
19. カーネルコードの以降の段階にブレークポイントを設定します。行 `result = result + 50;` のブレークポイントを削除して、プログラムを実行し続行します。

NateAGeek Memory Viewer を更新して、`output` 配列の内容が変更されたことを確認してください。

> **注:**
>
> Ubuntu* の電卓アプリを 16 進数モードで使用して、16 進数を同等の 10 進数に変換できます。

まとめ

メインスレッドは確かに `q.wait` で待機していますが、コードをステップ実行すると、カーネルの実行前、実行中、実行後のどの時点でも一時停止していることが確認できます。同じメインスレッドのブレークポイントに到達する前に、カーネル・ブレークポイントに到達できるため、どちらに最初に到達するかは不明です。プログラムを続行すると、最終的にメインスレッドは `q.wait` のブレークポイントで停止します。その間、カーネルは非同期で実行されます。すべてのカーネル・ブレークポイントが実行されると、メインスレッドはすでに `q.wait` を実行しているため、`q.wait` で再び停止することはありません。デバッグセッションが終了しないように、カーネル実行とは関係のない次のメインスレッド実行コードパスにブレーク ポイントを設定することを推奨します。

`output` 配列の内容は、デバッガーがトリガーされたデータ変更ブレークポイントで停止する前に変更される可能性があり、「いつ、どこで」変更されたかのデバッグには、特にメモリーフェンスに到達したかどうかを確認する場合は時間を要します。

Codeplay Software Ltd has published this article only as an opinion piece. Although every effort has been made to ensure the information contained in this post is accurate and reliable, Codeplay cannot and does not guarantee the accuracy, validity or completeness of this information. The information contained within this blog is provided "as is" without any representations or warranties, expressed or implied. Codeplay Sofware Ltd makes no representations or warranties in relation to the information in this post.