

Using Visual Studio Code with Intel® oneAPI Toolkits User Guide

Contents

Chapter 1: Using Visual Studio Code* to Develop Intel® oneAPI Applications

Using Visual Studio Code* with Locally Hosted Intel® oneAPI Toolkits.....	3
Developing a Visual Studio Code Project.....	4
Using Visual Studio Code* with Remote Intel® oneAPI Toolkits via SSH	9
Developing a Visual Studio Code Project for SSH Development	10
Developing a Visual Studio Code* Project for SSH Development on Windows Subsystem for Linux*	19
Using Visual Studio Code* with Intel® oneAPI Toolkits in a Docker* Container..	27
Developing a Visual Studio Code Project in a Docker Container.....	28
Intel® oneAPI Extensions for Visual Studio Code*	37
Environment Configurator for Intel® oneAPI Toolkits*	38
Analysis Configurator for Intel® oneAPI Toolkits*	39
GDB GPU Support for Intel® oneAPI Toolkits	44
DevCloud Connector for Intel® oneAPI Toolkits*	50
Search for a Specific Sample	51
Set the oneAPI Environment Manually	51
Working with Code	51
Notices and Disclaimers.....	52

Using Visual Studio Code* to Develop Intel® oneAPI Applications

1

Developing projects with VS Code for Intel oneAPI Toolkits can be achieved through different methods. Click on one of the methods below to get started:

- [Developing a Visual Studio Code Project](#)
- [Developing a Visual Studio Code Project for SSH Development](#)
- [Developing a Visual Studio Code Project in a Docker Container](#)
- [Using Visual Studio Code with Intel® DevCloud for oneAPI](#)

Using Visual Studio Code* with Locally Hosted Intel® oneAPI Toolkits

Using Visual Studio Code with Intel® oneAPI Toolkits*

This guide assumes you are familiar with C/C++ development and the Visual Studio Code (VS Code) editor. If you are new to Visual Studio Code, review these VS Code documentation links:

- [Setting up Visual Studio Code](#)
- [Install the C/C++ Extension Pack for Visual Studio Code](#)
- [Visual Studio Code Remote Development Extension Pack](#)
- [Visual Studio Code User and Workspace Settings](#)
- [Getting Started with C++ in Visual Studio Code](#)
- [Getting Started with CMake Tools on Linux](#)
- [Debugging C++ in Visual Studio Code](#)

Prerequisites:

If you have not configured your system and built and run a sample project, please refer to the appropriate toolkit Get Started guide and complete those steps:

- Local Linux development system (connected directly to your keyboard/video/mouse or remotely via RDP or VNC)
- [CMake and the Linux C/C++ development tools](#)
- [Visual Studio Code for Linux](#)
- [C/C++ Extension Pack for Visual Studio Code](#)
- Install at least one Intel® oneAPI Toolkit (see list below)

For information about the contents of downloading the Intel oneAPI Toolkits, see:

- [Install an Intel® oneAPI toolkit](#)
- Build and run a sample project with one of these toolkits:
 - Linux*:
 - [Intel® oneAPI Base Toolkit](#)
 - [Intel® HPC Toolkit](#)
 - [Intel® oneAPI IoT Toolkit](#)
 - [Intel® System Bring Up Toolkit](#)
 - [Intel® Rendering Toolkit](#)
 - [Intel® AI Tools](#)

When you have completed those steps, see [Developing a Visual Studio Code Project](#)

- **Developing a Visual Studio Code Project**
 - [Explore Samples Using Visual Studio Code*](#)
 - [Browse oneAPI samples using VS Code:](#)
 - [Configure the oneAPI Environment](#)
 - [Build and Run](#)
 - [Try Debugging \(CPU and GPU Only\) \(Preview\)](#)

Developing a Visual Studio Code Project

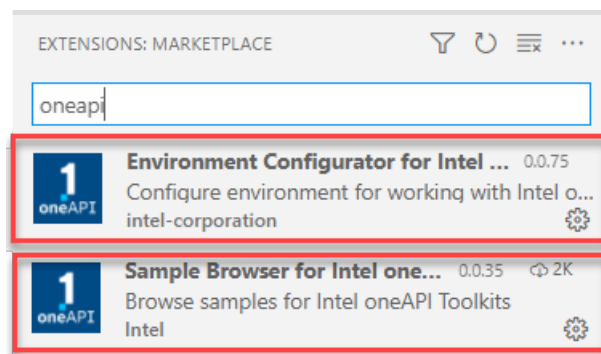
Using Visual Studio Code with Intel® oneAPI Toolkits*

The steps below describe how to install oneAPI extensions for Visual Studio Code and use them to set up your environment, browse samples and create projects. If you prefer to watch a video presentation of how to do these tasks, see [oneAPI Visual Studio Code Extensions](#).

Explore Samples Using Visual Studio Code*

Before working with oneAPI samples, it is recommended that you install the VS Code “Code Sample Browser for Intel oneAPI Toolkits” extension and the VS Code “Environment Configurator for Intel oneAPI Toolkits” extension.

Both can be quickly found in the VS Code extensions marketplace by typing “oneapi” into the marketplace search bar.



You may also choose to install the “Extension Pack for Intel oneAPI Toolkits”, which includes those two extensions as well as additional VS Code extensions to help you develop with oneAPI Toolkits.

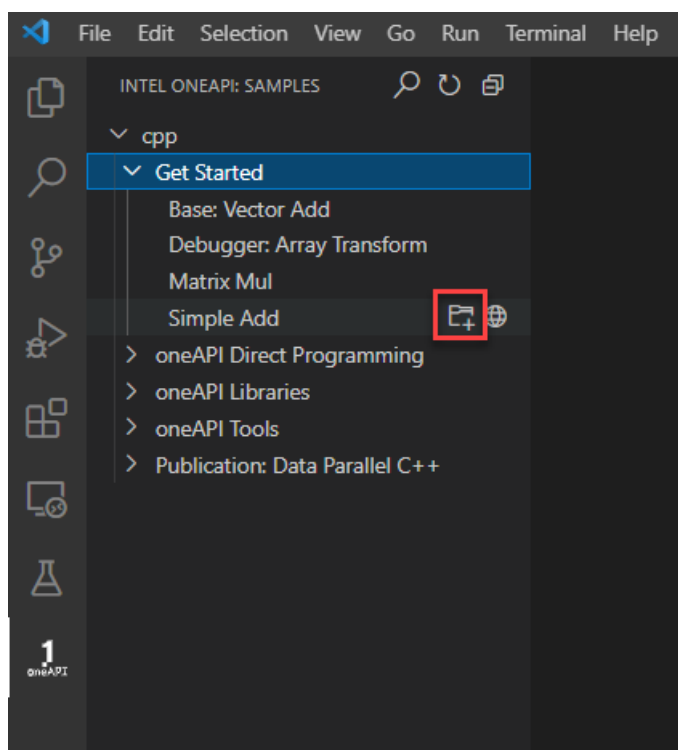
The steps below describe how to use these extensions to configure the oneAPI development environment for use with VS Code and use the Sample Browser to locate and create sample projects that help you learn how to use oneAPI features.



Browse oneAPI samples using VS Code:

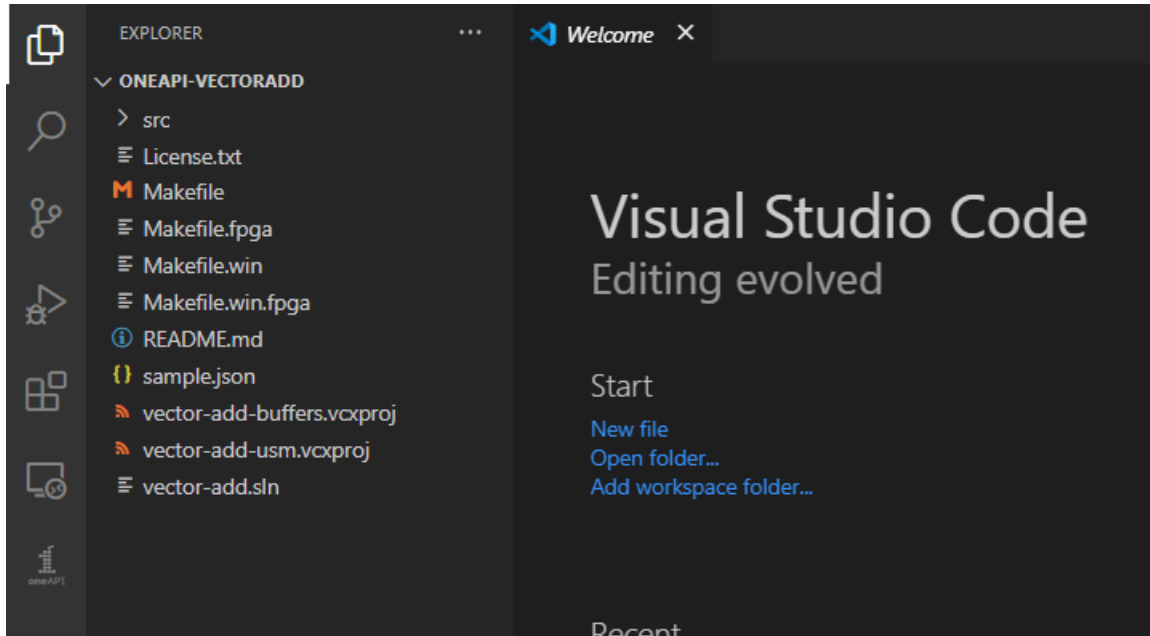
1. Click on the oneAPI button on the left navigation to view samples. If you do not have the extension installed, search the Extensions Marketplace for “Sample Browser for Intel oneAPI”.



1. A list of available samples will open in the left navigation.



1. To view the readme for the sample, click the  next to the sample. If you choose to build and run the sample, the readme will also be downloaded with the sample.
2. To build and run a sample, click the  to the right of the sample name.
3. Create a new folder for the sample. The sample will load in a new window:



Configure the oneAPI Environment

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI** to view options of the installed extensions.
3. Click on **Intel oneAPI: Initialize environment variables**.
4. Locate the setvars | oneapi-vars file:
 - Linux: By default in the Component Directory Layout, `setvars.sh` is located in `/opt/intel/oneapi/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
 - Linux: By default in the Unified Directory Layout, `oneapi-vars.sh` is located in `/opt/intel/oneapi/<toolkit-version>/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
 - Windows: By default in the Component Directory Layout, `setvars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\`
 - Windows: By default in the Unified Directory Layout, `oneapi-vars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\<toolkit-version>\`

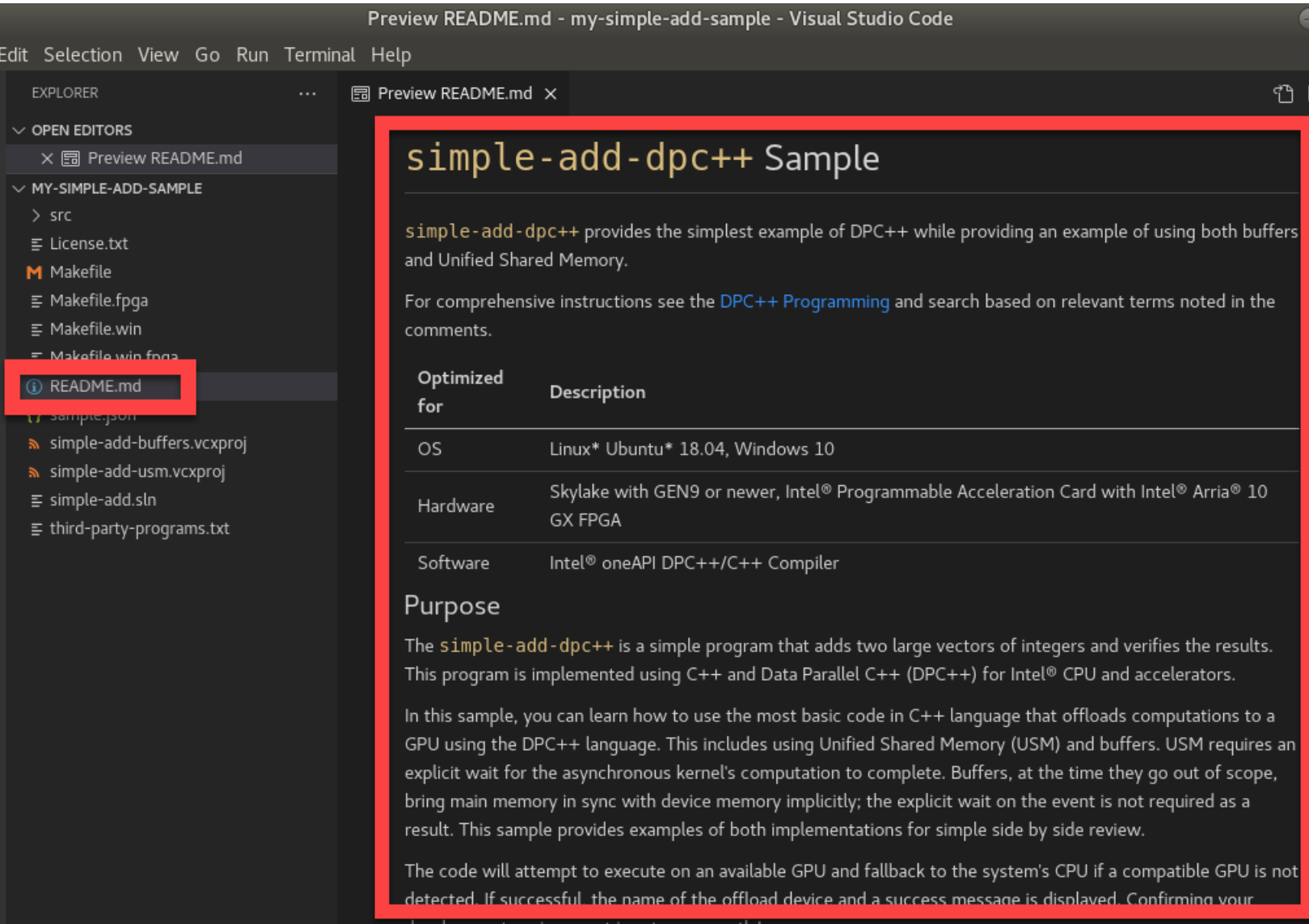
To understand more about the Unified Directory Layout, including how the environment is initialized and the advantages of using the layout, see [oneAPI Development Environment Setup](#).

If you customized the installation folder, setvars | oneapi-vars is in your custom folder.

5. In the case of multiple folders in workspace, select the appropriate one. All tasks, launches, and terminals created from VS Code will now contain the oneAPI environment.

Build and Run

Follow the instructions in the `README.md` for the sample.



NOTE Not all oneAPI sample projects use CMake. The README.md file for each sample specifies how to build the sample. We recommend that you check out the [CMake extension for VS Code](#) that is maintained by Microsoft.

Try Debugging (CPU and GPU Only) (Preview)

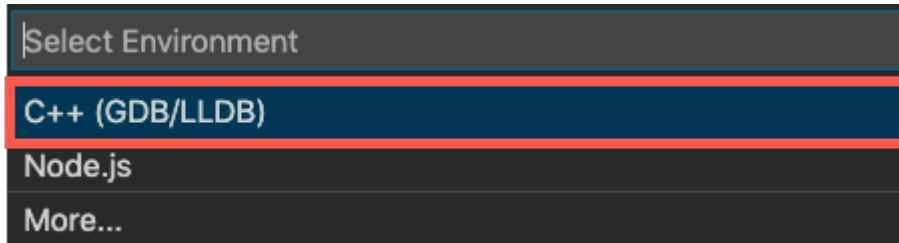
NOTE Intel® Distribution for GDB* does not currently support VS Code. You can use upstream gdb to debug. Debugging a local Windows host using VS Code is not supported. Debugging a local Linux host on CPU is supported using VS Code, but debugging on GPU requires a remote host. Debugging a remote Linux target from Windows or Linux host is supported using VS Code.

This section assumes that you can build your sample and have installed the [Microsoft VS Code C/C++ Extension](#). The C/C++ extension is required to configure the oneAPI C/C++ debugger.

The [Intel® oneAPI Base Toolkit](#) includes a special version of GNU* GDB (`gdb-oneapi`) designed to support oneAPI C/C++ applications. To debug your DPC++ application using this special debugger, you will need to make changes to the `.vscode/launch.json` configuration file.

1. Go to **Debug > Open Configurations**, and open the `launch.json` configuration settings.

NOTE If you are prompted to select a debug environment, choose **C++ (GDB/LLDB)**.



1. Copy the code shown below into your `launch.json` file, and replace the `"program":` property's value with the path to your project's executable (that is, the application that you are going to debug).

..note:

If VS Code doesn't recognize the application name, you may have to insert the full path and file name into the `launch.json` file's `"program":` property.

2. Add `gdb-oneapi` to your `launch.json` configuration's `"miDebuggerPath":` property.

NOTE The `gdb-oneapi` application should have been added to your path when you ran `setvars.sh` | `oneapi-vars.sh` to configure the oneAPI development environment, prior to starting VS Code. If you prefer, you can specify the full path and filename to the `gdb-oneapi` application in your `launch.json` file.

3. In some configurations, GDB may not be compatible with VS Code. If this happens, add the environment variable to disable `gdb-oneapi` support for GPU autolaunch. This can either be done in the environment prior to launching VS Code, or within the `launch.json`: `export INTELGT_AUTO_ATTACH_DISABLE=1`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/array-transform",
      "args": [
        "cpu"
      ],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "gdb-oneapi",
      "setupCommands": [
        {
```

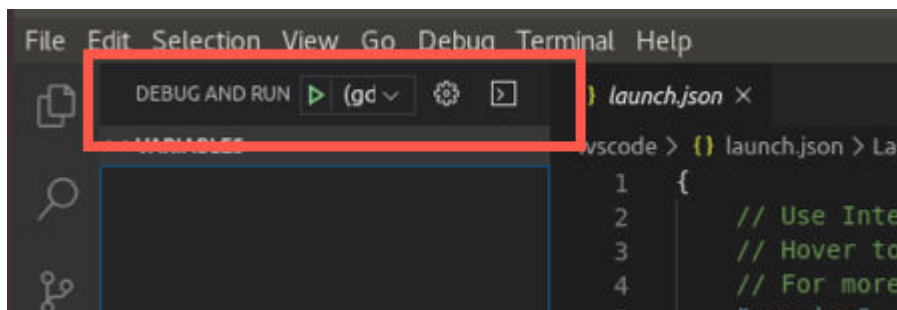


```
    "description": "Enable pretty-printing for gdb",  
    "text": "-enable-pretty-printing",  
    "ignoreFailures": true  
  },  
  {  
    "description": "Disable target async",  
    "text": "set target-async off",  
    "ignoreFailures": true  
  }  
]  
}  
]
```

4. Bring up the debug view by selecting the **Run** icon in the **Activity Bar**. You can also use the keyboard shortcut (**Ctrl+Shift+D**).



5. Start the run and debug session by clicking the green **DEBUG AND RUN** icon, or go to **Run > Start Debugging (F5)**.



Using Visual Studio Code* with Remote Intel® oneAPI Toolkits via SSH

Prerequisites:

NOTE For FPGA development, use the Intel® oneAPI DPC++/C++ Compiler.

If you have not configured your system and built and run a sample project, please refer to the appropriate toolkit Get Started guide and complete those steps:

- [Install an Intel® oneAPI toolkit](#)
- Build and run a sample project with one of these toolkits:
 - [Intel® oneAPI Base Toolkit](#)
 - [Intel® HPC Toolkit](#)
 - [Intel® oneAPI IoT Toolkit](#)

When you have completed those steps, refer to [Developing a Visual Studio Code Project for SSH Development](#) or [Developing a Visual Studio Code* Project for SSH Development on Windows Subsystem for Linux*](#) for steps on how to connect and develop your project.

- [Developing a Visual Studio Code Project for SSH Development](#)
 - [Set oneAPI Environment Variables on the Remote Host](#)
 - [Configure Visual Studio Code on Your Local Host](#)
 - [Connect to Your Remote Linux Target](#)
 - [Explore Samples Using Visual Studio Code*](#)
 - [Try Debugging \(CPU and GPU Only\) \(Preview\)](#)
 - [Disconnect from a Remote Host](#)
 - [Kill Remote VS Code Server](#)
- [Developing a Visual Studio Code* Project for SSH Development on Windows Subsystem for Linux*](#)
 - [Prerequisites](#)
 - [Set oneAPI Environment Variables on the WSL Target Linux](#)
 - [Configure Visual Studio Code on Your Local Host](#)
 - [Connect to Your WSL Linux System](#)
 - [Explore Samples Using Visual Studio Code*](#)
 - [Try Debugging \(CPU and GPU Only\) \(Preview\)](#)

Developing a Visual Studio Code Project for SSH Development

Using Visual Studio Code with Intel® oneAPI Toolkits*

This guide assumes you are familiar with C/C++ development and the Visual Studio Code (VS Code) editor. If you are new to Visual Studio Code, review these VS Code documentation links:

- [Setting up Visual Studio Code](#)
- [Install the C/C++ Extension Pack for Visual Studio Code](#)
- [Visual Studio Code Remote Development Extension Pack](#)
- [Visual Studio Code User and Workspace Settings](#)
- [Getting Started with C++ in Visual Studio Code](#)
- [Getting Started with CMake Tools on Linux](#)
- [Debugging C++ in Visual Studio Code](#)

Prerequisites:

If you have not configured your system and built and run a sample project, please refer to the appropriate toolkit Get Started guide and complete those steps:

- [Local host with Visual Studio Code \(VS Code\) and ssh installed](#)
- [Remote Linux target with an sshd server installed and running](#)
- [Remote Linux target with an Intel® oneAPI Toolkit installed](#)
- [CMake and the Linux C/C++ development tools](#)
- [Visual Studio Code for Linux](#)
- [C/C++ Extension Pack for Visual Studio Code](#)

- [Install at least one Intel® oneAPI Toolkit](#)

NOTE It is not necessary to install VS Code on your remote Linux system; it is needed only on your local system. This means that your remote Linux target can be a “headless” or CLI-only system; your remote Linux system does not require a GUI in order to be used for remote development.

The following procedure will use a Remote Host connection enabling you to edit and debug your code remotely. For instructions on how to develop locally, see [Local Usage of Visual Studio Code with Intel® oneAPI Toolkits on Linux*](#).

Set oneAPI Environment Variables on the Remote Host

In order to perform development on your remote Linux target, you must install an [Intel oneAPI Toolkit](#) and configure the oneAPI development environment variables on the remote Linux target.

1. Log in to your remote Linux target using `ssh` and install the Intel oneAPI Toolkit onto that target system.
2. Configure your remote Linux target so that the oneAPI development environment script (`setvars.sh` | `oneapi-vars.sh`) runs when VS Code initiates a remote connection.

NOTE Visual Studio Code does not currently provide a mechanism to automatically run scripts on your remote Linux target when it “remotes into” your target system (for example, running `setvars.sh` | `oneapi-vars.sh` remotely). There are a variety of ways to get around this issue, only one of which is presented here. See the [Bash Startup Files](#) man page for more options.

Add the following shell script lines to your remote Linux system’s `/etc/profile` script. This location (or in an `/etc/profile.d/` script) will ensure that all users of your remote Linux target development system will run the `setvars.sh` | `oneapi-vars.sh` environment script when they connect remotely using Visual Studio Code:

```
if [ -z "$SSH_TTY" ] && [ -n "$SSH_CLIENT" ]; then
    . /opt/intel/oneapi/setvars.sh &>/dev/null
fi
```

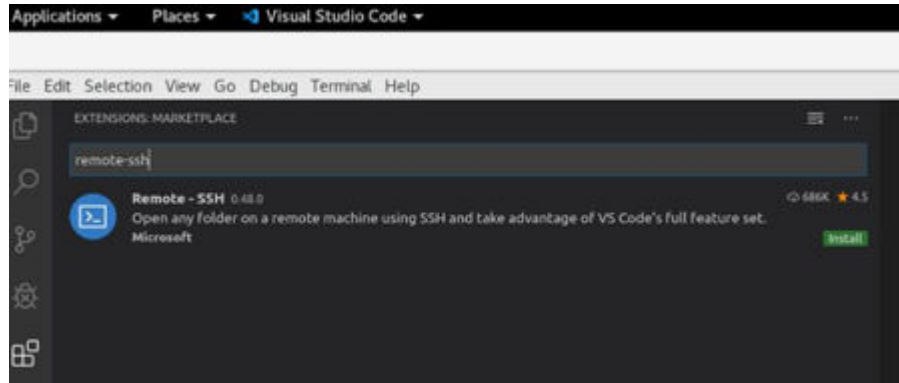
The above assumes that you have installed your oneAPI toolkits on your remote Linux target in the default “root/sudo” installation location (`/opt/intel/oneapi/`). If you have installed the oneAPI tools in a different location on your remote Linux system, you will have to adjust the path to `setvars.sh` | `oneapi-vars.sh`.

NOTE The script shown above *will not* execute the `setvars.sh` | `oneapi-vars.sh` script when you `ssh` into your remote Linux system or if you are using it directly with a terminal session. Remove the first and third lines if you want it to execute for such interactive terminal sessions. See [Configure Your CPU or GPU System](#) for more details regarding configuration of the oneAPI environment on a Linux system.

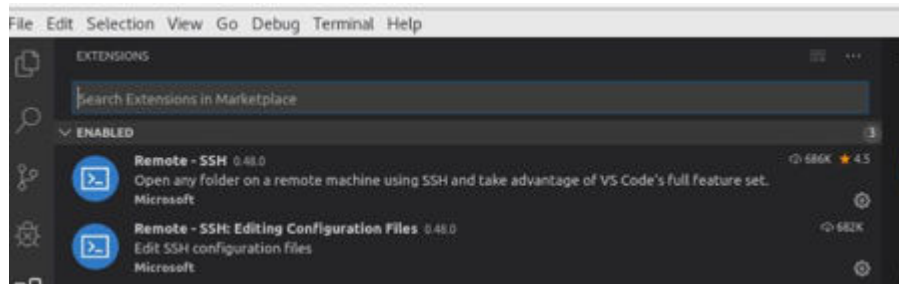
Configure Visual Studio Code on Your Local Host

You must install the VS Code “remote ssh” extension into your local copy of VS Code. This extension facilitates development on your remote Linux target.

1. Install the **Remote - SSH** extension by Microsoft. Click the **Extensions** icon and search for “Remote-SSH” in the search bar.



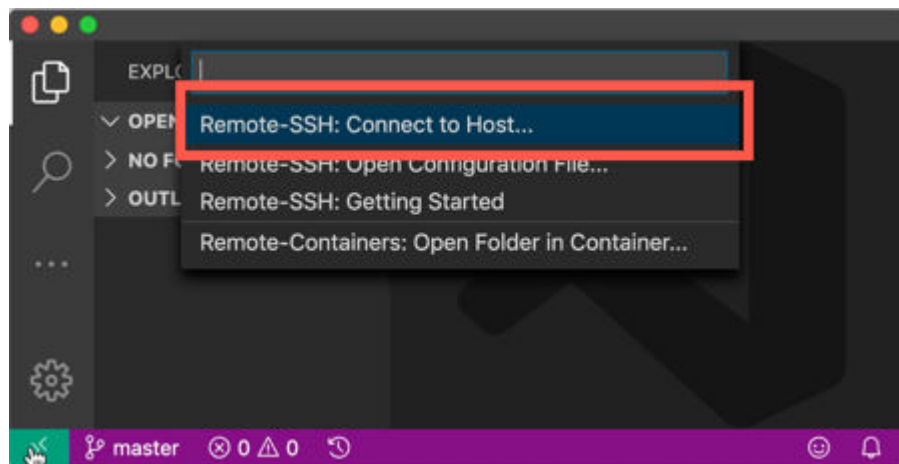
2. Click **install**. After installation, you see **Remote – SSH** in the **Installed Extensions** list.



Connect to Your Remote Linux Target

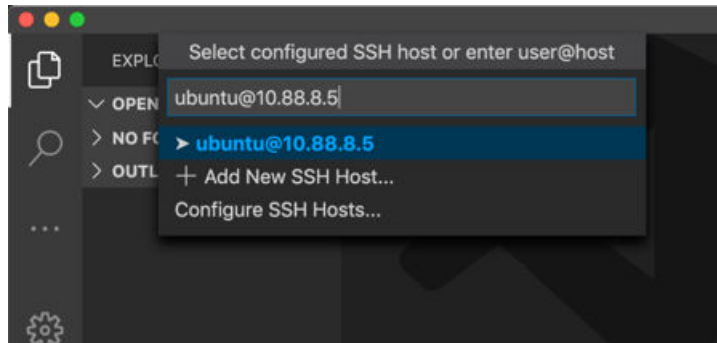
See the Visual Studio Code [Remote Development using SSH](#) documentation for detailed instructions, requirements, and additional information.

1. Click the **remote-ssh** icon located in the lower left corner of your VS Code window. A **remote-ssh commands** palette appears near the top of the VS Code window. Select **Remote-SSH: Connect to Host** in that command palette.

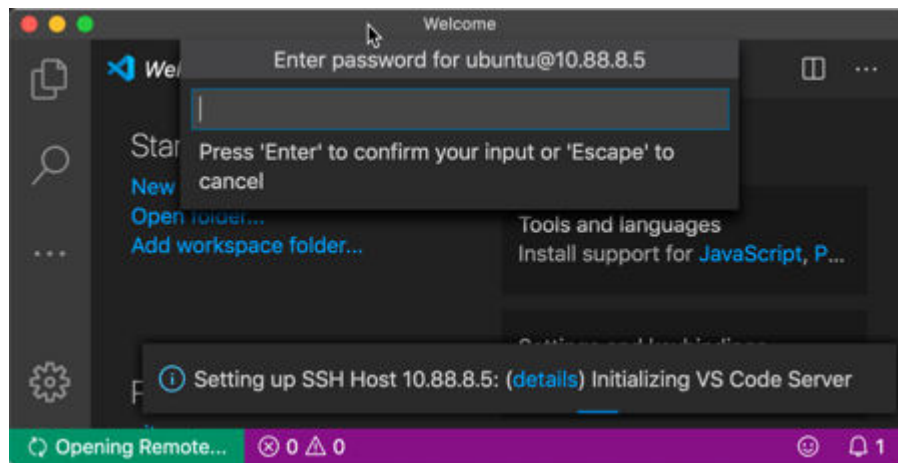


NOTE The first connection to your remote Linux target may require a minute or longer to download and install the necessary VS Code backend tools to your remote system. Subsequent connections to the remote system are generally faster. VS Code will indicate the installation process in the lower-right corner of your VS Code window.

2. Enter the username and an IP address or valid hostname for your remote Linux target, using the same format you would use for an SSH connection into that system, and then press **Enter**. In the image below, the username is ubuntu.



3. A new VS Code window opens and is connected to the remote host. If you do not have an SSH key-pair set up for this connection, you will be prompted to enter a password for your remote Linux target.



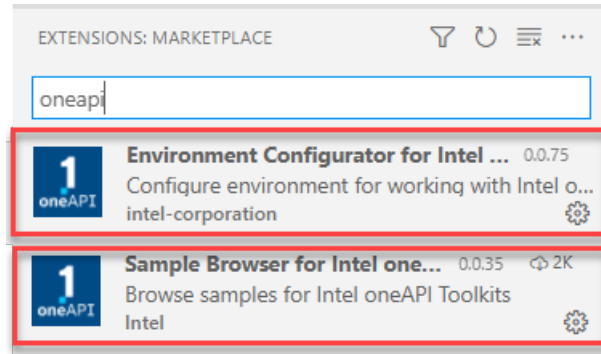
NOTE If you see errors about bad SSH file permissions when connecting, see [fixing SSH file permission errors](#). If your connection is hanging, you may need to enable TCP forwarding or respond to a server prompt. See [Tips and Tricks](#) for details.

4. Once logged in, add the **C/C++ extension** to your remote VS Code instance. This extension is required to debug a remote session using the VS Code debug interface.

Explore Samples Using Visual Studio Code*

Before working with oneAPI samples, it is recommended that you install the VS Code "Sample Browser for Intel oneAPI Toolkits" extension and the VS Code "Environment Configurator for Intel oneAPI Toolkits" extension.

Both can be quickly found in the VS Code extensions marketplace by typing "oneapi" into the marketplace search bar.



The steps below describe how to use these extensions to configure the oneAPI development environment for use with VS Code and use the Sample Browser to locate and create sample projects that help you learn how to use oneAPI features.

To configure the oneAPI environment in VS Code:

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI** to view options of the installed extensions.
3. Click on **Intel oneAPI: Initialize environment variables**.
4. Locate the `setvars.sh` | `oneapi-vars.sh` file:
 - Linux: By default in the Component Directory Layout, `setvars.sh` is located in `/opt/intel/oneapi/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
 - Linux: By default in the Unified Directory Layout, `oneapi-vars.sh` is located in `/opt/intel/oneapi/<toolkit-version>/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
 - Windows: By default in the Component Directory Layout, `setvars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\`
 - Windows: By default in the Unified Directory Layout, `oneapi-vars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\<toolkit-version>\`

If you customized the installation folder, `setvars.sh` | `oneapi-vars.sh` is in your custom folder.

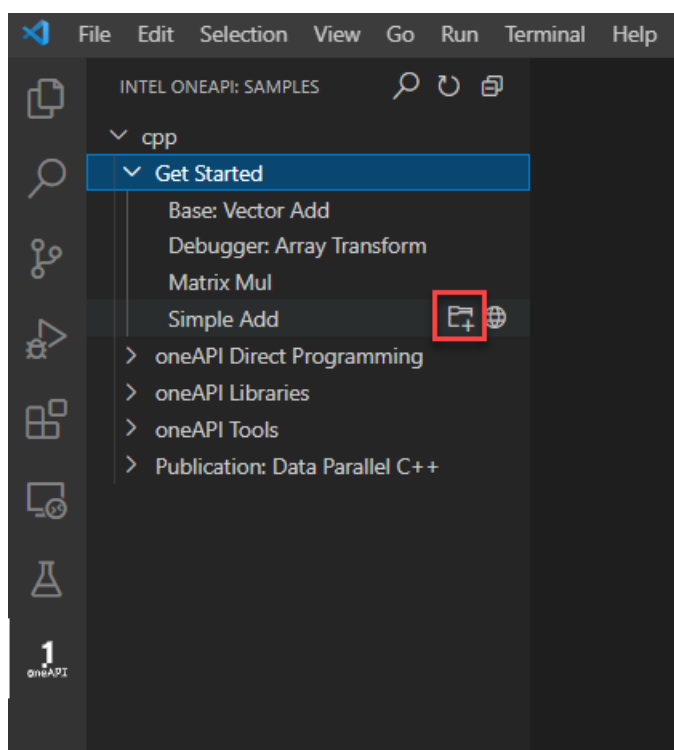
1. In the case of multiple folders in workspace, select the appropriate one. All tasks, launches, and terminals created from VS Code will now contain the oneAPI environment.

To browse oneAPI samples using VS Code:

1. Click on the oneAPI button on the left navigation to view samples. If you do not have the extension installed, search the Extensions Marketplace for "Sample Browser for Intel oneAPI".



2. A list of available samples will open in the left navigation.



3. To view the readme for the sample, click the



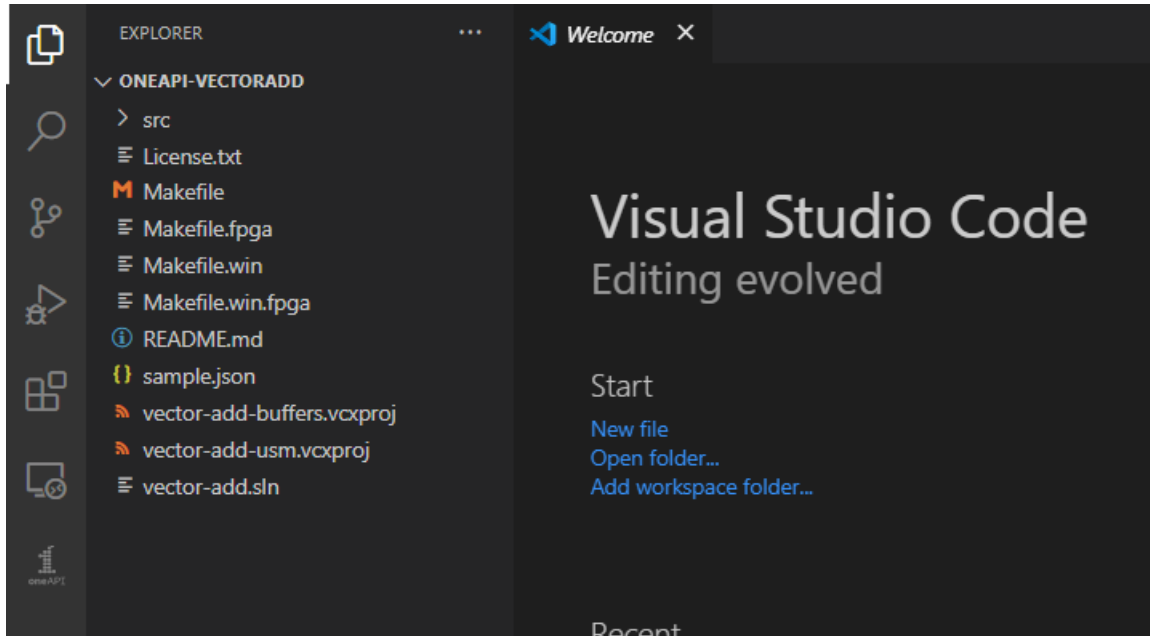
next to the sample. If you choose to build and run the sample, the readme will also be downloaded with the sample.

4. To build and run a sample, click the



to the right of the sample name.

5. Create a new folder for the sample. The sample will load in a new window:



6. Click README.md to view instructions for the sample.

NOTE Not all oneAPI sample projects use CMake. The README.md file for each sample specifies how to build the sample. We recommend that you check out the [CMake extension for VS Code](#) that is maintained by Microsoft.

Try Debugging (CPU and GPU Only) (Preview)

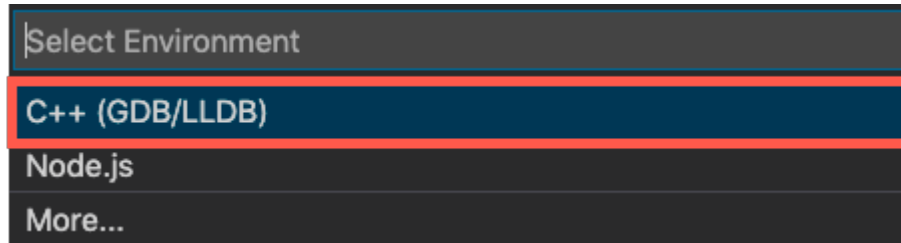
NOTE Intel® Distribution for GDB* does not currently support VS Code. You can use upstream gdb to debug. Debugging a local Windows host using VS Code is not supported. Debugging a local Linux host on CPU is supported using VS Code, but debugging on GPU requires a remote host. Debugging a remote Linux target from Windows or Linux host is supported using VS Code.

This section assumes that you can build your sample and have installed the [Microsoft VS Code C/C++ Extension](#). The C/C++ extension is required to configure the oneAPI C/C++ debugger.

The [Intel® oneAPI Base Toolkit](#) includes a special version of GNU* GDB (`gdb-oneapi`) designed to support oneAPI C/C++ applications. To debug your DPC++ application using this special debugger, you will need to make changes to the `.vscode/launch.json` configuration file.

1. Go to **Debug > Open Configurations**, and open the `launch.json` configuration settings.

NOTE If you are prompted to select a debug environment, choose **C++ (GDB/LLDB)**.



2. Copy the code shown below into your launch.json file, and replace the "program": property's value with the path to your project's executable (that is, the application that you are going to debug).

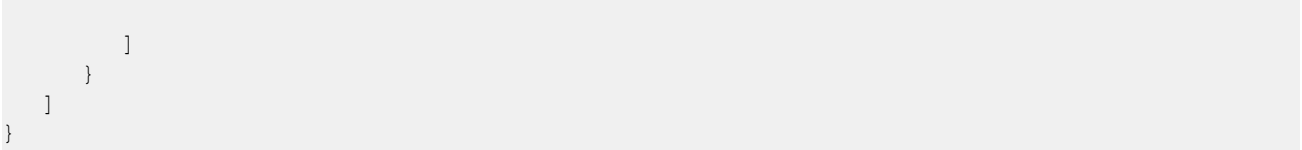
NOTE If VS Code doesn't recognize the application name, you may have to insert the full path and file name into the launch.json file's "program": property.

3. Add gdb-oneapi to your launch.json configuration's "miDebuggerPath": property.

NOTE The gdb-oneapi application should have been added to your path when you ran setvars.sh | oneapi-vars.sh to configure the oneAPI development environment, prior to starting VS Code. If you prefer, you can specify the full path and filename to the gdb-oneapi application in your launch.json file.

4. In some configurations, GDB may not be compatible with VS Code. If this happens, add the environment variable to disable `gdb-oneapi` support for GPU autolaunch. This can either be done in the environment prior to launching VS Code, or within the launch.json: `export INTELGT_AUTO_ATTACH_DISABLE=1`

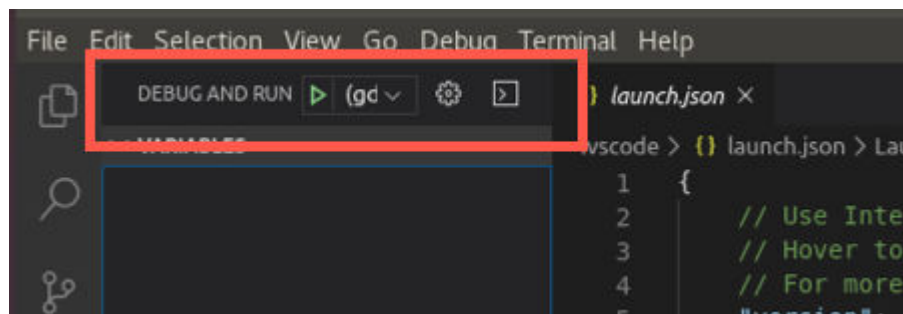
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/array-transform",
      "args": [
        "cpu"
      ],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "gdb-oneapi",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Disable target async",
          "text": "set target-async off",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```



5. Bring up the debug view by selecting the **Run** icon in the **Activity Bar**. You can also use the keyboard shortcut (**Ctrl+Shift+D**).



6. Start the run and debug session by clicking the green **DEBUG AND RUN** icon, or go to **Run > Start Debugging (F5)**.



Disconnect from a Remote Host

You can close the VS Code remote connection by selecting **File > Close Remote Connection** from the VS Code menu. Alternatively, click the colored **remote ssh** notification in the lower-left corner of the VS Code window and select **Close Remote Connection** from the list of **Remote-SSH** commands.

Kill Remote VS Code Server

If you need to kill the remote VS Code SSH server on your remote Linux target, perform the following steps:

1. Open the VS Code command palette (**View > Command Palette**).
2. Enter remote in the command palette dialog box.
3. Select **Remote-SSH: Kill VS Code Server on Host**.

You will be prompted to provide a `user@host` value or choose from a list of preconfigured SSH hosts in your `.ssh/config` file. Use the same credentials that you used to make your remote VS Code connection; VS Code will kill only the VS Code remote server instances that match that user.

NOTE VS Code provides no indication that completion of the kill command was successful. If you initiated a remote session with more than one remote user, you must perform the kill for each one.

Developing a Visual Studio Code* Project for SSH Development on Windows Subsystem for Linux*

Using Visual Studio Code* with Intel® oneAPI Toolkits

This guide assumes you are familiar with C/C++ development and the Visual Studio Code (VS Code) editor. If you are new to Visual Studio Code, review these VS Code documentation links:

- [Setting up Visual Studio Code](#)
- [Install the C/C++ Extension Pack for Visual Studio Code](#)
- [Visual Studio Code Remote Development Extension Pack](#)
- [Visual Studio Code User and Workspace Settings](#)
- [Getting Started with C++ in Visual Studio Code](#)
- [Getting Started with CMake Tools on Linux](#)
- [Debugging C++ in Visual Studio Code](#)

The following procedure will use a Remote Host connection enabling you to edit and debug your code using the Windows Subsystem for Linux* (WSL).

Prerequisites

- [Local host with Windows Subsystem for Linux* installed](#)
- [Local host with Visual Studio Code \(VS Code\) installed](#)
- [CMake and the Linux C/C++ development tools](#)
- [Visual Studio Code for Linux](#)
- [C/C++ Extension Pack for Visual Studio Code](#)
- A Linux distribution running on WSL with the [Intel® oneAPI Base Toolkit \(Base Kit\)](#) installed.

NOTE It is not necessary to install VS Code on your Linux target that is running on WSL. VS Code is only needed on your Windows development system. The Linux installed on WSL does not require a GUI in order to be used for Linux development.

Set oneAPI Environment Variables on the WSL Target Linux

In order to perform development on your remote WSL target, you must install (at minimum) the [Intel® oneAPI Base Toolkit](#) (Base Kit) and configure the oneAPI development environment variables on the WSL target Linux. The recommended installation method is [Installing via Linux* Package Managers](#).

1. Open a WSL terminal and install the Base Kit on to that target Linux system using the appropriate Linux package manager.

NOTE To get started, you need to install only the Base Kit on your WSL target; you can install additional oneAPI toolkits either now or at a later time.

2. Configure your WSL Linux target so that the oneAPI development environment script (`setvars.sh`) runs when VS Code initiates a connection.

NOTE Visual Studio Code does not currently provide a mechanism to automatically run scripts on your remote Linux target when it interacts with the WSL system (for example, running `setvars.sh` automatically). There are a variety of ways to get around this issue, only one of which is presented here. See the [Bash Startup Files](#) man page for more options.

Add the following shell script lines to your remote Linux system's `/etc/profile` script. This location (or in an `/etc/profile.d/` script) will ensure that all users of your WSL target development system will run the `setvars.sh` environment script when they connect using Visual Studio Code:

```
. /opt/intel/oneapi/setvars.sh &>/dev/null
```

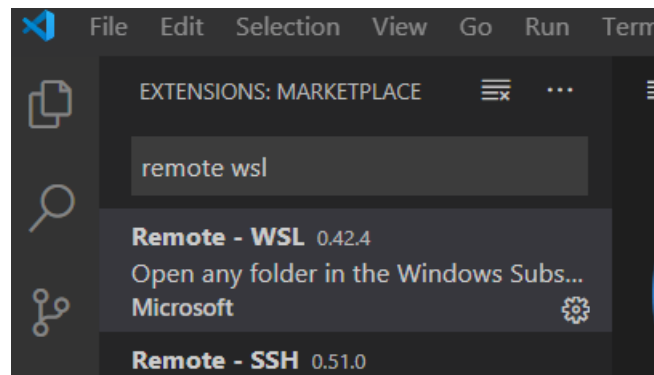
The above assumes that you have installed your oneAPI toolkits on the WSL system in the default "root/sudo" installation location (`/opt/intel/oneapi/`). If you have installed the oneAPI tools in a different location on your remote Linux system, you will have to adjust the path to `setvars.sh`.

NOTE See [Configure Your CPU or GPU System](#) for more details regarding configuration of the oneAPI environment on a Linux system.

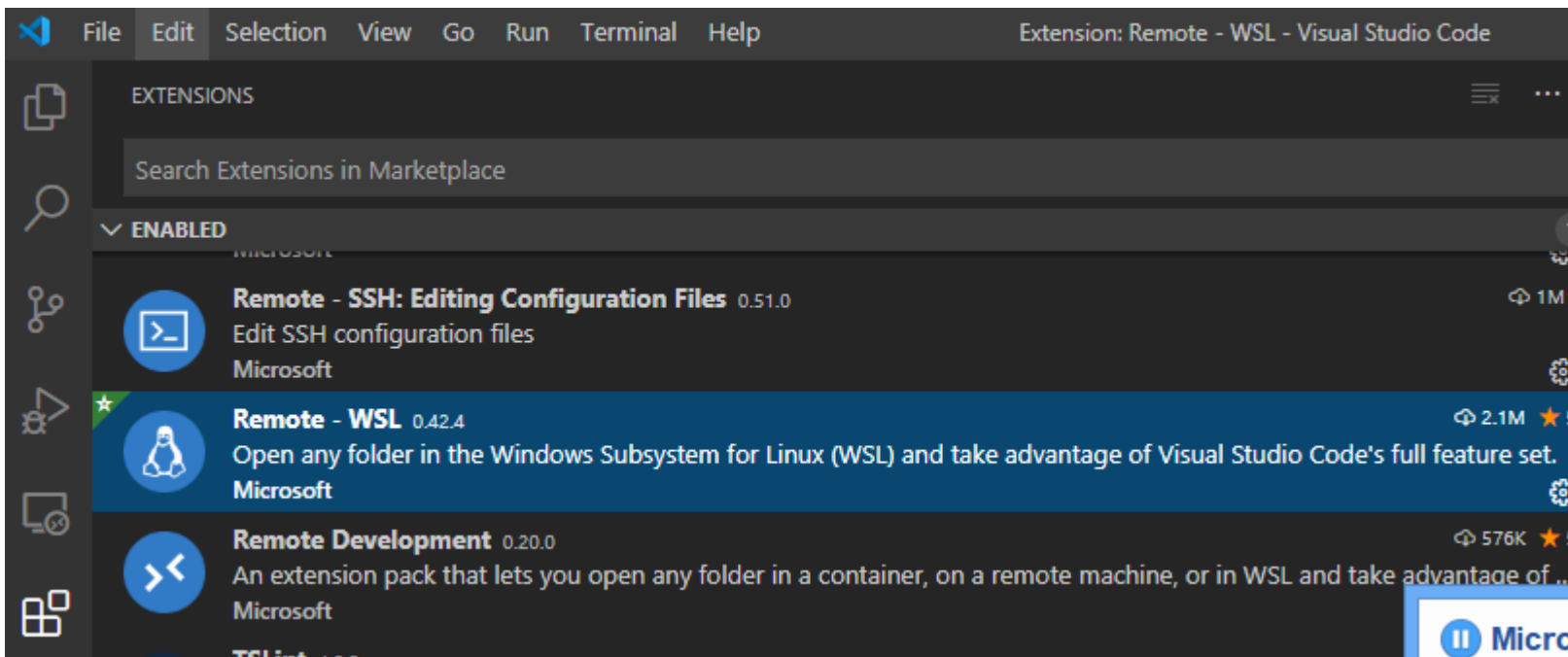
Configure Visual Studio Code on Your Local Host

You must install the VS Code "Remote - WSL" extension into your local copy of VS Code. This extension facilitates development on the WSL* system.

1. Install the **Remote - WSL** extension by Microsoft. Click the **Extensions** icon and search for "remote wsl" in the search bar.



2. Click **install**. After installation, you see **Remote – WSL** in the **Installed Extensions** list.



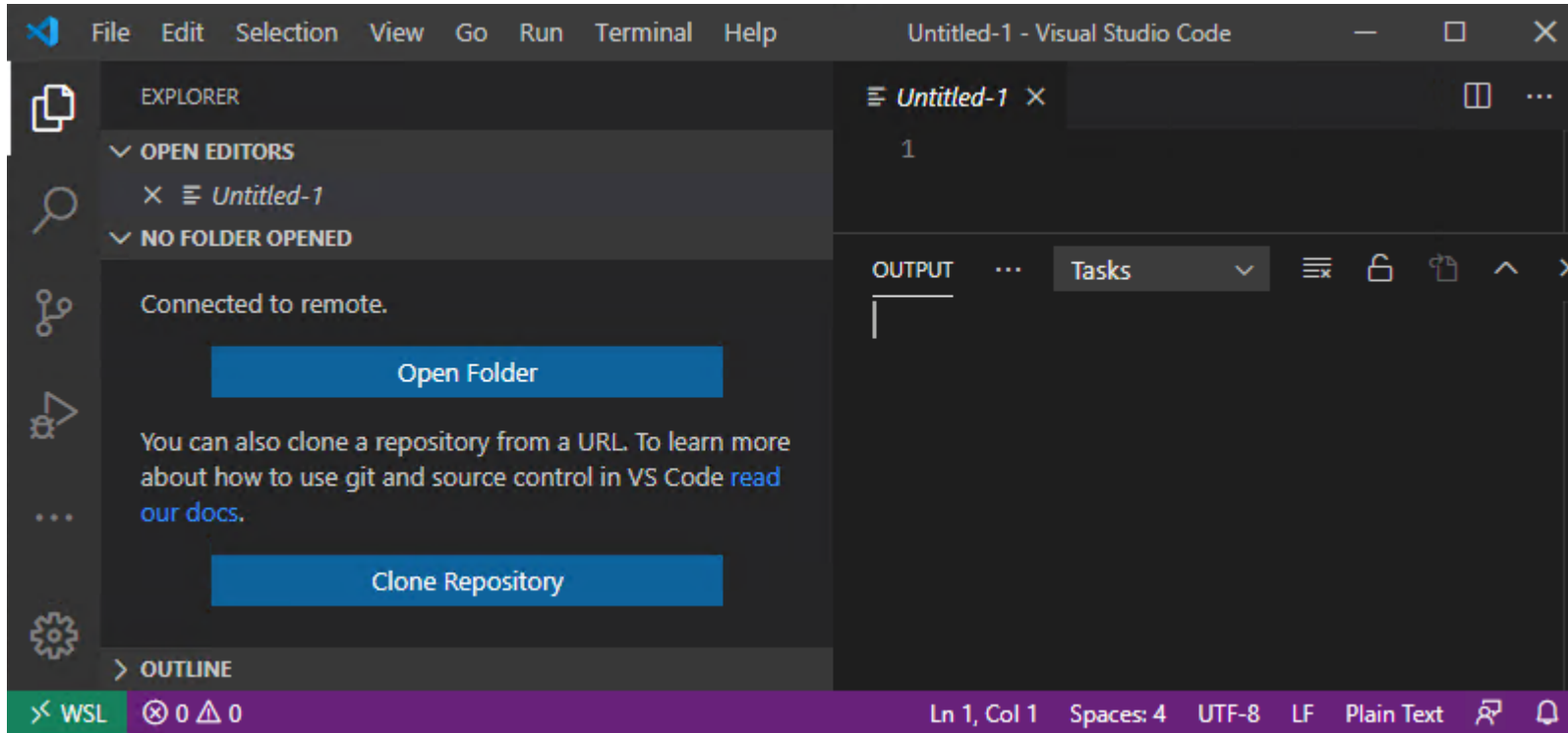
Connect to Your WSL Linux System

See the Visual Studio Code [Remote Development in WSL](#) documentation for detailed instructions, requirements, and additional information.

1. Open a WSL terminal for your desired Linux distribution. From within the terminal, invoke VS Code from the command line to start a local instance connected to WSL.

```
user@host:~$ code
```

2. A new VS Code window opens and is connected to the WSL system.

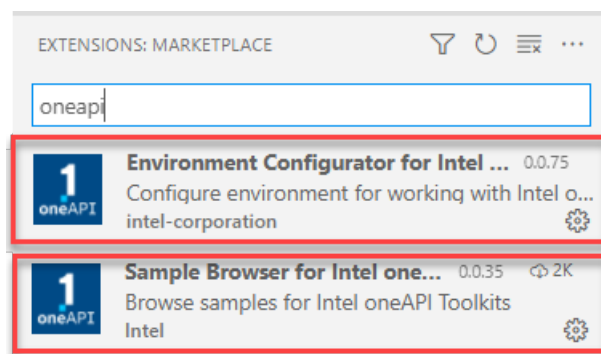


3. Once connected, add the **C/C++ extension** to the VS Code instance. This extension is required to debug a remote session using the VS Code debug interface.

Explore Samples Using Visual Studio Code*

Before working with oneAPI samples, it is recommended that you install the VS Code “Sample Browser for Intel oneAPI Toolkits” extension and the VS Code “Environment Configurator for Intel oneAPI Toolkits” extension.

Both can be quickly found in the VS Code extensions marketplace by typing “oneapi” into the marketplace search bar.



The steps below describe how to use these extensions to configure the oneAPI development environment for use with VS Code and use the Sample Browser to locate and create sample projects that help you learn how to use oneAPI features.

To configure the oneAPI environment in VS Code:

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI** to view options of the installed extensions.
3. Click on **Intel oneAPI: Initialize environment variables**.
4. Locate the setvars | oneapi-vars file:

- Linux: By default in the Component Directory Layout, `setvars.sh` is located in `/opt/intel/oneapi/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
- Linux: By default in the Unified Directory Layout, `oneapi-vars.sh` is located in `/opt/intel/oneapi/<toolkit-version>/` for root or sudo installations, or located in `~/intel/oneapi/` for local user installations.
- Windows: By default in the Component Directory Layout, `setvars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\`
- Windows: By default in the Unified Directory Layout, `oneapi-vars.bat` is located in `C:\Program Files(x86)\Intel\oneAPI\<toolkit-version>\`

If you customized the installation folder, `setvars` | `oneapi-vars` is in your custom folder.

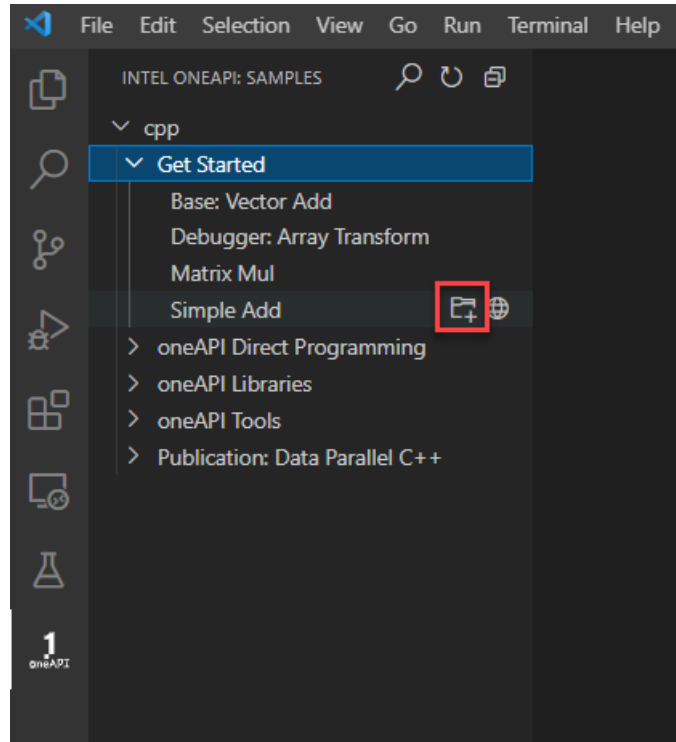
5. In the case of multiple folders in workspace, select the appropriate one. All tasks, launches, and terminals created from VS Code will now contain the oneAPI environment.



To browse oneAPI samples using VS Code:

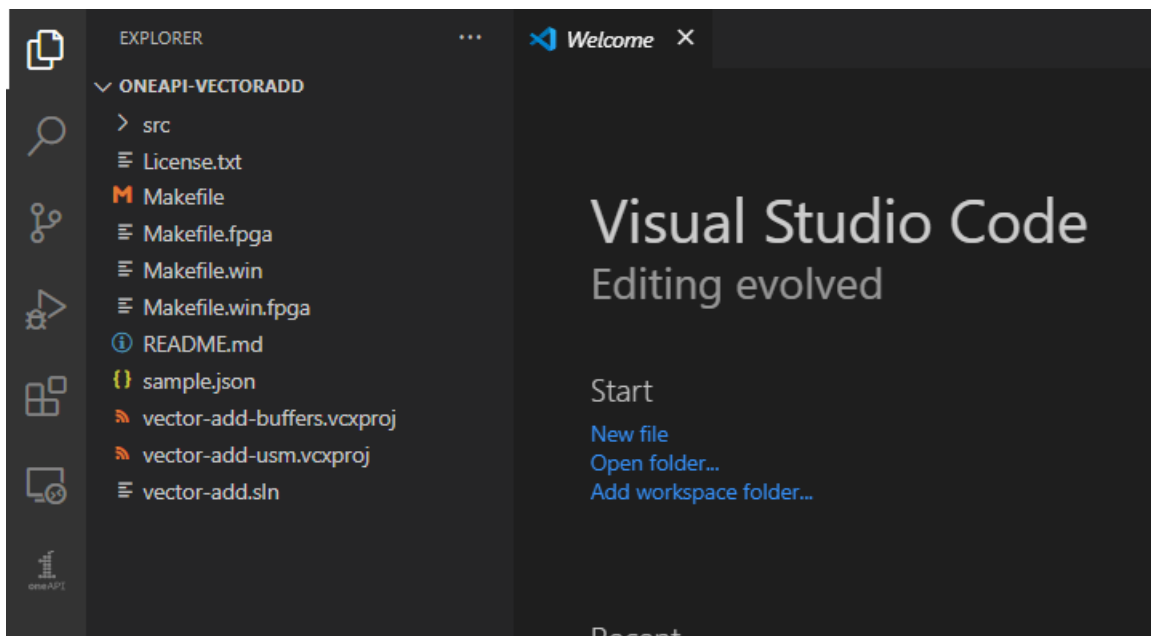
1. Click on the oneAPI button on the left navigation to view samples. If you do not have the extension installed, search the Extensions Marketplace for "Sample Browser for Intel oneAPI".



2. A list of available samples will open in the left navigation.



1. To view the readme for the sample, click the  next to the sample. If you choose to build and run the sample, the readme will also be downloaded with the sample.
2. To build and run a sample, click the  to the right of the sample name.
3. Create a new folder for the sample. The sample will load in a new window:



4. Click README.md to view instructions for the sample.

NOTE Not all oneAPI sample projects use CMake. The README.md file for each sample specifies how to build the sample. We recommend that you check out the [CMake extension for VS Code](#) that is maintained by Microsoft.

Try Debugging (CPU and GPU Only) (Preview)

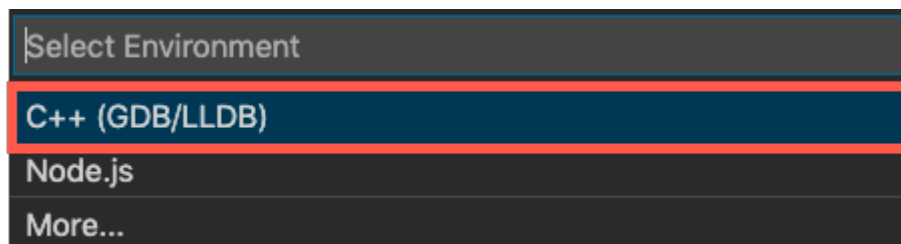
NOTE Intel® Distribution for GDB* does not currently support VS Code. You can use upstream gdb to debug. Debugging a local Windows host using VS Code is not supported. Debugging a local Linux host on CPU is supported using VS Code, but debugging on GPU requires a remote host. Debugging a remote Linux target from Windows or Linux host is supported using VS Code.

This section assumes that you can build your sample and have installed the [Microsoft VS Code C/C++ Extension](#). The C/C++ extension is required to configure the oneAPI C/C++ debugger.

The [Intel® oneAPI Base Toolkit](#) includes a special version of GNU* GDB (`gdb-oneapi`) designed to support oneAPI C/C++ applications. To debug your DPC++ application using this special debugger, you will need to make changes to the `.vscode/launch.json` configuration file.

1. Go to **Debug > Open Configurations**, and open the `launch.json` configuration settings.

NOTE If you are prompted to select a debug environment, choose **C++ (GDB/LLDB)**.



2. Copy the code shown below into your `launch.json` file, and replace the `"program":` property's value with the path to your project's executable (that is, the application that you are going to debug).

NOTE If VS Code doesn't recognize the application name, you may have to insert the full path and file name into the `launch.json` file's `"program":` property.

3. Add `gdb-oneapi` to your `launch.json` configuration's `"miDebuggerPath":` property.

NOTE The `gdb-oneapi` application should have been added to your path when you ran `setvars.sh | oneapi-vars.sh` to configure the oneAPI development environment, prior to starting VS Code. If you prefer, you can specify the full path and filename to the `gdb-oneapi` application in your `launch.json` file.

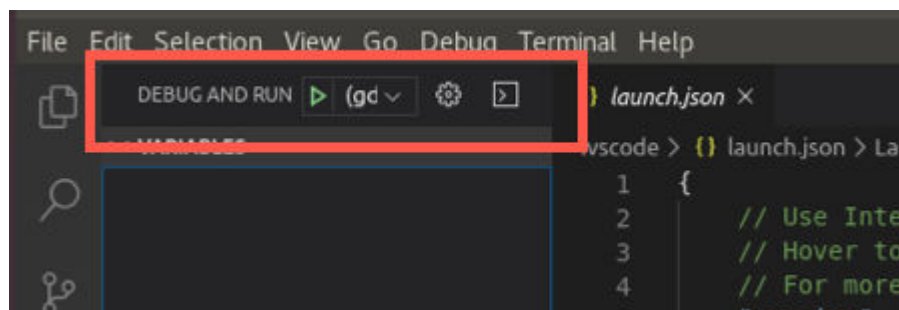
4. In some configurations, GDB may not be compatible with VS Code. If this happens, add the environment variable to disable `gdb-oneapi` support for GPU autolaunch. This can either be done in the environment prior to launching VS Code, or within the launch.json: `export INTELGT_AUTO_ATTACH_DISABLE=1`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/array-transform",
      "args": [
        "cpu"
      ],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "gdb-oneapi",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Disable target async",
          "text": "set target-async off",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

5. Bring up the debug view by selecting the **Run** icon in the **Activity Bar**. You can also use the keyboard shortcut (**Ctrl+Shift+D**).



6. Start the run and debug session by clicking the green **DEBUG AND RUN** icon, or go to **Run > Start Debugging (F5)**.



Using Visual Studio Code* with Intel® oneAPI Toolkits in a Docker* Container

Using Visual Studio Code* with Intel® oneAPI Toolkits

Prerequisites:

NOTE For FPGA development, use the Intel® oneAPI DPC++/C++ Compiler.

If you have not configured your system and built and run a sample project, please refer to the appropriate toolkit Get Started guide and complete those steps:

- [Install an Intel® oneAPI toolkit](#)
- Build and run a sample project with one of these toolkits:
 - [Intel® oneAPI Base Toolkit](#)
 - [Intel® HPC Toolkit](#)
 - [Intel® oneAPI IoT Toolkit](#)

When you have completed those steps, see [Developing a Visual Studio Code Project in a Docker Container](#).

- [Developing a Visual Studio Code Project in a Docker Container](#)

- [Configure the Container](#)
- [Build a Sample Project](#)
- [Try Debugging \(CPU and GPU Only\) \(Preview\)](#)
- [Disconnect from the Container](#)

Developing a Visual Studio Code Project in a Docker Container

Using Visual Studio Code* with Intel® oneAPI Toolkits

The Intel® oneAPI Toolkits support these compilers:

- Intel® oneAPI DPC++ Compiler
- Intel® Fortran Compiler
- Intel® C++ Compiler

Containers allow you to set up and configure environments for building, running and profiling oneAPI applications and distribute them using images:

- You can install an image containing an environment pre-configured with all the tools you need, then develop within that environment.
- You can save an environment and use the image to move that environment to another machine without additional setup.
- You can prepare containers with different sets of languages and runtimes, analysis tools, or other tools, as needed.

Download Docker* Image

You can download a Docker* image from the [Containers Repository](#).

NOTE The Docker image is ~5 GB and can take ~15 minutes to download. It will require 25 GB of disk space.

```
image=intel/oneapi-basekit
docker pull "$image"
```

Singularity Containers

Build a Singularity image using a [Singularity file](#).

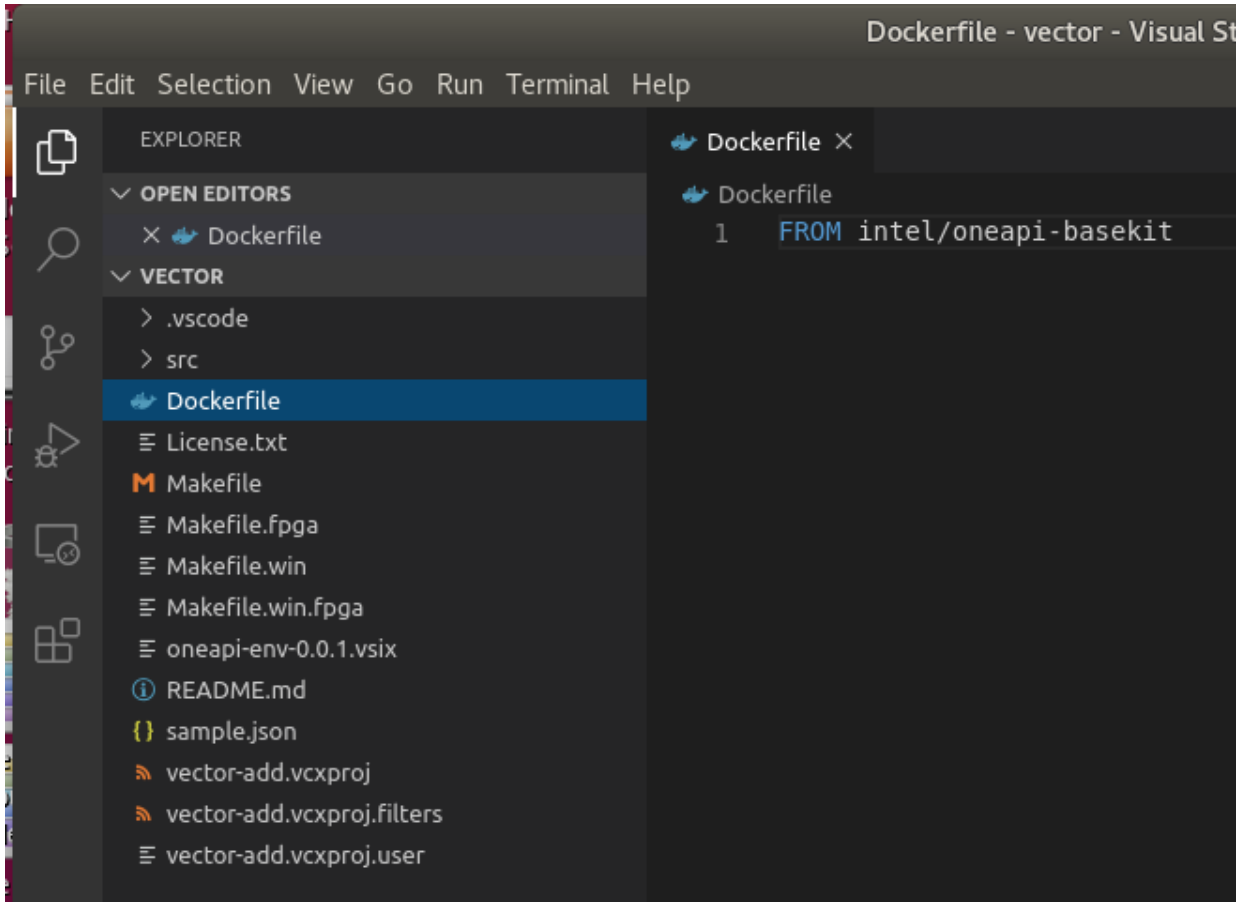
Create the Container

1. Inside the Intel oneAPI project you just created, create a new file named *Dockerfile* with the following contents.

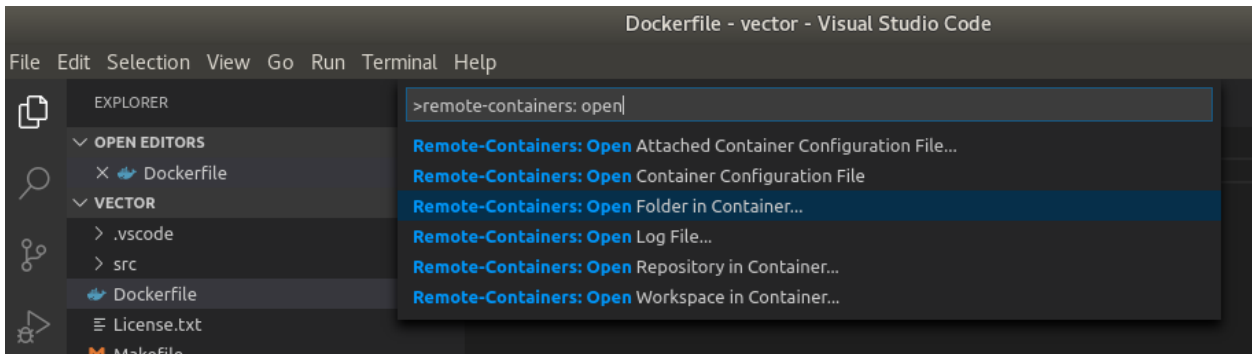
```
FROM intel/oneapi-basekit
```

This Dockerfile will create a local Docker image on your system based on a pre-configured oneAPI Docker container stored in Docker Hub.

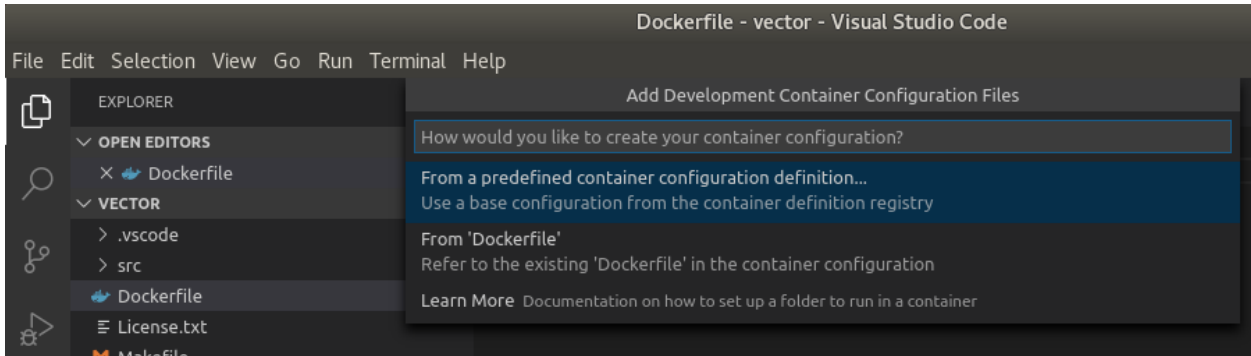
NOTE A complete list of preconfigured Docker containers for Intel oneAPI can be found here: <https://hub.docker.com/r/intel/oneapi>.



2. Open the Command Palette (Ctrl-Shift-P) and run **Remote-Containers: Open Folder in Remote Container** .



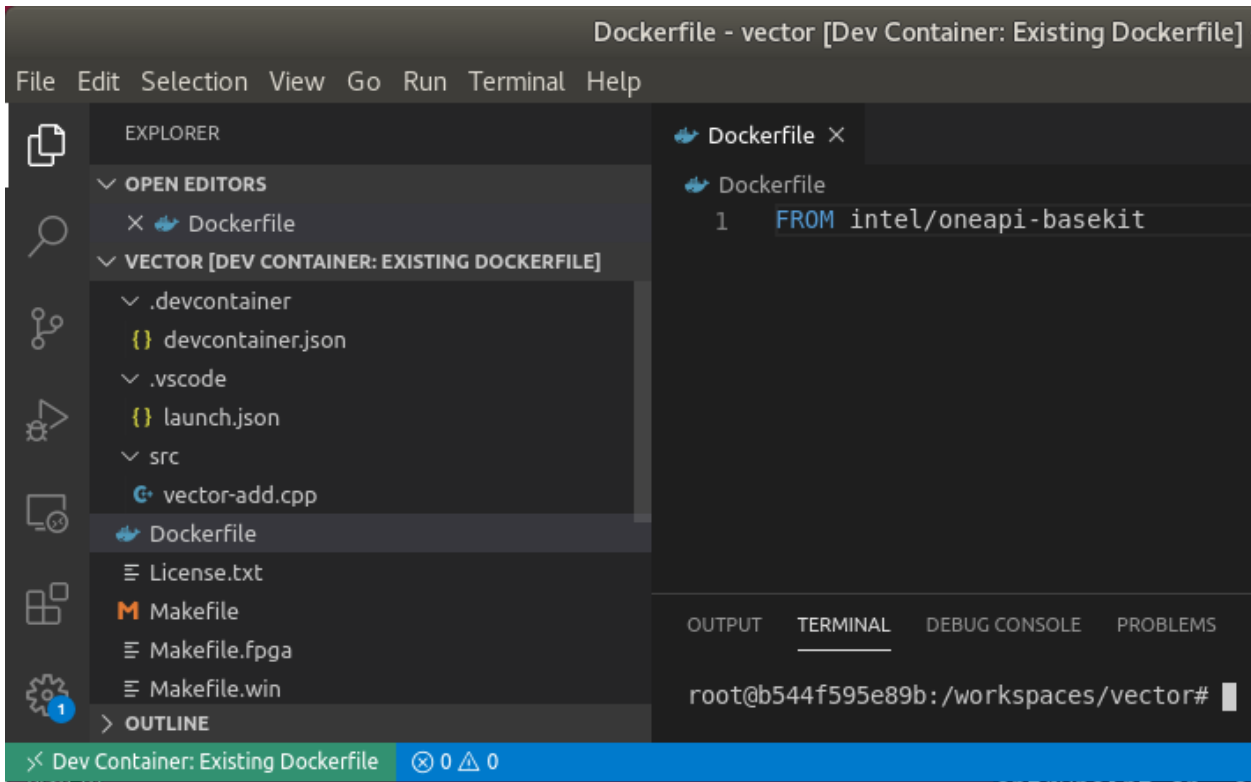
3. A folder selection window will appear. Choose the folder containing your Intel oneAPI project.
4. When prompted, choose **From 'Dockerfile'** .



5. VS Code will automatically use the Dockerfile you just created to download the image, create the container, and open the project inside of the container.

NOTE The duration of this step depends on the time it takes to download the image and create the container. Refer to the [Using Containers](#) documentation for more details.

6. VS Code will now show that it is connected to the container.



Configure the Container

In order to debug inside of the container, you must enable debugging flags and configure the Intel oneAPI development environment variables within the container. This can be done by editing the `.devcontainer/devcontainer.json` file.

NOTE Debugging is only available for CPU applications. Debugging Intel oneAPI GPU applications in VS Code is not supported at this time.

1. An extension is required to debug Intel oneAPI projects inside of a container. The [Microsoft VS Code C/C++ Extension](#) is required to configure the GDB debugger. This can be installed automatically by adding it to the container configuration file. Edit the "extensions" line in `.devcontainer/devcontainer.json`:

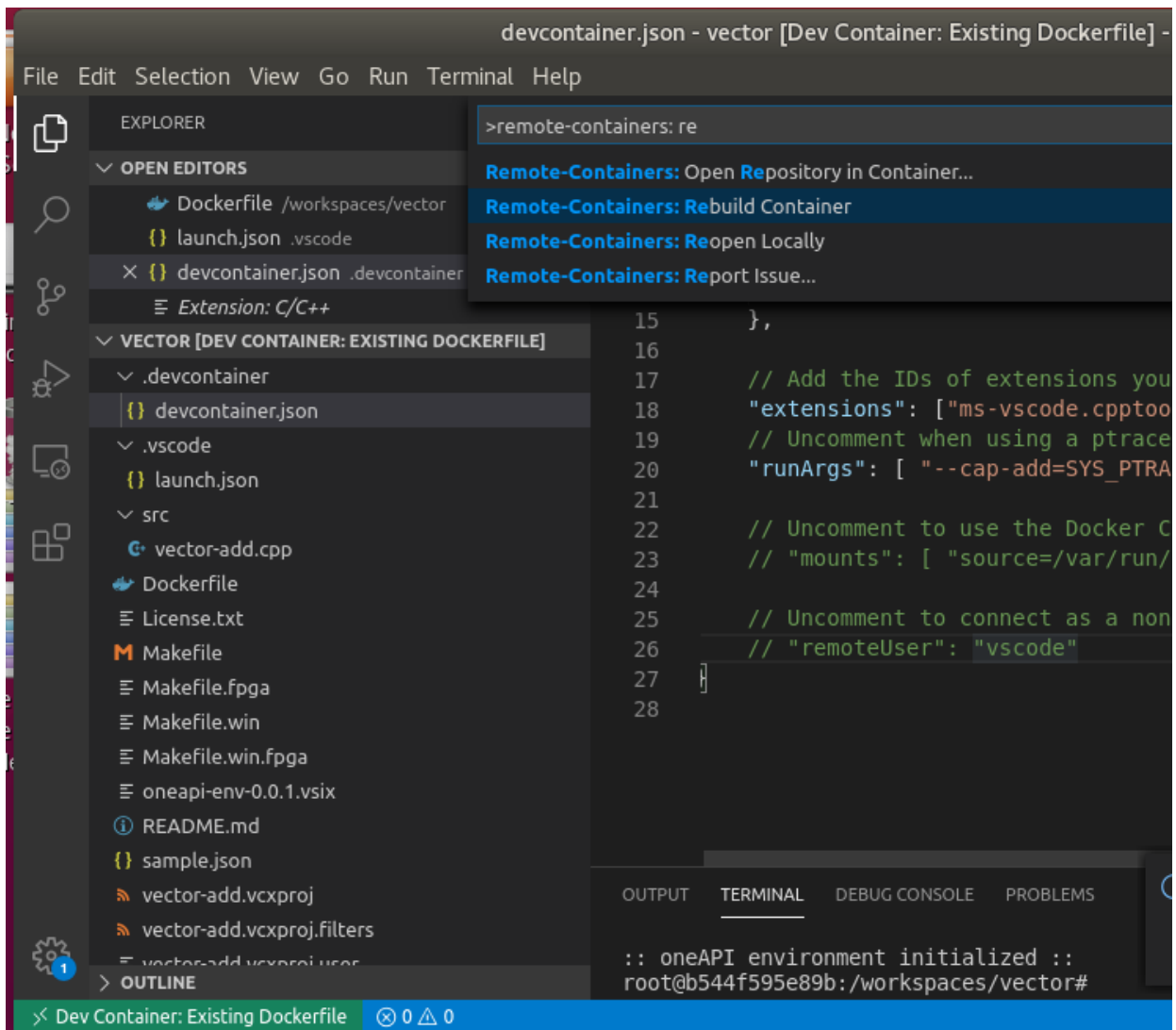
```
// Add the IDs of extensions you want installed when the container is created.
"extensions": ["ms-vs-code.cpptools"],
```

NOTE The Intel extension is not POR and it is not available in the Marketplace.

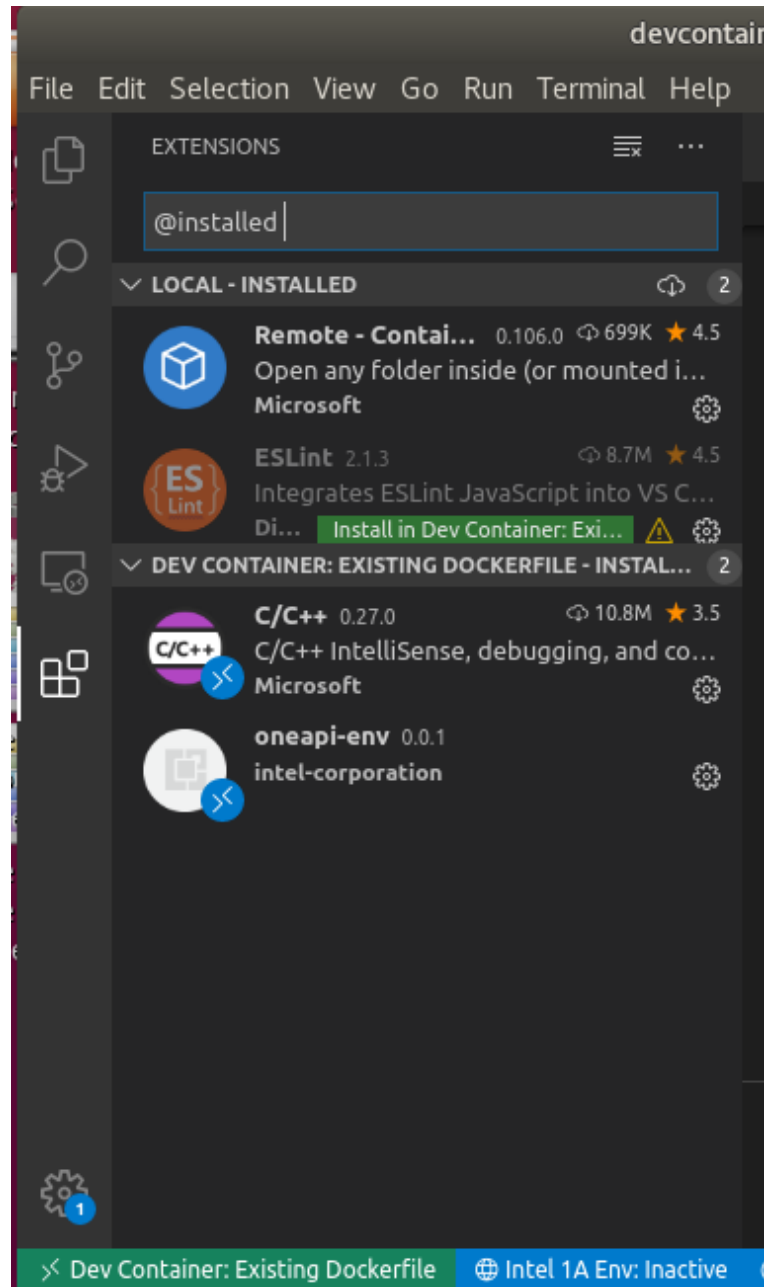
2. If you will be using a ptrace-based debugger for programming languages such as C++, Go, and Rust, you will need to enable debugging within the container. Uncomment the "runArgs" line in `.devcontainer/devcontainer.json`:

```
// Uncomment when using a ptrace-based debugger like C++, Go, and Rust
"runArgs": [ "--cap-add=SYS_PTRACE", "--security-opt", "seccomp=unconfined" ],
```

3. In order to apply these changes to the container, it must be rebuilt. Open the Command Palette and run **Remote-Containers: Rebuild Container**.



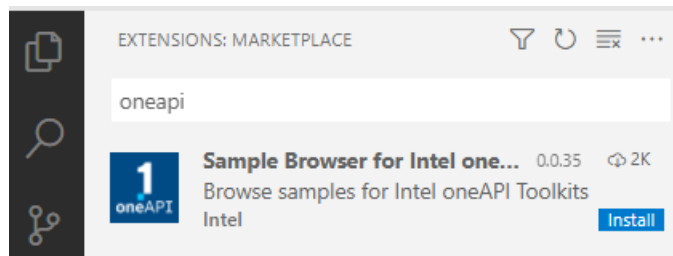
4. The container will be rebuilt and show that the extensions have been installed.



Build a Sample Project

To build the project, follow the instructions in README.md. Note that when you open a Terminal in VS Code (**Ctrl-Shift-**) while connected to a container, it opens a login terminal inside of the container.

Before working with oneAPI samples, it is recommended that you install the VS Code “Sample Browser for Intel oneAPI Toolkits” extension. It can be found in the VS Code extensions marketplace by typing “oneapi” into the marketplace search bar.

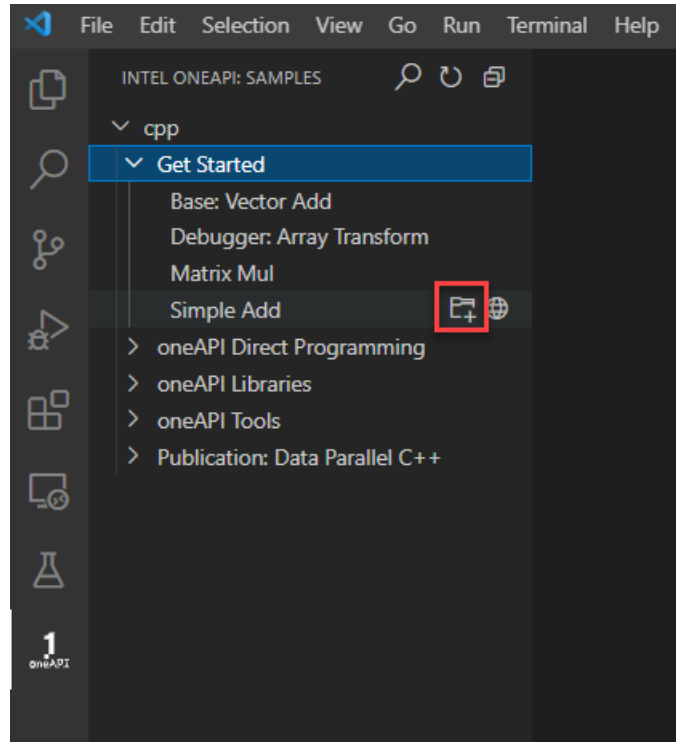




After the extension is installed, browse oneAPI samples using VS Code:

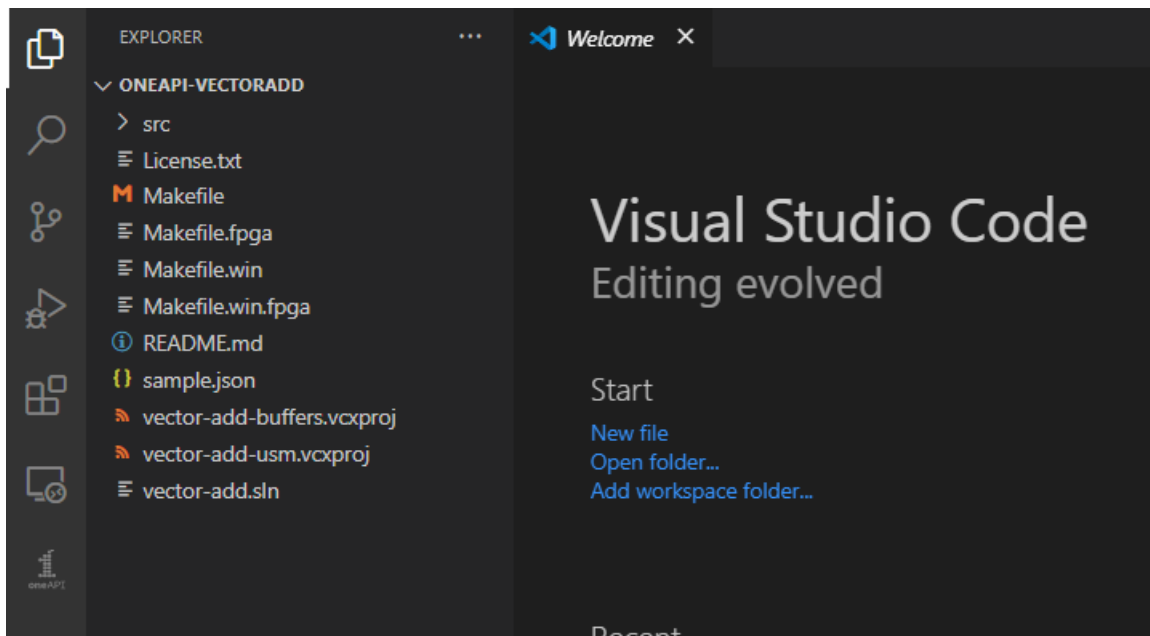
1. Click on the oneAPI button on the left navigation to view samples. If you do not have the extension installed, search the Extensions Marketplace for "Sample Browser for Intel oneAPI Toolkits".



2. A list of available samples will open in the left navigation.



3. To view the readme for the sample, click the  next to the sample. If you choose to build and run the sample, the readme will also be downloaded with the sample.
4. To build and run a sample, click the  to the right of the sample name.
5. Create a new folder for the sample. The sample will load in a new window:



6. Click README.md to view instructions for the sample.

NOTE Not all oneAPI sample projects use CMake. The README.md file for each sample specifies how to build the sample. We recommend that you check out the [CMake extension for VS Code](#) that is maintained by Microsoft.

Try Debugging (CPU and GPU Only) (Preview)

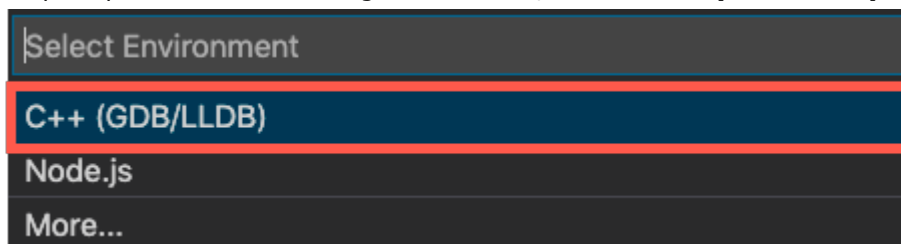
NOTE Intel® Distribution for GDB* does not currently support VS Code. You can use upstream gdb to debug. Debugging a local Windows host using VS Code is not supported. Debugging a local Linux host on CPU is supported using VS Code, but debugging on GPU requires a remote host. Debugging a remote Linux target from Windows or Linux host is supported using VS Code.

This section assumes that you can build your sample and have installed the [Microsoft VS Code C/C++ Extension](#). The C/C++ extension is required to configure the oneAPI C/C++ debugger.

The [Intel® oneAPI Base Toolkit](#) includes a special version of GNU* GDB (`gdb-oneapi`) designed to support oneAPI C/C++ applications. To debug your DPC++ application using this special debugger, you will need to make changes to the `.vscode/launch.json` configuration file.

1. Go to **Debug > Open Configurations**, and open the `launch.json` configuration settings.

NOTE If you are prompted to select a debug environment, choose **C++ (GDB/LLDB)**.



1. Copy the code shown below into your `launch.json` file, and replace the `"program":` property's value with the path to your project's executable (that is, the application that you are going to debug).

NOTE If VS Code doesn't recognize the application name, you may have to insert the full path and file name into the `launch.json` file's `"program":` property.

1. Add `gdb-oneapi` to your `launch.json` configuration's `"miDebuggerPath":` property.

NOTE The `gdb-oneapi` application should have been added to your path when you ran `setvars.sh | oneapi-vars.sh` to configure the oneAPI development environment, prior to starting VS Code. If you prefer, you can specify the full path and filename to the `gdb-oneapi` application in your `launch.json` file.

1. In some configurations, GDB may not be compatible with VS Code. If this happens, add the environment variable to disable `gdb-oneapi` support for GPU autolaunch. This can either be done in the environment prior to launching VS Code, or within the launch.json: `export`

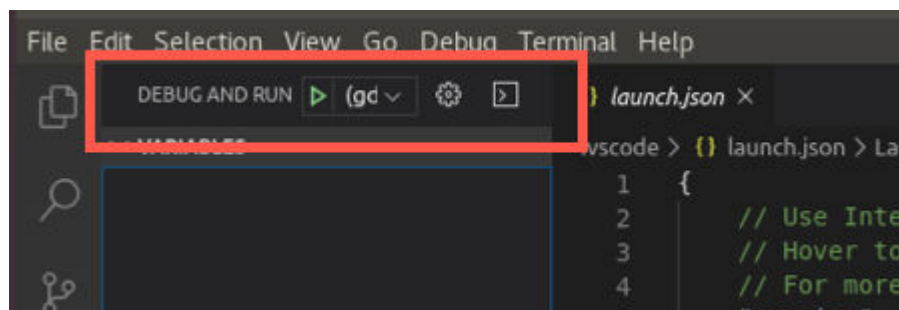
`INTELGTT_AUTO_ATTACH_DISABLE=1`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/array-transform",
      "args": [
        "cpu"
      ],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "gdb-oneapi",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Disable target async",
          "text": "set target-async off",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

2. Bring up the debug view by selecting the **Run** icon in the **Activity Bar**. You can also use the keyboard shortcut (**Ctrl+Shift+D**).



3. Start the run and debug session by clicking the green **DEBUG AND RUN** icon, or go to **Run > Start Debugging (F5)**.



Disconnect from the Container

You can close the VS Code connection to the container by selecting **File > Close Remote Connection** from the VS Code menu. Alternatively, click the colored **Dev Container** notification in the lower-left corner of the VS Code window and select **Close Remote Connection** from the list of **Remote-Container** commands.

Intel® oneAPI Extensions for Visual Studio Code*

Using Visual Studio Code* with Intel® oneAPI Toolkits

There are multiple extensions available for Intel® oneAPI Toolkits. The extensions may be installed separately, or you can install the **Extension Pack for Intel® oneAPI Toolkits** to install all of the oneAPI extensions for Visual Studio Code.

Select one of the extensions below to learn more:

- [Code Sample Browser for Intel® oneAPI Toolkits](#)
- [Environment Configurator for Intel® oneAPI Toolkits*](#)
 - [Set oneAPI Environment](#)
 - [Clear Environment Variables](#)
 - [Switch Environment](#)
- [Analysis Configurator for Intel® oneAPI Toolkits*](#)

- [Preparing Tasks](#)
- [Prepare Launch Configuration](#)
- [Building a Single cpp File](#)
- [Using Intel Analysis Tools](#)
- [How to Use IntelliSense for Code Developed with Intel oneAPI Toolkits](#)
- [Configure C++ Properties](#)
- [Display Tool Tips](#)
- [GDB GPU Support for Intel® oneAPI Toolkits](#)
 - [Prepare Launch Configuration](#)
 - [SIMD View](#)
 - [ThreadID](#)
 - [Name](#)
 - [SIMD Lanes](#)
 - [SIMD Lane Specific Breakthroughs](#)
 - [Symbolic Indication of SIMD Lanes](#)
 - [Choose SIMD Lane](#)
 - [Hardware Info](#)
 - [Differences between GDB and GDB-oneapi](#)
- [DevCloud Connector for Intel® oneAPI Toolkits*](#)
- [Search for a Specific Sample](#)

Environment Configurator for Intel® oneAPI Toolkits*

Using Visual Studio Code with Intel® oneAPI Toolkits*

The Environment Configurator for Intel® oneAPI Toolkits extension is a lightweight extension that provides control of the oneAPI development environment and makes it easier to configure oneAPI projects for build, run, and debug.

To install the extension:

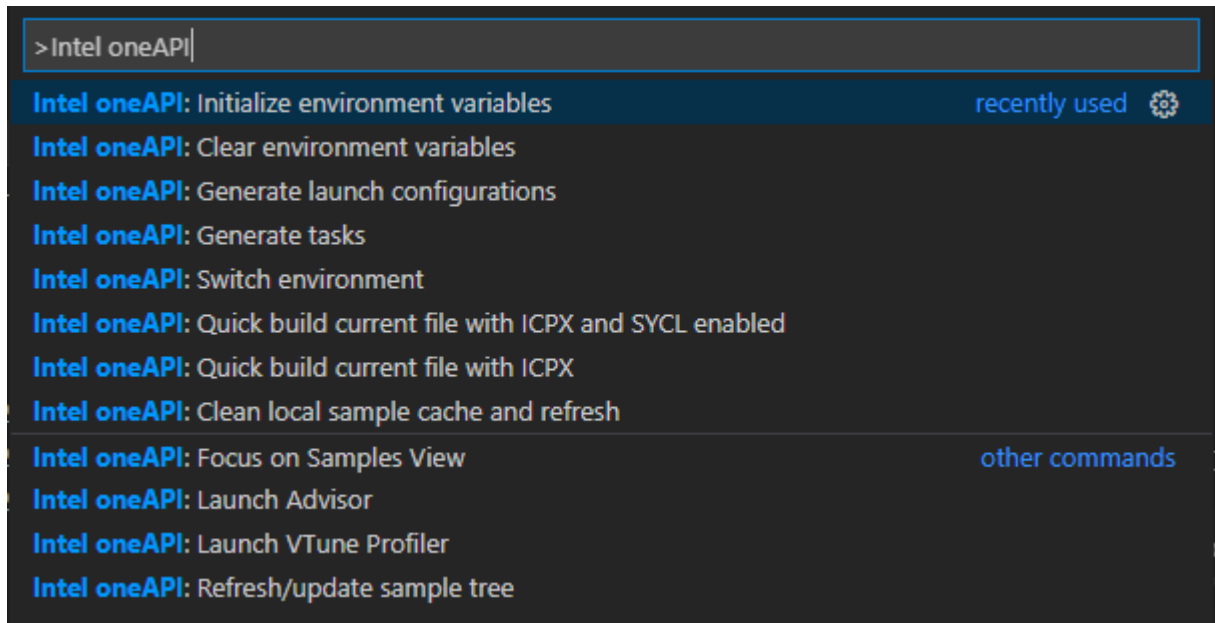
1. Open the Extension Marketplace in VS Code and search for *oneAPI*.
2. Install the extension titled **Environment Configurator for Intel oneAPI Toolkits**.

Note: You may also install this extension with the **Extension Pack for Intel® oneAPI Toolkits**.

Set oneAPI Environment

With this extension, you can set your environment from VS Code:

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI** to view options of the installed extensions.



3. Click on **Intel oneAPI: Initialize environment variables**. If you see a prompt to locate a file, navigate to where your `setvars` or `oneapi-vars` script is located:
 - Linux (Component Directory Layout): `<install_dir>/intel/oneapi/`.
 - Windows (Component Directory Layout): `<install_dir>\Intel\oneAPI\`
 - Linux (Unified Directory Layout): `<install_dir>/intel/oneapi/<toolkit-version>/`.
 - Windows (Unified Directory Layout): `<install_dir>\Intel\oneAPI\<toolkit-version>\`
4. All tasks, launches, and terminals created from VS Code will now contain the oneAPI environment. If you make a new connection such as ssh, repeat the steps above to configure the environment again.

Next, run a sample project using one of these methods: [Developing a Visual Studio Code Project](#), [Developing a Visual Studio Code Project for SSH Development](#), [Developing a Visual Studio Code Project in a Docker Container](#) or read about the [Analysis Configurator for Intel® oneAPI Toolkits*](#) extension.

Clear Environment Variables

To clear environment variables:

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI: Clear environment variables** and click on that command.

Switch Environment

If you are using multiple workspaces, you can quickly switch environments:

1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI: Switch environment** and click on that command.

Analysis Configurator for Intel® oneAPI Toolkits*

Using the Visual Studio Code with Intel® oneAPI Toolkits*

This is an extension for seamless integration with Intel® analysis tools, including the Intel® VTune™ Profiler and Intel® Advisor.

- [Intel® VTune™ Profiler](#) is a performance profiling tool that provides advanced sampling and profiling techniques to quickly analyze code, isolate issues and deliver insights for optimizing performance on modern processors.
- [Intel® Advisor](#) is for software architects and developers who need the right information and recommendations to make the best design and optimization decisions for efficient vectorization, threading, and offloading to accelerators.

To install the extension:

1. Open the Extension Marketplace in VS Code and search for *oneAPI*.
2. Install the extension titled **Analysis Configurator for Intel oneAPI Toolkits**.

Note: You may also install this extension with the **Extension Pack for Intel® oneAPI Toolkits**.

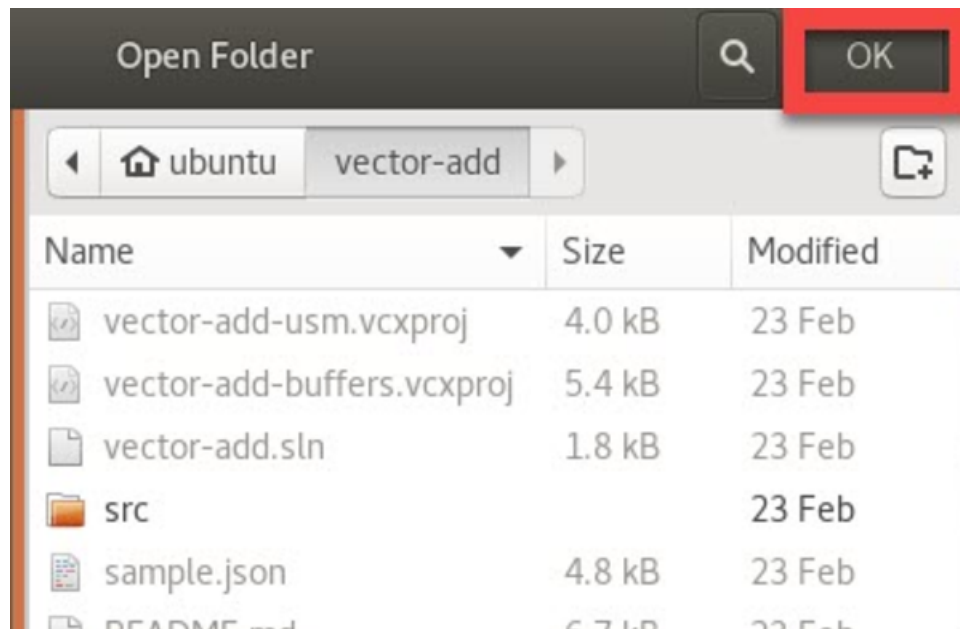
With this extension, you can perform multiple functions:

- Prepare Tasks
- Prepare Launch Configuration
- Use Intel Analysis Tools
- Build a single cpp file
- Use IntelliSense for Code Developed with Intel oneAPI Toolkits
- Configure C++ Properties
- Display tool tips

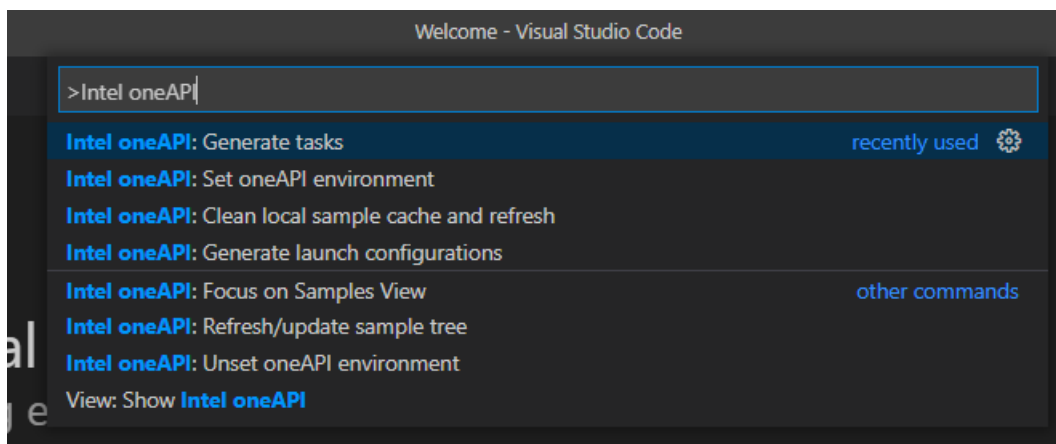
Preparing Tasks

This extension enables the ability to prepare tasks from make or cmake files:

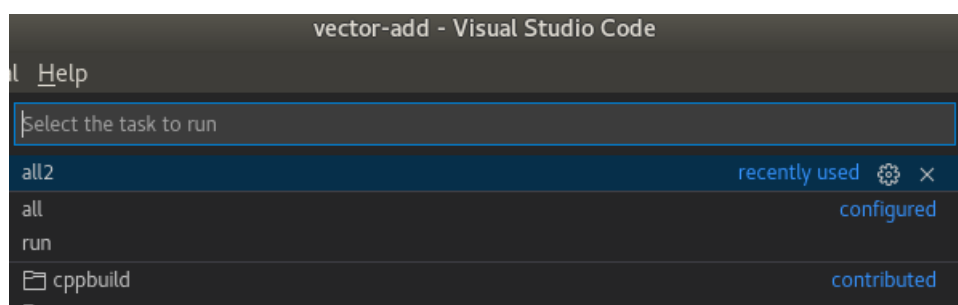
1. Using the VS Code explorer, click **File>Open Folder**.
2. Navigate to the folder where your project is located and click **OK**.



3. Press **Ctrl+Shift+P** (or View -> Command Palette...) to open the Command Palette.
4. Type Intel oneAPI and select Intel oneAPI: Generate tasks.



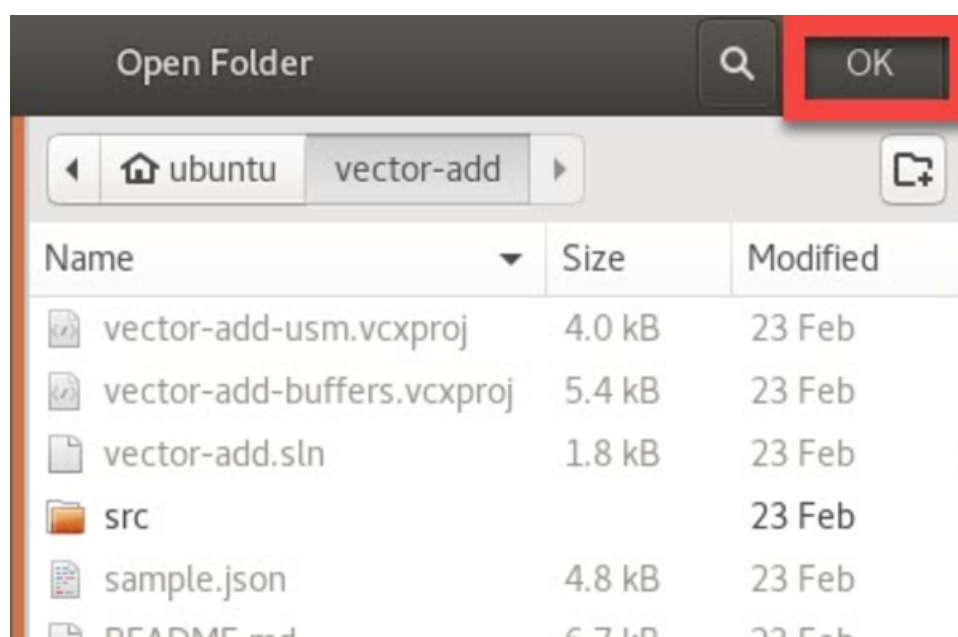
5. Follow the prompts to add targets from your make/cmake oneAPI project.
6. Run the target by selecting **Terminal > Run task....**
7. Select the task to run.



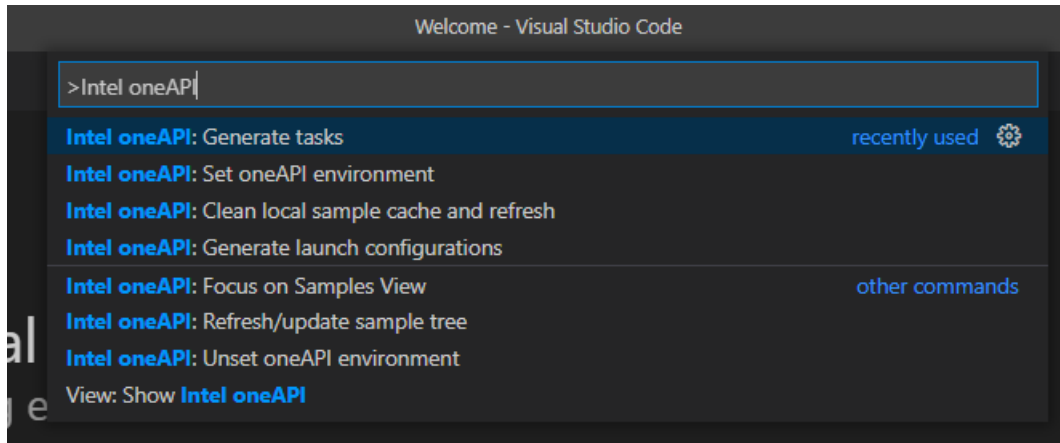
Prepare Launch Configuration

This extension enables the ability to prepare launch configurations for running and debugging projects created using Intel oneAPI toolkits:

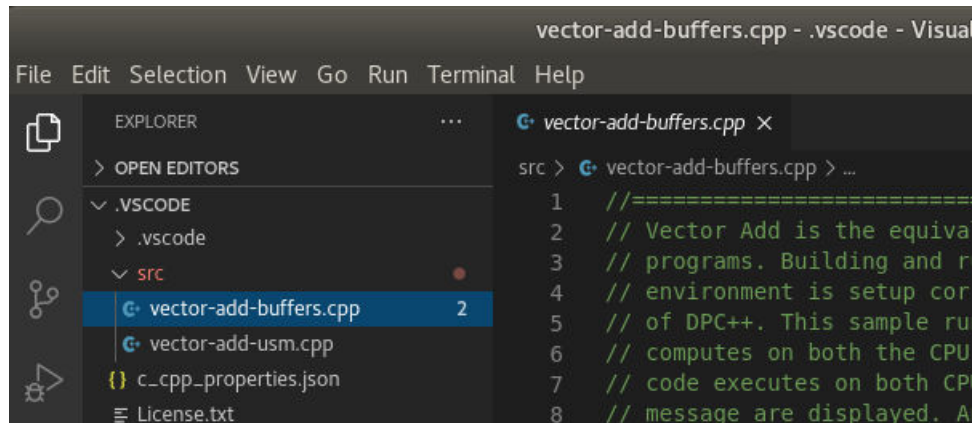
1. Using the VS Code explorer, click **File>Open Folder**.
2. Navigate to the folder where your project is located and click **OK**.



3. Press **Ctrl+Shift+P** (or View -> Command Palette...) to open the Command Palette.
4. Type Intel oneAPI and select Intel oneAPI: Generate launch configurations.



5. Follow the prompts to add launch configurations.
6. Using the VS Code Explorer, open the C++ file for your project.



7. The configuration is now available to debug and run using the gdb-oneapi debugger. To debug and run, click on the Run icon



or press **Ctrl+Shift+D**.

Building a Single cpp File

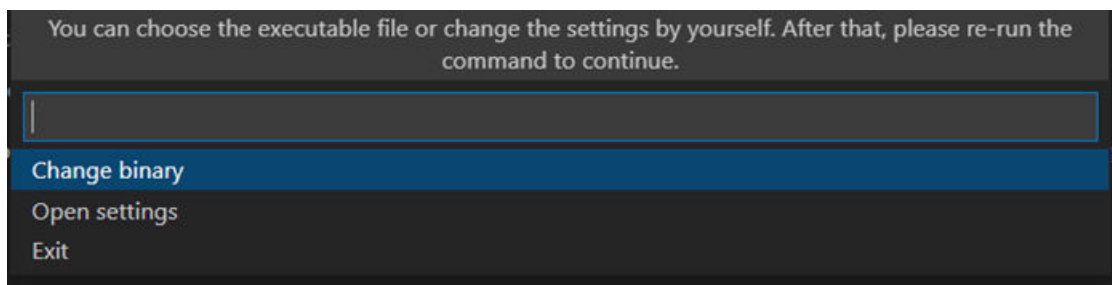
1. Open the cpp file you want to build.
2. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
3. Type Intel oneAPI and select Intel oneAPI: Quick build current file with ICPX.
4. If you want to build a file with SYCL enabled, choose the option Intel oneAPI: Quick build current file with ICPX and SYCL enabled.

Using Intel Analysis Tools

You need to have at least one of the above Intel analysis tools installed for this extension to work and be useful.

1. Open a Visual Studio Code project.
2. Build your project to create the executable you plan to analyze.
3. Press Ctrl+Shift+P (or **View -> Command Palette...**) to open the Command Palette.

4. Type Intel oneAPI and select Intel oneAPI:Launch Advisor or Intel oneAPI: Launch VTune Profiler. Select one of the options displayed:



- **Change binary:** show the list of all executable files in your working directory. Select an executable and select **Save** to save this path to in your workspace or select **Once** to make the change for this session only.

To see your changes, you can open **File-> Preferences -> Settings**, select the extensions tab and click on Analysis Configurator.

- **Open settings:** change the default path to the binary.
 - **Exit:** take no action and exit the command palette.
5. Select the executable you want to analyze. This needs to be done once for a workspace unless you want to analyze a different executable.
 6. Select the installation path of Intel Advisor or Intel VTune Profiler. This needs to be done once for a workspace.
 7. Enter the name of the project folder for the tool, or press enter to accept the default. This needs to be done once for a workspace.
 8. The extension will open the analyzer tool and pass the appropriate project parameters to the tool.

How to Use IntelliSense for Code Developed with Intel oneAPI Toolkits

1. Make sure that the [C/C++ extension](#) is already installed.
2. Press **Ctrl+Shift+P** and type C/C++ Edit Configurations (JSON). Press Enter. As a result, a `c_cpp_properties.json` file will be created in the `.vscode` folder.
3. Edit the file so that it looks like in the example:

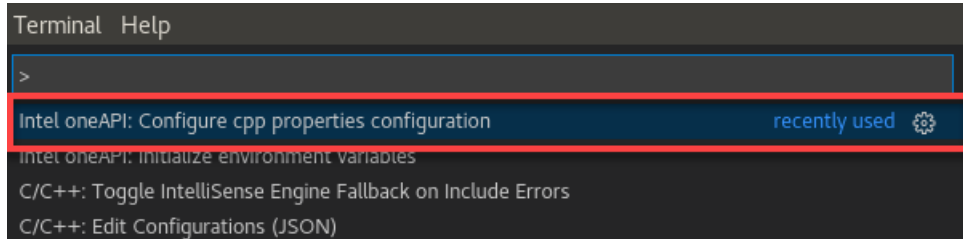
```
{
  "configurations": [
    {
      "name": "Linux",
      "includePath": [
        "${workspaceFolder}/**",
        "/opt/intel/oneapi/**"
      ],
      "defines": [],
      "compilerPath": "/opt/intel/oneapi/compiler/latest/linux/bin/dpcpp",
      "cStandard": "c17",
      "cppStandard": "c++17",
      "intelliSenseMode": "linux-clang-x64"
    }
  ],
  "version": 4
}
```

NOTE If necessary, replace the path to the oneAPI components with the one that is relevant for your installation folder.

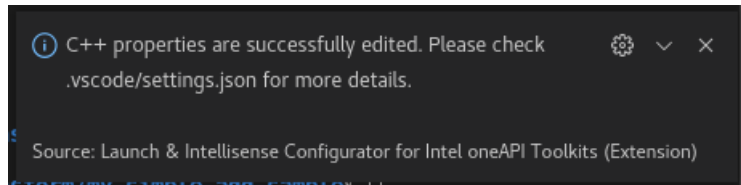
Configure C++ Properties

This extension provides the ability to configure the cpp properties includePath, defines, and compilerPath.

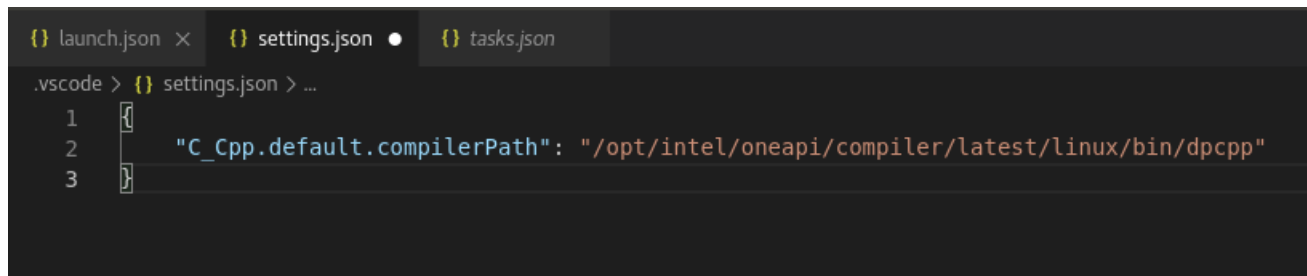
1. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
2. Type **Intel oneAPI: configure cpp properties configuration** and select it from the palette.



3. Select **cpp standard**.
4. Select **c standard**.
5. A message will appear in the lower right corner to confirm the properties have been configured.



1. To view or change the properties, open **settings.json** from the VS Code Explorer.
2. To make changes to the configuration, edit the default path in **settings.json**.



Display Tool Tips

To learn more about the tool tips function, see [Working with Code](#).

GDB GPU Support for Intel® oneAPI Toolkits

Using Visual Studio Code* with Intel® oneAPI Toolkits

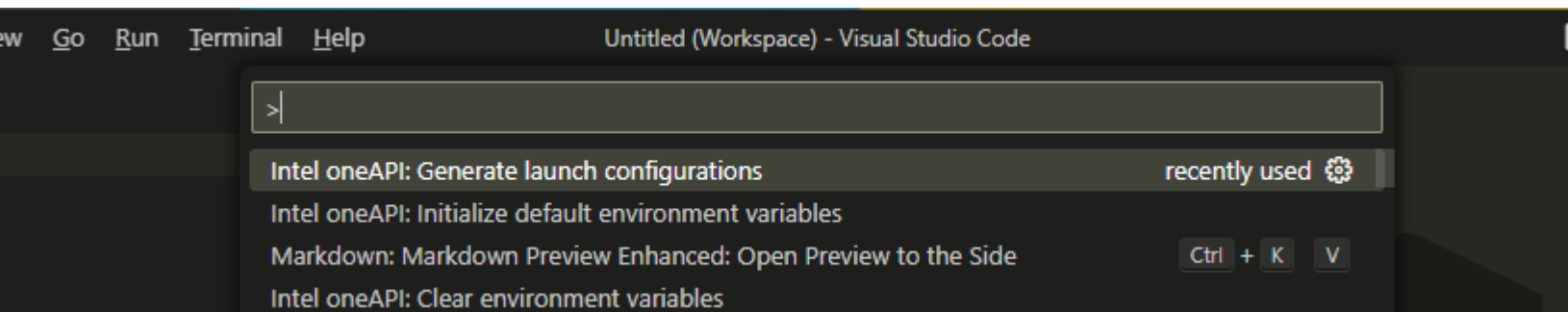
Prepare Launch Configuration

This extension for Visual Studio Code (VS Code) enables additional features of GPU debugging with the GDB for Intel® oneAPI toolkits. Start using this VS Code extension with guide [Get Started with Intel® Distribution for GDB* on Linux* OS Host](#).

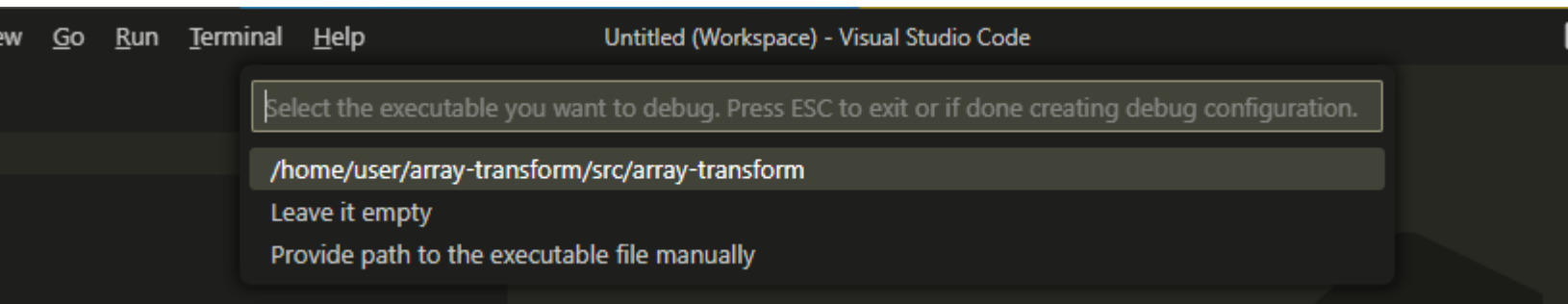
Note that this feature is only available for the Linux* target platform.

1. Open your project in VS Code.
2. Build your project with the `-g` and `-O0` options to prepare for debugging.
3. Press **Ctrl+Shift+P** (or **View -> Command Palette...**) to open the Command Palette.
4. Type **Intel oneAPI** to view options of the installed extensions.

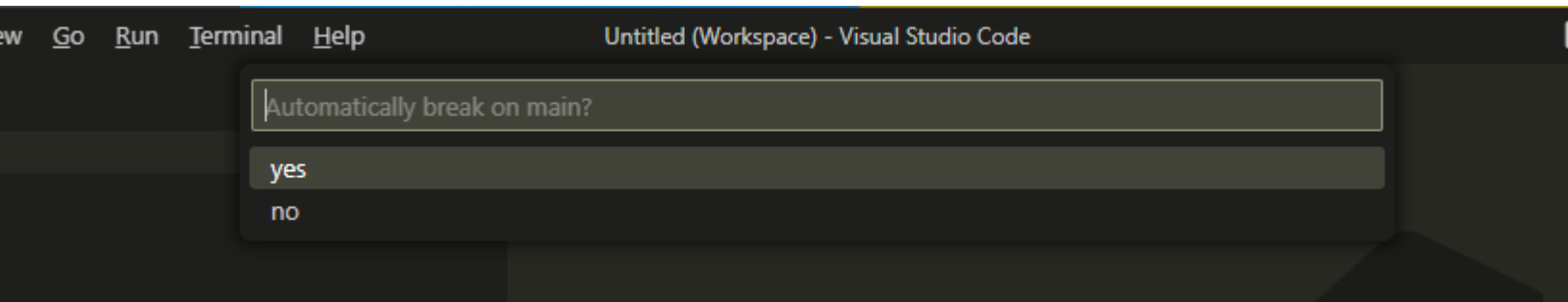
5. Click on **Intel oneAPI: Generate launch configurations**.



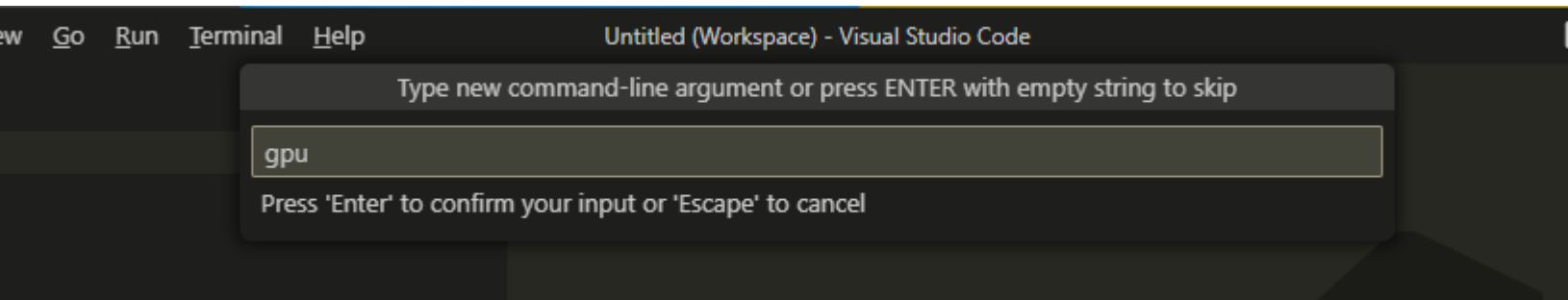
6. Follow the prompts to add launch configurations. First, select the executable:



7. Select **yes**, to automatically break on main.



8. Answer the prompt to set arguments to be passed to the application to debug. In the example below, an argument named `gpu` will be passed on the command line when the application being debugged is launched.



9. After completing all of the prompts, you will see the new argument in the `.vscode/launch.json` file:

launch.json - Untitled (Workspace) - Visual Studio Code

Selection View Go Run ...

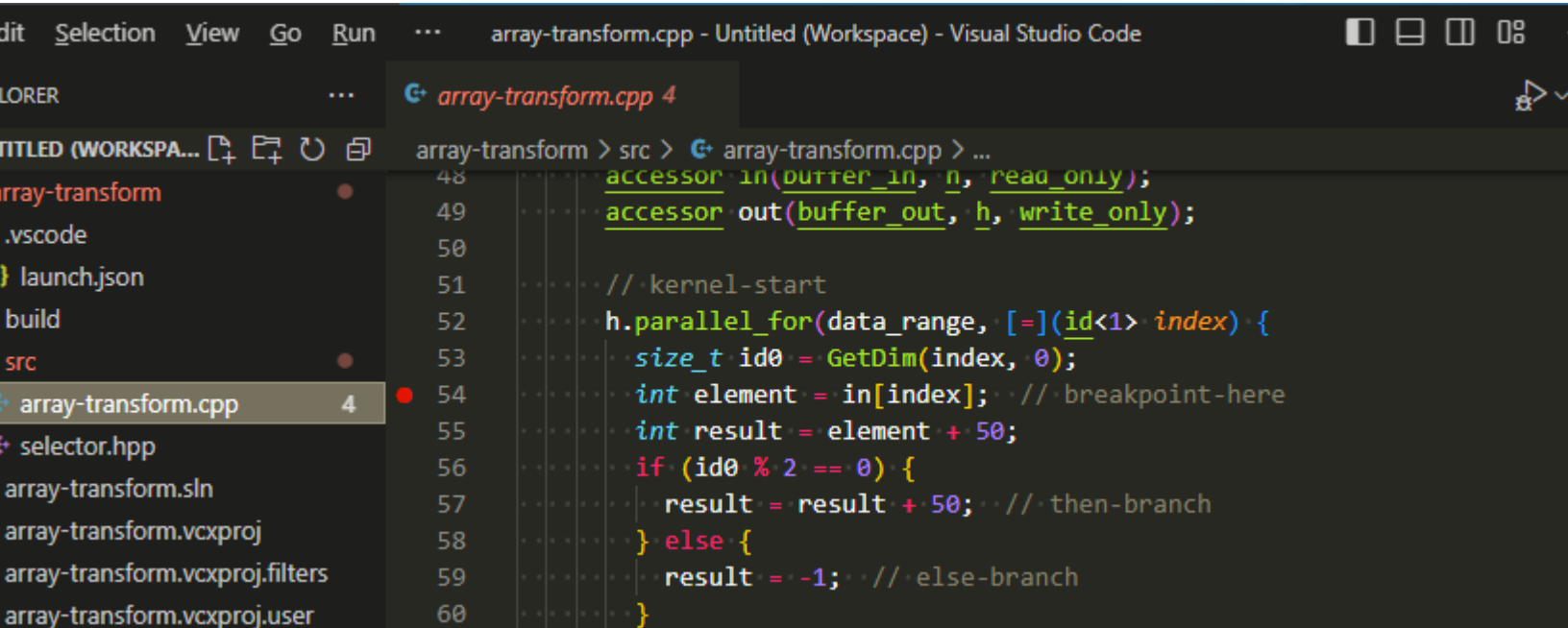
launch.json 1

array-transform > .vscode > {} launch.json > JSON Language Features > [] configurations

```
{
  "configurations": [
    {
      "comments": [
        "Full launch.json configuration details can be found here:",
        "https://code.visualstudio.com/docs/cpp/launch-json-reference"
      ],
      "name": "(gdb-oneapi) array-transform Launch",
      "type": "cppdbg",
      "request": "launch",
      "preLaunchTask": "",
      "postDebugTask": "",
      "program": "/home/user/array-transform/src/array-transform",
      "args": [
        "gpu"
      ],
      "stopAtEntry": true,
      "cwd": "${workspaceFolder}",
      "environment": [
        {
          "name": "ZET_ENABLE_PROGRAM_DEBUGGING",
          "value": "1"
        },
        {
          "name": "IGC_EnableGTLocationDebugging",
          "value": "1"
        }
      ],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "gdb-oneapi",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        },
        {
          "description": "Disable target async",
          "text": "set target-async off",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

Add Configuration...

10. Using the VS Code Explorer, open the source code file or files that contain the section of code you plan to debug. In the example below, the `array-transform.cpp` file has been opened and a breakpoint has been set at line #54.



11. Start debugging by clicking the Run and Debug button on the left. Then click the Debug button at the top. Note the pulldown list of launch configurations next to the Debug button. The pulldown list corresponds to the "name" fields in the `launch.json` file located in the `.vscode` folder of your project folder.

```

array-transform > src > array-transform.cpp >
48     ... accessor::in(buffer_in, n,
49     ... accessor::out(buffer_out, n,
50
51     ... // kernel-start
52     ... h.parallel_for(data_range,
53     ... size_t id0 = GetDim(ind
54     ... int element = in[index]
55     ... int result = element +
56     ... if (id0 % 2 == 0) {
57     ...     result = result + 50;
58     ... } else {
59     ...     result = -1; // else
60     ... }
61     ... out[index] = result;
62     ... });
63     ... // kernel-end
64     ... });
65
66     ... q.wait_and_throw();
67     ... } catch (sycl::exception const
68     ... cout << "fail; synchronous
69     ... return -1;
70     ... }
71
72     ... // Verify the output
73     ... for (int i = 0; i < length; i
74     ... int result = (i % 2 == 0) ?
75     ... if (output[i] != result) {
76     ...     cout << "fail; element "
77     ...     return -1;
78     ... }
79     ... }
80
81     ... cout << "success; result is c

```

Note that you can modify the configuration manually. For example, you may need to change:

- "args" - Example ["arg1", "arg2"]. If you are escaping characters, you will need to double escape them. For example, [{"\\\\"arg1\\\\"": true}] will send {"arg1": true} to your application.
- "stopAtEntry" - If set to true, the debugger should stop at the entry-point of the target (ignored on attach). Default value is false.
- "cwd" - Sets the working directory of the application launched by the debugger.

- "environment" - Environment variables to add to the environment for the program. Example:
[{ "name": "config", "value": "Debug" }], not [{ "config": "Debug" }].

For information about launch.json features, see [Configuring C/C++ debugging](#)

SIMD View

This extension provides a view in the debug view that displays the SIMD lane state of a Intel GPU thread. The view will automatically populate when hitting a GPU thread breakpoint.

ThreadID

The thread ID that GDB sees in the GPU inferior process. For example, 2.1 in the GDB console would be 1.

To rename the thread while debugging:

1. Press **CTRL+Shift+P** to open the Command Palette.
2. Type `Open Debug Console` and press enter.
3. Select the desired thread with: `-exec thread THREADID`
4. Rename the thread using; `-exec thread name YOURNAME`
5. Press Continue (F5) on your debug window to see renewed threads.

Name

The string name of the GPU thread.

SIMD Lanes

The status of the SIMD lanes in the thread. Dark blue represents active lanes stopped at a breakpoint, light blue indicates active lanes that do not meet breakpoint conditions and grey indicates inactive lanes. Typically the status will be SIMD8 (8 lanes). Ideally, all lanes should be active.

SIMD LANES		
ThreadID	Name	SIMD Lanes
2	Thread #6	01010101
3	Thread #7	01010101
4	Thread #8	01010101
5	Thread #9	01010101
6	Thread #10	01010101
7	Thread #11	01010101
8	Thread #12	01010101
1	Thread #5	01010101

SIMD Lane Specific Breakthroughs

Note that SIMD lane specific breakpoints are saved between sessions, but will be applied only after hitting a regular breakpoint inside the kernel.

There are several ways to set a SIMD lane specific breakpoint:

- Add a breakpoint by right-clicking using the keyword `-break-insert` and specifying the thread number and SIMD lane: `-break-insert -p THREADID -l SIMD LANE`
- Use the Intel oneAPI: Add SIMD conditional breakpoint function from the drop-down menu and specify the THREADID and SIMD LANE in format: `THREADID:SIMD LANE`

Symbolic Indication of SIMD Lanes

In the settings, you can specify an additional designation for active and inactive lanes using any text character. This may be useful for clearer recognition of lane status.

Choose SIMD Lane

You can choose a new SIMD lane by clicking on it. Choosing a new SIMD lane will show updated information in the **SELECTED LANE** tab, and extended thread information can be found using the debug console (command `-exec -thread-info`).

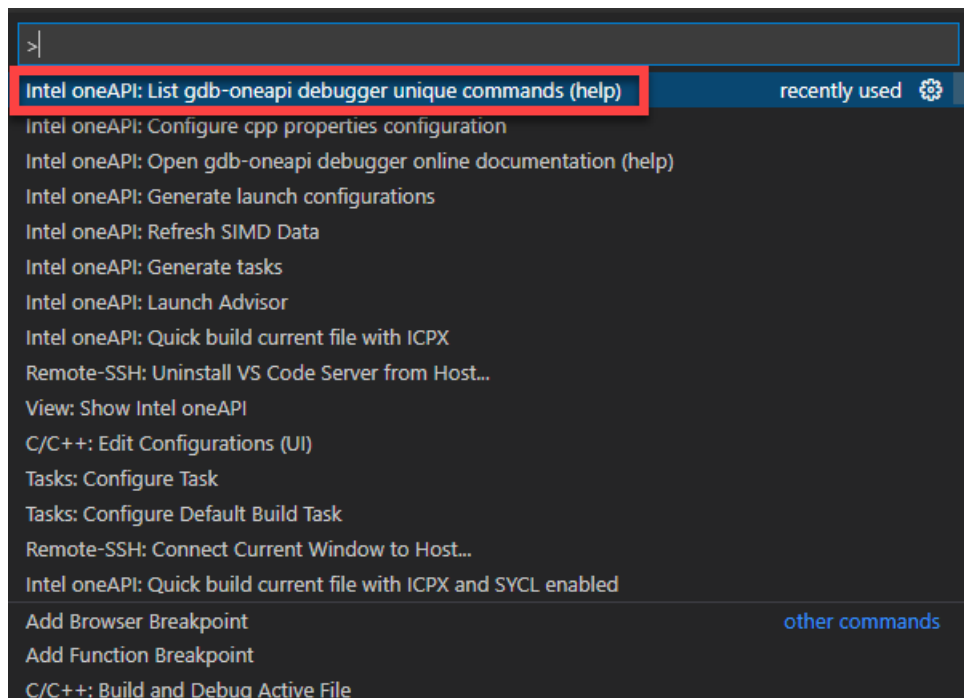
Hardware Info

You can see your device's info in a separate tab while debugging.

Differences between GDB and GDB-oneapi

To display the differences between these two distributions of GDB:

1. Press `Ctrl+Shift+P` (or **View -> Command Palette...**) to open the Command Palette.
2. Type **help** to see help commands.
3. Select **Intel oneAPI: List gdb-oneapi debugger unique commands (help)**.



4. A new window will open showing the differences and links to documentation.
5. If you want quick access to GDB-oneAPI Online Documentation, select **Intel oneAPI: Open gdb-oneapi debugger online documentation (help)**.

DevCloud Connector for Intel® oneAPI Toolkits*

Using Visual Studio Code* with Intel® oneAPI Toolkits

DevCloud allows you to develop, test, and run your workloads for free on a cluster of the latest Intel® hardware and software. With integrated Intel® optimized frameworks, tools, and libraries, you'll have everything you need for your projects.

To connect to DevCloud using Visual Studio Code:

1. [Register](#) for an Intel® DevCloud account.

2. Follow the set up steps to [Connect to DevCloud with Visual Studio Code](#).

Search for a Specific Sample

Using Visual Studio Code* with Intel® oneAPI Toolkits

Use the [oneAPI Samples Browser](#) to locate sample projects that are ready to build and run. Most samples require the DPC++ compiler (dpcpp) to build and run. The Intel compilers are available in the [Intel® oneAPI Base Toolkit](#).

Set the oneAPI Environment Manually

Using Visual Studio Code* with Intel® oneAPI Toolkits

The oneAPI Environment can be automatically run using the VS Code **Environment and Launch Configurator for Intel oneAPI Toolkits** extension. For more information on extensions that work with Intel® oneAPI Toolkits, see [Using the Environment Set up and Launch Configuration Extension](#).

As an alternative, you can manually run setvars:

1. Navigate to the install directory and source `setvars.sh`.

```
source <install_dir>/setvars.sh
```

The command above assumes you installed to the default folder. If you customized the installation folder, `setvars.sh` (or `oneapi-vars.sh`) is in your custom folder. See here for more information on [setvars.sh](#).

2. Launch VS Code:

```
code
```

To run `setvars` for Windows*:

1. From a terminal window, run `setvars.bat`.

```
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
```

The command above assumes you installed to the default folder. If you customized the installation folder, `setvars.bat` is in your custom folder.

2. From the same terminal window, launch VS Code:

```
"C:\Users\<username>\AppData\Local\Programs\Microsoft VS Code\Code.exe"
```

NOTE For Windows PowerShell* users, execute this command:

Working with Code

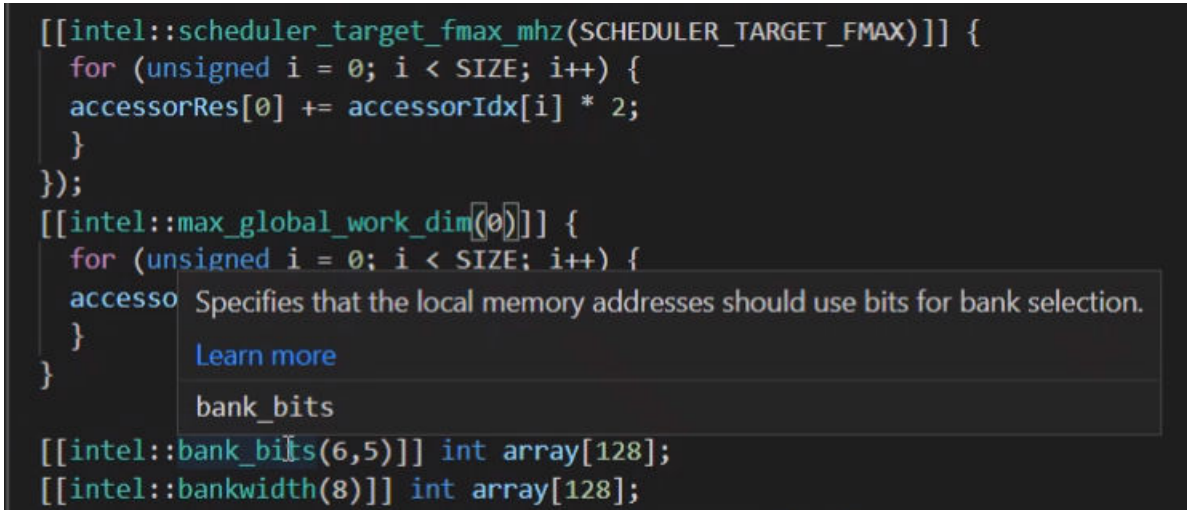
Using Visual Studio Code* with Intel® oneAPI Toolkits

With the [Analysis Configurator for Intel® oneAPI Toolkits*](#) extension, VS Code will display tool tips when working with FPGA code. These tips include:

- a description of the attribute
- a link to more information about that attribute
- suggestions for other attributes

To view tool tips and find out more about the attribute:

1. Click on the line of code where you want to see a tool tip and keep your pointer over the code. A tool tip will display. In the example below, a description for the `bank_bits` attribute is appearing in the pop up window.



```
[[intel::scheduler_target_fmax_mhz(SCHEDULER_TARGET_FMAX)]] {
    for (unsigned i = 0; i < SIZE; i++) {
        accessorRes[0] += accessorIdx[i] * 2;
    }
});
[[intel::max_global_work_dim(0)]] {
    for (unsigned i = 0; i < SIZE; i++) {
        accessorRes[0] += accessorIdx[i] * 2;
    }
}
[[intel::bank_bits(6,5)]] int array[128];
[[intel::bankwidth(8)]] int array[128];
```

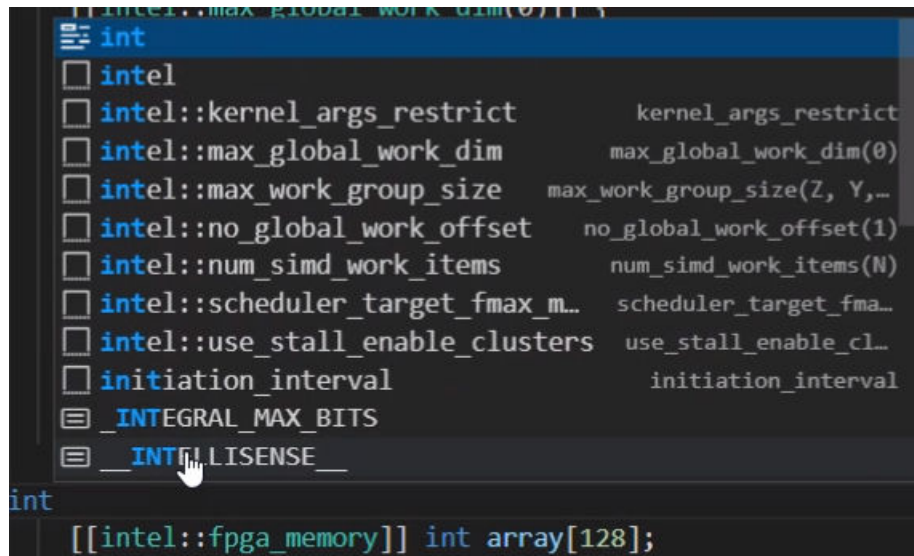
Specifies that the local memory addresses should use bits for bank selection.

[Learn more](#)

bank_bits

2. For more details about this attribute, click the **Learn more** link in the VS Code pop up window. To view suggestions for attributes:

1. In your code, type `intel::`. A list of available attributes will show as suggestions:



```
int
[[intel::max_global_work_dim(0)]] {
    for (unsigned i = 0; i < SIZE; i++) {
        accessorRes[0] += accessorIdx[i] * 2;
    }
}
[[intel::bank_bits(6,5)]] int array[128];
[[intel::bankwidth(8)]] int array[128];
```

- ☐ intel
- ☐ intel::kernel_args_restrict kernel_args_restrict
- ☐ intel::max_global_work_dim max_global_work_dim(0)
- ☐ intel::max_work_group_size max_work_group_size(Z, Y,...
- ☐ intel::no_global_work_offset no_global_work_offset(1)
- ☐ intel::num_simd_work_items num_simd_work_items(N)
- ☐ intel::scheduler_target_fmax_m... scheduler_target_fma...
- ☐ intel::use_stall_enable_clusters use_stall_enable_cl...
- ☐ initiation_interval initiation_interval
- ☒ _INTEGRAL_MAX_BITS
- ☒ _INTEL_LIENSENSE_

2. Click on an attribute to add it to your code.

Notices and Disclaimers

Using Visual Studio Code* with Intel® oneAPI Toolkits

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Product and Performance Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.