13th International Workshop on OpenCL and SYCL

IWOCL 2025

Debugging SYCL on Intel GPUs with Visual Studio and VS Code

Rakesh Ganesh, Intel

Rakesh Ganesh, Andria Pazarloglou, Sergey Bobko - Intel

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

Introduction

Debugging applications written in SYCL, OpenMP, and Fortran—especially those targeting heterogeneous architectures like CPUs and GPUs—is inherently complex. Intel actively contributes to GDB to enhance support for modern high-performance computing requirements, and the Intel® Distribution for GDB, part of the Intel® oneAPI Base Toolkit, builds on these advancements. It provides an efficient debugging solution for parallel and multithreaded applications developed in SYCL, OpenMP, and Fortran, and offers powerful capabilities to analyze GPU states, memory, and thread interactions.

This technical paper highlights the enhanced features of Intel® Distribution for GDB, including its integration into Visual Studio and VS Code. By leveraging a user-friendly interface, developers can debug applications more efficiently without relying on intrusive printf statements or direct interaction with the GDB CLI. This streamlined experience enables developers to focus on solving critical issues effectively on Intel CPUs and GPUs across both Windows and Linux environments.

Additionally, we will cover the detailed setup process for Intel® Distribution for GDB on both Windows and Linux machines, ensuring developers can configure their debugging environment seamlessly.

Windows – Visual Studio:

Debugging Environment and Setup

The full guide to getting started with Intel® Distribution for GDB on Windows can be found on the Intel's webpage online.

Prerequisites

Before we can debug our SYCL application on VS IDE, we must ensure the following prerequisites are met:

- Resizable BAR or Smart Access Memory must be enabled for debugging applications using Intel® Arc™ Graphic cards. Please follow the instructions to enable the Resizable BAR. For local GPU debugging, the Resizable BAR needs to be enabled on the host machine and for remote GPU debugging, this needs to be enabled on the target machine.
- For local GPU debugging, a Windows system with a combination of either an integrated and a discrete GPU, or multiple discrete GPUs is required. The list of supported GPU devices can be found here.
- For remote GPU debugging, two Windows systems are required: a host and a target. Microsoft Visual Studio* and Intel® oneAPI Base Toolkit must be installed on the host system. The application is deployed and run on the target system.

Software Installation

Local GPU debugging

- Install Microsoft Visual Studio* 2019 or 2022 on the host machine.
- Install the Intel® oneAPI Base Toolkit for Windows* OS on the host machine.
- Install the latest GPU drivers on the host machine.
- Run the target installer of Intel Distribution for GDB (gen_debugger_target.msi) located in the following path: %ProgramFiles(x86)%\Intel\oneAPI\debugger\latest\opt\debugger\target.

  While installing, select the checkbox under "Additional Installer Task" to set the TdrDelay to the default value of 300 seconds

Remote GPU debugging

- Install the latest GPU drivers on the target system.
- Install Microsoft Visual Studio remote debugger on the target.
- Install run-time dependencies by selecting Intel® oneAPI DPC++/C++ Compiler Runtime for Windows* from the list of runtime dependencies.
- Copy and run the target installer of Intel Distribution for GDB (gen_debugger_target.msi) from the host system to the target. The installer can be found on the host machine in the following path: %ProgramFiles(x86)%\Intel\oneAPI\debugger\latest\opt\debugger\target.

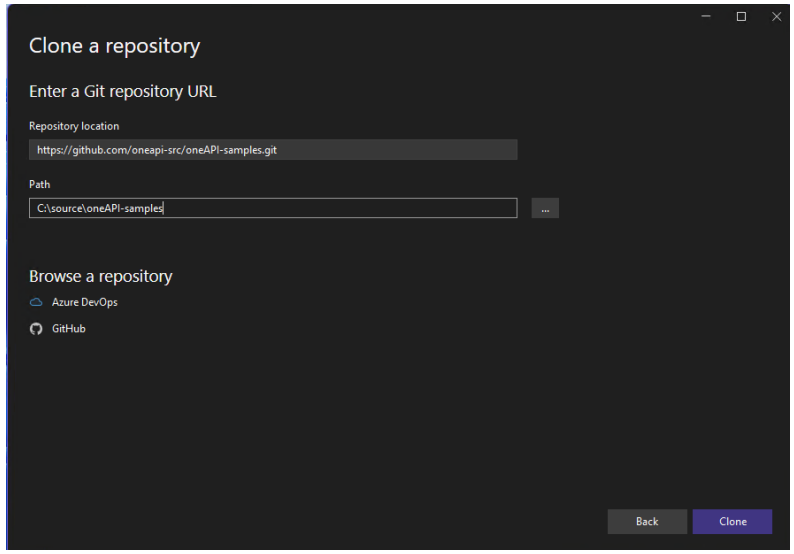
  While installing, select the checkbox under "Additional Installer Task" to set the TdrDelay to the default value of 300 seconds
- To enable Fortran debugging, install, in addition, the Fortran Compiler from the Intel® HPC Toolkit for Windows* OS.

Verify the setup and build the application

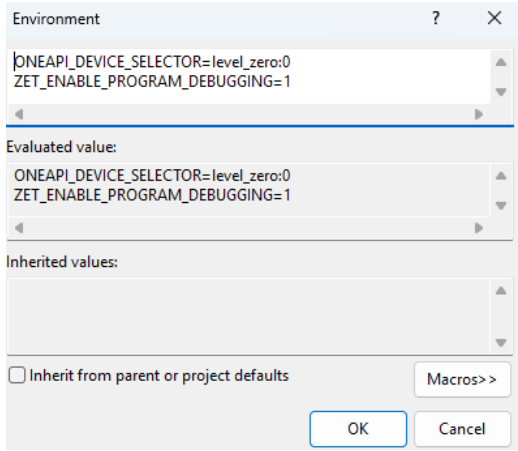
We now need to create a SYCL application that we want to debug. To help with the setup, Intel provides a set of sample applications that can be downloaded and used. Follow the steps to set up a sample application:

- Open a new VS instance and clone the oneAPI sample repository from the below path:

  https://github.com/oneapi-src/oneAPI-samples.git



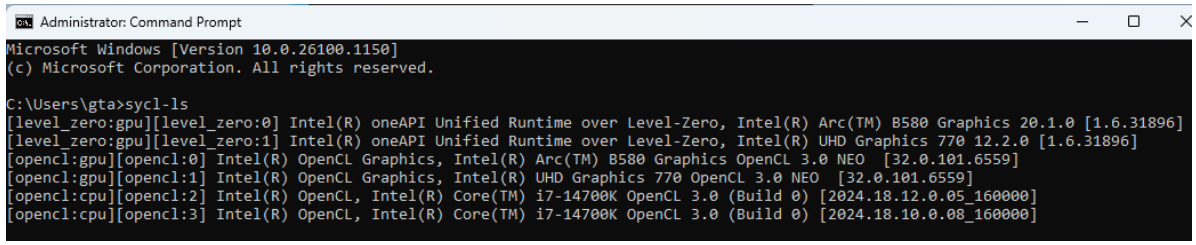
- Navigate to the path containing oneAPI samples and then to **oneAPI-samples\Tools\ApplicationDebugger\array-transform** and open the **array-transform.sln.**
- Open the project properties of the array-transform project.
- For local debugging, specify the following environment variables:



1> To set the value of **ONEAPI_DEVICE_SELECTOR**, open a new command prompt and run the below prompt:

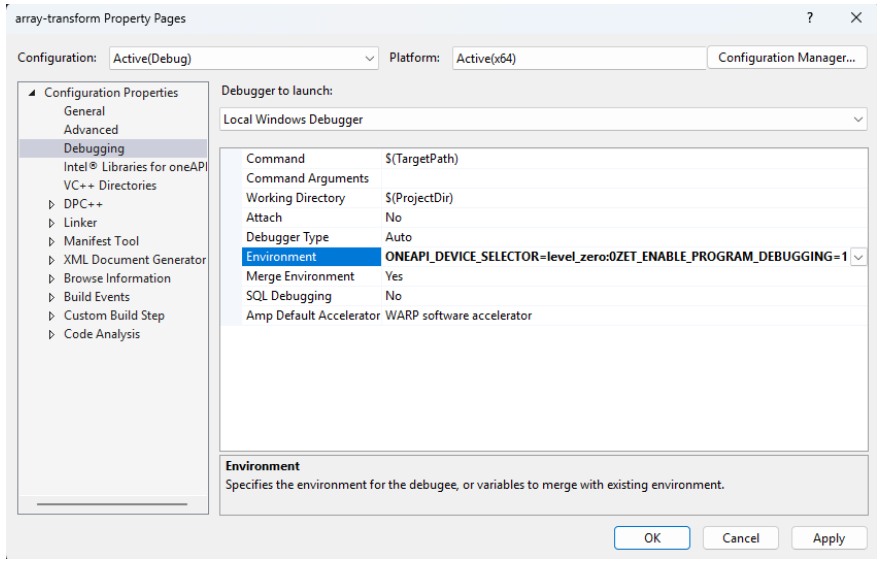
sycl-ls

You should see a similar output indicating CPU and GPU drivers.

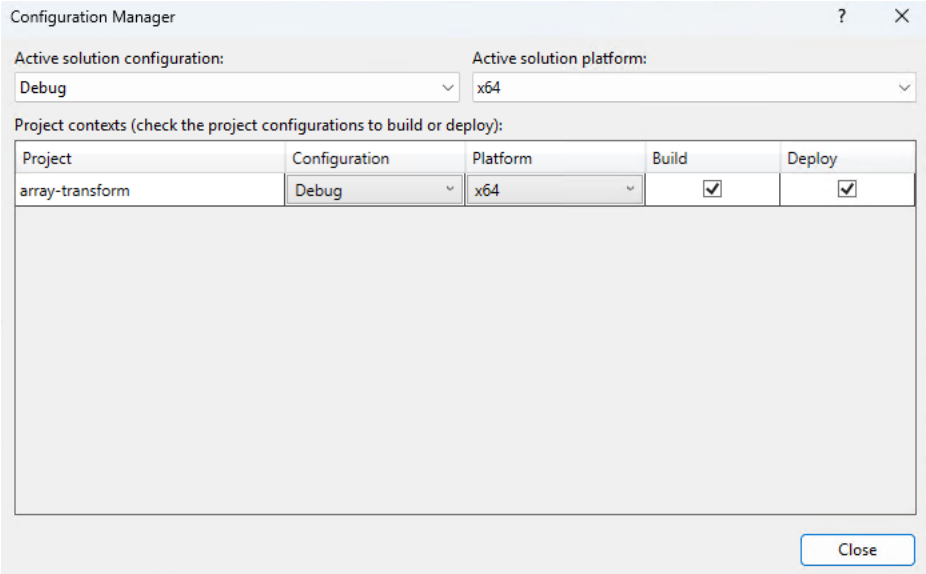


Based on the device you want to debug, specify the value of **ONEAPI_DEVICE_SELECTOR.** In this case we want to debug the Battlemage GPU, and we set the variable value as **level_zero:0**.

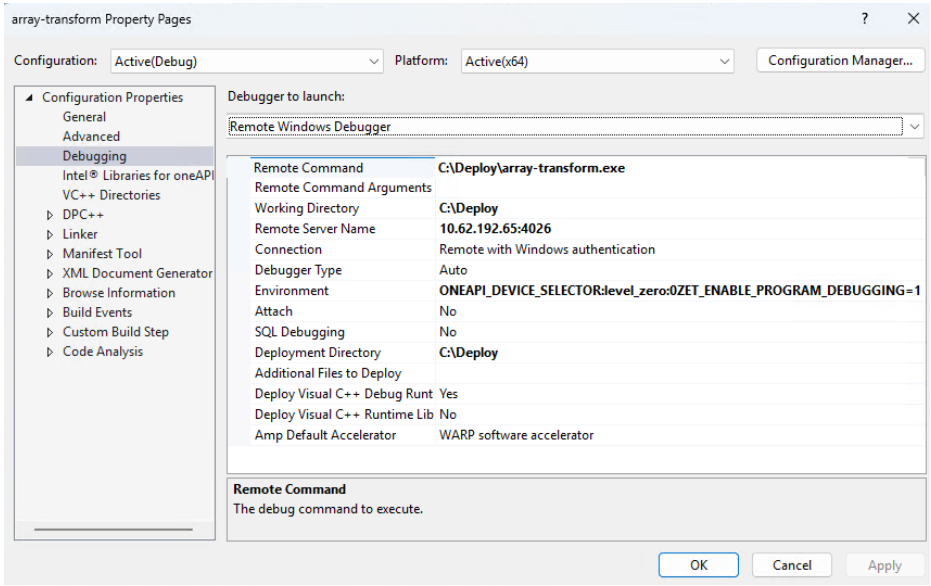
2> Set the environment variable **ZET_ENABLE_PROGRAM_DEBUGGING=1**.



- For remote debugging, follow the steps as below:
  1> Set the remote command value to the application executable you want to debug.
  2> Set the working directory to the location of this application executable.
  3> Specify the remote server's name to the target machine IP address which contains the GPU (or CPU) device you want to debug.
  4> Specify the environment variables as done for local debugging.
  5> Specify the deployment directory where you want your executable to be placed. Usually this is the same value as the working directory.
  6> Open the configuration manager and check the build and deploy checkboxes.

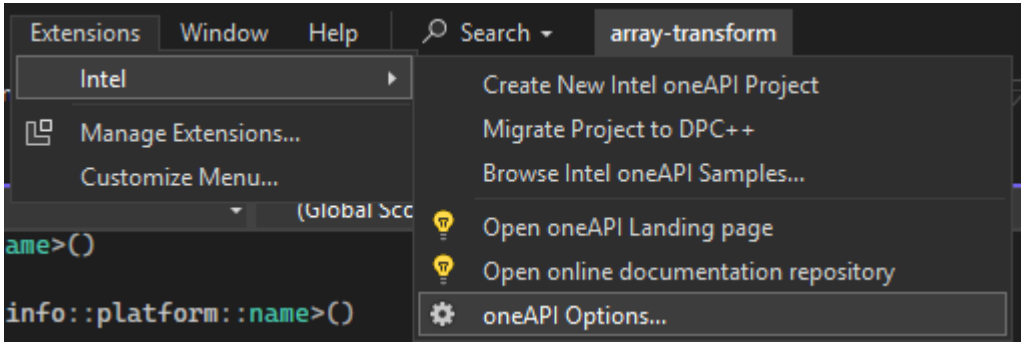


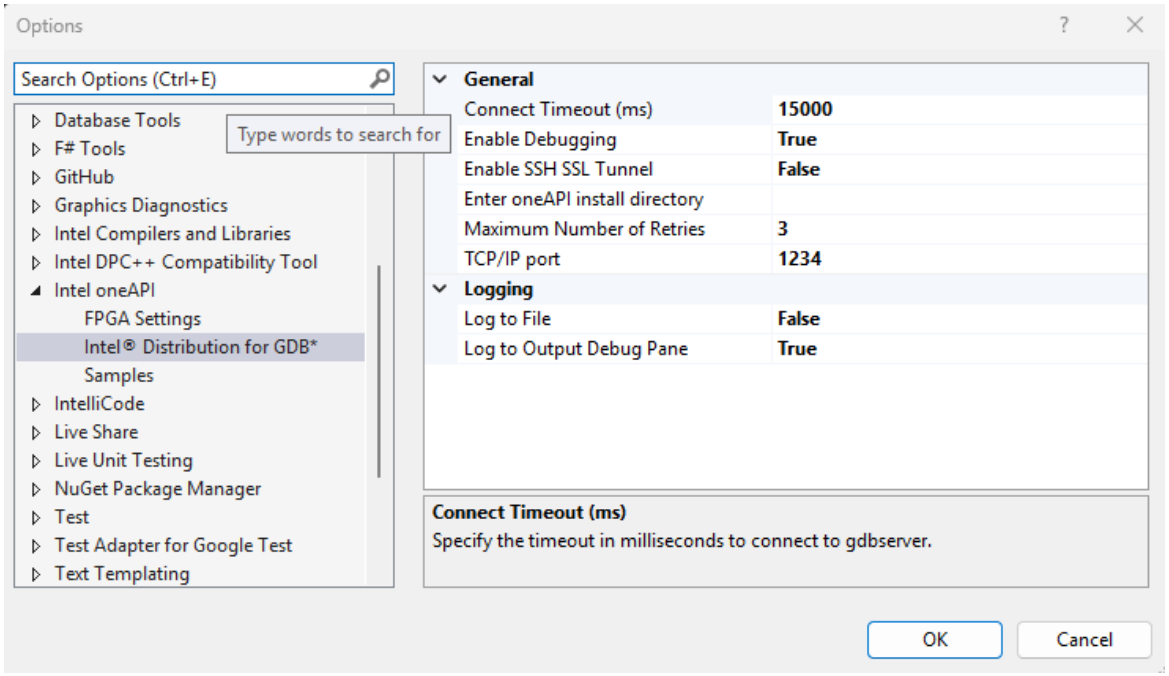
- Save the changes by clicking apply and close the property page.



Note: The general, DPC++ and the Linker settings are usually set in the sample application.

- Navigate to **Extensions > Intel > oneAPI Options**. Go the Intel® Distribution for GDB* and set the **Enable Debugging** to **true**.

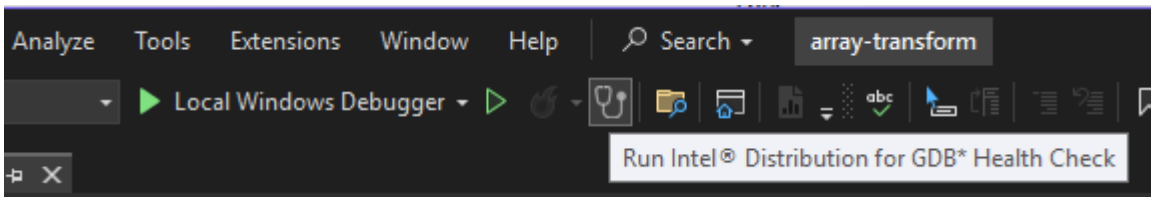




Use the [getting started with Intel® Distribution for GDB]() to explore more options.

Debug Health Check

We can also check that the setup is correct for local or remote debugging by selecting the desired debugging type and running the health checks.



This should provide the output in the output window (under Health Check Results). The failures can also be seen on error list window (under Build + IntelliSense).

```
Output
Show output from: Health Check Results
PASS        Multi-GPU Support
PASS        Level-Zero Version      Current level-zero version: 1.6.31896 1.6.31896
PASS        Debugger Check          Version:
PASS        Compiler Check          Version: (2025.0.4.20241205) (x86)\Intel\oneAPI\compiler\2025.0\bin\compiler (x86)\Intel\oneAPI\compiler\2025.0\bin\compiler\..\icx.cfg
PASS        GDB Env Variables
PASS        gdb-oneapi Check
PASS        gdbserver-ze Check
```

Debugging a GPU application

Now that we have set up our host and target machines (for remote debugging), we can start debugging the sample application.

Place a breakpoint inside the kernel offloaded to the GPU. It is indicated in the sample application code.

```
45
46  q.submit([&](auto &h) {
47    accessor in(buffer_in, h, read_only);
48    accessor out(buffer_out, h, write_only);
49
50    // kernel-start
51    h.parallel_for(data_range, [=](id<1> index) [[sycl::reqd_sub_group_size(32)]] {
52      size_t id0 = GetDim(index, 0);
53      int element = in[index];  // breakpoint-here
54      int result = element + 50;
55      if (id0 % 2 == 0) {
56        result = result + 50;  // then-branch
57      } else {
58        result = -1;  // else-branch
59      }
60      out[index] = result;
61    });
62    // kernel-end
63  });
```

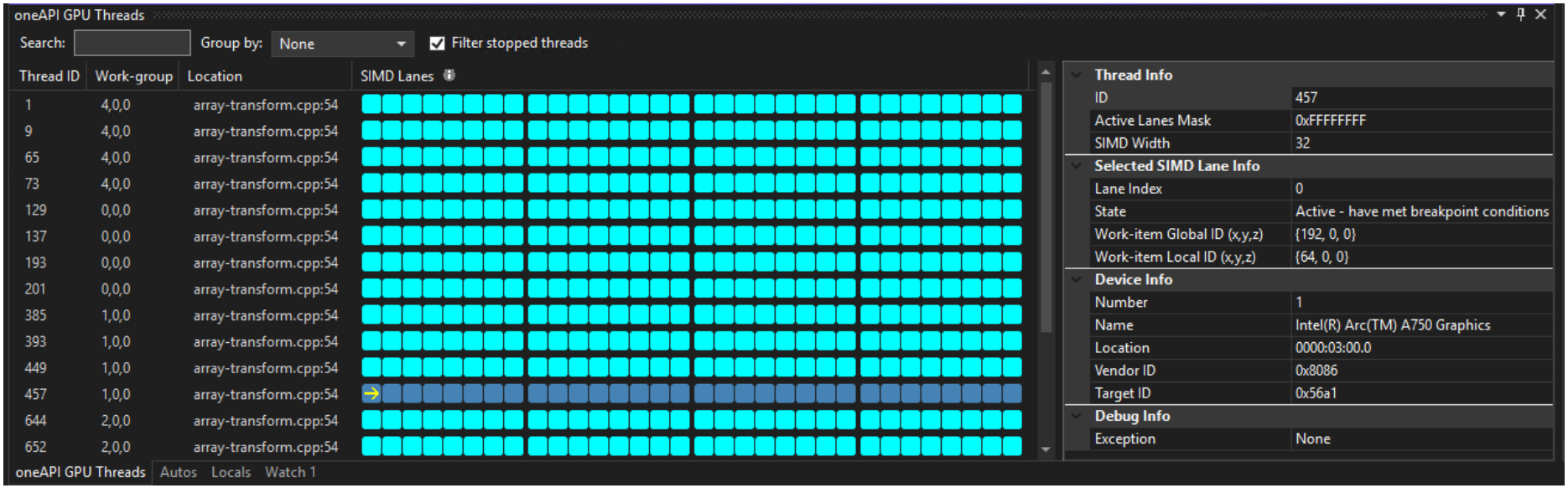
Debug the application by clicking the **Local Windows Debugger** icon. We should be able to hit the kernel breakpoint.

```
46  q.submit([&](auto &h) {
47    accessor in(buffer_in, h, read_only);
48    accessor out(buffer_out, h, write_only);
49
50    // kernel-start
51    h.parallel_for(data_range, [=](id<1> index) [[sycl::reqd_sub_group_size(32)]] {
52      size_t id0 = GetDim(index, 0);
53      int element = in[index];  // breakpoint-here
54      int result = element + 50;
55      if (id0 % 2 == 0) {
56        result = result + 50;  // then-branch
57      } else {
58        result = -1;  // else-branch
59      }
60      out[index] = result;
61    });
62    // kernel-end
63  });
```

Features

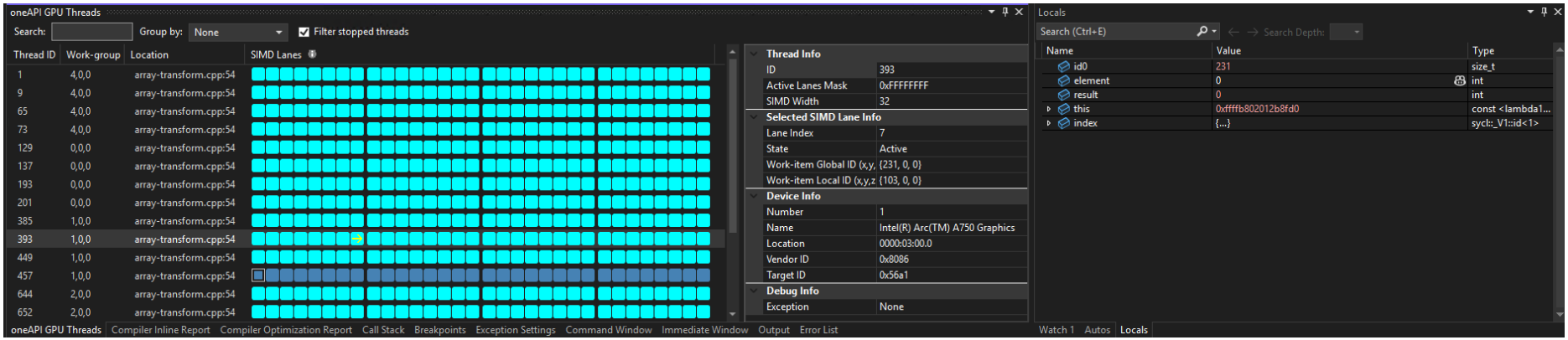
Intel oneAPI GPU Threads window

Once we hit the kernel breakpoint, navigate to **Debug > Windows > Intel oneAPI GPU Threads** and open the oneAPI GPU Threads Window. We should be able to see the active threads and SIMD lanes by default when the **Filter stopped threads** checkbox is selected.

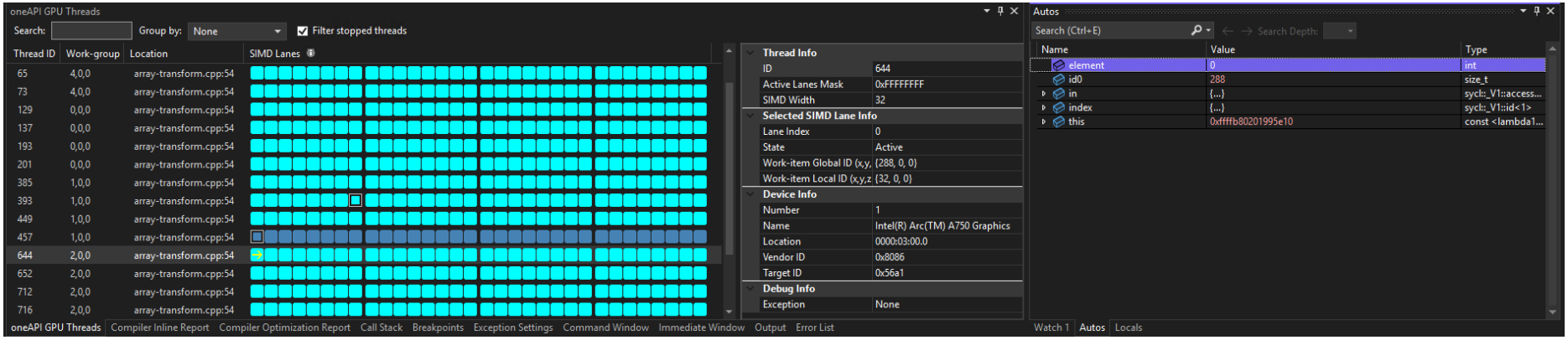


It is possible to view workgroup and location information for each active thread. The right part of the view displays information about the selected thread, SIMD lane and the current device. The Thread Info section contains the ID, Active Lanes Mask and the SIMD Width of the selected thread. The Selected SIMD Lane Info section contains Lane Index, State, Global ID and Local ID of the work-item selected. The Device Info part shows information regarding the current device used for offloading, such as Device Number, Name, Location, Vendor ID and Target ID.

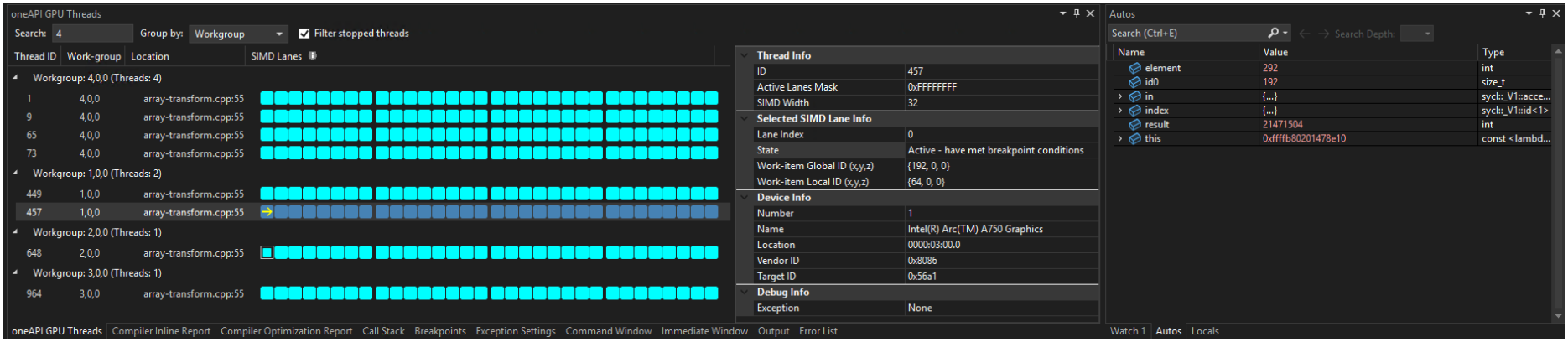
It is possible to switch to another active SIMD lane that does not meet the breakpoint condition by clicking it and see all the variable information for that lane on **Locals/Autos** window. The last selected lane can be identified by a small box around the SIMD lane.



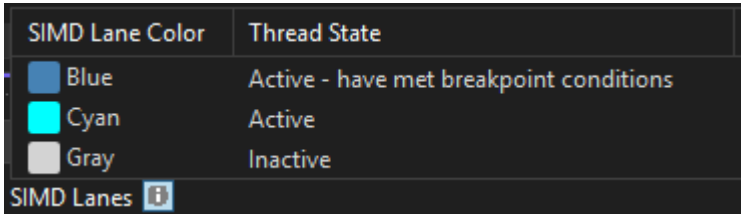
We can also double-click on another active thread to set it as the current thread. The first available SIMD lane will be selected for that thread. We can then inspect the lane information and local variables.



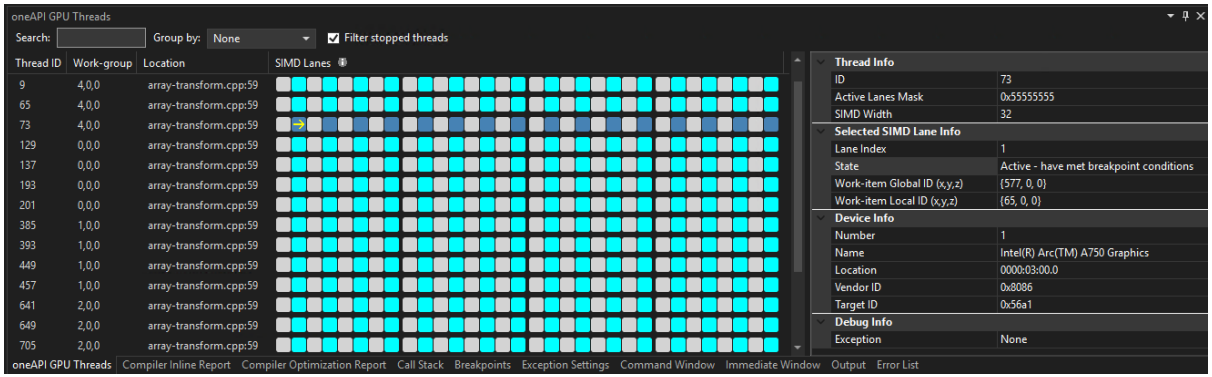
We can filter and group the data inside the oneAPI GPU thread window. To filter the data, enter the text we want to search in the text box next to **Search**. Similarly, we can select the field or the device we want to group the data by, by selecting a value from the drop down next to **Group by**.



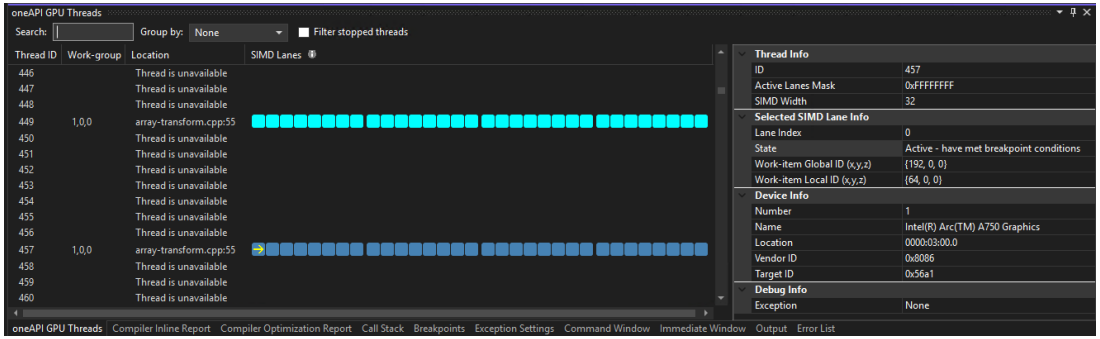
We can view SIMD lane colour scheme by clicking the information button next to **SIMD Lanes** column in the oneAPI GPU thread window. This opens a popup that signifies the meaning of each colour.



Conditions where the SIMD lanes are inactive can be easily identified in the oneAPI GPU Threads window.

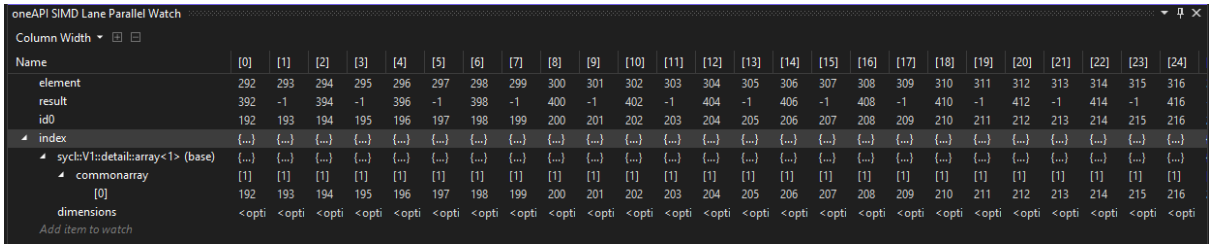


We can also see the unavailable threads when we uncheck the **Filter stopped threads checkbox.**



Intel oneAPI SIMD Lane Parallel Watch window

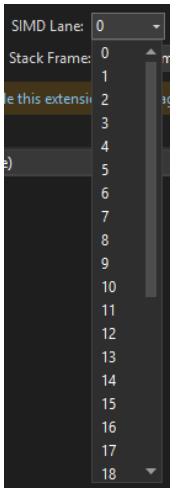
Once we hit the kernel breakpoint, navigate to **Debug > Windows > Intel oneAPI SIMD Lane Parallel Watch** and open the oneAPI SIMD Lane Parallel Watch window. This is an advanced watch window where we can see the value of variables in all the active lanes for a selected thread.



Toolbar Options

SIMD Lane Toolbar

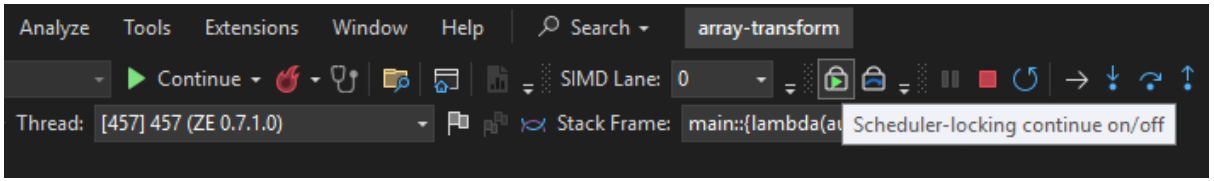
We can change the SIMD lane from the toolbar and inspect the change in local variables. To view SIMD Lane in the toolbar, navigate to **View > Toolbars > SIMD Lanes** and enable it.



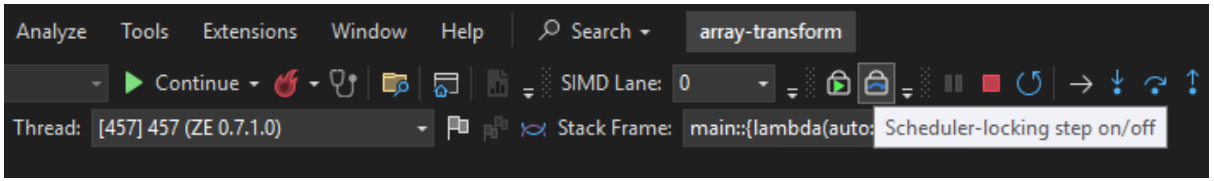
Scheduler-locking Continue and Scheduler-locking Step

When debugging the application, we can use the Scheduler-locking Continue and Scheduler-locking Step UI buttons in the toolbar to control the scheduler locking settings when continuing or stepping through the program.

When **Scheduler-locking Continue** is **on**, the scheduler gets locked for continuing commands during normal execution and record modes. For continuing commands other threads may not pre-empt the current thread. This setting is **off** by default.

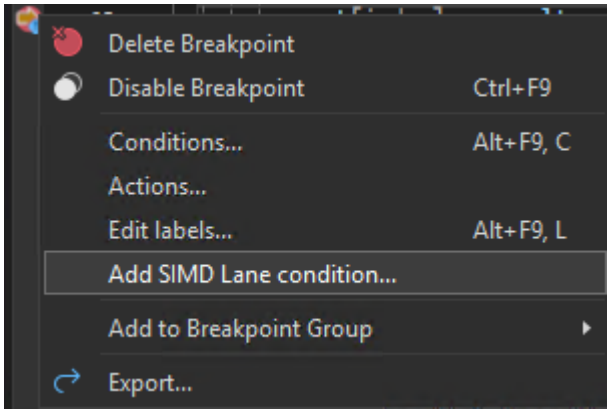


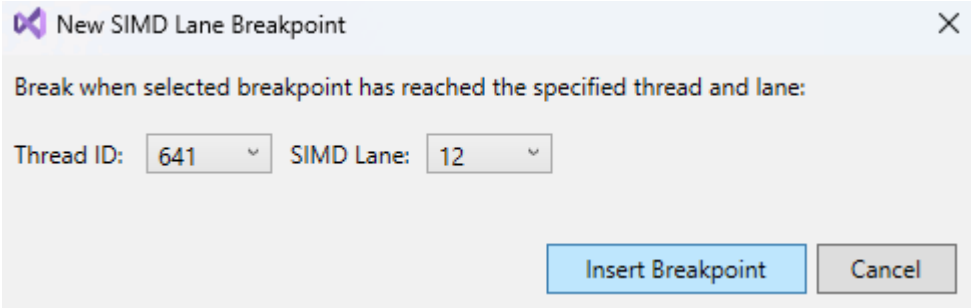
When **Scheduler-locking Step** is **on**, the scheduler gets locked for stepping commands during normal execution and record modes. While stepping, other threads may not pre-empt the current thread, so that the focus of debugging does not change unexpectedly. This setting is **off** by default.



SIMD Lane specific breakpoint

We can add a SIMD Lane specific breakpoint inside a kernel which respects the SIMD Lane conditions. To place SIMD lane specific breakpoint inside a kernel, we place an ordinary breakpoint. Once we hit this breakpoint, we must right click on the breakpoint. This opens a popup where you can select **Add SIMD Lane Condition...**.

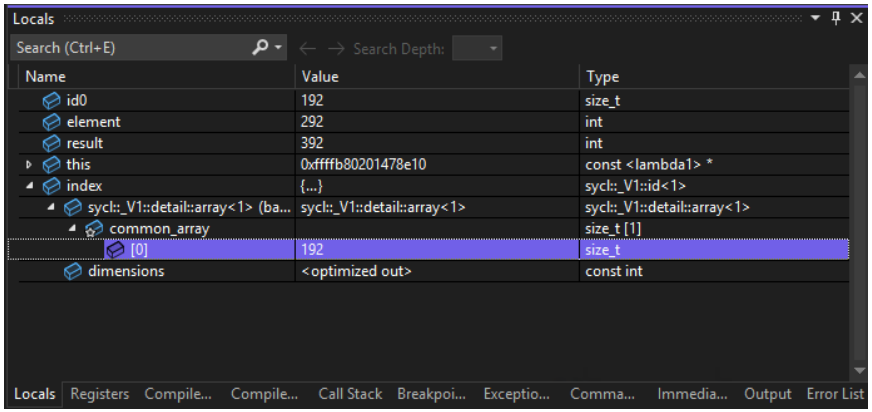




Visual Studio windows for GPU debugging

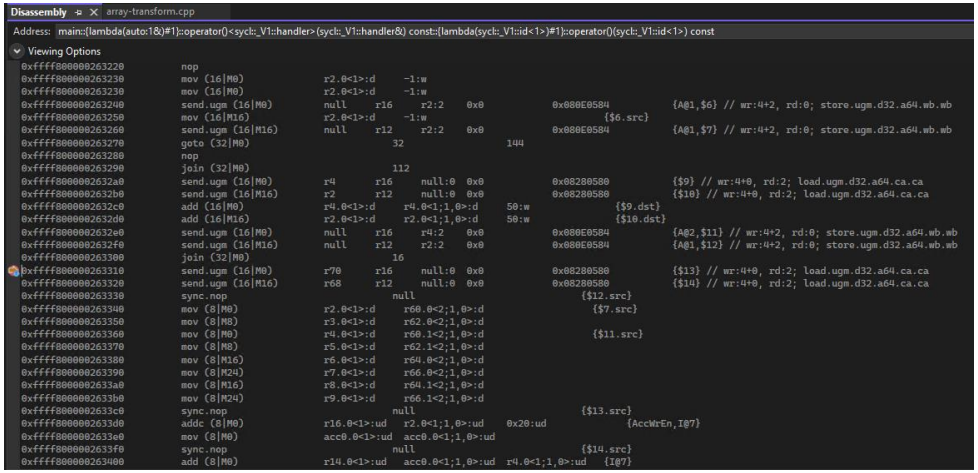
Locals

We can investigate local variables, by navigating to **Debug > Windows > Locals**.



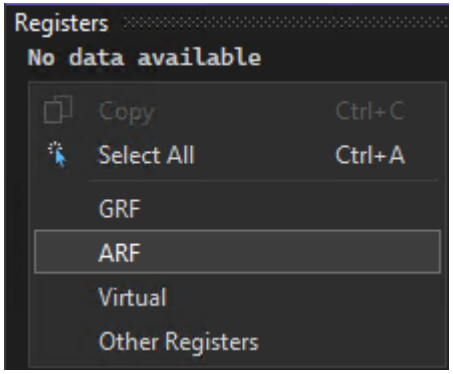
Disassembly

We can view the disassembly by navigating to **Debug > Windows > Disassembly.**

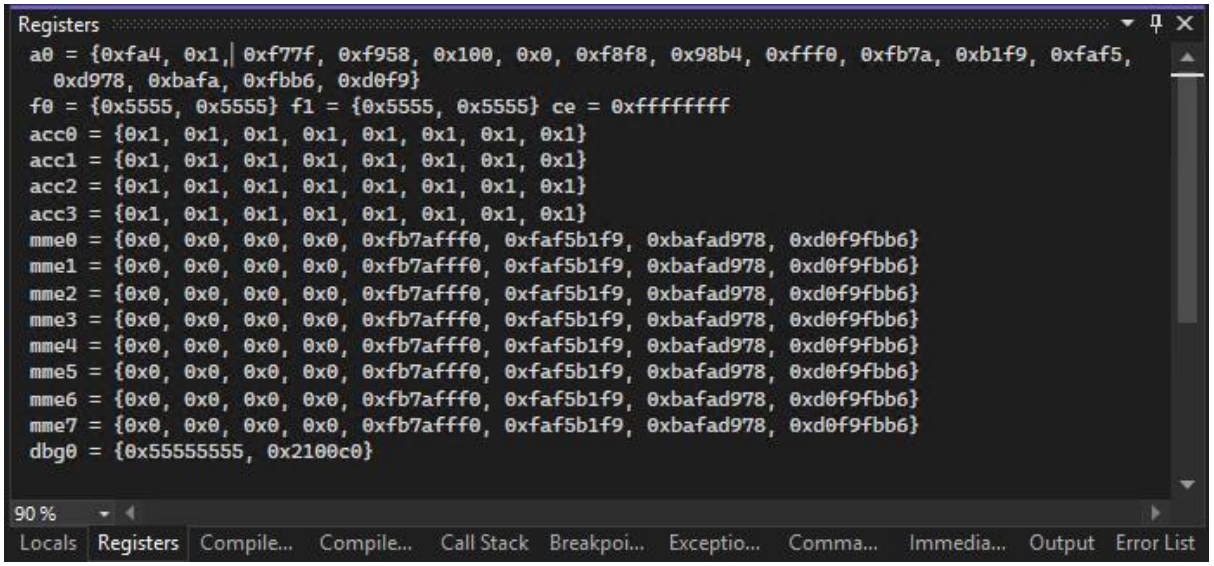


Registers

We can view respective registers when the kernel is offloaded to a GPU by right clicking inside the windows and selecting the register we want to view.



Below we see the ARF Registers displayed.



Below is a view of GRF Registers.

```
Registers
r0 = {0x190303, 0x1, 0x4010402, 0x0, 0x0, 0x1002, 0x0, 0x0}
r1 = {0x410040, 0x430042, 0x450044, 0x470046, 0x490048, 0x4b004a, 0x4d004c, 0x4f004e}
r2 = {0x198, 0xffffffff, 0x19a, 0xffffffff, 0x19c, 0xffffffff, 0x19e, 0xffffffff}
r3 = {0x1a0, 0xffffffff, 0x1a2, 0xffffffff, 0x1a4, 0xffffffff, 0x1a6, 0xffffffff}
r4 = {0x188, 0x0, 0x18a, 0x0, 0x18c, 0x0, 0x18e, 0x0}
r5 = {0x190, 0x0, 0x192, 0x0, 0x194, 0x0, 0x196, 0x0}
r6 = {0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7}
r7 = {0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf}
r8 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
r9 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
r10 = {0xce, 0xd0, 0xd0, 0xd2, 0xd2, 0xd4, 0xd4, 0xd6}
r11 = {0xd6, 0xd8, 0xd8, 0xda, 0xda, 0xdc, 0xdc, 0xde}
r12 = {0x147a260, 0xffffb802, 0x147a264, 0xffffb802, 0x147a268, 0xffffb802, 0x147a26c, 0xffffb802}

r13 = {0x147a270, 0xffffb802, 0x147a274, 0xffffb802, 0x147a278, 0xffffb802, 0x147a27c, 0xffffb802}

r14 = {0x147a280, 0xffffb802, 0x147a284, 0xffffb802, 0x147a288, 0xffffb802, 0x147a28c, 0xffffb802}
90 %
Locals  Registers  Compiler...  Compiler...  Call Stack  Breakpoi...  Exception...  Comman...  Immediat...  Output  Error List
```

Memory View

We can view the memory view by selecting **Debug > Window > Memory > Memory 1**. We can specify the memory address we want to inspect and enter it in the address field.

```
Memory 1
Address: 0xffffb8020147a1a0                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
0xFFFFB8020147A1A0  0124 0000 0125 0000 0126 0000 0127 0000 0128 0000 0129 0000 012a 0000 012b 0000 012c 0000 012d 0000 012e 0000 012f 0000 0130 0000 0131 0000 0132 0000 0133 0000
0xFFFFB8020147A1E0  0134 0000 0135 0000 0136 0000 0137 0000 0138 0000 0139 0000 013a 0000 013b 0000 013c 0000 013d 0000 013e 0000 013f 0000 0140 0000 0141 0000 0142 0000 0143 0000
0xFFFFB8020147A220  0188 0000 ffff ffff 018a 0000 ffff ffff 018c 0000 ffff ffff 018e 0000 ffff ffff 0190 0000 ffff ffff 0192 0000 ffff ffff 0194 0000 ffff ffff 0196 0000 ffff ffff
0xFFFFB8020147A260  0198 0000 ffff ffff 019a 0000 ffff ffff 019c 0000 ffff ffff 019e 0000 ffff ffff 01a0 0000 ffff ffff 01a2 0000 ffff ffff 01a4 0000 ffff ffff 01a6 0000 ffff ffff
0xFFFFB8020147A2A0  00ce 0000 0000 0000 00cf 0000 0000 0000 00d0 0000 0000 0000 00d1 0000 0000 0000 00d2 0000 0000 0000 00d3 0000 0000 0000 00d4 0000 0000 0000 00d5 0000 0000 0000
0xFFFFB8020147A2E0  00d6 0000 0000 0000 00d7 0000 0000 0000 00d8 0000 0000 0000 00d9 0000 0000 0000 00da 0000 0000 0000 00db 0000 0000 0000 00dc 0000 0000 0000 00dd 0000 0000 0000
0xFFFFB8020147A320  8e10 0147 b802 ffff 8e50 0147 b802 ffff 8e90 0147 b802 ffff 8ed0 0147 b802 ffff 8f10 0147 b802 ffff 8f50 0147 b802 ffff 8f90 0147 b802 ffff 8fd0 0147 b802 ffff
0xFFFFB8020147A360  9010 0147 b802 ffff 9050 0147 b802 ffff 9090 0147 b802 ffff 90d0 0147 b802 ffff 9110 0147 b802 ffff 9150 0147 b802 ffff 9190 0147 b802 ffff 91d0 0147 b802 ffff
0xFFFFB8020147A3A0  9210 0147 b802 ffff 9250 0147 b802 ffff 9290 0147 b802 ffff 92d0 0147 b802 ffff 9310 0147 b802 ffff 9350 0147 b802 ffff 9390 0147 b802 ffff 93d0 0147 b802 ffff
0xFFFFB8020147A3E0  9410 0147 b802 ffff 9450 0147 b802 ffff 9490 0147 b802 ffff 94d0 0147 b802 ffff 9510 0147 b802 ffff 9550 0147 b802 ffff 9590 0147 b802 ffff 95d0 0147 b802 ffff
0xFFFFB8020147A420  9c10 0147 b802 ffff 9c18 0147 b802 ffff 9c20 0147 b802 ffff 9c28 0147 b802 ffff 9c30 0147 b802 ffff 9c38 0147 b802 ffff 9c40 0147 b802 ffff 9c48 0147 b802 ffff
0xFFFFB8020147A460  9c50 0147 b802 ffff 9c58 0147 b802 ffff 9c60 0147 b802 ffff 9c68 0147 b802 ffff 9c70 0147 b802 ffff 9c78 0147 b802 ffff 9c80 0147 b802 ffff 9c88 0147 b802 ffff
0xFFFFB8020147A4A0  9c90 0147 b802 ffff 9c98 0147 b802 ffff 9ca0 0147 b802 ffff 9ca8 0147 b802 ffff 9cb0 0147 b802 ffff 9cb8 0147 b802 ffff 9cc0 0147 b802 ffff 9cc8 0147 b802 ffff
0xFFFFB8020147A4E0  9cd0 0147 b802 ffff 9cd8 0147 b802 ffff 9ce0 0147 b802 ffff 9ce8 0147 b802 ffff 9cf0 0147 b802 ffff 9cf8 0147 b802 ffff 9d00 0147 b802 ffff 9d08 0147 b802 ffff
Memory 1  Locals  Registers  Compiler Inline Report  Compiler Optimization Report  Call Stack  Breakpoints  Exception Settings  Command Window  Immediate Window  Output  Error List
```

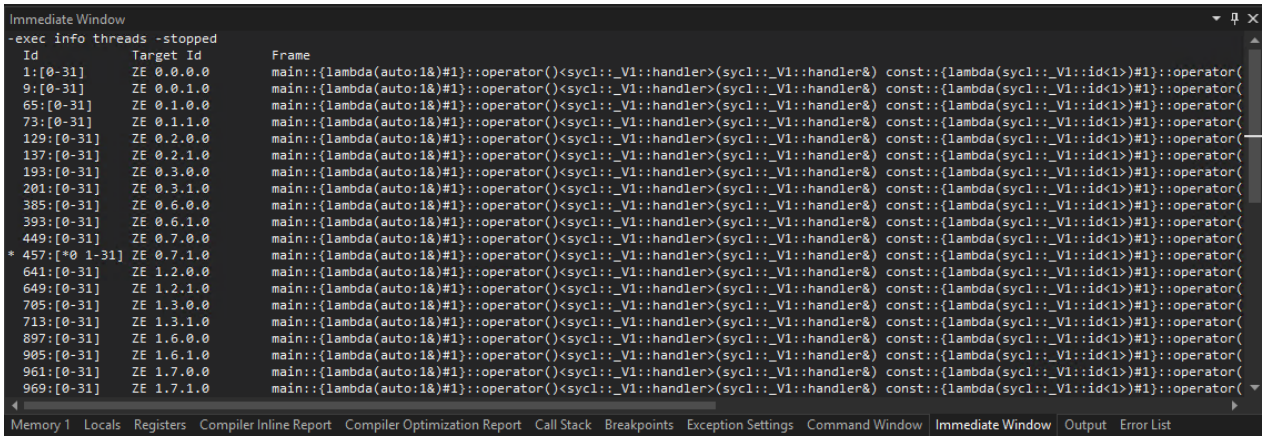
Debug Toolbar

The default debug toolbar available with Visual Studio can also be used when the kernel is offloaded to the GPU. The functionality provided by the debug toolbar are:

- Continue (F5)
- Pause (Ctrl+Alt+Break)
- Stop (Shift+F5)
- Restart (Ctrl+Shift+F5)
- Show Next Statement (Alt+Num*)
- Step Into (F11)
- Step Over (F10)
- Step Out (Shift+F11)

Immediate Window

For sycl developers who are more comfortable with command line interface, there is an immediate window where we can type gdb MI commands and see the results.



Linux – Visual Studio Code:

Debugging Environment and Setup

The full guide to getting started with Intel® Distribution for GDB on Linux can be found on the Intel's webpage online.

Prerequisites

Before we can debug our SYCL application on VS Code, we must ensure the following prerequisites are met:

- Resizable BAR or Smart Access Memory must be enabled for debugging applications using Intel® Arc™ Graphic cards. Please follow the instructions to enable the Resizable BAR.
- If you intend to debug on a GPU, first check whether your device is supported for debugging kernels offloaded to GPU by checking the list of supported accelerators. If your GPU device is not listed above, then a breakpoint inside the kernel won't be hit. In that case, you can still debug the offload to CPU. For more information, go to the GPU Driver Page.

Software Installation

The following software is required to debug a SYCL application on VS Code:

- Install VS Code on the host machine. We can also download VS Code on our Windows host and ssh to Linux target with the hardware accelerator.
- Install the Intel GPU drivers.
- Install Intel® oneAPI Base Toolkit.
- Install GDB with GPU Debug Support for Intel® oneAPI Toolkits extension on the VS Code. This needs to be installed on the target device.
- Install Environment Configurator for Intel Software Developer Tools on the target device optionally.

Set Up the GPU debugger

After installing Intel GPU drivers, we must take care of a few more things before we can start debugging our sample application on VS Code.

- We must enable i915 debug support in the Kernel. We can do this by setting the value of **prelim_enable_eu_debug** flag to **1** on the machine we want to debug by typing the following command:

for f in /sys/class/drm/card*/prelim_enable_eu_debug; do echo 1 | sudo tee "$f"; done

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   GITLENS   COMMENTS

rganesh@DUT1170DG2FRD  ~  for f in /sys/class/drm/card*/prelim_enable_eu_debug; do echo 1 | sudo tee "$f"; done
1
1
```

- We must enable i915 debug support in the kernel persistently. Also, by default, the GPU driver does not allow workloads to run on a GPU longer than a certain amount of time. To ensure that the driver does not kill long-running workloads by resetting the GPU to prevent hangs, we must disable hang check. Follow the below steps to enable i915 debug support and disable hang check:
    - Navigate to **/etc/default** and open the grub file in an editor.
    - Find the line **GRUB_CMDLINE_LINUX_DEFAULT="".**
    - Enter the following text between the quotes ("""):
      i915.debug_eu=1
      i915.enable_hangcheck=0

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   GITLENS   COMMENTS                    nano - default

  GNU nano 6.2                                                    grub
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=1
GRUB_TIMEOUT_STYLE="menu"
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="net.ifnames=0 biosdevname=0 i915.debug_eu=1 drm.debug=0xa i915.debugger_timeout_ms=0 i915.enable_hangcheck=0"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
# This works with Linux (no patch required) and with any kernel that obtains
# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
#GRUB_BADRAM="0x01234567,0xfefefefe,0x89abcdef,0xefefefef"
```

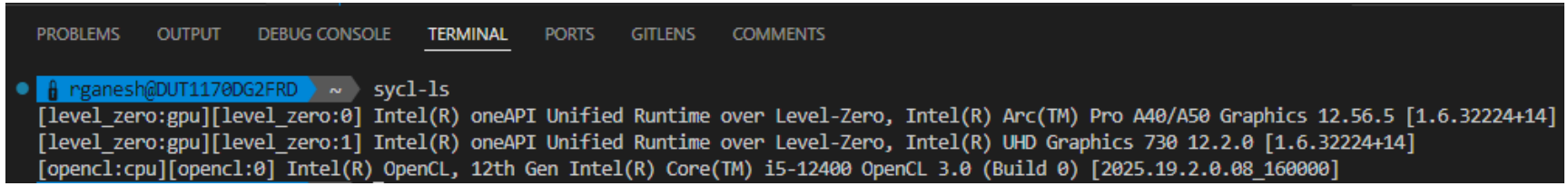
- Update GRUB and reboot for these changes to take effect by running the following command on the terminal:
  sudo update-grub
  sudo reboot now

- Setup oneAPI development environment by typing the following the terminal:
  source /opt/intel/oneapi/setvars.sh

  Note: This can be also done by using the [Environment Configurator for Intel Software Developer Tools]. We must open the command palette (Ctrl+Shit+P) and type **Intel oneAPI** to view the options. Then click **Intel oneAPI: Initialize default environment variables** and provide the path to **setvars.sh (/opt/intel/oneapi/setvars.sh)**. The advantage of using Environment Configurator is that the change now applies to all tasks, launch and new terminals, regardless of which folder it was originally associated with.
- Set up debug environment variables. Use the following environment variable to enable debugger support for Intel® oneAPI Level Zero:
  export ZET_ENABLE_PROGRAM_DEBUGGING=1

  Note: This can be also done later in the launch.json of the application we want to debug. This ensures that the variable is applicable only to the scope of the debug session.

- Perform system check to confirm system configuration is reliable by running the following command:
  sycl-ls



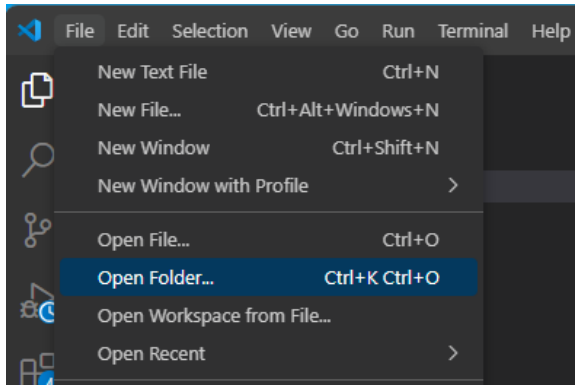
Build the application

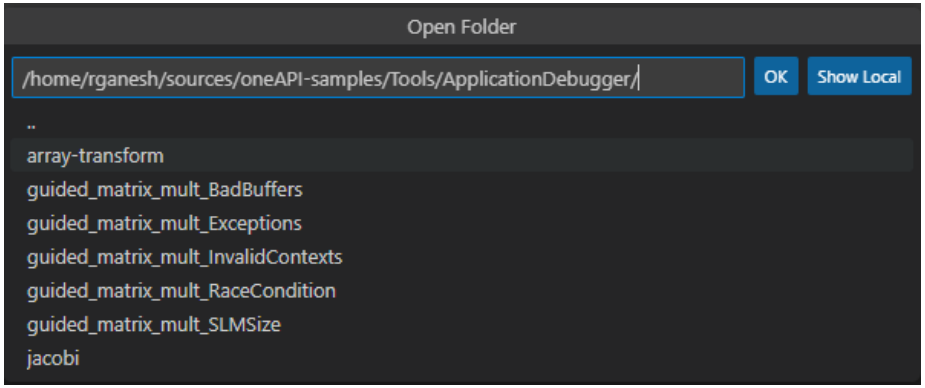
We should be now ready to debug a SYCL application on our Linux system. Here we shall use the sample array-transform application that is provided with the oneAPI Samples. Follow the steps to set up a sample application:

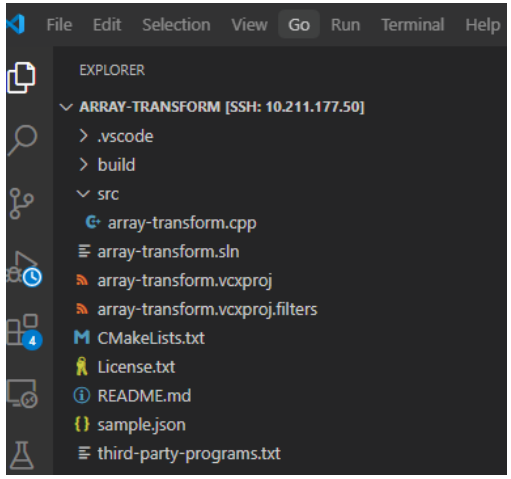
- Open a new terminal in a VS Code instance. Navigate to the folder where you want to place the source code and clone the oneAPI sample repository with the following command:

  git clone https://github.com/oneapi-src/oneAPI-samples.git

- Open the array-transform application in VS Code by navigating to **File > Open Folder** and specifying the path to array-transform application. It is in **oneAPI-samples/Tools/ApplicationDebugger/array-transform**. We can then see the folder structure of the application in the explorer view.







- Build your SYCL application by navigating to **oneAPI-samples/Tools/ApplicationDebugger/array-transform** from your terminal and running the following command:
  icpx -fsycl -g -O0 src/array-transform.cpp -o build/array-transform

  This shall place the array-transform ELF inside the build folder.

- Open the command palette (Ctrl+Shift+P) and type **Intel oneAPI**. Select **Intel oneAPI: Generate launch configurations.** Follow the prompts to add a SYCL launch configuration. If no environment variables were specified, we can specify them in the launch configuration.

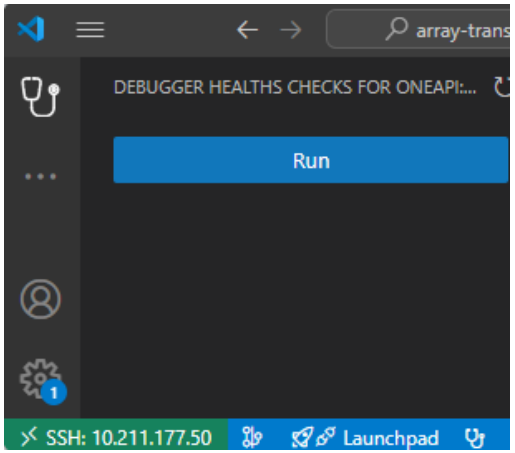
launch.json .vscode/launch.json/Launch Targets/{} ONEAPI_DEVICE_SELECTOR

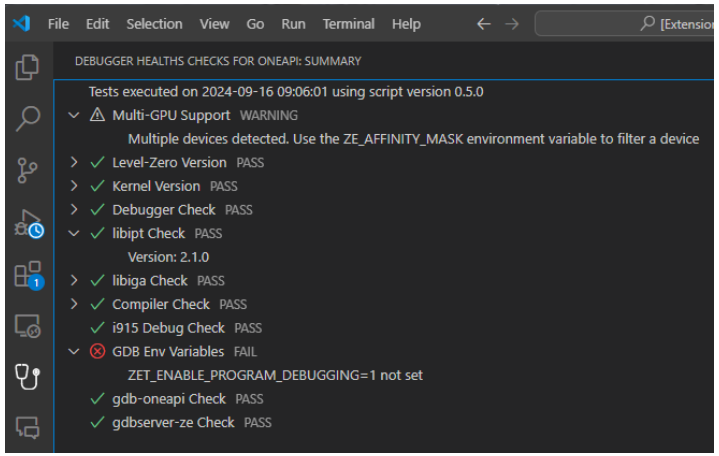
```
{
    "configurations": [
        {
            "name": "LaunchGdbOneApi",
            "miDebuggerPath": "/home/rganesh/./install/bin/gdb-oneapi",
            "MIMode": "gdb",
            "type": "cppdbg",
            "request": "launch",
            "preLaunchTask": "",
            "postDebugTask": "",
            "stopAtEntry": false,
            "program": "/home/rganesh/./sources/oneAPI-samples/Tools/ApplicationDebugger/array-transform/build/array-transform",
            "cwd": "${workspaceFolder}/build",
            "args": [],
            "environment": [
                {
                    "name": "ZET_ENABLE_PROGRAM_DEBUGGING",
                    "value": "1"
                },
                {
                    "name": "ONEAPI_DEVICE_SELECTOR",
                    "value": "level_zero:0"
                }
            ],
```

Debug Health Check

Similar to Windows, there is also a **Debugger Health Check** for Linux. We can run this to identify any setup and configuration issues. To run the health check, click the stethoscope icon in the status bar or click the stethoscope icon in the activity bar and click **Run.**

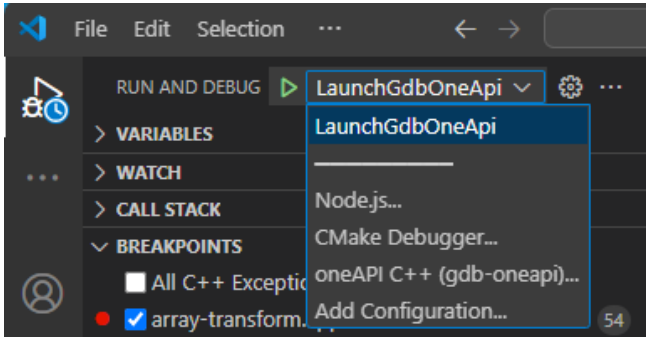
The tests will automatically execute, and results will be displayed in a tree format. Each check will show whether it passed, failed, or requires attention (warning). Hover over each test result to view additional information about the specific check, including version numbers and recommendations on how to resolve issues.





Debugging a GPU application

Now that we have already built our sample application and specified the path to built ELF in the launch.json file, we are ready to debug our SYCL application on VS Code. Place a breakpoint inside the kernel offloaded to the GPU which is indicated in the sample application code. Click **Run and Debug** from the activity bar, select the launch config we just created and click **Start Debugging**.



We should be able to hit the kernel breakpoint.

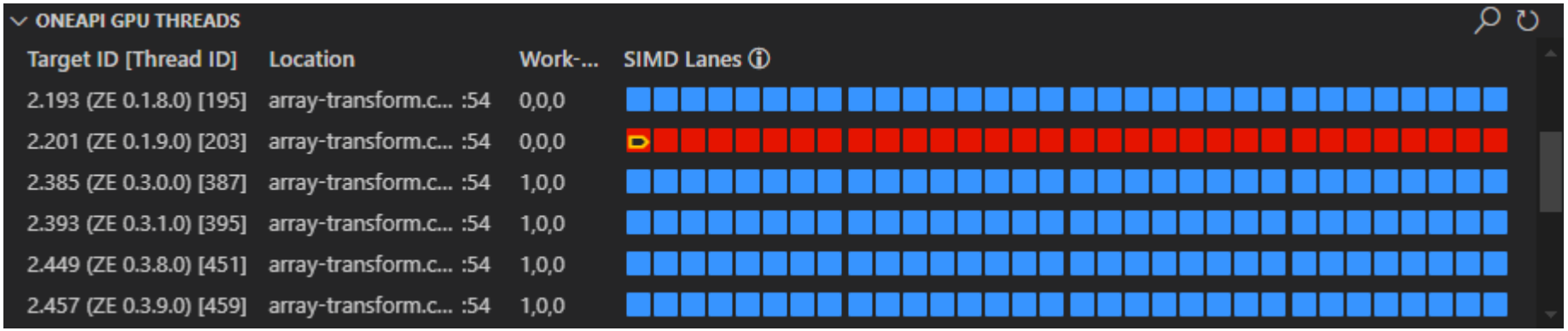
array-transform.cpp src/array-transform.cpp/ main(int, char * [])

```
26  int main(int argc, char *argv[]) {
35    try {
47      q.submit([&](auto &h) {
48        accessor in(buffer_in, h, read_only);
49        accessor out(buffer_out, h, write_only);
50
51        // kernel-start
52        h.parallel_for(data_range, [=](id<1> index) [[sycl::reqd_sub_group_size(32)]] {
53          size_t id0 = GetDim(index, 0);
54          int element = in[index];  // breakpoint-here
55          int result = element + 50;
56          if (id0 % 2 == 0) {
57            result = result + 50;  // then-branch
58          } else {
```

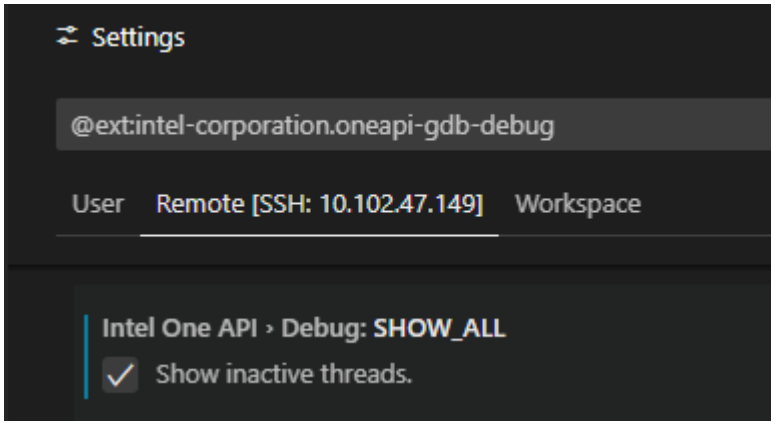
Features

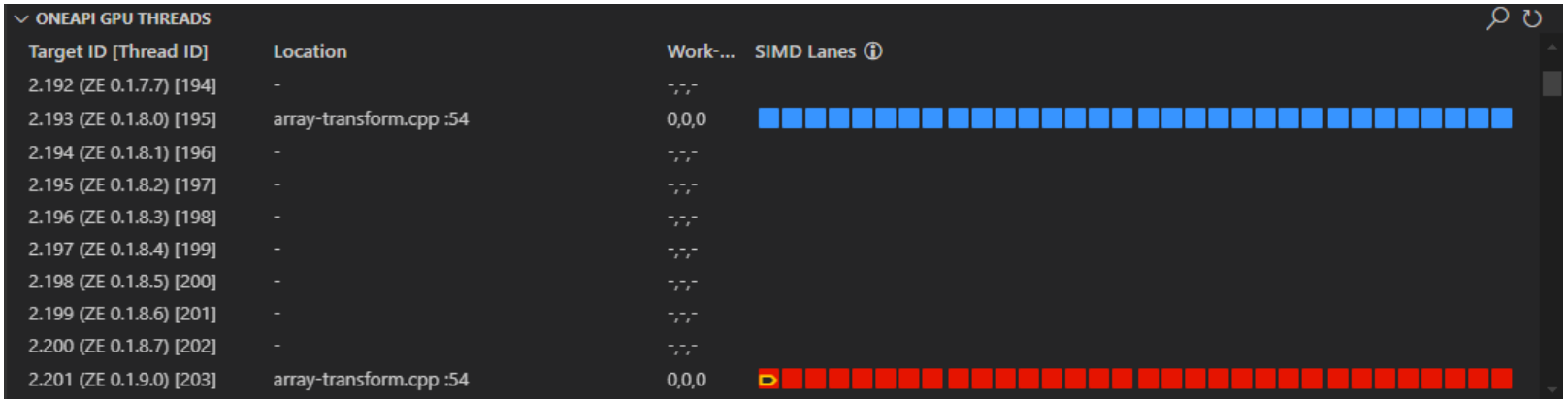
Intel oneAPI GPU Threads

Once we hit the kernel breakpoint, expand (if not already expanded) the **ONEAPI GPU THREADS** from the primary side bar. We should be able to see the active threads and SIMD lanes by default.

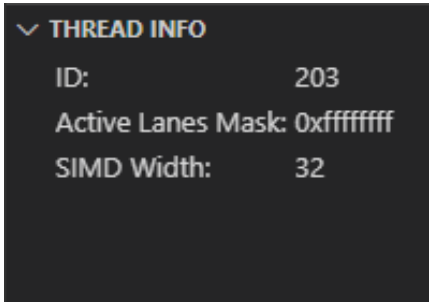


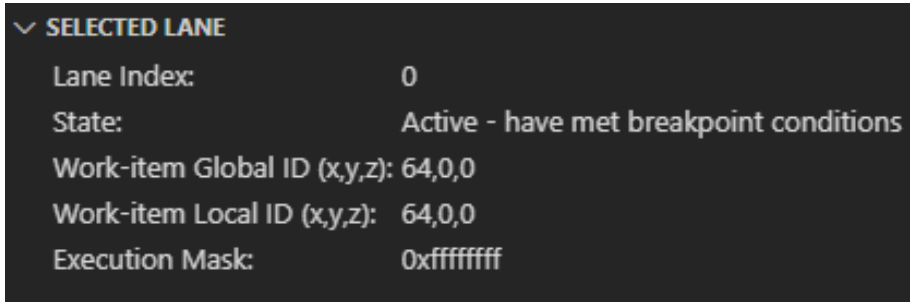
It is also possible to view all the threads (active and unavailable) by enabling the **Show inactive threads** in the extension settings.

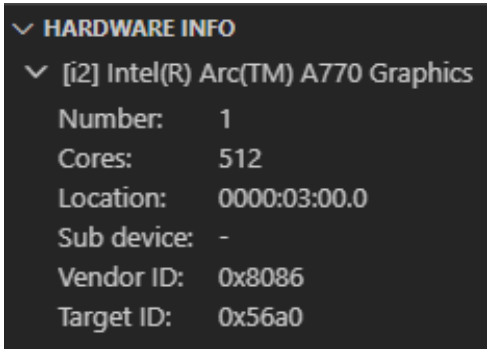




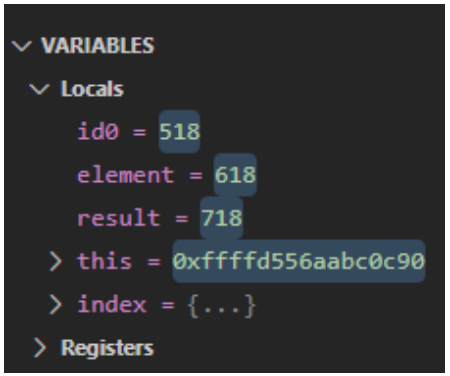
It is possible to view workgroup and location information for each active thread. We can also expand **THREAD INFO, SELECTED LANE** and **HARDWARE INFO** from the primary side bar. The Thread Info section contains the ID, Active Lanes Mask and the SIMD Width of the selected thread. The selected lane displays info about the Lane Index, State, Work-item Global ID, Work-item Local ID and the Execution Mask. The hardware info displays information regarding the current device used for offloading, such as Device Name, Number, Cores, Location, Sub device, Vendor ID and Target ID.

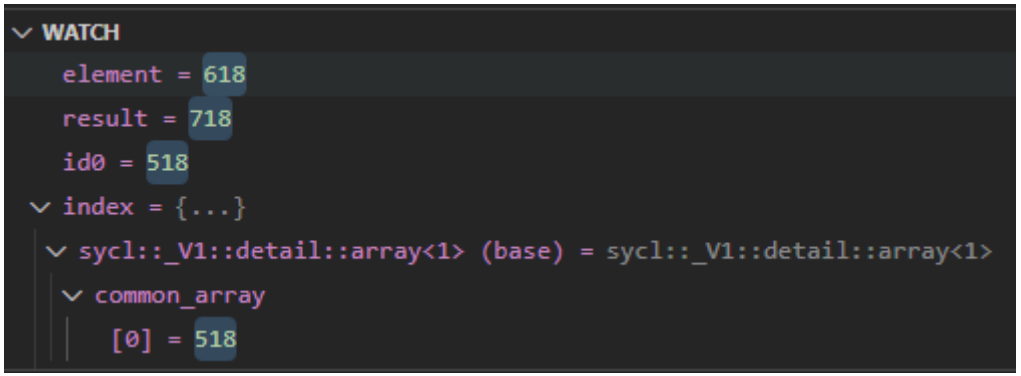


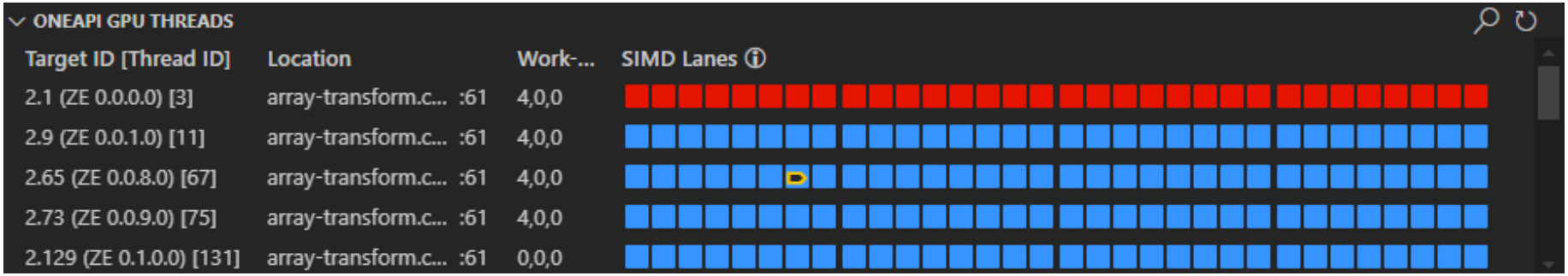




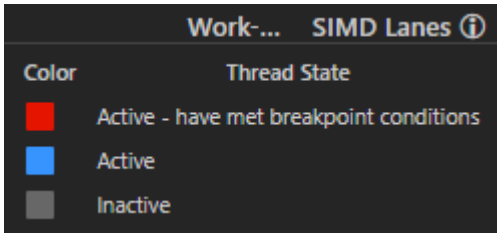
Just like in Visual Studio, it is also possible to switch to another active SIMD lane that does not meet the breakpoint condition by clicking it and see all the variable information for that lane on **VARIABLES/WATCH** window.



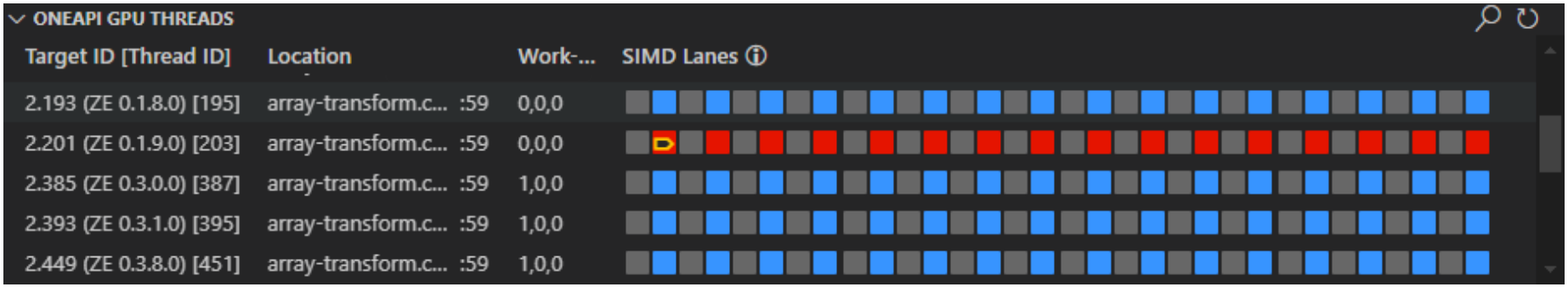




We can identify the SIMD lane colour scheme for the current VS Code theme by clicking the information button next to **SIMD Lanes** column in the ONEAPI GPU Threads window. This opens a popup that signifies the meaning of each colour

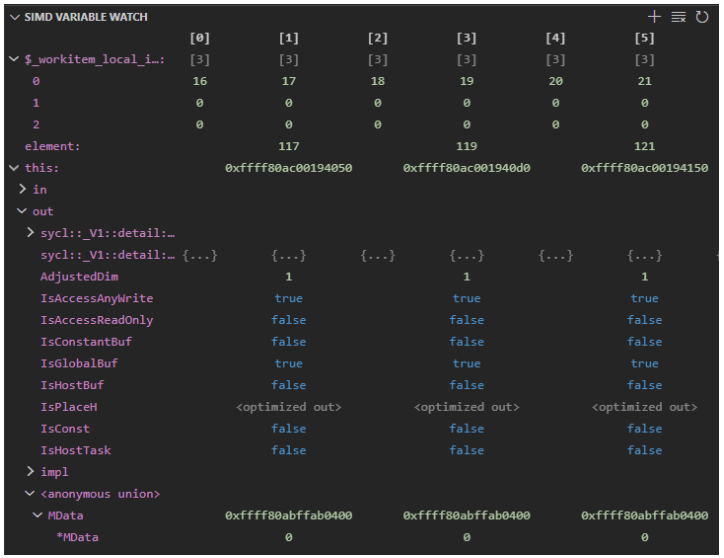


This colour scheme helps us identify conditions where some of the SIMD lanes are inactive in the ONEAPI GPU Threads window.



Intel oneAPI SIMD Variable Watch

The SIMD Variable Watch, when expanded, functions like the classic Watch panel but displays values for all SIMD lanes, making it convenient to compare values without the need for switching between lanes. We can add expressions that we want to be evaluated for active lanes, however GDB convenience variables are usually available for inactive lanes as well. For instance, values for $_workitem_global_id or $_workitem_local_id are evaluated for all the SIMD lanes in the current thread.

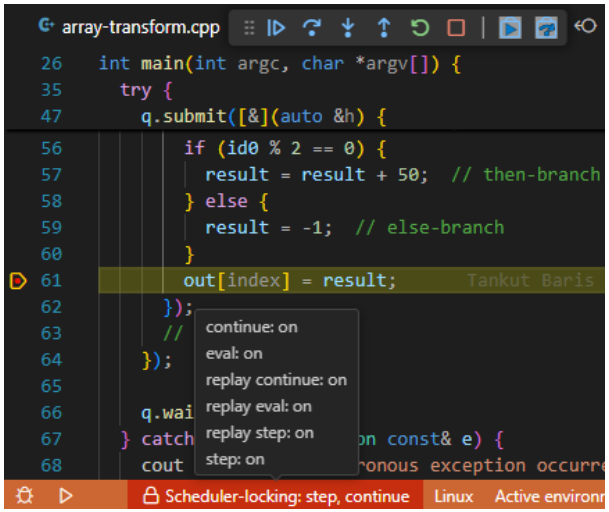


Scheduler-locking

Buttons in the debug toolbar provide quick access to turning scheduler-locking on or off for step and continue flags. Scheduler-locking controls how GDB handles other threads during debugging.

- step: When **on**, the scheduler is locked for stepping commands during normal execution and record modes. While stepping, other threads may not pre-empt the current thread, so that the focus of debugging does not change unexpectedly. This setting is **off** by default.
- continue: When **on**, the scheduler is locked for continuing commands during normal execution and record modes. For continuing commands other threads may not pre-empt the current thread. This setting is **off** by default.

The overall status of scheduler-locking is displayed in the status bar.



SIMD Lane specific breakpoint

We can also add a SIMD Lane specific breakpoint inside a kernel in VS Code, which respects the SIMD Lane conditions. To place SIMD lane specific breakpoint inside a kernel, we place an ordinary breakpoint. Once we hit this breakpoint, we must right-click on the desired line and select **Add Conditional Breakpoint...**. Choose **Expression** from the dropdown and use the following commands:

-break-insert -p <ThreadId> -l <SIMD Lane>

```
51      // kernel-start
52      h.parallel_for(data_range, [=](id<1> index) [[sycl
53        size_t id0 = GetDim(index, 0);
54        int element = in[index];  // breakpoint-here
55        int result = element + 50;
56        if (id0 % 2 == 0) {
57          result = result + 50;  // then-branch
58        } else {
59          result = -1;  // else-branch
```

Add Breakpoint
Add Conditional Breakpoint...
Add Logpoint...

```
51      // kernel-start
52      h.parallel_for(data_range, [=](id<1> index) [[sycl
53        size_t id0 = GetDim(index, 0);
54        int element = in[index];  // breakpoint-here
55        int result = element + 50;
56        if (id0 % 2 == 0) {
57          result = result + 50;  // then-branch
58        } else {
59          result = -1;  // else-branch
```

Expression -break-insert -p 203 -l 1

The new condition breakpoint can be inspected then in the UI by hovering on it.

```
57          result = result + 50;  // then-branch
```

Click to add a breakpoint

Condition "-break-insert -p 203 -l 1" : No symbol "break" in current context.

The conditional breakpoint hit is reported as an exception.

```
57          result = result + 50;  // then-branch
58        } else {
59          result = -1;  // else-branch
```

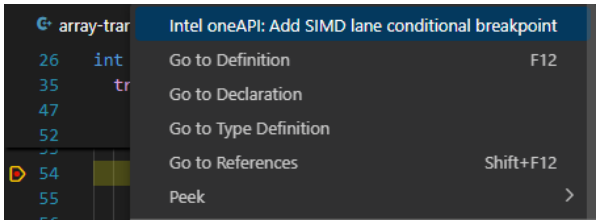
Exception has occurred.
Hit breakpoint 3 at 0xffff8000ffe95590.

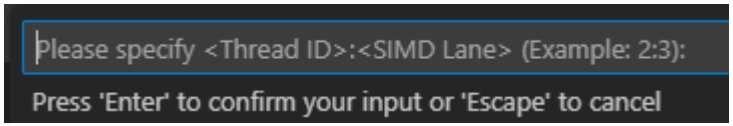
The same is reflected by a single SIMD Lane being active in the oneAPI GPU Thread view.

ONEAPI GPU THREADS

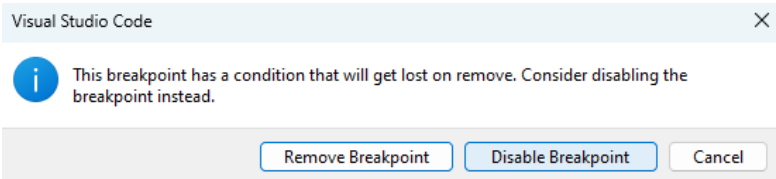
Target ID [Thread ID]	Location	Work-...	SIMD Lanes
2.137 (ZE 0.1.1.0) [139]	array-transform.c... :59	0,0,0	
2.193 (ZE 0.1.8.0) [195]	array-transform.c... :59	0,0,0	
2.201 (ZE 0.1.9.0) [203]	array-transform.c... :59	0,0,0	
2.385 (ZE 0.3.0.0) [387]	array-transform.c... :59	1,0,0	

It is also possible to add a SIMD Lane specific breakpoint by right clicking on the line where we want to place the breakpoint (not the line number) and selecting **Intel oneAPI: Add SIMD lane conditional breakpoint**. We shall then be prompted to specify the Thread ID and the SIMD Lane where we want to break. This method does not require us to remember any breakpoint specific syntaxes and provides the same result.



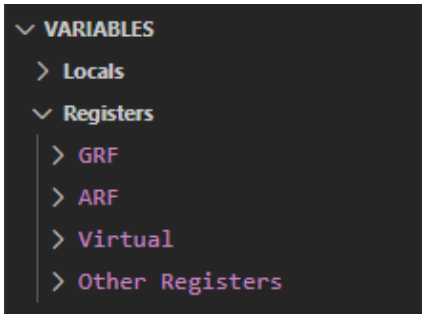


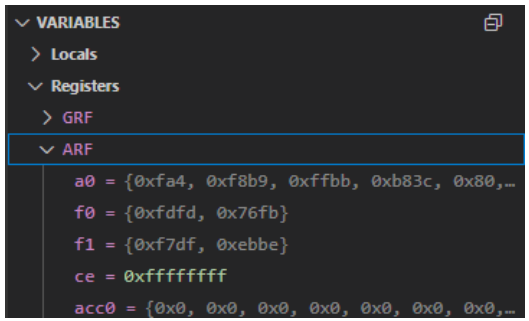
To remove or disable the conditional breakpoint, click on the breakpoint and perform the desired action.



Registers

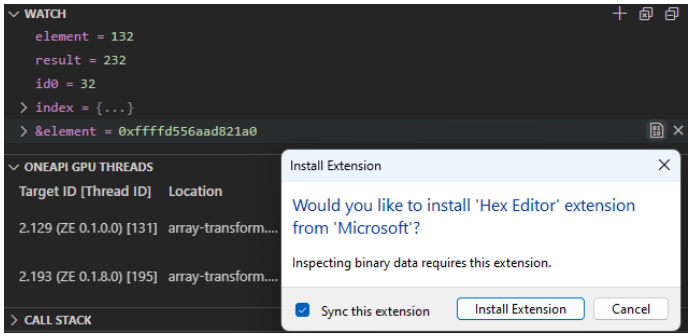
Viewing GPU registers possible inside VS Code by expanding the **VARIABLES** in the Primary Side Bar and scrolling down and expanding **Registers**. The various GPU registers are grouped together as GRF, ARF, Virtual and Other Registers. These can be further expanded and viewed.

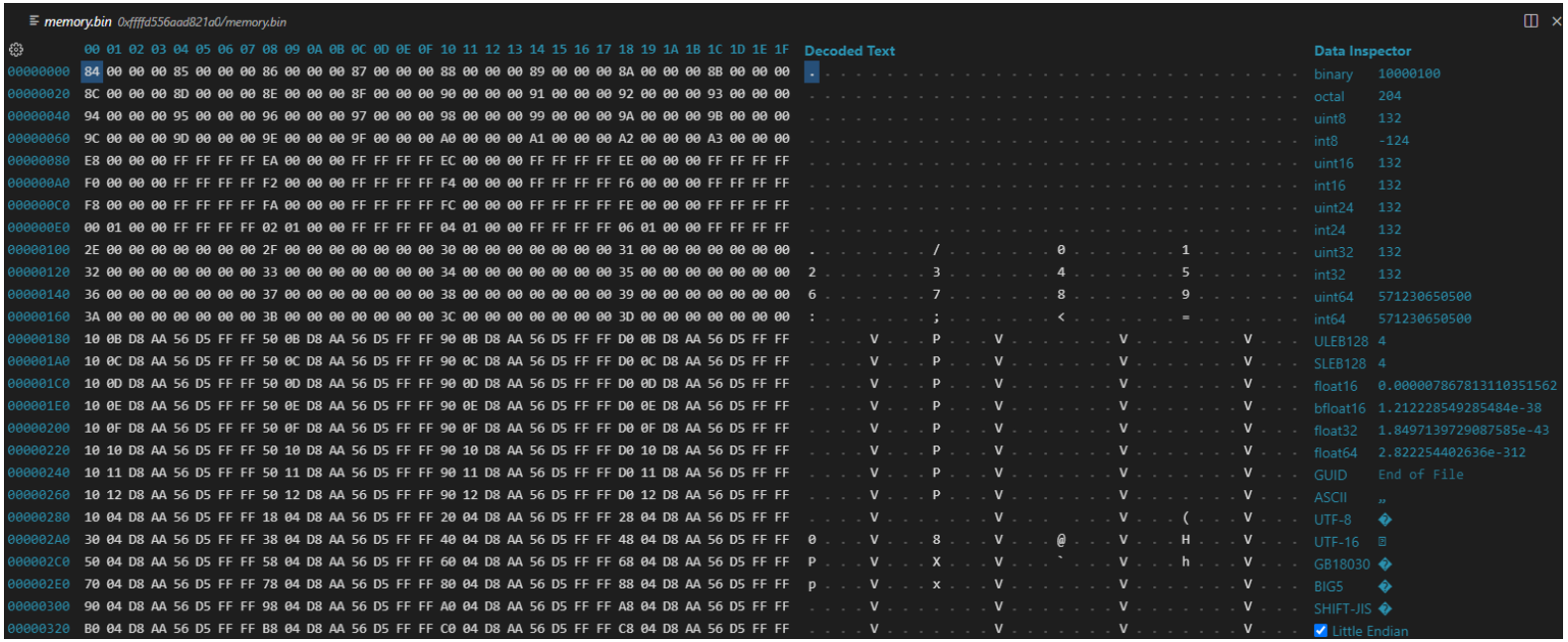




Memory View

We can view the memory view by viewing the address of a variable in the watch/variables window and clicking **View Binary Data**. This will prompt us to install Hex Editor extension from Microsoft. When installed, it will open a memory.bin, in which is possible to inspect conveniently large pieces of data. This functionality enables users to examine the memory space of Intel® GPU kernels.





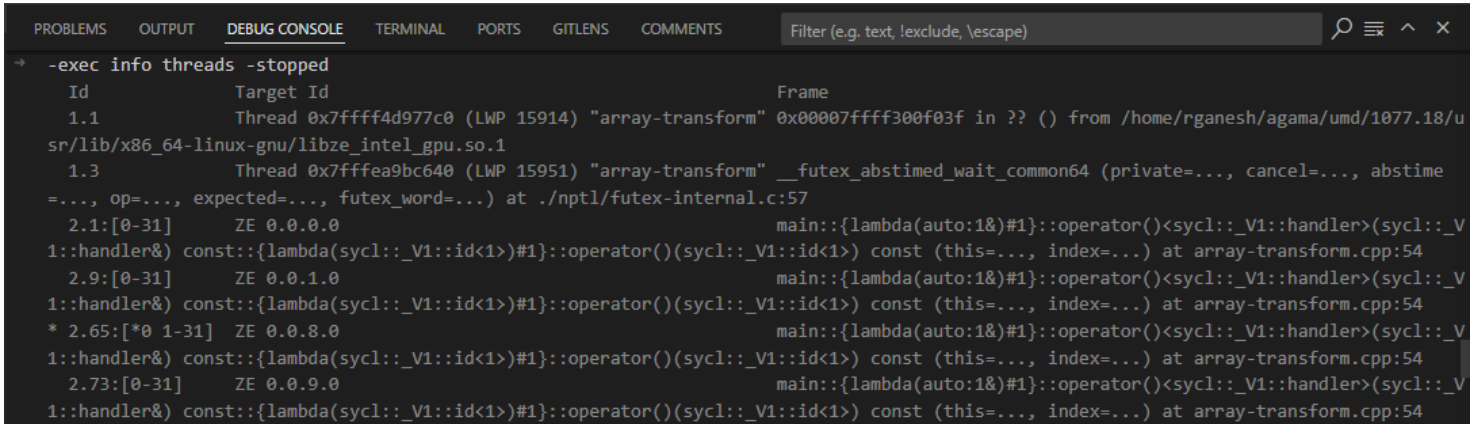
Debug Toolbar

The debug toolbar available with VS Code by default. This can also be used when the kernel is offloaded to the GPU. The functionality provided by the debug toolbar are:

- Continue (F5)
- Step Over (F10)
- Step Into (F11)
- Step Out (Shift+F11)
- Restart (Ctrl+Shift+F5)
- Stop (Shift
- Scheduler-locking continue (Intel oneAPI GDB extension)
- Scheduler-locking step (Intel oneAPI GDB extension)

Immediate Window

Again, for SYCL developers who are more comfortable with command line interface, there is **Debug Console** where we can type gdb MI commands and see the results.



Conclusion

The Intel® Distribution for GDB delivers a UI-rich debugging experience tailored for SYCL developers, integrating seamlessly with Visual Studio on Windows and VS Code on Linux. By abstracting away complex GDB commands, it enables developers to focus on writing high-performance SYCL code rather than struggling with low-level debugging intricacies. Features such as GPU state analysis using GPU Thread Window, variable inspection using SIMD Lane Variable Watch, in addition to Watch, Locals and Autos, Disassembly and Registers view, Memory view and kernel debugging, provide deep insights into SYCL execution, helping developers identify bottlenecks and optimize performance efficiently. As Intel continues to enhance its debugging ecosystem, future updates will further refine SYCL debugging workflows, empowering developers with cutting-edge tools for heterogeneous computing.