Debugging SYCL Programs on Heterogeneous Intel® Architectures

Barış Aktemur*
tankut.baris.aktemur@intel.com
Intel GmbH
Neubiberg, Germany

Markus Metzger
markus.t.metzger@intel.com
Intel GmbH
Neubiberg, Germany

Natalia Saiapova
natalia.saiapova@intel.com
Intel GmbH
Neubiberg, Germany

Mihails Strasuns
mihails.strasuns@intel.com
Intel GmbH
Neubiberg, Germany

ABSTRACT

Intel recently announced a large initiative named oneAPI that provides a direct programming model based on SYCL. As part of the oneAPI distribution, we developed a debugger that can be used for debugging SYCL programs that offload kernels to CPU, GPU, or FPGA emulator devices. The debugger is based on GDB. It allows programmers to inspect the host and kernel portion of their SYCL programs seamlessly in the same debug session. To realize the debugger, we made enhancements to GDB including SIMD-based thread views and C++-related improvements. In this work we present the general architecture of the debugger, provide a sample session of how it can be used to debug a SYCL kernel running on a GPU, and discuss the encountered and anticipated challenges during the development phase. Currently a beta version of the debugger is publicly available.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **Computing methodologies → Parallel programming languages**; • **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

sycl, gdb, debugging, heterogeneous computing

ACM Reference Format:
Barış Aktemur, Markus Metzger, Natalia Saiapova, and Mihails Strasuns. 2020. Debugging SYCL Programs on Heterogeneous Intel® Architectures. In *International Workshop on OpenCL (IWOCL '20), April 27–29, 2020, Munich, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3388333.3388646

*Authors are listed alphabetically.

IWOCL '20, April 27–29, 2020, Munich, Germany

ACM ISBN 978-1-4503-7531-3/20/04.
https://doi.org/10.1145/3388333.3388646

1 INTRODUCTION

Modern computing problems imply workload diversity. To achieve the best performance, programmers typically need to resort to a variety of hardware (e.g. CPU, GPU, accelerators) as well as libraries, tools, and APIs at the expense of increased development effort and complexity. Intel recently announced a large initiative, named oneAPI [6], that aims to address these problems by providing a unified programming model with which programmers can develop applications across multiple architectures. For a direct-programming approach, oneAPI supports the SYCL language [12].

Debugging is an indispensable part of the software development practice. Debug tools play a critical role for the developers to inspect programs and track down bugs. In this work, we present a debugger targeted for SYCL programs. Our debugger is based on GDB [9] and is published as part of the Intel oneAPI Base Toolkit[1].

In SYCL, a portion of computation, called the *kernel*, is offloaded to a SYCL device, or executed on the host CPU if no underlying device exists. The device can be the CPU, GPU, or an accelerator, e.g. an FPGA. Our debugger can be used for debugging SYCL programs that use the host or the CPU, GPU, or an FPGA software emulator as the device. In this paper we focus on targeting a GPU device – we think this is the most interesting use case. The debugger supports offloading to Intel CPU and Intel GT devices; however, the overall architecture should apply to any SYCL debugging scenario. The debugger can be used on the Linux and Windows operating systems. In this paper, we focus on the Linux use-case. The key features of our debugger include

- defining breakpoints inside and outside the kernel; performing common debugging actions for code executed on both the host CPU and the device, such as inspecting the registers and the memory, source-level as well as instruction-level stepping, and back-tracing.
- viewing the SIMD (Single Instruction Multiple Data) lanes of threads and changing the current lane. Expressions are evaluated in the context of the current SIMD lane.
- plug-in definitions to be able to evaluate expressions that contain certain SYCL template functions.

The contributions of this paper are the following:

[1] http://software.intel.com/oneapi

- We present the architecture of the debugger based on multi-target GDB that allows debugging both the host and the kernel portion of a SYCL program.
- We discuss challenges we faced during the implementation; in particular, regarding
  - implicit pass-by-reference arguments,
  - C++ functions with template parameters,
  - thread view with SIMD lanes,
  - modeling device threads, and
  - conditional breakpoints.

  While we were able to overcome the former three of these challenges, we are still working on the latter two. For each challenge, we present the problem and discuss the solution we applied or are currently evaluating.

A major advantage of SYCL from a debugging point of view is that a SYCL program can be executed completely on the host CPU, without dispatching any kernel to a device. This way, the users may be able to detect logical errors in their programs by using off-the-shelf debug tools. Nevertheless, some bugs reveal themselves only when executed on the device. For such cases, we think that device-targeting debuggers such as ours are invaluable, where the user can go all the way down to the level of machine instructions and registers.

There is a third class of debugging tools that provide device-agnostic, yet OpenCL-specific debug capabilities. Oclgrind [15] is such a tool. It is an OpenCL simulator that implements the OpenCL 1.2 API. It takes OpenCL C source or SPIR intermediate-level code and simulates an OpenCL device. Oclgrind provides a plug-in architecture where hooks for certain operations (e.g. kernel begin/end, memory access, work item begin/end) can be defined. Oclgrind leverages this architecture to also provide an interactive debugger with which breakpoints can be defined and memory/variables can be inspected. Because Oclgrind is a simulator, it is deemed to run orders of magnitude slower than an actual device execution. We believe that tools like Oclgrind and device-aware debuggers like ours are complementary to each other and provide the users with a comprehensive debug capability to tackle various types of bugs.

The paper is organized as follows: Section 2 gives background information on how a traditional debugger works and what happens during the runtime of a SYCL program. Section 3 presents the architecture of our debugger. We show in Section 4 how a sample SYCL program can be debugged. In Section 5, we talk about the challenges we faced. Section 6 discusses related work. Finally, in Section 7, we give our conclusions.

2 BACKGROUND

In this section we give high-level background information about the working principles of debuggers in general, and about debugging offloaded computation in the context of SYCL.

2.1 Debugging Applications

Debuggers are tools that allow developers to find issues in their program by controlling the execution of their program and by inspecting, or even modifying, program state during its execution.

There are many different types of debuggers for debugging user applications, device drivers, operating systems, or firmware. They all rely on some lower layer or, lacking that, a debug monitor inside the debuggee, to exercise control over the debuggee. We focus on application debuggers.

The debugger itself is a separate process. It relies on Operating System (OS) support for controlling the debuggee process. In order to start debugging, the debugger sends a request to the OS to attach itself to the debuggee process. The debuggee can either be an already running process or it can be a new process that has been launched by the debugger. The OS is responsible for checking permissions for one process to control another process.

Once permission has been granted and the debugger is attached to the debuggee, it receives debug events for the debuggee process. Any exception is delivered to the debugger instead of the debuggee[2]. The debugger may then decide to handle the exception or to inject it into the debuggee.

The OS further provides a means for reading and writing the debuggee process' memory and debuggee threads' register state. The debugger works on a copy of the register state that had been stored when the debuggee thread was scheduled out and that will be restored when the debuggee thread is scheduled in again. Any modifications made by the debugger take effect at that time.

Finally, the OS provides a means for the debugger to alter the execution of the debuggee process. The debugger may interrupt a running debuggee thread and resume it again. The debugger may further insert (code) breakpoints[3] that stop the debuggee at the intended location. In addition to code breakpoints, debuggers may also offer breakpoints that trigger when a given memory location is accessed, when a given value is written, or when a given register has a given value. All forms of breakpoints depend on hardware support for an efficient implementation.

The debugger's main task, then, is to translate between the source-level world of the user and the machine-level world of the program. It does so with the help of debug information (e.g. DWARF [4]) that is generated by the compiler together with the machine code. Debug information maps the machine code back to the source code from which it was generated.

2.2 Debugging Offloading

For SYCL applications, in addition to debugging the host part, we also need to debug kernels offloaded to attached devices. This requires a similar debug interface we use on the host for controlling the host application for each device. It also requires debug information to be generated both for the host application and for each kernel.

Building a SYCL application involves multiple compilation steps as shown in Figure 1. The host compiler extracts kernels from the SYCL source code. Those are translated into the SPIR-V [13] intermediate representation and stored in the application binary (or on the disk, as a separate file) for later processing. The host code is compiled normally.

[2]Some exceptions cannot be intercepted by the debugger. On Linux, for example, the SIGKILL signal terminates a process whether a debugger is attached or not. The debugger is notified of the debuggee's termination after the fact.

[3]Depending on the processor, breakpoints can be implemented as a special instruction that raises an exception when executed, as a field in the instruction opcode that makes the instruction raise an exception when executed, or as separate hardware logic that compares some (optionally masked) state against a configurable value.

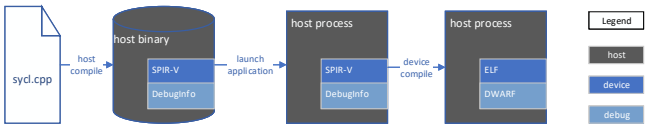

Figure 1: SYCL compilation flow.

When the SYCL application is executed, the device compiler translates kernels from SPIR-V into native device code for the device(s) selected by the application. In the case of Intel's oneAPI, this second compilation step produces an in-memory ELF [5] file.

Kernels could also be compiled ahead of time. In that case, the resulting ELF file would be stored in the host application binary. As with the dynamic compilation approach, the device ELF file would end up in the host application's memory where a debugger can find it.

Information about the source program is generated by the host compiler front-end and attached to the program's intermediate representation. For kernels, this information is attached to the kernel's SPIR-V using the format specified in the OpenCL.DebugInfo SPIR-V extension [17]. The device compiler takes it from there and adds DWARF debug information sections to device ELF file.

3 ARCHITECTURE

Figure 2 shows a high-level overview of the debug architecture for Intel GT devices on Linux. On Windows, we use a slightly different architecture, which is not covered here. The boxes in blue highlight areas where support for offloading debugging needs to be added. This section focuses on the lower layers. In Section 5.3 we describe some of the GDB and command-line interface (CLI) changes.

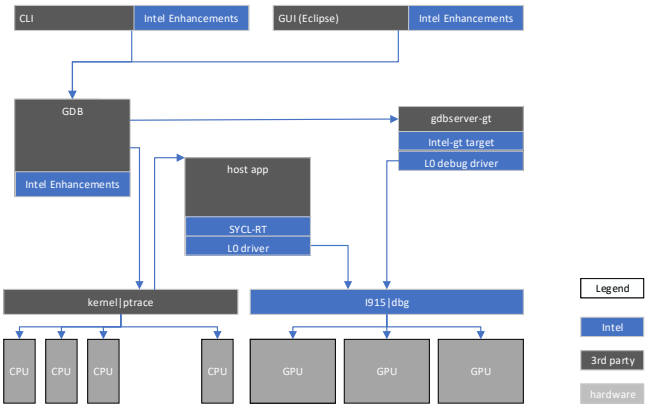

Figure 2: SYCL debug architecture.

We use GDB as the debug engine. For debugging the host application, we do not need any changes to standard GDB. For debugging offloaded kernels running on Intel GT devices, we retargeted GDB to support the Intel GT architecture and to control kernels via a debug interface provided by a companion driver to Intel's graphics device driver.

The debug interface provides access to device thread state and memory as well as run-control capabilities for stopping, resuming, and single-stepping device threads. As part of the oneAPI initiative, Intel is refining the debug interface and publishing it as part of their level-zero specification [10]. A goal of the level-zero interface is to enable third-party tool development. Our current implementation is using a predecessor of this interface.

GDB's new multi-target [1] feature allows debugging both host application and offloaded kernels in the same debug session. We represent Intel GT devices as separate inferiors[4] that are debugged via a remote connection to our retargeted gdbserver. The host application can be debugged via either a native or remote connection.

When the host application has started, we can create a second inferior and connect to our retargeted gdbserver. We pass the host process identifier and use it to attach to the Intel GT debug companion driver. Once we are attached, we can debug kernels offloaded by that host process. We automated those steps using GDB's Python API (see Section 4 for a demonstration).

For offloading to an Intel GT device, the SYCL runtime uses the Intel Graphics Compute Runtime for OpenCL (a.k.a. NEO), as its backend. NEO includes the Intel Graphics Compiler for OpenCL (IGC) [2] to jit-compile the kernel. An attached debugger changes some flows in NEO and IGC:

- IGC generates debug information and disables most optimizations.
- IGC generates a special cookie with a hard-coded breakpoint as the first instruction of every kernel. We use the hard-coded breakpoint to detect kernel submissions and the cookie to identify kernels.
- NEO generates device code to install a system routine for every kernel. The system routine is an exception handler that is invoked by the hardware for exceptions on the device such as breakpoint hits.
- NEO allocates a context save area to hold the thread state for every device context. Each thread is assigned a slot in this area. A thread's offset to its assigned slot is computed from its coordinates within the device. This memory is used by the system routine to store the thread state and hence needs to be mapped into the context's page tables.
  Our current implementation allocates the context save area per kernel. For some applications, however, that keep many kernels around, the memory overhead can become significant. We are currently investigating a different model where a single buffer can be shared.
- NEO informs the attached debugger about new kernels and maps a kernel's cookie to its corresponding ELF file.

Debuggers need to attach as early as possible so debug can be enabled for every kernel. Kernels that were built before the debugger attached cannot be debugged. The new level-zero debug interface attempts to relax this requirement.

A kernel is wrapped into a hardware context that contains instructions to initialize the hardware state and to launch the kernel. Those instructions specify the dimensions, for example, and hence define how many threads are needed to execute the kernel.

When the kernel is launched, threads are dispatched to execute kernel code. They hit the hard-coded breakpoint and enter the

[4] An *inferior* in GDB terminology is the program execution that is under inspection. Typically, it is a running process or a core-dump file.

system routine. The system routine stores the thread's state into its assigned context save area slot and sends an attention request to the CPU.

On the CPU side, the debug companion driver polls for device threads that requested attention. For each thread, the driver tries to determine the reason for the attention request and sends a corresponding debug event to the attached debugger.

When the first thread hits the kernel entry breakpoint, an event for a new kernel is generated. We use the kernel's cookie to find the corresponding ELF file. To reduce the overhead, the breakpoint is then removed and further hits from other threads are ignored.

Upon receiving the event for a new kernel, the debugger loads the kernel's ELF file and processes its DWARF debug information using standard GDB functionality. Currently, the debug stack only supports one kernel at a time per device, so an event for a new kernel implicitly means that the previous kernel completed. Before loading the new kernel's ELF file, we unload the previous kernel's.

We further remove existing threads and start over with a single thread for the new kernel. Additional threads are added as they hit breakpoints. A thread that does not hit any breakpoint is ignored. This allows us to keep the number of threads and the associated overhead manageable most of the time. See Section 5.4 for a discussion of that challenge.

3.1 Non-Stop vs. All-Stop

GDB supports two modes for controlling inferior processes: all-stop and non-stop. In *all-stop* mode, which is the default, any event stops the entire inferior process. If more than one thread report events at the same time, GDB queues the events and reports the first one to the user. When the user resumes the inferior, GDB reports the next queued event instead of resuming the process. Only when there are no more events to report, GDB resumes the inferior process.

In *non-stop* mode [16], an event stops only the eventing thread. Other threads continue running. Events are further reported immediately and asynchronous to user commands.

GDB further supports a hybrid mode where the target is running in non-stop mode and reports events asynchronously to GDB, yet GDB implements all-stop mode on top. To the user, it appears as all-stop mode. This all-stop on top of non-stop mode is supported by Linux native and remote targets.

When debugging heavily multi-threaded inferiors, stopping all other threads on an event can cause a fair amount of overhead. It further serializes debugging. The inferior does not make progress while the user inspects the state of a stopped thread. In non-stop mode, other threads continue running. When the user is done inspecting a thread after a breakpoint hit, chances are that the next thread is already waiting to be examined.

Kernels offloaded to GPUs are by nature very heavily multi-threaded. Everything that applies to multi-threaded applications applies to kernels, as well.

Asynchronously stopping individual threads, however, which is supported by non-stop mode, is not directly supported by Intel GT hardware and needs to be emulated by stopping all threads and resuming the ones that were not intended to be stopped. Repeatedly stopping individual threads may very soon cause a significant overhead, especially when doing this for all threads one-by-one, as GDB would when implementing all-stop on top of non-stop.

Asynchronously stopping all threads on a device, on the other hand, can be done efficiently with very little delay between threads. This allows for an efficient implementation of pure all-stop on Intel GT devices.

We implemented pure all-stop and are currently looking into extending our implementation to support non-stop, as well.

4 A SAMPLE DEBUG SESSION

In this section we present a sample debugging session[5] on the SYCL program shown in Figure 3. This is a complete SYCL program that can be compiled and run without any modifications. The program picks the default SYCL device, and submits a kernel that uses a `parallel_for` to process the elements of an input array and write into an output array. We are assuming that the default SYCL device on the machine is an Intel GT device; so, the kernel will be offloaded to that GPU. We are also assuming that the build flow explained in Section 3 has been followed. In particular, the compiler's debug information emission has been enabled and the optimizations have been disabled for both the host and the kernel portion of the code for a smooth debug experience. For host, this is achieved by passing the `-g -O0` flags to the compiler that comes with the oneAPI distribution. For the kernel, at the time of writing this paper, IGC turns on debug info and turns off optimizations in the presence of a debugger.

The debugging session can be started by passing the program executable as a command-line argument to the debugger — just like an ordinary debug session with GDB.

```
$ gdb-oneapi ./sample
GNU gdb (GDB) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.; (C) 2020 Intel Corp.
...
```

Next, we define breakpoints at the then-branch (line #17), else-branch (line #19), and at the end of the main function (line #35). Note that the former two breakpoints are inside the kernel while the latter is inside the host program.

```
(gdb) break 17
Breakpoint 1 at 0x405ad7: file sycl.cpp, line 17.
(gdb) break 19
Breakpoint 2 at 0x405ae7: file sycl.cpp, line 19.
(gdb) break 35
Breakpoint 3 at 0x4049aa: file sycl.cpp, line 35.
```

When we run the program, as stated in Section 3, the debugger automatically spawns a `gdbserver-gt` process to listen to debug events, and adds it to the debugging session as a new inferior. This is done via GDB's Python API. The user is given back the GDB prompt when the program execution hits a breakpoint:

```
(gdb) run
Starting program: /path/to/sample
...
<omitted output>
...
Thread 2.2:1 hit Breakpoint 2, compute(int*, int*)::... at sycl.cpp:19
19              point = -1;           // else-branch
```

The breakpoint at the else-branch is hit by Thread 2.2:1. (The else-branch is hit before the then-branch because of the instruction ordering chosen by IGC.) The thread id "2.2:1" means "inferior 2, thread 2, SIMD lane 1". That is, the stop event is received from a

[5] Parts of the debugger output have been omitted; these are denoted with ellipses. Some minor formatting was applied to fit the output to the space in this paper.

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

void compute(int input[], int output[]) {
  queue device_queue; // picks default device
  range<1> range{64};
  buffer<int, 1> buffer_in{input, range};
  buffer<int, 1> buffer_out{output, range};

  device_queue.submit([&](handler& cgh) {
    auto in = buffer_in.get_access<access::mode::read>(cgh);
    auto out = buffer_out.get_access<access::mode::write>(cgh);

    cgh.parallel_for<class kernel>(range, [=](id<1> index) {
      int point = in[index];
      if (index % 2 == 0)
        point = point + 1000; // then-branch
      else
        point = -1;           // else-branch
      out[index] = point;
    });
  });
}

int main() {
  int input[64];
  int output[64];

  // Initialize the input
  for (unsigned int i = 0; i < 64; i++)
    input[i] = i + 100;

  compute(input, output);

  return 0; // end of main
}
```

Figure 3: A sample SYCL program.

thread of inferior 2. This is the inferior that was created automatically by our debugger; it represents the kernel offloaded to the GT device. To see it, one can issue the `info inferiors` command. The '*' marker next to the inferior number denotes that it is the current inferior.

```
(gdb) info inferiors
  Num  Description    Connection
  1    process 32087  1 (native)
* 2    Remote target  2 (remote gdbserver-gt --attach - 32087)
```

Now that we are inside the kernel, the GT device registers can be inspected (e.g. via the `info registers all` command) or the GT code can be disassembled.

```
(gdb) x/i $pc
=> 0xfffe7b00 <...>: mov (8|M0) r23.0<1>:d -1:w
```

The threads running on a GT device are vectorized; i.e. they have SIMD lanes. The breakpoint hit was received from SIMD lane 1 of Thread 2.2. Let us display more information about Thread 2.2, which is our current thread.

```
(gdb) info thread 2.2
  Id          Target Id         Frame
* 2.2:1       Thread 1073741824 ... at sycl.cpp:19
  2.2:[3 5 7] Thread 1073741824 ... at sycl.cpp:19
```

The '*' marker denotes the current SIMD lane. The second row in the table above also lists lane ids inside square brackets, i.e. `[3 5 7]`. These are the lanes that are currently active. Recall that we stopped at the else-branch. Because of the condition in the code (i.e. `index % 2 == 0`), only the odd-indexed lanes are active. In the SIMD execution model, this is called branch/thread divergence [14].

Let us delete the breakpoint at the else-branch and resume the program. This time, we hit the then-branch breakpoint, where the even-indexed lanes are active.

```
(gdb) delete 2
(gdb) continue
Continuing.

Thread 2.2:0 hit Breakpoint 1, compute(int*, int*)::... at sycl.cpp:17
17              point = point + 1000; // then-branch
(gdb) info thread 2.2
  Id          Target Id         Frame
* 2.2:0       Thread 1073741824 ... at sycl.cpp:17
  2.2:[2 4 6] Thread 1073741824 ... at sycl.cpp:17
```

The kernel contains a local variable named `point`. Because the threads are SIMD, this local variable is vectorized. To inspect the values, let us print the local variable in the context of the current lane:

```
(gdb) print point
$1 = 100
```

We now switch the lane of the thread and print the variable again. This time we get its value from the perspective of the new lane.

```
(gdb) thread 2.2:4
[Switching to thread 2.2:4 (Thread 1073741824 lane 4)]
#0  compute(int*, int*)::$_0::... at sycl.cpp:17
17              point = point + 1000; // then-branch
(gdb) print point
$2 = 104
```

Alternatively, one can switch back to the first lane and inspect the chunk of memory where the variable `point` is located. Below we use the `x` command of GDB to examine the memory. The argument `/8dw` means "8 items, in decimal format, word-length each".

```
(gdb) thread 2.2:0
[Switching to thread 2.2:0 (Thread 1073741824 lane 0)]
#0  compute(int*, int*)::... at sycl.cpp:17
17              point = point + 1000; // then-branch
(gdb) x /8dw &point
0x5eb9160:      100     -1      102     -1
0x5eb9170:      104     -1      106     -1
```

Note that the `point` values at odd-indexed locations have already been set to -1 because the else-branch has already been executed.

At this point, one can do other standard debugger activities such as instruction-level or source-level stepping, back-tracing, etc. Let us delete the then-branch breakpoint and resume the program. We hit the breakpoint at the end of the main function.

```
(gdb) delete 1
(gdb) continue
Continuing.
[Switching to Thread 0x7ffff7fd9780 (LWP 32087)]

Thread 1.1 "sample" hit Breakpoint 3, main () at sycl.cpp:35
35        return 0; // end of main
```

GDB seamlessly switches to inferior 1 — the inferior that represents the host computation (see the output of `info inferiors` below). At this point, disassembling prints x86_64, not Intel GT, instructions.

```
(gdb) info inferiors
  Num  Description    Connection
* 1    process 32087  1 (native)
  2    Remote target  2 (remote gdbserver-gt --attach - 32087)
(gdb) x/i $pc
=> 0x4049aa <main()+106>:       add    $0x220,%rsp
```

Again, standard debugger actions can be performed, such as stepping, back-tracing, or inspecting the memory, but this time at the CPU side. Below we print host data contained in the `input` and `output` variables to illustrate this.

```
(gdb) print input
$3 = {100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110,
      ..., 162, 163}
(gdb) print output
$4 = {1100, -1, 1102, -1, 1104, -1, 1106, -1, 1108, -1, 1110,
      ..., 1162, -1}
```

5 CHALLENGES

We faced several challenges during the development process of the debugger. In this section we present each of these problems and the solution approach we took or are evaluating to apply.

The first two of the problems, namely, implicit pass-by-reference arguments and C++ functions with template parameters, are debugging difficulties caused by the C++ language and are made particularly visible by SYCL's use of C++. We were able to address these two problems by enhancing GDB and by utilizing its Xmethods mechanism for replacing function calls.

The latter three problems we discuss, namely, thread view with SIMD lanes, modeling device threads, and efficient implementation of conditional breakpoints, are related to the GPU device. We were able to overcome the thread view challenge by enhancing GDB. For the other two challenges, we are still evaluating potential solutions.

5.1 Implicit Pass-By-Reference Arguments

Several key SYCL classes such as `cl::sycl::id`, `cl::sycl::range`, and `cl::sycl::item`, follow a common by-value semantics defined by the SYCL Specification [12, §4.3.3]. SYCL API functions or user-defined functions may have pass-by-value parameters with one of the above-mentioned SYCL classes as its type. For instance, the `cl::sycl::accessor` class defines the methods below:

```
dataT &operator[](id<dimensions> index) const;
dataT operator[](id<dimensions> index) const;
```

Via this overloaded operator, data elements can be accessed conveniently as if accessing elements of an ordinary array. See, for example, lines #15 and #20 in Figure 3. An expression such as `in[index]` means that the `operator[]` method of the `in` object shall be invoked by passing to the method the `index` value as the argument. According to the C++ ABI, a pass-by-value parameter is implicitly pass-by-reference if its type is not trivially copyable [3, §3.1]. This essentially means that the compiler is supposed to figure out whether a type is implicitly pass-by-value or not. However, for expression evaluation during a debug session, finding out this information and following the ABI is the debugger's responsibility.

Suppose that a user is in the middle of a debugging session where they stopped inside the SYCL kernel; for instance, at line #17 in Figure 3. At this point, the user may want to print data elements to inspect the current state, e.g. by issuing the following command:

```
(gdb) print in[index]
```

Here, the debugger is being asked to evaluate `in[index]` and print the resulting value. This requires the debugger to correctly setup the program state and initiate a call of the `operator[]` method by passing it the `index` object as the argument. For this, the debugger must utilize the available debug information and deduce whether `index` shall be passed implicitly by reference.

When developing the debugger, we found out that GDB (and LLDB [18]) has bugs in making C++ function calls that have pass-by-value parameters. Although this is a general C++ topic, the bug was revealed particularly because such function calls are common in SYCL. We reported the bug[6] to GDB, and implemented patches to improve and fix its function call mechanism. The patches have been accepted for inclusion in the GDB code-base[7]. We have requested an addendum to the OpenCL Debug Information specification to include `FlagTypePassByValue` and `FlagTypePassByReference` in the debug info flags. This was needed for propagation of debug info from the front-end all the way to the jit-compiled kernel code. Our request was accepted and the flags were included in OpenCL Debug Information, Version 2 [17].

Calling kernel function from inside the debugger is not supported for Intel GT devices at the moment. Therefore, the problem we described applies to SYCL programs running on HOST or offloading to a CPU device.

5.2 C++ Functions with Template Parameters

The SYCL specification as well as Intel's implementation of that spec rely heavily on templated C++ classes and functions. C++ templates are instantiated by the compiler based on compile-time information. Much of that compile-time information is not available to the debugger and is not reconstructible from the debug information emitted by the compiler. This poses a problem if the debugger user wants to evaluate an expression that refers to a template instance. The symbol for that instance may not be resolvable because, for instance,

- the user referred to a template instance that did not exist in the source code, and thus, was not instantiated by the compiler.
- the debugger does not have sufficient context to correctly pick a symbol from a set of candidates (e.g. to do function overload resolution).
- the compiler omitted the code for a symbol due to inlining or because a function was not being used in the source code.

To give an example, suppose the user stopped at line #17 of the code in Figure 3, and wanted to print `index + 5`. The `index` variable is of type `cl::sycl::id<1>`, which has an overloaded `operator+`. However, the compiler does not emit code for it, because that operator is not used in the source. Hence, the debugger cannot evaluate the expression and results in degraded debug experience.

```
(gdb) print index + 5
Could not find operator+.
```

This is a well-known problem seen also in the case of C++ STL containers. GDB offers a mitigation via the so-called Xmethods in its

[6] https://sourceware.org/bugzilla/show_bug.cgi?id=25054
[7] The top commit hash of the patch series in GDB's git repository is c855a9125a.

Python API[8]. An Xmethod is a replacement for a function/method whose signature and the type of the receiver is known. Xmethods are defined in Python. If an Xmethod definition for a function exists, GDB invokes that instead of attempting to resolve and call the function in the debuggee's binary file. GCC provides pretty-printers and Xmethods for many STL containers[9]. We preferred the same approach and wrote Xmethod definitions for template SYCL methods that we deem critical for an improved debug experience. In particular, we provide Xmethods for the `operator[]` of the `accessor` class, which is a template function in Intel's SYCL implementation.

Isemann addresses this problem in the context of LLDB by leveraging the upcoming C++20 feature, *modules* [11]. LLDB uses Clang to parse expressions for evaluation. When Clang does not have sufficient information to parse a subexpression, LLDB takes the role of an external AST source and helps Clang continue parsing with proper context. For this, Isemann combines the information obtained from DWARF together with C++ modules, and uses this combined knowledge to instantiate templates and jit-compile the instance via LLVM. Although this approach still suffers from the lack of information about full compile-time environment, user experience can be greatly improved by providing a solution that applies to a majority of use cases. GDB supports a `compile` command[10] that compiles a given expression using `libcc1.so` and thus can be used to implement a similar design. However, we are not aware of any plans or attempts to pursue this approach in GDB further.

The Xmethod approach has a particular advantage for us. On Intel GT devices, inferior function calls are currently not supported. That is, the debugger cannot evaluate an expression when debugging a GPU kernel, if the expression involves a function call, regardless of being templated. However, because an Xmethod is a replacement for a function, from the user's perspective, it looks as if a function is being invoked. Hence, expressions that would otherwise fail can be evaluated, if there exist Xmethod definitions. This would not be possible with the C++ modules approach.

5.3 Thread View with SIMD Lanes

Intel GT devices expose parallelism in two dimensions: threads and data elements. First, each Intel GT device is equipped with several execution units (EU) that can execute multiple hardware threads concurrently. Second, each Intel GT instruction can process several data elements simultaneously. That is, they utilize the Single Instruction Multiple Data (SIMD) model.

SYCL kernels are written with a focus on computing a single data element, called a *work item*. A collection of work items (i.e. a *work group*) is computed on a single compute unit. Due to its SIMD nature, an Intel GT thread processes several work items at once. When it hits a breakpoint, the user is notified about the event and the program state can be inspected. If standard GDB were being used, only the first data element that is being processed by the stopped thread would be available to the user for inspecting conveniently. To examine other SIMD channels, the user would have to use advanced commands (e.g. see the use of `x/8wd &point` in Section 4). Also, a conditional breakpoint would not be hit, if the condition is set for a particular work item that is processed by a SIMD lane other than the first one.

To address these problems, we extended standard GDB's thread-related commands (e.g., `thread`, `info thread`, `thread apply`), as well as its breakpoint implementation, to take SIMD lanes into account. In particular, we added a current lane field to GDB's thread representation to make the evaluation context SIMD-aware.

The SIMD parallelization we refer to in this section is an implicit vectorization applied by the compiler. Note that SYCL also allows explicit, i.e. user-specified, vectorization via the `vec` class [12, §4.10.2]. In this case, explicit vectorization is applied on top of implicit vectorization. That is, each SIMD lane of a thread operates on vectorized data. Given that the compiler emits proper debug information for vector types, GDB is already able to present and enable inspection of explicit vectorization.

In the following, we describe the key properties of the SIMD model on Intel GT devices that have influenced our extension. We refer to the number of SIMD computation channels of an Intel GT instruction as the *execution size* [7]. For each instruction, its execution size can be 1, 2, 4, 8, 16, or 32. Each instruction includes a 32-bit execution mask that defines the enabled SIMD channels. SIMD channels may be disabled not only due to the execution size of the instruction, but also due to conditional control flow or the size of the underlying problem, when the number of data elements is not a multiple of the execution size of the instruction[11]. A kernel offloaded to an Intel GT device can be dispatched with either 8, 16, or 32 SIMD width [2]. This means that all instructions inside the kernel have their execution size smaller than or equal to the chosen SIMD width. In order to choose the optimal SIMD width for a kernel, IGC applies heuristics [2]. Currently, in the presence of the debugger, kernels are always dispatched with SIMD width 8.

We defined the requirements for our implementation as follows:

- If the target architecture does not support SIMD, or the thread that is currently in focus does not have SIMD lanes (i.e. its SIMD width is trivially 1), GDB's existing behavior remains unchanged for this thread. If the new SIMD syntax is not used and a thread has SIMD lanes, GDB's behavior remains mostly unchanged, except some output messages.
- A thread's SIMD width is not fixed. A hardware thread may switch between kernels dispatched with different SIMD widths. Even within the same kernel, execution size of some instructions might be smaller than the SIMD width of the kernel. This flexibility should be allowed.
- GDB shall display the SIMD lanes that are enabled by the execution mask only. Further, there is no distinction between a disabled SIMD lane (e.g. inside a branch) and a non-existent SIMD lane (e.g. lane 15 when the kernel was dispatched with SIMD width 8). Threads with all-zero execution mask are marked as inactive. We are considering this as an option for modeling unavailable threads (see Section 5.4).

[8] https://sourceware.org/gdb/onlinedocs/gdb/Xmethods-In-Python.html#Xmethods-In-Python

[9] https://gcc.gnu.org/svn/gcc/branches/gcc-9-branch/libstdc++-v3/python/libstdcxx/v6/

[10] https://sourceware.org/gdb/current/onlinedocs/gdb/Compiling-and-Injecting-Code.html

[11] The hardware disables some SIMD channels on the last dispatched thread in this case.

- After a stop event for a thread with SIMD lanes is received, GDB focuses on an enabled SIMD lane. The user can switch between enabled SIMD lanes.
- Mover commands (e.g., `step`, `next`, etc.) are applied to the underlying hardware thread.

Currently, our GDB extension for SIMD is only available for kernels offloaded to Intel GT devices. The extension relies heavily on the availability of the execution mask when a thread is stopped. Additionally, the compiler must generate debug information that describes SIMD. Currently, IGC does this by emitting a specific `DW_AT_description` field to the debug info entry of each variable that specifies the SIMD width of the vectorized version.

A variable's location attribute describes the location of lane zero; the debugger uses the specified SIMD width to infer the location for other SIMD lanes. This requires the debugger to understand the layout of vectorized objects. It works well for non-optimized code, but it does not allow the result of some transformation to be described. Consider, for example, a gather operation that loads one field out of a vector of structures into a register. We are currently looking into describing location expressions as functions of the SIMD lane, which would allow us to describe arbitrary layouts.

5.4 Modeling Device Threads

Creating a CPU thread involves a system call and some maintenance work to be done by the OS. This already introduces enough overhead that adding a notification to an attached debugger for the new thread does not introduce much intrusion into the execution of the debuggee process.

CPU threads created by one process further remain with that process until the thread is destroyed again using another system call and resulting in another notification to an attached debugger. Debuggers use those two notifications to maintain a precise model of threads in each debuggee process.

The situation is different on the GPU. For Intel GT devices, the hardware dispatches available GPU threads to active kernels. From the point of view of a kernel, the set of threads can change rapidly. From the point of view of a thread, it may be switching kernels rapidly or it may be switching between kernels and idle. The potentially high frequency at which threads change their assignments poses a challenge on how to model device threads in the debugger.

The model we implemented uses kernel entry breakpoints to synthesize thread creation events. We had initially also used kernel exit breakpoints to synthesize thread termination events. This has allowed us to model device threads similar to threads on the CPU. However, it has also caused very high intrusion, so we decided to omit exit breakpoints. To further reduce intrusion, we omitted kernel entry breakpoints, as well. The debugger only knows about threads that reported an event; a breakpoint hit in most cases. It does not know whether those threads are still dispatched to this kernel once the debugger resumes a thread.

We cannot omit kernel entry breakpoints completely, since we are also using them to detect kernel submissions. We need to stop a new kernel in order to place breakpoints in the kernel code.

To minimize, if not completely avoid, intrusion, we are currently looking into a different model. Traditionally, threads can either be stopped or running. We extend the thread state to allow threads to be unavailable, as well. An unavailable thread cannot be interacted with, similar to a running thread. Unlike a running thread, however, it cannot be stopped, either. We use this to model threads that are currently not dispatched to the kernel we are debugging.

Once a thread is dispatched to our kernel, it may stop either via an interrupt request from the debugger or by hitting a breakpoint. Its state changes to stopped. When the thread is resumed, its state changes to running. We do not know whether the thread is still running or has become unavailable until we try to stop it. At that time, its state transitions to stopped or unavailable.

This new model requires further changes to how the debugger is notified about new kernels and how we can guarantee that breakpoints are in place before threads are dispatched to execute those kernels.

5.5 Conditional Breakpoints

We expect users to be interested in inspecting their kernels for a single or a small set of work items. To do this, the conditional breakpoint mechanism of GDB can be used: A hit event of a conditional breakpoint is reported to the user only if the condition holds. Under the hood of the debugger's user interface, however, the following happens: Once a device thread stops, the notification of this event goes to gdbserver, which reports all stop events back to GDB. Then, GDB evaluates the condition of the breakpoint that has triggered the event. If the condition holds, GDB displays the event to the user; otherwise, the hit is ignored and the thread is resumed. This means that even if the condition is meant to hold for a single thread, potentially a large number of threads may hit a breakpoint and stop, only to be resumed by the debugger. This evaluation scheme may cause considerable delays when debugging kernels offloaded to the device, negatively impacting scalability of the debugger. We can mitigate this problem by moving the evaluation of the condition closer to the device. We consider the following options:

- We move the evaluation to gdbserver, so GDB is notified only about breakpoint hits for which the condition is true. This way, the communication between gdbserver and GDB can be avoided.
  To keep gdbserver relatively simple, this would require compiling the condition into a form that can easily be evaluated by gdbserver.
- We generate device code for evaluating the condition and inject it into the kernel. Only threads for which the condition is true hit the breakpoint.
- For a specific class of conditions, we can pre-generate parameterized code to evaluate the condition in the system routine. The debugger can instantiate the parameters and enable the evaluation. Work item-specific breakpoints could be implemented this way, for example. Threads for which the condition is false, return from the system routine. Only threads for which the condition is true are reported to gdbserver.
  While this approach is not as flexible as the above, it may be easier to implement.

All these approaches attempt to reduce the amount of handshaking between the debug interface, gdbserver, and GDB.

6 RELATED WORK

NVIDIA provides a GDB-based debugger for their CUDA programming language called CUDA-GDB [8]. Similar to our debugger, it can debug code executing on a GPU device. It enhances GNU GDB with an extensive set of debugging features for working with NVIDIA hardware to support inspecting GPU thread state, stepping though CUDA kernels, and analysing crashed CUDA kernel core files. This is implemented partially by modifying existing GDB commands (for example, `break`) and partially by introducing new command families prefixed with `cuda` (for example, `info cuda thread`).

Below is the output of a simple debugging session for one of the samples packaged with the CUDA distribution (vectorAdd). The program stops inside a kernel at a breakpoint:

```
Thread 1 "vectorAdd" hit Breakpoint 1, ... at vectorAdd.cu:39
39        C[i] = A[i] + B[i];
```

To print threads of the host application, we use GDB's 'info threads' command. Note that since the breakpoint was hit on the device, none of the threads is marked active.

```
Thread 1 "vectorAdd" hit Breakpoint 1, ... at vectorAdd.cu:39
39        C[i] = A[i] + B[i];
(cuda-gdb) info threads
  Id   Target Id         Frame
  1    Thread ... in cuVDPAUCtxCreate ()
   from /usr/lib/x86_64-linux-gnu/libcuda.so.1
  2    Thread ... in accept4 ()
   from /lib/x86_64-linux-gnu/libc.so.6
  3    Thread ... in poll ()
   from /lib/x86_64-linux-gnu/libc.so.6
  4    Thread ... in pthread_cond_timedwait@@GLIBC_2.3.2 ()
   from /lib/x86_64-linux-gnu/libpthread.so.0
```

To print device threads, we use the corresponding 'info cuda threads' command.

```
(cuda-gdb) info cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx   Count
                 Virtual PC     Filename   Line
Kernel 0
*  (0,0,0)   (0,0,0)    (11,0,0) (127,0,0)    2944
        0x0000000000b65df8 vectorAdd.cu     39
  (11,0,0) (128,0,0)    (11,0,0) (159,0,0)      32
        0x0000000000b65df0 vectorAdd.cu     37
  (11,0,0) (160,0,0)    (12,0,0)  (95,0,0)     192
        0x0000000000b65df8 vectorAdd.cu     39
  (12,0,0)  (96,0,0)    (12,0,0) (127,0,0)      32
        0x0000000000b65df0 vectorAdd.cu     37
... (more lines omitted)
```

Such CUDA-specific commands provide detailed information about GPU-specific concepts (devices, blocks, warps, threads etc.) but that information is generally hidden from standard GDB commands. Note how in the breakpoint example GDB reports that it was hit by "Thread 1" because it does not see GPU threads as actual threads.

One of our design goals is to blend the device interaction with the existing GDB commands and concepts as much as possible to provide a seamless integration of host application and offloaded kernels in a single debug session.

Difference in abstraction models becomes more visible with mover commands (i.e. run-control commands that make threads step or resume execution). In CUDA-GDB, single-stepping a device kernel always advances all active threads in the warp[12] currently in focus (and never others). GDB's user-controllable scheduling settings such as `scheduler-locking` and `schedule-multiple` are not used. In contrast, we expose GPU threads in GDB in the very same way that host threads are exposed. For this reason, scheduling settings and existing commands of GDB do have the expected impact on the GPU threads in our debugger. We believe this approach fits well to the SYCL philosophy that device switching is mostly seamless from a developer point of view, whereas CUDA prefers an approach that makes GPU a completely distinct case with its own set of commands and rules.

[12] A "thread" in CUDA terminology corresponds to a "SIMD lane" in Intel GT; a "warp" corresponds to a "thread".

7 CONCLUSIONS AND FUTURE WORK

We developed a debugger as part of Intel's oneAPI initiative. The debugger is based on GDB and can be used to debug SYCL programs. We presented the general architecture of the debugger that utilizes the multi-target feature of GDB, so that the host and kernel part of the program can be debugged seamlessly in the same session, even when the kernel is running on an Intel GT device. To this aim, we added Intel GT target-specific definitions to GDB and addressed a number of challenges. In particular, we enhanced GDB with SIMD lane management, used GDB's Xmethods mechanism as a replacement for calling template methods of SYCL, and improved GDB's ability to call C++ functions with pass-by-value parameters. For our enhancements and additions, we aimed to avoid deviating from existing GDB mechanisms so that users can leverage their existing experience when using our debugger.

The work presented here is at the beta phase and is on-going. For the future, we anticipate challenges related to scalability due to a high number of threads that exist on GPUs. We discussed device thread modeling and handling of conditional breakpoints as two of these potential challenges.

Although we mostly focused on the Linux use case with the SYCL kernel running on an Intel GT device, the debugger also supports the Windows operating system, and also the programs that offload to a CPU or FPGA software emulator as their target SYCL device. The debugger is available publicly as part of the Intel oneAPI Base Toolkit at http://software.intel.com/oneapi.

ACKNOWLEDGMENTS

Several past and present members of Intel's Application Debugger team contributed to various parts of the debugger presented here. We thank Fabian Schnell and the anonymous reviewers for their feedback on an earlier version of this paper. CUDA, GCC, GDB, Khronos, Linux, LLDB, NVIDIA, OpenCL, Python, SYCL, SPIR-V, Windows, and other names and brands may be claimed as the property of others.

REFERENCES

[1] Pedro Alves. 2019. *Multi-target GDB.* https://github.com/palves/gdb/tree/palves/multi-target

[2] Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, and et al. 2019. IGC: The Open Source Intel Graphics Compiler. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. 254–265.

[3] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. 2017. *Itanium C++ ABI.* https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[4] DWARF Debugging Information Format Committee. 2017. *DWARF Debugging Information Format.* http://www.dwarfstd.org

[5] Tool Interface Standard Committee. 1995. *Executable and Linking Format Specification.* https://refspecs.linuxbase.org/elf/elf.pdf

[6] Intel Corporation. [n.d.]. *oneAPI Programming Model.* http://oneapi.com

[7] Intel Corporation. 2016. Intel(R) Open Source HD Graphics, Intel Iris(TM) Graphics, and Intel Iris(TM) Pro Graphics. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-skl-vol07-3d_media_gpgpu.pdf Volume 7: 3D-Media-GPGPU.

[8] NVIDIA Corporation. [n.d.]. *CUDA-GDB: The NVIDIA CUDA debugger for Linux and QNX targets.* https://docs.nvidia.com/cuda/cuda-gdb

[9] The GDB developer community. [n.d.]. *GDB: The GNU Project Debugger.* https://www.gnu.org/software/gdb/

[10] Intel. [n.d.]. *oneAPI Level 0 Specification.* https://spec.oneapi.com/oneL0/index.html

[11] Raphael Isemann. 2019. *Beyond debug information: Improving program reconstruction in LLDB using C++ modules.* Master's thesis. Chalmers University of Technology.

[12] Ronan Keryell, Maria Rovatsou, and Lee Howes. 2019. *SYCL Specification, version 1.2.1.* Khronos OpenCL Working Group – SYCL subgroup. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[13] John Kessenich, Boaz Ouriel, and Raun Krisch. 2019. *SPIR-V Specification.* https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf

[14] David B. Kirk and Wen-mei W. Hwu. 2016. *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[15] James Price and Simon McIntosh-Smith. 2015. Oclgrind: An Extensible OpenCL Device Simulator. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15).* Article 12, 7 pages.

[16] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. 2008. Non-stop Multi-Threaded Debugging in GDB. In *Proceedings of the GCC Developers' Summit.* 117–128.

[17] Alexey Sotkin. 2019. *OpenCL.DebugInfo.100 Information Extended Instruction Set Specification.* https://www.khronos.org/registry/spir-v/specs/unified1/OpenCL.DebugInfo.100.pdf

[18] The LLDB Team. [n.d.]. *The LLDB Debugger.* http://lldb.llvm.org/