Hindawi Publishing Corporation
Advances in Sojware Engineering
Volume 2015, Article ID 940628, 14 pages
http://dx.doi.org/10.1155/2015/940628

Research Article

LTTng CLUST: A System-Wide Unified CPU and GPU Tracing Tool for OpenCL Applications

David Couturier and Michel R. Dagenais

Department of Computer and Software Engineering, Polytechnique Montreal, P.O. Box 6079, Station Downtown, Montreal, QC, Canada H3C 3A7

Correspondence should be addressed to David Couturier; david.couturier@polymtl.ca

Received 14 April 2015; Accepted 1 July 2015

Academic Editor: Moreno Marzolla

Copyright © 2015 D. Couturier and M. R. Dagenais. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

As computation schemes evolve and many new tools become available to programmers to enhance the performance of their applications, many programmers started to look towards highly parallel platforms such as *Graphical Processing Unit* (GPU). Offload g computations that can take advantage of the architecture of the GPU is a technique that has proven fruitful in recent years. This technology enhances the speed and responsiveness of applications. Also, as a side effect, it reduces the power requirements for those applications and therefore extends portable devices battery life and helps computing clusters to run more power effici tly. Many performance analysis tools such as `LTTng`, `strace` and `SystemTap` already allow *Central Processing Unit* (CPU) tracing and help programmers to use CPU resources more effici tly. On the GPU side, different tools such as `Nvidia`'s `Nsight`, `AMD`'s `CodeXL`, and third party `TAU` and `VampirTrace` allow tracing *Application Programming Interface* (API) calls and OpenCL kernel execution. The e tools are useful but are completely separate, and none of them allow a unifie CPU-GPU tracing experience. We propose an extension to the existing scalable and highly effici t `LTTng` tracing platform to allow unifie tracing of GPU along with CPU's full tracing capabilities.

1. Introduction

Tracing programs has been a common technique from the beginning. Tracers such as DTrace [1], strace [2], SystemTap [3], and LTTng [4] have been around for many years. Not only do they allow recording kernel trace events but some also allow recording user space events. This is a very good alternative to debuggers for finding software bugs and especially performance related issues that concern the execution on the CPU side. Th rapid changes in computation never stop to bring new challenges. As part of those rapid changes, we have seen GPU acceleration become a more common practice in computing. Indeed, the different highly parallel architectures of the graphic accelerators, compared to the conventional sequential oriented CPU, represent a very attractive tool for most of the graphical work that can be achieved on a computer. This became the standard in recent years. Most operating systems have even added a graphic accelerator as a hardware requirement. Indeed, its processing power is being harnessed in many of the operating system's embedded tools and helps make computations more fluid, therefore enhancing user experience. Th evolution of classical graphical acceleration in games and user interfaces brought architecture changes to the GPU. The graphical accelerators evolved from the graphical pipeline to a more programmable architecture. Programmers quickly figured how to offload raw computations to the graphical accelerator and GPU computing, also referred as *General Purpose Graphical Processing Unit* (GPGPU), was born. APIs such as Nvidia (http://www.nvidia.com/), *Compute Unified Device Architecture* (CUDA) (https://developer.nvidia.com/cuda-zone/), and Apple's open standard *Open Computing Language* (OpenCL) [5] (developed by the Khronos Group) were created and adapted to standardize the usage of GPUs as GPGPUs. As part of this change, many tools were implemented to help programmers better understand how the new architecture handles the workloads on the GPGPU. Th two major manufacturers, *Advanced Micro Devices*

(AMD) (http://www.amd.com/en-us/products/graphics/) and Nvidia provided their analysis, tracing and debugging tools: CodeXL (http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/) and Nsight (http://www.nvidia.com/object/nsight.html), respectively.

Unfortunately, none of the GPGPU tracing tools are currently capable of seamlessly providing both a comprehensive GPU trace alongside with a fully capable CPU oriented trace that tracers such as DTrace, strace, SystemTap, or LTTng already provide. Also, current tools do not allow for a system-wide tracing of all processes using the GPU: current techniques require the user to manually specify what processes to trace.

In this paper, we present a solution to acquire a system-wide trace of OpenCL GPGPU kernel executions for a better understanding of the increasingly parallel architecture that GPUs provide. Thi solution is also presented as an extension of the already existing CPU tracer LTTng to allow for unifi d CPU-GPU trace gathering and analysis capabilities.

In addition, we take a look at kernel trace point analysis for the open source Intel i915 drivers that support OpenCL, and for which an open source library for OpenCL, called *Beignet* (https://wiki.freedesktop.org/www/Software/Beignet/), is provided.

Thi paper is divided in four main parts. First, in Section 2, we explore the CPU and GPU tracing options and methods that are currently available. Then, in Section 3, the architecture, implementation choices and algorithms are explained, followed by a performance analysis of the solution proposed in Section 4. Finally, we conclude and outline possible future work in Section 5.

2. Related Work

Tracing is the act of recording events during the execution of a program at run-time. This is commonly used with not only *GNU's Not Unix* (GNU)/Linux [3, 4, 6] but also all major operating systems such as Unix [1] and Windows (https://msdn.microsoft.com/en-us/library/windows/desktop/bb96-8803.aspx). Th main purpose is to provide a tool for the developers to record some logs about the execution of programs and detect complex problems such as race conditions.

In this section, we fi st present the different available tracing platforms that are available. Secondly, we discuss what kind of information trace analysis provides, followed by current GPU tracing architectures, tracing tools, and OpenCL tracing capabilities. Then the topic of time keeping is discussed: many different timing metrics are available and they need to be explored. Moreover, the device timing metrics that can be accessed from the OpenCL library on the GPU may not be using the same source as the host: synchronization methods to help match those two separate sources will be discussed (Section 2.4). Finally, we talk about tracing in the device drivers as a complementary measure of OpenCL tracing.

2.1. CPU Tracing. Tracing on GNU/Linux is somewhat easier than doing it on Windows or other proprietary operating systems. One good reason for that is because we have access to its source code. Since our work focuses on the GNU/Linux operating system, we will direct our attention to this environment and explore available tools on this platform. The major tools identifi d are strace [2], DTrace [1], SystemTap [3], and LTTng [4].

2.1.1. Strace. Th main benefit of strace [2] is its ease of use and the fact that it has been around since the early 90s. Th strace functionality allows the user to record all or some specifi system calls of a given program or a list of processes. Th output is either sent to *stderr* or redirected to a file. The content of the output is the name of the recorded system call with its parameters and the return value of the call. For instance, recording only the file open and close operations of a program can be done by executing the following command: `strace -e open, close./myProgram`.

Despite the fact that strace is easy to use, its most obvious limitation lies in the fact that it uses a single output file for trace recording and therefore degrades performance when tracing multithreaded applications since it has to use *mutexes* to write from the multiple threads to the single file. Thi issue is discussed in [7].

Strace also has a hidden performance weakness: when tracing an application, the API used to get the system call information (*ptrace*) adds an overhead of two system calls per system call that the traced program performs [8]. Thi can have a dramatic impact on the system due to the known overhead of system calls. The efore, the strace approach does not seem to be suitable in the long run.

2.1.2. DTrace. DTrace [1] originated from the Solaris operating system. Solaris is a Unix variant developed by Sun Microsystems which was later bought by Oracle. Aft r the acquisition of Sun Microsystems, Oracle ported the tracing tool to the GNU/Linux *Operating System* (OS) [9]. DTrace relies on hooking scripts to probes that are placed in the operating system kernel. Tho e scripts can then process and record system calls or other information. A script is compiled into *bytecode* in order to allow fast execution through a small and effici t kernel based virtual machine. DTrace also provides a library called *User Statically Defined Tracing* (USDT) that programmers can use to instrument the code of their applications and perform *user space* tracing [1].

One major issue with DTrace is its usage of synchronization mechanisms that produce a large tracing overhead when tracing heavy workloads such as multithreaded applications [7]. Since programs nowadays are mostly multithreaded, this is a serious limitation.

2.1.3. SystemTap. Like DTrace, SystemTap [3] provides a framework for hooking scripts to probes in the kernel. It also offers user space tracepoint capabilities and may qualify as a system-wide tracing tool. The main difference with DTrace is that it originated on GNU/Linux and the scripts are compiled to native code instead of bytecode.

While SystemTap allows for user space tracing, a feature needed for OpenCL tracing, the compiled tracepoint script

is executed in kernel space. Th s forces the system to go back and forth from user space to kernel space in order to execute the script. Thi adds signific ntly to the cost of executing tracepoints and leads to higher overhead.

Again, like DTrace, SystemTap suff rs considerably from the synchronization mechanisms used: creating a large overhead when tracing heavy applications [7]. For the same reason as DTrace, heavily multithreaded applications will suff r from a very large overhead with SystemTap.

2.1.4. LTTng. Kernel space tracing for LTTng [4] is also based on hooking to *kprobes* and trace points. The approach to get the information is different from DTrace and SystemTap: modules are written in native *C* and compiled to executable code. LTTng was designed with the idea of tracing multithreaded applications with low overhead: each CPU thread has separate output buffers in order to minimize the need for synchronization methods when writing the trace events. As they are generated, before being written to disk, the trace events are stored in a circular buff r that has a specified capacity. In the worst case, if the event data throughput exceeds the disk throughput, events are intentionally discarded, rather than blocking the traced application, in order to minimize the tracing overhead.

User space event tracing is also provided with the *LTTng-UST* library and, unlike SystemTap, the user space tracepoint is recorded in a different channel entirely in user space. This allows for minimal tracing overhead [10]. As it will be discussed in Section 2.3.3, tracing OpenCL can be achieved at the user space level. Th advantages of LTTng over the other available options justifie its choice as a tracing platform for this project.

2.2. Trace Analysis. Getting the trace information is only one part of the tracing procedure. The encoding used to record traces and the potentially huge size of traces impose strict requirements on the analysis tools. For instance, LTTng provides viewing tools such as *babeltrace* (https://www.efficios.com/babeltrace/) and *Trace Compass* (http://projects.eclipse.org/projects/tools.tracecompass/) that allow decoding binary traces to human readable format (*babeltrace*) and interactive visualization and navigation (*Trace Compass*).

Visualization tools provide a very good understanding of program execution and allow for quick overall understanding of potential execution bottlenecks and synchronization issues. More advanced analysis can also be performed on traces, for example, pattern searching algorithms for uncovering specifi execution patterns within traces [11, 12], model checking [13], and statistics computation [14, 15].

2.3. GPU Tracing. Research teams around the world have spent a great amount of time developing applications to enable programmers to have a better understanding of the GPU kernel execution [16, 17]. It is thus relevant to examine GPU architectures. GPUs started as specialized hardware designed to deal with the display monitor. It first evolved into a graphical pipeline of operations and then towards the more programable highly parallel architecture that we have today [18].

Many programming APIs were implemented to support GPGPU but Nvidia's CUDA and the open standard API OpenCL (supported on several different hardware architectures) were the result of need for high level API [18]. Both APIs provide functions to harness the computing power of graphics cards.

Th GPU highly parallel architecture allows not only faster but also more energy effici t operations. This characteristic leads to longer battery life for portable devices and better performance per watt (http://top500.org/list/2014/11/).

2.3.1. GPU Architecture. Th reason behind the efficie y of GPUs compared to CPUs in highly parallel situations is that the GPU offers hundreds and even thousands of smaller computation cores built under the *Single Instruction Multiple Data* (SIMD) architecture. Not all computations can take advantage of such architecture. Thi is why only parts of programs can be run on GPUs. The SIMD architecture allows multiple different data elements to be processed simultaneously under the same computing instruction [18].

Diff rent form factors are available: integrated graphics are very popular in the portable computing world but dedicated graphics cards still hold a signific nt computing power advantage. AMD and Nvidia both off r integrated graphics and dedicated graphics devices, while Intel only offers integrated graphics. All architectures differ a lot from one companie to another and also between models. The performance may vary depending on the application: some architectures are better suited for some kind of computations and vice versa.

Computing on GPUs is a technique that derived from a workaround on the existing capabilities of available hardware. Th library that makes OpenCL work on GNU/Linux is a library that positions itself between the program and the drivers but still operates in user space. Figure 1 shows the position of Intel's open source OpenCL library for GNU/Linux called *Beignet*. Computing on the GPU with OpenCL is done asynchronously: commands are sent to the GPU driver command queue and when the device is ready to receive commands, they are transferred from the host driver command queue to the device command queue. Th driver is notified when the commands are done executing.

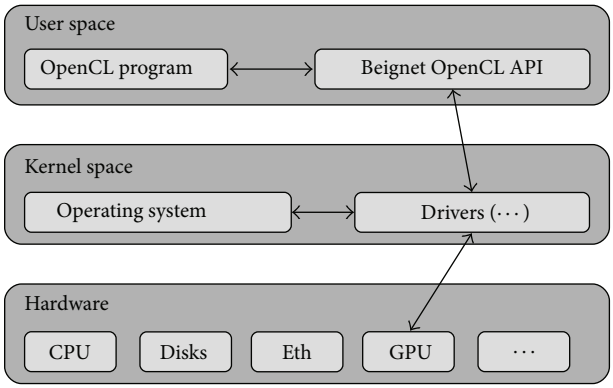
2.3.2. Available Tools. As discussed in [18], tracing tools for GPUs are primarily proprietary tools provided by the manufacturers:

(1) AMD: CodeXL,

(2) Nvidia: Nsight,

(3) Intel: VTune.

Thi d party tools such as *TAU* (https://www.cs.uoregon.edu/research/tau/home.php) and *VampirTrace* (http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace/) provide much of the same information as the proprietary ones but also add support for other computing APIs such as *Message Passing Interface* (MPI).



Figur e 1: Beignet OpenCL library position in the system.

Th information that these tools can provide is the arguments and timing of the API calls to the OpenCL library, as well as some instrumentation that [19] presents in order to get timing for the execution of the events on the graphics card, using profiling tools that OpenCL can provide. CodeXL, Nsight, VTune, TAU, and VampirTrace provide a visualization of the API calls, with related OpenCL context and GPU activity (memory transfers and kernel execution on the device) underneath the API calls.

None of these tools provide a system-wide analysis of all activities related to a specific device (GPU). Also, they do not feature system tracing information that could help provide a more accurate picture of the execution context when debugging. Our solution addresses this issue by proposing a system-wide tracing technique that integrates with LTTng for simultaneous GPU and system tracing.

2.3.3. OpenCL Profiling Capabilities. As described by [19], tracing the execution of a program is fairly straightforward. OpenCL provides a function to gather information about the execution of a kernel or a data transfer: we will refer to those events as *asynchronous commands*. This information has to be accessed asynchronously after the asynchronous command completion. The easiest way to collect this information is to subscribe to a callback function that is defin d for the event associated with the asynchronous command, using the *clSetEventCallback* function. As Figure 2 shows, the timing information that concerns the start and end of the kernel execution can be accessed after the end of the kernel execution (at point (5) in Figure 2). For the information to be accessed, the programmer shall use the *clGetEventProfilingInfo* function. Many different timing metrics can be accessed from this function. Here is a list of available information and their description from the open source specification [20]:

(1) Command queued:

the command has been added to the host queue.

(2) Command submitted:

the command has been transferred from the host queue to the device queue.

(3) Command execution start:

the command started executing on the device.

(4) Command execution end:

the command is done executing on the device.

All of those metrics allow access to a 64-bit integer device time counter at the moment specifi d [20].

Knowing these four time specific tions, we are now able to defin precisely the content of the three waiting zones where the commands can reside:

(1) On the host driver waiting queue.

(2) On the device waiting queue.

(3) Executing on the device.

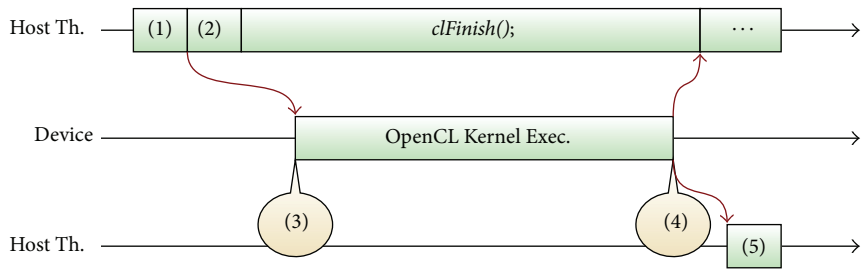
Being able to understand the content of the queues allows the programmer to understand his program resource usage. Then he can modify how his program interacts with the device and enqueues commands in order to maximize computation resources usage. For instance, instead of waiting during data transfers, it is often feasible to overlap computations on one set of data with the transfers for retrieving the previous set and sending the next set.

2.4. Time Keeping. Th importance of the source time clock selection is exposed in [4]. Not all clocks have the monotonic property and in order to ensure that all events are recorded sequentially as they happen, the use of a monotonic clock is the key. The real time clock, for instance, can get set to an earlier time by an administrator or by *Network Time Protocol* (NTP). Using the *C* function `clock_gettime (CLOCK_MONOTONIC, [...]);` to access the time reference helps preserve the monotonicity of the trace.

It is important to understand that this clock does not have an official start point. It only guarantees that a reading will not have a timestamp that precedes past readings. Its usage is restricted to comparing events timing within the same computer session, since the timer is reset to an unspecifi d value at reboot time.

Th same principle is also valid for GPU time stamps: they rely on on-chip hardware to get the time and as the CPU monotonic clock, this time stamp only allows for the computation of the difference between two time stamps (the time delta) taken from the same device. Again, this time stamp origin does not refer to any specific time and is therefore diff rent from the CPU time stamp. Synchronization methods are required to match the CPU event time stamp to the GPU time stamp. Thi problem was already discussed and a synchronization solution was suggested by [21] for synchronizing network traces. The same method may be applied to GPU tracing.

2.5. i915 Tracing. Another way to trace GPU activity is to record already existing trace points that are present in the Intel *i915* GPU drivers. This approach allows monitoring the commands on the host queue and sent to the device, without the need for an interception library such as the solution



(1) *clEnqueueNDRangeKernel(. . . , &event)*; (computing kernel launch)
(2) *clSetEventCallback(event, . . .)*; (subscription to a callback for the end of the kernel execution)
(3) Start time of the computation on the device
(4) End time of the computation on the device
(5) Callback function that records the information ((3), (4) and other)

Figur e 2: OpenCL API profiling.

Ta ble 1: Recording capabilities differences between CLUST and GPU driver monitoring.

	CLUST	i915 monitoring
Host side command queue	√	√
Device side command queue	√	∗
Command-specifi metrics	√	×

∗: We know that a command is either in the device's command queue or currently executing on the device.

presented in this paper: *OpenCL User Space Tracepoint* (CLUST). The upside of this technique is that it monitors all the GPU activity, including graphics (*Open Graphics Library* (OpenGL)).

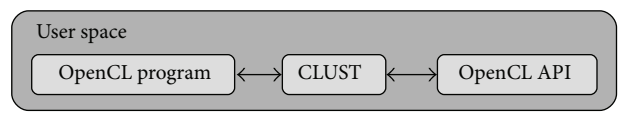
One disadvantage of this technique is that it does not allow modifying the applications execution and as Table 1 shows, the only data available is the current content of the host side and device side queues, as well as commands that are being executed on the device.

Th reason why *i915* monitoring does not allow getting command-specifi timing metrics is because in CLUST, in order to access a command start and end time, a profiling fla is enabled when a command queue is created. This leads to extra event data being pushed to the GPU and filled with timing data on the GPU side (events that are invisible to the GPU drivers).

Therefore, tracing *i915* drivers would be more benefici l as a complement of tracing with CLUST. Thi would allow getting a trace that monitors all GPU activity on the host side and also getting the more detailed information regarding OpenCL that CLUST provides.

3. Implementation

In this section, we present the implementation details of our solution that addresses the identifi d challenges, followed by a system-wide tracing solution and ways to visualize the OpenCL trace.



Figur e 3: CLUST position within the user space.

3.1. CLUST Implementation. As discussed in Section 2.3.3, the OpenCL API can be divided into two separate function types: synchronous and asynchronous:

(1) Synchronous API calls:
calls that do not enqueue tasks on the GPU.
(2) Asynchronous API calls:
calls that enqueue tasks on the GPU

Tho e two types of functions require a different approach for recording relevant tracing metrics.

3.1.1. Synchronous OpenCL Tracing. As seen in Section 2.3.1, the OpenCL library operates in user space. Th purpose of CLUST is to trace an application without the need for recompiling it from its source code. Therefore, CLUST is a library that wraps all function calls of the OpenCL library and automatically instruments all the calls to the API. Figure 3 shows how the API calls are intercepted by the CLUST library. The library file, named *libCLUST.so*, has to be preloaded using `LD_PRELOAD` before launching the program. In the constructor of the library, the OpenCL symbols are loaded from the real OpenCL library and assigned to function pointers that have the same name with the `reallib_` prefix Then, all OpenCL functions are redefined, and the start and end of the API calls are recorded using *Linux Trace Toolkit next generation* (LTTng)-*User-Space Tracing* (UST) (see Listing 1).

3.1.2. Asynchronous OpenCL Tracing. Thi type of recording works well for synchronous function calls. However, for asynchronous function calls such as kernel command enqueue and data transfer command enqueue, a different approach

```
(1) cl_int clWaitForEvents(cl_uint num_events, const cl_event *event_list) {
(2)     tracepoint(clust_provider, cl_clWaitForEvent_start, num_events);
(3)     cl_int ret = reallib_clWaitForEvents(num_events, event_list);
(4)     tracepoint(clust_provider, cl_clWaitForEvent_end);
(5)     return ret;
(6) }
```

List ing 1

```
(1) cl_int clEnqueueWriteBuffer(cl_command_queue[···], cl_event * event)     {
(2)     // Check if device tracing enabled?
(3)     const bool trace =
        __tracepoint_clust_provider__
        _clust_device_event.state;
(4)     bool toDelete = event == NULL;
(5)     if(caa_unlikely(trace)) {
(6)         if(toDelete) { // Dynamic eventallocation
(7)             event = malloc(sizeof(cl_event));
(8)         }
(9)     }
(10)
(11)    tracepoint(clust_provider,
        cl_clEnqueueWriteBuffer_start, [...relevant
         arguments to record...]);
(12)    cl_int ret = reallib_clEnqueueWriteBuffer(command_queue, buffer, blocking_write,
        offset, cb, ptr, num_events_in_wait_list, event_wait_list, event);
(13)    tracepoint(clust_provider, cl_clEnqueueWriteBuffer_end);
(14)
(15)    if(caa_unlikely(trace)) {
(16)        int r = reallib_clSetEventCallback(*
        event, CL_COMPLETE, &eventCompleted, (toDelete)?&ev_delete:&ev_keep);
(17)        if(r != CL_SUCCESS) { [...errormanagement...] }
(18)    }
(19)
(20)     return ret;
(21) }
```

List ing 2

must be considered. All of the asynchronous commands have an event provided as parameter. This parameter may be `NULL`, when the caller does not intend to use an event. When being `NULL`, because CLUST internally needs it, an event has to be allocated and later freed in order to avoid memory leaks. Here is an example with the asynchronous function *clEnqueueWriteBuffer* (see Listing 2).

Since this function only enqueues a command in the command processor of the driver, we have to wait until the end of the command execution to get access to more valuable timing metrics. Th callback function *eventCompleted* will be called at the end of the command execution. This is where the interesting timing metrics can be accessed. We access the four timing metrics discussed in Section 2.3.3 using the *reallib_clGetEventProfilingInfo* function and since we do not know which queue and command type the callback is referring to, we also access the command type id and the command queue id that are associated with the event, using the *reallib_clGetEventInfo* function. All collected information is then recorded in a LTTng UST trace point and the *event* is freed, if the callback parameter specifies that it has to be.

Unfortunately, the timing data collected in this way cannot be written to the trace in sequential time order, since this information was accessed after the execution of the command. Thi leads to problems at trace analysis time since existing tools assume that the events are written to the trace in a sequential time order.

In addition to recording API function call start and end times, the tracepoints can include relevant metrics to allow the analysis of other important information. This relevant data can be important for later analysis and allow for derived analysis, such as calculating the throughput of data transfers by saving the size in bytes of the data being transferred. By associating this measurement with the timing data recorded

by the asynchronous method described previously, we can obtain the throughput of each data transfer. Other metrics such as the queue id can allow for a better understanding of the queues content.

3.2. System-Wide OpenCL Profiling. One of the main advantages of CLUST over existing tracing tools is that it can be used to profile not only one process at a time but also all concurrent OpenCL activity simultaneously.

Thi can be achieved by forcing *LD_PRELOAD* of the *libCLUST.so* library directly from the */etc/environment* variable defin tion file. Every program will inherit the overridden OpenCL symbols and then all OpenCL applications would automatically be traced when LTTng's CLUST tracing is enabled. Th results section (Section 4) show that the overhead of CLUST when the library is preloaded but not tracing is very small. Therefore, this could even be used in production, for 24/7 operation, without significant impact on system performance.

3.3. Nonmonotonic Tracepoint Management. Previously, we identifi d two different types of CLUST events: the synchronous and asynchronous events. As seen in Section 3.1.1, tracing synchronous events is simple and the timestamp linked to the events is added by LTTng-UST using the monotonic clock at the time of the tracepoint execution.

Th challenge lies in collecting asynchronous timestamp data; since the device event timing metrics are accessed after the execution of the command, the timestamp linked with the LTTng-UST event shall not be used to display the device event timing. Th timestamps reside in the payload of the tracepoint and the payload timing data that has to be used.

The trace analysis tools, by default, use the tracepoint timestamp for display and analysis. Special treatment would thus be required to indicate that in fact an event contains information about four asynchronous events, each with its own timestamp. Furthermore, these embedded asynchronous event timestamps may not appear sequentially in the trace and use a different time source. Analysis tools such as *Trace Compass* (http://projects.eclipse.org/projects/tools.tracecompass/) are designed to deal with very large traces and process events for display and analysis in a single pass, assuming a monotonic progression of timestamps. Different approaches can be used to solve this problem; their implementation is discussed in the next section.

3.3.1. Writing in Different Trace Channels. LTTng records traces in different channels, depending on the logical CPU id, in order to record traces with low overhead. Using additional channels for asynchronous OpenCL events would be possible. The delay between the synchronous events and the asynchronous events could be tolerated in this way. However, this would only work if the asynchronous events are always executed in submission order. One channel would be created for each time value type (Kernel queued, Kernel submitted, Kernel execution start, and Kernel execution end). If the submission order is always followed and asynchronous callback events are received in the same order, the events could be written in each channel when the callback is received and would remain in sorted time order in each channel. However, since the command scheduling on the device is not necessarily first-in fi st-out, this technique cannot be used.

3.3.2. Sorting. If there is an upper bound on the extent of reordering needed among commands, it is possible to buff r the events in memory and not output events before every preceeding event has been seen. When information about an asynchronous command is obtained through a callback, four events are described. By correlating this information with the *clEnqueueNDRangeKernel* calls, it would be possible to determine when all data from earlier events has arrived and it is safe to write some of these events to the trace.

Figure 2 depicts the execution of a call to enqueue an OpenCL kernel and its execution on the device. Ignoring the calls to *clSetEventCallback* that are not traced (action performed by CLUST), the order in which the trace is written to the file is the following:

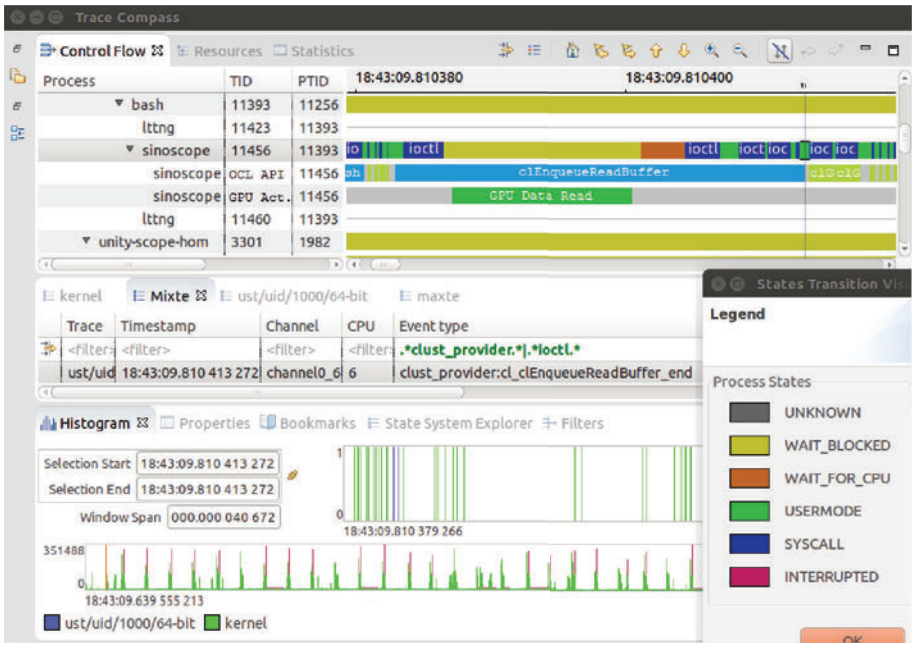
(1) *clEnqueueNDRangeKernel* start,
(2) *clEnqueueNDRangeKernel* end,
(3) *clFinish* start,
(4) *clFinish* end,
(5) Kernel execution timing metrics: queued time, submit time, start time, and end time.

Th reprocessing of the trace would extract the four events and require reordering the tracepoints as follows:

(1) *clEnqueueNDRangeKernel* start,
(2) Kernel queued to the host's driver,
(3) Kernel submitted to the GPU,
(4) *clEnqueueNDRangeKernel* end,
(5) *clFinish* start,
(6) Kernel execution start,
(7) Kernel execution end,
(8) *clFinish* end time.

When a command is queued with *clEnqueueNDRangeKernel*, the corresponding event can be used to mark the appearance of that kernel, k_i. The callback event for k_i is received later with the execution timing metrics. Th latest kernel queued with *clEnqueueNDRangeKernel* at that time, k_j, where $j > i$, is noted. The four embedded events in the k_i callback (Kernel queued, Kernel submitted, Kernel execution start, and Kernel execution end) are then buffered until they can be written safely to the trace because it can be verifie that all information about earlier events has been received. Thi happens when all callbacks for events up to k_j have been received. Indeed, any kernel starting after k_j cannot have any of its four embedded events with timestamps earlier than those in k_i.

Th number of such events that can be *in fli ht* is bounded by the total size of the queues (in the library, in the driver, and on the card). This buffering and proper



Figur e 4: Trace Compass GPU trace integration example.

sorting could be achieved at trace generation time, when receiving callbacks in CLUST. Alternatively, it could be a separate postprocessing step or be performed at trace reading time in *Trace Compass* as an input filter. Doing this at trace generation time, rather than as a separate step, may be more effici t overall, because it avoids writing and later reading and reprocessing. However, minimizing the traced system perturbation is the first priority, which makes this solution less attractive.

If the upper bound on the number of events to buffer for sorting is too large, one can decide to resort to general sorting, using one of the available algorithms for external sorting, such as fusion sort. The downside is then having to reprocess the whole trace file before being able to visualize it. This also does not comply with live trace analysis requirements. At least, the asynchronous events should be recorded in their own channel, to separate them from the other sorted events and minimize the size of the trace fil needing a reordering step.

3.4. GPU-CPU Trace Synchronization. As explored in Section 2.4, the timing sources of the CPU trace and the GPU trace are not the same. For this reason, the time relation between both time sources needs to be evaluated. This will allow aligning properly the CPU trace with the GPU timing metrics collected.

To compute the time relation, we must find the OpenCL API enqueue functions within the CLUST LTTng-UST trace and determine the most closely related device event tracepoint:

(1) The queued time (GPU) of the asynchronous command is after the event time of the OpenCL enqueue API call (CPU).

(2) The end time of the asynchronous command (GPU) is before the event time (CPU) for the callback providing the device events timing metrics.

These two inequalities can then be used to obtain the linear relation between the two timing sources by applying the Convex Hull synchronization algorithm, as proposed in [21], or the more effici t improvement, suitable for live streaming, proposed in [22]. In order to perform trace synchronization, we will need to ensure that all relevant information is gathered in the trace. LTTng-UST already records the *cpu_id* and time stamp (CPU) linked to the OpenCL API call start and end tracepoints as well as the callback event tracepoint. Th additional information required in the tracepoint event payloads is the callback event pointer in order to properly match the *clEnqueueNDRangeKernel* start tracepoint with the corresponding callback event tracepoint.

3.5. OpenCL Trace Visualization. Th main visualization tool for LTTng is `Trace Compass` (previously known as the *Tracing and Monitoring Framework* plugin within the Eclipse Linux Tools project). This tool already provides visualization for LTTng kernel space tracing and user space tracing. As shown in Figure 4, the left tree displays a hierarchy of processes. The principle is to link the OpenCL activity as a child to the associated process. Thi implies that there should be one child per thread performing API calls to OpenCL and two additional children per context linked to the parent: one for displaying data transfers between the host and the device and the other one for displaying the OpenCL kernel execution. Figure 4 shows how the trace unification enhances the information shown to users.

In addition to the *Control Flow* view, the *Resources* view can be used to display GPU usage: two lines per GPU: one for

Table 2: Synchronous OpenCL API function overhead benchmark.

Loop Size	Base ave. (ns/call)	Base Std. dev. (ns/call)	Preload ave. (ns/call)	Preload Std. dev. (ns/call)	Trace ave. (ns/call)	Trace Std. dev. (ns/call)	Preload overhead (ns/call)	Trace overhead (ns/call)
1	16	3	18	3	383	8	2	367
10	5.2	0.5	7.8	0.6	366.5	2.2	2.6	361.3
10^2	4.64	0.04	6.66	0.05	365.68	6.38	2.02	361.04
10^3	4.291	0.006	6.058	0.028	365.168	2.88	1.767	360.877
10^4	4.277	0.012	6.283	0.036	359.780	13.425	2.006	355.503
10^5	4.526	0.005	6.484	0.101	359.379	1.055	1.958	354.853
10^6	4.531	0.029	6.467	0.097	363.313	5.138	1.936	358.782
10^7	4.537	0.018	6.499	0.150	361.145	2.791	1.962	356.608
10^8	4.535	0.022	6.460	0.026	361.108	1.966	1.925	356.573

Sample size = 100.

the data transfers and the second one for OpenCL kernel execution.

It is important to consider that the architecture of modern GPUs allow for concurrent multikernel execution on the same device, via different streams. If the GPU has multiple streams, it then requires more than one row for displaying OpenCL kernel execution.

4. Results

In this section, we discuss the results and primarily the overhead of using the OpenCL tracing library CLUST. First, we expose the details of our test platform and the different tests configurations that will be exercised. Then, we measure the library's overhead for both types of tracepoints. Secondly, we measure the impact of tracing on a real OpenCL program. Finally, examples of how the combined CPU and GPU traces can help troubleshoot issues and enhance application performance will be presented and discussed.

4.1. Test Setup. To demonstrate the low impact of CLUST and LTTng-UST tracepoints, we benchmarked different scenarios of execution with the following different factors considered:

(1) Sample size:

we measured 100 samples of different sizes ranging from 1 call to 10^8 calls. The need for different sample sizes is to test whether the recording of many tracepoints overwhelm the system or if the design of LTTng really allows for maximum performance.

(2) Benchmark configuration:

in order to establish a baseline and isolate the impact of the different components, we measured three different execution configurations:

(a) Base: tracing disabled, without CLUST.
This will serve as the reference for our overhead tests.

(b) Preload: tracing disabled, with CLUST preloaded.
When replacing the symbols of the OpenCL library, an overhead is added: the two disabled LTTng-UST tracepoints and the indirection of the pointer to the actual OpenCL function.

(c) Trace: tracing enabled with CLUST preloaded.
This is the same as the preload but with tracing enabled: the overhead of storing and recording to disk the events will be added to the preload overhead.

(3) Traced element type:

as discussed earlier, there are two types of functions in the OpenCL API: the synchronous API functions that do not require any GPU interaction and the asynchronous API functions that enqueue commands to interact with the GPU. We suspect that the impact of CLUST for asynchronous functions is higher since a callback is hooked to every asynchronous OpenCL function calls.

We show in Tables 2, 3, and 4 the average time in nanosecond for each benchmark configurations (*Base*, *Preload*, and *Trace*) as well as the standard deviation. The overhead displayed is the difference between the measurement and the *Base*.

The benchmarks were performed on our test platform running an Intel i7-4770, 32 GB of memory and using the integrated GPU (HD4600) with the *Beignet* OpenCL development drivers. Tracing was done with LTTng (v2.6.0) and run on Ubuntu Desktop 14.04 LTS with an upgraded kernel (v3.18.4).

4.2. Synchronous Function Tracing Overhead. For the synchronous API calls, the data is displayed in Table 2. The conclusions we can observe from these results are the following:

(1) The average base time (*BaseAverage*) for a call to clGetPlatformID is about 4.5 ns.

(2) The average "preloaded" time (*PreloadAverage*) for a call to clGetPlatformID is about 6.5 ns.

Ta ble 3: Asynchronous OpenCL API function overhead benchmark.

Buff r Size (Byte)	Base ave. (ns/call)	Base Std. dev. (ns/call)	Preload ave. (ns/call)	Preload Std. dev. (ns/call)	Trace ave. (ns/call)	Trace Std. dev. (ns/call)	Preload overhead (ns/call)	Trace overhead (ns/call)
4	149.51	1.47	164.7	1.1	7000.6	261.8	15.2	6851.1
40	158.99	0.92	168.7	1.3	7026.8	289.5	9.7	6867.8
400	156.15	1.50	174.7	1.3	7269.3	240.6	18.5	7113.2
4×10^3	188.44	1.14	226.7	1.3	7043.6	244.2	38.3	6855.2
4×10^4	1499.76	5.47	1503.3	5.6	8393.0	227.2	3.6	6893.3
4×10^5	17805.67	134.31	17862.1	16.1	25404.7	276.3	56.4	7599.0

Sample size = 100; loop size = 1000.

Ta ble 4: Real application tracing timing.

Width (pixel)	Height (pixel)	Base ave. (ns/iter.)	Base Std. dev. (ns/iter.)	Preload ave. (ns/iter.)	Preload Std. dev. (ns/iter.)	CLUST ave. (ns/iter.)	CLUST Std. dev. (ns/iter.)	CLUST + LTTng ave. (ns/iter.)	CLUST + LTTng Std. dev. (ns/iter.)
1	1	35198	2162	36399	1941	50890	2297	58838	3364
10	1	35183	1916	35702	821	51883	2315	58265	3135
10	10	36031	1936	36890	1941	50758	1931	59619	3531
10	100	37937	1868	39067	169	55820	2731	61108	2645
100	100	56770	2440	59709	2135	75073	2661	84746	2232
1000	100	250694	3371	251165	3268	268726	3388	280299	4534
1280	720	1951826	4104	1951965	4443	1976916	4528	1988445	4747
1920	1080	4466096	6345	4466777	5597	4491589	5636	4511394	5509

Sample size = 1000; loop size = 100.

(3) Since each "preloaded" call makes two UST tracepoint calls (without recording anything), we can calculate that the average time for a disabled UST tracepoint (T_1) is only 1 nanosecond:

$$T_1 = \frac{PreloadAverage - BaseAverage}{2} = 1\ \text{ns}. \quad (1)$$

(4) Looking at the "traced" time ($TraceAverage$), we can see that no matter how many tracepoints we generate, the performance per tracepoint is not aff cted and remains steady.

(5) The "traced" metrics show that the overhead of tracing synchronous OpenCL API calls is about 361 ns minus the base call time of 4.5 ns, which give us 356.5 ns for recording two UST tracepoints, or 178.2 ns per tracepoint.

4.3. Asynchronous Function Tracing Overhead. As displayed in Figure 2, recording asynchronous event timing metrics on OpenCL requires hooking a callback function to the event. Th call to an enqueuing function will have the API function's enqueue start and end tracepoints recorded and when the command is done processing by the GPU, another custom tracepoint that contains the timing metrics will be created. Th overhead of such event is expected to be larger than any other synchronous API call since it contains a tracepoint with 44 bytes of payload. This payload has to be generated from 4 calls to *clGetEventProfilingInfo* and 2 calls to *clGetEventInfo*. Thi adds up to the overhead of tracing but it is important to keep in mind that the asynchronous calls are generally performed only a limited number of times per iteration in a computation as compared to synchronous calls. Table 3 shows the average time and standard deviation for various executions of data read from the device to the host using the *clEnqueueReadBuffer* function, configured for blocking read as a benchmark. Since we have previously shown in Table 2 that the performance of tracing is unaffected by the number of tracepoints to record, this table will show the performance for a fi ed number of iterations with different buffer sizes in order to compare the overhead of tracing as a function of the size of the data transferred.

This approach shows that the weight of tracing such calls is constant as a function of the size of the buff r being transferred, and the tracing overhead impact per tracepoint is lowered when using larger buffers. Unfortunately, the OpenCL read command time is greatly increased and small buffer reads suffer the most in relative terms. On our test bench, with the `Beignet` OpenCL drivers, the time measurements are more accurate since the PCI-Express communication latency is not present for integrated graphics.

Measurements have shown that this overhead is caused by two different sources: the calls to the *clGetEventProfilingInfo* function for gathering the start and end times on

the device (approximately 1600 ns per call) and the overhead of providing an event to the OpenCL enqueue function (again, twice the same 1600 ns `ioctl` syscall). The *Beignet* library performs a heavy *syscall* to access and write the recorded start and end time of asynchronous functions: this results in a cost of four times 1600 ns, more than 90% of the overall tracing overhead. Th recording of the events with LTTng-UST has been measured to be insignific nt, as shown in Table 2. Therefore, when tracing is disabled, the data gathering is not performed and the call to the UST tracepoint is also not performed. Avoiding the callsoverwhelm to extract information when tracing is disabled keeps the CLUST library overhead as low as possible when not active.

Thi very high overhead for retrieving two long integers from the GPU may be reduced by modifying the driver and possibly the GPU microcode to automatically retrieve this information at the end of the execution of the asynchronous command, when OpenCL profiling is enabled.

4.4. Real OpenCL Program Tracing Overhead. Knowing the cost of tracing individual synchronous and asynchronous OpenCL calls does not represent the typical overhead of tracing an OpenCL program. For testing purposes, we used a synthetic benchmark that performs 14 synchronous OpenCL calls and 2 asynchronous calls per iteration. The program generates an image of variable width and height, from one simple run of a very light OpenCL kernel.

We measured the performance for CLUST tracing alone and then the overhead of tracing the operating system kernel with LTTng, for a unified trace overhead measurement. Results are displayed in Table 4.

4.4.1. CLUST Tracing. Th expected CLUST tracing overhead is at least 14× ∼ 359 ns + 2× ∼ 7030 ns =∼ 19086 ns per iteration. The results are displayed in Table 5.

As we can see, the overhead of tracing has a signific nt impact on small sized computation tasks. However, when the program is working on bigger images, for more typical resolutions such as 720 p and 1080 p, the relative tracing overhead factor is considerably reduced as compared to the complexity of the calculation itself. GPGPU applications are usually more effici t on larger data sets anyway, again because of the fi ed startup costs of sending the commands to the GPU. Figure 5 shows the relative overhead (R_O from (2)) of tracing with CLUST as a function of the workload:

$$R_O = \frac{TraceAverage - BaseAverage}{BaseAverage}. \quad (2)$$

We can also verify the expected tracing overhead computed at the beginning of this section (∼19086 ns overhead per call); we can see that the measured tracing overhead for the real OpenCL programs varies from 14727 ns to 25494 ns and averages at 18990 ns, a measurement that is within 0.5% of the expected tracing overhead.

4.4.2. CLUST Tracing + LTTng Kernel Tracing. Th main advantage of our solution, as compared to existing CPU tracing and GPU tracing approaches, is the unifie tracing capabilities. Our solution allows recording the CPU trace as well as the GPU trace in the same trace format. The gathered performance data is displayed in Table 4 under the *CLUST + LTTng* columns.

As Figure 5 shows, the tracing overhead of the unifi d CPU + GPU tracing is yet again very low for larger computations. One of the reasons why the unifi d tracing overhead does not signific ntly differ from the CLUST tracing overhead is because the GPU execution speed is not affected by CPU tracing. The overhead is therefore a bit higher than the CLUST tracing overhead alone but this difference fades as the workload size gets larger.

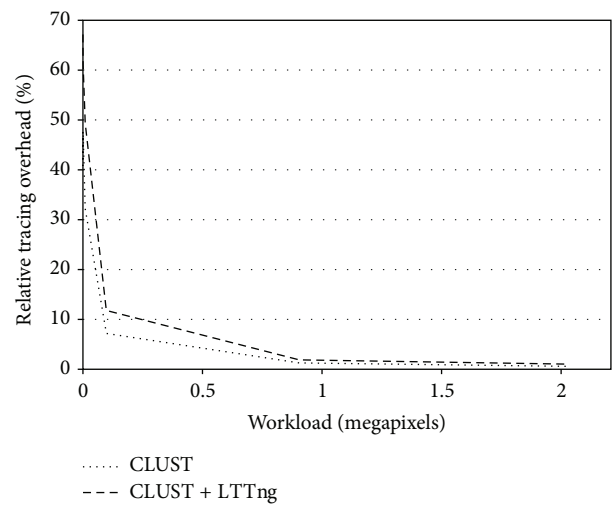
It is interesting to point out that the longer the iterations are, the more LTTng kernel events are recorded. This can be observed by the growth of the overhead per iteration in the *CLUST + LTTng Overhead* column of Table 5.

Th addition of LTTng kernel tracing to the GPU trace allows programmers to find links between the system state and the GPU execution speed. This can lead to performance improvements and bug solving via trace analysis.

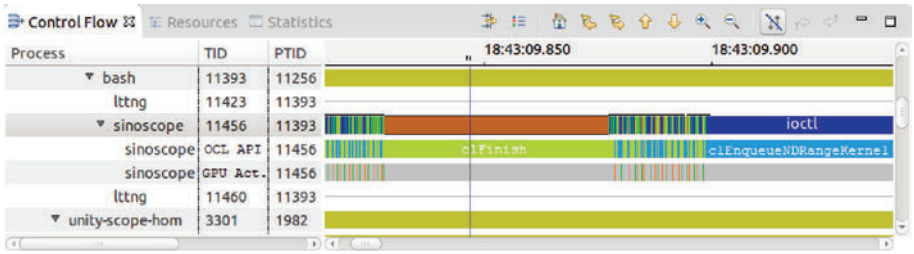
Ta ble 5: Overhead of using CLUST and CLUST + LTTng.

Width (pixel)	Height (pixel)	Preload overhead (ns/iter.)	CLUST overhead (ns/iter.)	CLUST + LTTng overhead (ns/iter.)
1	1	1201	15692	23640
10	1	520	16700	23082
10	10	859	14727	23588
10	100	1131	17883	23171
100	100	2939	18303	27976
1000	100	471	18032	29605
1280	720	139	25090	36619
1920	1080	681	25493	45298

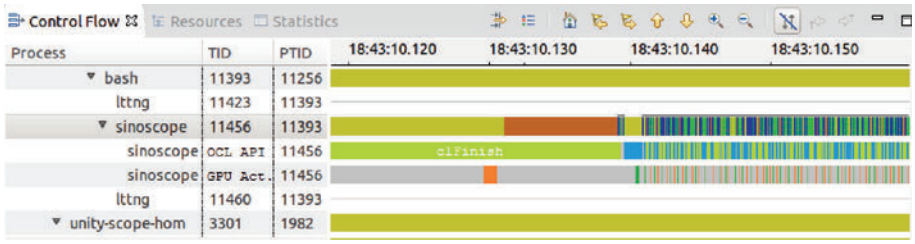
Sample size = 1000; loop size = 100.



Figur e 5: Overhead of CLUST tracing and unifi d CLUST + LTTng tracing relative to the workload size.



Figur e 6: Unifie CPU-GPU trace view that depicts latency induced by CPU preemption.



Figur e 7: Unifie CPU-GPU trace view that depicts latency induced by GPU sharing.

4.5. CLUST Use Cases. Th biggest advantage of CLUST is not only its low overhead characteristics but also the new kind of information it provides to the programmers. Having access to a unifi d CPU-GPU trace and its analysis tools can help troubleshoot different types of problems or even help enhance application performance through GPU pipeline analysis.

Examples that depict this kind of troubleshooting scenarios and enhancements will be demonstrated in this section.

4.6. Typical Execution. As a reference for the following use cases, Figure 4 shows a close-up view of one iteration of the program that will be analyzed. It consists of a trace of the program that was presented in Section 4.4 and used to gather performance data. Thi iteration showed no abnormality and will therefore serve as a base for the upcoming use cases.

4.7. Abnormal Execution. There are two primary types of abnormal executions. The first type occurs when the CPU is preempted, the second occurs when the GPU is being used by different processes.

4.7.1. CPU Preemption. CPU preemption occurs when the scheduler gives CPU time to another process. For GPU intensive applications, this can lead to catastrophic performance because the CPU constantly gets interrupted and put on wait for device (GPU in our case) when executing calculations or communicating with the device. Figure 6 shows an example of how a unified trace visualization tool would help spot this kind of latency. Th orange section of the system kernel trace shows that the CPU is in the *WAIT_FOR_CPU* state while the call to a read buffer OpenCL function was completed. Also, the sections on the right and the left of the orange section show normal iteration times. A programmer could fix this problem by making sure that the thread that manages the OpenCL calls has a higher priority than normal and making sure that no other process uses all other available resources.

4.7.2. Shared GPU Resources. A second issue that can be observed is when the GPU device is shared between multiple processes. Th GPU's primary task is to deal with rendering the screen. This process can interfere with the OpenCL kernel execution, and data transfers between the host and the device. Figure 7 shows a great example and its effect on the system. In yellow, we can see that the system is in state *WAIT_BLOCKED* for a long time before being preempted (in orange). On the right side of Figure 7, we can see that the process is running normally. This can be caused by the execution of a concurrent application that uses the GPU resources. If the OpenCL program analyzed is critical, making sure that no other process interferes with it can fix this problem.

4.8. Nonoptimized Execution. Another kind of flaw that OpenCL analysis can outline is nonoptimized OpenCL programs. Due to the different architectures of the GPUs, adapting to the device characteristics is a very important feature of OpenCL programs. Pipeline enhancement and kernel occupancy are just two of the main focuses for optimization. In both cases, the goal is to maximize the device and host resources utilization.

4.8.1. OpenCL Pipeline Enhancement. One of the best ways to optimize the speed of an OpenCL application is to properly use its pipeline. OpenCL has the capability to process simultaneously device communications and OpenCL kernel executions. Using OpenCL's asynchronous command enqueuing functions, it is possible to maximize the GPU resources usage. As we can see in both Figures 6 and 7, the gray area of the *GPU Act.* represents idle time for the GPU.

These views show that most of the computing and communication time is not being used.

Depending on the application type, the idle time can be reduced by dividing the calculations when possible and enqueuing multiple commands in different command queues to ensure that the device always has work to do. As an example, we divided the workload of our application benchmark and noticed a 28% speedup when running 13 command queues instead of one (with one `pthread` per command queue). We also measured a speedup of up to 72% for 12 command queues on a different dedicated GPU. The bigger speedup on the dedicated GPU can be explained by the larger overhead of interacting with the device: the latency to communicate with a dedicated GPU is higher than for an integrated graphics GPU. Therefore, dedicated GPU configurations are well suited for pipeline enhancement. Aft r 12 and 13 command queues and threads, we started to notice a slow down when we reached the device saturation point.

CLUST records the required information to allow the analysis and visualization of this kind of behaviour.

4.8.2. OpenCL Kernel Occupancy. Unfortunately, CLUST cannot currently extract and record the required metrics on the device regarding kernel occupancy since this information is not available in the OpenCL standard. However, having a record of the *clEnqueueNDRangeKernel* parameters and knowing the device architecture, can help evaluate how the computation was divided and if compute blocks were executed while using a maximum of available resources on the GPU.

5. Conclusion and Future Work

In this paper, we explored the GPU tracing challenges and addressed the need for a unifi d CPU-GPU tracing tool. This allows programmers to have a better global overview of the execution of OpenCL powered applications within the whole system. We demonstrated the very low overhead of intercepting OpenCL calls to insert LTTng-UST tracepoints, not only when tracing is inactive, but also when tracing is enabled. Thi minimizes the impact on the system and maintains the spirit of LTTng as a very low overhead tracing tool. To our knowledge, we presented the only unifi d GPU-CPU tracing tool available in the open literature.

In complement to this, we demonstrated the analysis and visualization capabilities that a unifie trace can provide, and the type of problems that it can help uncover.

As this is a valid solution for OpenCL tracing, future enhancements would be to implement OpenGL and CUDA UST tracing in order to have a global understanding of all GPU activity. Kernel space driver tracing could also be a great improvement for understanding the host's queue. Trace Compass's unified views also need to be fine-tuned. Thi tracing model not only can serve for GPU tracing but also paves the road for other heterogeneous computing analyses.

Th code for this project is open source and available on https://github.com/dcouturier/CLUST.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Th authors would like to thank Ericsson, Natural Sciences and Engineering Research Council of Canada (NSERC), and Effici for making this research possible.

References

[1] B. Gregg and J. Mauro, *DTrace; Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*, vol. 26, Book News, 2011, http://search.proquest.com/docview/869984396.

[2] S. E. Fagan, "Tracing BSD system calls," *Dr. Dobb's Journal*, vol. 23, no. 3, p. 38, 1998, http://search.proquest.com/docview/2027-19549.

[3] W. C. Don Domingo, *SystemTap 2.7—System-Tap Beginners Guide: Introduction to SystemTap*, 2013.

[4] M. Desnoyers, *Low-impact operating system tracing [Ph.D. thesis]*, École Polytechnique de Montréal, 2009.

[5] OpenCL—the open standard for parallel programming of heterogeneous systems, 2015, https://www.khronos.org/opencl/.

[6] Strace project: strace(1) Linux Manual Pages, 2010, http://man7.org/linux/man-pages/man1/strace.1.html.

[7] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffe ing scheme for kernel tracing," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 65–81, 2012.

[8] B. Gregg, "Strace wow much syscall," 2014, http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html.

[9] B. Gregg, *Tracing Summit 2014: From DTrace To Linux*, Brendan Gregg (Netix), 2014.

[10] D. Goulet, *Unified kernel/user-space efficient Linux tracing architecture [M.S. thesis]*, École Polytechnique de Montréal, Montréal, Canada, 2012.

[11] H. Waly and B. Ktari, "A complete framework for kernel trace analysis," in *Proceedings of the 24th Canadian Conference on Electrical and Computer Engineering (CCECE '11)*, pp. 001426–001430, May 2011.

[12] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Advances in Software Engineering*, vol. 2012, Article ID 140368, 12 pages, 2012.

[13] A. Palnitkar, P. Saggurti, and S.-H. Kuang, "Finite state machine trace analysis program," in *Proceedings of the International Verilog HDL Conference*, pp. 52–57, IEEE, Santa Clara, Calif, USA, March 1994.

[14] N. Ezzati-Jivan and M. R. Dagenais, "A framework to compute statistics of system parameters from very large trace files," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 43–54, 2013.

[15] N. Ezzati-Jivan and M. R. Dagenais, "Cube data model for multilevel statistics computation of live execution traces," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1069–1091, 2015.

[16] R. Dietrich, F. Schmitt, R. Widera, and M. Bussmann, "Phase-based profili g in GPGPU kernels," in *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW '12)*, pp. 414–423, September 2012.

[17] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazel-Wood, "Analyzing program flow within a many-kernel OpenCL application," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4*, pp. 10:1–10:8, ACM, New York, NY, USA, 2011.

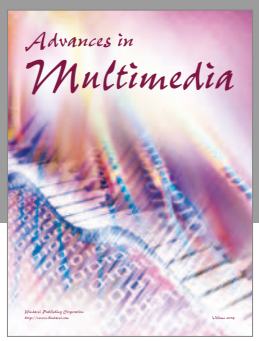
[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

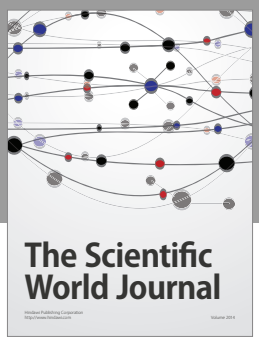
[19] G. Juckeland, "Trace-based performance analysis for hardware accelerators," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Mller, W. E. Nagel, and M. M. Resch, Eds., pp. 93–104, Springer, Berlin, Germany, 2012.

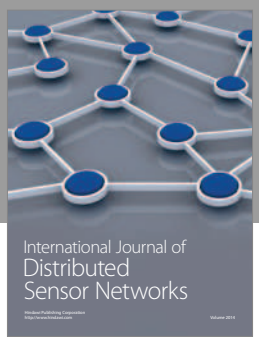
[20] Khronos Group, *OpenCL Reference Pages*, Khronos Group, 2011, https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.

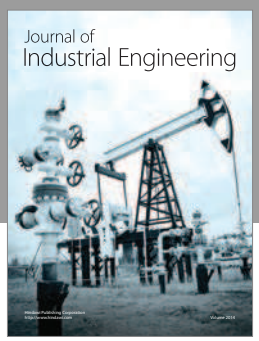
[21] B. Poirier, R. Roy, and M. Dagenais, "Accurate offli synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.

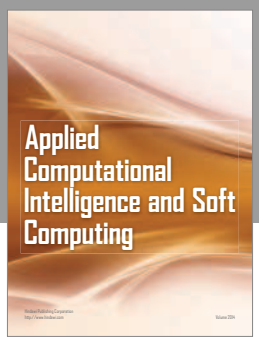
[22] M. Jabbarifar, "On line trace synchronization for large scale distributed systems," ProQuest, UMI Dissertations Publishing, 2013, http://search.proquest.com/docview/1561560787.

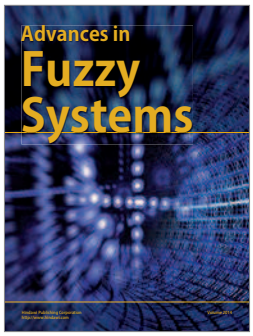


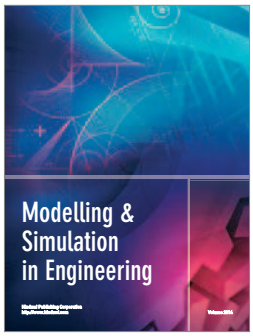


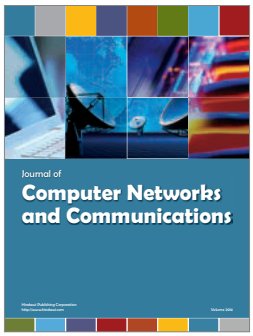


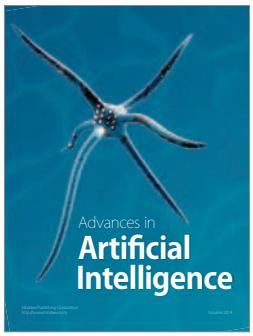
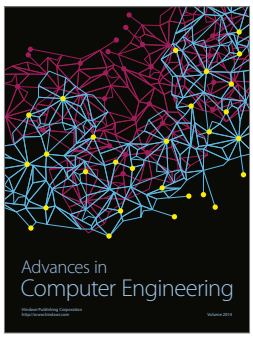
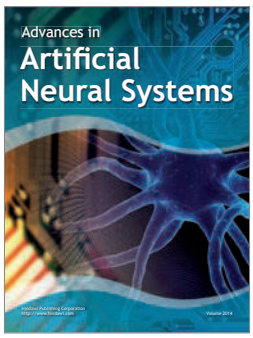
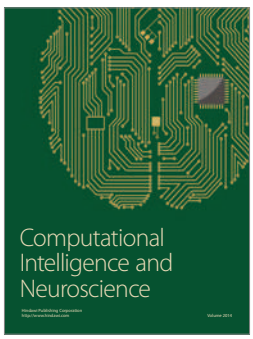










Hindawi

Submit your manuscripts at
http://www.hindawi.com