Introduction to GPU "Low-level" Programming

How does that even work?

Thomas Applencourt (apl@anl.gov)
July 31, 2023
• "If it’s simple, it’s always false. If it’s not, it’s unusable.” Paul Valéry
• "Trust, but verify” Russian proverb
• And sorry in advance for the 3h long lecture…¹

¹“Stay awhile and listen…” Deckard Cain
1. Introduction

2. Programming Model API / Runtime

3. Kernel Language / Compilation

4. Bonus: And where the fun begins

5. Conclusion

6. Q&A
Introduction
Who I Am

• Part of the Performance Engineer Group at Argonne\(^2\)
• Main focus on Aurora Compiler and Runtime
  • So Expertise in Intel Toolchain, the rest will be more hand-wavy...\(^3\)
• Member of the SYCL committee

\(^2\)Like Vitali and Serves\(h\)
\(^3\)I’m sure some people in the audience will be able to answers any AMD/NVIDIA question
Non-Goals of this lecture

• Teach you CUDA, Hip, Level Zero\(^4\)
• You are all smart, if you need to learn it you can find super nice tutorial online

\(^4\)But maybe I will teach you some OpenCL...
Goal of this lecture

• Give you some foundation to understand the difference and similitude between multiple low-level programming models ("Any fool can know. The point is to understand." Ernest Kinoy)
• Make clear the layering approach of current toolchains
One big question

- Why one "need" to use CUDA for NVIDIA hardware, Hip for AMD, and L0 for Intel?\(^5\)

\(^5\)But then how can OpenCL be portable?
Programming Language for GPU

- Programming Model / Runtime\(^6\)
- Kernel Language, and Kernel Execution

\(^6\)This is the only thing who matter, this rest is trivial
Programming Model API / Runtime
Programming Model API / Runtime

Runtime and Runtimes
What is a runtime

A runtime is a implementation of a programming language’s execution model
"All problems in computer science can be solved by another level of indirection”
David Wheeler
Comment on this diagram

- The Linux Drivers are the "lower-level" we will discuss. Huge effort to implement.\(^7\)
- The level on top (CUDA Driver, HSA, Level Zero) abstracts away a little bit more of the hardware, but still provides a lot of control \(^8\)
- The last level (Cuda Runtime, HIP) are "fully" hardware independent

\(^7\)See nice blog post about the Linux M4 drivers
\(^8\)Sweet spot to write higher-lever runtime
Taking about higher-level Programming model

Kokkos → OpenMP offload → Level Zero

OpenCL → hip → hsa

Sycl → Cuda Runtime

Cuda Driver
Example of Paths

- OpenCL -> *\textsuperscript{9}
- OpenMP Offload -> HSA
- Kokkos -> Cuda Runtime -> Cuda Driver
- HIP -> L0\textsuperscript{10}

\textsuperscript{9}Yes, I Like OpenCL... Soon you will too!
\textsuperscript{10}Maybe more surprising, we will talk about this more at the end
• In short we have a ”High Level” programming model. Used by Application.
• A ”low-level” programming model that the high-level runtime is written with
• Each layer of abstraction is a trade-off between flexibility/performance and convenience/productivity

All of this is relative to who you are talking with.
“Sad” Truth

• No technical reason for having so much “intermediate” programming model
  • hipcc was a perl script that did ‘s/cu/hip/g’ to avoid copyright infringement\(^\text{11}\)

• Always hard to have a standard (*insert XKCD*)

• OpenCL is the standard, but low-adoption by vendors

• Please don’t let vendors make the same mistake with new ML accelerators!

\(^{11}\)Not a lawyer, but the "recent" supreme court Google vs Oracle may help
Hip and Cuda Runtime should not be used anymore.

- Too low-level for Application who want to use nice C++ construct\(^\text{12}\)
- Too high-level for people who have advance use-cases.

Kokkos, Sycl OpenMPOffload already bypass HIP / CUDA runtime, so no "overhead" by using those programming models

\(^{12}\)Come on, who wants to cast the output of malloc...
You are young and not yet totally jaded, so I share my dream with you!
Programming Model API / Runtime

Main concepts (shared)
They all have good old C API

Pointer Everywhere (output is the return error code, Cuda Driver Example)\textsuperscript{13}

```c
int count;
err = cuDeviceGetCount(&count);
```

API can be called twice (OpenCL Example, similar in L0)

```c
// Get number of platorm
cl_uint platformCount;
clGetPlatformIDs(0, NULL, &platformCount);
cl_platform_id* platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
// Populate the newly allocated array
errr = clGetPlatformIDs(platformCount, platforms, NULL);
```

\textsuperscript{13} cu* == cuda driver, cuda* == cuda runtime.
And Pointer of Pointer, and strut!

In C, malloc return a null pointer when it fail. Not in cuda!

```c
CUresult cuMemAllocHost ( void** pp, size_t bytesize )
```

Some API use struct to avoid 200 parameters

```c
ze_command_queue_desc_t commandQueueDesc = {
    ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC,
    NULL,
    computeQueueGroupOrdinal,
    0, // index
    0, // flags
    ZE_COMMAND_QUEUE_MODE_DEFAULT,
    ZE_COMMAND_QUEUE_PRIORITY_NORMAL
};
```

```c
ze_command_queue_handle_t hCommandQueue;
errno = zeCommandQueueCreate(hContext, hDevice,
    &commandQueueDesc, &hCommandQueue);
```
What are a GPU Runtime’s Main Goals?

- Find devices, load your kernels
- Allocate GPU Memory
- Transfer Memory: From and To the device
- Execute your commands

Not that hard!
• OpenCL view of the world
• All other programming models are roughly the same
• (Please ignore MemObject and Sampler)

---

\(^{14}\) Picture from the OpenCL Doc
\(^{15}\) Maybe more when it’s a UML diagram
Platform: a collection of Device sharing some property

```c
cl_int clGetPlatformIDs( cl_uint num_entries,
                         cl_platform_id* platforms,
                         cl_uint* num_platforms);

ze_result_t zeDriverGet(uint32_t *pCount,
                         ze_driver_handle_t *phDrivers)

CUresult cuDeviceGetCount ( int* count )
CUresult cuDeviceGet ( CUdevice* device, int ordinal)

% Always fun to change the naming convention
cudaError_t cudaGetDeviceCount ( int* count )
cudaError_t cudaGetDevice ( int* device )
```
• Devices are bound to a Context
• The context holds all the management data to control and use the device.

1  % One Device
2    CUresult cuCtxCreate_v3 ( CUcontext* pctx, CUexecAffinityParam* paramsArray,
3                                int numParams, unsigned int flags, CUdevice dev )
4  % All the Device in the Platform
5    ze_result_t zeContextCreate(ze_driver_handle_t hDriver,
6                                  const ze_context_desc_t *desc,
7                                  ze_context_handle_t *phContext)
For example in L0, it’s forbidden to exchange memory between different contexts (whether they share the same device or not)\textsuperscript{16}

- Any "advanced" use cases (multi-device, multi-process, interface with other library) need to be aware of context.

\textsuperscript{16}I think it’s the same in CUDA...
Note on CUDA: contexts are one of the major differences between runtime and driver APIs

- CUDA driver is a state machine. You pop and push context on a stack

```c
CUresult cuCtxSetCurrent ( CUcontext ctx )
```

- And are not exposed to the runtime API (uses an implicit primary context). So be careful!
[...] for example, because if all plug-ins run in the same process, they will all share a context but will likely have no way to communicate with each other. So, if one of them calls cudaDeviceReset() after finishing all its CUDA work, the other plug-ins will fail because the context they were using was destroyed without their knowledge. ¹⁷

¹⁷ https://docs.nvidia.com/cuda/cuda-runtime-api/driver-vs-runtime-api.html
Submit ”command” to queue/stream. Commands can be

- Memory Copy
- Kernel Submission
- Synchronization
- ...

Queue / Stream
Differences between programming models

• In L0, queues are out-of-order\(^1^8\)
• In CUDA runtime and driver and HIP, streams are in-order\(^1^9\)
• In OpenCL, they can be both
• In HSA, it’s a ring buffer of packets
• Out-of-order queues can be a source of error\(^2^0\)

\(^1^8\) Just checked, the latest L0 version we know have a ZE_COMMAND_QUEUE_FLAG_IN_ORDER flags.
\(^1^9\) For more complex use cases, use cuda-graph
\(^2^0\) But are powerful, will see later
Command Submission

Commands can be submitted Asynchronously or in blocking manner

```c
CUresult cuMemcpy ( CUdeviceptr dst, CUdeviceptr src,
   size_t ByteCount ) % Where is my stream?!
CUresult cuMemcpyAsync ( CUdeviceptr dst, CUdeviceptr src, size_t ByteCount,
   CUstream hStream )
```

```c
cl_int clEnqueueSVMMemcpy(
   cl_command_queue command_queue,
   cl_bool blocking_copy,
   void* dst_ptr,
   const void* src_ptr,
   size_t size,
   cl_uint num_events_in_wait_list,
   const cl_event* event_wait_list,
   cl_event* event);
```

- Async is a common source of error
If it’s asynchronous you need to synchronize

- via Event (specify dependencies for fine grained synchronization)
- via Barrier (for coarse synchronization)
OpenCL, L0\(^{21}\):

1. `zeCommandListAppendMemoryCopy(..., &e1) // e1 will be signaled at completion`
2. `zeCommandListAppendMemoryCopy(..., &e2) // e2 will be signaled at completion`
3. `ze_event_t depend_in [2] {e1,e2};`
4. `zeCommandListAppendMemoryCopy(..., 2, depend_in, &e3) // Inputs and output`
5. `zeEventHostSynchronize(e)`

- `hipMemcpyAsync // Hip, Cuda have optional Async, default blocking`
- `hipEventRecord`
- `hipEventSynchronize`

(for more fancy use cases, use cuda-graph)

\(^{21}\)So elegant
**Synchronization**

Wait on queue / stream (wait until all the work has been done)

1. `zeCommandQueueSynchronize`
2. `cudaStreamSynchronize`
3. `cudaDeviceSynchronize` % Whoa?

Coarse grain. Use with caution.
Programming Model API / Runtime

Memory Allocation
Types of GPU Memory

- Device Memory: Accessible only on the particular device
- Shared Memory: Accessible by both the host and the device
  - This may impact performance, Different migration strategies
  - Can be migrated via prefetching
- Host memory
  - "Pinned" memory. CPU memory but has been registered by the runtime.
  - May required for some optimizations / performance
- Malloc-ed Memory

---

22 Read the documentation to know if it’s accessible by OTHER devices. Context, wink, wink
23 Nvidia calls it ”Managed”
24 Do not confuse with prefetch of memory inside a kernel
25 OpenCL has buffer, but lets not go that way...
But future GPUs will be integrated!

- Doesn’t matter,
- NUMA is bad, Locality is good.
- Please don’t use shared everywhere...
• Wrong data-transfer is the number one bug when doing GPU programming.
• Start with shared-allocation, then trace/profile and optimize\textsuperscript{26}
• Premature Optimization is the root of all evil, but not profiling is eviler!

\textsuperscript{26}SYCL has a nice buffer/accessors to solve the data-transfer problem
Allocates size bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging.  

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.htm
Best Case Scenario

- All data fit on the GPU
- move everything over
- do a ton of computation
- move back

You should aim for them. If you cannot, we will discuss other strategies latter.\(^{28}\). **Memory transfers are expensive. Don’t do it!** Or at least try...

\(^{28}\)Please not that it’s the same in CPU. Keep data in cache
Programming Model API / Runtime

Kernel Submission
Magic / Syntactic sugar\textsuperscript{29}

\begin{verbatim}
1    mykernel<<<blocks, threads, shared_mem, stream>>>(args);
\end{verbatim}

But just call HSA / Cuda Driver behind the scenes.

\textsuperscript{29}Haha, no lambda. Haha, new non C syntax...
behind the seen: OpenCL Cuda

```c
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t *global_work_offset,
    const size_t *global_work_size,
    const size_t *local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)

CUresult cuLaunchKernel ( CUfunction f, unsigned int  gridDimX, unsigned int  gridDimY, unsigned int  gridDimZ, unsigned int  blockDimX, unsigned int  blockDimY, unsigned int  blockDimZ, unsigned int  sharedMemBytes, CUstream hStream, void** kernelParams, void** extra )
```
Behind the seen: Level zero, HSA

1 % Similar in L0, Count versus Size, and by kernel
2 zeKernelSetGroupSize(hKernel, groupSizeX, 1, 1);
3 ze_group_count_t groupCount = { numGroupsX, 1, 1 };
4 zeCommandListAppendLaunchKernel(hCommandList, hKernel, &groupCount, NULL, 0,
5    → NULL);

6 % HSA  Werner told you get a packet from queue and then signaling, but still same idea
7 typedef struct hsa_kernel_dispatch_packet_s { uint16_t header ;
8    uint16_t setup;
9    uint16_t workgroup_size_x ; uint16_t workgroup_size_y ; uint16_t workgroup_size_z ;
10    → uint16_t reserved0;
11    uint32_t grid_size_x ;
12    uint32_t grid_size_y ;
13    uint32_t grid_size_z ;
14    uint32_t private_segment_size ; uint32_t group_segment_size ;
15
Launching Kernel

30 From "Design of OpenCL Framework for Embedded Multi-core Processors"
Loading GPU Kernel

- Your code was split between hosts and GPU code
- Your kernels need to loaded by the GPU runtime!

1. `clCreateProgramWithSource`
2. `clCreateProgramWithIL`
3. `clCreateProgramWithBinary`
4. `zeModuleCreate ( ZE_MODULE_FORMAT_IL_SPIRV | ZE_MODULE_FORMAT_NATIVE)`
5. `cuModuleLoad`
Programming Model API / Runtime

Wrapping Up: Going through an OpenCL Example
Example of OpenCL

- "Low-level" Code (I will guess that cuda-driver will be similar, L0 more verbose, and HSA ever more)
Load Platform and Device

```
cl_uint platform_idx = (cl_uint) atoi(argv[1]);
cl_uint device_idx = (cl_uint) atoi(argv[2]);

clGetPlatformIDs(0, NULL, &platform_count);
cl_platform_id* platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platform_count);
clGetPlatformIDs(platform_count, platforms, NULL);
cl_platform_id platform = platforms[platform_idx];

cl_uint device_count;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &device_count);
cl_device_id* devices = (cl_device_id*) malloc(sizeof(cl_device_id) * device_count);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, device_count, devices, NULL);
cl_device_id device = devices[device_idx];
```
// A context is a platform with a set of available devices for that platform.
cl_context context = clCreateContext(0, device_count, devices, NULL, NULL, &err);
cl_command_queue queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);

// Create the program
cl_program program = clCreateProgramWithSource(context, 1, &kernelstring, NULL, &err);
clBuildProgram(program, device_count, devices, "", NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "hello_world", &err);
```c
#define WORK_DIM 1

size_t global0 = (size_t)atoi(argv[3]);
const size_t global[WORK_DIM] = { global0 };
size_t local0 = (size_t)atoi(argv[4]);
const size_t local[WORK_DIM] = { local0 };

% No vent
clEnqueueNDRangeKernel(queue, kernel, WORK_DIM, NULL,
                        global, local, 0, NULL, NULL);
clFinish(queue);
```
Programming Model API / Runtime

Notes on Performance
Async is Key to good performance

- You want to keep the GPU busy
- When the GPU is computing something, the CPU should start preparing the next batch of work
- Importance of asynchronously

Async: [OpenMP][AMDGPU] Switch host-device memory copy to asynchronous version (real thing: https://reviews.llvm.org/D115279)
Overlapping is Key to good performance

- PCI is damn slow!\(^{31}\)
  - PCI 64 GB/s (unidirectional)
  - HBM 410 GB/s
  - GPU 50 TFlops+
- Recompute is better than to load
- Overlap compute and data-transfers
- PCI is bidirectional so please do H2D and D2H at the same time!
- Avoid over-synchronization!

\(^{31}\)And for integrated architectures, you have NUMA so same things... Data-movement will always be more expensive than compute
So how to achieve concurrency?

Importance of asynchronously!\textsuperscript{32}

- Submit kernels to multiple stream / queue
- Submit kernels to an out-of-order queue.

\textsuperscript{32}Or use multiple thread / process but this is cheating
• You see, all the same. And lot of Bridge between them!
• Context, Queue, Execution Space, synchronize
• Some are less verbose more high level (HIP/CUDA runtime) but you loose some flexibility\(^{33}\)
• IMO HIP/CUDA runtime are in a weird intermediate level.

\(^{33}\) And need to deal with some state-machine...
But does this matter? This sounds trivial

Real Time Experience (controversial)

- Experience: The runtime performance is far more important than the kernel performance
- Improving Kernel performance will give you a few percent; doing too much data-transfer will slowdown your code 100x.
Kernel Language / Compilation
Kernel Language / Compilation

Compilation
• The GPU code needs to be loaded!
• Let’s see how they are compiled
How to write GPU programming

```c
% Cuda HIP
__global__ void cuda_hello()
{
    printf("Hello World from GPU!\n");
}

% OpenCL
__kernel void hello_world()
{
    printf("Hello World from GPU!\n");
}
```

So much difference!

- What can you put in device code has limitation depending on the Hardware / Compiler
- No dynamic allocation, No throw, No recursion, No virtual Functions, ...
Compiling: A little more

(Note that GCC bypass LLVM IR to generate PTX and AMDGCN \(^{34}\))

\(^{34}\) or nobody generate AMDGCN and everybody HIP IL, not clear...
Moving to Single Source

```c
#pragma omp target
printf("Hello World!\n");

// Lambda
Q.single_task([] {printf("Hello World!\n");}).wait();
```

Ask yourself how can you implement this, what "low-level" call you need? We will see it in the Bonus part!
Kernel Language / Compilation

Kernel Language: GPU programming 101
• Just think of the GPU as a CPU with lots of threads executing SIMD instruction
• GPU programs are pretty boring:
  • Use Shared Local Memory\(^{35}\) when possible to not read from the Main Memory
  • Be careful of register pressure
  • "Nothing Special"\(^ {36}\)
  • GPU are fast because they force you to NOT synchronize between threads that live in difference work-group. Freedom versus Performance.

\(^{35}\)Shared Memory in cuda
\(^{36}\)"coalescing memory" == Don’t do random access...
• In SIMD, conditional are implemented as masking, and then execute both branch\textsuperscript{37}

• Same in GPU, But Some GPU have some ”thread divergence” capability\textsuperscript{38}

Volta independent thread scheduling enables interleaved execution of statements from divergent branches. This enables execution of fine-grain parallel algorithms where threads within a warp may synchronize and communicate.

\textsuperscript{37}Both branch if both branch are needed. This is why is better to sort your data...

\textsuperscript{38}After votla for Nvidia
Synchronization Possibilities

From An Introduction to the OpenCL Programming Model by Jonathan A. Thompson
Index space

From "An Introduction to the OpenCL Programming Model" by Jonathan A. Thompson
atomic

- Progress Forward guarantees are complicated...
- I really don’t want to open this can of worms.
- Advice: **Please don’t use atomic to synchronize**
- For an overview of the fun see:
  TowardsAlignmentofParallelisminSYCLandISOC++ by John Pennycook
• GPUs do not have the same ISA as CPU, so two compilation phases
• GPU are fast because they are stupid, and they are just a big SIMD 10k threads CPU
• Lots of good tutorials online for GPU Kernel Programming
Bonus: And where the fun begins
Bonus: And where the fun begins

High Level versus Low-level: Interoperability
So if I’m High Level Language I will be slow?

- Maybe, Maybe not
- Depends on your usage
- Depends on the quality of the runtime
- IMO Cuda Runtime is at the same level as SYCL
• Don’t be afraid of the runtime
• They have bugs and performance bugs\textsuperscript{39}
• Please benchmark, trace, and report
• If you find bugs, just use interopts!

\textsuperscript{39} But maybe less than your code, just because more people use them...
All (at least I hope so) "high-level" can (SHOULD!) should interoperate with some low-level runtime.

```cpp
#pragma omp interop device(id), targetsync : o)
auto hPlatform = static_cast<ze_driver_handle_t>(
    omp_get_interop_ptr(o, omp_ipr_platform, &err))

#pragma omp interop device(id), targetsync : o)
auto hPlatform = static_cast<ze_driver_handle_t>(
    omp_get_interop_ptr(o, omp_ipr_platform, &err));
```

```cpp
#Sycl interop with CUDA
cuStream_T s = get_native<backend::cuda>(Q);
```
Best of both worlds

- Use high-level by default
- Go down when required
Bonus: And where the fun begins

Tracer: How to?
The world of tracer

- Tracing the runtime is mandatory to understand what is going on.
- And it’s fun. Like the satisfaction of putting clarity in life full of chaos.
  - Does Cuda Runtime “foo«»” really call cuda driver ”Launch Kernel”?
  - Does memcpy use Copy Engine or Compute kernel?
  - Why is my code is slow after my 1M gpu device alloc?
What does Tracer do?

- Dump Arguments and timestamp before and after the call
- Provide some analysis tools

```c
clEnqueueMemcpyINTEL_entry:
{ command_queue: 0x181a540, blocking: CL_FALSE,
dst_ptr: 0xffffc001ffd80000, src_ptr: 0x00007f5b20088280, size: 64,
  num_events_in_wait_list: 0,
  event_wait_list: 0x0, event: 0x7ffc4ac01378, event_wait_list_vals: [] }

clEnqueueMemcpyINTEL_exit:
{ errcode_ret_val: CL_SUCCESS, event_val: 0x1dffb30 }
```
How to create callback? intersect API call!

- But How to intercept API call?
- Some APIs provide Callback
- You can always write your own intercept layers!
Famous Example: Inte’s Intercept Layer for OpenCL Applications
https://github.com/intel/opencl-intercept-layer

Nice explanation by Rafal Cieslak https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs/
Interception, hello-world

Please replace rand by some favorite API calls

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("%d\n", rand() % 100);
}
```

$ gcc -shared -fPIC inspect_rand.c -o inspect_rand.so -ldl
$ ./a.out
42

How can we intercept this call?
PRELOAD + dlopen = <3

```c
$ cat inspect_rand.c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
typedef int (*rand)(void);
int rand(void)
{
    printf("Rand_begin");
    orig_rand_f_type orig_rand;
    orig_rand = (orig_rand_f_type)dlsym(RTLD_NEXT,"open");
    int o = orig_rand();
    printf("Rand_end");
    return o;
}
$ gcc -shared -fPIC inspect_rand.c -o inspect_rand.so -ldl
$ LD_PRELOAD=$PWD/inspect_rand.so ./a.out
Rand_begin
99
Rand_end
```
In reality, don’t do that!

- Of course, a little more complicated in real life, but this is what some tools are doing. For example THAPI/iprof that we will show later
- But more and more APIs provide a clean callback mechanism
The clean way: Loader

Application

OpenCL ICD Loader

func 1
func 2

Layer 1
func 2

Layer n
func 1
func 2

Driver 1

Driver 2
OpenCL Example

https://github.com/Kerilk/OpenCL-Layers-Tutorial

```c
static CL_API_ENTRY cl_int CL_API_CALL clGetPlatformIDs_wrap(
    cl_uint num_entries,
    cl_platform_id* platforms,
    cl_uint* num_platforms) {
    fprintf(stderr, "clGetPlatformIDs(num_entries: %d)\n", num_entries);
    cl_int res = tdispatch->clGetPlatformIDs(num_entries, platforms, num_platforms);
    if (res == CL_SUCCESS && num_platforms)
        fprintf(stderr, ", *num_platforms: %d\n", *num_platforms);
    return res;
}

static void _init_dispatch() {
    dispatch.clGetPlatformIDs = &clGetPlatformIDs_wrap;
}
```

And then just put this library and your path, and your tada.
Bonus: And where the fun begins

Tracer Example: THAPI
Why yet another tracer?

• At the beginning no tracer for Level Zero
• Still not one common tracer for low-level-programming
• So we wrote one!
  • THAPI Support: Cuda Runtime, Cuda Driver, HIP, L0, OpenCL, OpenMP-T
```
> ./iprof -t ./a.out
{
  thread_type: ompt_thread_initial, thread_data: 0x00007f5b0cf0ac48
}
ompt_callback_target:
  {
    kind: ompt_target, endpoint: ompt_scope_end, device_num: 0, task_data: 0x0000000000000000,
    target_id: 1, codeptr_ra: 0x00007f5b26fa47e0
  }
[
...
]
ompt_callback_target_data_op_intel:
  {
    endpoint: ompt_scope_begin, target_id: 1, host_op_id: 7, optype:
      ompt_target_data_transfer_to_device,
    src_addr: 0x00007f5b20088280, src_device_num: -10, dest_addr: 0xffffc001ffd80000,
    dest_device_num: 0, bytes: 131072, codeptr_ra: 0x00007f5b26fa47e0
  }
clEnqueueMemcpyINTEL_entry:
  {
    command_queue: 0x181a540, blocking: CL_FALSE,
    dst_ptr: 0xffffc001ffd80000, src_ptr: 0x00007f5b20088280, size: 64, num_events_in_wait_list: 0,
    event_wait_list: 0x0, event: 0x7ffc4ac01378, event_wait_list_vals: []
  }
clEnqueueMemcpyINTEL_exit:
  {
    errcode_ret_val: CL_SUCCESS, event_val: 0x1dffb30
  }
ompt_callback_target_data_op_intel:
  {
    endpoint: ompt_scope_end, target_id: 1, host_op_id: 7, optype:
      ompt_target_data_transfer_to_device,
    src_addr: 0x00007f5b20088280, src_device_num: -10, dest_addr: 0xffffc001ffd80000,
    dest_device_num: 0, bytes: 131072, codeptr_ra: 0x00007f5b26fa47e0
  }
```
Tally

```
$iprof ./target_teams_distribute_parallel_do.out # Using Level0 backend of OpenMP
TRACE:
Trace location: /home/tapplencourt/lttng-traces/iprof-20210408-204629

<table>
<thead>
<tr>
<th>Backend</th>
<th>Hostnames</th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NAME</td>
<td>TIME</td>
<td>TIME(%)</td>
<td>CALLS</td>
</tr>
<tr>
<td>ompt_target</td>
<td>3.65ms</td>
<td>100.00%</td>
<td>1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>3.65ms</td>
<td>100.00%</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backend</th>
<th>Hostnames</th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NAME</td>
<td>TIME</td>
<td>TIME(%)</td>
<td>CALLS</td>
</tr>
<tr>
<td>ompt_target_data_alloc</td>
<td>1.97ms</td>
<td>54.19%</td>
<td>4</td>
</tr>
<tr>
<td>ompt_target_data_transfer_to_device</td>
<td>1.26ms</td>
<td>34.63%</td>
<td>5</td>
</tr>
<tr>
<td>ompt_target_data_transfer_from_device</td>
<td>250.76us</td>
<td>6.91%</td>
<td>1</td>
</tr>
<tr>
<td>ompt_target_submit_intel</td>
<td>155.04us</td>
<td>4.27%</td>
<td>1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>3.65ms</td>
<td>100.00%</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backend</th>
<th>Hostnames</th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZE</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NAME</td>
<td>TIME</td>
<td>TIME(%)</td>
<td>CALLS</td>
</tr>
<tr>
<td>zeModuleCreate</td>
<td>846.26ms</td>
<td>96.89%</td>
<td>1</td>
</tr>
<tr>
<td>zeCommandListAppendMemoryCopy</td>
<td>10.73ms</td>
<td>1.23%</td>
<td>12</td>
</tr>
<tr>
<td>TOTAL</td>
<td>873.46ms</td>
<td>100.00%</td>
<td>117</td>
</tr>
</tbody>
</table>

Device profiling | Hostnames | Processes | Threads | Devices |
| NAME                | TIME                  | TIME(%)               | CALLS | AVERAGE | MIN | MAX |
| zeMemoryCopy(DM)     | 64.48us               | 7.14%               | 1     | 64.48us | 64.48us | 64.48us |
| _omp_target_data_alloc | 27.84us       | 3.08%               | 1     | 27.84us | 27.84us | 27.84us |
| TOTAL                | 902.72us               | 100.00%              | 13     | 902.72us | 902.72us | 902.72us |
```
## Timeline

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1663919851.2</td>
<td>exaerf-sdcloud-pvc11.jf.intel.com</td>
</tr>
<tr>
<td>173</td>
<td>exaerf-sdcloud-pvc11.jf.intel.com</td>
</tr>
<tr>
<td>zeCommandQueue 1</td>
<td>zeMemoryCopy(MD)</td>
</tr>
<tr>
<td>zeCommandQueue 2</td>
<td></td>
</tr>
<tr>
<td>1663919851.2</td>
<td>exaerf-sdcloud-pvc11.jf.intel.com</td>
</tr>
<tr>
<td>exaerf-sdcloud-pvc11.jf.intel.com</td>
<td>Process 31496</td>
</tr>
<tr>
<td>Thread 31496</td>
<td>omp__omp_target_enter_data</td>
</tr>
<tr>
<td>zeM__zeCommandQueueSynchronize</td>
<td>zeM__zeCommandQueueSynchronize</td>
</tr>
<tr>
<td>83/91</td>
<td>Argonne National Laboratory</td>
</tr>
</tbody>
</table>
Bonus: And where the fun begins

Building an Hip Runtime: One example
• LLVM OpenMP is build on top of HSA
• SYCL is build on top of L0
• Can we build Hip on top of L0?
A detour on Quine and Multi-quine

A quine is a computer program which takes no input and produces a copy of its own source code as its only output.

```python
>>> c = 'c = %r; print(c %% c)'; print(c % c)
c = 'c = %r; print(c %% c)'; print(c % c)
```

Multi-quine

(https://github.com/rvantonder/pentaquine/tree/master/src):

```bash
1 $ python pentaquine.py py | diff pentaquine.py -
2 $ python pentaquine.py cc > pentaquine.c ; cc pentaquine.c
3 $ ./pentaquine cc | diff pentaquine.c -
4 $ ./pentaquine py | diff pentaquine.py -
```
• As a proof of concept lets see how can we mimic Allocate Host
• Real ECP project ChipStart lead by Brice Videau
  https://github.com/CHIP-SPV/chipStar
• In real life, far more complicated. (see source code for more fun)
Allocate Host Memory

```
1. hipError_t hipMallocHost( void **ptr, size_t size) ^^I^^I

Level Zero Equivalent

1. zeMemAllocDevice(ze_context_handle_t hContext, const ze_device_mem_alloc_desc_t *
                      device_desc, size_t size, size_t alignment, ze_device_handle_t hDevice, void *
                      pptr)
```

- Hip implicit initializes and terminates the runtime. Not the case for L0\textsuperscript{40}
- Hip has no explicit concept or device\textsuperscript{41}

\textsuperscript{40} And HSA, and cuda driver
\textsuperscript{41} State machine :(), like Cuda Driver
• Singleton to initialize the level zero runtime\(^{42}\)
• Replicate the HIP state-machine: Create a default context, with a default device\(^{43}\)
• Use this context and device to allocate memory \(^{44}\)

---

\(^{42}\)What about race conditions in multi-threaded?
\(^{43}\)Or all the devices?
\(^{44}\)Alignment? Option to pass to `ze_device_mem_alloc_desc_t`?
// We must only initialize the driver once, even if urPlatformGet() is called multiple times. Declaring the return value as "static" ensures it's only called once.

static ze_result_t ZeResult = ZE_CALL_NOCHECK(zeInit, (0));

Since C++ 11 also check std::call_once and once_flag
Conclusion

- Nothing is impossible
- Can always climb up or down the abstraction ladder
- Hip on top of Level Zero is a valid path
Conclusion
• Layers of Programming Model\textsuperscript{45}
  • Sharing some common concept (loading kernel, async commands, nd-range, queue/stream)
• You may want to use High-Level Language (OpenMP, Kokkos, SYCL) and use interopt to lower-level (L0, Cuda Driver, HSA) if required
• Trace, play with the runtime!\textsuperscript{46}

\textsuperscript{45}Soon to be a complete graph
\textsuperscript{46}And on Intel Platform, just email me
Q&A