Layers for OpenCL

Brice Videau

Argonne National Laboratory

27th April 2021

Application Complexity

More Complex Environment

- Hardware is becoming increasingly heterogeneous,
- Application have adapted by adding additional layers of abstraction between programmers and the hardware they target,
- Sometimes difficult to understand and debug these applications.

Libraries are Windows into Applications

- Applications rely on libraries as bridge to the hardware,
- Libraries present consistent *Application Programming Interfaces* APIs,
- Library APIs define *programming models*,
- Programming Model usage can be *validated*.

Programming Model's Usage Validation

What information is required to validate a programming model's usage? (sorted by increasing cost/complexity)

- What functions of the API are called and when,
- With what parameters they were called, and what was returned,
- To what data were those parameters referring to,
- What was the previous state of the programming model.

Example: validate *malloc/free* usage

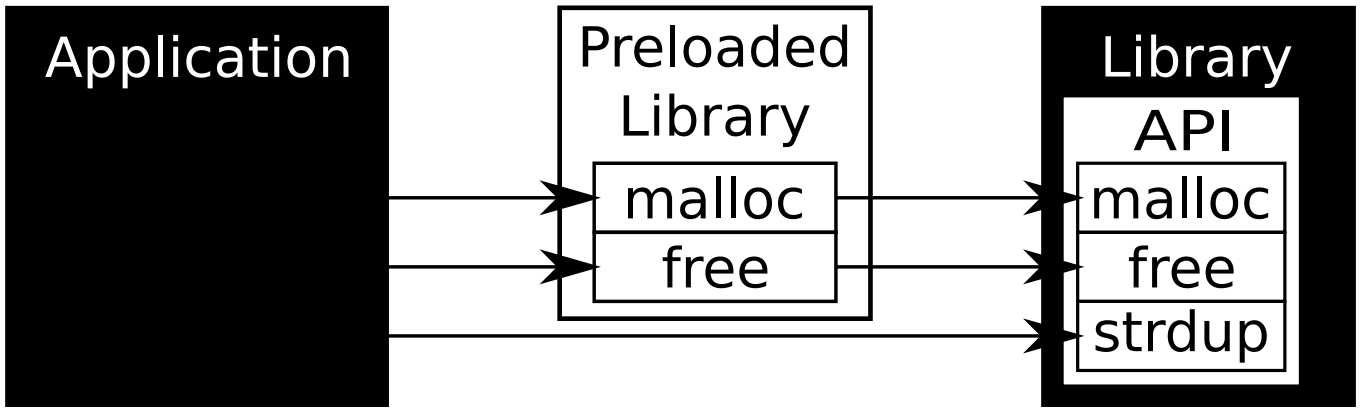
- trace memory allocations and store returned pointers in a set,
- trace memory release, check that pointer is in the set and remove it.

Can these goals be achieved outside the application?
Can we modify the behavior of the programming model?

Library Interception

For shared libraries, *interception* (or interposition) is usually used.

- Create a shared library that implements functions to be traced or modified
- use the real library underneath (several strategies exist here),
- Use the dynamic linker to *preload* this library.



Famous Example: **Intel**'s Intercept Layer for OpenCL Applications

Library Interception: Limitations

Several limitations come with interception:

- Intercepting calls depends on platform specific functionalities,
- Chaining interception libraries can prove difficult if they don't follow the same protocol,
- Depending on the methods used, there can be quite a lot of boiler-plate code involved.

Can we alleviate some of these issues?

Layers as Plugins

What if this interception could be provided as a plugin by the library itself?
Examples: Layers for the Vulkan API, PMPI, OMPT, ...

Benefits

- Factoring of platform specific concerns,
- Factoring of boilerplate code,
- Clear protocol to load layers,
- Standardized API for layer development.

Design and implement **Layers for OpenCL**.

Design and Implementation

How does the OpenCL *Installable Client Driver* (ICD) Loader work?

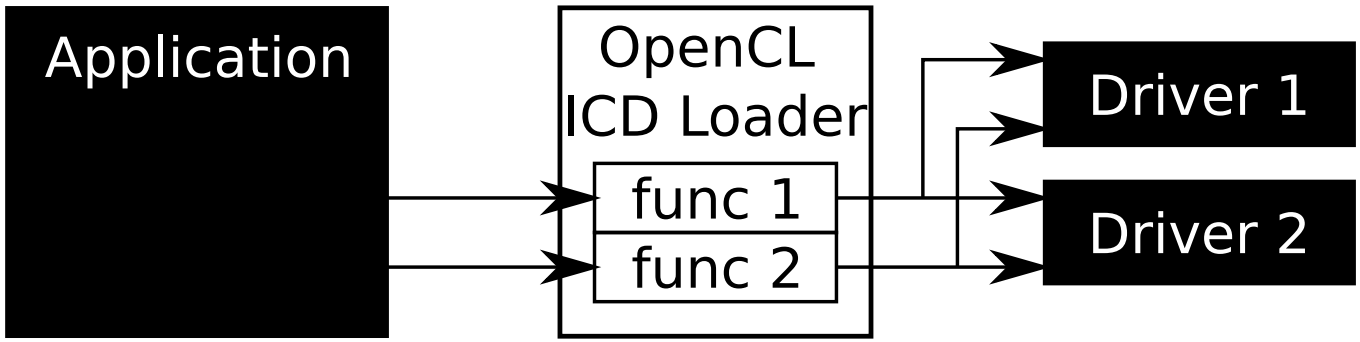
What is the OpenCL ICD Loader?

The OpenCL Installable Client Driver Loader (ICD-Loader) is the library that:

- Allows using several OpenCL driver from different vendors through the same interface,
- Takes care of loading and verifying those drivers (they should implement `cl_khr_icd`),
- Offers options for users to load specific drivers,
- On Unix derivatives it is named `libOpenCL`, with a suffix that depends on the platform (`.so`, `.dylib`, ...). On Windows it is named `OpenCL.dll`,

Original OpenCL ICD Loader API Call Workflow

The OpenCL ICD loader is a router for OpenCL calls, directing them to the correct vendor driver:



In the Beginning Was the Dispatch Table...

In order to support several implementations simultaneously, OpenCL Object are required to conform to this definition:

```
1  /* Dispatch table providing pointers to vendor
2   * implementation of the OpenCL API
3   */
4  typedef struct _cl_icd_dispatch {
5    /* OpenCL 1.0 */
6    cl_api_clGetPlatformIDs clGetPlatformIDs;
7    cl_api_clGetPlatformInfo clGetPlatformInfo;
8    cl_api_clGetDeviceIDs clGetDeviceIDs;
9    cl_api_clGetDeviceInfo clGetDeviceInfo;
10   /* ... */
11 }
12 struct _cl_device_id { cl_icd_dispatch *dispatch; };
13 typedef struct _cl_device_id *cl_device_id;
```

Objects carry around references (function pointers) to the functions (API entry points) that need to be used with those objects.

Original OpenCL Loader API Dispatch Example

The OpenCL ICD Loader's job is to:

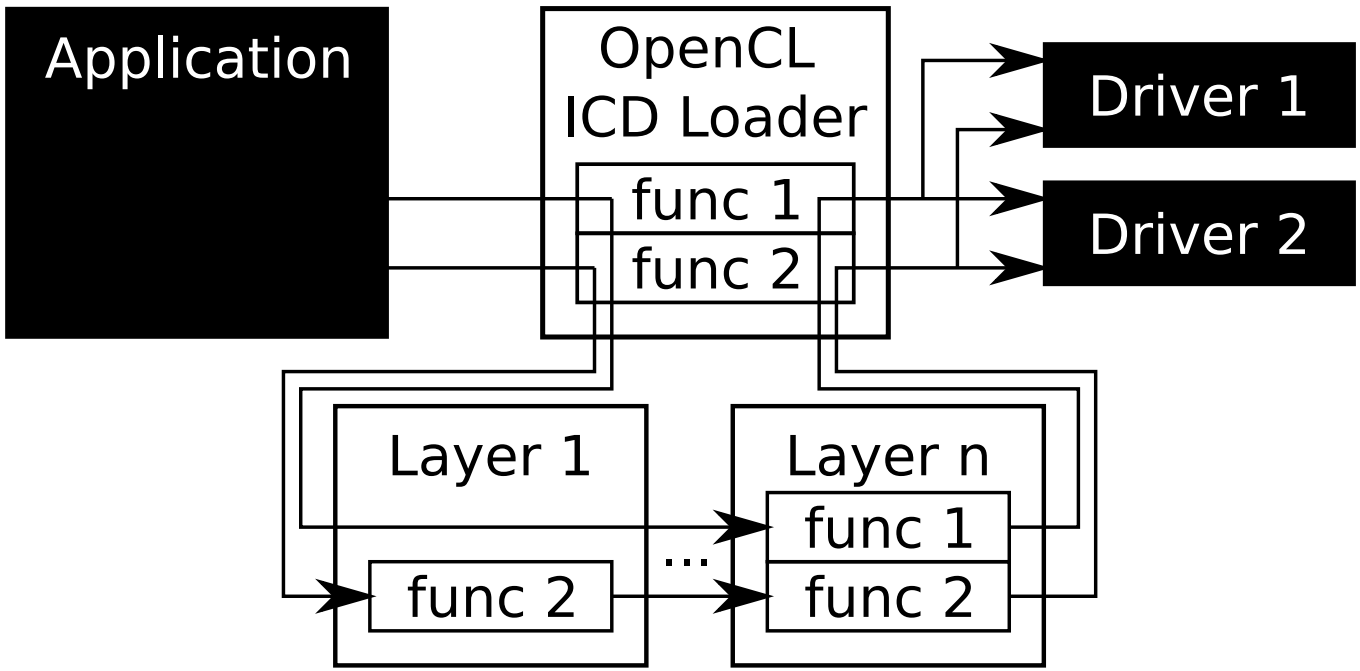
- check for *NULL* handle,
- call the correct function in the dispatch table.

```
1  cl_int clGetDeviceInfo(cl_device_id device,
2     cl_device_info param_name, size_t param_value_size,
3     void* param_value, size_t* param_value_size_ret)
4  {
5    if (!device) return CL_INVALID_DEVICE;
6    return device->dispatch->clGetDeviceInfo(device,
7      param_name, param_value_size,
8      param_value, param_value_size_ret);
9  }
```

OpenCL ICD Loader with Layers API Call Workflow

With layer support enabled, the OpenCL ICD loader will:

- first redirect calls to the different active layers,
- then dispatch the call to the correct vendor driver.



Loader Layers Implementation Idea

How can this scheme be implemented?

Constraints:

- No undue overhead when Layers are not used,
- Chain as many layers as required,
- Layers should be easy to write and understand.

Idea (layer side):

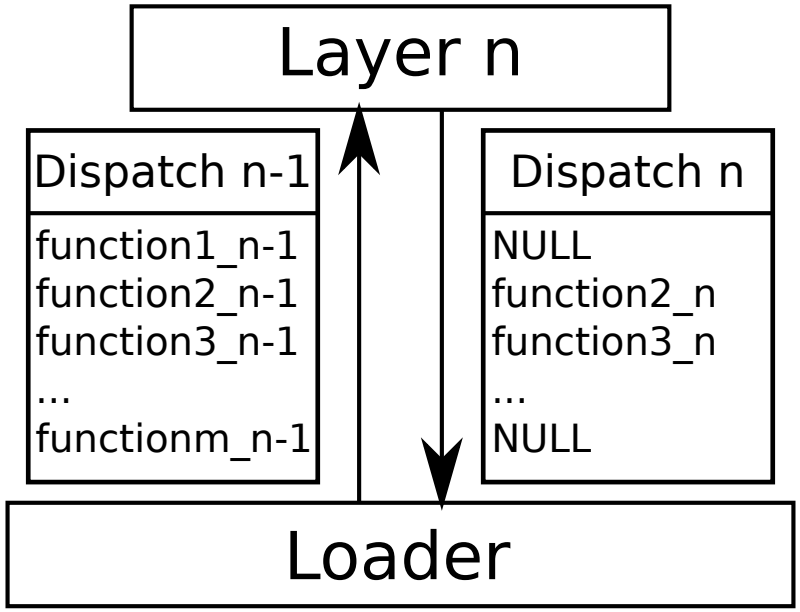
- Layers will be given a complete dispatch table to call into,
- Any OpenCL call a layer will make will have to be through this table,
- Layers will provide a (potentially incomplete) dispatch table of the functions they implement to the loader.

Layers Implementation Idea Continued

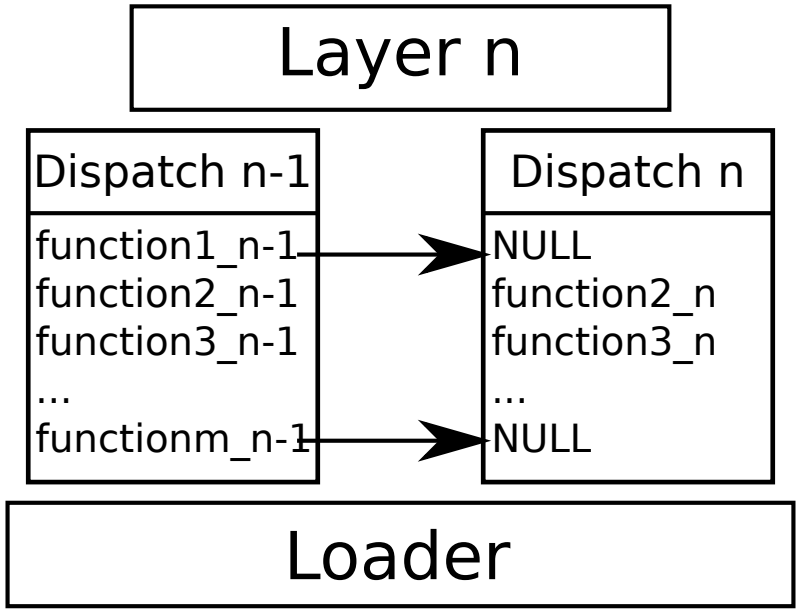
Idea (loader side):

- The loader will maintain a list of successfully loaded layers,
- Each new layer is added to the front of the list,
- Each layer is passed the previous layer dispatch table as a target, or the loader internal dispatch table if it is the first,
- Incomplete layer tables are completed with their target entries.

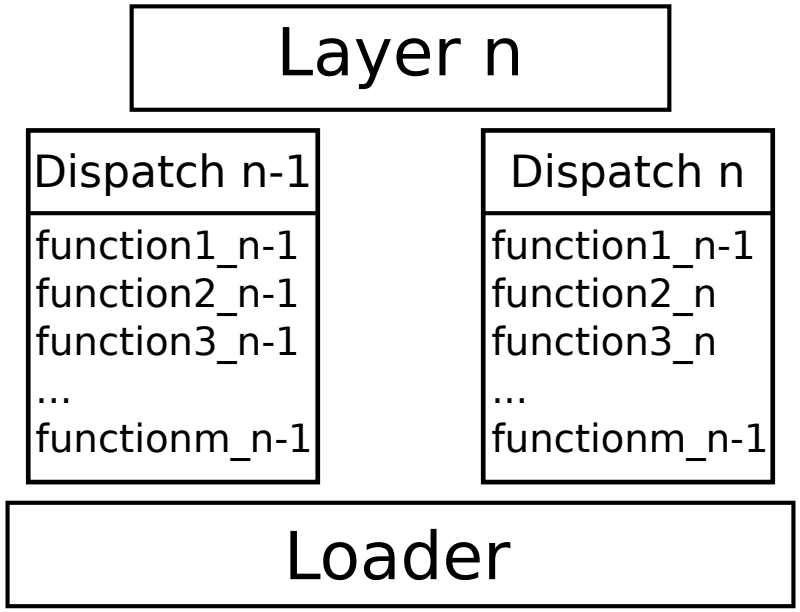
Loader Layers Implementation Illustration



Loader Layers Implementation Illustration



Loader Layers Implementation Illustration



Original OpenCL Loader API Dispatch Example

The OpenCL ICD Loader's job is now to:

- check *if* there are layers, and *then* forward to the first layer,
- *else*, check for *NULL* handle and call the correct function in the dispatch table.

```
1  cl_int clGetDeviceInfo(cl_device_id device,
2     cl_device_info param_name, size_t param_value_size,
3     void* param_value, size_t* param_value_size_ret)
4  {
5    if (khrFirstLayer)
6      return khrFirstLayer->dispatch.clGetDeviceInfo(
7        device, param_name, param_value_size,
8        param_value, param_value_size_ret);
9    if (!device) return CL_INVALID_DEVICE;
10   return device->dispatch->clGetDeviceInfo(device,
11     param_name, param_value_size,
12     param_value, param_value_size_ret);
13 }
```

The cost of layers when there is no active layer is 1 pointer check.

Layer Development and Usage

OpenCL Layer Plugin

- An OpenCL layer plugin is a shared library,
- It must export 2 function symbols:
  - `clGetLayerInfo`
  - `clInitLayer`
- `clGetLayerInfo` is used to query information about the layer, for now it is limited to the version of the layer API that is supported,
- `clInitLayer` is used to negotiate between the loader and the layer and exchange dispatch tables. Especially, if the loader dispatch table is too small, the layer can refuse to initialize. This function can use the provided dispatch table to make calls to OpenCL if it needs to do so.
- The formal extension is called `cl_loader_layers` and can be found here: cl_loader_layers.asciidoc

clGetLayerInfo API

```
1  typedef cl_uint cl_layer_info;
2  typedef cl_uint cl_layer_api_version;
3  #define CL_LAYER_API_VERSION        0x4240
4  #define CL_LAYER_API_VERSION_100 100
5  cl_int clGetLayerInfo(cl_layer_info  param_name,
6                        size_t         param_value_size,
7                        void           *param_value,
8                        size_t         *param_value_size_ret);
```

- `param_name`: Name of the requested parameter (e.g. `CL_LAYER_API_VERSION`)
- `param_value_size`: Size in bytes of memory pointed to by `param_value`
- `param_value`: Pointer to the variable where the info should be stored
- `param_value_size_ret`: Optional pointer to a variable to query the info size

`Returns`:

- `CL_SUCCESS` on success
- `CL_INVALID_VALUE` if `param_name` is not one of the supported values or if size in bytes specified by `param_value_size` is < size of return type and `param_value` is not a NULL value.

clGetLayerInfo API Implementation

```
 8  #include <CL/cl_layer.h>
 9
10  CL_API_ENTRY cl_int CL_API_CALL clGetLayerInfo(
11      cl_layer_info  param_name,
12      size_t         param_value_size,
13      void          *param_value,
14      size_t        *param_value_size_ret) {
15    switch (param_name) {
16    case CL_LAYER_API_VERSION:
17      if (param_value) {
18        if (param_value_size < sizeof(cl_layer_api_version))
19          return CL_INVALID_VALUE;
20        *((cl_layer_api_version *)param_value) = CL_LAYER_API_VERSION_100;
21      }
22      if (param_value_size_ret)
23        *param_value_size_ret = sizeof(cl_layer_api_version);
24      break;
25    default:
26      return CL_INVALID_VALUE;
27    }
28    return CL_SUCCESS;
29  }
```

clInitLayer API

```
1  cl_int clInitLayer(cl_uint                  num_entries,
2                     const cl_icd_dispatch  *target_dispatch,
3                     cl_uint                *num_entries_ret,
4                     const cl_icd_dispatch **layer_dispatch_ret);
```

- num_entries: number of entries in the dispatch table provided by the loader
- target_dispatch: pointer to the dispatch table, provided by the loader, that the layer must redirect it's call to
- num_entries_ret: address to store the number of entries in the layer dispatch table
- layer_dispatch_ret: address to store the pointer to the layer dispatch table

Returns:

- CL_SUCCESS on success
- CL_INVALID_VALUE if num_entries is insufficient for the layer or if target_dispatch is a NULL value, or num_entries_ret is a NULL value, or layer_dispatch_ret is a NULL value.

clInitLayer API Implementation

```
31  static struct _cl_icd_dispatch dispatch;
32  static const struct _cl_icd_dispatch *tdispatch;
33  static void _init_dispatch();
34  CL_API_ENTRY cl_int CL_API_CALL clInitLayer(
35      cl_uint                         num_entries,
36      const struct _cl_icd_dispatch  *target_dispatch,
37      cl_uint                        *num_entries_out,
38      const struct _cl_icd_dispatch **layer_dispatch_ret) {
39    if (!target_dispatch || !num_entries_out || !layer_dispatch_ret)
40      return CL_INVALID_VALUE;
41    /* Check that the loader does not provide us with a dispatch table
42     * smaller than the one we've been compiled with. */
43    if(num_entries < sizeof(dispatch)/sizeof(dispatch.clGetPlatformIDs))
44      return CL_INVALID_VALUE;
45
46    tdispatch = target_dispatch;
47    _init_dispatch();
48    *layer_dispatch_ret = &dispatch;
49    *num_entries_out = sizeof(dispatch)/sizeof(dispatch.clGetPlatformIDs);
50    return CL_SUCCESS;
51  }
```

Simple Wrapper

Display parameters and results of `clGetPlatformIDs`. Initialize the layer dispatch table.

```
53  #include <stdio.h>
54  static CL_API_ENTRY cl_int CL_API_CALL clGetPlatformIDs_wrap(
55      cl_uint num_entries,
56      cl_platform_id* platforms,
57      cl_uint* num_platforms) {
58    fprintf(stderr, "clGetPlatformIDs(num_entries: %d, platforms: %p, num_platforms: %p)\n",
59            num_entries, platforms, num_platforms);
60    cl_int res = tdispatch->clGetPlatformIDs(num_entries, platforms, num_platforms);
61    fprintf(stderr, "clGetPlatformIDs result: %d", res);
62    if (res == CL_SUCCESS && num_platforms)
63      fprintf(stderr, ", *num_platforms: %d", *num_platforms);
64    fprintf(stderr, "\n");
65    return res;
66  }
67
68  static void _init_dispatch() {
69    dispatch.clGetPlatformIDs = &clGetPlatformIDs_wrap;
70  }
```

This covers the bare minimum code required to create a layer.

Building an OpenCL Layer

- An OpenCL Layer is a shared library that will be opened as a plugin.
- on Linux it can be as simple as:

```
gcc -shared -fPIC -Wall example_layer.c -I/path/to/OpenCL-Headers/ -o libExampleLayer.so
```

- Portable solution using CMake:

```
1  cmake_minimum_required (VERSION 3.1)
2  project (example_layer VERSION 1.0)
3  set (CMAKE_C_STANDARD 99)
4  set (CMAKE_C_STANDARD_REQUIRED ON)
5  option (OPENCL_HEADER_PATH "Path to OpenCL Headers" OFF)
6
7  set (EXAMPLE_LAYER_SOURCES example_layer.c)
8  if (WIN32)
9      list (APPEND EXAMPLE_LAYER_SOURCES example_layer.def)
10 endif ()
11
12 if (OPENCL_HEADER_PATH)
13     include_directories (${OPENCL_HEADER_PATH})
14 endif ()
15 add_library (ExampleLayer MODULE ${EXAMPLE_LAYER_SOURCES})
```

- example_layer.def

```
1  EXPORTS
2  clGetLayerInfo
3  clInitLayer
```

OpenCL ICD Loader Layers Configuration Options

How can Layers be loaded?

- For now through `OPENCL_LAYERS` environment variable,
- Accepts a list of colon (Unixes) or semi-colon (Windows) separated layers,
- Depending on system configuration, you may need to provide a full path,
- Layers are loaded in order, and added in front of the layer chain, so the first listed layer will be the last to be called.

OpenCL ICD Loader Layers Usage Example

- Simple instance with the example layer previously defined and an even simpler layer that print functions called:

```
OPENCL_LAYERS=libExampleLayer.so:libPrintLayer.so clinfo
```

- Excerpt from output:

```
clGetPlatformIDs
clGetPlatformIDs(num_entries: 0, platforms: (nil), num_platforms: 0x7ffd4bc7d140)
clGetPlatformIDs result: 0, *num_platforms: 1
Number of platforms                               1
clGetPlatformIDs
clGetPlatformIDs(num_entries: 1, platforms: 0x5597e98ab4d0, num_platforms: (nil))
clGetPlatformIDs result: 0
clGetPlatformInfo
clGetPlatformInfo
  Platform Name                                   Intel(R) OpenCL HD Graphics
```

- Reference:

```
Number of platforms                               1
  Platform Name                                   Intel(R) OpenCL HD Graphics
```

Conclusion and Perspectives

Conclusion

Simple Experimental Layers for OpenCL

- Experimental implementation provides support for interception and modification,
- Multiple Layers supported,
- Simple API for implementing and configuring,
- Layers should be portable between platforms (recompilation needed).

Perspectives

Develop Layers for OpenCL

- Validation:
  - Akin to Vulkan's validation layers,
  - Handle validation, parameter validation, object lifetime, ...
- Tracing/debugging,
- Compatibility...

Evolve Layer Specification

- Feedback from layer development and community,
- Enrich layer API,
- System-wide configuration options for layers,
- Programmatically controlled layers (in-application)?

Questions

Questions?

Acknowledgement

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.