

# IRIS Internals

Jungwon Kim

IRIS mini workshop 2022

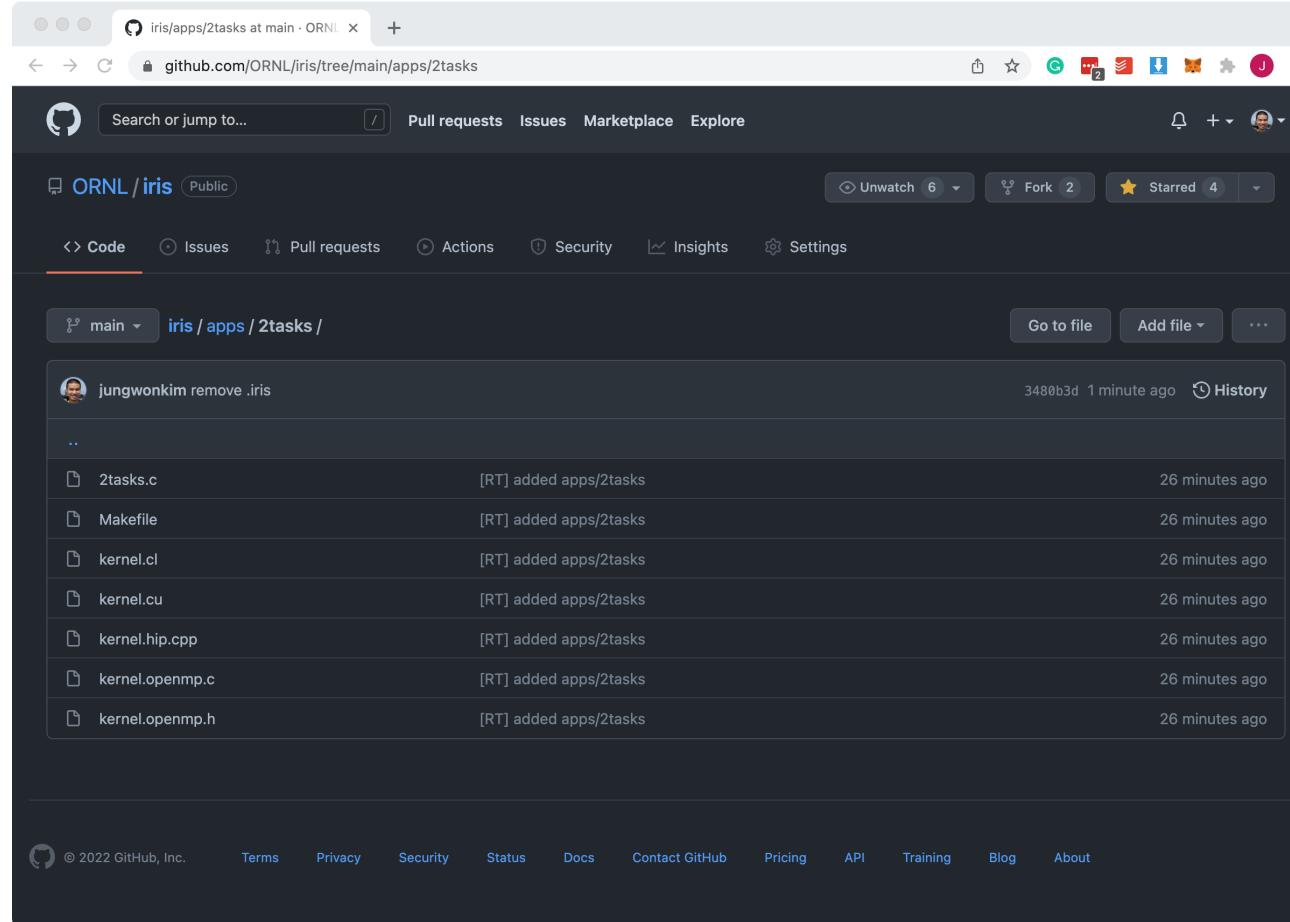
January 4, 2022

# IRIS

- A unified framework across multiple programming platforms
  - <https://iris-programming.github.io/>
- An open-source software under the BSD 3-Clause license
  - <https://github.com/ORNL/iris>
- A user-level library written in C++11
- A node-level programming system

# Our Target Application: apps/2tasks

- <https://github.com/ORNL/iris/tree/main/apps/2tasks>



# Our Target Machine: ExCL/Cousteau

- 2x AMD EPYC 7272 CPUs + 2x AMD MI100 GPUs

```
● ● ●  ✎ 1  ssh  ✎ 1  +
eck@cousteau:~/work/iris/apps/2tasks$ lscpu | grep 'Socket(s)\|Model name'
Socket(s):                                2
Model name:                               AMD EPYC 7272 12-Core Processor
eck@cousteau:~/work/iris/apps/2tasks$ rocm-smi --showhw

===== ROCm System Management Interface =====
===== Concise Hardware Info =====
GPU DID GFX RAS SDMA RAS UMC RAS VBIOS          BUS
0   738c ENABLED ENABLED ENABLED 113-D3430500-030 0000:29:00.0
1   738c ENABLED ENABLED ENABLED 113-D3431500-100 0000:85:00.0
=====
===== End of ROCm SMI Log =====
eck@cousteau:~/work/iris/apps/2tasks$
```

# 2tasks ( $Z[] = X[] + Y[]$ ) on Cousteau

Mem X

Mem Y

Mem Z

**Kernel 0**

```
(float* dst, float* src)
{
    dst[i] = src[i];
}
```

**Kernel 1**

```
(float* dst, float* src)
{
    dst[i] += src[i];
}
```

**Task 0**

H2D(X)

Kernel0(Z, X)

**Task 1**

H2D(Y)

Kernel1(Z, Y)

D2H(Z)

AMD  
CPU  
OMP

AMD  
GPU  
HIP

AMD  
GPU  
HIP

# 2tasks/kernel.hip.cpp

A screenshot of a terminal window titled "kernel.hip.cpp". The window contains C++ code for two kernels, kernel0 and kernel1, using the HIP runtime library. The code includes memory access via pointers dst and src, and calculations involving blockDim and threadIdx.

```
1 #include <hip/hip_runtime.h>
2
3 extern "C" __global__ void kernel0(float* dst, float* src) {
4     int id = blockIdx.x * blockDim.x + threadIdx.x;
5     dst[id] = src[id];
6 }
7
8 extern "C" __global__ void kernel1(float* dst, float* src) {
9     int id = blockIdx.x * blockDim.x + threadIdx.x;
10    dst[id] += src[id];
11 }
12
```

The terminal window also shows the file path "kernel.hip.cpp", the line count "12L", and the character count "301C". The status bar at the bottom indicates the file is a C++ file ("cpp") and shows the current position as "8%".

# 2tasks/2tasks.c



```
2tasks.c
1 #include <iris/iris.h>
2 #include <stdio.h>
3 #include <malloc.h>
4
5 int main(int argc, char** argv) {
6     iris_init(&argc, &argv, 1);
7
8     size_t SIZE = 8;
9     float *X, *Y, *Z;
10
11    X = (float*) malloc(SIZE * sizeof(float));
12    Y = (float*) malloc(SIZE * sizeof(float));
13    Z = (float*) malloc(SIZE * sizeof(float));
14
15    for (int i = 0; i < SIZE; i++) {
16        X[i] = i;
17        Y[i] = i * 10;
18    }
19
20    printf("X [");
21    for (int i = 0; i < SIZE; i++) printf(" %.3f.", X[i]);
22    printf("]\n");
23    printf("Y [");
24    for (int i = 0; i < SIZE; i++) printf(" %.3f.", Y[i]);
25    printf("]\n");
26
27    iris_mem mem_X;
28    iris_mem mem_Y;
29    iris_mem mem_Z;
30    iris_mem_create(SIZE * sizeof(float), &mem_X);
31    iris_mem_create(SIZE * sizeof(float), &mem_Y);
32    iris_mem_create(SIZE * sizeof(float), &mem_Z);
33
```

Mem X

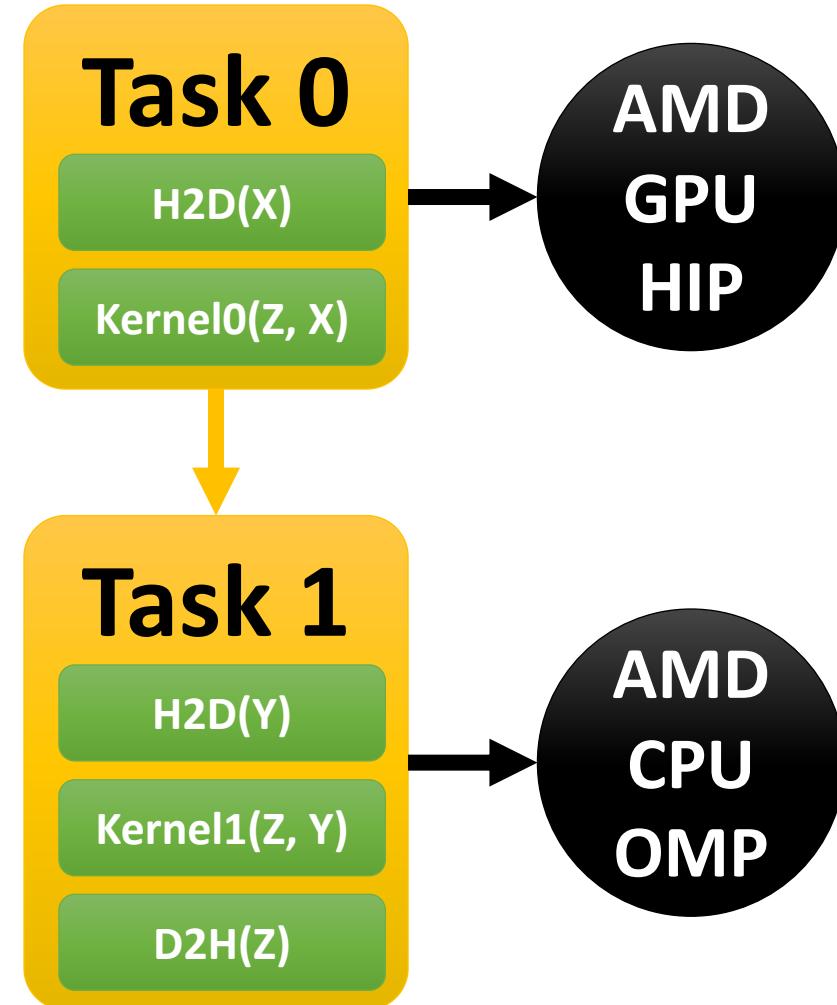
Mem Y

Mem Z

# 2tasks/2tasks.c

```
2tasks.c buffers
ssh
2tasks.c
34  iris_task task0;
35  iris_task_create(&task0);
36  iris_task_h2d_full(task0, mem_X, X);
37  void* task0_params[2] = { mem_Z, mem_X };
38  int task0_params_info[2] = { iris_w, iris_r };
39  iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40  iris_task_submit(task0, iris_gpu, NULL, 1);
41
42  iris_task task1;
43  iris_task_create(&task1);
44  iris_task_h2d_full(task1, mem_Y, Y);
45  void* task1_params[2] = { mem_Z, mem_Y };
46  int task1_params_info[2] = { iris_rw, iris_r };
47  iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48  iris_task_d2h_full(task1, mem_Z, Z);
49  iris_task_submit(task1, iris_cpu, NULL, 1);
50
51  printf("Z [ ");
52  for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53  printf("]\n");
54
55  iris_task_release(task0);
56  iris_task_release(task1);
57
58  iris_mem_release(mem_X);
59  iris_mem_release(mem_Y);
60  iris_mem_release(mem_Z);
61
62  iris_finalize();
63  return 0;
64 }
```

NORMAL > 2tasks.c c utf-8[unix] 98% N:64/65 ≡ 1

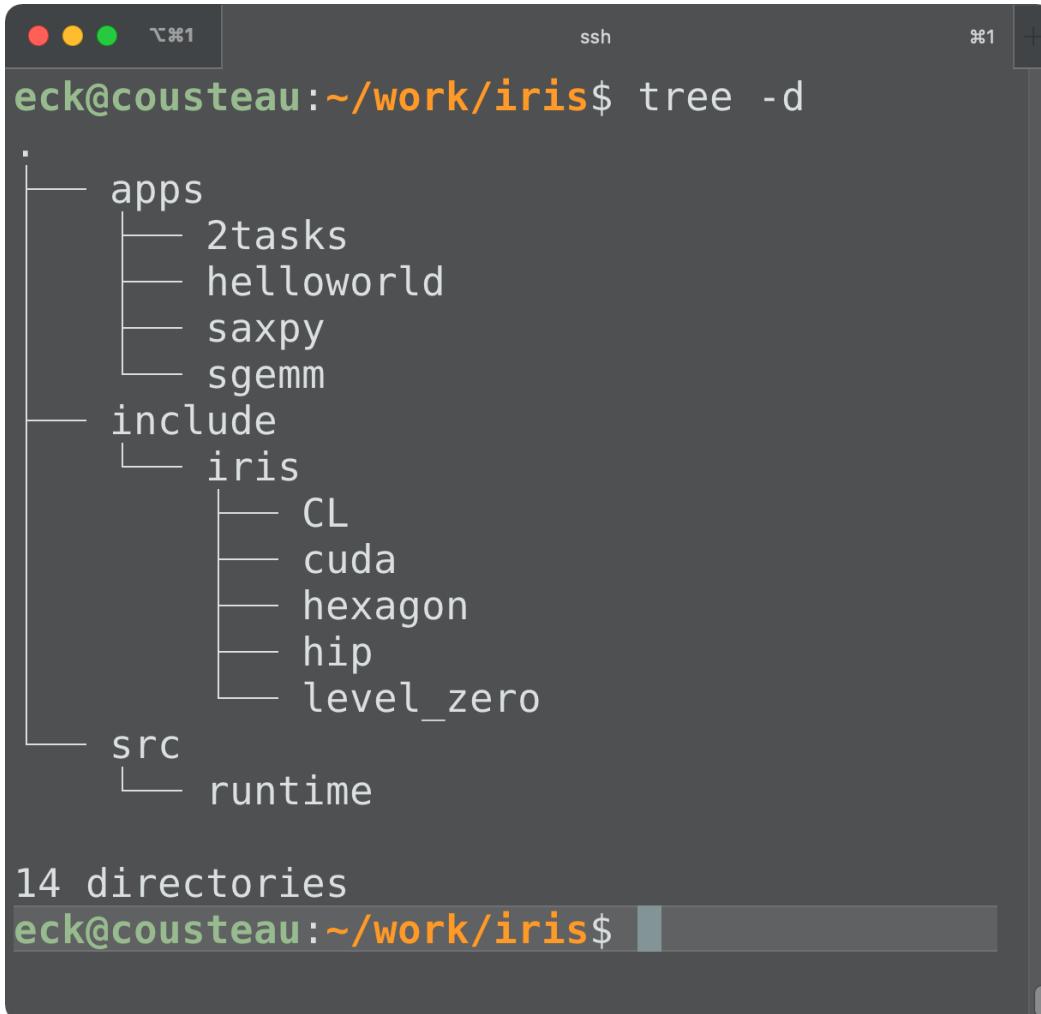


# 2tasks/2tasks.c

$$Z[] = X[] + Y[]$$

```
● ● ●  ↻⌘1 ssh ⌘1 +  
eck@cousteau:~/work/iris/apps/2tasks$ ./2tasks  
X [ 0.  1.  2.  3.  4.  5.  6.  7.]  
Y [ 0. 10. 20. 30. 40. 50. 60. 70.]  
Z [ 0. 11. 22. 33. 44. 55. 66. 77.]  
eck@cousteau:~/work/iris/apps/2tasks$
```

# The IRIS Directory Structure



```
eck@cousteau:~/work/iris$ tree -d
.
├── apps
│   ├── 2tasks
│   ├── helloworld
│   ├── saxpy
│   └── sgemm
└── include
    └── iris
        ├── CL
        ├── cuda
        ├── hexagon
        ├── hip
        └── level_zero
└── src
    └── runtime
14 directories
eck@cousteau:~/work/iris$
```

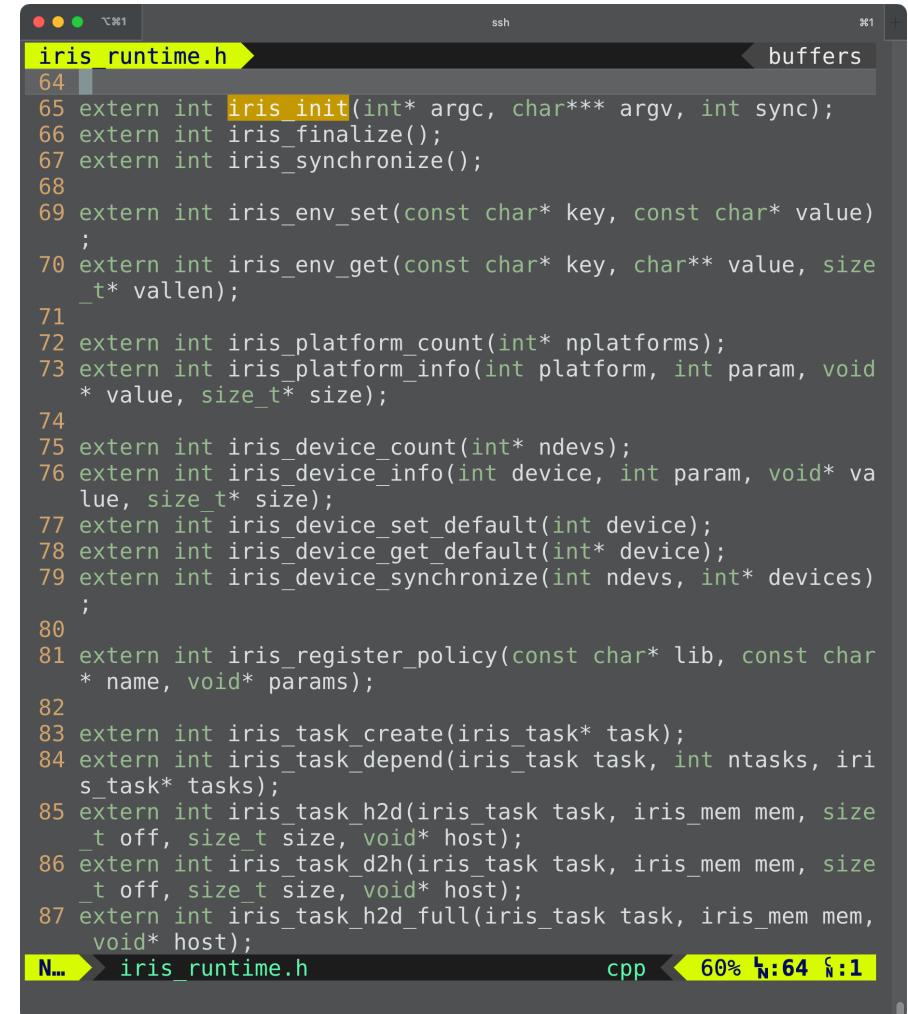
Directory	Contents
apps/*	example applications
<i>include/iris</i>	<i>IRIS API headers (C, C++, Fortran, Python)</i>
include/iris/CL, cuda, hexagon, hip, level_zero	headers of 3rd party programming platforms
<i>src/runtime</i>	<i>IRIS runtime source code</i>

# iris\_init() @ include/iris/iris.h, iris\_runtime.h



```
2tasks.c  buffers
1 #include <iris/iris.h>
2 #include <stdio.h>
3 #include <malloc.h>
4
5 int main(int argc, char** argv) {
6     iris_init(&argc, &argv, 1);
7
8     size_t SIZE = 8;
9     float *X, *Y, *Z;
10
11    X = (float*) malloc(SIZE * sizeof(float));
12    Y = (float*) malloc(SIZE * sizeof(float));
13    Z = (float*) malloc(SIZE * sizeof(float));
14
15    for (int i = 0; i < SIZE; i++) {
16        X[i] = i;
17        Y[i] = i * 10;
18    }
19
20    printf("X [");
21    for (int i = 0; i < SIZE; i++) printf(" %3.0f.", X[i]);
22    printf("]\n");
23    printf("Y [");
24    for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Y[i]);
25    printf("]\n");
26
27    iris_mem mem_X;
28    iris_mem mem_Y;
29    iris_mem mem_Z;
30    iris_mem_create(SIZE * sizeof(float), &mem_X);
31    iris_mem_create(SIZE * sizeof(float), &mem_Y);
32    iris_mem_create(SIZE * sizeof(float), &mem_Z);
33
```

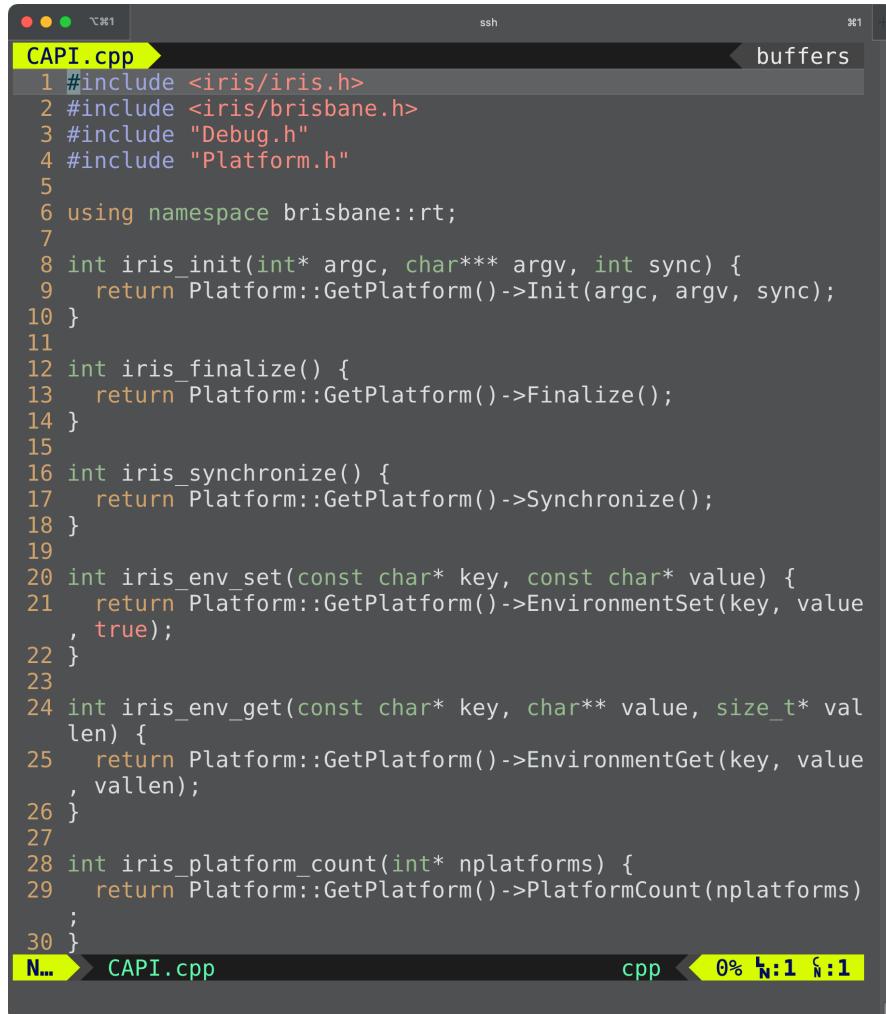
NORMAL 2tasks.c c utf-8[unix] 50% N:33/65 ≡ ⌂



```
iris_runtime.h  buffers
64
65 extern int iris_init(int* argc, char*** argv, int sync);
66 extern int iris_finalize();
67 extern int iris_synchronize();
68
69 extern int iris_env_set(const char* key, const char* value)
70 ;
71 extern int iris_env_get(const char* key, char** value, size
72 _t* vallen);
73 extern int iris_platform_count(int* nplatforms);
74 extern int iris_device_count(int* ndevs);
75 extern int iris_device_info(int device, int param, void*
76 value, size_t* size);
77 extern int iris_device_set_default(int device);
78 extern int iris_device_get_default(int* device);
79 extern int iris_device_synchronize(int ndevs, int* devices)
80 ;
81 extern int iris_register_policy(const char* lib, const char*
82 * name, void* params);
83 extern int iris_task_create(iris_task* task);
84 extern int iris_task_depend(iris_task task, int ntasks, iri
85 s_task* tasks);
86 extern int iris_task_h2d(iris_task task, iris_mem mem, size
87 _t off, size_t size, void* host);
88 extern int iris_task_d2h(iris_task task, iris_mem mem, size
89 _t off, size_t size, void* host);
90 extern int iris_task_h2d_full(iris_task task, iris_mem mem,
91 void* host);
```

N... iris\_runtime.h cpp 60% N:64 ⌂

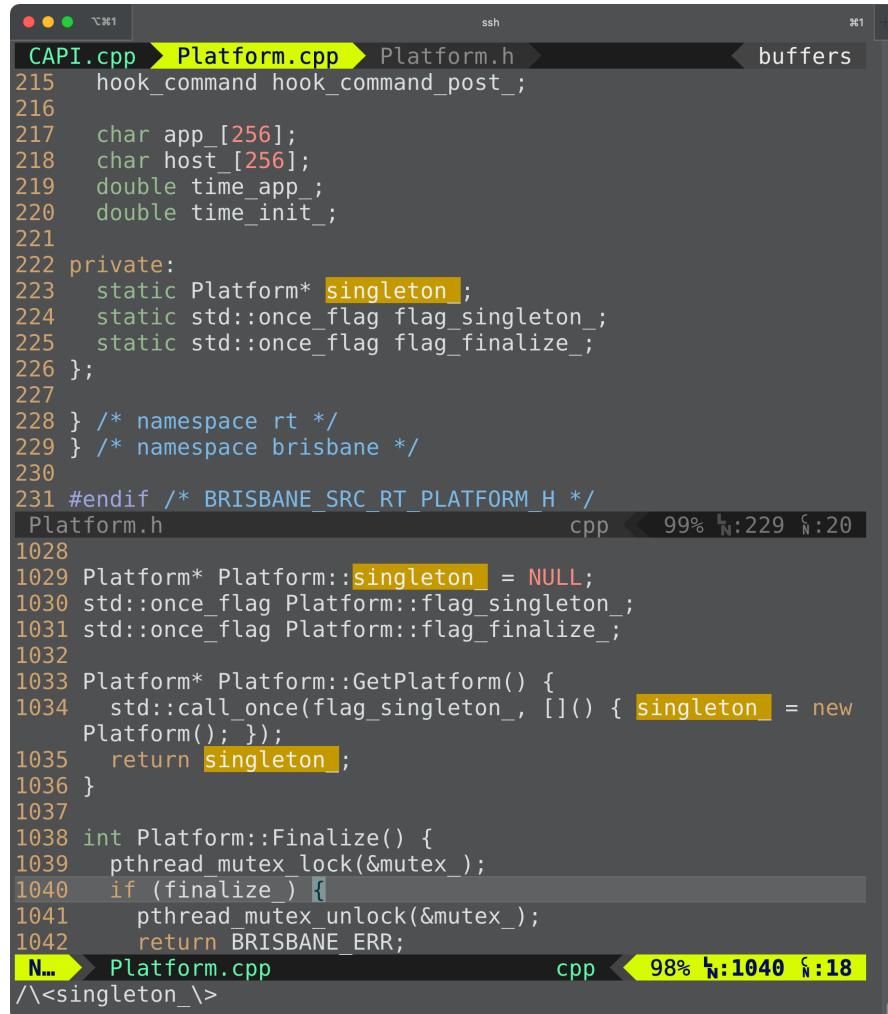
# src/runtime/CAPI.cpp



```
CAPI.cpp
1 #include <iris/iris.h>
2 #include <iris/brisbane.h>
3 #include "Debug.h"
4 #include "Platform.h"
5
6 using namespace brisbane::rt;
7
8 int iris_init(int* argc, char*** argv, int sync) {
9     return Platform::GetPlatform()->Init(argc, argv, sync);
10 }
11
12 int iris_finalize() {
13     return Platform::GetPlatform()->Finalize();
14 }
15
16 int iris_synchronize() {
17     return Platform::GetPlatform()->Synchronize();
18 }
19
20 int iris_env_set(const char* key, const char* value) {
21     return Platform::GetPlatform()->EnvironmentSet(key, value
22 , true);
23 }
24 int iris_env_get(const char* key, char** value, size_t* val
len) {
25     return Platform::GetPlatform()->EnvironmentGet(key, value
26 , vallen);
27 }
28 int iris_platform_count(int* nplatforms) {
29     return Platform::GetPlatform()->PlatformCount(nplatforms)
30 }
```

- All the IRIS C API functions are defined in CAPI.cpp
  - The IRIS C++, Fortran, Python APIs are just wrappers of CAPI.cpp
- Every IRIS C API function calls a member function of Platform

# src/runtime/Platform.cpp

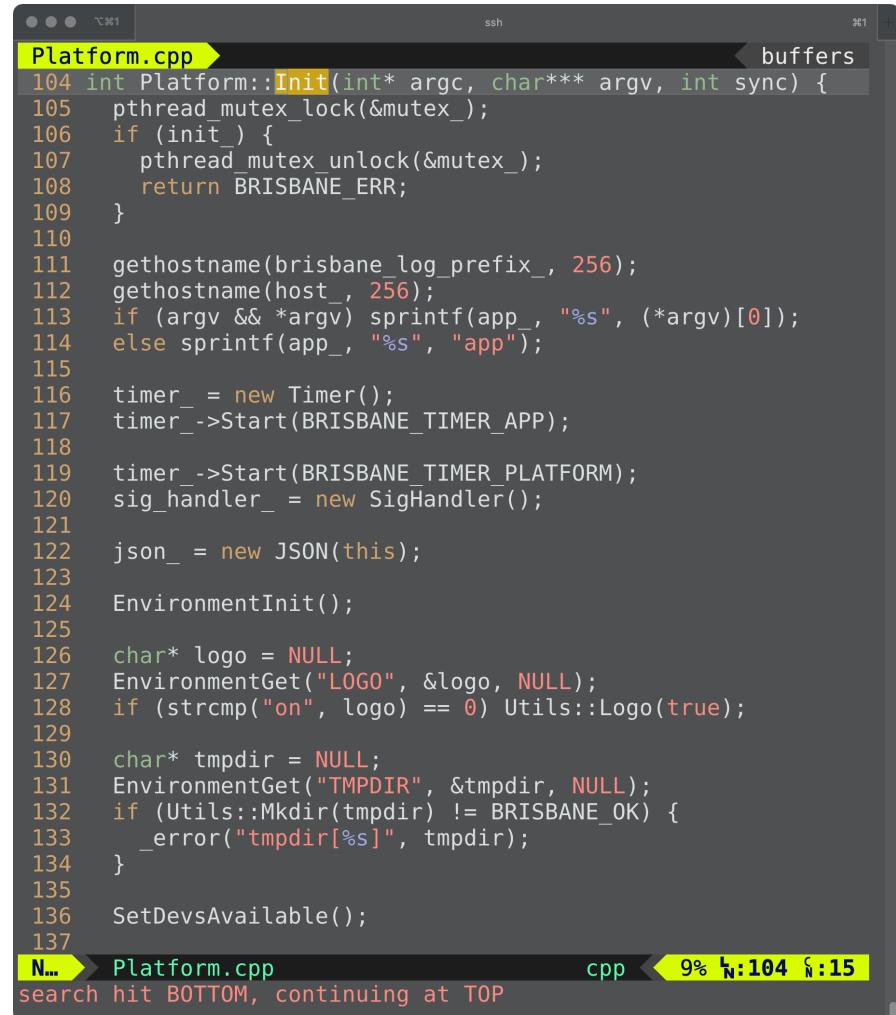


A screenshot of a terminal window displaying the contents of the `Platform.cpp` file. The window has tabs for `CAPI.cpp`, `Platform.cpp` (which is the active tab), and `Platform.h`. The code implements a singleton pattern for the `Platform` class. It includes declarations for `hook_command` and `hook_command_post_`, and defines `app_`, `host_`, `time_app_`, and `time_init_` variables. The `private` section contains the definition of the `singleton` variable as a static pointer to `Platform`, along with `std::once_flag`s for initialization and finalization. The `GetPlatform` method uses `std::call_once` to initialize the singleton if it is null. The `Finalize` method releases the mutex. The code is annotated with `/* BRISBANE_SRC_RT_PLATFORM_H */` at the top and `/\<singleton\_>` at the bottom.

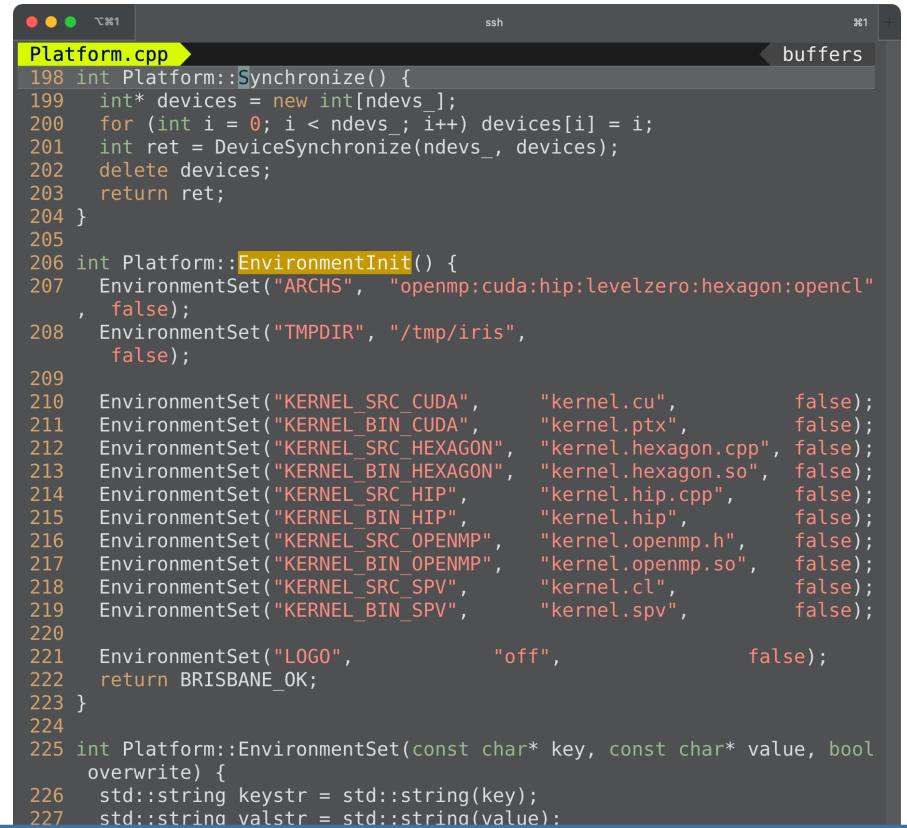
```
CAPI.cpp > Platform.cpp > Platform.h > buffers
215     hook_command hook_command_post_;
216
217     char app_[256];
218     char host_[256];
219     double time_app_;
220     double time_init_;
221
222 private:
223     static Platform* singleton;
224     static std::once_flag flag_singleton;
225     static std::once_flag flag_finalize;
226 };
227
228 } /* namespace rt */
229 } /* namespace brisbane */
230
231 #endif /* BRISBANE_SRC_RT_PLATFORM_H */
Platform.h
Platform.cpp 99% N:229 F:20
1028
1029 Platform* Platform::singleton_ = NULL;
1030 std::once_flag Platform::flag_singleton;
1031 std::once_flag Platform::flag_finalize;
1032
1033 Platform* Platform::GetPlatform() {
1034     std::call_once(flag_singleton_, []() { singleton_ = new Platform(); });
1035     return singleton_;
1036 }
1037
1038 int Platform::Finalize() {
1039     pthread_mutex_lock(&mutex_);
1040     if (finalize_) {
1041         pthread_mutex_unlock(&mutex_);
1042         return BRISBANE_ERR;
1043     }
1044 }
N... Platform.cpp cpp 98% N:1040 F:18
/\<singleton\_>
```

- The IRIS Platform manages everything in the IRIS runtime
  - Devices, memories, kernels, tasks, scheduler, workers, loaders, and etc
- Platform is a singleton instance in the whole IRIS execution environment

# src/runtime/Platform.cpp :: Init()



```
Platform.cpp buffers
104 int Platform::Init(int* argc, char*** argv, int sync) {
105     pthread_mutex_lock(&mutex_);
106     if (init_) {
107         pthread_mutex_unlock(&mutex_);
108         return BRISBANE_ERR;
109     }
110
111     gethostname(brisbane_log_prefix_, 256);
112     gethostname(host_, 256);
113     if (argv && *argv) sprintf(app_, "%s", (*argv)[0]);
114     else sprintf(app_, "%s", "app");
115
116     timer_ = new Timer();
117     timer_->Start(BRISBANE_TIMER_APP);
118
119     timer_->Start(BRISBANE_TIMER_PLATFORM);
120     sig_handler_ = new SigHandler();
121
122     json_ = new JSON(this);
123
124     EnvironmentInit();
125
126     char* logo = NULL;
127     EnvironmentGet("LOGO", &logo, NULL);
128     if (strcmp("on", logo) == 0) Utils::Logo(true);
129
130     char* tmpdir = NULL;
131     EnvironmentGet("TMPDIR", &tmpdir, NULL);
132     if (Utils::Mkdir(tmpdir) != BRISBANE_OK) {
133         _error("tmpdir[%s]", tmpdir);
134     }
135
136     SetDevsAvailable();
137
N... Platform.cpp      cpp 9% h:104 h:15
search hit BOTTOM, continuing at TOP
```



```
Platform.cpp buffers
198 int Platform::Synchronize() {
199     int* devices = new int[ndeps_];
200     for (int i = 0; i < ndeps_; i++) devices[i] = i;
201     int ret = DeviceSynchronize(ndeps_, devices);
202     delete devices;
203     return ret;
204 }
205
206 int Platform::EnvironmentInit() {
207     EnvironmentSet("ARCHS", "openmp:cuda:hip:levelzero:hexagon:opencl",
208                     false);
209     EnvironmentSet("TMPDIR", "/tmp/iris",
210                     false);
211
212     EnvironmentSet("KERNEL_SRC_CUDA", "kernel.cuda", false);
213     EnvironmentSet("KERNEL_BIN_CUDA", "kernel.ptx", false);
214     EnvironmentSet("KERNEL_SRC_HEXAGON", "kernel.hexagon.cpp", false);
215     EnvironmentSet("KERNEL_BIN_HEXAGON", "kernel.hexagon.so", false);
216     EnvironmentSet("KERNEL_SRC_HIP", "kernel.hip.cpp", false);
217     EnvironmentSet("KERNEL_BIN_HIP", "kernel.hip", false);
218     EnvironmentSet("KERNEL_SRC_OPENMP", "kernel.openmp.h", false);
219     EnvironmentSet("KERNEL_BIN_OPENMP", "kernel.openmp.so", false);
220     EnvironmentSet("KERNEL_SRC_SPV", "kernel.cl", false);
221     EnvironmentSet("KERNEL_BIN_SPV", "kernel.spv", false);
222
223     EnvironmentSet("LOGO", "off", false);
224
225     int Platform::EnvironmentSet(const char* key, const char* value, bool
226                                 overwrite) {
227         std::string keystr = std::string(key);
228         std::string valstr = std::string(value);
```

You can overwrite the IRIS environment variables by

- `$ export IRIS_ARCHS=hip:openmp`
- `iris_env_set("ARCHS", "hip:openmp");`

# src/runtime/Platform.cpp :: InitHIP()



```
Platform.cpp
138 char* archs = NULL;
139 EnvironmentGet("ARCHS", &archs, NULL);
140 _info("IRIS architectures[%s]", archs);
141 const char* delim = " :.,";
142 char arch_str[128];
143 memset(arch_str, 0, 128);
144 strncpy(arch_str, archs, strlen(archs));
145 char* rest = arch_str;
146 char* a = NULL;
147 while ((a = strtok_r(rest, delim, &rest))) {
148     if (strcasecmp(a, "cuda") == 0) {
149         if (!loaderCUDA_) InitCUDA();
150     } else if (strcasecmp(a, "hip") == 0) {
151         if (!loaderHIP_) InitHIP();
152     } else if (strcasecmp(a, "levelzero") == 0) {
153         if (!loaderLevelZero_) InitLevelZero();
154     } else if (strcasecmp(a, "opencl") == 0) {
155         if (!loaderOpenCL_) InitOpenCL();
156     } else if (strcasecmp(a, "openmp") == 0) {
157         if (!loaderOpenMP_) InitOpenMP();
158     } else if (strcasecmp(a, "hexagon") == 0) {
159         if (!loaderHexagon_) InitHexagon();
160     } else _error("not support arch[%s]", a);
161 }
162 if (ndevs_enabled_ > ndevs_) ndevs_enabled_ = ndevs_;
163 polyhedral_ = new Polyhedral();
164 polyhedral_available_ = polyhedral_->Load() == BRISBANE_
OK;
165 if (polyhedral_available_)
166     filter_task_split_ = new FilterTaskSplit(polyhedral_,
this);
167
168 brisbane_kernel null_brs_kernel;
169 KernelCreate("brisbane_null", &null_brs_kernel);
N... Platform.cpp          cpp      15%  N:168  ⌂:3
```



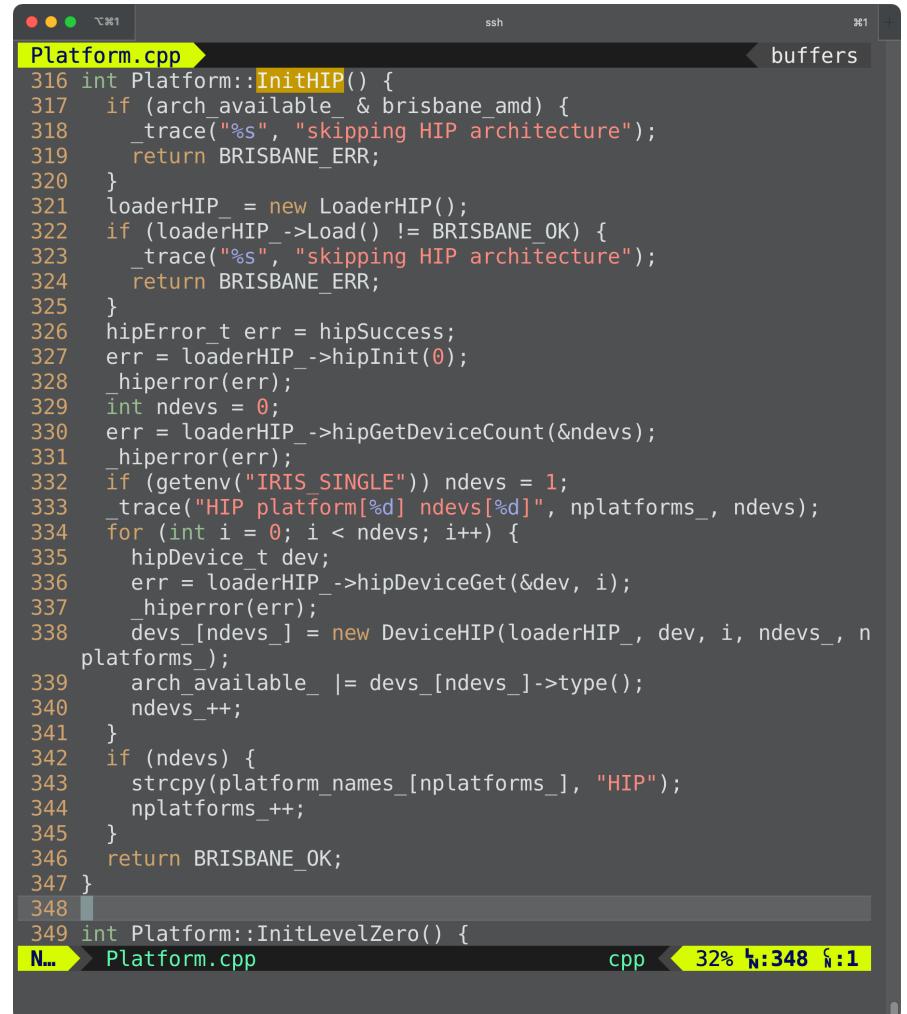
```
Platform.cpp
316 int Platform::InitHIP() {
317     if (arch_available_ & brisbane_amd) {
318         _trace("%s", "skipping HIP architecture");
319         return BRISBANE_ERR;
320     }
321     loaderHIP_ = new LoaderHIP();
322     if (loaderHIP_->Load() != BRISBANE_OK) {
323         _trace("%s", "skipping HIP architecture");
324         return BRISBANE_ERR;
325     }
326     hipError_t err = hipSuccess;
327     err = loaderHIP_->hipInit(0);
328     _hiperror(err);
329     int ndevs = 0;
330     err = loaderHIP_->hipGetDeviceCount(&ndevs);
331     _hiperror(err);
332     if (getenv("IRIS_SINGLE")) ndevs = 1;
333     _trace("HIP platform[%d] ndevs[%d]", nplatforms_, ndevs);
334     for (int i = 0; i < ndevs; i++) {
335         hipDevice_t dev;
336         err = loaderHIP_->hipDeviceGet(&dev, i);
337         _hiperror(err);
338         devs_[ndevs_] = new DeviceHIP(loaderHIP_, dev, i, ndevs_, n
platforms_);
339         arch_available_ |= devs_[ndevs_]->type();
340         ndevs++;
341     }
342     if (ndevs) {
343         strcpy(platform_names_[nplatforms_], "HIP");
344         nplatforms++;
345     }
346     return BRISBANE_OK;
347 }
348
349 int Platform::InitLevelZero() {
N... Platform.cpp          cpp      32%  N:348  ⌂:1
```

# src/runtime/LoaderHIP.cpp



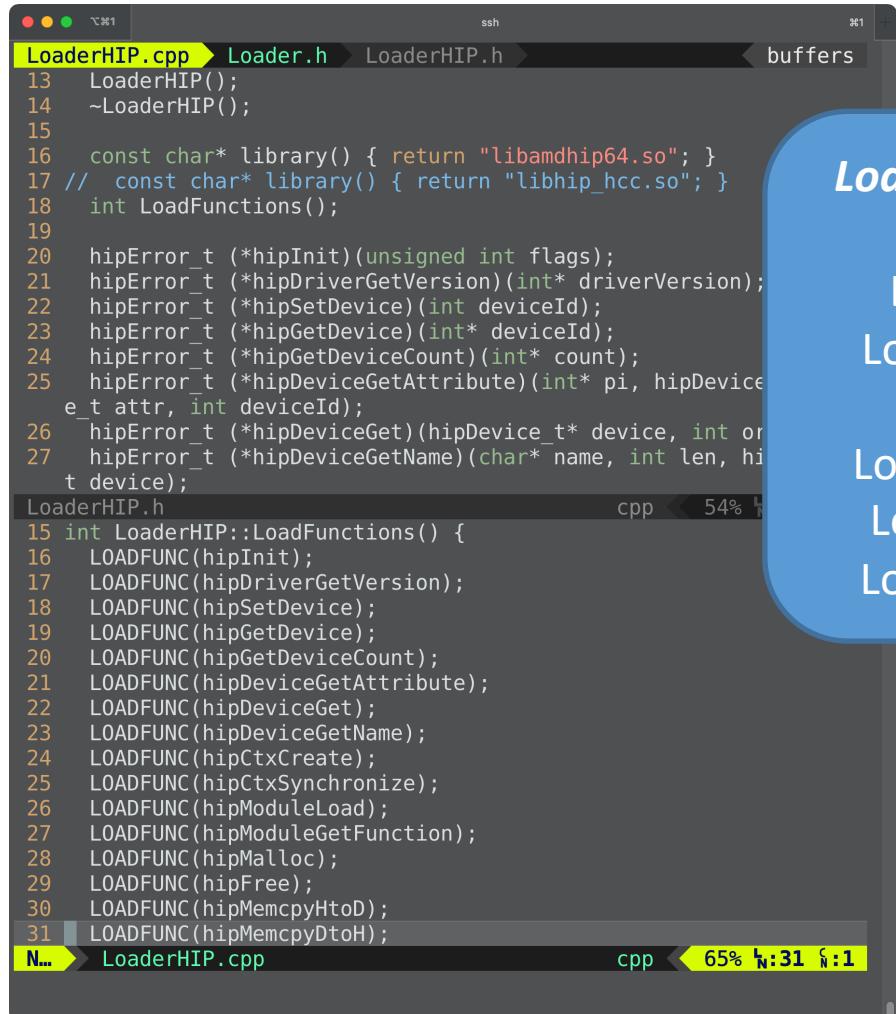
```
LoaderHIP.cpp  Loader.h  LoaderHIP.h  buffers
13  LoaderHIP();
14  ~LoaderHIP();
15
16  const char* library() { return "libamdhip64.so"; }
17 // const char* library() { return "libhip_hcc.so"; }
18  int LoadFunctions();
19
20  hipError_t (*hipInit)(unsigned int flags);
21  hipError_t (*hipDriverGetVersion)(int* driverVersion);
22  hipError_t (*hipSetDevice)(int deviceId);
23  hipError_t (*hipGetDevice)(int* deviceId);
24  hipError_t (*hipGetDeviceCount)(int* count);
25  hipError_t (*hipDeviceGetAttribute)(int* pi, hipDevice_
e_t attr, int deviceId);
26  hipError_t (*hipDeviceGet)(hipDevice_t* device, int ordinal);
27  hipError_t (*hipDeviceGetName)(char* name, int len, hipDevice_
t device);
LoaderHIP.h          cpp      54%  1:27  1:1
15 int LoaderHIP::LoadFunctions() {
16  LOADFUNC(hipInit);
17  LOADFUNC(hipDriverGetVersion);
18  LOADFUNC(hipSetDevice);
19  LOADFUNC(hipGetDevice);
20  LOADFUNC(hipGetDeviceCount);
21  LOADFUNC(hipDeviceGetAttribute);
22  LOADFUNC(hipDeviceGet);
23  LOADFUNC(hipDeviceGetName);
24  LOADFUNC(hipCtxCreate);
25  LOADFUNC(hipCtxSynchronize);
26  LOADFUNC(hipModuleLoad);
27  LOADFUNC(hipModuleGetFunction);
28  LOADFUNC(hipMalloc);
29  LOADFUNC(hipFree);
30  LOADFUNC(hipMemcpyHtoD);
31  LOADFUNC(hipMemcpyDtoH);
N...  LoaderHIP.cpp          cpp      65%  1:31  1:1
```

Seeking  
*libamdhip64.so* in  
\$LD\_LIBRARY\_PATH  
and loading its HIP  
API functions



```
Platform.cpp  buffers
316 int Platform::InitHIP() {
317  if (arch_available_ & brisbane_amd) {
318    _trace("%s", "skipping HIP architecture");
319    return BRISBANE_ERR;
320  }
321  loaderHIP_ = new LoaderHIP();
322  if (loaderHIP_->Load() != BRISBANE_OK) {
323    _trace("%s", "skipping HIP architecture");
324    return BRISBANE_ERR;
325  }
326  hipError_t err = hipSuccess;
327  err = loaderHIP_->hipInit(0);
328  _hiperror(err);
329  int ndevs = 0;
330  err = loaderHIP_->hipGetDeviceCount(&ndevs);
331  _hiperror(err);
332  if (getenv("IRIS_SINGLE")) ndevs = 1;
333  _trace("HIP platform[%d] ndevs[%d]", nplatforms_, ndevs);
334  for (int i = 0; i < ndevs; i++) {
335    hipDevice_t dev;
336    err = loaderHIP_->hipDeviceGet(&dev, i);
337    _hiperror(err);
338    devs_[ndevs_] = new DeviceHIP(loaderHIP_, dev, i, ndevs_, n
platforms_);
339    arch_available_ |= devs_[ndevs_]->type();
340    ndevs_++;
341  }
342  if (ndevs) {
343    strcpy(platform_names_[nplatforms_], "HIP");
344    nplatforms_++;
345  }
346  return BRISBANE_OK;
347 }
348
349 int Platform::InitLevelZero() {
N...  Platform.cpp          cpp      32%  1:348  1:1
```

# src/runtime/LoaderXXX.cpp



```
LoaderHIP.cpp  Loader.h  LoaderHIP.h  buffers
13 LoaderHIP();
14 ~LoaderHIP();
15
16 const char* library() { return "libamdhip64.so"; }
17 // const char* library() { return "libhip_hcc.so"; }
18 int LoadFunctions();
19
20 hipError_t (*hipInit)(unsigned int flags);
21 hipError_t (*hipDriverGetVersion)(int* driverVersion);
22 hipError_t (*hipSetDevice)(int deviceId);
23 hipError_t (*hipGetDevice)(int* deviceId);
24 hipError_t (*hipGetDeviceCount)(int* count);
25 hipError_t (*hipDeviceGetAttribute)(int* pi, hipDevice_
e_t attr, int deviceId);
26 hipError_t (*hipDeviceGet)(hipDevice_t* device, int or
t device);
27 hipError_t (*hipDeviceGetName)(char* name, int len, hi
t device);
LoaderHIP.h      cpp  54% ↵
15 int LoaderHIP::LoadFunctions() {
16     LOADFUNC(hipInit);
17     LOADFUNC(hipDriverGetVersion);
18     LOADFUNC(hipSetDevice);
19     LOADFUNC(hipGetDevice);
20     LOADFUNC(hipGetDeviceCount);
21     LOADFUNC(hipDeviceGetAttribute);
22     LOADFUNC(hipDeviceGet);
23     LOADFUNC(hipDeviceGetName);
24     LOADFUNC(hipCtxCreate);
25     LOADFUNC(hipCtxSynchronize);
26     LOADFUNC(hipModuleLoad);
27     LOADFUNC(hipModuleGetFunction);
28     LOADFUNC(hipMalloc);
29     LOADFUNC(hipFree);
30     LOADFUNC(hipMemcpyHtoD);
31     LOADFUNC(hipMemcpyDtoH);
N...  LoaderHIP.cpp  cpp  65% ↵ 65% ↵ 31 ↵ 1
```

## Loader Subclasses

LoaderCUDA  
LoaderHexagon  
LoaderHIP  
LoaderLevelZero  
LoaderOpenCL  
LoaderOpenMP



```
LoaderCUDA.cpp  LoaderCUDA.h  buffers
12 LoaderCUDA();
13 ~LoaderCUDA();
14
15 const char* library_precheck() { return "cuInit"; }
16 const char* library() { return "libcuda.so"; }
17 int LoadFunctions();
18
19 CUresult (*cuInit)(unsigned int Flags);
20 CUresult (*cuDriverGetVersion)(int* driverVersion);
21 CUresult (*cuDeviceGet)(CUdevice* device, int ordinal);
22 CUresult (*cuDeviceGetAttribute)(int* pi, CUdevice_attribute a
ttrib, CUdevice dev);
23 CUresult (*cuDeviceGetCount)(int* count);
24 CUresult (*cuDeviceGetName)(char* name, int len, CUdevice dev)
;
25 CUresult (*cuCtxCreate)(CUcontext* pctx, unsigned int flags,CU
device dev);
LoaderCUDA.h      cpp  23% ↵ 12 ↵ 1
13 int LoaderCUDA::LoadFunctions() {
14     LOADFUNC(cuInit);
15     LOADFUNC(cuDriverGetVersion);
16     LOADFUNC(cuDeviceGet);
17     LOADFUNC(cuDeviceGetAttribute);
18     LOADFUNC(cuDeviceGetCount);
19     LOADFUNC(cuDeviceGetName);
20     LOADFUNCSYM(cuCtxCreate, cuCtxCreate_v2);
21     LOADFUNC(cuCtxSynchronize);
22     LOADFUNC(cuStreamAddCallback);
23     LOADFUNC(cuStreamCreate);
24     LOADFUNC(cuStreamSynchronize);
25     LOADFUNC(cuModuleGetFunction);
26     LOADFUNC(cuModuleLoad);
27     LOADFUNC(cuModuleGetTexRef);
28     LOADFUNCSYM(cuTexRefSetAddress, cuTexRefSetAddress_v2);
29     LOADFUNC(cuTexRefSetAddressMode);
N...  LoaderCUDA.cpp  cpp  64% ↵ 29 ↵ 1
```

# src/runtime/DeviceHIP.cpp



```
DeviceHIPP.cpp Device.h buffers
19 class Device {
20 public:
21     Device(int devno, int platform);
22     virtual ~Device();
23
24     virtual void TaskPre(Task* task) { return; }
25     virtual void TaskPost(Task* task) { return; }
26
27     void Execute(Task* task);
28
29     void ExecuteInit(Command* cmd);
30     void ExecuteKernel(Command* cmd);
31     void ExecuteMalloc(Command* cmd);
32     void ExecuteH2D(Command* cmd);
33     void ExecuteH2DNP(Command* cmd);
34     void ExecuteD2H(Command* cmd);
35     void ExecuteMap(Command* cmd);
36
37     DeviceHIP(LoaderHIP* ld, hipDevice_t dev, int nplatforms_, int ndevs_);
38
39     DeviceHIP(LoaderHIP* ld, hipDevice_t dev, int nplatforms_, int ndevs_) : Device(devno, platform) {
40         ld_ = ld;
41         max_arg_idx_ = 0;
42         shared_mem_bytes_ = 0;
43         ordinal_ = ordinal;
44         dev_ = dev;
45         strcpy(vendor_, "Advanced Micro Devices");
46         err_ = ld->hipDeviceGetName(name_, sizeof(name_), dev_);
47         _hiperror(err_);
48         type_ = brisbane_amd;
49         model_ = brisbane_hip;
50         err_ = ld->hipDriverGetVersion(&driver_version_);
51         _hiperror(err_);
52         sprintf(version_, "AMD HIP %d", driver_version_);
53     }
54
55     @
56     @
57 }
```

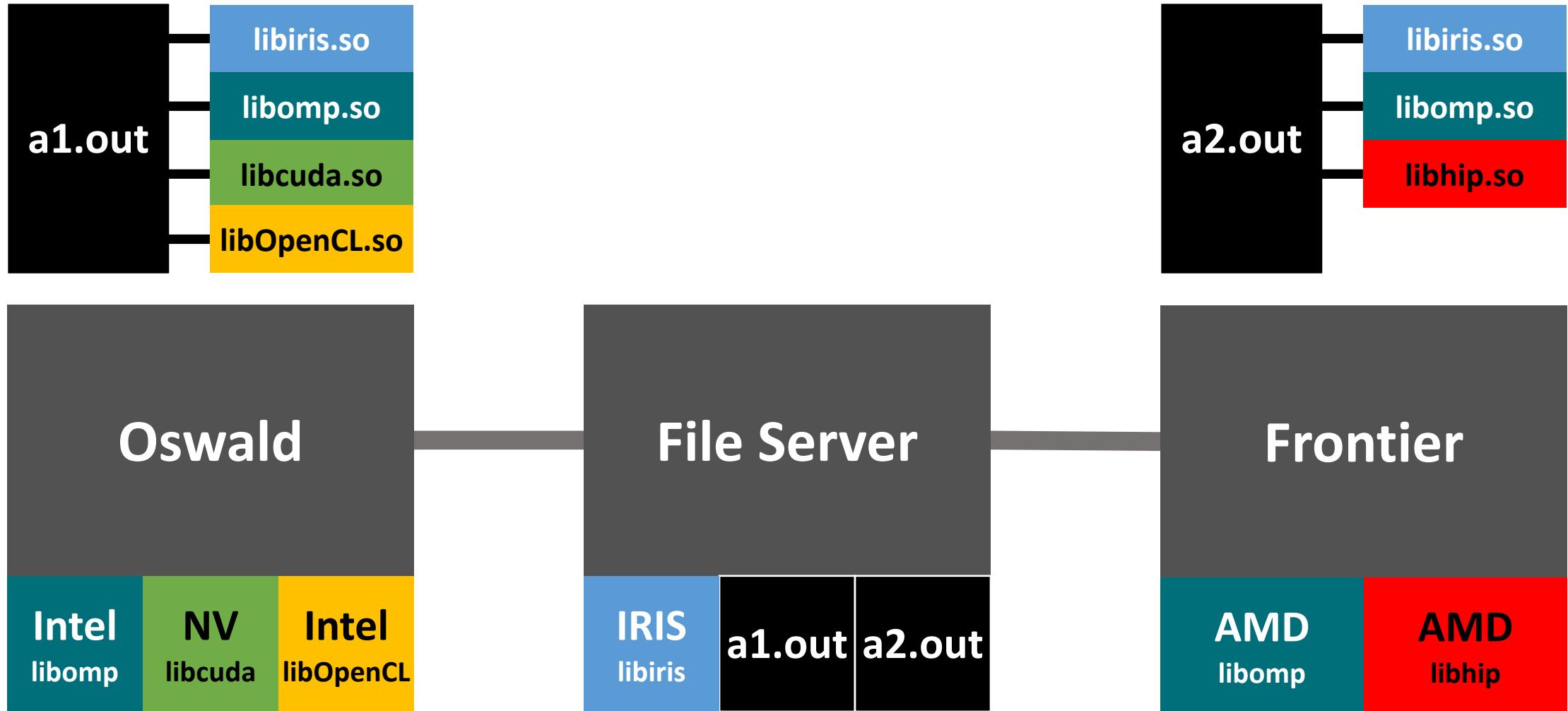
## Device Subclasses

DeviceCUDA  
DeviceHexagon  
DeviceHIP  
DeviceLevelZero  
DeviceOpenCL  
DeviceOpenMP

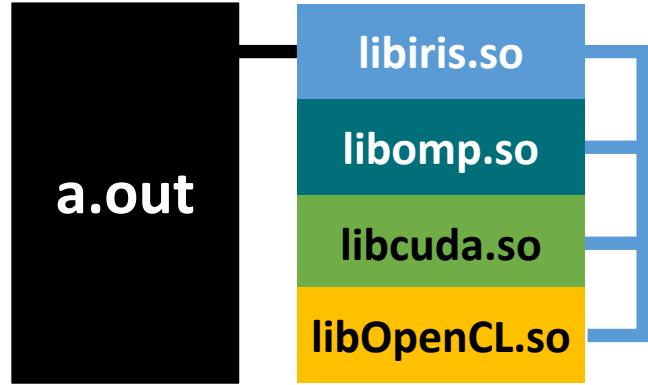


```
Platform.cpp buffers
316 int Platform::InitHIP() {
317     if (arch_available_ & brisbane_amd) {
318         _trace("%s", "skipping HIP architecture");
319         return BRISBANE_ERR;
320     }
321     loaderHIP_ = new LoaderHIP();
322     if (loaderHIP_->Load() != BRISBANE_OK) {
323         _trace("%s", "skipping HIP architecture");
324         return BRISBANE_ERR;
325     }
326     hipError_t err = hipSuccess;
327     err = loaderHIP_->hipInit(0);
328     _hiperror(err);
329     int ndevs = 0;
330     err = loaderHIP_->hipGetDeviceCount(&ndevs);
331     _hiperror(err);
332     if (getenv("IRIS_SINGLE")) ndevs = 1;
333     _trace("HIP platform[%d] ndevs[%d]", nplatforms_, ndevs);
334     for (int i = 0; i < ndevs; i++) {
335         hipDevice_t dev;
336         err = loaderHIP_->hipDeviceGet(&dev, i);
337         _hiperror(err);
338         devs_[ndevs_] = new DeviceHIP(loaderHIP_, dev, i, ndevs_, nplatforms_);
339         arch_available_ |= devs_[ndevs_]->type();
340         ndevs_++;
341     }
342     if (ndevs) {
343         strcpy(platform_names_[nplatforms_], "HIP");
344         nplatforms_++;
345     }
346     return BRISBANE_OK;
347 }
348
349 int Platform::InitLevelZero() {
350 }
```

# Without DPL: Not Portable Executable

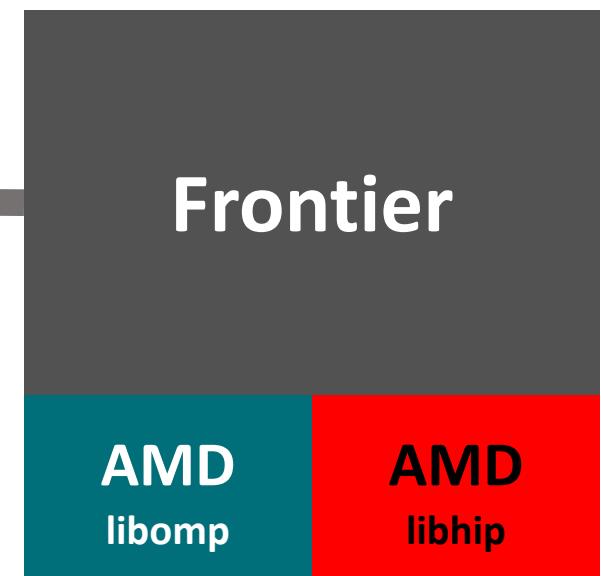
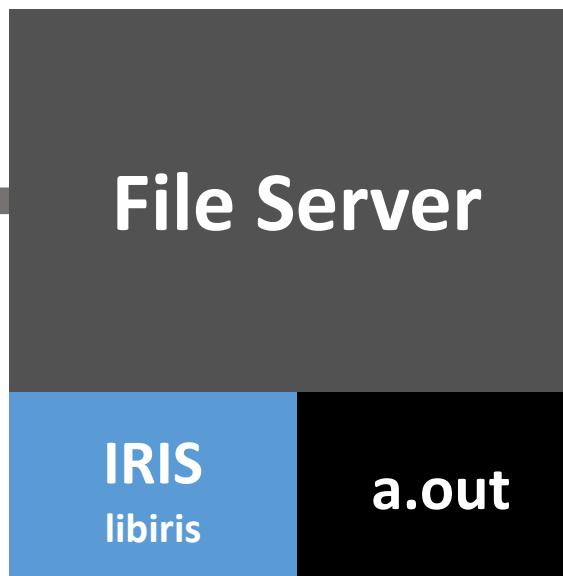
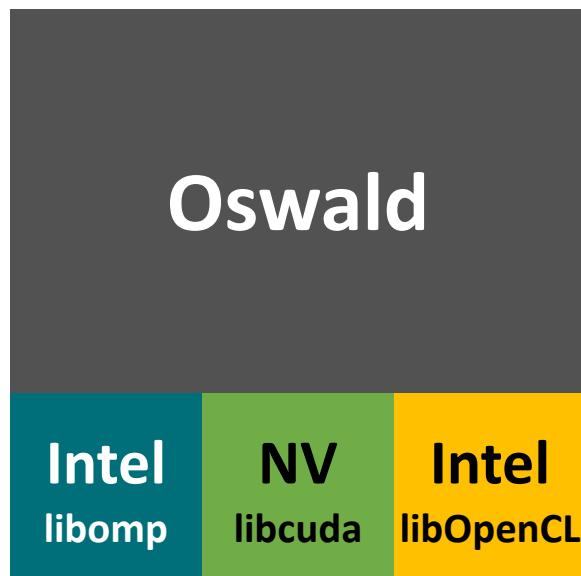
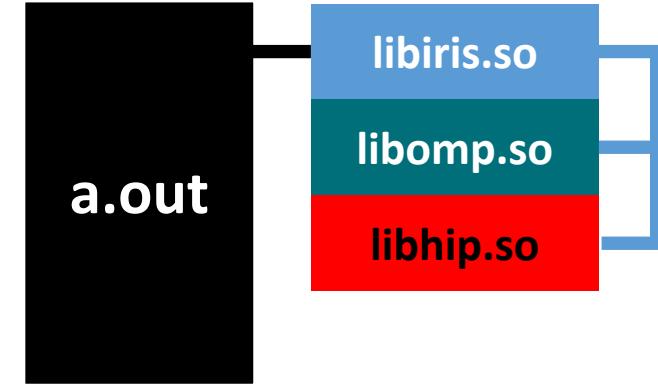


# Dynamic Platform Loader: Portable Executable



\$ CC app.c -o a.out -liris

- DPL automatically loads all available platforms on run time.
- **Private linkchain**



# src/runtime/QueueTask.cpp

```
Platform.cpp ssh buffers
167
168 brisbane_kernel null_brs_kernel;
169 KernelCreate("brisbane_null", &null_brs_kernel);
170 null_kernel_ = null_brs_kernel->class_obj;
171
172 if (enable_profiler_) {
173     profilers_[nprofilers_++] = new ProfilerDOT(thi
174     profilers_[nprofilers_++] = new ProfilerGoogle(
175 his);
176 }
177 present_table_ = new PresentTable();
178 queue_ = new QueueTask(this);
179 pool_ = new Pool(this);
180
181 InitScheduler();
182 InitWorkers();
183 InitDevices(sync);
184
185 _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d] scheduler[%d] hub[%d] polyhedral[%d] profile[%d]",
186     nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != N
187     ULL, scheduler_ ? scheduler_->hub_available() : 0,
188     polyhedral_available_, enable_profiler_);
189
190 timer_->Stop(BRISBANE_TIMER_PLATFORM);
191
192 init_ = true;
193
194 pthread_mutex_unlock(&mutex_);
195
196 return BRISBANE_OK;
197
N... Platform.cpp cpp 18% h:197 l:1
```

QueueTask:  
*Application Task*  
Queue

Out-or-order queue

```
QueueTask.h ssh buffers
16 QueueTask(Platform* platform);
17 ~QueueTask();
18
19 bool Enqueue(Task* task);
20 bool Dequeue(Task** task);
21 size_t Size();
22 bool Empty();
23
24 private:
25 Platform* platform_;
26 std::list<Task*> tasks_;
27 pthread_mutex_t mutex_;
28 Task* last_sync_task_;
29 bool enable_profiler_;
30 };
31
32 } /* namespace rt */
N... QueueTask.h cpp 54% h:19 l:16
7
8 QueueTask::QueueTask(Platform* platform) {
9     platform_ = platform;
10    enable_profiler_ = platform->enable_profiler();
11    last_sync_task_ = NULL;
12    pthread_mutex_init(&mutex_, NULL);
13 }
14
15 QueueTask::~QueueTask() {
16     pthread_mutex_destroy(&mutex_);
17 }
18
19 bool QueueTask::Dequeue(Task** task) {
20     pthread_mutex_lock(&mutex_);
21     if (tasks_.empty()) {
22         pthread_mutex_unlock(&mutex_);
23         return false;
24     }
25     Task* task_ = tasks_.front();
26     tasks_.pop_front();
27     *task = task_;
28     pthread_mutex_unlock(&mutex_);
29     return true;
30 }
31
32 QueueTask.cpp cpp 18% h:13 l:1
```

# src/runtime/Scheduler.cpp

```
Platform.cpp ssh buffers
167
168 brisbane_kernel null_brs_kernel;
169 KernelCreate("brisbane_null", &null_brs_kernel);
170 null_kernel_ = null_brs_kernel->class_obj;
171
172 if (enable_profiler_) {
173     profilers_[nprofilers_++] = new ProfilerDOT(thi
174     profilers_[nprofilers_++] = new ProfilerGoogle
175 his);
176 }
177 present_table_ = new PresentTable();
178 queue_ = new QueueTask(this);
179 pool_ = new Pool(this);
180
181 InitScheduler();
182 InitWorkers();
183 InitDevices(sync);
184
185 _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d] schedu
186 ler[%d] hub[%d] polyhedral[%d] profile[%d]",
187 nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != N
188 ULL, scheduler_ ? scheduler_->hub_available() : 0,
189 polyhedral_available_, enable_profiler_);
190
191 timer_->Stop(BRISBANE_TIMER_PLATFORM);
192
193 init_ = true;
194
195 pthread_mutex_unlock(&mutex_);
196 }
197
N... Platform.cpp cpp 18% h:197 l:1
```

*Scheduler*  
a pthread instance  
scheduling the  
Application Task  
Queue

```
Platform.cpp Scheduler.cpp buffers
988
989 int Platform::InitScheduler() {
990     if (ndevs_ == 1) {
991         _info("No scheduler ndevs[%d]", ndevs_);
992         return BRISBANE_OK;
993     }
994     _info("Scheduler ndevs[%d] ndevs_enabled[%d]", ndevs_, ndevs_
995     enabled_);
996     scheduler_ = new Scheduler(this);
997     scheduler_->Start();
998     return BRISBANE_OK;
999 }
1000 int Platform::InitWorkers() {
1001     if (ndevs_ == 1) {
1002         workers_[0] = new Worker(devs_[0], this, true);
1003         workers_[0]->Start();
1004     }
N... Platform.cpp cpp 93% h:988 l:1
80
81 void Scheduler::Run() {
82     while (true) {
83         Sleep();
84         if (!running_) break;
85         Task* task = NULL;
86         while (queue_->Dequeue(&task)) Submit(task);
87     }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
91     dev->worker()->Enqueue(task);
92     if (hub_available_) hub_client_->TaskInc(dev->devno(), 1);
93 }
94
95 void Scheduler::Submit(Task* task) {
96     if (!ndevs_) {
Scheduler.cpp cpp 64% h:93 l:1
```

# src/runtime/Worker.cpp

```
Platform.cpp ssh buffers
167
168 brisbane_kernel null_brs_kernel;
169 KernelCreate("brisbane_null", &null_brs_kernel);
170 null_kernel_ = null_brs_kernel->class_obj;
171
172 if (enable_profiler_) {
173     profilers_[nprofilers_++] = new ProfilerDOT(this);
174     profilers_[nprofilers_++] = new ProfilerGoogleCloud(this);
175 }
176
177 present_table_ = new PresentTable();
178 queue_ = new QueueTask(this);
179 pool_ = new Pool(this);
180
181 InitScheduler();
182 InitWorkers();
183 InitDevices(sync);
184
185 _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d]
186     scheduler[%d] hub[%d] polyhedral[%d] profile[%d]",
187     nplatforms_, ndevs_, ndevs_enabled_, scheduler_
188     ? scheduler_->hub_available() : 0,
189     polyhedral_available_, enable_profiler_);
190
191 timer_->Stop(BRISBANE_TIMER_PLATFORM);
192
193 init_ = true;
194
195 pthread_mutex_unlock(&mutex_);
196
197 return BRISBANE_OK;
198 }
```

## Worker

a pthread instance

N workers for  
N devices

scheduling a Ready  
Queue (Multiple  
Producers Single  
Consumer Lock-free  
In-order Queue)

```
Platform.cpp ssh buffers
1000 int Platform::InitWorkers() {
1001     if (ndevs_ == 1) {
1002         workers_[0] = new Worker(devs_[0], this, true);
1003         workers_[0]->Start();
1004         return BRISBANE_OK;
1005     }
1006     for (int i = 0; i < ndevs_; i++) {
1007         workers_[i] = new Worker(devs_[i], this);
1008         workers_[i]->Start();
1009     }
1010     return BRISBANE_OK;
1011 }
```

Platform.cpp cpp 94% h:1000 l:1

```
19     if (scheduler_) consistency_ = scheduler_->consistency();
20     else consistency_ = NULL;
21     single_ = single;
22     if (single) queue_ = platform->queue();
23     else queue_ = new QueueReady(128);
24     busy_ = false;
25 }
```

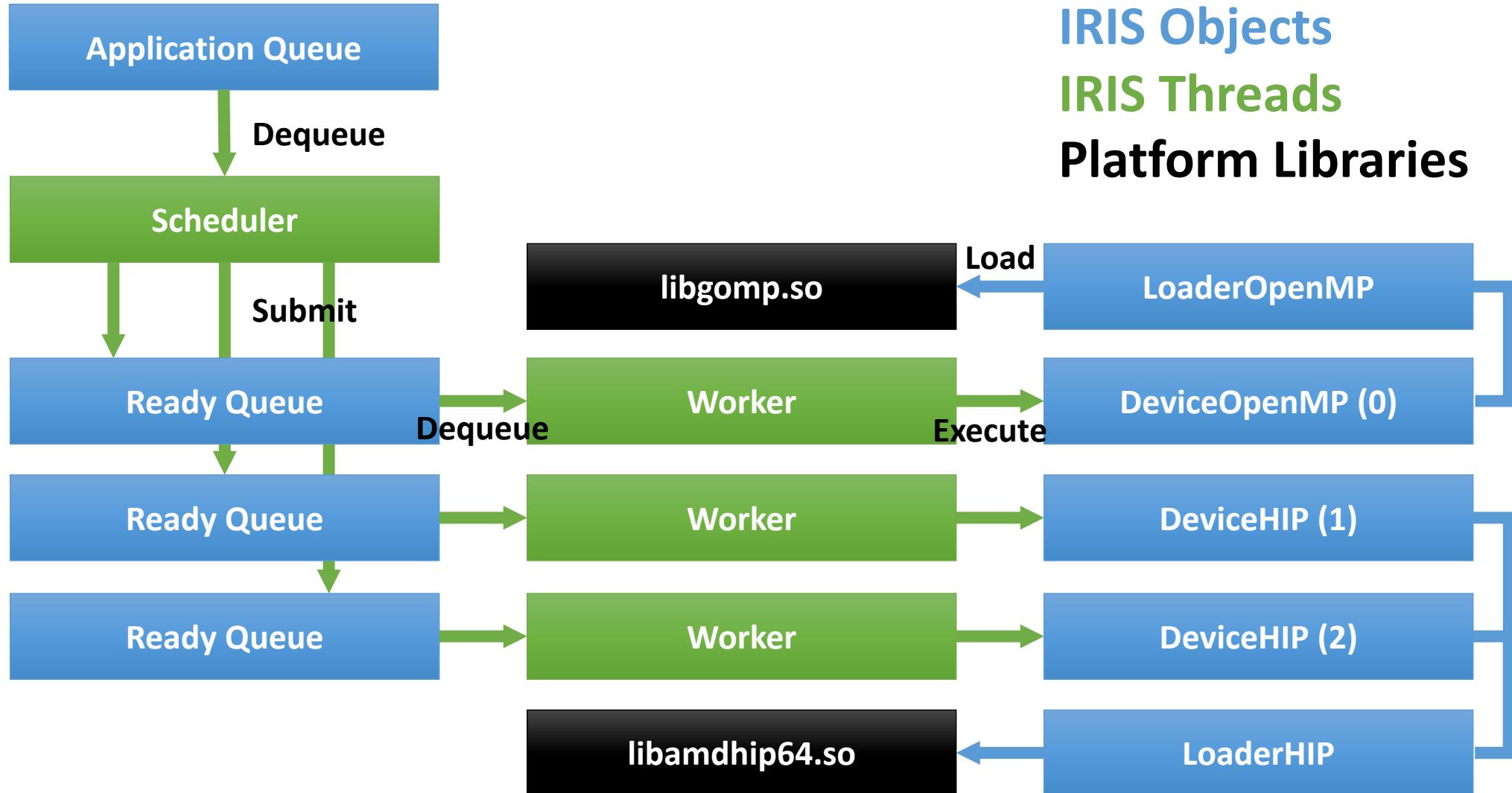
Worker.cpp cpp 35% h:26 l:1

```
27 Worker::~Worker() {
28     if (single_) delete queue_;
29 }
```

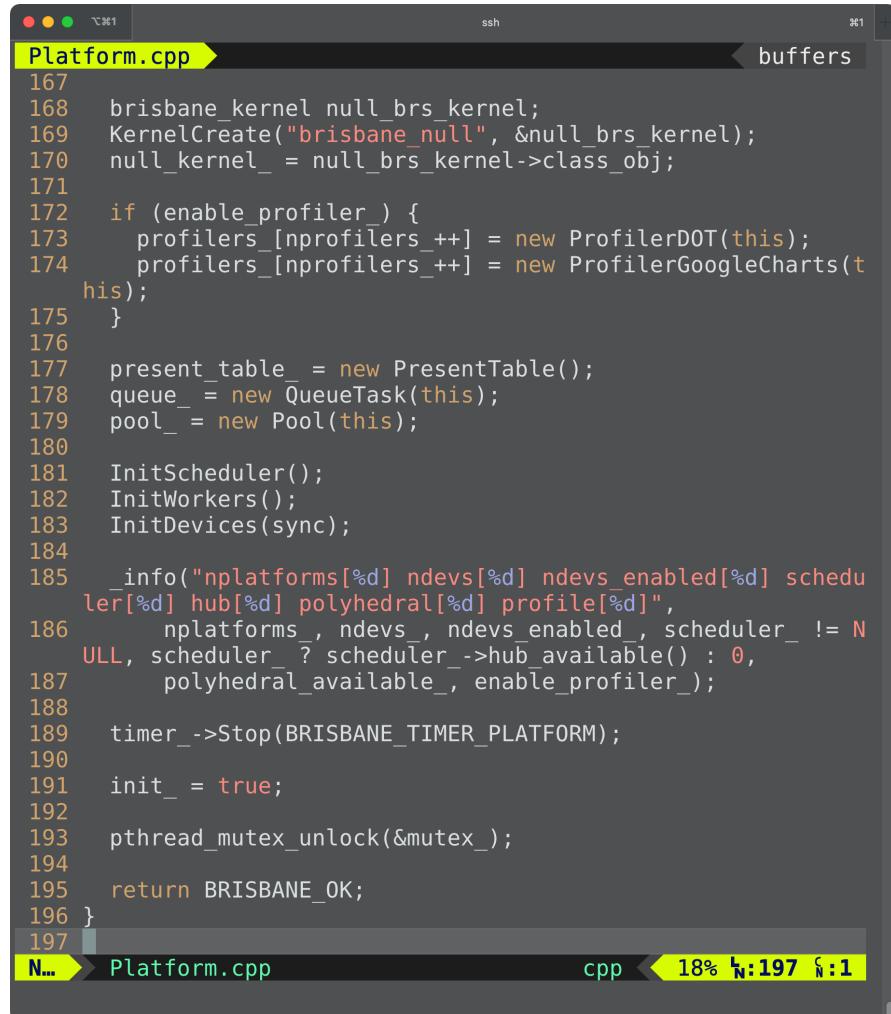
Worker.cpp cpp 93% h:69 l:14

```
59
60 void Worker::Run() {
61     while (true) {
62         Sleep();
63         if (!running_) break;
64         Task* task = NULL;
65         while (queue_->Dequeue(&task)) Execute(task);
66     }
67 }
68
69 unsigned long Worker::ntasks()
```

# The IRIS Platform So Far



# src/runtime/Platform.cpp



```
Platform.cpp buffers
167
168     brisbane_kernel null_brs_kernel;
169     KernelCreate("brisbane_null", &null_brs_kernel);
170     null_kernel_ = null_brs_kernel->class_obj;
171
172     if (enable_profiler_) {
173         profilers_[nprofilers_++] = new ProfilerDOT(this);
174         profilers_[nprofilers_++] = new ProfilerGoogleCharts(t
his);
175     }
176
177     present_table_ = new PresentTable();
178     queue_ = new QueueTask(this);
179     pool_ = new Pool(this);
180
181     InitScheduler();
182     InitWorkers();
183     InitDevices(sync);
184
185     _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d] schedu
ler[%d] hub[%d] polyhedral[%d] profile[%d]",
186           nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != N
ULL, scheduler_ ? scheduler_->hub_available() : 0,
187           polyhedral_available_, enable_profiler_);
188
189     timer_->Stop(BRISBANE_TIMER_PLATFORM);
190
191     init_ = true;
192
193     pthread_mutex_unlock(&mutex_);
194
195     return BRISBANE_OK;
196 }
197
```



```
Platform.cpp buffers
502
503     int Platform::InitDevices(bool sync) {
504         if (!ndebs_) {
505             dev_default_ = -1;
506             __error("%s", "NO AVAILABLE DEVICES!");
507             return BRISBANE_ERR;
508         }
509         char* c = getenv("IRIS_DEVICE_DEFAULT");
510         if (c) dev_default_ = atoi(c);
511
512         Task** tasks = new Task*[ndebs_];
513         for (int i = 0; i < ndebs_; i++) {
514             tasks[i] = new Task(this);
515             tasks[i]->set_system();
516             Command* cmd = Command::CreateInit(tasks[i]);
517             tasks[i]->AddCommand(cmd);
518             workers_[i]->Enqueue(tasks[i]);
519         }
520         if (sync) for (int i = 0; i < ndebs_; i++) tasks[i]->Wait();
521         delete[] tasks;
522         return BRISBANE_OK;
523     }
524
525     int Platform::PlatformCount(int* nplatforms) {
526         if (nplatforms) *nplatforms = nplatforms_;
527         return BRISBANE_OK;
528     }
529
530     int Platform::PlatformInfo(int platform, int param, void* value
, size_t* size) {
531         if (platform >= nplatforms_) return BRISBANE_ERR;
532         switch (param) {
533             case brisbane_name:
534                 if (*size) *size = strlen(platform_names_[platform]);
535                 strcpy((char*) value, platform_names_[platform]);
536         }
537     }
```

# src/runtime/Task.h

A screenshot of a terminal window with a dark background. It shows two tabs: "Task.cpp" and "Task.h". The "Task.h" tab is active, displaying the following C++ code:

```
92 private:
93     char name_[64];
94     bool given_name_;
95     Task* parent_;
96     int ncmds_;
97     Command* cmd_[64];
98     Command* cmd_kernel_;
99     Command* cmd_last_;
100    Device* dev_;
101    int devno_;
102    Platform* platform_;
103    Scheduler* scheduler_;
104    std::vector<Task*> subtasks_;
105    size_t subtasks_complete_;
106    void* arch_;
107
108    Task** depends_;
109    int depends_max_;
110    int ndepends_;
111
112    int brs_policy_;
113    int brs_policy_perm_;
114    char opt_[64];
115    bool sync_;
116
117    int type_;
118    int status_;
119    bool perm_;
120    bool user_;
121    bool system_;
122
123    double time_;
124    double time_start_;
125    double time_end_;
```

The status bar at the bottom shows "Task.h" is the current file, "cpp" is the language, "83%" is the completion percentage, "w:116" is the width, and "l:1" is the line number.

A screenshot of a terminal window with a dark background. It shows two tabs: "Platform.cpp" and "Task.h". The "Platform.cpp" tab is active, displaying the following C++ code:

```
502
503 int Platform::InitDevices(bool sync) {
504     if (!ndeps_) {
505         dev_default_ = -1;
506         __error("No available devices!");
507         return BRISBANE_ERR;
508     }
509     char* c = getenv("IRIS_DEVICE_DEFAULT");
510     if (c) dev_default_ = atoi(c);
511
512     Task** tasks = new Task*[ndeps_];
513     for (int i = 0; i < ndeps_; i++) {
514         tasks[i] = new Task(this);
515         tasks[i]->set_system();
516         Command* cmd = Command::CreateInit(tasks[i]);
517         tasks[i]->AddCommand(cmd);
518         workers_[i]->Enqueue(tasks[i]);
519     }
520     if (sync) for (int i = 0; i < ndeps_; i++) tasks[i]->Wait();
521     delete[] tasks;
522     return BRISBANE_OK;
523 }
524
525 int Platform::PlatformCount(int* nplatforms) {
526     if (nplatforms) *nplatforms = nplatforms_;
527     return BRISBANE_OK;
528 }
529
530 int Platform::PlatformInfo(int platform, int param, void* value
, size_t* size) {
531     if (platform >= nplatforms_) return BRISBANE_ERR;
532     switch (param) {
533         case brisbane_name:
534             if (*size) *size = strlen(platform_names_[platform]);
535             strcpy((char*) value, platform_names_[platform]);
536     }
537 }
```

The status bar at the bottom shows "Platform.cpp" is the current file, "cpp" is the language, "50%" is the completion percentage, "w:535" is the width, and "l:15" is the line number.

# src/runtime/Command.cpp

```
Task.cpp > Task.h > Command.cpp > Pool.cpp > buffers
30     return new Task(platform_, BRISBANE_TASK, NULL);
31 #endif
32 }
33
34 Command* Pool::GetCommand(Task* task, int type) {
35 #if BRISBANE_POOL_ENABLED
36     Command* cmd = cmds_[cid_++];
37     cmd->Set(task, type);
38     return cmd;
39 #else
40     return new Command(task, type);
41 #endif
42 }
43
44 } /* namespace rt */
45 } /* namespace brisbane */
46
Pool.cpp
Command.cpp
      cpp    100%  l:46 n:1
55     return time_;
56 }
57
58 Command* Command::Create(Task* task, int type) {
59     return task->platform()->pool()->GetCommand(task, type);
60 }
61
62 Command* Command::CreateInit(Task* task) {
63     return Create(task, BRISBANE_CMD_INIT);
64 }
65
66 Command* Command::CreateKernel(Task* task, Kernel* kernel,
67     int dim, size_t* off, size_t* gws, size_t* lws) {
68     Command* cmd = Create(task, BRISBANE_CMD_KERNEL);
69     cmd->kernel_ = kernel;
70     //cmd->kernel_args_ = kernel->ExportArgs();
71
N... Command.cpp
      cpp    24%  l:65 n:1
```

```
Platform.cpp
502
503 int Platform::InitDevices(bool sync) {
504     if (!ndeps_) {
505         dev_default_ = -1;
506         __error("No available devices!");
507         return BRISBANE_ERR;
508     }
509     char* c = getenv("IRIS_DEVICE_DEFAULT");
510     if (c) dev_default_ = atoi(c);
511
512     Task** tasks = new Task*[ndeps_];
513     for (int i = 0; i < ndeps_; i++) {
514         tasks[i] = new Task(this);
515         tasks[i]->set_system();
516         Command* cmd = Command::CreateInit(tasks[i]);
517         tasks[i]->AddCommand(cmd);
518         workers_[i]->Enqueue(tasks[i]);
519     }
520     if (sync) for (int i = 0; i < ndeps_; i++) tasks[i]->Wait();
521     delete[] tasks;
522     return BRISBANE_OK;
523 }
524
525 int Platform::PlatformCount(int* nplatforms) {
526     if (nplatforms) *nplatforms = nplatforms_;
527     return BRISBANE_OK;
528 }
529
530 int Platform::PlatformInfo(int platform, int param, void* value
, size_t* size) {
531     if (platform >= nplatforms_) return BRISBANE_ERR;
532     switch (param) {
533         case brisbane_name:
534             if (*size) *size = strlen(platform_names_[platform]);
535             strcpy((char*) value, platform_names_[platform]);
536     }
N... Platform.cpp
      cpp    50%  l:535 n:15
```

# src/runtime/Task.cpp

```
Task.cpp buffers
138 void Task::CompleteSub() {
139     pthread_mutex_lock(&mutex_subtasks_);
140     if (++subtasks_complete_ == subtasks_.size()) Complete();
141     pthread_mutex_unlock(&mutex_subtasks_);
142 }
143 }
144
145 void Task::Wait() {
146     pthread_mutex_lock(&mutex_complete_);
147     if (status_ != BRISBANE_COMPLETE)
148         pthread_cond_wait(&complete_cond_, &mutex_complete_);
149     pthread_mutex_unlock(&mutex_complete_);
150 }
151
152 void Task::AddSubtask(Task* subtask) {
153     subtask->set_parent(this);
154     subtask->set_brs_policy(brs_policy_);
155     subtasks_.push_back(subtask);
156 }
157
158 bool Task::HasSubtasks() {
159     return !subtasks_.empty();
160 }
161
162 void Task::AddDepend(Task* task) {
163     if (depends_ == NULL) depends_ = new Task*[depends_max_];
164     for (int i = 0; i < ndepends_; i++) if (task == depends_[i]) return;
165     if (ndepends_ == depends_max_-1) {
166         Task** old = depends_;
167         depends_max_*= 2;
168         depends_ = new Task*[depends_max_];
169         memcpy(depends_, old, ndepends_* sizeof(Task*));
170     }
N... Task.cpp      cpp 77% h:151 l:1
"Task.cpp" 196L, 5102C written
```

```
Platform.cpp buffers
502
503 int Platform::InitDevices(bool sync) {
504     if (!ndebs_) {
505         dev_default_ = -1;
506         __error("No available devices!");
507         return BRISBANE_ERR;
508     }
509     char* c = getenv("IRIS_DEVICE_DEFAULT");
510     if (c) dev_default_ = atoi(c);
511
512     Task** tasks = new Task*[ndebs_];
513     for (int i = 0; i < ndebs_; i++) {
514         tasks[i] = new Task(this);
515         tasks[i]->set_system();
516         Command* cmd = Command::CreateInit(tasks[i]);
517         tasks[i]->AddCommand(cmd);
518         workers_[i]->Enqueue(tasks[i]);
519     }
520     if (sync) for (int i = 0; i < ndebs_; i++) tasks[i]->Wait();
521     delete[] tasks;
522     return BRISBANE_OK;
523 }
524
525 int Platform::PlatformCount(int* nplatforms) {
526     if (nplatforms) *nplatforms = nplatforms_;
527     return BRISBANE_OK;
528 }
529
530 int Platform::PlatformInfo(int platform, int param, void* value
, size_t* size) {
531     if (platform >= nplatforms_) return BRISBANE_ERR;
532     switch (param) {
533         case brisbane_name:
534             if (*size) *size = strlen(platform_names_[platform]);
535             strcpy((char*) value, platform_names_[platform]);
N... Platform.cpp      cpp 50% h:535 l:1
Platform.cpp 196L, 5102C written
```

# src/runtime/Worker.cpp

A screenshot of a terminal window with a dark background. The title bar says "Platform.cpp > Worker.cpp". The buffer area contains the code for the Worker class. The code handles task enqueueing, execution, and management. It includes methods like Enqueue, Execute, Run, and various consistency and scheduler interactions. Line numbers are visible on the left.

```
34 }
35
36 void Worker::Enqueue(Task* task) {
37     while (!queue_->Enqueue(task)) { }
38     Invoke();
39 }
40
41 void Worker::Execute(Task* task) {
42     if (!task->Executable()) return;
43     task->set_dev(dev_);
44     if (task->marker()) {
45         dev_->Synchronize();
46         task->Complete();
47         return;
48     }
49     busy_ = true;
50     if (scheduler_) scheduler_->StartTask(task, this);
51     if (consistency_) consistency_->Resolve(task);
52     dev_->Execute(task);
53     if (!task->cmd_last()) {
54         if (scheduler_) scheduler_->CompleteTask(task, this);
55         //task->Complete();
56     }
57     busy_ = false;
58 }
59
60 void Worker::Run() {
61     while (true) {
62         Sleep();
63         if (!running_) break;
64         Task* task = NULL;
65         while (queue_->Dequeue(&task)) Execute(task);
66     }
67 }
68 }
```

Worker.cpp    cpp    54% ⏵:40 ⏵:1

A screenshot of a terminal window with a dark background. The title bar says "Platform.cpp". The buffer area contains the code for the Platform class. It includes methods for initializing devices, creating tasks, and managing platform counts. The code uses C-style memory management (new, delete) and interacts with the Task and Command classes. Line numbers are visible on the left.

```
502
503 int Platform::InitDevices(bool sync) {
504     if (!ndevs_) {
505         dev_default_ = -1;
506         __error("No available devices!");
507         return BRISBANE_ERR;
508     }
509     char* c = getenv("IRIS_DEVICE_DEFAULT");
510     if (c) dev_default_ = atoi(c);
511
512     Task** tasks = new Task*[ndevs_];
513     for (int i = 0; i < ndevs_; i++) {
514         tasks[i] = new Task(this);
515         tasks[i]->set_system();
516         Command* cmd = Command::CreateInit(tasks[i]);
517         tasks[i]->AddCommand(cmd);
518         workers_[i]->Enqueue(tasks[i]);
519     }
520     if (sync) for (int i = 0; i < ndevs_; i++) tasks[i]->Wait();
521     delete[] tasks;
522     return BRISBANE_OK;
523 }
524
525 int Platform::PlatformCount(int* nplatforms) {
526     if (nplatforms) *nplatforms = nplatforms_;
527     return BRISBANE_OK;
528 }
529
530 int Platform::PlatformInfo(int platform, int param, void* value
, size_t* size) {
531     if (platform >= nplatforms_) return BRISBANE_ERR;
532     switch (param) {
533         case brisbane_name:
534             if (*size) *size = strlen(platform_names_[platform]);
535             strcpy((char*) value, platform_names_[platform]);
536     }
537 }
```

Platform.cpp    cpp    50% ⏵:535 ⏵:15

# src/runtime/Thread.cpp



```
Platform.cpp > Worker.cpp buffers
34 }
35
36 void Worker::Enqueue(Task* task) {
37     while (!queue_->Enqueue(task)) { }
38     Invoke();
39 }
40
41 void Worker::Execute(Task* task) {
42     if (!task->Executable()) return;
43     task->set_dev(dev_);
44     if (task->marker()) {
45         dev_->Synchronize();
46         task->Complete();
47         return;
48     }
49     busy_ = true;
50     if (scheduler_) scheduler_->StartTask(task, this);
51     if (consistency_) consistency_->Resolve(task);
52     dev_->Execute(task);
53     if (!task->cmd_last()) {
54         if (scheduler_) scheduler_->CompleteTask(task, this);
55         //task->Complete();
56     }
57     busy_ = false;
58 }
59
60 void Worker::Run() {
61     while (true) {
62         Sleep();
63         if (!running_) break;
64         Task* task = NULL;
65         while (queue_->Dequeue(&task)) Execute(task);
66     }
67 }
68
```

Worker.cpp    cpp 54% N:40 F:1



```
Thread.cpp buffers
15     sem_destroy(&sem_);
16 }
17
18 void Thread::Start() {
19     if (thread_) return;
20     running_ = true;
21     pthread_create(&thread_, NULL, &Thread::ThreadFunc, this)
22 ;
23 }
24 void Thread::Stop() {
25     if (!thread_) return;
26     running_ = false;
27     Invoke();
28     pthread_join(thread_, NULL);
29     thread_ = (pthread_t) NULL;
30 }
31
32 void Thread::Sleep() {
33     sem_wait(&sem_);
34 }
35
36 void Thread::Invoke() {
37     sem_post(&sem_);
38 }
39
40 void* Thread::ThreadFunc(void* argp) {
41     ((Thread*) argp)->Run();
42     return NULL;
43 }
44
45 } /* namespace rt */
46 } /* namespace brisbane */
47
```

N... > Thread.cpp    cpp 100% N:47 F:1

E486: Pattern not found: asdf

# src/runtime/Device.cpp

```
Platform.cpp > Worker.cpp buffers
34 }
35
36 void Worker::Enqueue(Task* task) {
37     while (!queue_->Enqueue(task)) { }
38     Invoke();
39 }
40
41 void Worker::Execute(Task* task) {
42     if (!task->Executable()) return;
43     task->set_dev(dev_);
44     if (task->marker()) {
45         dev_->Synchronize();
46         task->Complete();
47         return;
48     }
49     busy_ = true;
50     if (scheduler_) scheduler_->StartTask(task, this);
51     if (consistency_) consistency_->Resolve(task);
52     dev_->Execute(task);
53     if (!task->cmd_last()) {
54         if (scheduler_) scheduler_->CompleteTask(task, this);
55         //task->Complete();
56     }
57     busy_ = false;
58 }
59
60 void Worker::Run() {
61     while (true) {
62         Sleep();
63         if (!running_) break;
64         Task* task = NULL;
65         while (queue_->Dequeue(&task)) Execute(task);
66     }
67 }
68
```

Worker.cpp    cpp    54%  h:40  l:1

```
Device.cpp buffers
35     delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%lx]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61 #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //     if (++q_ >= nqueues_) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", task->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
75 }
```

Device.cpp    cpp    27%  h:73  l:14    [225]tr...

# src/runtime/Device.cpp

```
Device.cpp
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
75     if (SupportJIT()) {
76         char* tmpdir = NULL;
77         char* src = NULL;
78         char* bin = NULL;
79         Platform::GetPlatform()->EnvironmentGet("TMPDIR", &tmpdir, NULL);
80         Platform::GetPlatform()->EnvironmentGet(kernel_src(), &src, NULL);
81         Platform::GetPlatform()->EnvironmentGet(kernel_bin(), &bin, NULL);
82         bool stat_src = Utils::Exist(src);
83         bool stat_bin = Utils::Exist(bin);
84         if (!stat_src && !stat_bin) {
85             _error("NO KERNEL SRC[%s] NO KERNEL BIN[%s]", src, bin);
86         } else if (!stat_src && stat_bin) {
87             strncpy(kernel_path_, bin, strlen(bin));
88         } else if (stat_src && !stat_bin) {
89             sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
90             errid_ = Compile(src);
91         } else {
92             long mtime_src = Utils::Mtime(src);
93             long mtime_bin = Utils::Mtime(bin);
94             if (mtime_src > mtime_bin) {
95                 sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
96                 errid_ = Compile(src);
97             } else
98                 strncpy(kernel_path_, bin, strlen(bin));
99         }
100        if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
101    errid_ = Init();
102    if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
103    double time = timer_->Stop(BRISBANE_TIMER_INIT);
104    cmd->SetTime(time);
105    enable_ = true;
106 }
107
108 void Device::ExecuteKernel(Command* cmd) {
109     timer_->Start(BRISBANE_TIMER_KERNEL);
NORMAL Device.cpp      cpp 39% 1:07 1:1 [225]tr...
```

```
Device.cpp
35     delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%02x]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61    #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //     if (++q_ >= nqueues_) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", task->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
NORMAL Device.cpp      cpp 27% 1:73 1:14 [225]tr...
```

# src/runtime/DeviceHIP.cpp

```
Device.cpp
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
75     if (SupportJIT()) {
76         char* tmpdir = NULL;
77         char* src = NULL;
78         char* bin = NULL;
79         Platform::GetPlatform()->EnvironmentGet("TMPDIR", &tmpdir, NULL);
80         Platform::GetPlatform()->EnvironmentGet(kernel_src(), &src, NULL);
81         Platform::GetPlatform()->EnvironmentGet(kernel_bin(), &bin, NULL);
82         bool stat_src = Utils::Exist(src);
83         bool stat_bin = Utils::Exist(bin);
84         if (!stat_src && !stat_bin) {
85             _error("NO KERNEL SRC[%s] NO KERNEL BIN[%s]", src, bin);
86         } else if (!stat_src && stat_bin) {
87             strncpy(kernel_path_, bin, strlen(bin));
88         } else if (stat_src && !stat_bin) {
89             sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
90             errid_ = Compile(src);
91         } else {
92             long mtime_src = Utils::Mtime(src);
93             long mtime_bin = Utils::Mtime(bin);
94             if (mtime_src > mtime_bin) {
95                 sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
96                 errid_ = Compile(src);
97             } else
98                 strncpy(kernel_path_, bin, strlen(bin));
99         }
100        if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
101    errid_ = Init();
102    if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
103    double time = timer_->Stop(BRISBANE_TIMER_INIT);
104    cmd->SetTime(time);
105    enable_ = true;
106 }
107
108 void Device::ExecuteKernel(Command* cmd) {
109     timer_->Start(BRISBANE_TIMER_KERNEL);
NORMAL Device.cpp      cpp 39% 4:107 1:1 [225]tr...
```

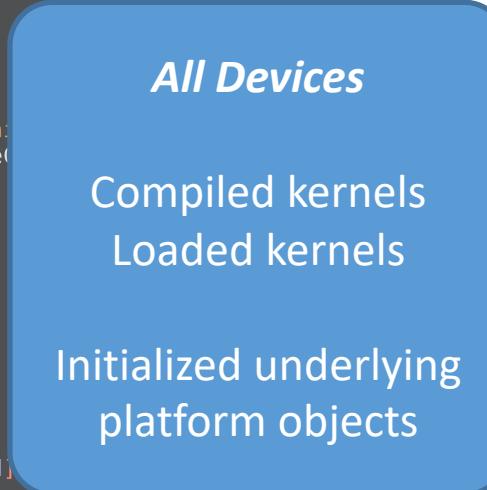
```
Device.cpp > DeviceHIP.cpp
24     type_ = brisbane_amd;
25     model_ = brisbane_hip;
26     err_ = ld_->hipDriverGetVersion(&driver_version_);
27     _hiperror(err_);
28     sprintf(version, "AMD HIP %d", driver_version_);
29     _info("device[%d] platform[%d] vendor[%s] device[%s] ordinal[%d] type[%d] version[%s]", devno_, platform_, vendor_, name_, ordinal_, type_, version_);
30 }
31
32 DeviceHIP::~DeviceHIP() {
33 }
34
35 int DeviceHIP::Compile(char* src) {
36     char cmd[256];
37     memset(cmd, 0, 256);
38     sprintf(cmd, "hipcc --genco %s -o %s", src, kernel_path_);
39     if (system(cmd) != EXIT_SUCCESS) {
40         _error("cmd[%s]", cmd);
41         return BRISBANE_ERR;
42     }
43     return BRISBANE_OK;
44 }
45
46 int DeviceHIP::Init() {
47     int tb, mc, bx, by, bz, dx, dy, dz, ck, ae;
48     err_ = ld_->hipSetDevice(ordinal_);
49     _hiperror(err_);
50     err_ = ld_->hipGetDevice(&devid_);
51     _hiperror(err_);
52     err_ = ld_->hipDeviceGetAttribute(&tb, hipDeviceAttributeMaxThreadsPerBlock, devid_);
53     err_ = ld_->hipDeviceGetAttribute(&mc, hipDeviceAttributeMultiprocessorCount, devid_);
54     err_ = ld_->hipDeviceGetAttribute(&bx, hipDeviceAttributeMaxBlockDimX, devid_);
55     err_ = ld_->hipDeviceGetAttribute(&by, hipDeviceAttributeMaxBlockDimY, devid_);
56     err_ = ld_->hipDeviceGetAttribute(&bz, hipDeviceAttributeMaxBlockDimZ, devid_);
57     err_ = ld_->hipDeviceGetAttribute(&dx, hipDeviceAttributeMaxGridDimX, devid_);
NORMAL DeviceHIP.cpp      cpp 21% 4:45 1:1 [225]tr...
```

# src/runtime/DeviceHIP.cpp

```
Device.cpp
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
75     if (SupportJIT()) {
76         char* tmpdir = NULL;
77         char* src = NULL;
78         char* bin = NULL;
79         Platform::GetPlatform()->EnvironmentGet("TMPDIR", &tmpdir, NULL);
80         Platform::GetPlatform()->EnvironmentGet(kernel_src(), &src, NULL);
81         Platform::GetPlatform()->EnvironmentGet(kernel_bin(), &bin, NULL);
82         bool stat_src = Utils::Exist(src);
83         bool stat_bin = Utils::Exist(bin);
84         if (!stat_src && !stat_bin) {
85             _error("NO KERNEL SRC[%s] NO KERNEL BIN[%s]", src, bin);
86         } else if (!stat_src && stat_bin) {
87             strncpy(kernel_path_, bin, strlen(bin));
88         } else if (stat_src && !stat_bin) {
89             sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
90             errid_ = Compile(src);
91         } else {
92             long mtime_src = Utils::Mtime(src);
93             long mtime_bin = Utils::Mtime(bin);
94             if (mtime_src > mtime_bin) {
95                 sprintf(kernel_path_, "%s/%s-%d", tmpdir, bin, devno_);
96                 errid_ = Compile(src);
97             } else
98                 strncpy(kernel_path_, bin, strlen(bin));
99         }
100        if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
101    errid_ = Init();
102    if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
103    double time = timer_->Stop(BRISBANE_TIMER_INIT);
104    cmd->SetTime(time);
105    enable_ = true;
106 }
107
108 void Device::ExecuteKernel(Command* cmd) {
109     timer_->Start(BRISBANE_TIMER_KERNEL);
NORMAL Device.cpp      cpp 39% 4:107 4:1 [225]tr...
```

```
DeviceHIP.cpp
44 }
45
46 int DeviceHIP::Init() {
47     int tb, mc, bx, by, bz, dx, dy, dz, ck, ae;
48     err = ld->hipSetDevice(ordinal_);
49     _hiperror(err);
50     err = ld->hipGetDevice(&devid_);
51     _hiperror(err);
52     err = ld->hipDeviceGetAttribute(&tb, hipDeviceAttributeMaxThreadsPerBlock, devid_);
53     err = ld->hipDeviceGetAttribute(&mc, hipDeviceAttributeMultiprocessorCount, devid_);
54     err = ld->hipDeviceGetAttribute(&bx, hipDeviceAttributeMaxBlockDimX, devid_);
55     err = ld->hipDeviceGetAttribute(&by, hipDeviceAttributeMaxBlockDimY, devid_);
56     err = ld->hipDeviceGetAttribute(&bz, hipDeviceAttributeMaxBlockDimZ, devid_);
57     err = ld->hipDeviceGetAttribute(&dx, hipDeviceAttributeMaxGridDimX, devid_);
58     err = ld->hipDeviceGetAttribute(&dy, hipDeviceAttributeMaxGridDimY, devid_);
59     err = ld->hipDeviceGetAttribute(&dz, hipDeviceAttributeMaxGridDimZ, devid_);
60     err = ld->hipDeviceGetAttribute(&ck, hipDeviceAttributeConcurrentKernels, devid_);
61     max_work_group_size_ = tb;
62     max_compute_units_ = mc;
63     max_block_dims_[0] = bx;
64     max_block_dims_[1] = by;
65     max_block_dims_[2] = bz;
66     max_work_item_sizes_[0] = (size_t) bx * (size_t) dx;
67     max_work_item_sizes_[1] = (size_t) by * (size_t) dy;
68     max_work_item_sizes_[2] = (size_t) bz * (size_t) dz;
69
70     _info("devid[%d] max compute units[%zu] max work group size[%zu] max work item sizes[%zu,%zu,%zu] max block dims[%d,%d,%d] concurrent kernels[%d]", devid, max_compute_units_, max_work_group_size_, max_work_item_sizes_[0], max_work_item_sizes_[1], max_work_item_sizes_[2], max_block_dims_[0], max_block_dims_[1], max_block_dims_[2], ck);
71
72     char* path = kernel_path_;
73     char* src = NULL;
74     size_t srclen = 0;
75     if (Utils::ReadFile(path, &src, &srclen) == BRISBANE_ERR) {
76         _error("dev[%d][%s] has no kernel file [%s]", devno_, name_, path);
77         return BRISBANE_OK;
78     }
79     _trace("dev[%d][%s] kernels[%s]", devno_, name_, path);
80     ld->Lock();
81     err = ld->hipModuleLoad(&module_, path);
82     ld->Unlock();
83     if (err_ != hipSuccess) {
84         _hiperror(err);
85         _error("srclen[%zu] src\n%s", srclen, src);
86         if (src) free(src);
87         return BRISBANE_ERR;
88     }
89     if (src) free(src);
90     return BRISBANE_OK;
91 }
92
93 int DeviceHIP::MemAlloc(void** mem, size_t size) {
94     void** hipmem = mem;
95     err = ld->hipMalloc(hipmem, size);
96     _hiperror(err);
NORMAL DeviceHIP.cpp      cpp utf-8[unix] 43% 4:93/213 4:16
```

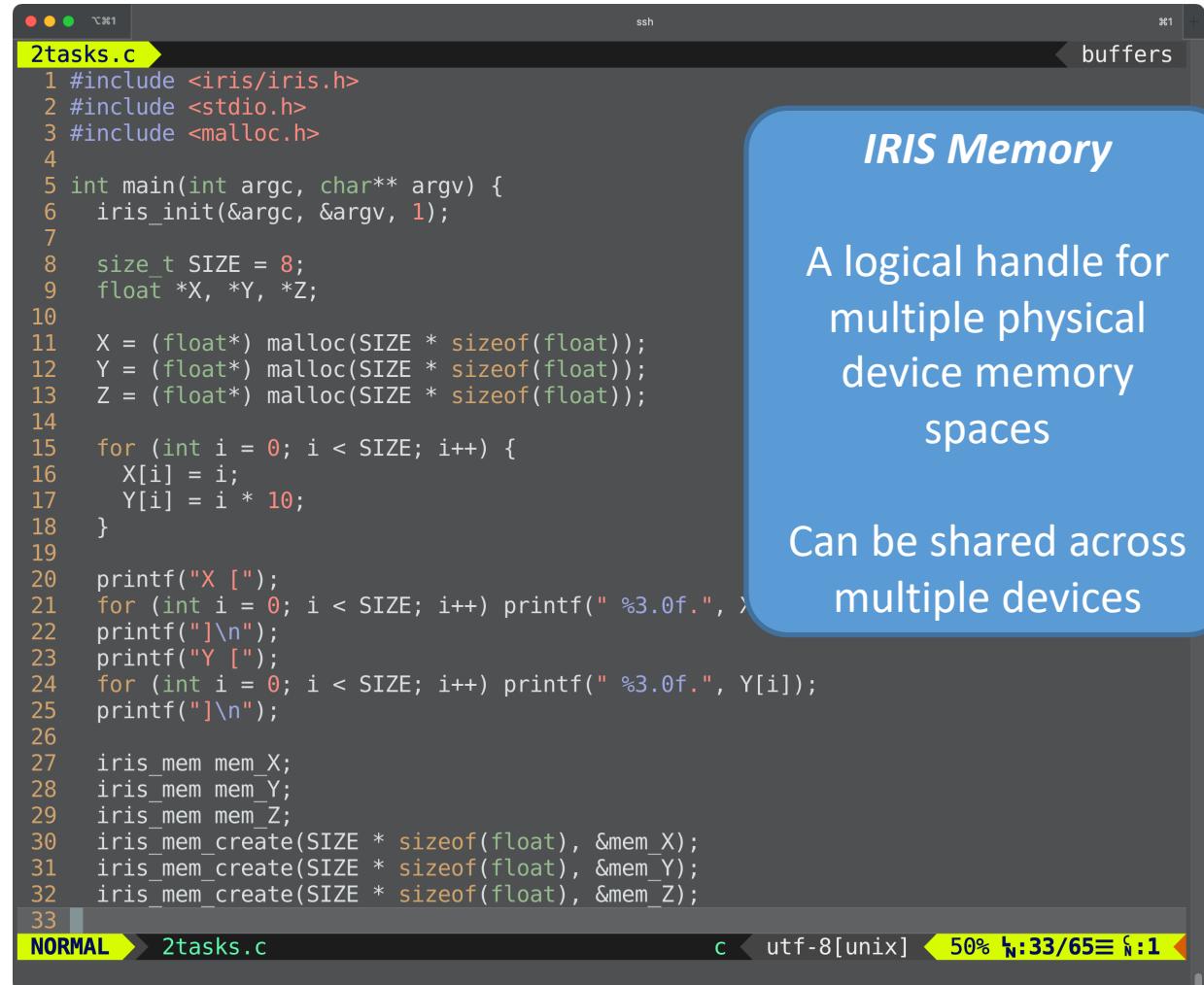
# src/runtime/Platform.cpp



```
167  
168     brisbane_kernel null_brs_kernel;  
169     KernelCreate("brisbane_null", &null_brs_kernel);  
170     null_kernel_ = null_brs_kernel->class_obj;  
171  
172     if (enable_profiler_) {  
173         profilers_[nprofilers_++] = new ProfilerDOT(th:  
174         profilers_[nprofilers_++] = new ProfilerGoogle  
his);  
175     }  
176  
177     present_table_ = new PresentTable();  
178     queue_ = new QueueTask(this);  
179     pool_ = new Pool(this);  
180  
181     InitScheduler();  
182     InitWorkers();  
183     InitDevices(sync);  
184  
185     _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d]  
ler[%d] hub[%d] polyhedral[%d] profile[%d]",  
186         nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != N  
ULL, scheduler_ ? scheduler_->hub_available() : 0,  
187         polyhedral_available_, enable_profiler_);  
188  
189     timer_->Stop(BRISBANE_TIMER_PLATFORM);  
190  
191     init_ = true;  
192  
193     pthread_mutex_unlock(&mutex_);  
194  
195     return BRISBANE_OK;  
196 }  
197  
N... Platform.cpp  cpp  18%  l:197  f:1
```

```
502  
503     int Platform::InitDevices(bool sync) {  
504         if (!ndebs_) {  
505             dev_default_ = -1;  
506             __error("%s", "NO AVAILABLE DEVICES!");  
507             return BRISBANE_ERR;  
508         }  
509         char* c = getenv("IRIS_DEVICE_DEFAULT");  
510         if (c) dev_default_ = atoi(c);  
511  
512         Task** tasks = new Task*[ndebs_];  
513         for (int i = 0; i < ndebs_; i++) {  
514             tasks[i] = new Task(this);  
515             tasks[i]->set_system();  
516             Command* cmd = Command::CreateInit(tasks[i]);  
517             tasks[i]->AddCommand(cmd);  
518             workers_[i]->Enqueue(tasks[i]);  
519         }  
520         if (sync) for (int i = 0; i < ndebs_; i++) tasks[i]->Wait();  
521         delete[] tasks;  
522         return BRISBANE_OK;  
523     }  
524  
525     int Platform::PlatformCount(int* nplatforms) {  
526         if (nplatforms) *nplatforms = nplatforms_;  
527         return BRISBANE_OK;  
528     }  
529  
530     int Platform::PlatformInfo(int platform, int param, void* value  
, size_t* size) {  
531         if (platform >= nplatforms_) return BRISBANE_ERR;  
532         switch (param) {  
533             case brisbane_name:  
534                 if (*size) *size = strlen(platform_names_[platform]);  
535                 strcpy((char*) value, platform_names_[platform]);  
N... Platform.cpp  cpp  50%  l:535  f:15
```

# iris\_mem\_create()



```
2tasks.c    ssh    buffers
1 #include <iris/iris.h>
2 #include <stdio.h>
3 #include <malloc.h>
4
5 int main(int argc, char** argv) {
6     iris_init(&argc, &argv, 1);
7
8     size_t SIZE = 8;
9     float *X, *Y, *Z;
10
11    X = (float*) malloc(SIZE * sizeof(float));
12    Y = (float*) malloc(SIZE * sizeof(float));
13    Z = (float*) malloc(SIZE * sizeof(float));
14
15    for (int i = 0; i < SIZE; i++) {
16        X[i] = i;
17        Y[i] = i * 10;
18    }
19
20    printf("X [");
21    for (int i = 0; i < SIZE; i++) printf(" %.3f.", X[i]);
22    printf("]\n");
23    printf("Y [");
24    for (int i = 0; i < SIZE; i++) printf(" %.3f.", Y[i]);
25    printf("]\n");
26
27    iris_mem mem_X;
28    iris_mem mem_Y;
29    iris_mem mem_Z;
30    iris_mem_create(SIZE * sizeof(float), &mem_X);
31    iris_mem_create(SIZE * sizeof(float), &mem_Y);
32    iris_mem_create(SIZE * sizeof(float), &mem_Z);
33
```

**IRIS Memory**

A logical handle for multiple physical device memory spaces

Can be shared across multiple devices

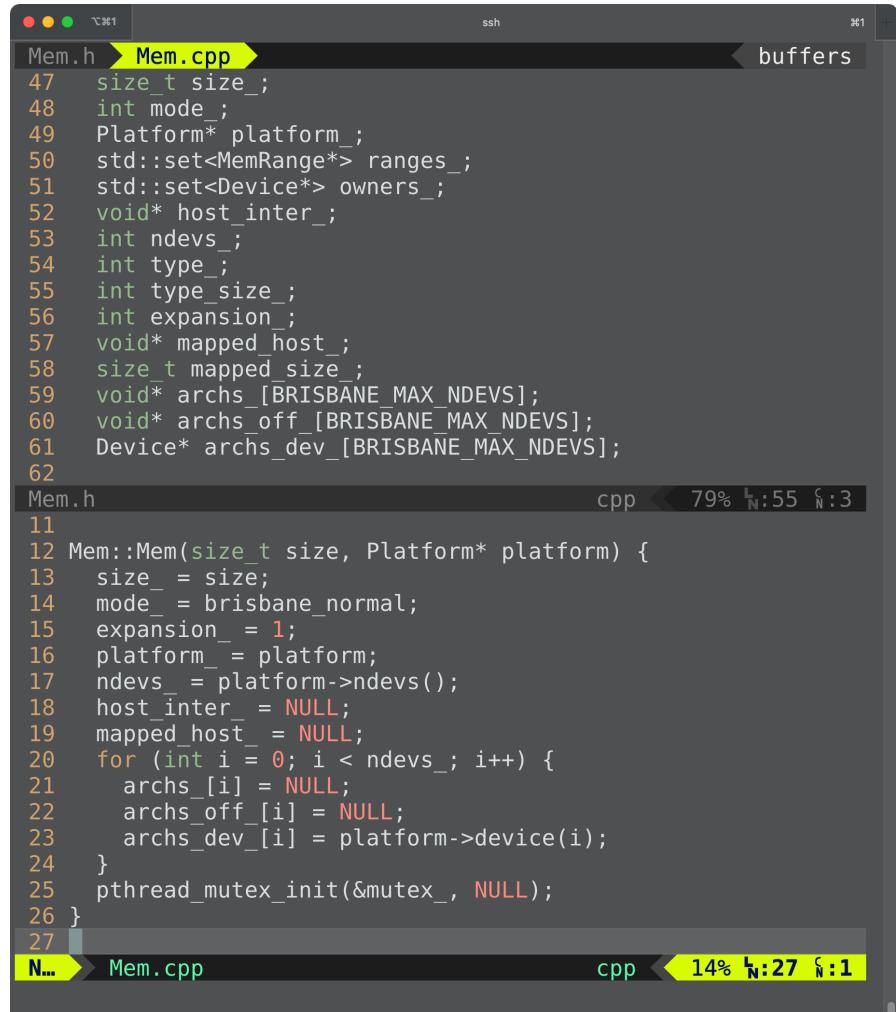
NORMAL ➤ 2tasks.c c utf-8[unix] 50% N:33/65 ≡ 1:1

```
Platform.cpp    CAPI.cpp    buffers
98 }
99
100 int iris_task_release(iris_task task) {
101    return Platform::GetPlatform()->TaskRelease(task);
102 }
103
104 int iris_mem_create(size_t size, iris_mem* mem) {
105    return Platform::GetPlatform()->MemCreate(size, mem);
106 }
107
108 int iris_mem_release(iris_mem mem) {
109    return Platform::GetPlatform()->MemRelease(mem);
110 }
111
112 int iris_timer_now(double* time) {
113    return Platform::GetPlatform()->TimerNow(time);
114 }
CAPI.cpp    cpp 28% N:111 1:1
839 Task* task = brs_task->class_obj;
840 Mem* mem = brs_mem->class_obj;
841 Command* cmd = Command::CreateReleaseMem(task, mem);
842 task->AddCommand(cmd);
843 return BRISBANE_OK;
844 }
845
846 int Platform::MemCreate(size_t size, brisbane_mem* brs_mem)
847 {
848    Mem* mem = new Mem(size, this);
849    if (brisbane_mem *brs_mem = mem->struct_obj());
850    mems_.insert(mem);
851    return BRISBANE_OK;
852 }
```

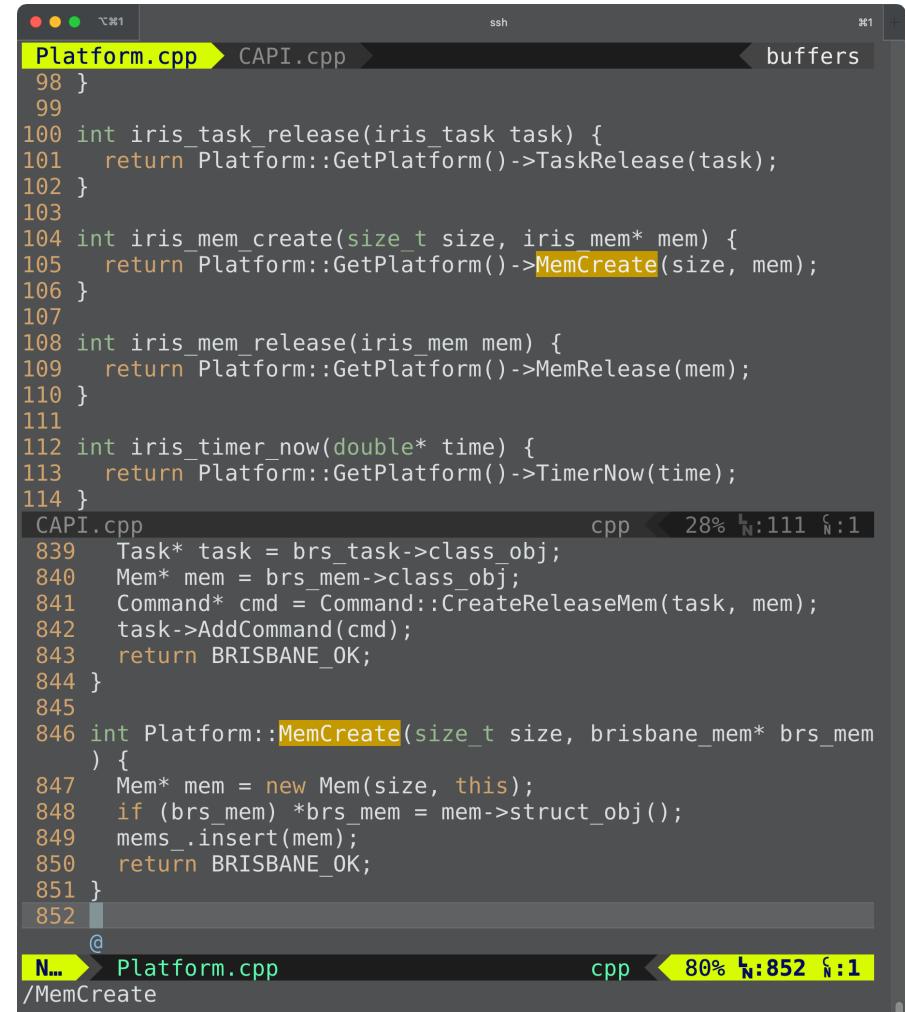
Platform.cpp ➤ CAPI.cpp ➤ buffers

Platform.cpp ➤ Platform.cpp 80% N:852 1:1

# src/runtime/Mem.cpp



```
Mem.h > Mem.cpp buffers
47     size_t size_;
48     int mode_;
49     Platform* platform_;
50     std::set<MemRange*> ranges_;
51     std::set<Device*> owners_;
52     void* host_inter_;
53     int ndevs_;
54     int type_;
55     int type_size_;
56     int expansion_;
57     void* mapped_host_;
58     size_t mapped_size_;
59     void* archs_[BRISBANE_MAX_NDEVS];
60     void* archs_off_[BRISBANE_MAX_NDEVS];
61     Device* archs_dev_[BRISBANE_MAX_NDEVS];
62
Mem.h                                         cpp    79%  N:55  ⏴:3
11
12 Mem::Mem(size_t size, Platform* platform) {
13     size_ = size;
14     mode_ = brisbane_normal;
15     expansion_ = 1;
16     platform_ = platform;
17     ndevs_ = platform->ndevs();
18     host_inter_ = NULL;
19     mapped_host_ = NULL;
20     for (int i = 0; i < ndevs_; i++) {
21         archs_[i] = NULL;
22         archs_off_[i] = NULL;
23         archs_dev_[i] = platform->device(i);
24     }
25     pthread_mutex_init(&mutex_, NULL);
26 }
27
N... > Mem.cpp                                         cpp < 14%  N:27  ⏴:1
```



```
Platform.cpp > CAPI.cpp buffers
98 }
99
100 int iris_task_release(iris_task task) {
101     return Platform::GetPlatform()->TaskRelease(task);
102 }
103
104 int iris_mem_create(size_t size, iris_mem* mem) {
105     return Platform::GetPlatform()->MemCreate(size, mem);
106 }
107
108 int iris_mem_release(iris_mem mem) {
109     return Platform::GetPlatform()->MemRelease(mem);
110 }
111
112 int iris_timer_now(double* time) {
113     return Platform::GetPlatform()->TimerNow(time);
114 }
CAPI.cpp                                         cpp    28%  N:111  ⏴:1
839     Task* task = brs_task->class_obj;
840     Mem* mem = brs_mem->class_obj;
841     Command* cmd = Command::CreateReleaseMem(task, mem);
842     task->AddCommand(cmd);
843     return BRISBANE_OK;
844 }
845
846 int Platform::MemCreate(size_t size, brisbane_mem* brs_mem
847 ) {
848     Mem* mem = new Mem(size, this);
849     if (brs_mem) *brs_mem = mem->struct_obj();
850     mems_.insert(mem);
851     return BRISBANE_OK;
852 }
853
N... > Platform.cpp                                         cpp < 80%  N:852  ⏴:1
/MemCreate
```

# iris\_task\_h2d\_full()

A screenshot of a terminal window titled "buffers". Inside, there is a code editor showing the "2tasks.c" file. The code defines two tasks, task0 and task1, which perform matrix multiplication from host memory to device memory. Task0 multiplies X by Y to produce Z. Task1 multiplies Y by Z to produce X. Both tasks are submitted to the iris\_gpu. After the tasks are completed, the results are printed to the console. The code uses the IRIS API for task creation, execution, and memory management.

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

NORMAL 2tasks.c c utf-8[unix] 98% N:64/65 ≡ 1

A screenshot of a terminal window titled "buffers". Inside, there is a code editor showing the "Platform.cpp" file. This file contains implementations for various task-related functions. The "TaskH2D" function creates a task to copy from host memory to device memory. The "TaskD2H" function creates a task to copy from device memory to host memory. The "TaskH2DFull" and "TaskD2HFull" functions are similar to their counterparts but also handle the release of host memory after the task is completed. The code uses pointers to class objects and command objects to manage the task creation process.

```
Platform.cpp buffers
726
727     int Platform::TaskH2D(brisbane_task brs_task, brisbane_mem
728         brs_mem, size_t off, size_t size, void* host) {
729         Task* task = brs_task->class_obj;
730         Mem* mem = brs_mem->class_obj;
731         Command* cmd = Command::CreateH2D(task, mem, off, size,
732             host);
733         task->AddCommand(cmd);
734         return BRISBANE_OK;
735     }
736
737     int Platform::TaskD2H(brisbane_task brs_task, brisbane_mem
738         brs_mem, size_t off, size_t size, void* host) {
739         Task* task = brs_task->class_obj;
740         Mem* mem = brs_mem->class_obj;
741         Command* cmd = Command::CreateD2H(task, mem, off, size,
742             host);
743         task->AddCommand(cmd);
744         return BRISBANE_OK;
745     }
746
747     int Platform::TaskH2DFull(brisbane_task brs_task, brisbane_
748         _mem brs_mem, void* host) {
749         return TaskH2D(brs_task, brs_mem, 0ULL, brs_mem->class_o
750             bj->size(), host);
751     }
752
753     int Platform::TaskD2HFull(brisbane_task brs_task, brisbane_
754         _mem brs_mem, void* host) {
755         return TaskD2H(brs_task, brs_mem, 0ULL, brs_mem->class_o
756             bj->size(), host);
757     }
758
759     @
760 N... Platform.cpp cpp 70% N:746 1
```

# iris\_task\_kernel()

A screenshot of a terminal window titled "buffers". Inside, there is a code editor showing the file "2tasks.c". The code defines two tasks, task0 and task1, which perform matrix multiplication (A = B \* C) and print the result. Task0 takes matrices B and C as input and produces matrix A. Task1 takes matrix A and matrix C as input and produces matrix B. The code uses the IRIS API for task creation, memory allocation, and command submission.

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

NORMAL ➤ 2tasks.c c utf-8[unix] 98% N:64/65 ⌂ 1

A screenshot of a terminal window titled "buffers". Inside, there is a code editor showing the file "Platform.cpp". This file contains implementations for various IRIS API functions. The "TaskKernel" function creates a task object, initializes its parameters, and adds a command to the task's queue. The "TaskKernelSelector" function creates a command object, sets its selector kernel, and adds it to the task's queue. The "TaskHost" function creates a host task object and adds a command to its queue.

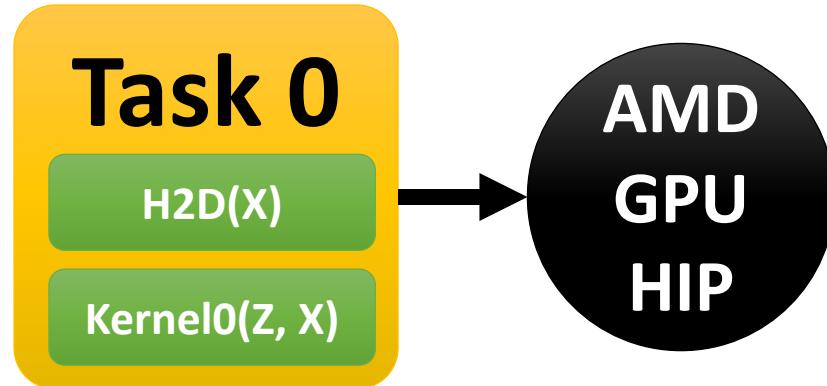
```
Platform.cpp buffers
692     task->AddCommand(cmd);
693     return BRISBANE_OK;
694 }
695
696 int Platform::TaskKernel(brisbane_task brs_task, const char* name, int dim, size_t* off, size_t* gws, size_t* lws, int nparams, void** params, size_t* params_off, int* params_info, size_t* memranges) {
697     Task* task = brs_task->class_obj;
698     Kernel* kernel = GetKernel(name);
699     Command* cmd = Command::CreateKernel(task, kernel, dim, off, gws, lws, nparams, params, params_off, params_info, memranges);
700     task->AddCommand(cmd);
701     return BRISBANE_OK;
702 }
703
704 int Platform::TaskKernelSelector(brisbane_task brs_task, brisbane_selector_kernel func, void* params, size_t params_size) {
705     Task* task = brs_task->class_obj;
706     Command* cmd = task->cmd_kernel();
707     if (!cmd) return BRISBANE_ERR;
708     cmd->set_selector_kernel(func, params, params_size);
709     return BRISBANE_OK;
710 }
711
712 int Platform::TaskHost(brisbane_task brs_task, brisbane_host_task func, void* params) {
713     Task* task = brs_task->class_obj;
714     Command* cmd = Command::CreateHost(task, func, params);
715     task->AddCommand(cmd);
716     return BRISBANE_OK;
717 }
```

N... ➤ Platform.cpp cpp 66% N:704 ⌂ 15

# iris\_task\_submit()

```
2tasks.c      buffers
34  iris_task task0;
35  iris_task_create(&task0);
36  iris_task_h2d_full(task0, mem_X, X);
37  void* task0_params[2] = { mem_Z, mem_X };
38  int task0_params_info[2] = { iris_w, iris_r };
39  iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40  iris_task_submit(task0, iris_gpu, NULL, 1);
41
42  iris_task task1;
43  iris_task_create(&task1);
44  iris_task_h2d_full(task1, mem_Y, Y);
45  void* task1_params[2] = { mem_Z, mem_Y };
46  int task1_params_info[2] = { iris_rw, iris_r };
47  iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48  iris_task_d2h_full(task1, mem_Z, Z);
49  iris_task_submit(task1, iris_cpu, NULL, 1);
50
51  printf("Z [ ");
52  for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53  printf("]\n");
54
55  iris_task_release(task0);
56  iris_task_release(task1);
57
58  iris_mem_release(mem_X);
59  iris_mem_release(mem_Y);
60  iris_mem_release(mem_Z);
61
62  iris_finalize();
63  return 0;
64 }
```

NORMAL ➤ 2tasks.c c utf-8[unix] 98% N:64/65 ≡ 1



# iris\_task\_submit()

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

The terminal shows the file 2tasks.c with line numbers 34 to 64. The code creates two tasks, task0 and task1, using iris\_task\_create(). Task0 is a kernel0 task with parameters mem\_X and mem\_Z. Task1 is a kernel1 task with parameters mem\_Y and mem\_Z. Both tasks are submitted to the iris\_gpu device. The code then prints the value of Z. Finally, it releases the memory and finalizes the system.

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
75
76 void Scheduler::Enqueue(Task* task) {
77     while (!queue_->Enqueue(task)) { }
78     Invoke();
79 }
80
81 void Scheduler::Run() {
82     while (true) {
83         Sleep();
84         if (!running_) break;
85         Task* task = NULL;
86         while (queue_->Dequeue(&task)) Submit(task);
87     }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
91     Scheduler.cpp CPP 62% N:90 F:39
92     ...
93 }
94
95
96 int Platform::TaskSubmit(brisbane_task brs_task, int brs_policy, const char* opt, int sync) {
97     Task* task = brs_task->class_obj;
98     task->Submit(brs_policy, opt, sync);
99     if (recording_) json_->RecordTask(task);
100    if (scheduler_) {
101        FilterSubmitExecute(task);
102        scheduler_->Enqueue(task);
103    } else workers_[0]->Enqueue(task);
104    if (sync) task->Wait();
105    return BRISBANE_OK;
106 }
107
108 int Platform::TaskWait(brisbane_task brs_task) {
109     Task* task = brs_task->class_obj;
110     Platform.cpp CPP 76% N:809 F:15
```

The terminal shows the Scheduler.cpp file with line numbers 75 to 110. It defines the Enqueue and Run methods for the Scheduler. The Enqueue method adds tasks to a queue and calls the Invoke method. The Run method repeatedly sleeps, checks if it's running, dequeues tasks from the queue, and submits them directly or through the scheduler. The TaskSubmit method delegates to the Platform::TaskSubmit method, which handles the actual submission logic based on policy, options, and synchronization requirements.

# iris\_task\_submit()

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

The terminal shows the file 2tasks.c with line numbers 34 to 64. The code creates two tasks, task0 and task1, using iris\_task\_create(). Task0 is a kernel0 task with parameters mem\_X and mem\_Z. Task1 is a kernel1 task with parameters mem\_Y and mem\_Z. Both tasks are submitted to the iris\_gpu device. The code then prints the value of Z. Finally, it releases the memory and finalizes the IRIS environment.

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
75
76 void Scheduler::Enqueue(Task* task) {
77     while (!queue_->Enqueue(task)) { }
78     Invoke();
79 }
80
81 void Scheduler::Run() {
82     while (true) {
83         Sleep();
84         if (!running_) break;
85         Task* task = NULL;
86         while (queue_->Dequeue(&task)) Submit(task);
87     }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
91     Scheduler.cpp CPP 62% N:90 F:39
92     ...
93 }
94
95
96 int Platform::TaskSubmit(brisbane_task brs_task, int brs_policy, const char* opt, int sync) {
97     Task* task = brs_task->class_obj;
98     task->Submit(brs_policy, opt, sync);
99     if (recording_) json_->RecordTask(task);
100    if (scheduler_) {
101        FilterSubmitExecute(task);
102        scheduler_->Enqueue(task);
103    } else workers_[0]->Enqueue(task);
104    if (sync) task->Wait();
105    return BRISBANE_OK;
106 }
107
108 int Platform::TaskWait(brisbane_task brs_task) {
109     Task* task = brs_task->class_obj;
110     Platform.cpp CPP 76% N:809 F:15
```

The terminal shows the Scheduler.cpp file with line numbers 75 to 110. It defines the Enqueue and Run methods for the Scheduler. The Enqueue method adds tasks to a queue and calls the Invoke method. The Run method enters a loop, sleeps, and then dequeues and submits tasks from the queue. The SubmitTaskDirect method delegates to the Platform::TaskSubmit method. The Platform::TaskSubmit method creates a Task object, sets its policy, options, and sync flag, and then adds it to the scheduler's queue or workers' queue depending on the sync flag. It also records the task in the json\_ object if recording is enabled.

# src/runtime/QueueTask.cpp

```
QueueTask.cpp > Task.cpp > buffers
97     cmd->type() == BRISBANE_CMD_H2DNP || cmd->type() == BRISBANE_CMD_D2H)
98     cmd_last_ = cmd;
99 }
100 void Task::ClearCommands() {
101     for (int i = 0; i < ncmds_; i++) delete cmd[i];
102     ncmds_ = 0;
103 }
104
105 bool Task::Dispatchable() {
106     for (int i = 0; i < ndepends_; i++) {
107         if (depends_[i]->status() != BRISBANE_COMPLETE) return false;
108     }
109     return true;
110 }
111
112 bool Task::Executable() {
113     pthread_mutex_lock(&mutex_executable_);
114     if (status_ == BRISBANE_NONE || (perm_ && status_ == BRISBANE_COMPLETE)) {
115         status_ = BRISBANE_RUNNING;
116     }
117 }
118
119 bool QueueTask::Dequeue(Task** task) {
120     pthread_mutex_lock(&mutex_);
121     if (tasks_.empty()) {
122         pthread_mutex_unlock(&mutex_);
123         return false;
124     }
125     for (std::list<Task*>::iterator I = tasks_.begin(), E = tasks_.end(); I != E; ++I) {
126         Task* t = *I;
127         if (!t->Dispatchable()) continue;
128         if (t->marker() && I != tasks_.begin()) continue;
129         *task = t;
130         //todo: debug this!
131         tasks_.erase(I);
132         pthread_mutex_unlock(&mutex_);
133         return true;
134     }
135     pthread_mutex_unlock(&mutex_);
136     return false;
137 }
NORMAL > QueueTask.cpp      cpp 53% 1:37/69 1:1
E486: Pattern not found: dsf
```

```
...> Structs.h > CAPI.cpp > Scheduler.cpp > buffers
75
76 void Scheduler::Enqueue(Task* task) {
77     while (!queue_->Enqueue(task)) { }
78     Invoke();
79 }
80
81 void Scheduler::Run() {
82     while (true) {
83         Sleep();
84         if (!running_) break;
85         Task* task = NULL;
86         while (queue_->Dequeue(&task)) Submit(task);
87     }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
91     N... > Scheduler.cpp      cpp 62% 1:90 1:39
92 }
93
94 int Platform::TaskSubmit(brisbane_task brs_task, int brs_p
95 olicy, const char* opt, int sync) {
96     Task* task = brs_task->class_obj;
97     task->Submit(brs_policy, opt, sync);
98     if (recording_) json_->RecordTask(task);
99     if (scheduler_) {
100         FilterSubmitExecute(task);
101         scheduler_->Enqueue(task);
102     } else workers_[0]->Enqueue(task);
103     if (sync) task->Wait();
104     return BRISBANE_OK;
105 }
106
107 int Platform::TaskWait(brisbane_task brs_task) {
108     Task* task = brs_task->class_obj;
109     Platform.cpp      cpp 76% 1:809 1:15
110 }
```

# src/runtime/Scheduler.cpp

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
94
95 void Scheduler::Submit(Task* task) {
96     if (!ndeps_) {
97         if (!task->marker()) _error("%s", "no device");
98         task->Complete();
99         return;
100    }
101    if (task->marker()) {
102        std::vector<Task*>* subtasks = task->subtasks();
103        for (std::vector<Task*>::iterator I = subtasks->begin(),
104             E = subtasks->end(); I != E; ++I) {
105            Task* subtask = *I;
106            int dev = subtask->devno();
107            workers_[dev]->Enqueue(subtask);
108        }
109    }
110    if (!task->HasSubtasks()) {
111        SubmitTask(task);
112        return;
113    }
114    std::vector<Task*>* subtasks = task->subtasks();
115    for (std::vector<Task*>::iterator I = subtasks->begin(),
116         E = subtasks->end(); I != E; ++I)
117        SubmitTask(*I);
118    }
119 void Scheduler::SubmitTask(Task* task) {
120     int brs_policy = task->brs_policy();
121     char* opt = task->opt();
122     int ndeps = 0;
123     Device* devs[BRISBANE_MAX_NDEVS];
124     if (brs_policy < BRISBANE_MAX_NDEVS) {
125         if (brs_policy >= ndeps_) ndeps = 0;
N... > Scheduler.cpp          cpp 86% N:125 F:17
```

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
75
76 void Scheduler::Enqueue(Task* task) {
77     while (!queue_->Enqueue(task)) { }
78     Invoke();
79 }
80
81 void Scheduler::Run() {
82     while (true) {
83         Sleep();
84         if (!running_) break;
85         Task* task = NULL;
86         while (queue_->Dequeue(&task)) Submit(task);
87     }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
N... > Scheduler.cpp          cpp 62% N:90 F:39
91 }
92
93 int Platform::TaskSubmit(brisbane_task brs_task, int brs_p
94     olicy, const char* opt, int sync) {
95     Task* task = brs_task->class_obj;
96     task->Submit(brs_policy, opt, sync);
97     if (recording_) json_->RecordTask(task);
98     if (scheduler_) {
99         FilterSubmitExecute(task);
100        scheduler_->Enqueue(task);
101    } else workers_[0]->Enqueue(task);
102    if (sync) task->Wait();
103    return BRISBANE_OK;
104 }
105
106 int Platform::TaskWait(brisbane_task brs_task) {
107     Task* task = brs_task->class_obj;
Platform.cpp          cpp 76% N:809 F:15
```

# src/runtime/Scheduler.cpp

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
94
95 void Scheduler::Submit(Task* task) {
96     if (!ndevs_) {
97         if (!task->marker()) _error("%s", "no device");
98         task->Complete();
99         return;
100    }
101   if (task->marker()) {
102       std::vector<Task*>* subtasks = task->subtasks();
103       for (std::vector<Task*>::iterator I = subtasks->begin(),
104 , E = subtasks->end(); I != E; ++I) {
105           Task* subtask = *I;
106           int dev = subtask->devno();
107           workers_[dev]->Enqueue(subtask);
108       }
109   }
110   if (!task->HasSubtasks()) {
111       SubmitTask(task);
112       return;
113   }
114   std::vector<Task*>* subtasks = task->subtasks();
115   for (std::vector<Task*>::iterator I = subtasks->begin(),
116 E = subtasks->end(); I != E; ++I)
117       SubmitTask(*I);
118
119 void Scheduler::SubmitTask(Task* task) {
120     int brs_policy = task->brs_policy();
121     char* opt = task->opt();
122     int ndevs = 0;
123     Device* devs[BRISBANE_MAX_NDEVS];
124     if (brs_policy < BRISBANE_MAX_NDEVS) {
125         if (brs_policy >= ndevs_) ndevs = 0;
126     } else {
127         ndevs = 1;
128         devs[0] = devs_[brs_policy];
129     }
130 } else policies_->GetPolicy(brs_policy, opt)->GetDevices(
131     task, devs, &ndevs);
132     if (ndevs == 0) {
133         int dev_default = platform ->device_default();
134         _trace("no device for policy[0x%x], run the task on dev
135 ice[%d]", brs_policy, dev_default);
136         ndevs = 1;
137         devs[0] = devs_[dev_default];
138     }
139     for (int i = 0; i < ndevs; i++) {
140         devs[i]->worker()->Enqueue(task);
141         if (hub_available_) hub_client_->TaskInc(devs[i]->devno
142 (), 1);
143     }
144 } /* namespace rt */
N... Scheduler.cpp      cpp      86% h:125 l:17
```

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
114     std::vector<Task*>* subtasks = task->subtasks();
115     for (std::vector<Task*>::iterator I = subtasks->begin(),
116 E = subtasks->end(); I != E; ++I)
117         SubmitTask(*I);
118
119 void Scheduler::SubmitTask(Task* task) {
120     int brs_policy = task->brs_policy();
121     char* opt = task->opt();
122     int ndevs = 0;
123     Device* devs[BRISBANE_MAX_NDEVS];
124     if (brs_policy < BRISBANE_MAX_NDEVS) {
125         if (brs_policy >= ndevs_) ndevs = 0;
126     } else {
127         ndevs = 1;
128         devs[0] = devs_[brs_policy];
129     }
130 } else policies_->GetPolicy(brs_policy, opt)->GetDevices(
131     task, devs, &ndevs);
132     if (ndevs == 0) {
133         int dev_default = platform ->device_default();
134         _trace("no device for policy[0x%x], run the task on dev
135 ice[%d]", brs_policy, dev_default);
136         ndevs = 1;
137         devs[0] = devs_[dev_default];
138     }
139     for (int i = 0; i < ndevs; i++) {
140         devs[i]->worker()->Enqueue(task);
141         if (hub_available_) hub_client_->TaskInc(devs[i]->devno
142 (), 1);
143     }
144 } /* namespace rt */
N... Scheduler.cpp      cpp      99% h:143 l:5
```

# src/runtime/Policies.cpp

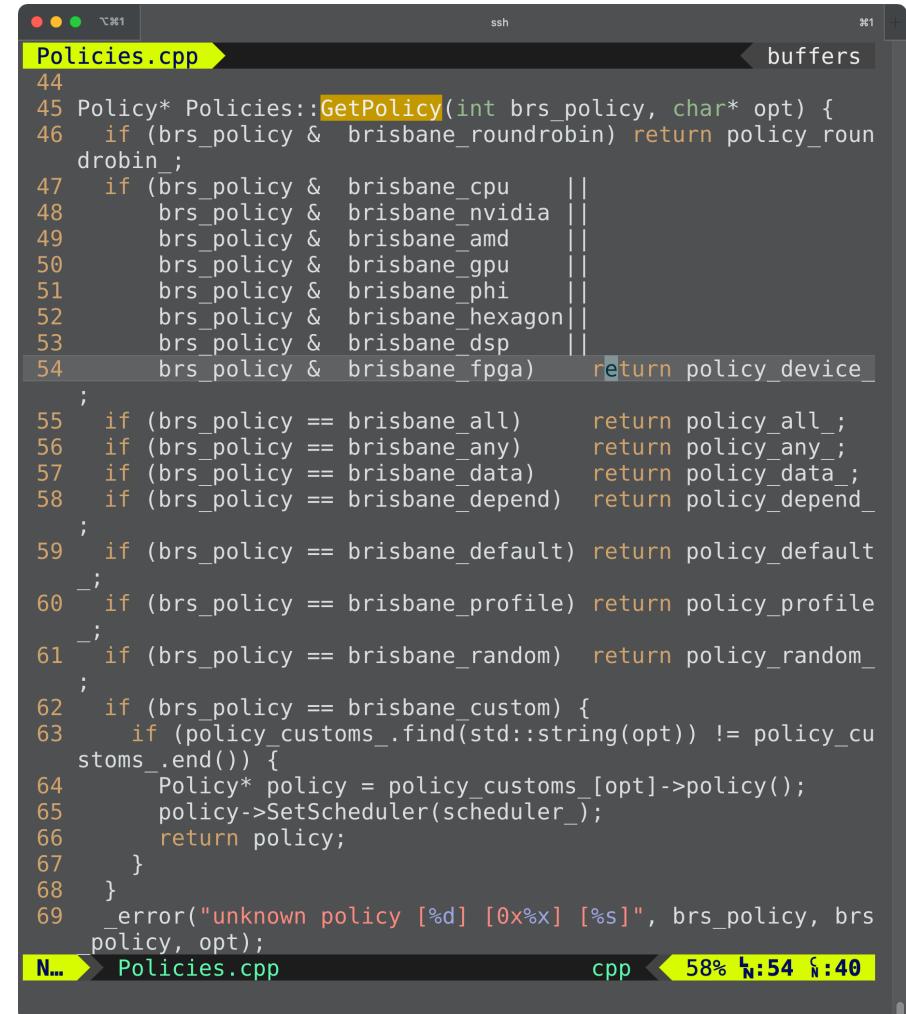
```
Policies.cpp
5 #include "PolicyAny.h"
6 #include "PolicyData.h"
7 #include "PolicyDefault.h"
8 #include "PolicyDepend.h"
9 #include "PolicyDevice.h"
10 #include "PolicyProfile.h"
11 #include "PolicyRandom.h"
12 #include "PolicyRoundRobin.h"
13 #include "Platform.h"
14
15 namespace brisbane {
16 namespace rt {
17
18 Policies::Policies(Scheduler* scheduler) {
19     scheduler_ = scheduler;
20     policy_all_ = new PolicyAll(scheduler_);
21     policy_any_ = new PolicyAny(scheduler_);
22     policy_data_ = new PolicyData(scheduler_);
23     policy_default_ = new PolicyDefault(scheduler_);
24     policy_depend_ = new PolicyDepend(scheduler_);
25     policy_device_ = new PolicyDevice(scheduler_);
26     policy_profile_ = new PolicyProfile(scheduler_, this)
27 ;
28     policy_random_ = new PolicyRandom(scheduler_);
29     policy_roundrobin_ = new PolicyRoundRobin(scheduler_);
30 }
31 Policies::~Policies() {
32     delete policy_all_;
33     delete policy_any_;
34     delete policy_data_;
35     delete policy_default_;
36     delete policy_depend_;
37     delete policy_device_;
38 }
```

```
...> Structs.h > CAPI.cpp > Scheduler.cpp
114     std::vector<Task*>* subtasks = task->subtasks();
115     for (std::vector<Task*>::iterator I = subtasks->begin(),
116         E = subtasks->end(); I != E; ++I)
117         SubmitTask(*I);
118
119 void Scheduler::SubmitTask(Task* task) {
120     int brs_policy = task->brs_policy();
121     char* opt = task->opt();
122     int nDevs = 0;
123     Device* devs[BRISBANE_MAX_NDEVS];
124     if (brs_policy < BRISBANE_MAX_NDEVS) {
125         if (brs_policy >= nDevs_) nDevs = 0;
126         else {
127             nDevs = 1;
128             devs[0] = devs_[brs_policy];
129         }
130     } else policies_->GetPolicy(brs_policy, opt)->GetDevices(
131         task, devs, &nDevs);
132     if (nDevs == 0) {
133         int dev_default = platform_->device_default();
134         _trace("no device for policy[0x%x], run the task on dev
135             ice[%d]", brs_policy, dev_default);
136         nDevs = 1;
137         devs[0] = devs_[dev_default];
138     }
139     for (int i = 0; i < nDevs; i++) {
140         devs[i]->worker()->Enqueue(task);
141         if (hub_available_) hub_client_->TaskInc(devs[i]->devno
142             (), 1);
143     }
144 }
```

# src/runtime/Policies.cpp



```
5 #include "PolicyAny.h"
6 #include "PolicyData.h"
7 #include "PolicyDefault.h"
8 #include "PolicyDepend.h"
9 #include "PolicyDevice.h"
10 #include "PolicyProfile.h"
11 #include "PolicyRandom.h"
12 #include "PolicyRoundRobin.h"
13 #include "Platform.h"
14
15 namespace brisbane {
16 namespace rt {
17
18 Policies::Policies(Scheduler* scheduler) {
19     scheduler_ = scheduler;
20     policy_all_ = new PolicyAll(scheduler_);
21     policy_any_ = new PolicyAny(scheduler_);
22     policy_data_ = new PolicyData(scheduler_);
23     policy_default_ = new PolicyDefault(scheduler_);
24     policy_depend_ = new PolicyDepend(scheduler_);
25     policy_device_ = new PolicyDevice(scheduler_);
26     policy_profile_ = new PolicyProfile(scheduler_, this)
27     ;
28     policy_random_ = new PolicyRandom(scheduler_);
29     policy_roundrobin_ = new PolicyRoundRobin(scheduler_);
30 }
31 Policies::~Policies() {
32     delete policy_all_;
33     delete policy_any_;
34     delete policy_data_;
35     delete policy_default_;
36     delete policy_depend_;
37     delete policy_device_;
38 }
```



```
44
45 Policy* Policies::GetPolicy(int brs_policy, char* opt) {
46     if (brisbane_roundrobin) return policy_roundrobin_;
47     if (brisbane_policy & brisbane_cpu || brs_policy & brisbane_nvidia || brs_policy & brisbane_amd || brs_policy & brisbane_gpu || brs_policy & brisbane_phi || brs_policy & brisbane_hexagon || brs_policy & brisbane_dsp || brs_policy & brisbane_fpga) return policy_device_
48     ;
49     if (brisbane_policy == brisbane_all) return policy_all_;
50     if (brisbane_policy == brisbane_any) return policy_any_;
51     if (brisbane_policy == brisbane_data) return policy_data_;
52     if (brisbane_policy == brisbane_depend) return policy_depend_;
53     if (brisbane_policy == brisbane_default) return policy_default_;
54     if (brisbane_policy == brisbane_profile) return policy_profile_;
55     if (brisbane_policy == brisbane_random) return policy_random_;
56     if (brisbane_policy == brisbane_custom) {
57         if (policy_customs_.find(std::string(opt)) != policy_customs_.end()) {
58             Policy* policy = policy_customs_[opt]->policy();
59             policy->SetScheduler(scheduler_);
60             return policy;
61         }
62     }
63     _error("unknown policy [%d] [0x%08x] [%s]", brs_policy, brs_policy, opt);
64 }
```

# src/runtime/PolicyDevice.cpp

```
PolicyDevice.cpp
1 #include "PolicyDevice.h"
2 #include "Debug.h"
3 #include "Device.h"
4 #include "Task.h"
5
6 namespace brisbane {
7 namespace rt {
8
9 PolicyDevice::PolicyDevice(Scheduler* scheduler) {
10    SetScheduler(scheduler);
11 }
12
13 PolicyDevice::~PolicyDevice() {
14 }
15
16 void PolicyDevice::GetDevices(Task* task, Device** devs, int* ndevs) {
17    int brs_policy = task->brs_policy();
18    int n = 0;
19    for (int i = 0; i < ndevs_; i++) {
20        Device* dev = devs_[i];
21        if ((dev->type() & brs_policy) == dev->type()) {
22            devs[n++] = dev;
23        }
24    }
25    *ndevs = n;
26 }
27
28 } /* namespace rt */
29 } /* namespace brisbane */
```

N... PolicyDevice.cpp cpp 100% N:29 F:1  
"PolicyDevice.cpp" 29L, 576C

```
Policies.cpp
44
45 Policy* Policies::GetPolicy(int brs_policy, char* opt) {
46    if (brisbane_roundrobin) return policy_roundrobin_;
47    if (brisbane_policy & brisbane_cpu) ||
48        brs_policy & brisbane_nvidia ||
49        brs_policy & brisbane_amd ||
50        brs_policy & brisbane_gpu ||
51        brs_policy & brisbane_phi ||
52        brs_policy & brisbane_hexagon ||
53        brs_policy & brisbane_dsp ||
54        brs_policy & brisbane_fpga) return policy_device_;
55    if (brisbane_policy == brisbane_all) return policy_all_;
56    if (brisbane_policy == brisbane_any) return policy_any_;
57    if (brisbane_policy == brisbane_data) return policy_data_;
58    if (brisbane_policy == brisbane_depend) return policy_depend_;
59    if (brisbane_policy == brisbane_default) return policy_default_;
60    if (brisbane_policy == brisbane_profile) return policy_profile_;
61    if (brisbane_policy == brisbane_random) return policy_random_;
62    if (brisbane_policy == brisbane_custom) {
63        if (policy_customs_.find(std::string(opt)) != policy_customs_.end()) {
64            Policy* policy = policy_customs_[opt]->policy();
65            policy->SetScheduler(scheduler_);
66            return policy;
67        }
68    }
69    _error("unknown policy [%d] [0x%x] [%s]", brs_policy, brs_policy, opt);
```

N... Policies.cpp cpp 58% N:54 F:40

# src/runtime/PolicyDevice.cpp

```
PolicyDevice.cpp buffers
1 #include "PolicyDevice.h"
2 #include "Debug.h"
3 #include "Device.h"
4 #include "Task.h"
5
6 namespace brisbane {
7 namespace rt {
8
9 PolicyDevice::PolicyDevice(Scheduler* scheduler) {
10    SetScheduler(scheduler);
11 }
12
13 PolicyDevice::~PolicyDevice() {
14 }
15
16 void PolicyDevice::GetDevices(Task* task, Device** devs, int* ndevs) {
17    int brs_policy = task->brs_policy();
18    int n = 0;
19    for (int i = 0; i < ndevs_; i++) {
20        Device* dev = devs_[i];
21        if ((dev->type() & brs_policy) == dev->type()) {
22            devs[n++] = dev;
23        }
24    }
25    *ndevs = n;
26 }
27
28 } /* namespace rt */
29 } /* namespace brisbane */
```

~  
~  
~  
N... PolicyDevice.cpp cpp 100% h:29 l:1  
"PolicyDevice.cpp" 29L, 576C

```
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers
114    std::vector<Task*>* subtasks = task->subtasks();
115    for (std::vector<Task*>::iterator I = subtasks->begin(),
116         E = subtasks->end(); I != E; ++I)
117        SubmitTask(*I);
118
119 void Scheduler::SubmitTask(Task* task) {
120    int brs_policy = task->brs_policy();
121    char* opt = task->opt();
122    int ndevs = 0;
123    Device* devs[BRISBANE_MAX_NDEVS];
124    if (brs_policy < BRISBANE_MAX_NDEVS) {
125        if (brs_policy >= ndevs_) ndevs = 0;
126        else {
127            ndevs = 1;
128            devs[0] = devs_[brs_policy];
129        }
130    } else policies_->GetPolicy(brs_policy, opt)->GetDevices(
131        task, devs, &ndevs);
132    if (ndevs == 0) {
133        int dev_default = platform_->device_default();
134        _trace("no device for policy[0x%x], run the task on dev
ice[%d]", brs_policy, dev_default);
135        ndevs = 1;
136        devs[0] = devs_[dev_default];
137    }
138    for (int i = 0; i < ndevs; i++) {
139        devs[i]->worker()->Enqueue(task);
140        if (hub_available_) hub_client_->TaskInc(devs[i]->devno
(), 1);
141    }
142
143 } /* namespace rt */
N... Scheduler.cpp cpp 99% h:143 l:5
```

# src/runtime/Device.cpp

```
Device.cpp
35     delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%x]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61 #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //    if (++q_ >= nqueues) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", task->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_>Start(BRISBANE_TIMER_INIT);
NORMAL Device.cpp      cpp 27% h:73 l:14 [225]tr...
```

```
...> Structs.h > CAPI.cpp > Scheduler.cpp
114     std::vector<Task*>* subtasks = task->subtasks();
115     for (std::vector<Task*>::iterator I = subtasks->begin(),
116          E = subtasks->end(); I != E; ++I)
117         SubmitTask(*I);
118
119 void Scheduler::SubmitTask(Task* task) {
120     int brs_policy = task->brs_policy();
121     char* opt = task->opt();
122     int ndevs = 0;
123     Device* devs[BRISBANE_MAX_NDEVS];
124     if (brs_policy < BRISBANE_MAX_NDEVS) {
125         if (brs_policy >= ndevs_) ndevs = 0;
126         else {
127             ndevs = 1;
128             devs[0] = devs_[brs_policy];
129         }
130     } else policies_->GetPolicy(brs_policy, opt)->GetDevices(
131         task, devs, &ndevs);
132     if (ndevs == 0) {
133         int dev_default = platform_->device_default();
134         _trace("no device for policy[0x%x], run the task on dev
ice[%d]", brs_policy, dev_default);
135         ndevs = 1;
136         devs[0] = devs_[dev_default];
137     }
138     for (int i = 0; i < ndevs; i++) {
139         devs[i]->worker()->Enqueue(task);
140         if (hub_available_) hub_client_->TaskInc(devs[i]->devno
()), 1);
141     }
142
143 } /* namespace rt */
NORMAL Scheduler.cpp      cpp 99% h:143 l:5
```

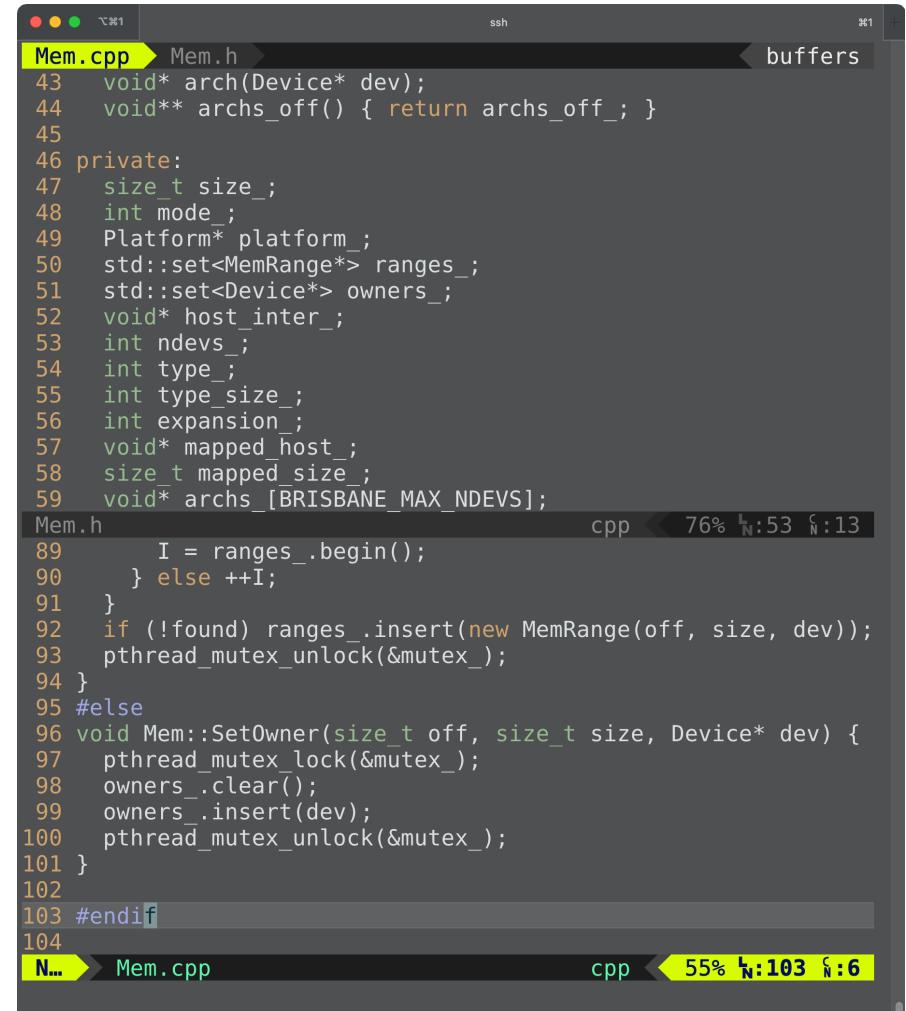
# src/runtime/Device.cpp

```
Device.cpp
35     delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%x]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61 #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //    if (++q_ >= nqueues) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", task->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
NORMAL Device.cpp      cpp 27% 1:73 1:14 [225]tr...
```

```
PolicyDevice.cpp Device.cpp
168     _trace("dev[%d] malloc[%p]", devno_, arch);
169 }
170
171 void Device::ExecuteH2D(Command* cmd) {
172     Mem* mem = cmd->mem();
173     size_t off = cmd->off(0);
174     size_t size = cmd->size();
175     bool exclusive = cmd->exclusive();
176     void* host = cmd->host();
177     if (exclusive) mem->SetOwner(off, size, this);
178     else mem->AddOwner(off, size, this);
179     timer_->Start(BRISBANE_TIMER_H2D);
180     errid_ = MemH2D(mem, off, size, host);
181     if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
182     double time = timer_->Stop(BRISBANE_TIMER_H2D);
183     cmd->SetTime(time);
184     Command* cmd_kernel = cmd->task()->cmd_kernel();
185     if (cmd_kernel) cmd_kernel->kernel()->history()->AddH2D(cmd, this, time);
186     else Platform::GetPlatform()->null_kernel()->history()->AddH2D(cmd, this, time);
187 }
188
189 void Device::ExecuteH2DNP(Command* cmd) {
190     Mem* mem = cmd->mem();
191     size_t off = cmd->off(0);
192     size_t size = cmd->size();
193 //    if (mem->IsOwner(off, size, this)) return;
194     return ExecuteH2D(cmd);
195 }
196
197 void Device::ExecuteD2H(Command* cmd) {
198     Mem* mem = cmd->mem();
199     size_t off = cmd->off(0);
N... Device.cpp      cpp 69% 1:188 1:1
```

# Owner List of IRIS Memory

- An IRIS memory has an owner list
- An owner list can have zero or more owner
- Empty when the memory is created
- An owner of an IRIS memory is a device that has the latest copy of the IRIS memory content



The screenshot shows a terminal window with two tabs: "Mem.cpp" and "Mem.h". The "Mem.h" tab is active, displaying the following C++ code:

```
43     void* arch(Device* dev);
44     void** archs_off() { return archs_off_; }
45
46 private:
47     size_t size_;
48     int mode_;
49     Platform* platform_;
50     std::set<MemRange*> ranges_;
51     std::set<Device*> owners_;
52     void* host_inter_;
53     int ndevs_;
54     int type_;
55     int type_size_;
56     int expansion_;
57     void* mapped_host_;
58     size_t mapped_size_;
59     void* archs_[BRISBANE_MAX_NDEVS];
60
61     Mem.h
62         I = ranges_.begin();
63     } else ++I;
64     }
65     if (!found) ranges_.insert(new MemRange(off, size, dev));
66     pthread_mutex_unlock(&mutex_);
67 }
68 #else
69 void Mem::SetOwner(size_t off, size_t size, Device* dev) {
70     pthread_mutex_lock(&mutex_);
71     owners_.clear();
72     owners_.insert(dev);
73     pthread_mutex_unlock(&mutex_);
74 }
75
76 #endif
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
```

The "Mem.cpp" tab is also visible at the bottom of the window.

# src/runtime/DeviceHIP.cpp

```
DeviceHIP.cpp buffers
100
101 int DeviceHIP::MemFree(void* mem) {
102     void* hipmem = mem;
103     err_ = ld_->hipFree(hipmem);
104     _hiperror(err_);
105     if (err_ != hipSuccess) return BRISBANE_ERR;
106     return BRISBANE_OK;
107 }
108
109 int DeviceHIP::MemH2D(Mem* mem, size_t off, size_t size, void* host) {
110     void* hipmem = mem->arch(this);
111     err_ = ld_->hipMemcpyHtoD((char*) hipmem + off, host, size);
112     _hiperror(err_);
113     if (err_ != hipSuccess) return BRISBANE_ERR;
114     return BRISBANE_OK;
115 }
116
117 int DeviceHIP::MemD2H(Mem* mem, size_t off, size_t size, void* host) {
118     void* hipmem = mem->arch(this);
119     err_ = ld_->hipMemcpyDtoH(host, (char*) hipmem + off, size);
120     _hiperror(err_);
121     if (err_ != hipSuccess) return BRISBANE_ERR;
122     return BRISBANE_OK;
123 }
124
125 int DeviceHIP::KernelGet(void** kernel, const char* name) {
126     hipFunction_t* hipkernel = (hipFunction_t*) kernel;
127     err_ = ld_->hipModuleGetFunction(hipkernel, module_, name);
128     _hiperror(err_);
N... DeviceHIP.cpp      cpp 58% N:124 F:1
```

```
PolicyDevice.cpp Device.cpp buffers
168     _trace("dev[%d] malloc[%p]", devno_, arch);
169 }
170
171 void Device::ExecuteH2D(Command* cmd) {
172     Mem* mem = cmd->mem();
173     size_t off = cmd->off(0);
174     size_t size = cmd->size();
175     bool exclusive = cmd->exclusive();
176     void* host = cmd->host();
177     if (exclusive) mem->SetOwner(off, size, this);
178     else mem->AddOwner(off, size, this);
179     timer_->Start(BRISBANE_TIMER_H2D);
180     errid_ = MemH2D(mem, off, size, host);
181     if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
182     double time = timer_->Stop(BRISBANE_TIMER_H2D);
183     cmd->SetTime(time);
184     Command* cmd_kernel = cmd->task()->cmd_kernel();
185     if (cmd_kernel) cmd_kernel->kernel()->history()->AddH2D(cmd, this, time);
186     else Platform::GetPlatform()->null_kernel()->history()->AddH2D(cmd, this, time);
187 }
188
189 void Device::ExecuteH2DNP(Command* cmd) {
190     Mem* mem = cmd->mem();
191     size_t off = cmd->off(0);
192     size_t size = cmd->size();
193     // if (mem->IsOwner(off, size, this)) return;
194     return ExecuteH2D(cmd);
195 }
196
197 void Device::ExecuteD2H(Command* cmd) {
198     Mem* mem = cmd->mem();
199     size_t off = cmd->off(0);
N... Device.cpp      cpp 69% N:188 F:1
```

# src/runtime/DeviceHIP.cpp

```
DeviceHIP.cpp buffers
100
101 int DeviceHIP::MemFree(void* mem) {
102     void* hipmem = mem;
103     err_ = ld_->hipFree(hipmem);
104     _hiperror(err_);
105     if (err_ != hipSuccess) return BRISBANE_ERR;
106     return BRISBANE_OK;
107 }
108
109 int DeviceHIP::MemH2D(Mem* mem, size_t off, size_t size, void* host) {
110     void* hipmem = mem->arch(this);
111     err_ = ld_->hipMemcpyHtoD((char*) hipmem + off, host, size);
112     _hiperror(err_);
113     if (err_ != hipSuccess) return BRISBANE_ERR;
114     return BRISBANE_OK;
115 }
116
117 int DeviceHIP::MemD2H(Mem* mem, size_t off, size_t size, void* host) {
118     void* hipmem = mem->arch(this);
119     err_ = ld_->hipMemcpyDtoH(host, (char*) hipmem + off, size);
120     _hiperror(err_);
121     if (err_ != hipSuccess) return BRISBANE_ERR;
122     return BRISBANE_OK;
123 }
124
125 int DeviceHIP::KernelGet(void** kernel, const char* name) {
126     hipFunction_t* hipkernel = (hipFunction_t*) kernel;
127     err_ = ld_->hipModuleGetFunction(hipkernel, module_, name);
128     _hiperror(err_);
N... DeviceHIP.cpp      cpp 58% N:124 F:1
```

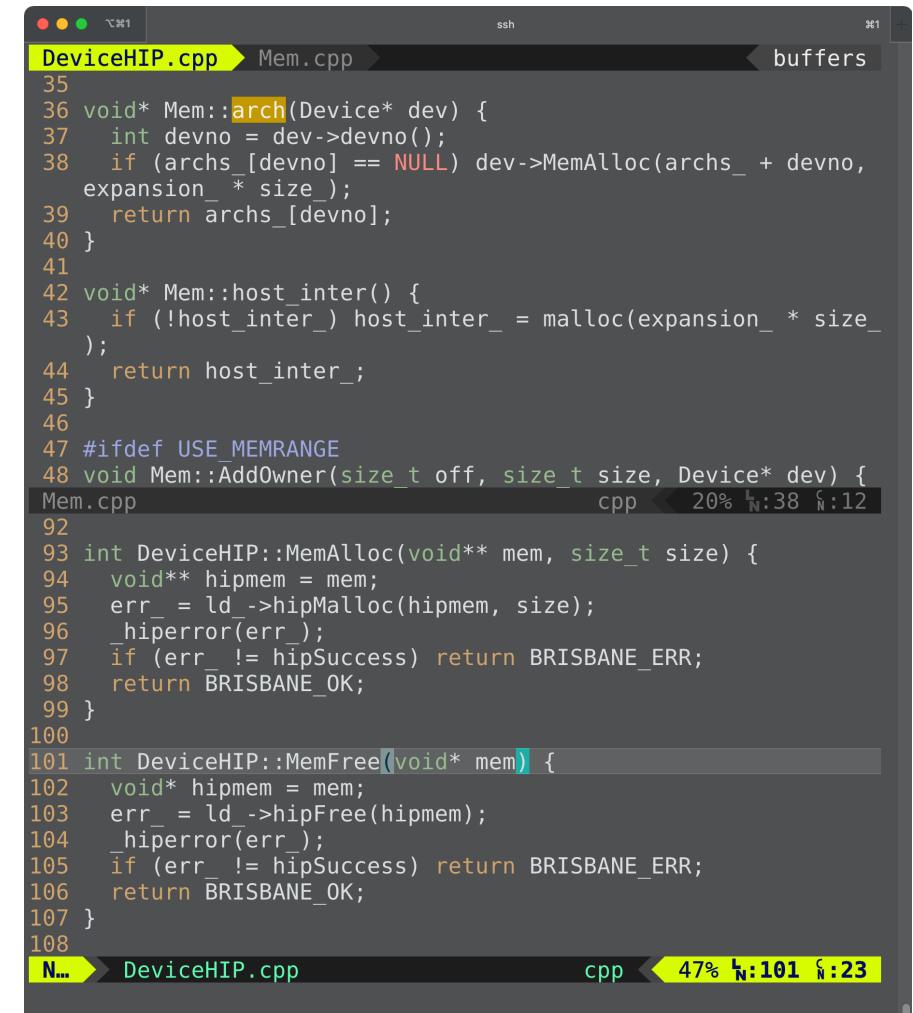
```
DeviceHIP.cpp Mem.cpp buffers
35
36 void* Mem::arch(Device* dev) {
37     int devno = dev->devno();
38     if (archs_[devno] == NULL) dev->MemAlloc(archs_ + devno,
39         expansion_* size_);
40     return archs_[devno];
41 }
42 void* Mem::host_inter() {
43     if (!host_inter_) host_inter_ = malloc(expansion_* size_);
44     return host_inter_;
45 }
46
47 #ifdef USE_MEMRANGE
48 void Mem::AddOwner(size_t off, size_t size, Device* dev) {
49     Mem.cpp      cpp 20% N:38 F:12
50
51     int DeviceHIP::MemAlloc(void** mem, size_t size) {
52         void** hipmem = mem;
53         err_ = ld_->hipMalloc(hipmem, size);
54         _hiperror(err_);
55         if (err_ != hipSuccess) return BRISBANE_ERR;
56         return BRISBANE_OK;
57     }
58
59     int DeviceHIP::MemFree(void* mem) {
60         void* hipmem = mem;
61         err_ = ld_->hipFree(hipmem);
62         _hiperror(err_);
63         if (err_ != hipSuccess) return BRISBANE_ERR;
64         return BRISBANE_OK;
65     }
66
67     N... DeviceHIP.cpp      cpp 47% N:101 F:23

```

# IRIS Memory Arch Table

Memory

archs_[]	API	Device
0	<code>posix_memalign(size_)</code>	<b>AMD CPU [0]</b>
1	<code>hipMalloc(size_)</code>	<b>AMD GPU [1]</b>
2	<code>hipMalloc(size_)</code>	<b>AMD GPU [2]</b>

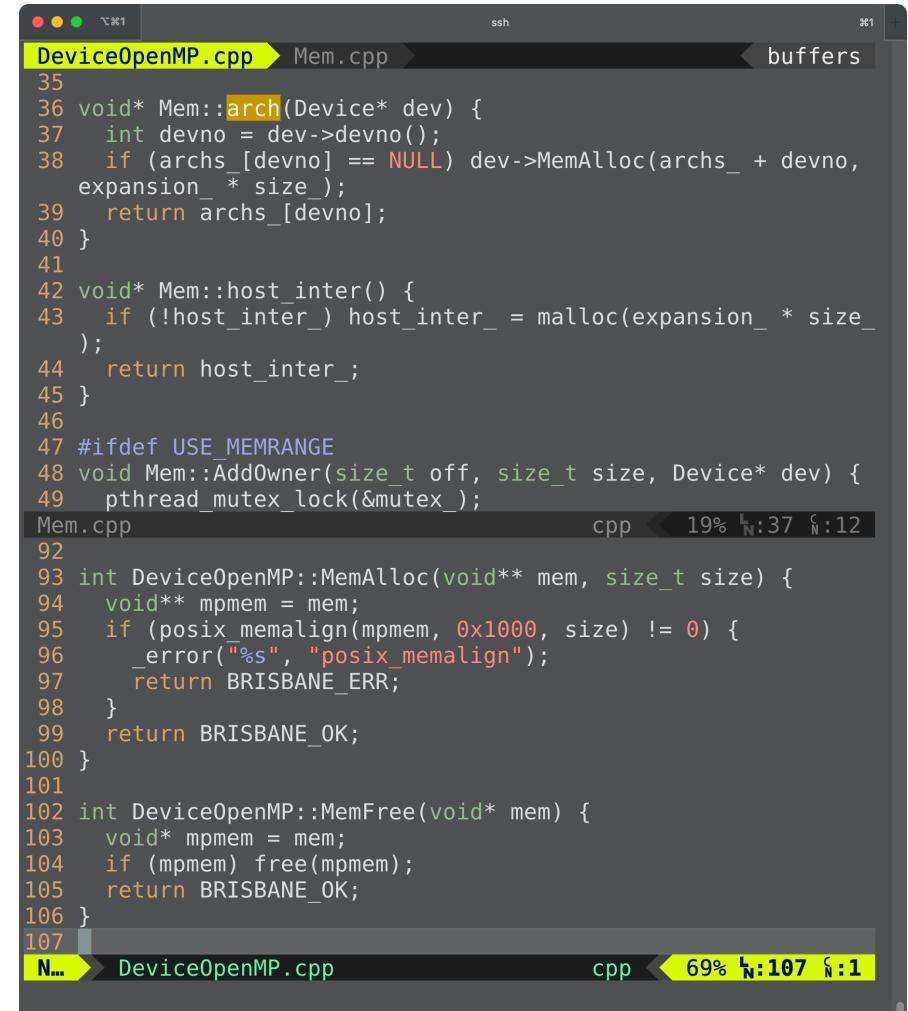


```
DeviceHIP.cpp Mem.cpp buffers
35
36 void* Mem::arch(Device* dev) {
37     int devno = dev->devno();
38     if (archs_[devno] == NULL) dev->MemAlloc(archs_ + devno,
39         expansion_* size_);
40     return archs_[devno];
41 }
42 void* Mem::host_inter() {
43     if (!host_inter_) host_inter_ = malloc(expansion_* size_);
44     return host_inter_;
45 }
46
47 #ifdef USE_MEMRANGE
48 void Mem::AddOwner(size_t off, size_t size, Device* dev) {
49     Mem.cpp                                         cpp 20%  N:38  l:12
50
51     int DeviceHIP::MemAlloc(void** mem, size_t size) {
52         void** hipmem = mem;
53         err_ = ld_->hipMalloc(hipmem, size);
54         _hiperror(err_);
55         if (err_ != hipSuccess) return BRISBANE_ERR;
56         return BRISBANE_OK;
57     }
58
59     int DeviceHIP::MemFree(void* mem) {
60         void* hipmem = mem;
61         err_ = ld_->hipFree(hipmem);
62         _hiperror(err_);
63         if (err_ != hipSuccess) return BRISBANE_ERR;
64         return BRISBANE_OK;
65     }
66
67     N... DeviceHIP.cpp                                         cpp 47%  N:101  l:23
68 }
```

# IRIS Memory Arch Table

## Memory

archs_[]	API	Device
0	posix_memalign(size_)	AMD CPU [0]
1	hipMalloc(size_)	AMD GPU [1]
2	hipMalloc(size_)	AMD GPU [2]



```
DeviceOpenMP.cpp Mem.cpp buffers
35
36 void* Mem::arch(Device* dev) {
37     int devno = dev->devno();
38     if (archs_[devno] == NULL) dev->MemAlloc(archs_ + devno,
39         expansion_* size_);
40     return archs_[devno];
41 }
42 void* Mem::host_inter() {
43     if (!host_inter_) host_inter_ = malloc(expansion_* size_);
44     return host_inter_;
45 }
46
47 #ifdef USE_MEMRANGE
48 void Mem::AddOwner(size_t off, size_t size, Device* dev) {
49     pthread_mutex_lock(&mutex_);
50     Mem.cpp                                         cpp  19%  N:37  N:12
51
52     int DeviceOpenMP::MemAlloc(void** mem, size_t size) {
53         void** mpmem = mem;
54         if (posix_memalign(mppmem, 0x1000, size) != 0) {
55             _error("%s", "posix_memalign");
56             return BRISBANE_ERR;
57         }
58         return BRISBANE_OK;
59     }
60
61     int DeviceOpenMP::MemFree(void* mem) {
62         void* mppmem = mem;
63         if (mppmem) free(mppmem);
64         return BRISBANE_OK;
65     }
66
67     N... DeviceOpenMP.cpp                                         cpp  69%  N:107  N:1
68 }
```

# src/runtime/DeviceHIP.cpp

```
DeviceHIP.cpp buffers ssh
100
101 int DeviceHIP::MemFree(void* mem) {
102     void* hipmem = mem;
103     err_ = ld_->hipFree(hipmem);
104     _hiperror(err_);
105     if (err_ != hipSuccess) return BRISBANE_ERR;
106     return BRISBANE_OK;
107 }
108
109 int DeviceHIP::MemH2D(Mem* mem, size_t off, size_t size, void* host) {
110     void* hipmem = mem->arch(this);
111     err_ = ld_->hipMemcpyHtoD((char*) hipmem + off, host, size);
112     _hiperror(err_);
113     if (err_ != hipSuccess) return BRISBANE_ERR;
114     return BRISBANE_OK;
115 }
116
117 int DeviceHIP::MemD2H(Mem* mem, size_t off, size_t size, void* host) {
118     void* hipmem = mem->arch(this);
119     err_ = ld_->hipMemcpyDtoH(host, (char*) hipmem + off, size);
120     _hiperror(err_);
121     if (err_ != hipSuccess) return BRISBANE_ERR;
122     return BRISBANE_OK;
123 }
124
125 int DeviceHIP::KernelGet(void** kernel, const char* name) {
126     hipFunction_t* hipkernel = (hipFunction_t*) kernel;
127     err_ = ld_->hipModuleGetFunction(hipkernel, module_, name);
128     _hiperror(err_);
N... DeviceHIP.cpp      cpp 58% N:124 F:1
```

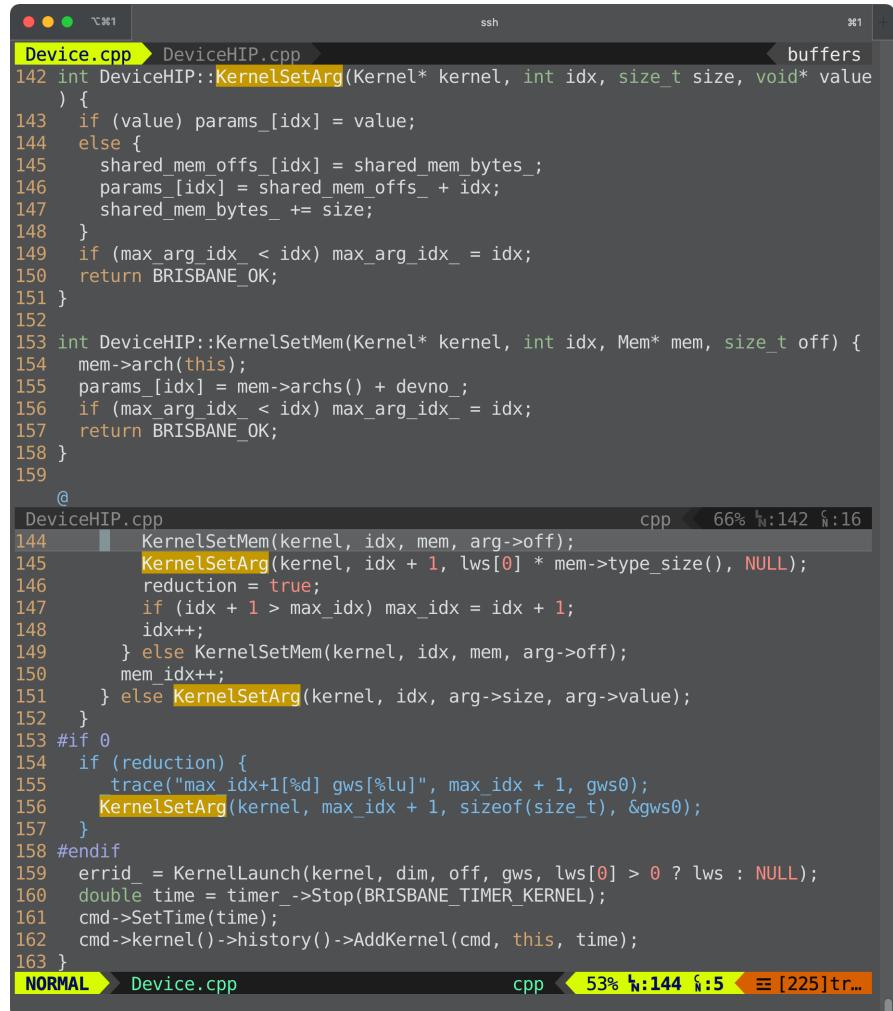
```
PolicyDevice.cpp Device.cpp buffers ssh
168     _trace("dev[%d] malloc[%p]", devno_, arch);
169 }
170
171 void Device::ExecuteH2D(Command* cmd) {
172     Mem* mem = cmd->mem();
173     size_t off = cmd->off(0);
174     size_t size = cmd->size();
175     bool exclusive = cmd->exclusive();
176     void* host = cmd->host();
177     if (exclusive) mem->SetOwner(off, size, this);
178     else mem->AddOwner(off, size, this);
179     timer_->Start(BRISBANE_TIMER_H2D);
180     errid_ = MemH2D(mem, off, size, host);
181     if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
182     double time = timer_->Stop(BRISBANE_TIMER_H2D);
183     cmd->SetTime(time);
184     Command* cmd_kernel = cmd->task()->cmd_kernel();
185     if (cmd_kernel) cmd_kernel->kernel()->history()->AddH2D(cmd, this, time);
186     else Platform::GetPlatform()->null_kernel()->history()->AddH2D(cmd, this, time);
187 }
188
189 void Device::ExecuteH2DNP(Command* cmd) {
190     Mem* mem = cmd->mem();
191     size_t off = cmd->off(0);
192     size_t size = cmd->size();
193     // if (mem->IsOwner(off, size, this)) return;
194     return ExecuteH2D(cmd);
195 }
196
197 void Device::ExecuteD2H(Command* cmd) {
198     Mem* mem = cmd->mem();
199     size_t off = cmd->off(0);
N... Device.cpp      cpp 69% N:188 F:1
```

# src/runtime/Device.cpp

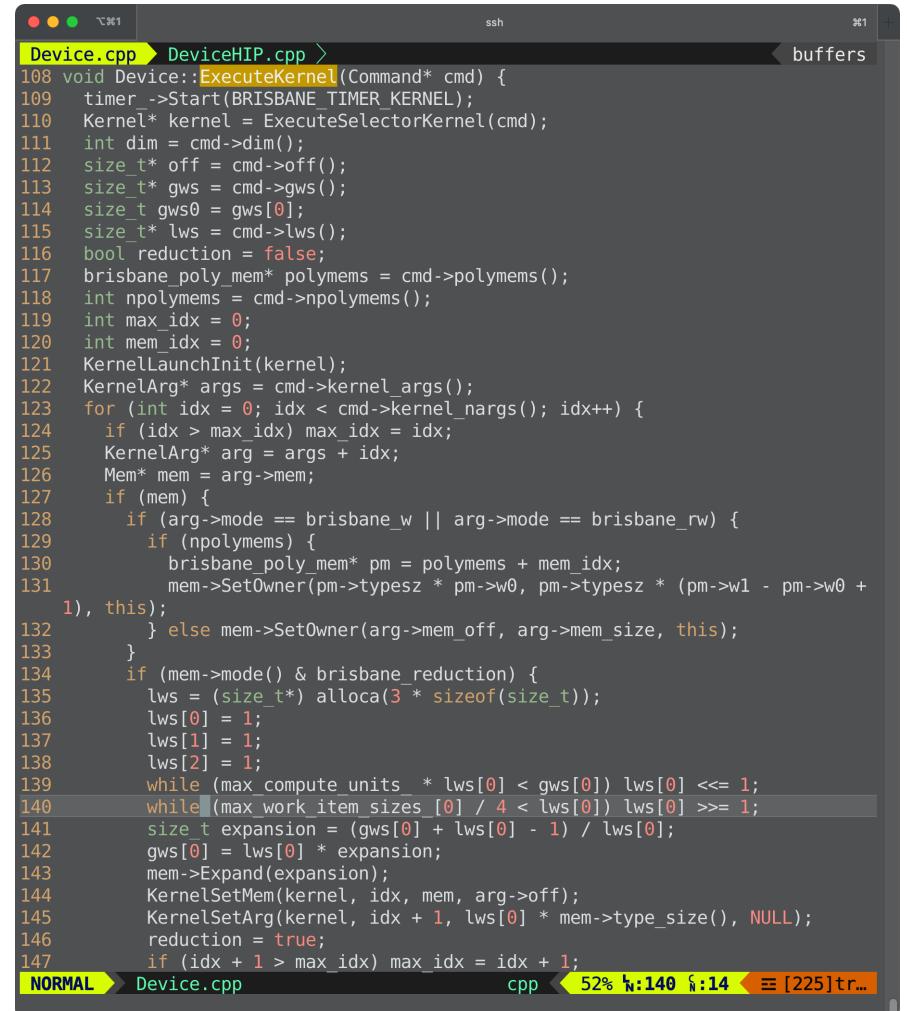
```
Device.cpp
35 delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%x]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61 #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //    if (++q_ >= nqueues_) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", tas
k->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_->Start(BRISBANE_TIMER_INIT);
NORMAL Device.cpp      cpp 27% h:73 l:14 ≡ [225]tr...
```

```
Device.cpp  DeviceHIP.cpp 
108 void Device::ExecuteKernel(Command* cmd) {
109     timer_->Start(BRISBANE_TIMER_KERNEL);
110     Kernel* kernel = ExecuteSelectorKernel(cmd);
111     int dim = cmd->dim();
112     size_t* off = cmd->off();
113     size_t* gws = cmd->gws();
114     size_t gws0 = gws[0];
115     size_t* lws = cmd->lws();
116     bool reduction = false;
117     brisbane_poly_mem* polymems = cmd->polymems();
118     int npolymems = cmd->npolymems();
119     int max_idx = 0;
120     int mem_idx = 0;
121     KernelLaunchInit(kernel);
122     KernelArg* args = cmd->kernel_args();
123     for (int idx = 0; idx < cmd->kernel_nargs(); idx++) {
124         if (idx > max_idx) max_idx = idx;
125         KernelArg* arg = args + idx;
126         Mem* mem = arg->mem;
127         if (mem) {
128             if (arg->mode == brisbane_w || arg->mode == brisbane_rw) {
129                 if (!polymems) {
130                     brisbane_poly_mem* pm = polymems + mem_idx;
131                     mem->SetOwner(pm->typesz * pm->w0, pm->typesz * (pm->wl - pm->w0 +
1), this);
132                 } else mem->SetOwner(arg->mem_off, arg->mem_size, this);
133             }
134             if (mem->mode() & brisbane_reduction) {
135                 lws = (size_t*) alloca(3 * sizeof(size_t));
136                 lws[0] = 1;
137                 lws[1] = 1;
138                 lws[2] = 1;
139                 while (max_compute_units_ * lws[0] < gws[0]) lws[0] <<= 1;
140                 while [(max_work_item_sizes[0] / 4 < lws[0]) lws[0] >>= 1;
141                 size_t expansion = (gws[0] + lws[0] - 1) / lws[0];
142                 gws[0] = lws[0] * expansion;
143                 mem->Expand(expansion);
144                 KernelSetMem(kernel, idx, mem, arg->off);
145                 KernelSetArg(kernel, idx + 1, lws[0] * mem->type_size(), NULL);
146                 reduction = true;
147                 if (idx + 1 > max_idx) max_idx = idx + 1;
NORMAL Device.cpp      cpp 52% h:140 l:14 ≡ [225]tr...
```

# src/runtime/DeviceHIP.cpp

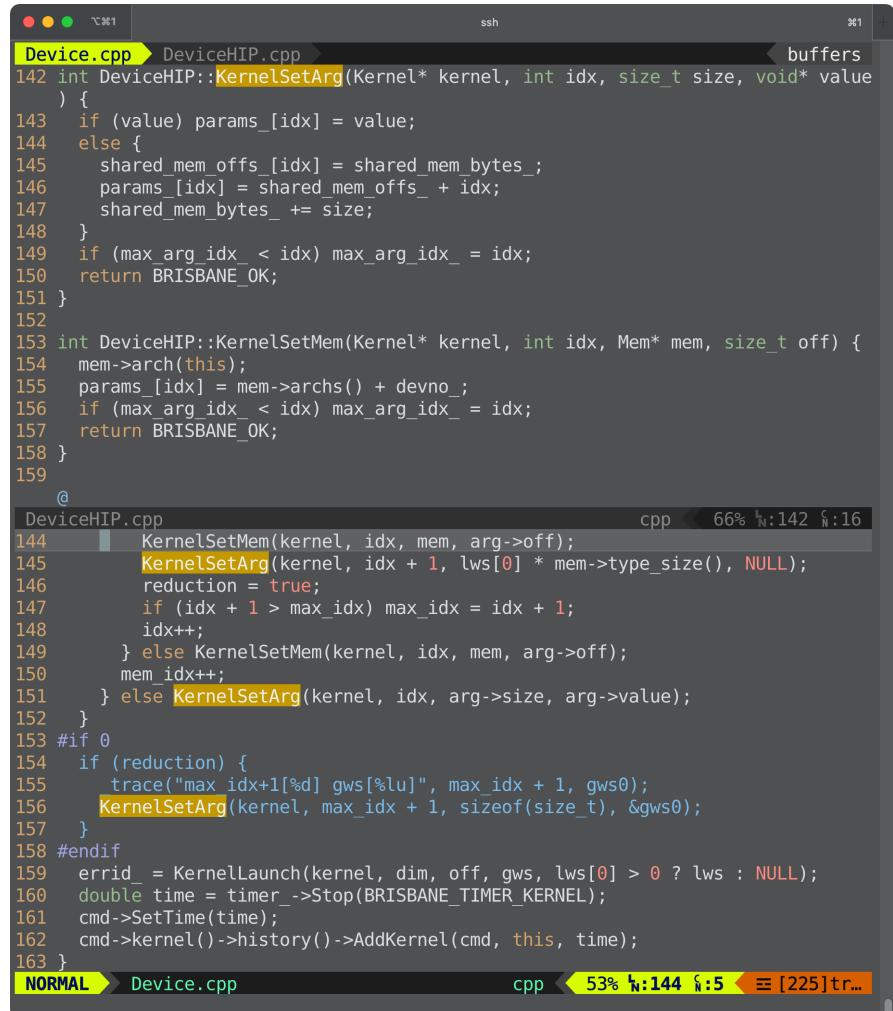


```
Device.cpp DeviceHIP.cpp buffers
142 int DeviceHIP::KernelSetArg(Kernel* kernel, int idx, size_t size, void* value)
143 {
144     if (value) params_[idx] = value;
145     else {
146         shared_mem_offs_[idx] = shared_mem_bytes_;
147         params_[idx] = shared_mem_offs_ + idx;
148         shared_mem_bytes_ += size;
149     }
150     if (max_arg_idx_ < idx) max_arg_idx_ = idx;
151     return BRISBANE_OK;
152 }
153 int DeviceHIP::KernelSetMem(Kernel* kernel, int idx, Mem* mem, size_t off) {
154     mem->arch(this);
155     params_[idx] = mem->archs() + devno_;
156     if (max_arg_idx_ < idx) max_arg_idx_ = idx;
157     return BRISBANE_OK;
158 }
159 @
| DeviceHIP.cpp                                         cpp  66% h:142 f:16
144     KernelSetMem(kernel, idx, mem, arg->off);
145     KernelSetArg(kernel, idx + 1, lws[0] * mem->type_size(), NULL);
146     reduction = true;
147     if (idx + 1 > max_idx) max_idx = idx + 1;
148     idx++;
149 } else KernelSetMem(kernel, idx, mem, arg->off);
150     mem_idx++;
151 } else KernelSetArg(kernel, idx, arg->size, arg->value);
152 }
153 #if 0
154     if (reduction) {
155         trace("max_idx+1[%d] gws[%lu]", max_idx + 1, gws0);
156         KernelSetArg(kernel, max_idx + 1, sizeof(size_t), &gws0);
157     }
158 #endif
159     errid_ = KernelLaunch(kernel, dim, off, gws, lws[0] > 0 ? lws : NULL);
160     double time = timer_->Stop(BRISBANE_TIMER_KERNEL);
161     cmd->SetTime(time);
162     cmd->kernel()->history()->AddKernel(cmd, this, time);
163 }
NORMAL Device.cpp                                         cpp  53% h:144 f:5 [225]tr...
```

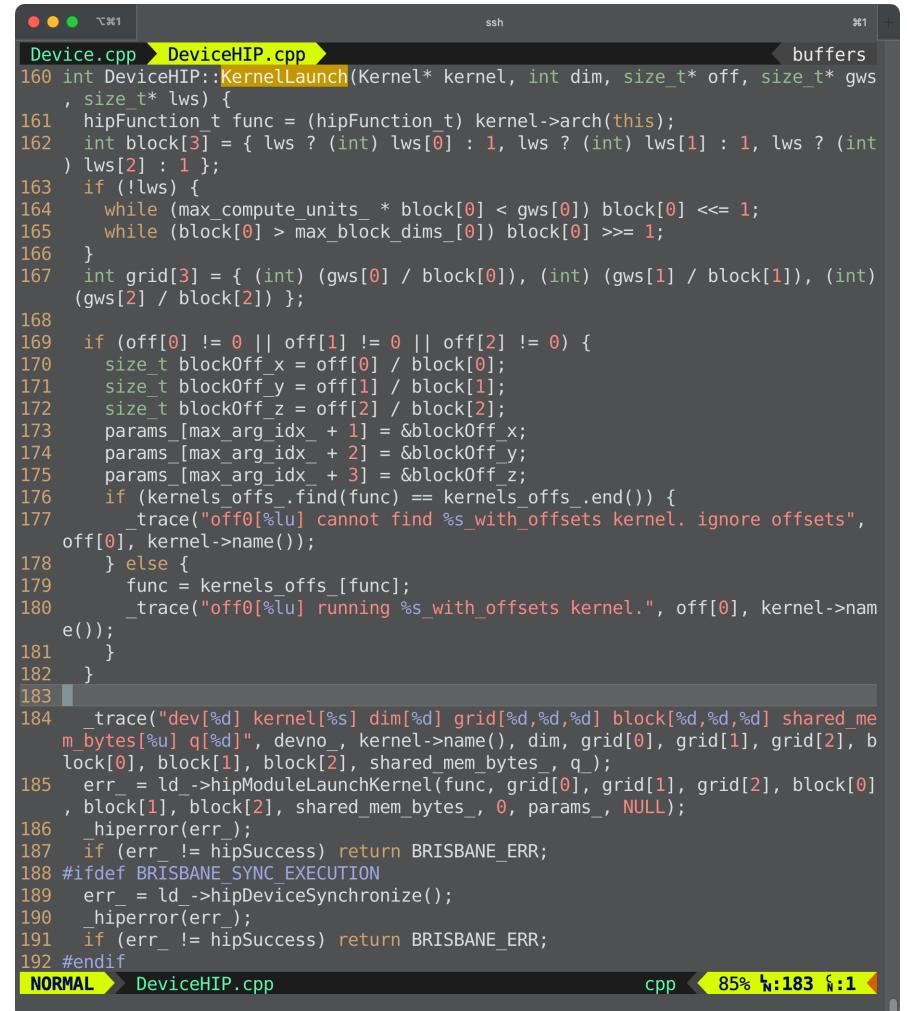


```
Device.cpp DeviceHIP.cpp buffers
108 void Device::ExecuteKernel(Command* cmd) {
109     timer_->Start(BRISBANE_TIMER_KERNEL);
110     Kernel* kernel = ExecuteSelectorKernel(cmd);
111     int dim = cmd->dim();
112     size_t* off = cmd->off();
113     size_t* gws = cmd->gws();
114     size_t gws0 = gws[0];
115     size_t* lws = cmd->lws();
116     bool reduction = false;
117     brisbane_poly_mem* polymems = cmd->polymems();
118     int npolymems = cmd->npolymems();
119     int max_idx = 0;
120     int mem_idx = 0;
121     KernelLaunchInit(kernel);
122     KernelArg* args = cmd->kernel_args();
123     for (int idx = 0; idx < cmd->kernel_nargs(); idx++) {
124         if (idx > max_idx) max_idx = idx;
125         KernelArg* arg = args + idx;
126         Mem* mem = arg->mem;
127         if (mem) {
128             if (arg->mode == brisbane_w || arg->mode == brisbane_rw) {
129                 if (!polymems) {
130                     brisbane_poly_mem* pm = polymems + mem_idx;
131                     mem->SetOwner(pm->typesz * pm->w0, pm->typesz * (pm->w1 - pm->w0 + 1), this);
132                 } else mem->SetOwner(arg->mem_off, arg->mem_size, this);
133             }
134             if (mem->mode() & brisbane_reduction) {
135                 lws = (size_t*) alloca(3 * sizeof(size_t));
136                 lws[0] = 1;
137                 lws[1] = 1;
138                 lws[2] = 1;
139                 while (max_compute_units_ * lws[0] < gws[0]) lws[0] <<= 1;
140                 while [(max_work_item_sizes_[0] / 4 < lws[0]) lws[0] >>= 1];
141                 size_t expansion = (gws[0] + lws[0] - 1) / lws[0];
142                 gws[0] = lws[0] * expansion;
143                 mem->Expand(expansion);
144                 KernelSetMem(kernel, idx, mem, arg->off);
145                 KernelSetArg(kernel, idx + 1, lws[0] * mem->type_size(), NULL);
146                 reduction = true;
147                 if (idx + 1 > max_idx) max_idx = idx + 1;
148             }
149         }
150     }
NORMAL Device.cpp                                         cpp  52% h:140 f:14 [225]tr...
```

# src/runtime/DeviceHIP.cpp



```
Device.cpp DeviceHIP.cpp buffers
142 int DeviceHIP::KernelSetArg(Kernel* kernel, int idx, size_t size, void* value
143 ) {
144     if (value) params_[idx] = value;
145     else {
146         shared_mem_offs_[idx] = shared_mem_bytes_;
147         params_[idx] = shared_mem_offs_ + idx;
148     }
149     shared_mem_bytes_ += size;
150     if (max_arg_idx_ < idx) max_arg_idx_ = idx;
151     return BRISBANE_OK;
152 }
153 int DeviceHIP::KernelSetMem(Kernel* kernel, int idx, Mem* mem, size_t off) {
154     mem->arch(this);
155     params_[idx] = mem->archs() + devno_;
156     if (max_arg_idx_ < idx) max_arg_idx_ = idx;
157     return BRISBANE_OK;
158 }
159
@ DeviceHIP.cpp
144     KernelSetMem(kernel, idx, mem, arg->off);
145     KernelSetArg(kernel, idx + 1, lws[0] * mem->type_size(), NULL);
146     reduction = true;
147     if (idx + 1 > max_idx) max_idx = idx + 1;
148     idx++;
149 } else KernelSetMem(kernel, idx, mem, arg->off);
150     mem_idx++;
151 } else KernelSetArg(kernel, idx, arg->size, arg->value);
152 }
153 #if 0
154     if (reduction) {
155         _trace("max_idx+1[%d] gws[%lu]", max_idx + 1, gws0);
156         KernelSetArg(kernel, max_idx + 1, sizeof(size_t), &gws0);
157     }
158 #endif
159     errid_ = KernelLaunch(kernel, dim, off, gws, lws[0] > 0 ? lws : NULL);
160     double time = timer_->Stop(BRISBANE_TIMER_KERNEL);
161     cmd->SetTime(time);
162     cmd->kernel()->history()->AddKernel(cmd, this, time);
163 }
```

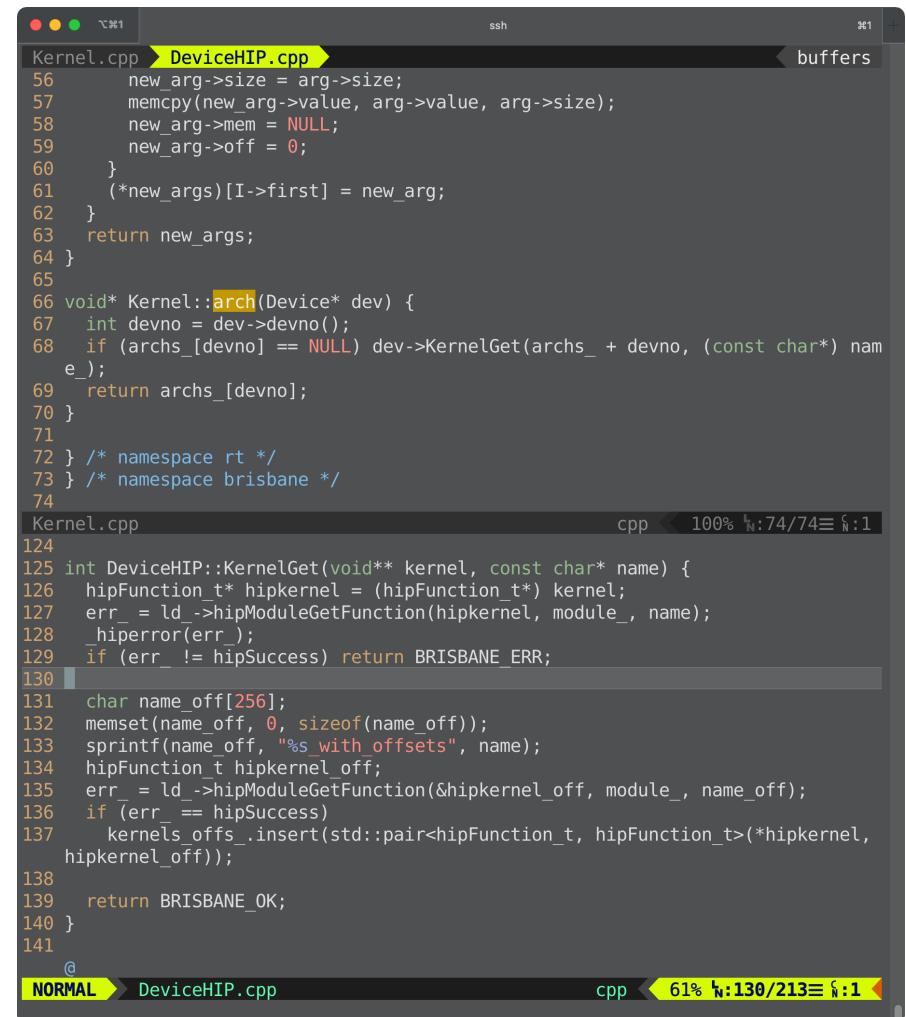


```
Device.cpp DeviceHIP.cpp buffers
160 int DeviceHIP::KernelLaunch(Kernel* kernel, int dim, size_t* off, size_t* gws,
161     size_t* lws) {
162     hipFunction_t func = (hipFunction_t) kernel->arch(this);
163     int block[3] = { lws ? (int) lws[0] : 1, lws ? (int) lws[1] : 1, lws ? (int)
164     lws[2] : 1 };
165     if (!lws) {
166         while (max_compute_units_ * block[0] < gws[0]) block[0] <<= 1;
167         while (block[0] > max_block_dims_[0]) block[0] >>= 1;
168     }
169     int grid[3] = { (int) (gws[0] / block[0]), (int) (gws[1] / block[1]), (int)
170     (gws[2] / block[2]) };
171     if (off[0] != 0 || off[1] != 0 || off[2] != 0) {
172         size_t blockOff_x = off[0] / block[0];
173         size_t blockOff_y = off[1] / block[1];
174         size_t blockOff_z = off[2] / block[2];
175         params_[max_arg_idx_ + 1] = &blockOff_x;
176         params_[max_arg_idx_ + 2] = &blockOff_y;
177         params_[max_arg_idx_ + 3] = &blockOff_z;
178     } else {
179         func = kernels_offs_[func];
180         _trace("off[0>%lu] running %s_with_offsets kernel.", off[0], kernel->n
181         am());
182     }
183
184     _trace("dev[%d] kernel[%s] dim[%d] grid[%d,%d,%d] block[%d,%d,%d] shared_
185     m_bytes[%u] q[%d]", devno_, kernel->name(), dim, grid[0], grid[1], grid[2],
186     block[0], block[1], block[2], shared_mem_bytes_, q);
187     err_ = ld_->hipModuleLaunchKernel(func, grid[0], grid[1], grid[2], block[0],
188     block[1], block[2], shared_mem_bytes_, 0, params_, NULL);
189     hipererror(err_);
190     if (err_ != hipSuccess) return BRISBANE_ERR;
191 #ifndef BRISBANE_SYNC_EXECUTION
192     err_ = ld_->hipDeviceSynchronize();
193     if (err_ != hipSuccess) return BRISBANE_ERR;
194 #endif
195 }
```

# IRIS Kernel Arch Table

Kernel

archs_[]	API	Device
0	void*	AMD CPU [0]
1	hipFunction_t	AMD GPU [1]
2	hipFunction_t	AMD GPU [2]



```
Kernel.cpp DeviceHIP.cpp buffers
56     new_arg->size = arg->size;
57     memcpy(new_arg->value, arg->value, arg->size);
58     new_arg->mem = NULL;
59     new_arg->off = 0;
60   }
61   (*new_args)[I->first] = new_arg;
62 }
63 return new_args;
64 }

65 void* Kernel::arch(Device* dev) {
66   int devno = dev->devno();
67   if (archs_[devno] == NULL) dev->KernelGet(archs_ + devno, (const char*) name_);
68   return archs_[devno];
69 }
70 }

71 /* namespace rt */
72 /* namespace brisbane */
73

74
Kernel.cpp
cpp 100% 1:74/74≡ 1:1
124
125 int DeviceHIP::KernelGet(void** kernel, const char* name) {
126   hipFunction_t* hipkernel = (hipFunction_t*) kernel;
127   err_ = ld_->hipModuleGetFunction(hipkernel, module_, name);
128   _hiperror(err_);
129   if (err_ != hipSuccess) return BRISBANE_ERR;
130
131   char name_off[256];
132   memset(name_off, 0, sizeof(name_off));
133   sprintf(name_off, "%s_with_offsets", name);
134   hipFunction_t hipkernel_off;
135   err_ = ld_->hipModuleGetFunction(&hipkernel_off, module_, name_off);
136   if (err_ == hipSuccess)
137     kernels_offs_.insert(std::pair<hipFunction_t, hipFunction_t>(*hipkernel,
138                         hipkernel_off));
139
140   return BRISBANE_OK;
141 }

@ NORMAL DeviceHIP.cpp
cpp 61% 1:130/213≡ 1:1
```

# src/runtime/Device.cpp

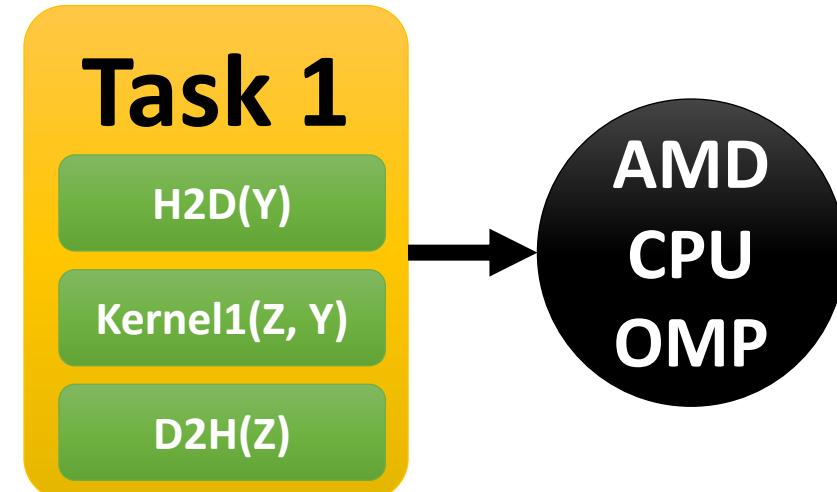
```
Device.cpp
35     delete timer_;
36 }
37
38 void Device::Execute(Task* task) {
39     busy_ = true;
40     if (hook_task_pre_) hook_task_pre_(task);
41     TaskPre(task);
42     for (int i = 0; i < task->ncmds(); i++) {
43         Command* cmd = task->cmd(i);
44         if (hook_command_pre_) hook_command_pre_(cmd);
45         switch (cmd->type()) {
46             case BRISBANE_CMD_INIT: ExecuteInit(cmd); break;
47             case BRISBANE_CMD_KERNEL: ExecuteKernel(cmd); break;
48             case BRISBANE_CMD_MALLOC: ExecuteMalloc(cmd); break;
49             case BRISBANE_CMD_H2D: ExecuteH2D(cmd); break;
50             case BRISBANE_CMD_H2DNP: ExecuteH2DNP(cmd); break;
51             case BRISBANE_CMD_D2H: ExecuteD2H(cmd); break;
52             case BRISBANE_CMD_MAP: ExecuteMap(cmd); break;
53             case BRISBANE_CMD_RELEASE_MEM: ExecuteReleaseMem(cmd); break;
54             case BRISBANE_CMD_HOST: ExecuteHost(cmd); break;
55             case BRISBANE_CMD_CUSTOM: ExecuteCustom(cmd); break;
56             default: _error("cmd type[0x%x]", cmd->type());
57         }
58         if (hook_command_post_) hook_command_post_(cmd);
59 #ifndef BRISBANE_SYNC_EXECUTION
60         if (cmd->last()) AddCallback(task);
61 #endif
62     }
63     TaskPost(task);
64     if (hook_task_post_) hook_task_post_(task);
65 //    if (++q_ >= nqueues_) q_ = 0;
66     if (!task->system()) _trace("task[%lu] complete dev[%d][%s] time[%lf]", tas
k->uid(), devno(), name(), task->time());
67 #ifdef BRISBANE_SYNC_EXECUTION
68     task->Complete();
69 #endif
70     busy_ = false;
71 }
72
73 void Device::ExecuteInit(Command* cmd) {
74     timer_>Start(BRISBANE_TIMER_INIT);
NORMAL Device.cpp 27% 1:73 1:14 [225]tr...
```

```
Task.cpp
123
124 void Task::Complete() {
125     pthread_mutex_lock(&mutex_complete_);
126     status_ = BRISBANE_COMPLETE;
127     pthread_cond_broadcast(&complete_cond_);
128     pthread_mutex_unlock(&mutex_complete_);
129     if (parent_) parent_->CompleteSub();
130     else {
131         if (dev_) dev_->worker()->TaskComplete(this);
132         else if (scheduler_) scheduler_->Invoke();
133     }
134     for (int i = 0; i < ndepends_; i++)
135         if (depends_[i]->user() && !depends_[i]->perm()) depends_[i]->Release();
136     if (user_ && !perm_) Release();
137 }
138
139 void Task::CompleteSub() {
140     pthread_mutex_lock(&mutex_subtasks_);
141     if (++subtasks_complete_ == subtasks_.size()) Complete();
142     pthread_mutex_unlock(&mutex_subtasks_);
143 }
144
145 void Task::Wait() {
146     pthread_mutex_lock(&mutex_complete_);
147     if (status_ != BRISBANE_COMPLETE)
148         pthread_cond_wait(&complete_cond_, &mutex_complete_);
149     pthread_mutex_unlock(&mutex_complete_);
150 }
151
152 void Task::AddSubtask(Task* subtask) {
153     subtask->set_parent(this);
154     subtask->set_brs_policy(brs_policy_);
155     subtasks_.push_back(subtask);
156 }
157
158 bool Task::HasSubtasks() {
159     return !subtasks_.empty();
160 }
161
162 void Task::AddDepend(Task* task) {
163     if (depends_ == NULL) depends_ = new Task*[depends_max];
NORMAL Task.cpp 70% 1:138/196 1:1 [170]tr...
```

# iris\_task\_submit()

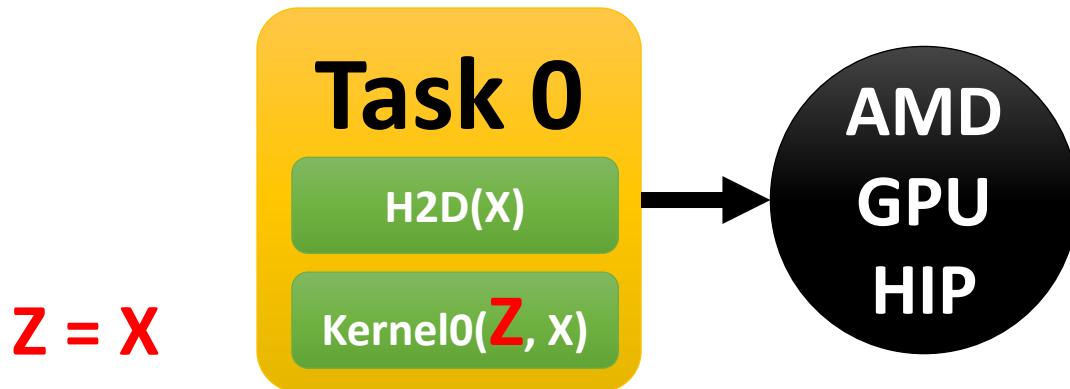
```
2tasks.c      buffers
34  iris_task task0;
35  iris_task_create(&task0);
36  iris_task_h2d_full(task0, mem_X, X);
37  void* task0_params[2] = { mem_Z, mem_X };
38  int task0_params_info[2] = { iris_w, iris_r };
39  iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40  iris_task_submit(task0, iris_gpu, NULL, 1);
41
42  iris_task task1;
43  iris_task_create(&task1);
44  iris_task_h2d_full(task1, mem_Y, Y);
45  void* task1_params[2] = { mem_Z, mem_Y };
46  int task1_params_info[2] = { iris_rw, iris_r };
47  iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48  iris_task_d2h_full(task1, mem_Z, Z);
49  iris_task_submit(task1, iris_cpu, NULL, 1);
50
51  printf("Z [ ");
52  for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53  printf("]\n");
54
55  iris_task_release(task0);
56  iris_task_release(task1);
57
58  iris_mem_release(mem_X);
59  iris_mem_release(mem_Y);
60  iris_mem_release(mem_Z);
61
62  iris_finalize();
63  return 0;
64 }
```

NORMAL ➤ 2tasks.c      c utf-8[unix] 98% N:64/65 ≡ 1

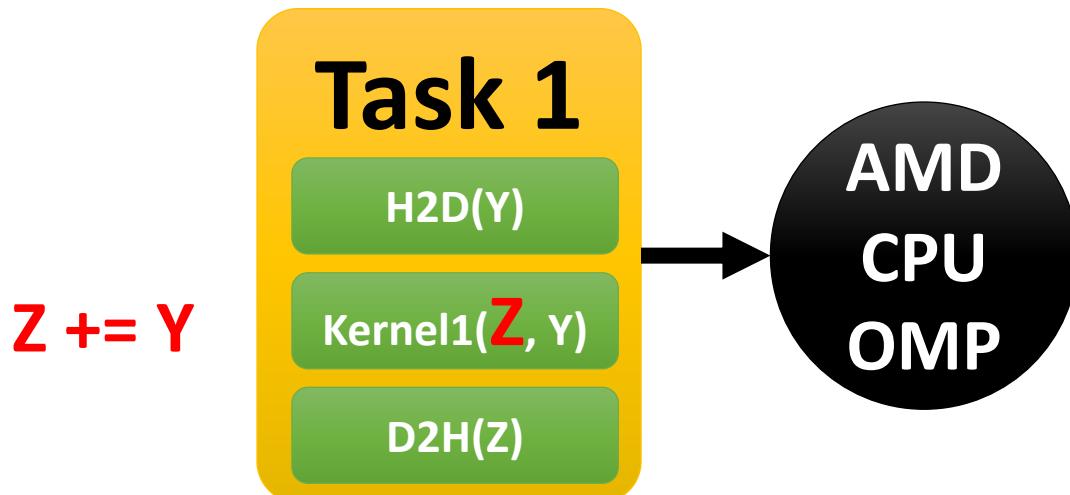


# src/runtime/Consistency.cpp

```
Worker.cpp Consistency.cpp buffers
40
41 void Worker::Execute(Task* task) {
42     if (!task->Executable()) return;
43     task->set_dev(dev_);
44     if (task->marker()) {
45         dev_->Synchronize();
46         task->Complete();
47         return;
48     }
49     busy_ = true;
50     if (scheduler_) scheduler_->StartTask(task, this);
51     if (consistency_) consistency_->Resolve(task);
52     dev_->Execute(task);
53     if (!task->cmd_last()) {
54         if (scheduler_) scheduler_->CompleteTask(task, this);
55         //task->Complete();
56     }
57     busy_ = false;
58 }
59
NORMAL Worker.cpp CPP 54% h:40/74 l:1
13 Consistency::Consistency(Scheduler* scheduler) {
14     scheduler_ = scheduler;
15 }
16
17 Consistency::~Consistency() {
18 }
19
20 void Consistency::Resolve(Task* task) {
21     if (task->system()) return;
22     for (int i = 0; i < task->ncmds(); i++) {
23         Command* cmd = task->cmd(i);
24         switch (cmd->type()) {
25             case BRISBANE_CMD_KERNEL:      ResolveKernel(task, cmd);      break;
26             case BRISBANE_CMD_D2H:          ResolveD2H(task, cmd);        break;
27         }
28     }
29 }
30
31 void Consistency::ResolveKernel(Task* task, Command* cmd) {
32 // if (task->parent()) return;
Consistency.cpp CPP 15% h:21/134 l:16
```



Z: RAW dependency b/w Task 0 (GPU) and Task 1 (CPU)



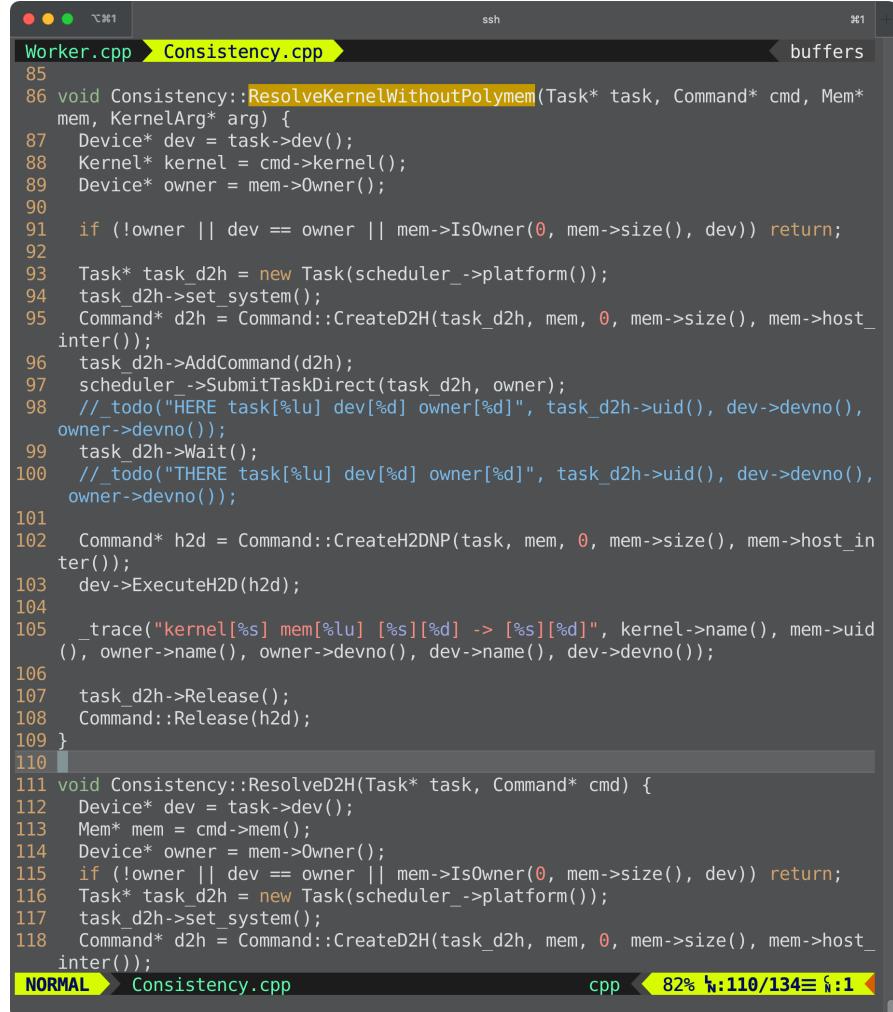
# src/runtime/Consistency.cpp

```
Worker.cpp Consistency.cpp buffers
40
41 void Worker::Execute(Task* task) {
42     if (!task->Executable()) return;
43     task->set_dev(dev_);
44     if (task->marker()) {
45         dev_->Synchronize();
46         task->Complete();
47         return;
48     }
49     busy_ = true;
50     if (scheduler_) scheduler_->StartTask(task, this);
51     if (consistency_) consistency_->Resolve(task);
52     dev_->Execute(task);
53     if (!task->cmd_last()) {
54         if (scheduler_) scheduler_->CompleteTask(task, this);
55         //task->Complete();
56     }
57     busy_ = false;
58 }
59
NORMAL > Worker.cpp      cpp 54% h:40/74 ≡ f:1
13 Consistency::Consistency(Scheduler* scheduler) {
14     scheduler_ = scheduler;
15 }
16
17 Consistency::~Consistency() {
18 }
19
20 void Consistency::Resolve(Task* task) {
21     if (task->system()) return;
22     for (int i = 0; i < task->ncmds(); i++) {
23         Command* cmd = task->cmd(i);
24         switch (cmd->type()) {
25             case BRISBANE_CMD_KERNEL:    ResolveKernel(task, cmd);    break;
26             case BRISBANE_CMD_D2H:       ResolveD2H(task, cmd);       break;
27         }
28     }
29 }
30
31 void Consistency::ResolveKernel(Task* task, Command* cmd) {
32 // if (task->parent()) return;
Consistency.cpp      cpp 15% h:21/134 ≡ f:1
```

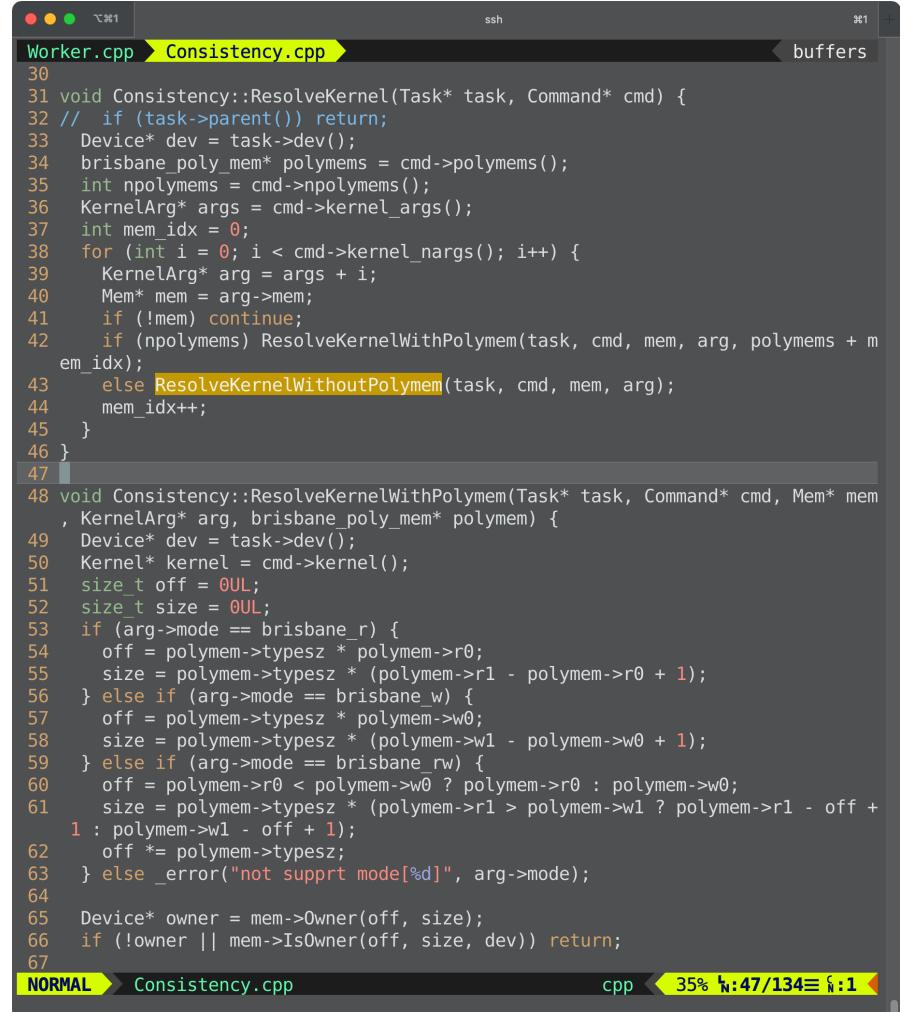
  

```
Worker.cpp Consistency.cpp buffers
30
31 void Consistency::ResolveKernel(Task* task, Command* cmd) {
32 // if (task->parent()) return;
33     Device* dev = task->dev();
34     brisbane_poly_mem* polymems = cmd->polymems();
35     int npolymems = cmd->npolymems();
36     KernelArg* args = cmd->kernel_args();
37     int mem_idx = 0;
38     for (int i = 0; i < cmd->kernel_nargs(); i++) {
39         KernelArg* arg = args + i;
40         Mem* mem = arg->mem;
41         if (!mem) continue;
42         if (npolymems) ResolveKernelWithPolymem(task, cmd, mem, arg, polymems + mem_idx);
43         else ResolveKernelWithoutPolymem(task, cmd, mem, arg);
44         mem_idx++;
45     }
46 }
47
48 void Consistency::ResolveKernelWithPolymem(Task* task, Command* cmd, Mem* mem,
        , KernelArg* arg, brisbane_poly_mem* polymem) {
49     Device* dev = task->dev();
50     Kernel* kernel = cmd->kernel();
51     size_t off = 0UL;
52     size_t size = 0UL;
53     if (arg->mode == brisbane_r) {
54         off = polymem->typesz * polymem->r0;
55         size = polymem->typesz * (polymem->r1 - polymem->r0 + 1);
56     } else if (arg->mode == brisbane_w) {
57         off = polymem->typesz * polymem->w0;
58         size = polymem->typesz * (polymem->w1 - polymem->w0 + 1);
59     } else if (arg->mode == brisbane_rw) {
60         off = polymem->r0 < polymem->w0 ? polymem->r0 : polymem->w0;
61         size = polymem->typesz * (polymem->r1 > polymem->w1 ? polymem->r1 - off +
62         1 : polymem->w1 - off + 1);
62         off *= polymem->typesz;
63     } else _error("not suprt mode[%d]", arg->mode);
64
65     Device* owner = mem->Owner(off, size);
66     if (!owner || mem->IsOwner(off, size, dev)) return;
67
NORMAL > Consistency.cpp      cpp 35% h:47/134 ≡ f:1
```

# src/runtime/Consistency.cpp



```
Worker.cpp > Consistency.cpp buffers
85
86 void Consistency::ResolveKernelWithoutPolymem(Task* task, Command* cmd, Mem*
mem, KernelArg* arg) {
87     Device* dev = task->dev();
88     Kernel* kernel = cmd->kernel();
89     Device* owner = mem->owner();
90
91     if (!owner || dev == owner || mem->IsOwner(0, mem->size(), dev)) return;
92
93     Task* task_d2h = new Task(scheduler_->platform());
94     task_d2h->set_system();
95     Command* d2h = Command::CreateD2H(task_d2h, mem, 0, mem->size(), mem->host_
inter());
96     task_d2h->AddCommand(d2h);
97     scheduler_->SubmitTaskDirect(task_d2h, owner);
98     //_todo("HERE task[%lu] dev[%d] owner[%d]", task_d2h->uid(), dev->devno(),
99            owner->devno());
99     task_d2h->Wait();
100    //_todo("THERE task[%lu] dev[%d] owner[%d]", task_d2h->uid(), dev->devno(),
101          owner->devno());
102
103    Command* h2d = Command::CreateH2DNP(task, mem, 0, mem->size(), mem->host_in
ter());
104    dev->ExecuteH2D(h2d);
105
106    _trace("kernel[%s] mem[%lu] [%s][%d] -> [%s][%d]", kernel->name(), mem->uid
(), owner->name(), owner->devno(), dev->name(), dev->devno());
107
108    task_d2h->Release();
109    Command::Release(h2d);
110
111 void Consistency::ResolveD2H(Task* task, Command* cmd) {
112     Device* dev = task->dev();
113     Mem* mem = cmd->mem();
114     Device* owner = mem->owner();
115     if (!owner || dev == owner || mem->IsOwner(0, mem->size(), dev)) return;
116
117     Task* task_d2h = new Task(scheduler_->platform());
118     task_d2h->set_system();
119     Command* d2h = Command::CreateD2H(task_d2h, mem, 0, mem->size(), mem->host_
inter());
NORMAL > Consistency.cpp      cpp < 82% 110/134 ≡ 1:1
```



```
Worker.cpp > Consistency.cpp buffers
30
31 void Consistency::ResolveKernel(Task* task, Command* cmd) {
32     // if (task->parent()) return;
33     Device* dev = task->dev();
34     brisbane_poly_mem* polymems = cmd->polymems();
35     int npolymems = cmd->npolymems();
36     KernelArg* args = cmd->kernel_args();
37     int mem_idx = 0;
38     for (int i = 0; i < cmd->kernel_nargs(); i++) {
39         KernelArg* arg = args + i;
40         Mem* mem = arg->mem;
41         if (!mem) continue;
42         if (npolymems) ResolveKernelWithPolymem(task, cmd, mem, arg, polymems + m
em_idx);
43         else ResolveKernelWithoutPolymem(task, cmd, mem, arg);
44         mem_idx++;
45     }
46 }
47
48 void Consistency::ResolveKernelWithPolymem(Task* task, Command* cmd, Mem* mem
, KernelArg* arg, brisbane_poly_mem* polymem) {
49     Device* dev = task->dev();
50     Kernel* kernel = cmd->kernel();
51     size_t off = 0UL;
52     size_t size = 0UL;
53     if (arg->mode == brisbane_r) {
54         off = polymem->typesz * polymem->r0;
55         size = polymem->typesz * (polymem->r1 - polymem->r0 + 1);
56     } else if (arg->mode == brisbane_w) {
57         off = polymem->typesz * polymem->w0;
58         size = polymem->typesz * (polymem->w1 - polymem->w0 + 1);
59     } else if (arg->mode == brisbane_rw) {
60         off = polymem->r0 < polymem->w0 ? polymem->r0 : polymem->w0;
61         size = polymem->typesz * (polymem->r1 > polymem->w1 ? polymem->r1 - off +
62                                     1 : polymem->w1 - off + 1);
62         off *= polymem->typesz;
63     } else _error("not suprt mode[%d]", arg->mode);
64
65     Device* owner = mem->owner(off, size);
66     if (!owner || mem->IsOwner(off, size, dev)) return;
67
NORMAL > Consistency.cpp      cpp < 35% 47/134 ≡ 1:1
```

# iris\_task\_finalize()

A screenshot of a terminal window titled 'ssh' showing the '2tasks.c' file. The code implements two tasks, task0 and task1, using the IRIS API. Task0 performs a matrix multiplication from X to Z, while Task1 performs a transpose operation from Y to Z. Both tasks are submitted to the CPU. After completion, the results are printed to the console.

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

NORMAL ➤ 2tasks.c c utf-8[unix] 98% ↵:64/65 ≡ ↵:1

A screenshot of a terminal window titled 'ssh' showing the 'Platform.cpp' file. This file contains the implementation of the Platform::GetPlatform() function and the Finalize() method. The Finalize() method prints execution times for various timer events and releases the mutex used for synchronization.

```
Platform.cpp buffers
1023     t_d2h += history->t_d2h();
1024 }
1025     _info("total kernel[%lf] h2d[%lf] d2h[%lf]", t_ker, t_h2d, t_d2h);
1026     return BRISBANE_OK;
1027 }
1028
1029 Platform* Platform::singleton_ = NULL;
1030 std::once_flag Platform::flag_singleton_;
1031 std::once_flag Platform::flag_finalize_;
1032
1033 Platform* Platform::GetPlatform() {
1034     std::call_once(flag_singleton_, []() { singleton_ = new Platform(); });
1035     return singleton_;
1036 }
1037
1038 int Platform::Finalize() {
1039     pthread_mutex_lock(&mutex_);
1040     if (finalize_) {
1041         pthread_mutex_unlock(&mutex_);
1042         return BRISBANE_ERR;
1043     }
1044     int ret_id = Synchronize();
1045     ShowKernelHistory();
1046     time_app_ = timer()->Stop(BRISBANE_TIMER_APP);
1047     time_init_ = timer()->Total(BRISBANE_TIMER_PLATFORM);
1048     _info("total execution time:[%lf] sec. initialize:[%lf] sec. t-i:[%lf] sec",
1049           time_app_, time_init_, time_app_ - time_init_);
1050     _info("t10[%lf] t11[%lf] t12[%lf] t13[%lf]", timer()->Total(10), timer()->
1051           Total(11), timer()->Total(12), timer()->Total(13));
1052     _info("t14[%lf] t15[%lf] t16[%lf] t17[%lf]", timer()->Total(14), timer()->
1053           Total(15), timer()->Total(16), timer()->Total(17));
1054     _info("t18[%lf] t19[%lf] t20[%lf] t21[%lf]", timer()->Total(18), timer()->
1055           Total(19), timer()->Total(20), timer()->Total(21));
1056     finalize_ = true;
1057     pthread_mutex_unlock(&mutex_);
1058 }
1059
1060 /* namespace rt */
1061 /* namespace brisbane */
```

NORMAL ➤ Platform.cpp cpp 99% ↵:1056/1059 ≡ [380]tr...

# src/runtime/Platform.cpp

```
2tasks.c buffers
34     iris_task task0;
35     iris_task_create(&task0);
36     iris_task_h2d_full(task0, mem_X, X);
37     void* task0_params[2] = { mem_Z, mem_X };
38     int task0_params_info[2] = { iris_w, iris_r };
39     iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_params_info);
40     iris_task_submit(task0, iris_gpu, NULL, 1);
41
42     iris_task task1;
43     iris_task_create(&task1);
44     iris_task_h2d_full(task1, mem_Y, Y);
45     void* task1_params[2] = { mem_Z, mem_Y };
46     int task1_params_info[2] = { iris_rw, iris_r };
47     iris_task_kernel(task1, "kernel1", 1, NULL, &SIZE, NULL, 2, task1_params, task1_params_info);
48     iris_task_d2h_full(task1, mem_Z, Z);
49     iris_task_submit(task1, iris_cpu, NULL, 1);
50
51     printf("Z [");
52     for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Z[i]);
53     printf("]\n");
54
55     iris_task_release(task0);
56     iris_task_release(task1);
57
58     iris_mem_release(mem_X);
59     iris_mem_release(mem_Y);
60     iris_mem_release(mem_Z);
61
62     iris_finalize();
63     return 0;
64 }
```

NORMAL ➤ 2tasks.c c utf-8[unix] 98% ↵:64/65 ≡ ↵:1

```
Platform.cpp buffers
1023     t_d2h += history->t_d2h();
1024 }
1025     _info("total kernel[%lf] h2d[%lf] d2h[%lf]", t_ker, t_h2d, t_d2h);
1026     return BRISBANE_OK;
1027 }
1028
1029 Platform* Platform::singleton_ = NULL;
1030 std::once_flag Platform::flag_singleton_;
1031 std::once_flag Platform::flag_finalize_;
1032
1033 Platform* Platform::GetPlatform() {
1034     std::call_once(flag_singleton_, []() { singleton_ = new Platform(); });
1035     return singleton_;
1036 }
1037
1038 int Platform::Finalize() {
1039     pthread_mutex_lock(&mutex_);
1040     if (finalize_) {
1041         pthread_mutex_unlock(&mutex_);
1042         return BRISBANE_ERR;
1043     }
1044     int ret_id = Synchronize();
1045     ShowKernelHistory();
1046     time_app_ = timer()->Stop(BRISBANE_TIMER_APP);
1047     time_init_ = timer()->Total(BRISBANE_TIMER_PLATFORM);
1048     _info("total execution time:[%lf] sec. initialize:[%lf] sec. t-i:[%lf] sec",
1049           time_app_, time_init_, time_app_ - time_init_);
1050     _info("t10[%lf] t11[%lf] t12[%lf] t13[%lf]", timer()->Total(10), timer()->
1051       Total(11), timer()->Total(12), timer()->Total(13));
1052     _info("t14[%lf] t15[%lf] t16[%lf] t17[%lf]", timer()->Total(14), timer()->
1053       Total(15), timer()->Total(16), timer()->Total(17));
1054     _info("t18[%lf] t19[%lf] t20[%lf] t21[%lf]", timer()->Total(18), timer()->
1055       Total(19), timer()->Total(20), timer()->Total(21));
1056     finalize_ = true;
1057     pthread_mutex_unlock(&mutex_);
1058 }
1059
1060 /* namespace rt */
1061 /* namespace brisbane */
```

NORMAL ➤ Platform.cpp cpp 99% ↵:1056/1059 ≡ [380]tr...

# src/runtime/Platform.cpp

```
Platform.cpp > buffers
180
181     InitScheduler();
182     InitWorkers();
183     InitDevices(sync);
184
185     _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d] scheduler[%d] hub[%d] po
186         lyhedral[%d] profile[%d]",
187         nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != NULL, scheduler_ ?
188             scheduler_->hub_available() : 0,
189             polyhedral_available_, enable_profiler_);
190
191     timer_->Stop(BRISBANE_TIMER_PLATFORM);
192
193     init_ = true;
194
195     pthread_mutex_unlock(&mutex_);
196 }
197
198 int Platform::Synchronize() {
199     int* devices = new int[ndevs_];
200     for (int i = 0; i < ndevs_; i++) devices[i] = i;
201     int ret = DeviceSynchronize(ndevs_, devices);
202     delete devices;
203     return ret;
204 }
205
206 int Platform::EnvironmentInit() {
207     EnvironmentSet("ARCS", "openmp:cuda:hip:levelzero:hexagon:opencl", false);
208     EnvironmentSet("TMPDIR", "/tmp/iris", false);
209
210     EnvironmentSet("KERNEL_SRC_CUDA", "kernel.cu", false);
211     EnvironmentSet("KERNEL_BIN_CUDA", "kernel.ptx", false);
212     EnvironmentSet("KERNEL_SRC_HEXAGON", "kernel.hexagon.cpp", false);
213     EnvironmentSet("KERNEL_BIN_HEXAGON", "kernel.hexagon.so", false);
214     EnvironmentSet("KERNEL_SRC_HIP", "kernel.hip.cpp", false);
215     EnvironmentSet("KERNEL_BIN_HIP", "kernel.hip", false);
216     EnvironmentSet("KERNEL_SRC_OPENMP", "kernel.openmp.h", false);
217
NORMAL ➤ Platform.cpp      cpp 19% ↵:205/1059 ≡ ⌂:1 ≡ [380]tr...
search hit BOTTOM, continuing at TOP
```

```
Platform.cpp > buffers
1023     t_d2h += history->t_d2h();
1024 }
1025     _info("total kernel[%lf] h2d[%lf] d2h[%lf]", t_ker, t_h2d, t_d2h);
1026     return BRISBANE_OK;
1027 }
1028
1029 Platform* Platform::singleton_ = NULL;
1030 std::once_flag Platform::flag_singleton_;
1031 std::once_flag Platform::flag_finalize_;
1032
1033 Platform* Platform::GetPlatform() {
1034     std::call_once(flag_singleton_, []() { singleton_ = new Platform(); });
1035     return singleton_;
1036 }
1037
1038 int Platform::Finalize() {
1039     pthread_mutex_lock(&mutex_);
1040     if (finalize_) {
1041         pthread_mutex_unlock(&mutex_);
1042         return BRISBANE_ERR;
1043     }
1044     int ret_id = Synchronize();
1045     ShowKernelHistory();
1046     time_app_ = timer()->Stop(BRISBANE_TIMER_APP);
1047     time_init_ = timer()->Total(BRISBANE_TIMER_PLATFORM);
1048     _info("total execution time:[%lf] sec. initialize:[%lf] sec. t-i:[%lf] sec
1049           ", time_app_, time_init_, time_app_ - time_init_);
1050     _info("t10[%lf] t11[%lf] t12[%lf] t13[%lf]", timer()->Total(10), timer()->
1051           Total(11), timer()->Total(12), timer()->Total(13));
1052     _info("t14[%lf] t15[%lf] t16[%lf] t17[%lf]", timer()->Total(14), timer()->
1053           Total(15), timer()->Total(16), timer()->Total(17));
1054     _info("t18[%lf] t19[%lf] t20[%lf] t21[%lf]", timer()->Total(18), timer()->
1055           Total(19), timer()->Total(20), timer()->Total(21));
1056     finalize_ = true;
1057 } /* namespace rt */
1058 } /* namespace brisbane */
1059
NORMAL ➤ Platform.cpp      cpp 99% ↵:1056/1059 ≡ ⌂:1 ≡ [380]tr...
```

# src/runtime/Platform.cpp

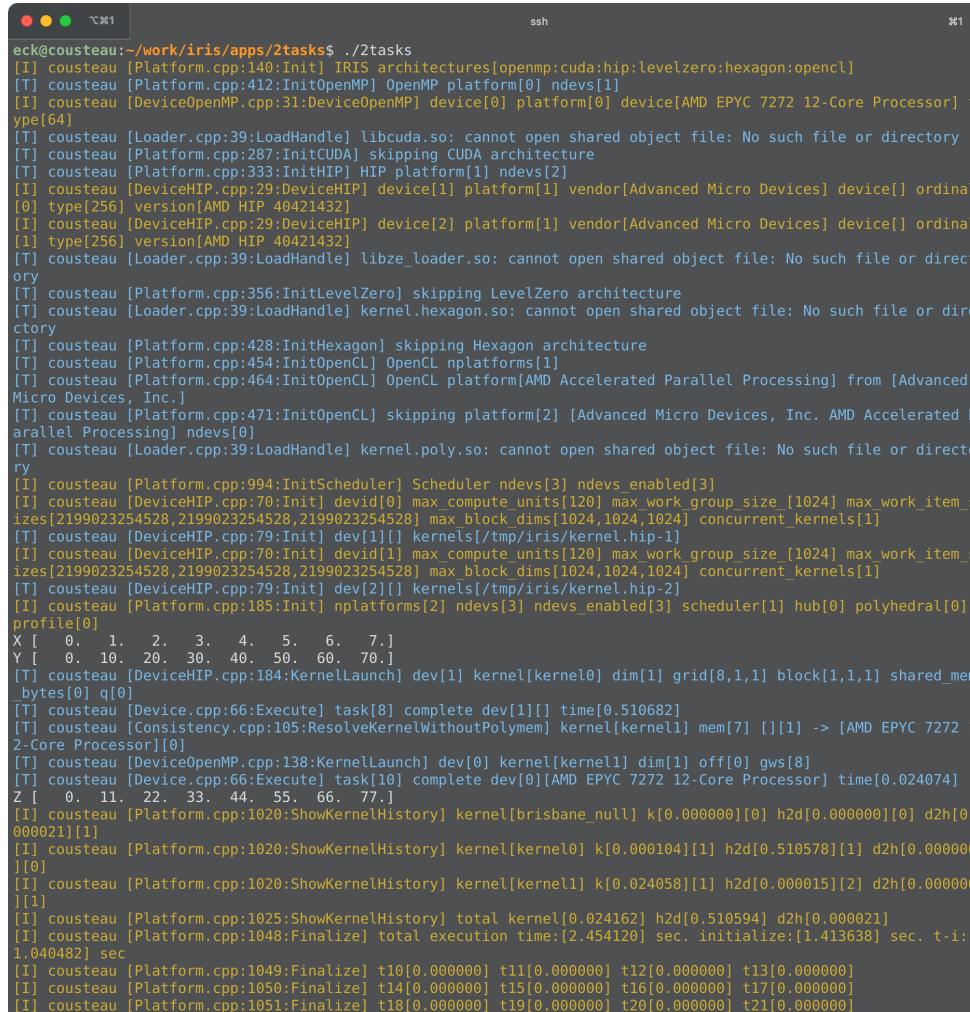
```
Platform.cpp > buffers
180
181     InitScheduler();
182     InitWorkers();
183     InitDevices(sync);
184
185     _info("nplatforms[%d] ndevs[%d] ndevs_enabled[%d] scheduler[%d] hub[%d] po
186         lyhedral[%d] profile[%d]", nplatforms_, ndevs_, ndevs_enabled_, scheduler_ != NULL, scheduler_ ?
187             scheduler_->hub_available() : 0,
188             polyhedral_available_, enable_profiler_);
189
190     timer_->Stop(BRISBANE_TIMER_PLATFORM);
191
192     init_ = true;
193
194     pthread_mutex_unlock(&mutex_);
195
196 } // Platform::Init()
197
198 int Platform::Synchronize() {
199     int* devices = new int[ndevs_];
200     for (int i = 0; i < ndevs_; i++) devices[i] = i;
201     int ret = DeviceSynchronize(ndevs_, devices);
202     delete devices;
203     return ret;
204 }
205
206 int Platform::EnvironmentInit() {
207     EnvironmentSet("ARCS", "openmp:cuda:hip:levelzero:hexagon:opencl", false);
208     EnvironmentSet("TMPDIR", "/tmp/iris", false);
209
210     EnvironmentSet("KERNEL_SRC_CUDA", "kernel.cu", false);
211     EnvironmentSet("KERNEL_BIN_CUDA", "kernel.ptx", false);
212     EnvironmentSet("KERNEL_SRC_HEXAGON", "kernel.hexagon.cpp", false);
213     EnvironmentSet("KERNEL_BIN_HEXAGON", "kernel.hexagon.so", false);
214     EnvironmentSet("KERNEL_SRC_HIP", "kernel.hip.cpp", false);
215     EnvironmentSet("KERNEL_BIN_HIP", "kernel.hip", false);
216     EnvironmentSet("KERNEL_SRC_OPENMP", "kernel.openmp.h", false);
217
218     _error("Platform::EnvironmentInit() failed");
219 }
```

NORMAL ➤ Platform.cpp ⇧ 19% ⌂ 205/1059 ≡ ⌂ 1 ⌂ [380] tr...  
search hit BOTTOM, continuing at TOP

```
Platform.cpp > buffers
568     return BRISBANE_OK;
569 }
570
571 int Platform::DeviceGetDefault(int* device) {
572     *device = dev_default_;
573     return BRISBANE_OK;
574 }
575
576 int Platform::DeviceSynchronize(int ndevs, int* devices) {
577     Task* task = new Task(this, BRISBANE_MARKER);
578     if (scheduler_) {
579         for (int i = 0; i < ndevs; i++) {
580             if (devices[i] >= ndevs_) {
581                 _error("devices[%d]", devices[i]);
582                 continue;
583             }
584             Task* subtask = new Task(this, BRISBANE_MARKER);
585             subtask->set_devno(devices[i]);
586             task->AddSubtask(subtask);
587         }
588         scheduler_->Enqueue(task);
589     } else workers_[0]->Enqueue(task);
590     task->Wait();
591     return task->Ok();
592 }
593
594 int Platform::PolicyRegister(const char* lib, const char* name, void* params)
595 {
596     return scheduler_->policies()->Register(lib, name, params);
597 }
598
599 int Platform::RegisterCommand(int tag, int device, command_handler handler)
600 {
601     for (int i = 0; i < ndevs_; i++)
602         if (devs_[i]->type() == device) devs_[i]->RegisterCommand(tag, handler);
603     return BRISBANE_OK;
604 }
605
606 int Platform::RegisterHooksTask(hook_task pre, hook_task post) {
607     hook_task_pre_ = pre;
608     hook_task_post_ = post;
609 }
```

NORMAL ➤ Platform.cpp ⇧ 55% ⌂ 593/1059 ≡ ⌂ 1 ⌂ [380] tr...  
search hit BOTTOM, continuing at TOP

# cmake -DCMAKE\_TYPE=DEBUG



```
eck@cousteau:~/work/iris/apps/2tasks$ ./2tasks
[I] cousteau [Platform.cpp:140:Init] IRIS architectures[openmp:cuda:hip:levelzero:hexagon:opencl]
[T] cousteau [Platform.cpp:412:InitOpenMP] OpenMP platform[0] ndevs[1]
[II] cousteau [DeviceOpenMP.cpp:31:DeviceOpenMP] device[0] platform[0] device[AMD EPYC 7272 12-Core Processor] type[64]
[T] cousteau [Loader.cpp:39:LoadHandle] libcuda.so: cannot open shared object file: No such file or directory
[T] cousteau [Platform.cpp:287:InitCUDA] skipping CUDA architecture
[T] cousteau [Platform.cpp:333:InitHIP] HIP platform[1] ndevs[2]
[II] cousteau [DeviceHIP.cpp:29:DeviceHIP] device[1] platform[1] vendor[Advanced Micro Devices] device[] ordinal[0] type[256] version[AMD HIP 40421432]
[II] cousteau [DeviceHIP.cpp:29:DeviceHIP] device[2] platform[1] vendor[Advanced Micro Devices] device[] ordinal[1] type[256] version[AMD HIP 40421432]
[T] cousteau [Loader.cpp:39:LoadHandle] libze_loader.so: cannot open shared object file: No such file or directory
[T] cousteau [Platform.cpp:356:InitLevelZero] skipping LevelZero architecture
[T] cousteau [Loader.cpp:39:LoadHandle] kernel.hexagon.so: cannot open shared object file: No such file or directory
[T] cousteau [Platform.cpp:428:InitHexagon] skipping Hexagon architecture
[T] cousteau [Platform.cpp:454:InitOpenCL] OpenCL nplatforms[1]
[T] cousteau [Platform.cpp:464:InitOpenCL] OpenCL platform[AMD Accelerated Parallel Processing] from [Advanced Micro Devices, Inc.]
[T] cousteau [Platform.cpp:471:InitOpenCL] skipping platform[2] [Advanced Micro Devices, Inc. AMD Accelerated Parallel Processing] ndevs[0]
[T] cousteau [Loader.cpp:39:LoadHandle] kernel.poly.so: cannot open shared object file: No such file or directory
[II] cousteau [Platform.cpp:994:InitScheduler] Scheduler ndevs[3] ndevs_enabled[3]
[II] cousteau [DeviceHIP.cpp:70:Init] devid[0] max_compute_units[128] max_work_group_size[1024] max_work_item_sizes[2199023254528,2199023254528,2199023254528] max_block_dims[1024,1024,1024] concurrent_kernels[1]
[T] cousteau [DeviceHIP.cpp:79:Init] dev[1][] kernels[/tmp/iris/kernel.hip-1]
[II] cousteau [DeviceHIP.cpp:70:Init] devid[1] max_compute_units[128] max_work_group_size[1024] max_work_item_sizes[2199023254528,2199023254528,2199023254528] max_block_dims[1024,1024,1024] concurrent_kernels[1]
[T] cousteau [DeviceHIP.cpp:79:Init] dev[2][] kernels[/tmp/iris/kernel.hip-2]
[II] cousteau [Platform.cpp:185:Init] nplatforms[2] ndevs[3] ndevs_enabled[3] scheduler[1] hub[0] polyhedral[0] profile[0]
X [ 0. 1. 2. 3. 4. 5. 6. 7.]
Y [ 0. 10. 20. 30. 40. 50. 60. 70.]
[T] cousteau [DeviceHIP.cpp:184:KernelLaunch] dev[1] kernel[kernel0] dim[1] grid[8,1,1] block[1,1,1] shared_mem_bytes[0] q[0]
[T] cousteau [Device.cpp:66:Execute] task[8] complete dev[1][] time[0.510682]
[T] cousteau [Consistency.cpp:105:ResolveKernelWithoutPolymem] kernel[kernel1] mem[7] [][] -> [AMD EPYC 7272 12-Core Processor][0]
[T] cousteau [DeviceOpenMP.cpp:138:KernelLaunch] dev[0] kernel[kernel1] dim[1] off[0] gws[8]
[T] cousteau [Device.cpp:66:Execute] task[10] complete dev[0][AMD EPYC 7272 12-Core Processor] time[0.024074]
Z [ 0. 11. 22. 33. 44. 55. 66. 77.]
[II] cousteau [Platform.cpp:1020>ShowKernelHistory] kernel[brisbane_null] k[0.000000][0] h2d[0.000000][0] d2h[0.000021][1]
[II] cousteau [Platform.cpp:1020>ShowKernelHistory] kernel[kernel0] k[0.000104][1] h2d[0.510578][1] d2h[0.000000][0]
[II] cousteau [Platform.cpp:1020>ShowKernelHistory] kernel[kernel1] k[0.024058][1] h2d[0.000015][2] d2h[0.000000][1]
[II] cousteau [Platform.cpp:1025>ShowKernelHistory] total kernel[0.024162] h2d[0.510594] d2h[0.000021]
[II] cousteau [Platform.cpp:1048:Finalize] total execution time:[2.454120] sec. initialize:[1.413638] sec. t-i:[1.040482] sec
[II] cousteau [Platform.cpp:1049:Finalize] t10[0.000000] t11[0.000000] t12[0.000000] t13[0.000000]
[II] cousteau [Platform.cpp:1050:Finalize] t14[0.000000] t15[0.000000] t16[0.000000] t17[0.000000]
[II] cousteau [Platform.cpp:1051:Finalize] t18[0.000000] t19[0.000000] t20[0.000000] t21[0.000000]
```