IRIS Internals

Jungwon Kim

IRIS mini workshop 2022

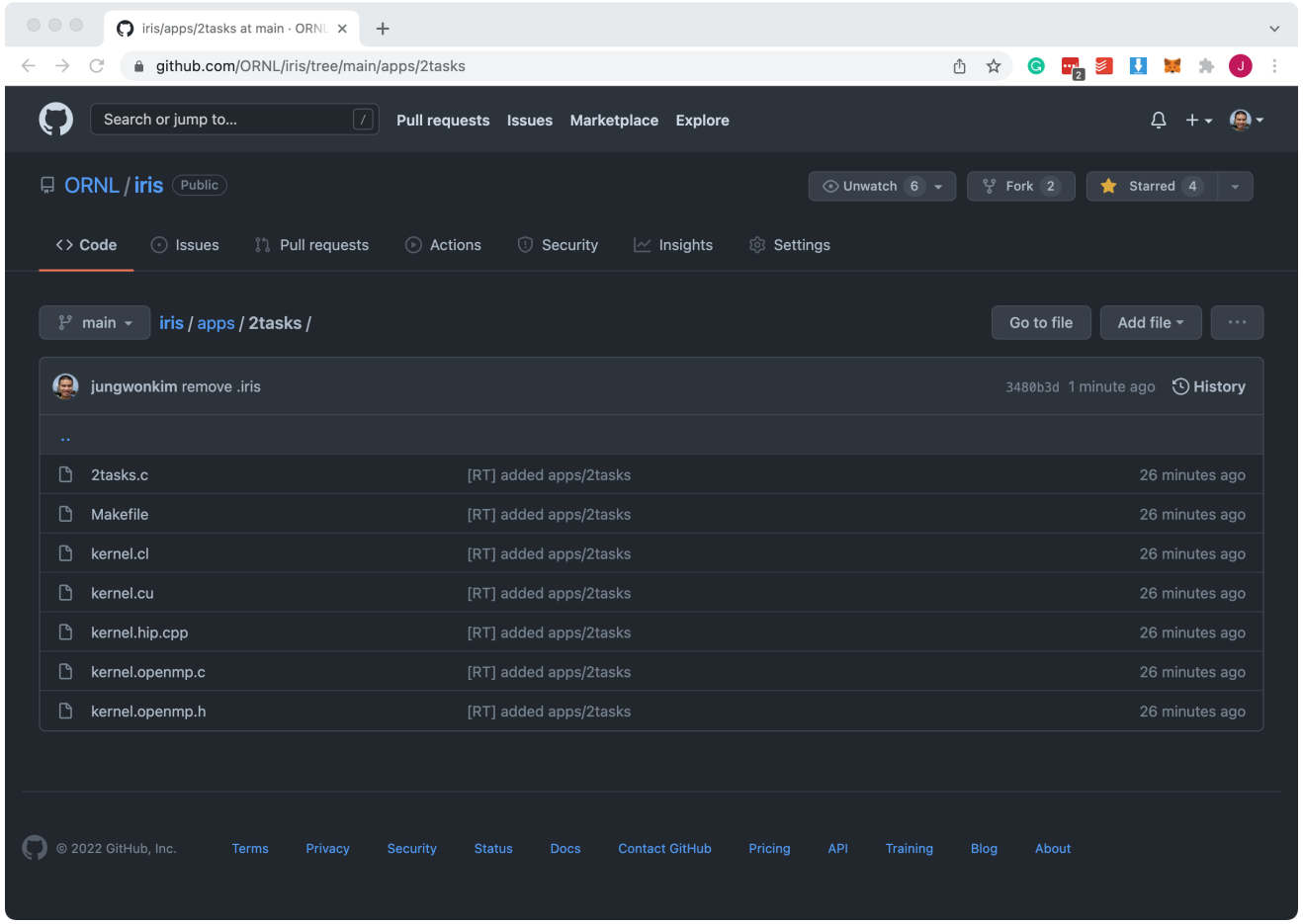
January 4, 2022

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

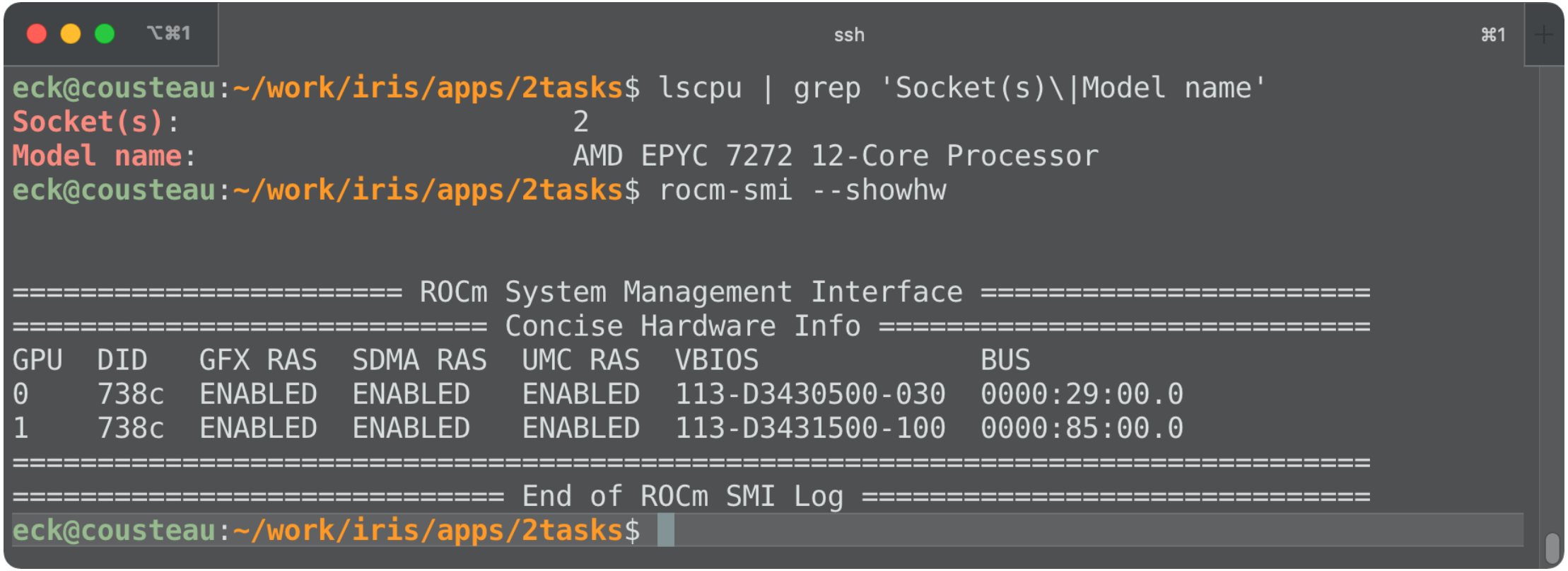
Application: apps/2tasks

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

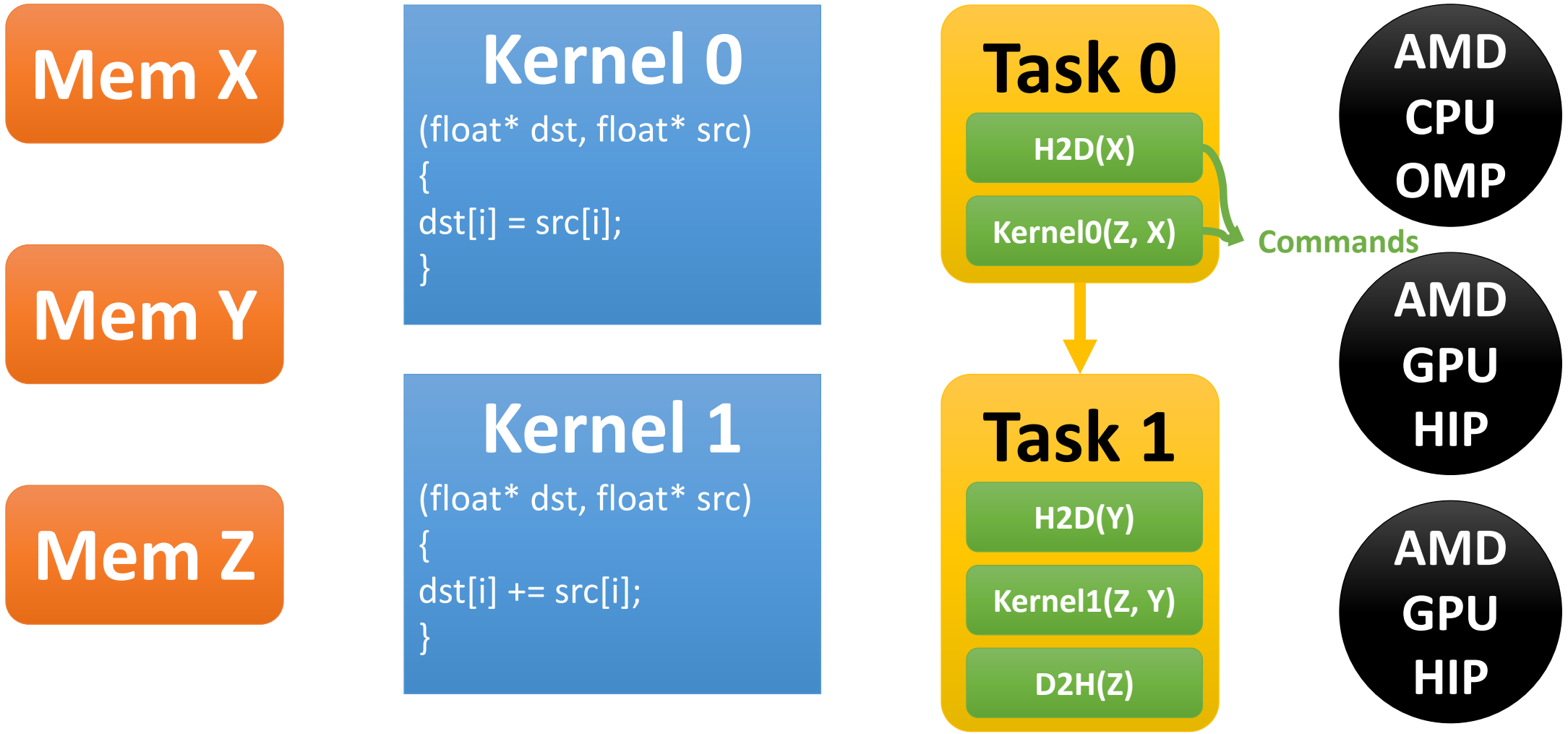


Machine: ExCL/Cousteau

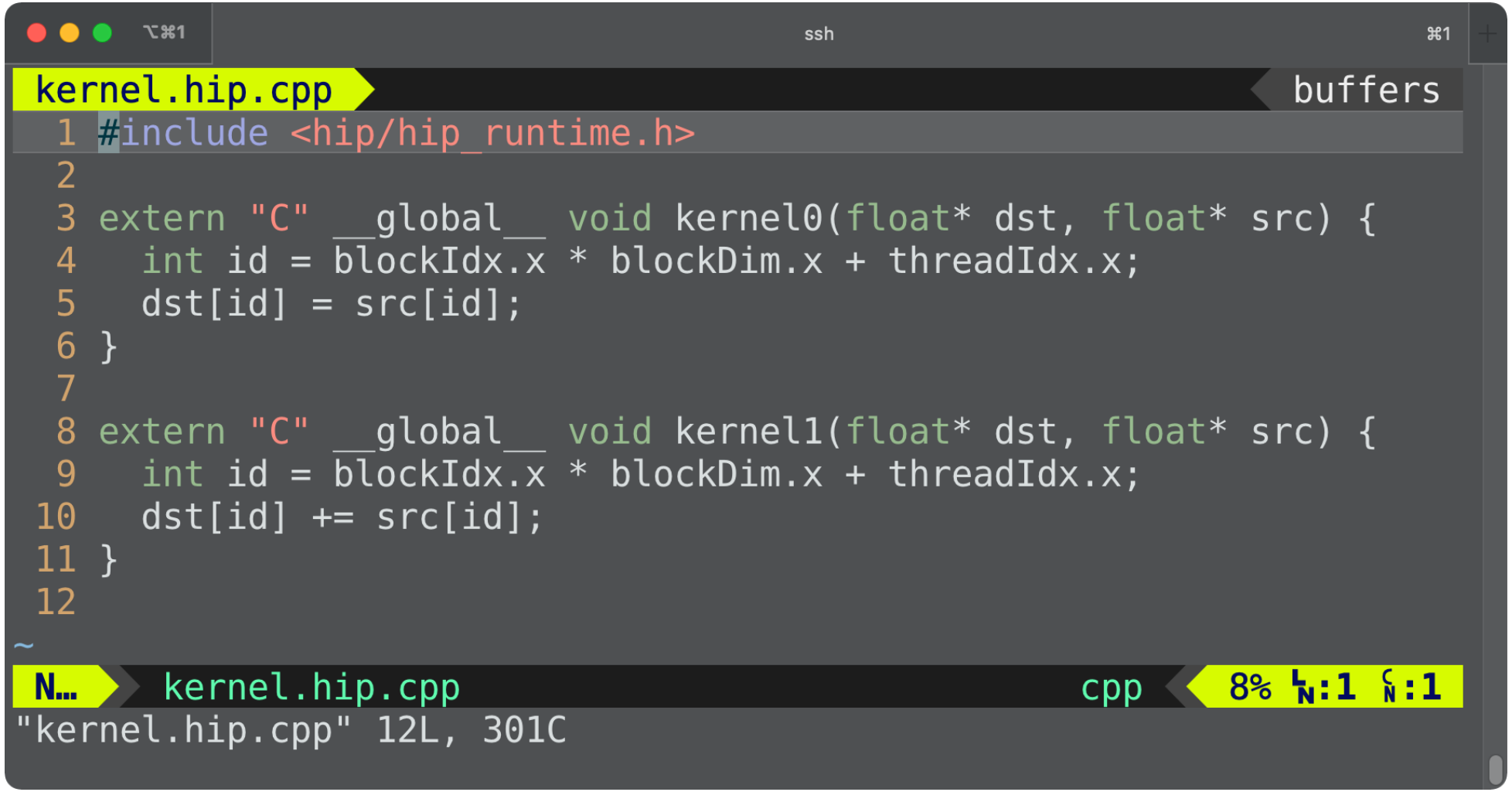
- 2x AMD EPYC 7272 CPUs + 2x AMD MI100 GPUs



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



2tasks/kernel.hip.cpp



2tasks/2tasks.c

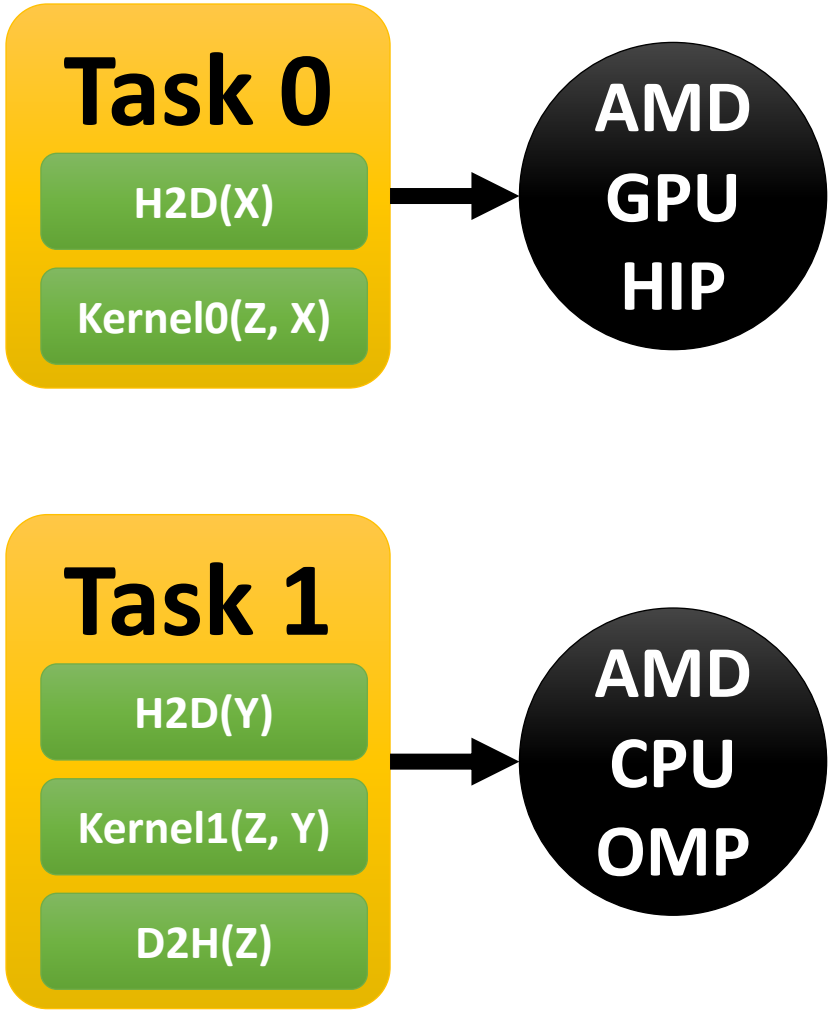
Mem X

Mem Y

Mem Z

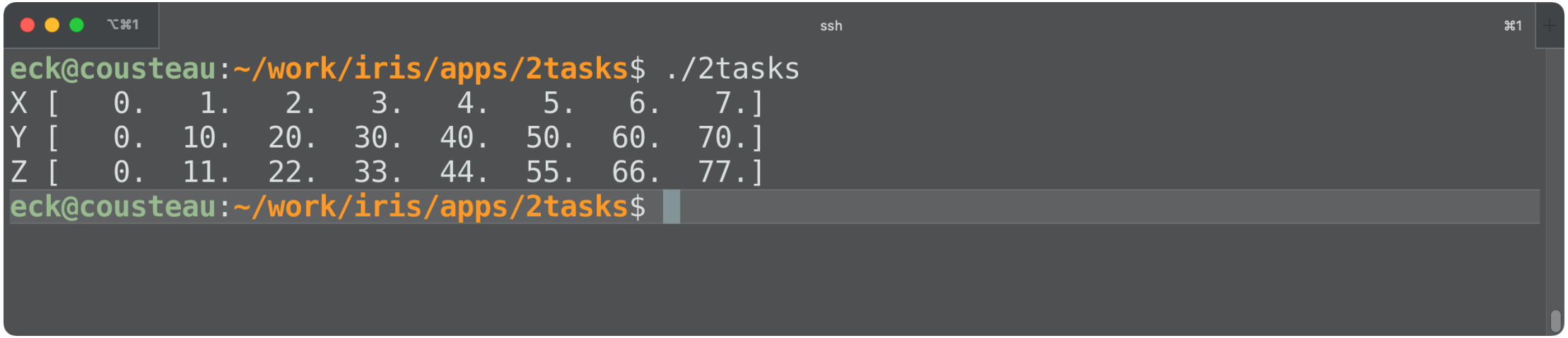
2tasks/2tasks.c

```
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_par
   ams_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_par
   ams_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
```

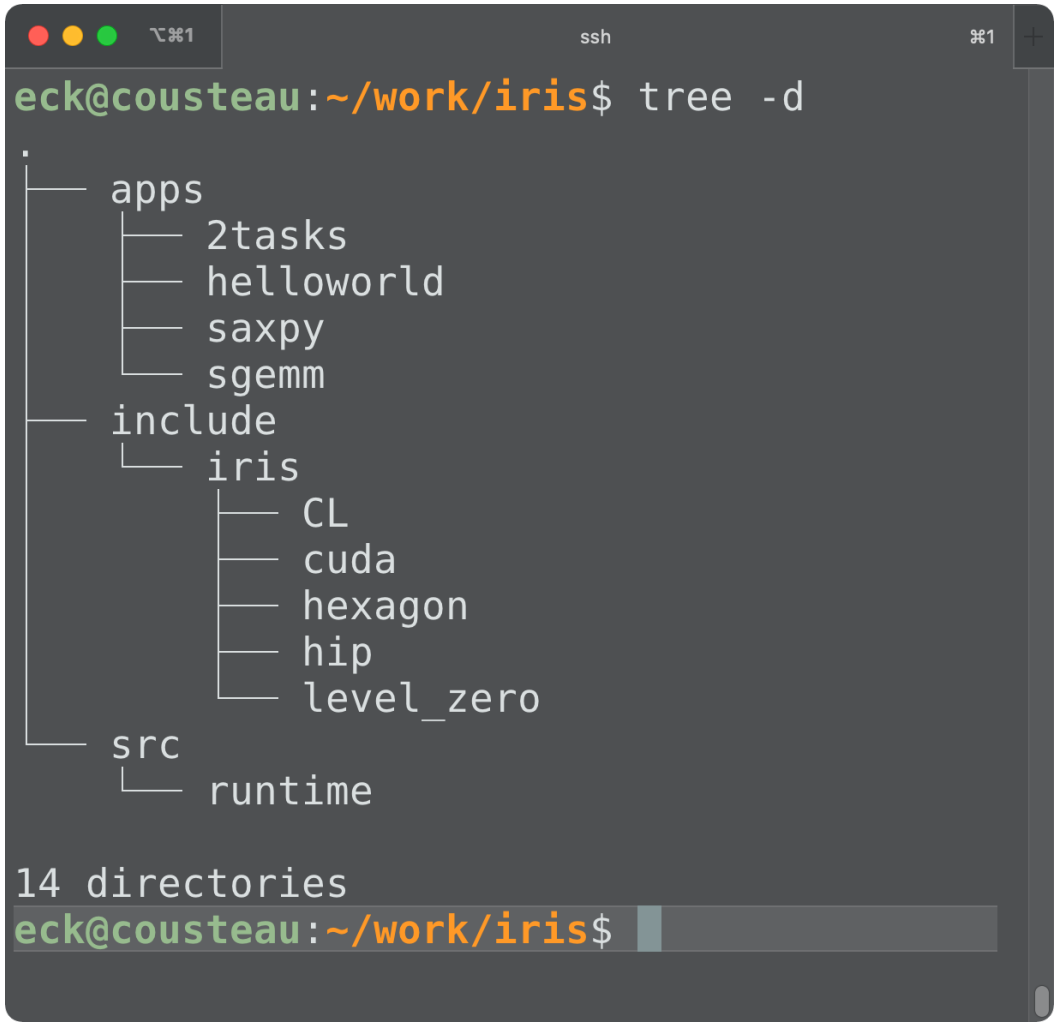


2tasks/2tasks.c

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



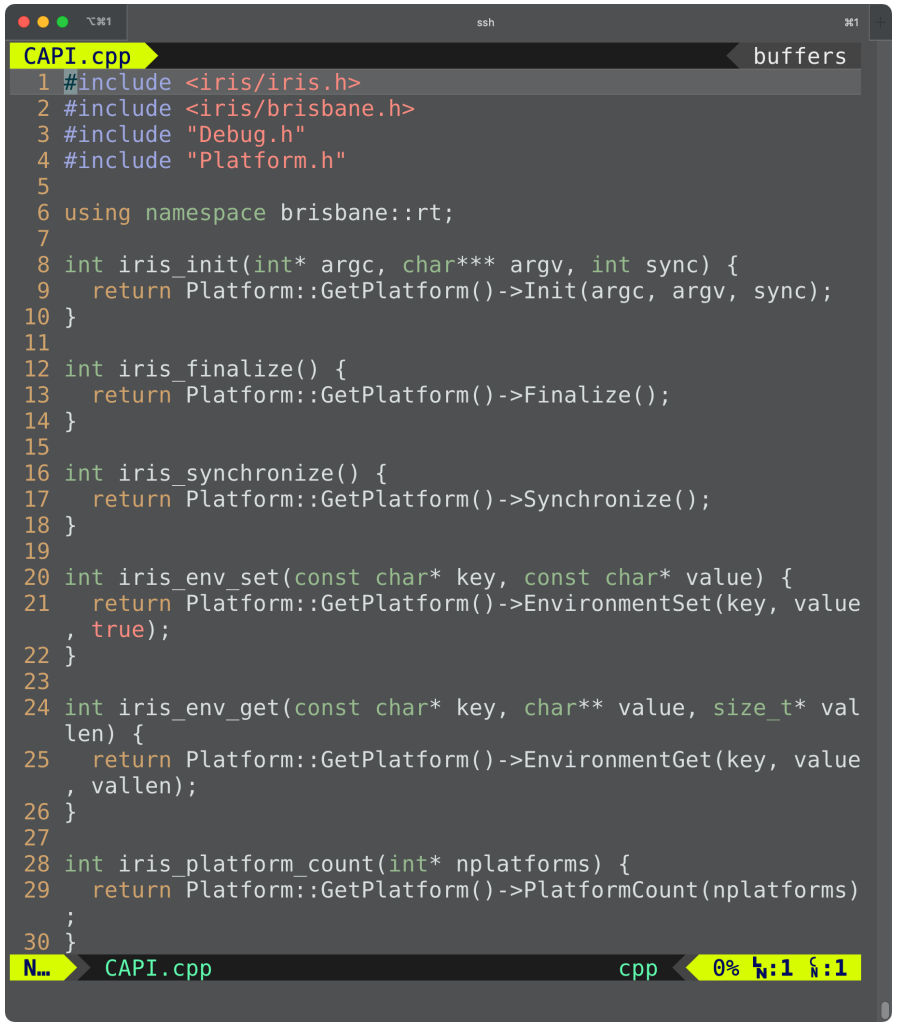
The IRIS Directory Structure



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

iris_init() @ include/iris/iris.h, iris_runtime.h

src/runtime/CAPI.cpp



- 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

- The IRIS Platform manages everything in the IRIS runtime
  - Devices, memories, kernels, tasks, scheduler, workers, loaders, consistency manager, queues

- Platform is a singleton instance in the whole IRIS execution environment

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

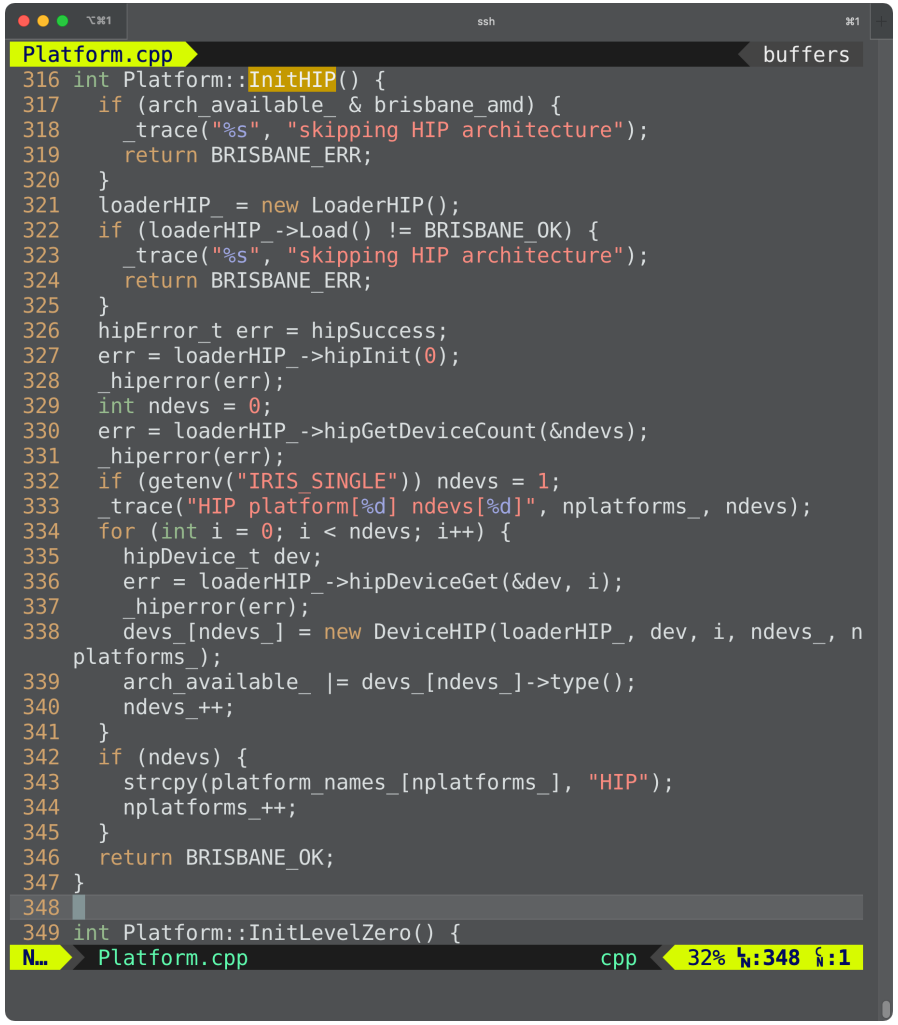
```
Platform.cpp                                                    buffers
104 int Platform::Init(int* argc, char*** argv, int sync) {
105   pthread_mutex_lock(&mutex_);
106   if (init_) {
107     pthread_mutex_unlock(&mutex_);
108     return BRISBANE_ERR;
109   }
110
111   gethostname(brisbane_log_prefix_, 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% ␤:104 ☰:15
search hit BOTTOM, continuing at TOP
```

```
Platform.cpp                                                    buffers
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("ARCHS",  "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   EnvironmentSet("KERNEL_BIN_OPENMP",   "kernel.openmp.so",   false);
218   EnvironmentSet("KERNEL_SRC_SPV",      "kernel.cl",          false);
219   EnvironmentSet("KERNEL_BIN_SPV",      "kernel.spv",         false);
220
221   EnvironmentSet("LOGO",           "off",               false);
222   return BRISBANE_OK;
223 }
224
225 int Platform::EnvironmentSet(const char* key, const char* value, bool
     overwrite) {
226   std::string keystr = std::string(key);
227   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()



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% ℕ:27 ℂ: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% ℕ:31 ℂ: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% ℕ:348 ℂ: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
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% ℕ:31 ℂ: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% ℕ:12 ℂ: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% ℕ:29 ℂ:1

src/runtime/DeviceHIP.cpp

DeviceHIP.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);
```

Device.h cpp 26%

```
15 DeviceHIP::DeviceHIP(LoaderHIP* ld, hipDevice_t dev, int
   , int devno, int platform) : Device(devno, platform) {
16   ld_ = ld;
17   max_arg_idx_ = 0;
18   shared_mem_bytes_ = 0;
19   ordinal_ = ordinal;
20   dev_ = dev;
21   strcpy(vendor_, "Advanced Micro Devices");
22   err_ = ld_->hipDeviceGetName(name_, sizeof(name_), dev_);
23   _hiperror(err_);
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_);
```

N… DeviceHIP.cpp cpp 7% ℓN:15 cN:10

Device Subclasses

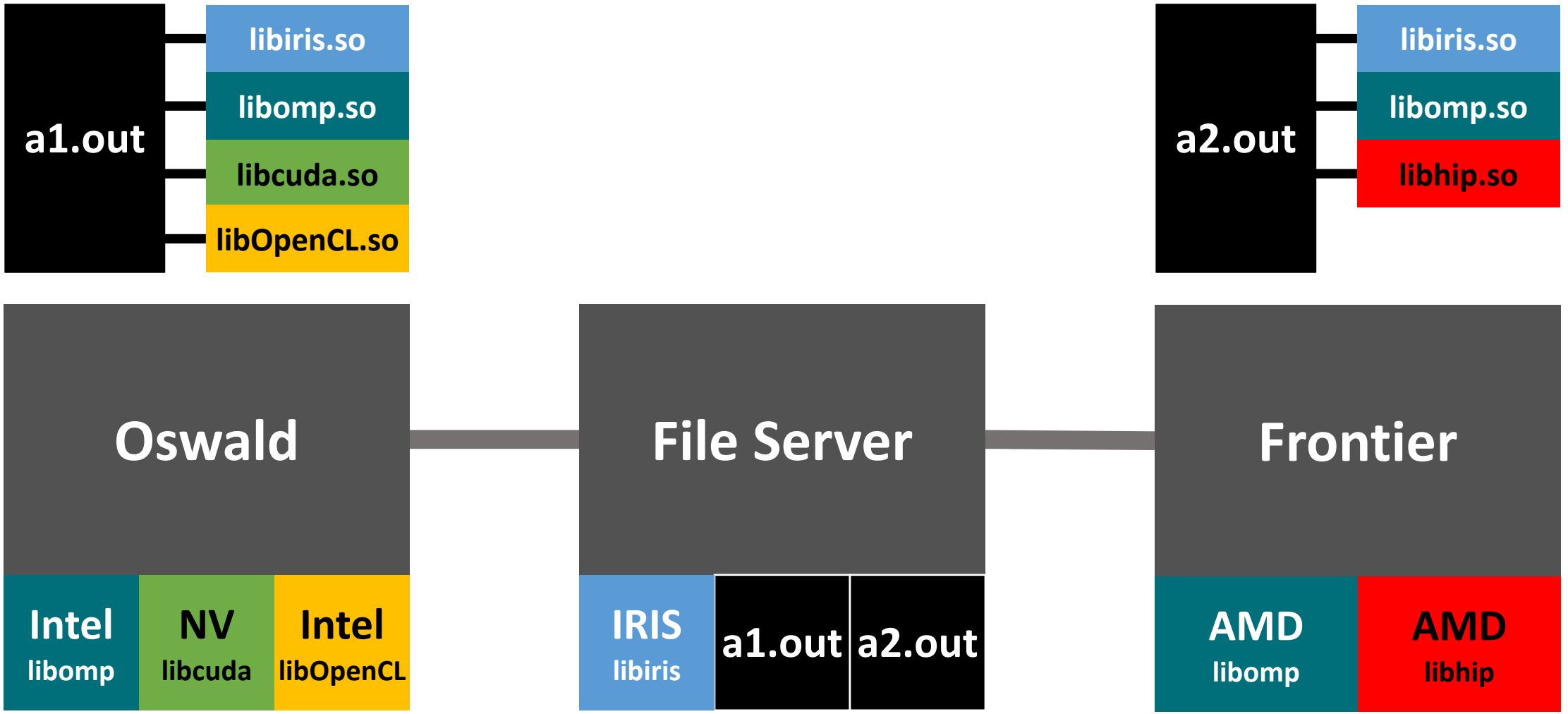
DeviceCUDA
DeviceHexagon
DeviceHIP
DeviceLevelZero
DeviceOpenCL
DeviceOpenMP

Platform.cpp buffers

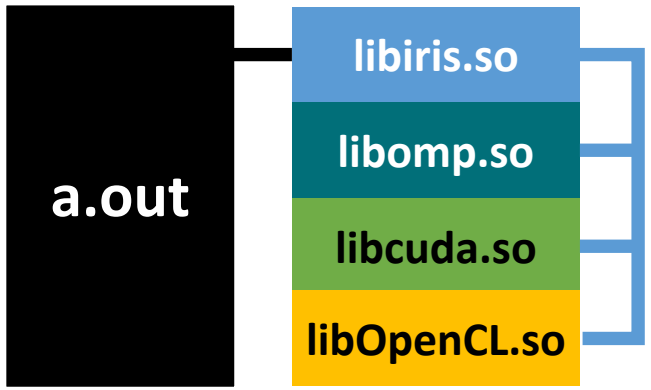
```
316 int Platform::InitHIP() {
317   if (arch_available_ & brisbane_amd) {
318     _trace("%s", "skipping HIP architecture");
319     return BRISBANE_ERR;
320   }
321   loaderHIP_ = new LoaderHIP();
322   if (loaderHIP_->Load() != BRISBANE_OK) {
323     _trace("%s", "skipping HIP architecture");
324     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_, 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% ℓN:348 cN:1

Without DPL: Not Portable Executable

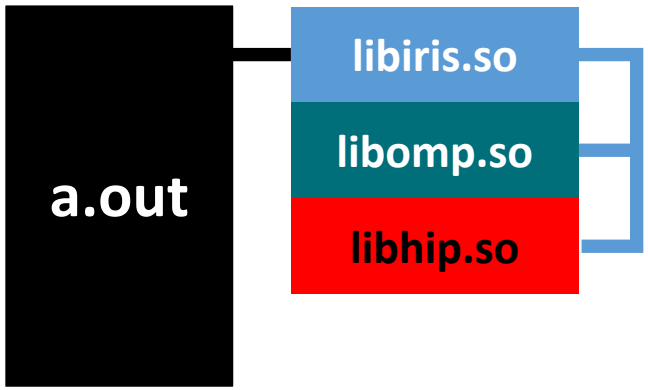


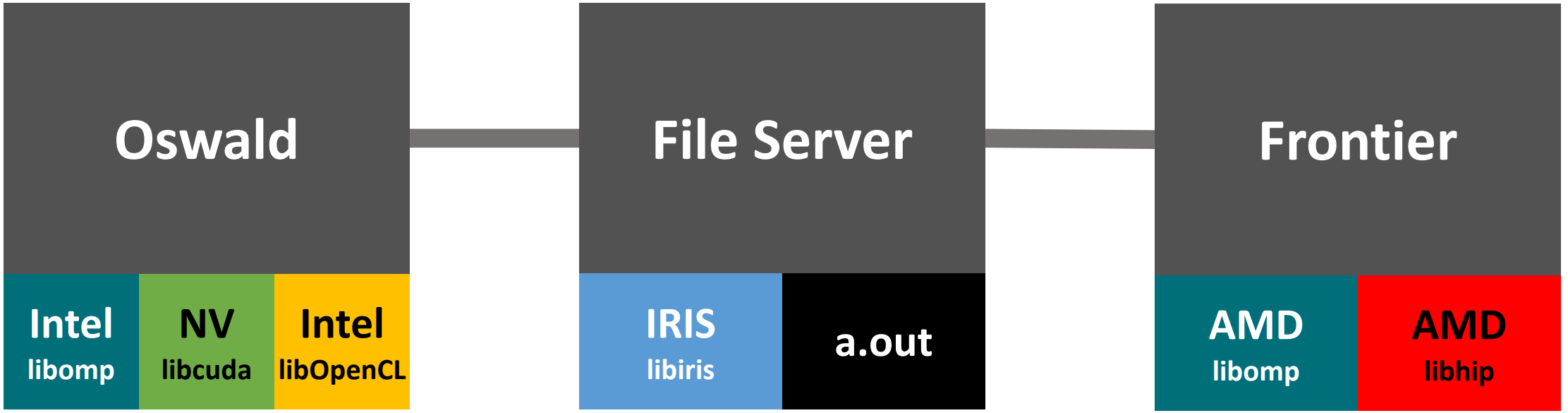
Dynamic Platform Loader: Portable Executable



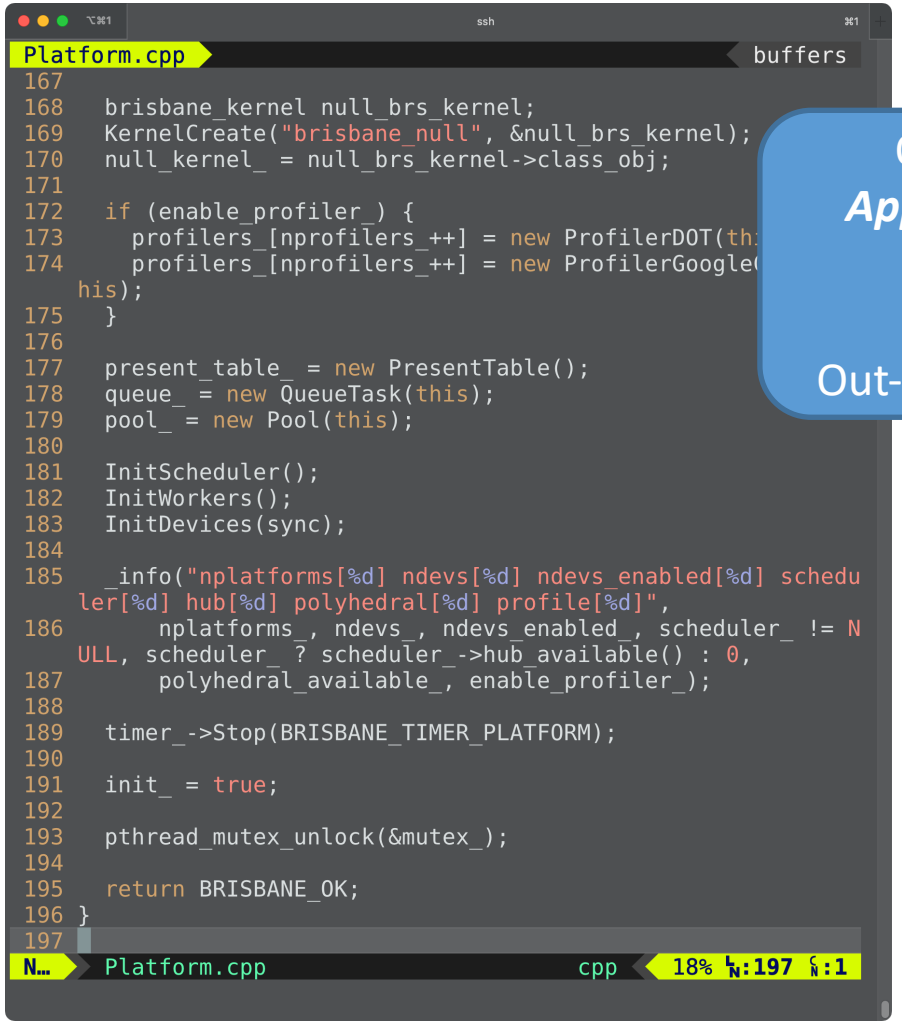
$ CC app.c -o a.out -liris

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



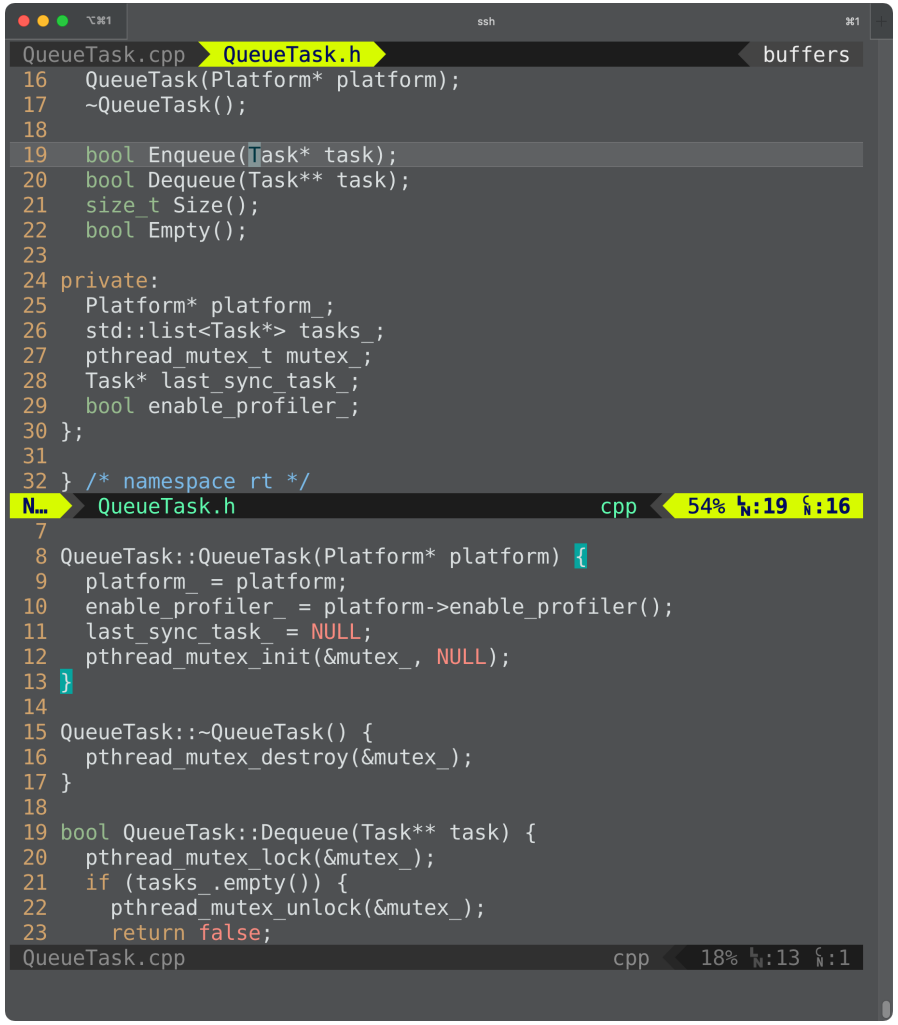


src/runtime/QueueTask.cpp



QueueTask:
Application Task Queue

Out-or-order queue

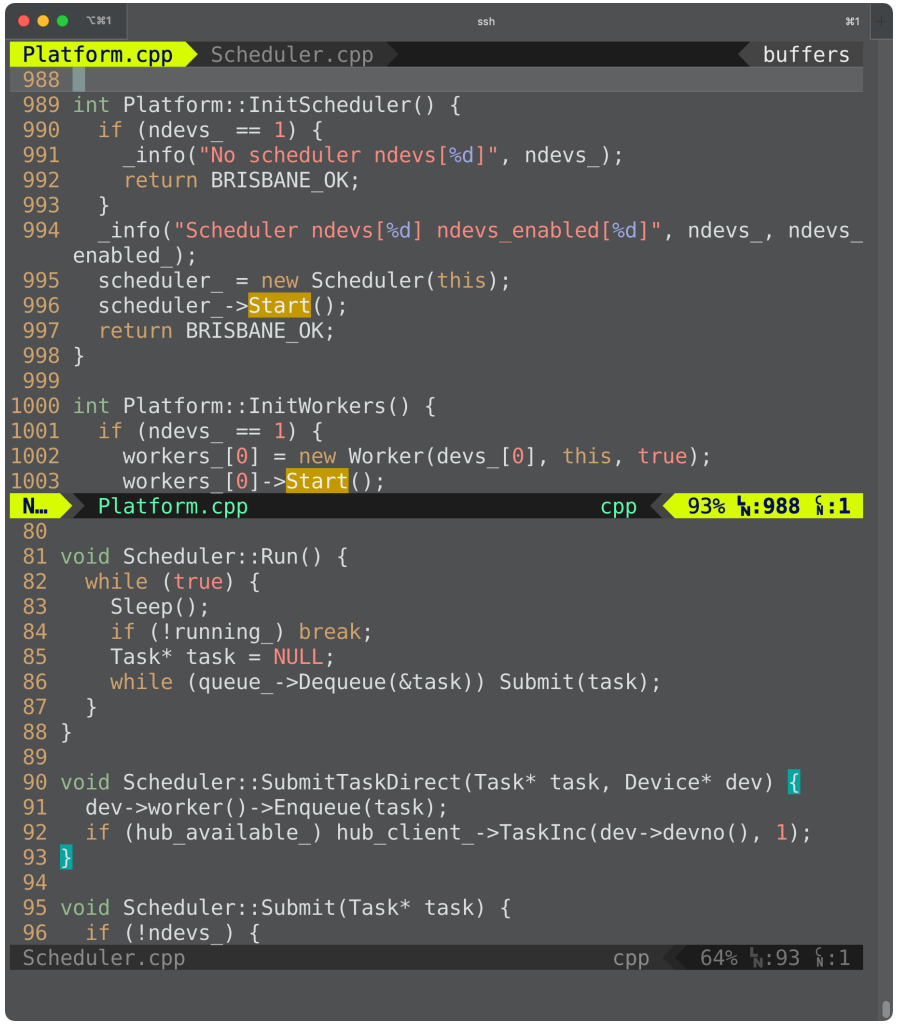


src/runtime/Scheduler.cpp

Scheduler

a pthread instance

scheduling the Application Task Queue



src/runtime/Worker.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(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_, schedule
    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 c:1
```

Worker

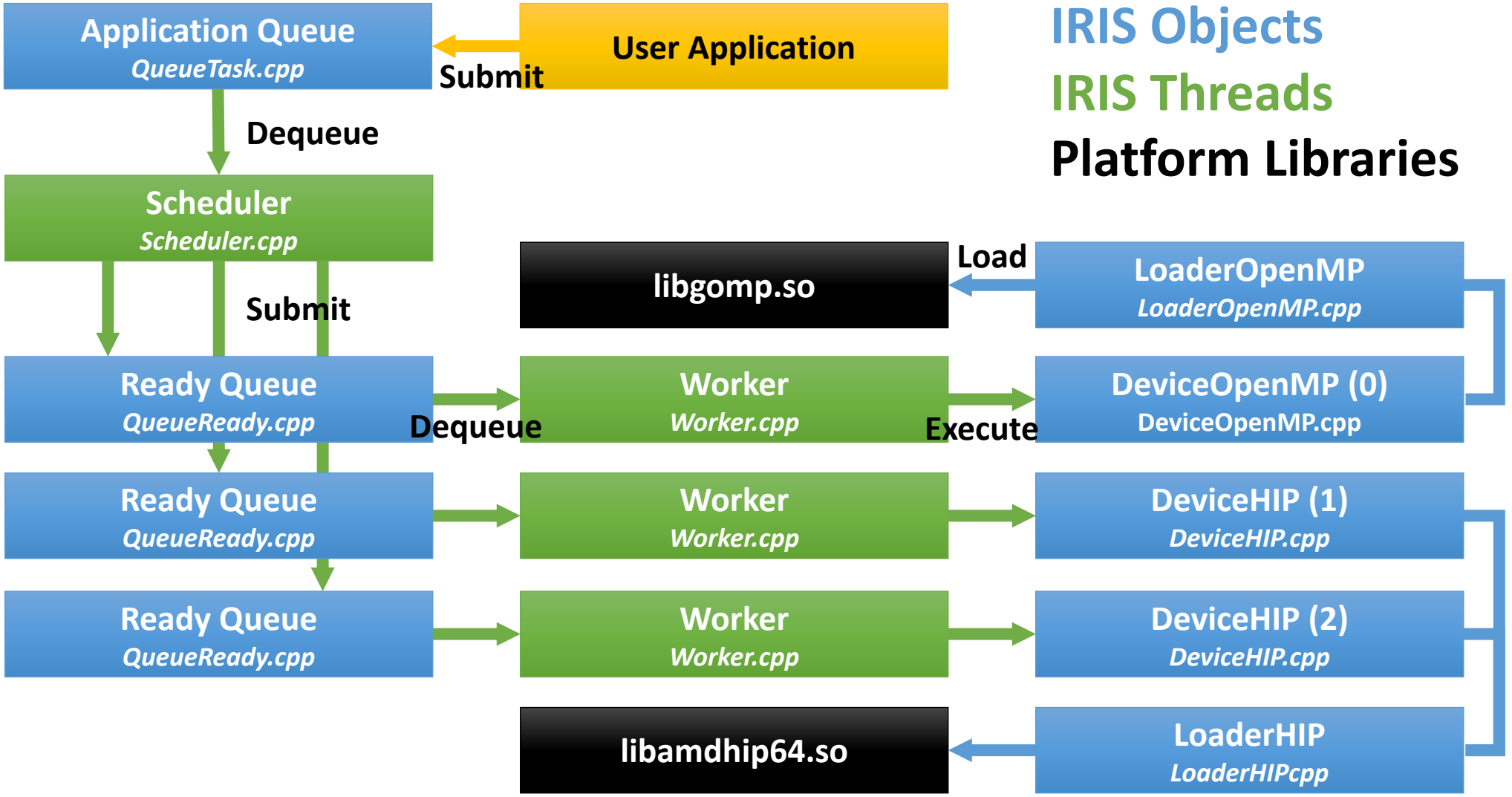
a pthread instance

N workers for
N devices

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

```
Platform.cpp  Worker.cpp                                        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;
N…  Platform.cpp                                  cpp   94% ℓ:1000 c: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 }
 26
 27 Worker::~Worker() {
 28   if (single_) delete queue_;
 29 }
Worker.cpp                                        cpp   35% ℓ:26 c:1
 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() {
Worker.cpp                                        cpp   93% ℓ:69 c:14
```

The IRIS Platform So Far

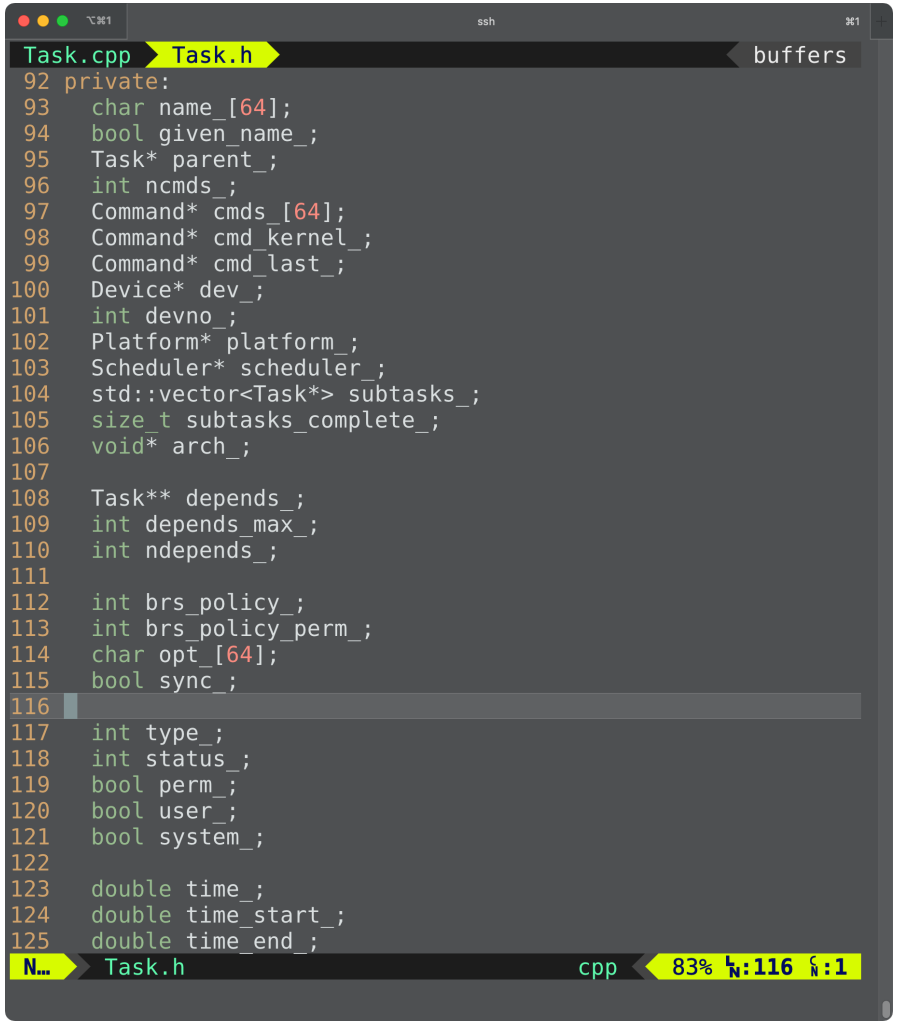


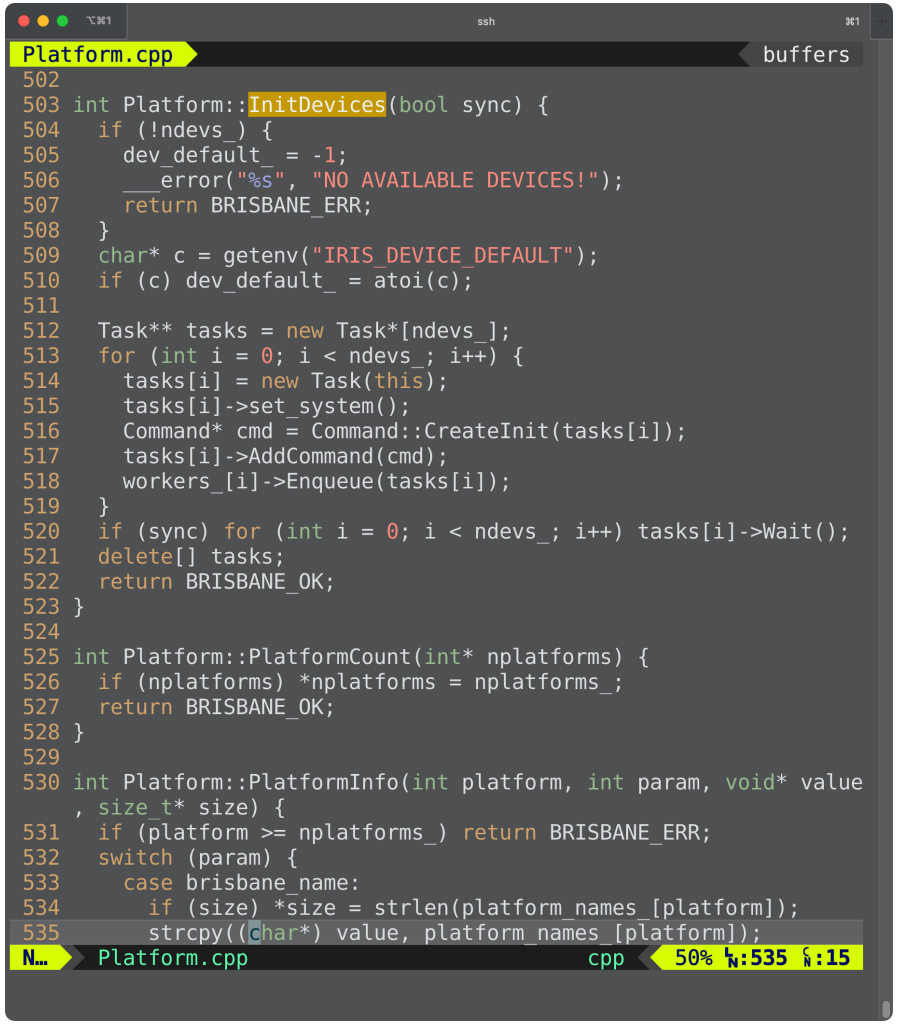
src/runtime/Platform.cpp

```
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
N…  Platform.cpp                                  cpp   18% ℓ:197 c:1
```

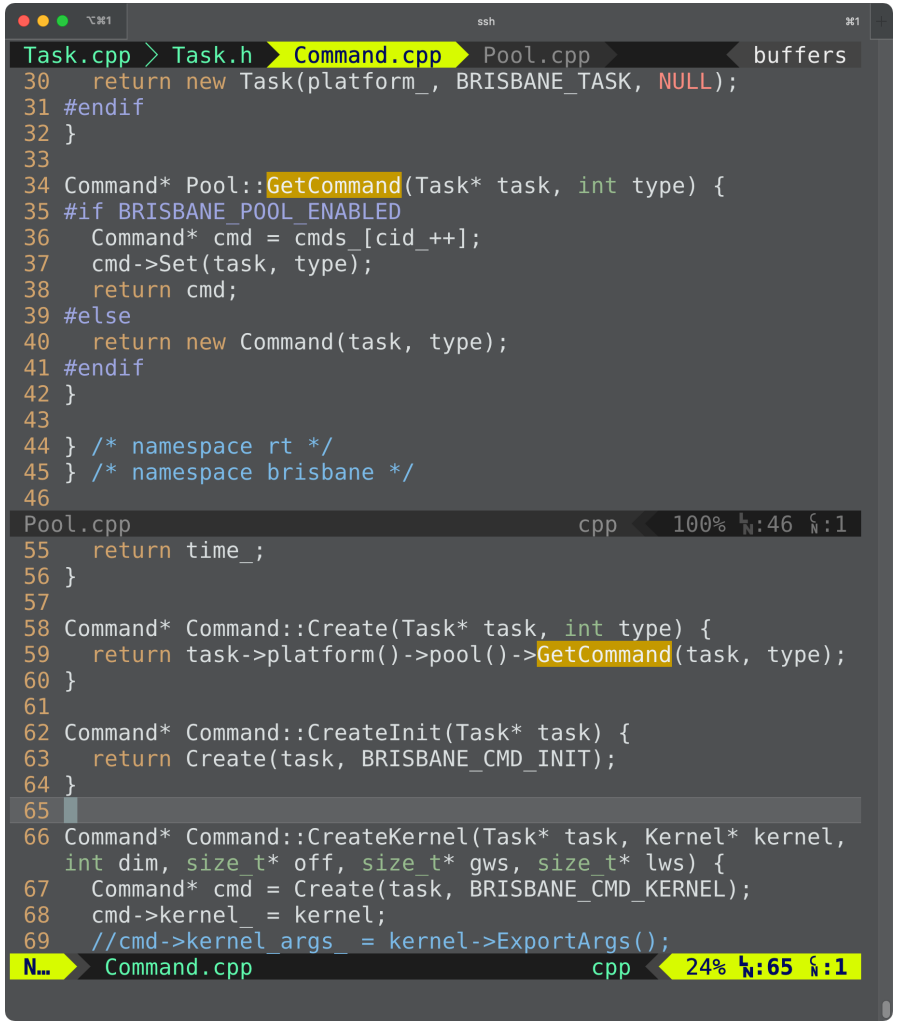
```
Platform.cpp                                                    buffers
502
503 int Platform::InitDevices(bool sync) {
504   if (!ndevs_) {
505     dev_default_ = -1;
506     ___error("%s", "NO AVAILABLE DEVICES!");
507     return BRISBANE_ERR;
508   }
509   char* c = getenv("IRIS_DEVICE_DEFAULT");
510   if (c) dev_default_ = atoi(c);
511
512   Task** tasks = new Task*[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]);
N…  Platform.cpp                                  cpp   50% ℓ:535 c:15
```

src/runtime/Task.h





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                                  cpp  100% ℓ:46 c: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,
   int dim, size_t* off, size_t* gws, size_t* lws) {
67   Command* cmd = Create(task, BRISBANE_CMD_KERNEL);
68   cmd->kernel_ = kernel;
69   //cmd->kernel_args_ = kernel->ExportArgs();
N…  Command.cpp                           cpp  24% ℓ:65 c:1
```

```
Platform.cpp                                              buffers
502
503 int Platform::InitDevices(bool sync) {
504   if (!ndevs_) {
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*[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]);
N…  Platform.cpp                          cpp  50% ℓ:535 c:15
```

src/runtime/Task.cpp

Task.cpp buffers

```
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_];
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% ℓ:151 c:1

"Task.cpp" 196L, 5102C written

Platform.cpp buffers

```
502
503 int Platform::InitDevices(bool sync) {
504   if (!ndevs_) {
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*[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]);
```

N… Platform.cpp cpp 50% ℓ:535 c:15

src/runtime/Worker.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
```

N… Worker.cpp cpp 54% ℓ:40 c:1

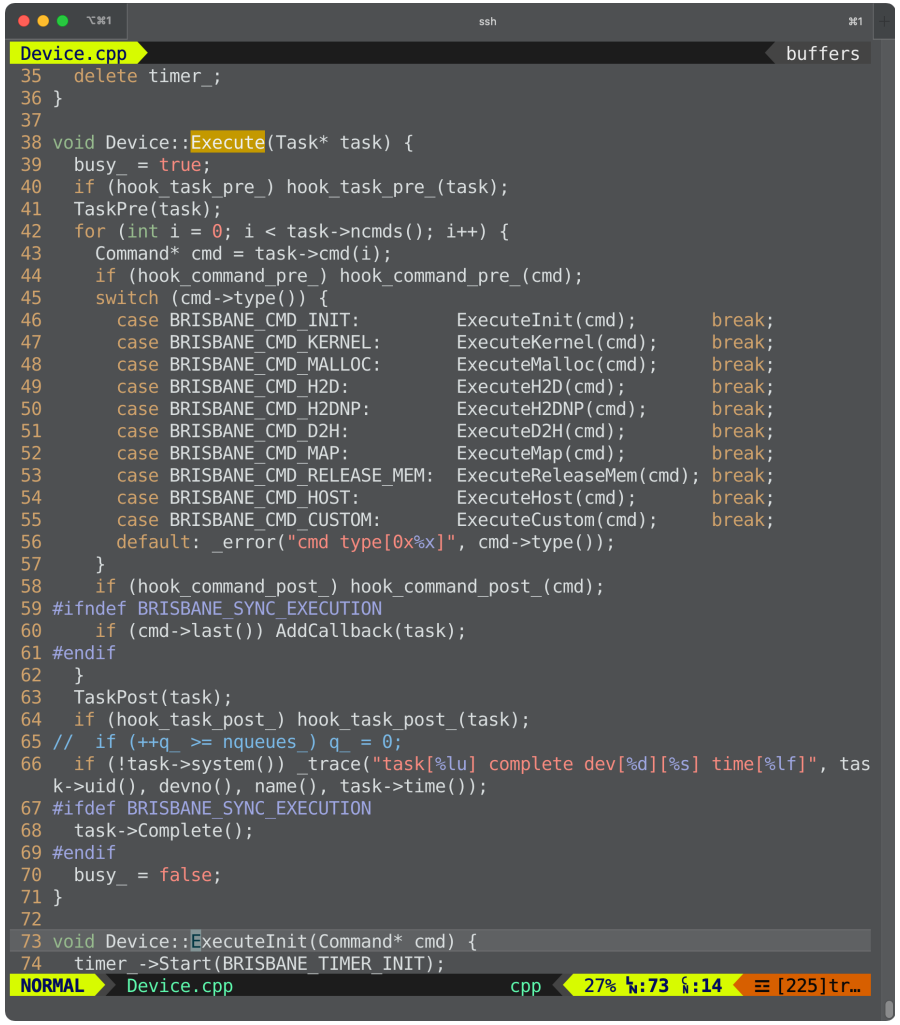
Platform.cpp buffers

```
502
503 int Platform::InitDevices(bool sync) {
504   if (!ndevs_) {
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*[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]);
```

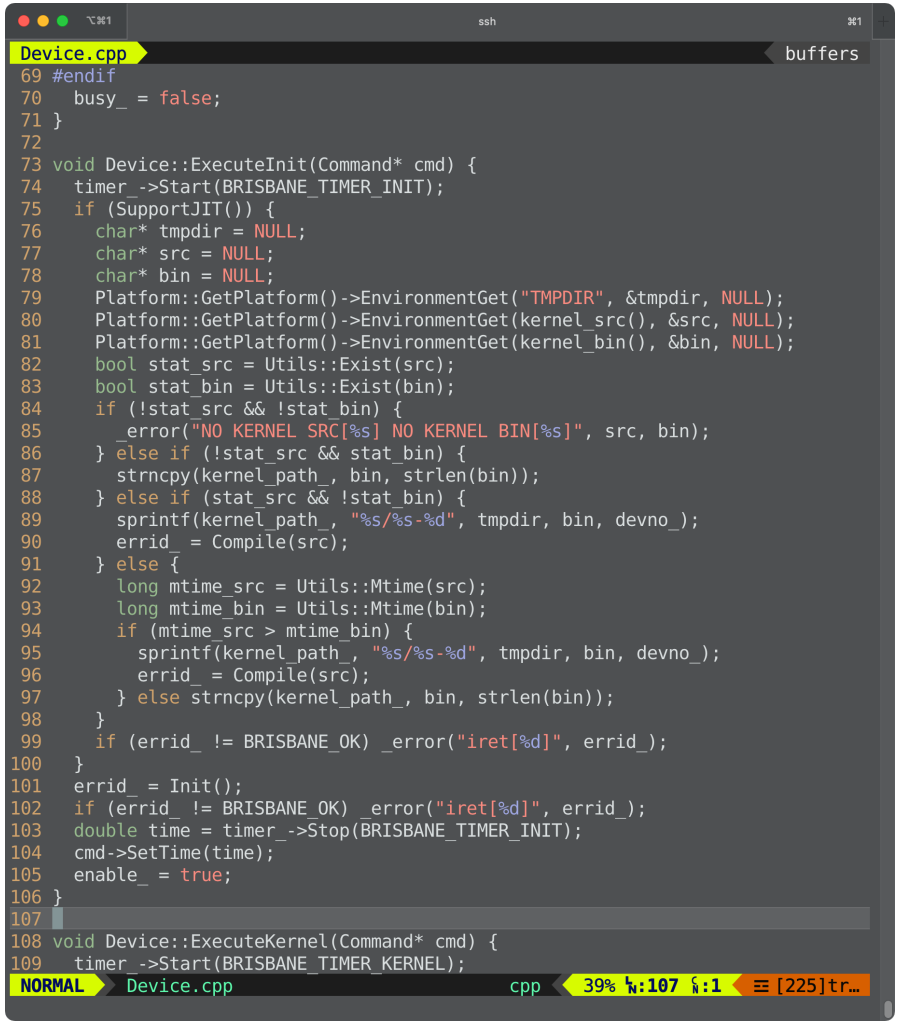
N… Platform.cpp cpp 50% ℓ:535 c:15

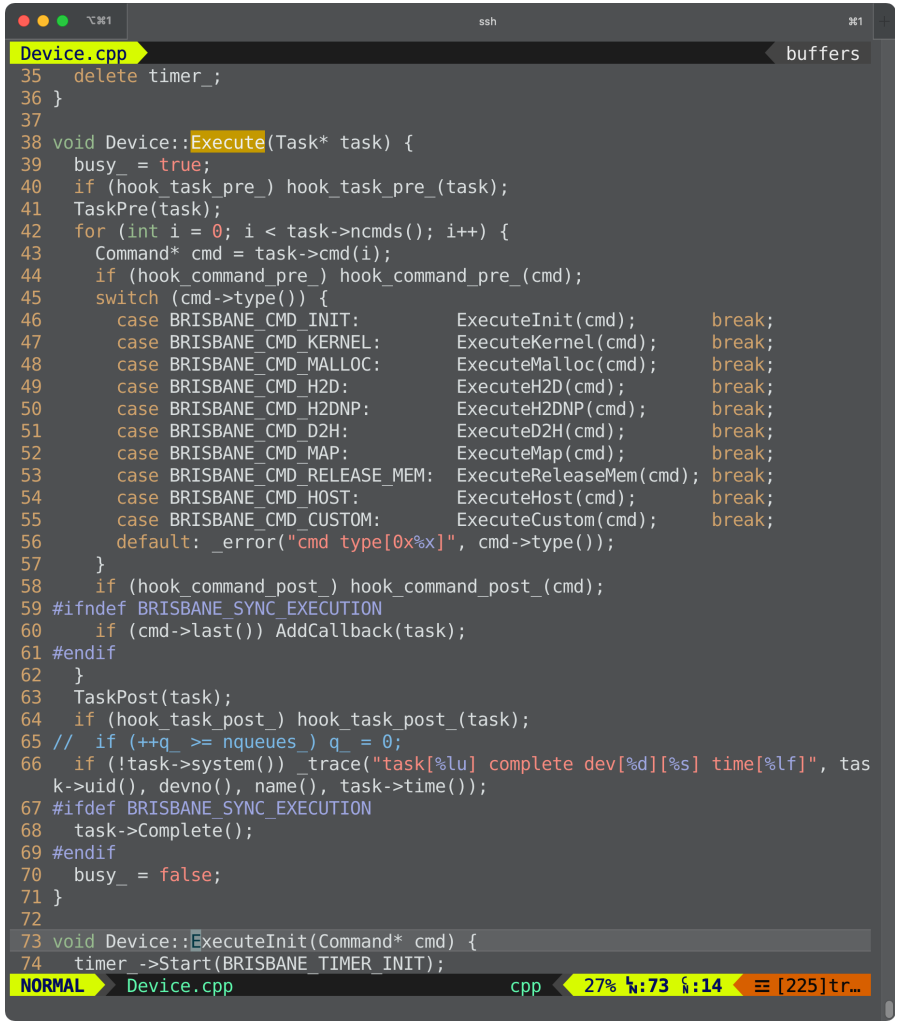
src/runtime/Thread.cpp

src/runtime/Device.cpp



src/runtime/Device.cpp





src/runtime/DeviceHIP.cpp

Device.cpp buffers

```
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 strncpy(kernel_path_, bin, strlen(bin));
98     }
99     if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
100  }
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% ℓ:107 c:1 ☰ [225]tr…

Device.cpp DeviceHIP.cpp buffers

```
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] v
   ersion[%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, hipDeviceAttributeMultiprocessorCoun
   t, devid_);
54   err_ = ld_->hipDeviceGetAttribute(&bx, hipDeviceAttributeMaxBlockDimX, devi
   d_);
55   err_ = ld_->hipDeviceGetAttribute(&by, hipDeviceAttributeMaxBlockDimY, devi
   d_);
56   err_ = ld_->hipDeviceGetAttribute(&bz, hipDeviceAttributeMaxBlockDimZ, devi
   d_);
57   err_ = ld_->hipDeviceGetAttribute(&dx, hipDeviceAttributeMaxGridDimX, devid
   );
```

NORMAL DeviceHIP.cpp cpp 21% ℓ:45 c:1

src/runtime/DeviceHIP.cpp

```
Device.cpp                                                    buffers
 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 strncpy(kernel_path_, bin, strlen(bin));
 98     }
 99     if (errid_ != BRISBANE_OK) _error("iret[%d]", errid_);
100   }
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% ␤:107 ␡:1  ☰ [225]tr…
```

```
DeviceHIP.cpp  Worker.cpp  Device.cpp                                 buffers
 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% ␤:93/213☰ ␡:16
```

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(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 c:1
```

All Devices

Compiled kernels
Loaded kernels

Initialized underlying platform objects

```
Platform.cpp                                                    buffers
502
503 int Platform::InitDevices(bool sync) {
504   if (!ndevs_) {
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*[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]);
N…  Platform.cpp                                  cpp   50% ℓ:535 c:15
```

iris_mem_create()

2tasks.c buffers

```c
#include <iris/iris.h>
#include <stdio.h>
#include <malloc.h>

int main(int argc, char** argv) {
  iris_init(&argc, &argv, 1);

  size_t SIZE = 8;
  float *X, *Y, *Z;

  X = (float*) malloc(SIZE * sizeof(float));
  Y = (float*) malloc(SIZE * sizeof(float));
  Z = (float*) malloc(SIZE * sizeof(float));

  for (int i = 0; i < SIZE; i++) {
    X[i] = i;
    Y[i] = i * 10;
  }

  printf("X [");
  for (int i = 0; i < SIZE; i++) printf(" %3.0f.", X
  printf("]\n");
  printf("Y [");
  for (int i = 0; i < SIZE; i++) printf(" %3.0f.", Y[i]);
  printf("]\n");

  iris_mem mem_X;
  iris_mem mem_Y;
  iris_mem mem_Z;
  iris_mem_create(SIZE * sizeof(float), &mem_X);
  iris_mem_create(SIZE * sizeof(float), &mem_Y);
  iris_mem_create(SIZE * sizeof(float), &mem_Z);

```

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

IRIS Memory

A logical handle for multiple physical device memory spaces

Can be shared across multiple devices

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 }
```

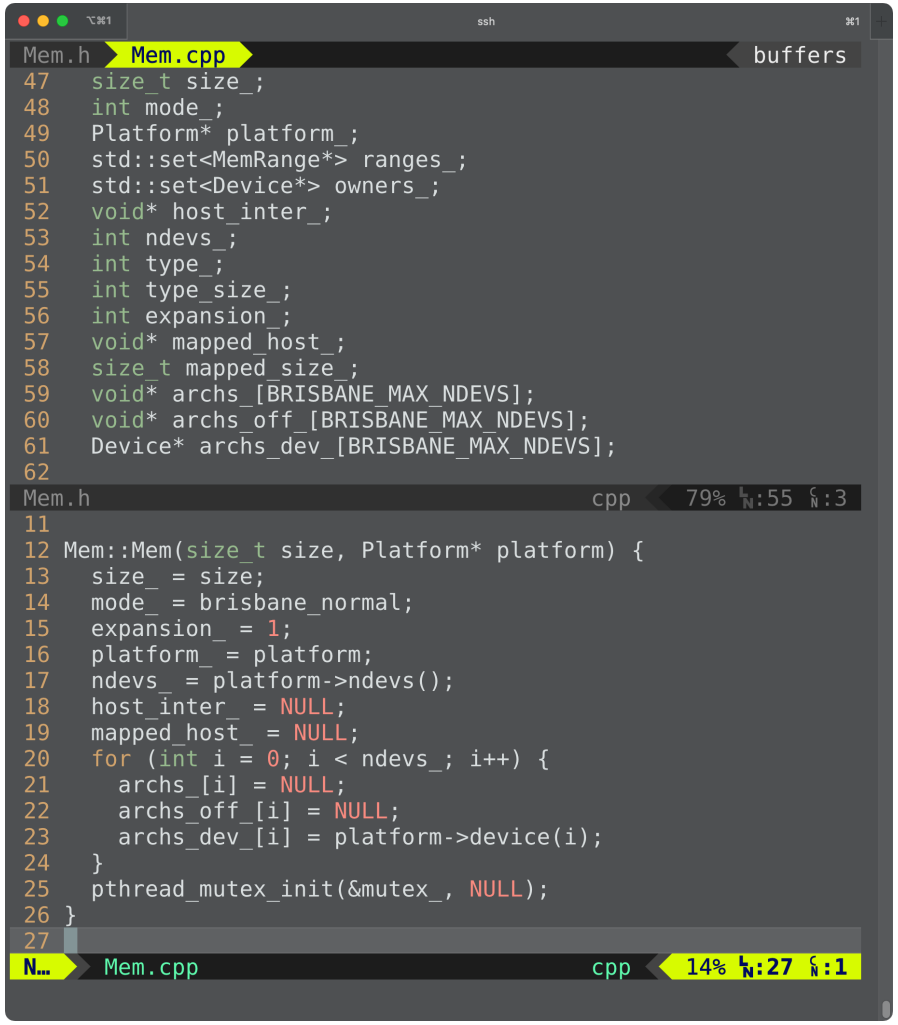
CAPI.cpp cpp 28% ☰:111 ☰:1

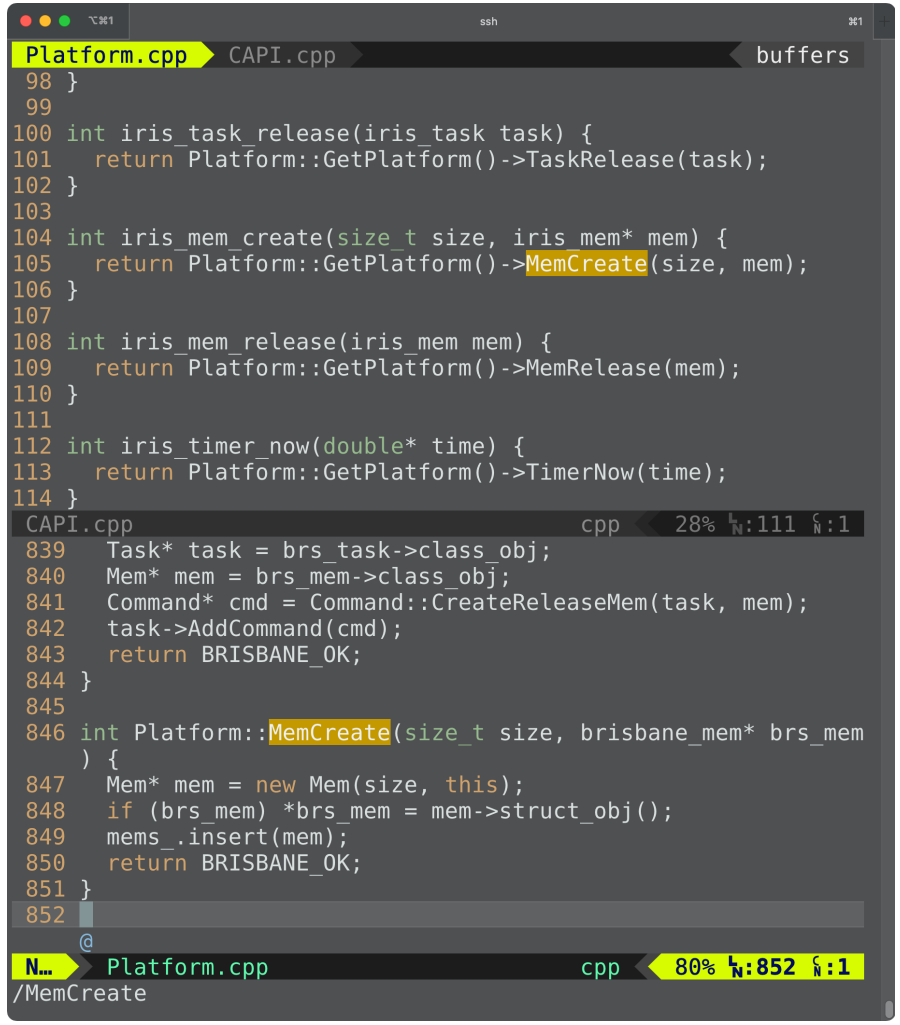
```
839   Task* task = brs_task->class_obj;
840   Mem* mem = brs_mem->class_obj;
841   Command* cmd = Command::CreateReleaseMem(task, mem);
842   task->AddCommand(cmd);
843   return BRISBANE_OK;
844 }
845
846 int Platform::MemCreate(size_t size, brisbane_mem* brs_mem
    ) {
847   Mem* mem = new Mem(size, this);
848   if (brs_mem) *brs_mem = mem->struct_obj();
849   mems_.insert(mem);
850   return BRISBANE_OK;
851 }
852
```

N… Platform.cpp cpp 80% ☰:852 ☰:1

/MemCreate

src/runtime/Mem.cpp





iris_task_h2d_full()

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_par
   ams_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_par
   ams_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 }
```

Platform.cpp Task.h iris_runtime.h

```
726
727 int Platform::TaskH2D(brisbane_task brs_task, brisbane_mem
     brs_mem, size_t off, size_t size, void* host) {
728   Task* task = brs_task->class_obj;
729   Mem* mem = brs_mem->class_obj;
730   Command* cmd = Command::CreateH2D(task, mem, off, size,
    host);
731   task->AddCommand(cmd);
732   return BRISBANE_OK;
733 }
734
735 int Platform::TaskD2H(brisbane_task brs_task, brisbane_mem
     brs_mem, size_t off, size_t size, void* host) {
736   Task* task = brs_task->class_obj;
737   Mem* mem = brs_mem->class_obj;
738   Command* cmd = Command::CreateD2H(task, mem, off, size,
    host);
739   task->AddCommand(cmd);
740   return BRISBANE_OK;
741 }
742
743 int Platform::TaskH2DFull(brisbane_task brs_task, brisbane
    _mem brs_mem, void* host) {
744   return TaskH2D(brs_task, brs_mem, 0ULL, brs_mem->class_o
    bj->size(), host);
745 }
746
747 int Platform::TaskD2HFull(brisbane_task brs_task, brisbane
    _mem brs_mem, void* host) {
748   return TaskD2H(brs_task, brs_mem, 0ULL, brs_mem->class_o
    bj->size(), host);
749 }
750
```

iris_task_kernel()

2tasks.c

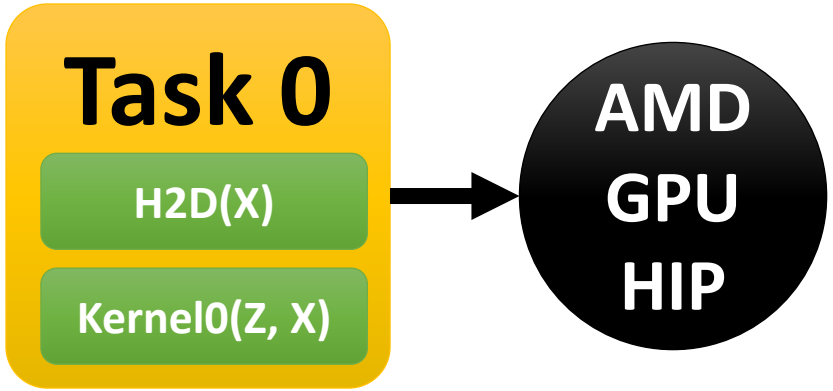
```
34   iris_task task0;
35   iris_task_create(&task0);
36   iris_task_h2d_full(task0, mem_X, X);
37   void* task0_params[2] = { mem_Z, mem_X };
38   int task0_params_info[2] = { iris_w, iris_r };
39   iris_task_kernel(task0, "kernel0", 1, NULL, &SIZE, NULL, 2, task0_params, task0_par
     ams_info);
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_par
     ams_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 }
```

Platform.cpp Task.h iris_runtime.h

```
692   task->AddCommand(cmd);
693   return BRISBANE_OK;
694 }
695
696 int Platform::TaskKernel(brisbane_task brs_task, const cha
    r* name, int dim, size_t* off, size_t* gws, size_t* lws, i
    nt 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, m
    emranges);
700   task->AddCommand(cmd);
701   return BRISBANE_OK;
702 }
703
704 int Platform::TaskKernelSelector(brisbane_task brs_task, b
    risbane_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_ho
    st_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 }
```

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_par
   ams_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_par
   ams_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
```



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_par
   ams_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_par
   ams_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

...> 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% ㏑:90 ㏇:39

```
794 }
795
796 int Platform::TaskSubmit(brisbane_task brs_task, int brs_p
    olicy, const char* opt, int sync) {
797   Task* task = brs_task->class_obj;
798   task->Submit(brs_policy, opt, sync);
799   if (recording_) json_->RecordTask(task);
800   if (scheduler_) {
801     FilterSubmitExecute(task);
802     scheduler_->Enqueue(task);
803   } else workers_[0]->Enqueue(task);
804   if (sync) task->Wait();
805   return BRISBANE_OK;
806 }
807
808 int Platform::TaskWait(brisbane_task brs_task) {
809   Task* task = brs_task->class_obj;
```

Platform.cpp cpp 76% ㏑:809 ㏇: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_par
   ams_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_par
   ams_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

.../ Structs.h / CAPI.cpp Scheduler.cpp buffers

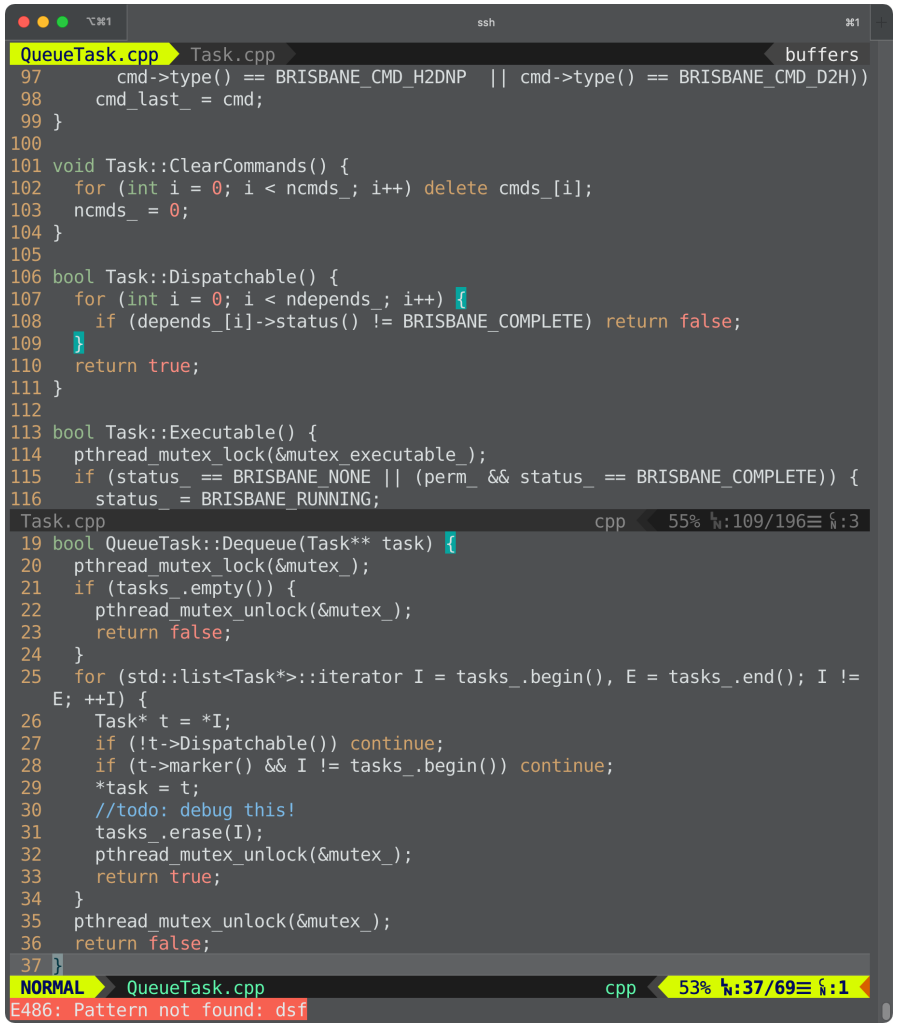
```
75
76 void Scheduler::Enqueue(Task* task) {
77   while (!queue_->Enqueue(task)) { }
78   Invoke();
79 }
80
81 void Scheduler::Run() {
82   while (true) {
83     Sleep();
84     if (!running_) break;
85     Task* task = NULL;
86     while (queue_->Dequeue(&task)) Submit(task);
87   }
88 }
89
90 void Scheduler::SubmitTaskDirect(Task* task, Device* dev) {
```

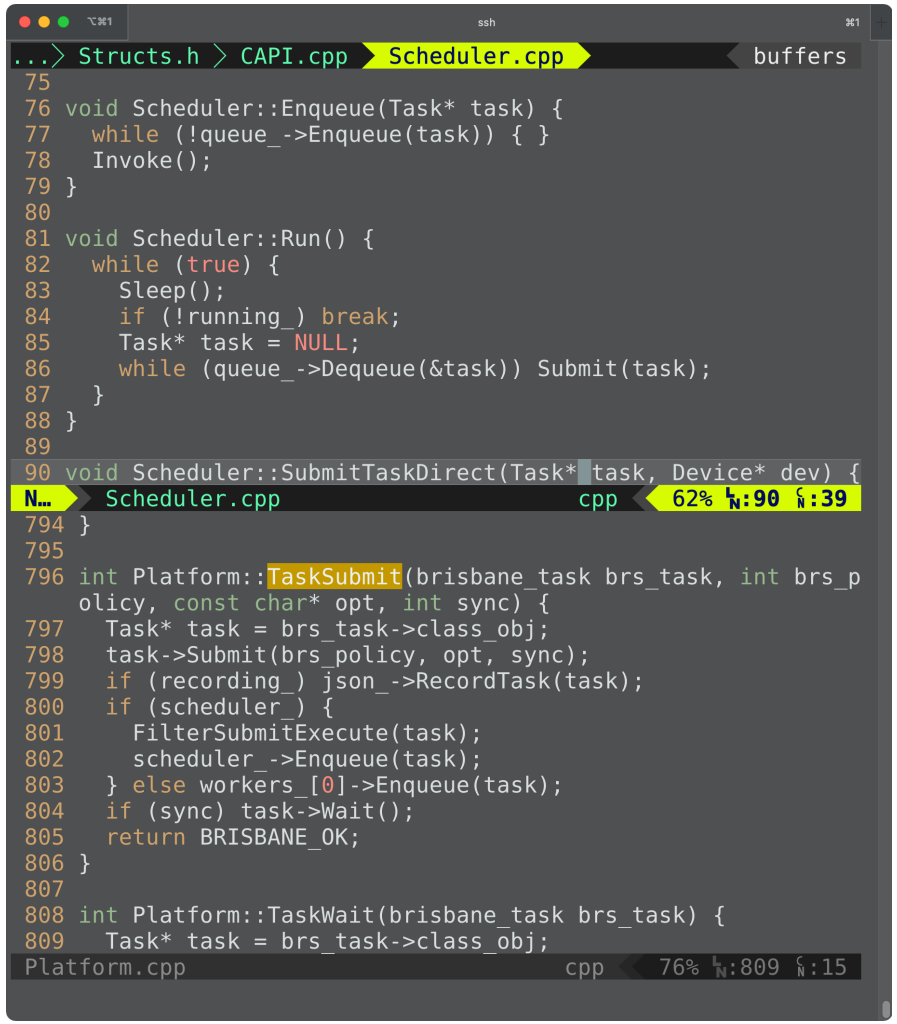
N… Scheduler.cpp cpp 62% ㏑:90 ㏇:39

```
794 }
795
796 int Platform::TaskSubmit(brisbane_task brs_task, int brs_p
    olicy, const char* opt, int sync) {
797   Task* task = brs_task->class_obj;
798   task->Submit(brs_policy, opt, sync);
799   if (recording_) json_->RecordTask(task);
800   if (scheduler_) {
801     FilterSubmitExecute(task);
802     scheduler_->Enqueue(task);
803   } else workers_[0]->Enqueue(task);
804   if (sync) task->Wait();
805   return BRISBANE_OK;
806 }
807
808 int Platform::TaskWait(brisbane_task brs_task) {
809   Task* task = brs_task->class_obj;
```

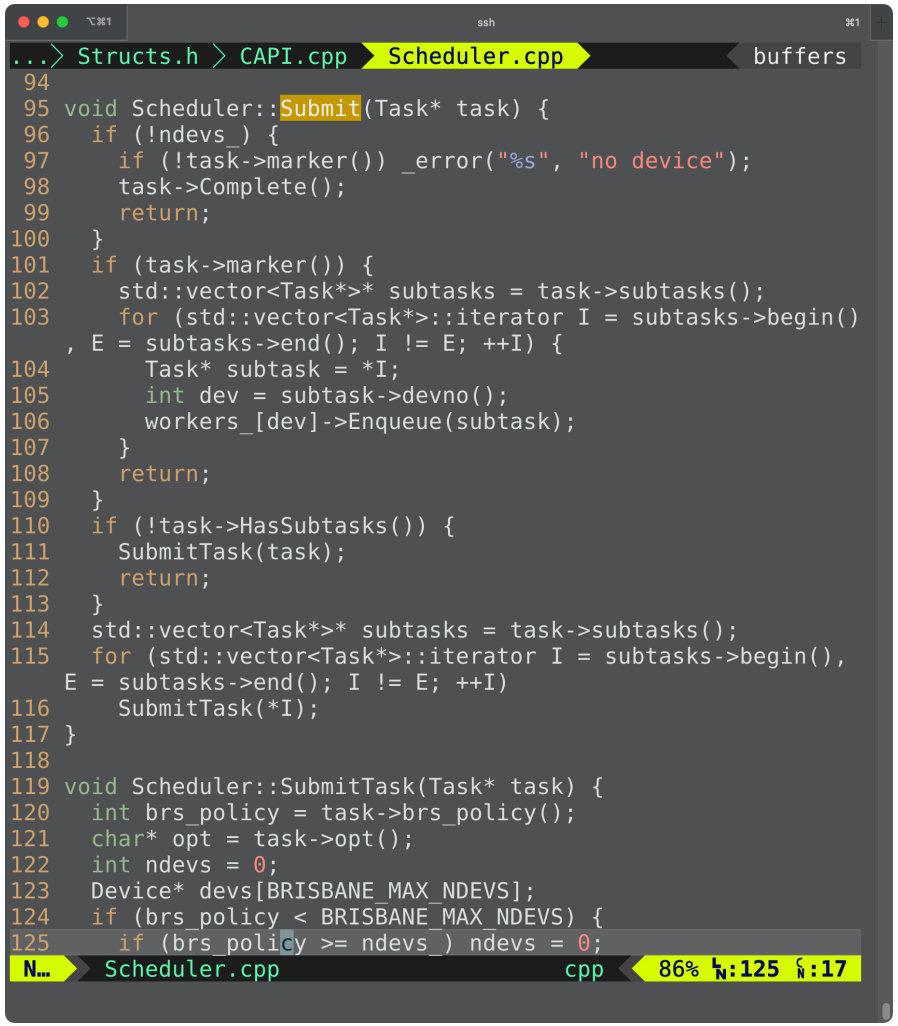
Platform.cpp cpp 76% ㏑:809 ㏇:15

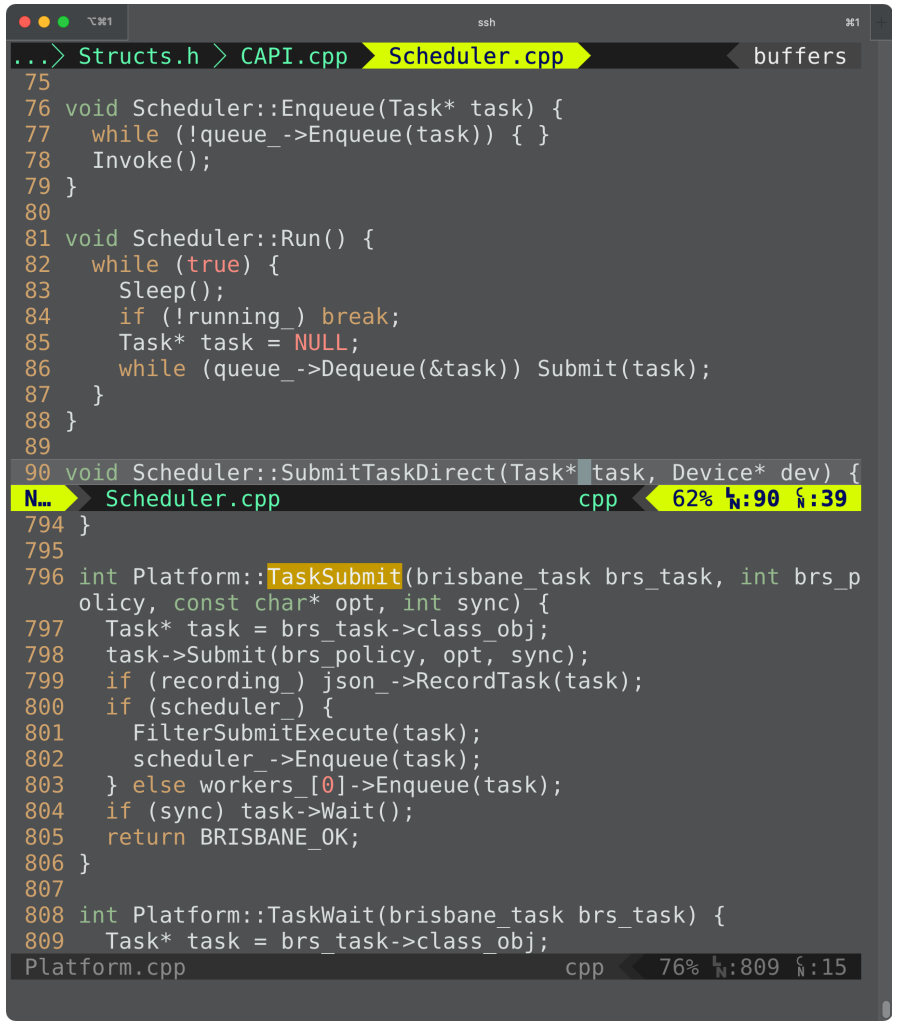
src/runtime/QueueTask.cpp



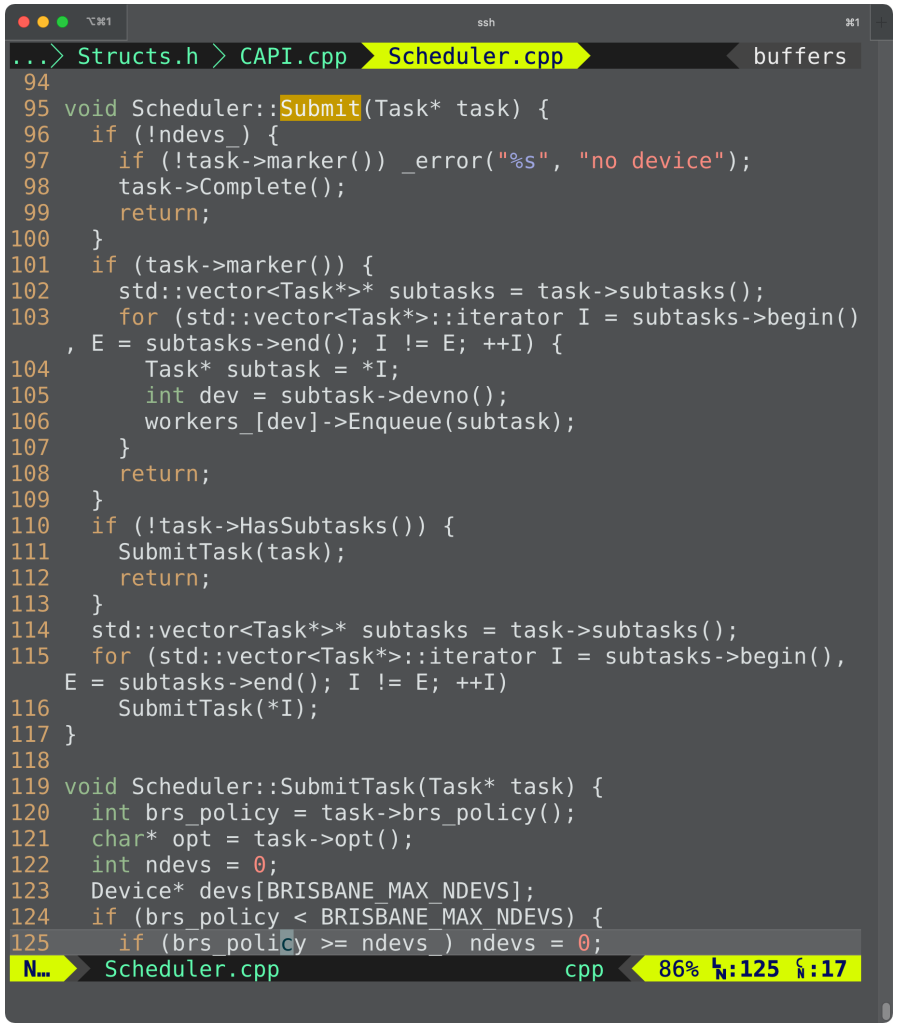


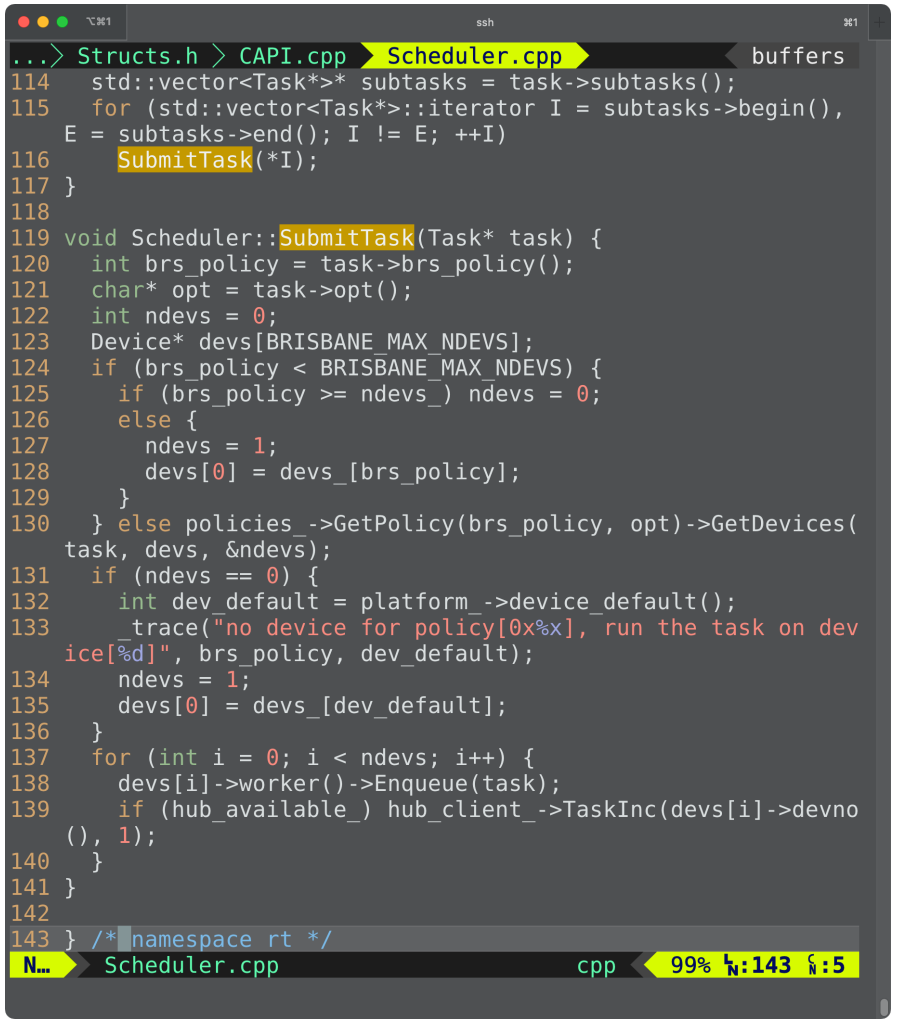
src/runtime/Scheduler.cpp





src/runtime/Scheduler.cpp





src/runtime/Policies.cpp

Policies.cpp buffers

```
 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   policy_random_       = new PolicyRandom(scheduler_);
28   policy_roundrobin_   = new PolicyRoundRobin(scheduler_);
29 }
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_;
```

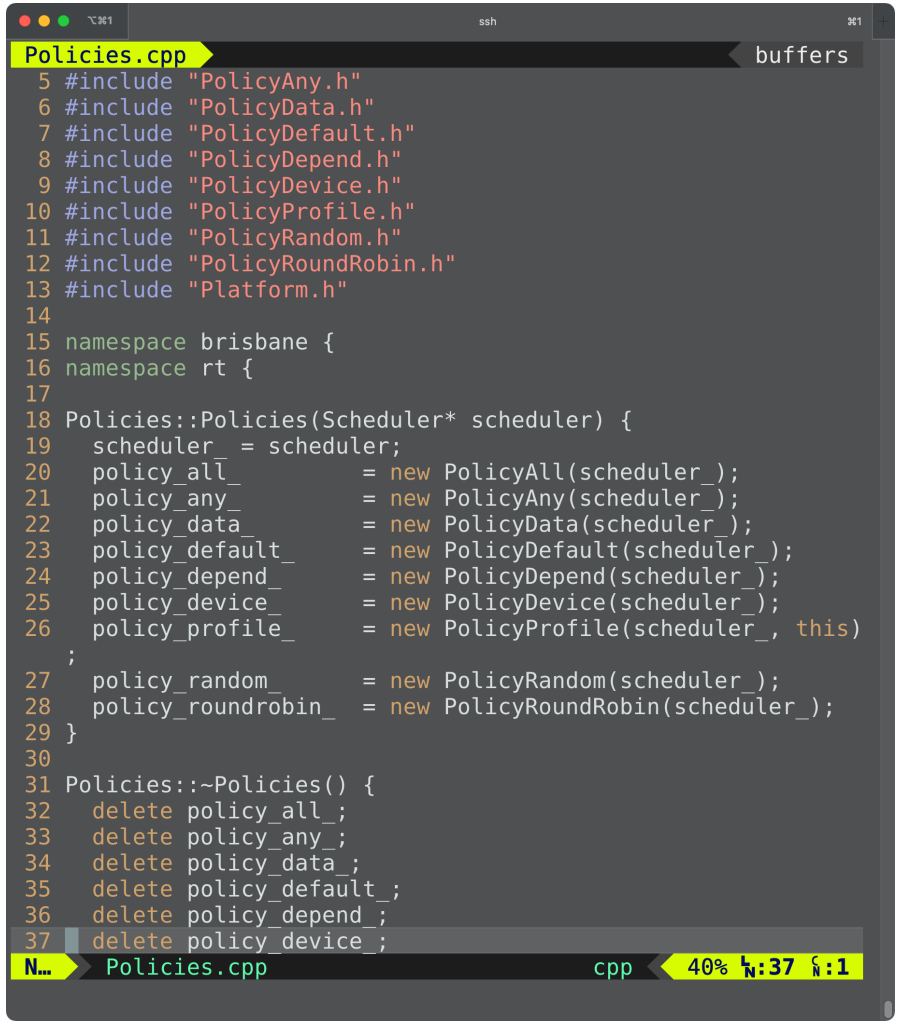
N… Policies.cpp cpp 40% ℓ:37 c:1

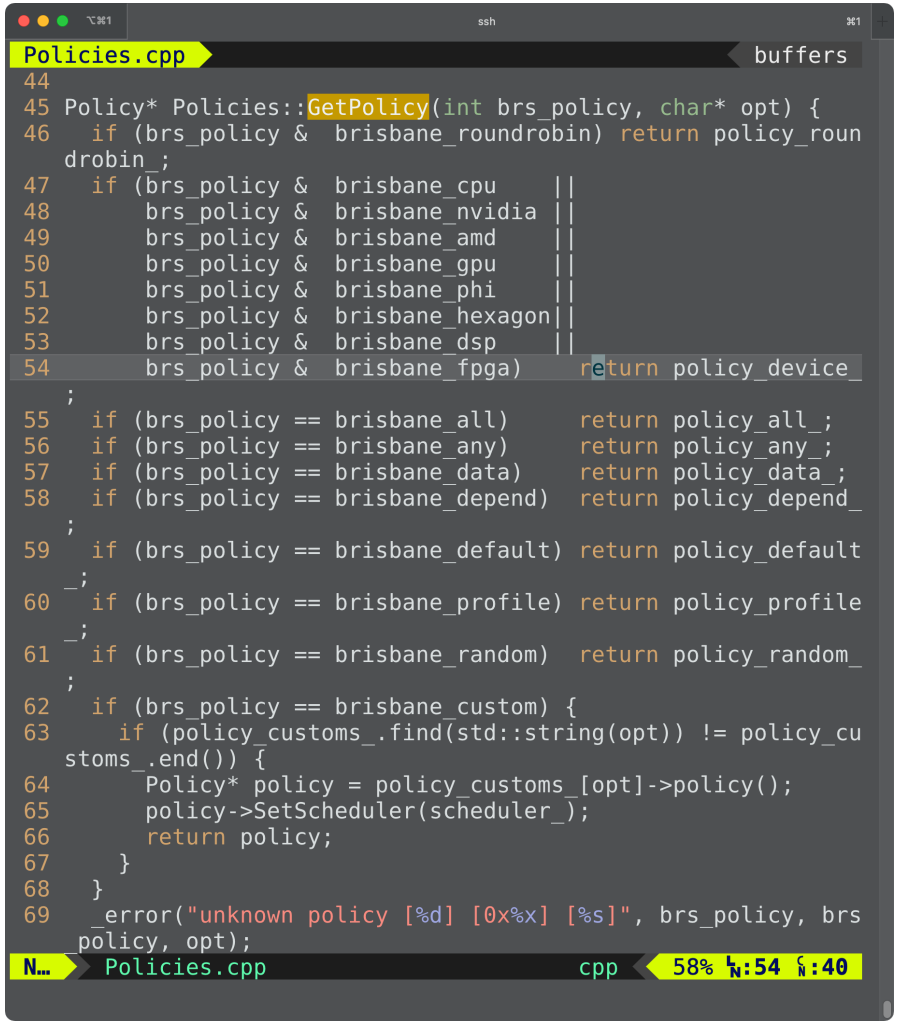
...> Structs.h > CAPI.cpp > Scheduler.cpp buffers

```
114   std::vector<Task*>* subtasks = task->subtasks();
115   for (std::vector<Task*>::iterator I = subtasks->begin(),
   E = subtasks->end(); I != E; ++I)
116     SubmitTask(*I);
117 }
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(
   task, devs, &ndevs);
131   if (ndevs == 0) {
132     int dev_default = platform_->device_default();
133     _trace("no device for policy[0x%x], run the task on dev
   ice[%d]", brs_policy, dev_default);
134     ndevs = 1;
135     devs[0] = devs_[dev_default];
136   }
137   for (int i = 0; i < ndevs; i++) {
138     devs[i]->worker()->Enqueue(task);
139     if (hub_available_) hub_client_->TaskInc(devs[i]->devno
   (), 1);
140   }
141 }
142
143 } /* namespace rt */
```

N… Scheduler.cpp cpp 99% ℓ:143 c:5

src/runtime/Policies.cpp





src/runtime/PolicyDevice.cpp

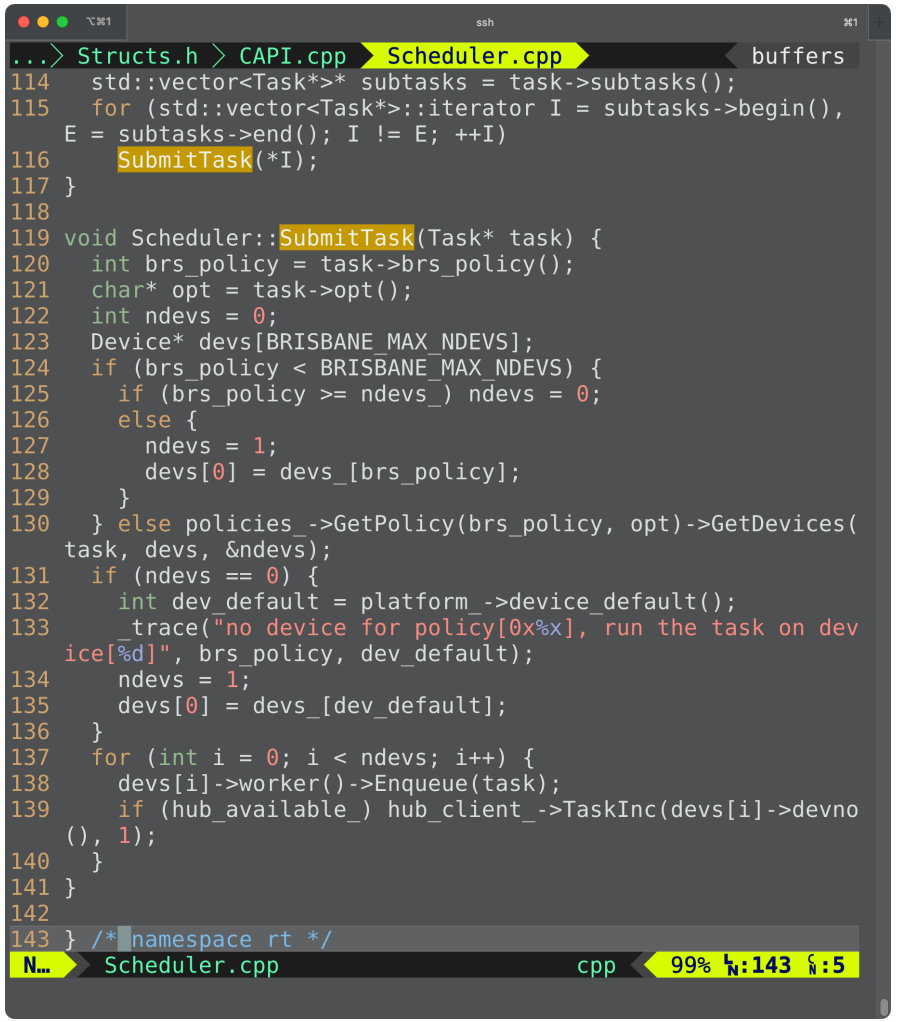
PolicyDevice.cpp buffers

```
 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, in
   t* 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% ㏑:29 ㏇:1
"PolicyDevice.cpp" 29L, 576C
```

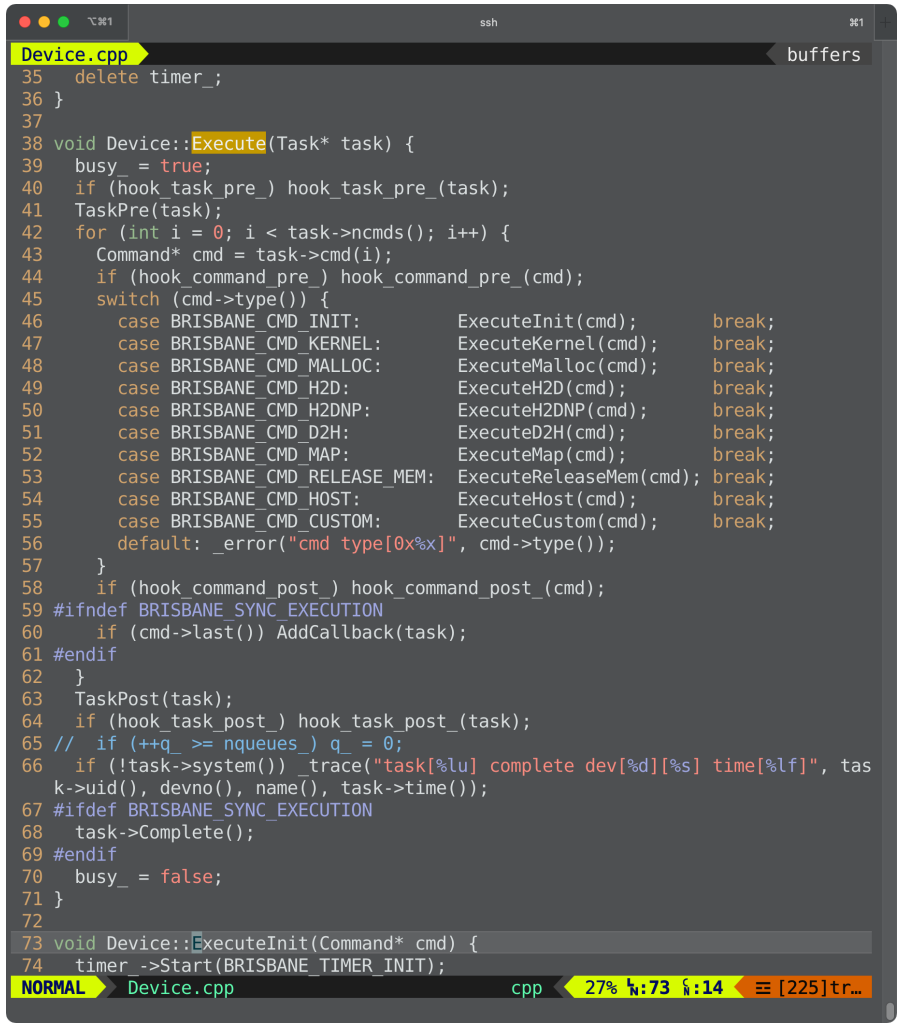
Policies.cpp buffers

```
44
45 Policy* Policies::GetPolicy(int brs_policy, char* opt) {
46   if (brs_policy &  brisbane_roundrobin) return policy_roun
   drobin_;
47   if (brs_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 (brs_policy == brisbane_all)     return policy_all_;
56   if (brs_policy == brisbane_any)     return policy_any_;
57   if (brs_policy == brisbane_data)    return policy_data_;
58   if (brs_policy == brisbane_depend)  return policy_depend_
   ;
59   if (brs_policy == brisbane_default) return policy_default
   _;
60   if (brs_policy == brisbane_profile) return policy_profile
   _;
61   if (brs_policy == brisbane_random)  return policy_random_
   ;
62   if (brs_policy == brisbane_custom) {
63     if (policy_customs_.find(std::string(opt)) != policy_cu
   stoms_.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% ㏑:54 ㏇:40
```

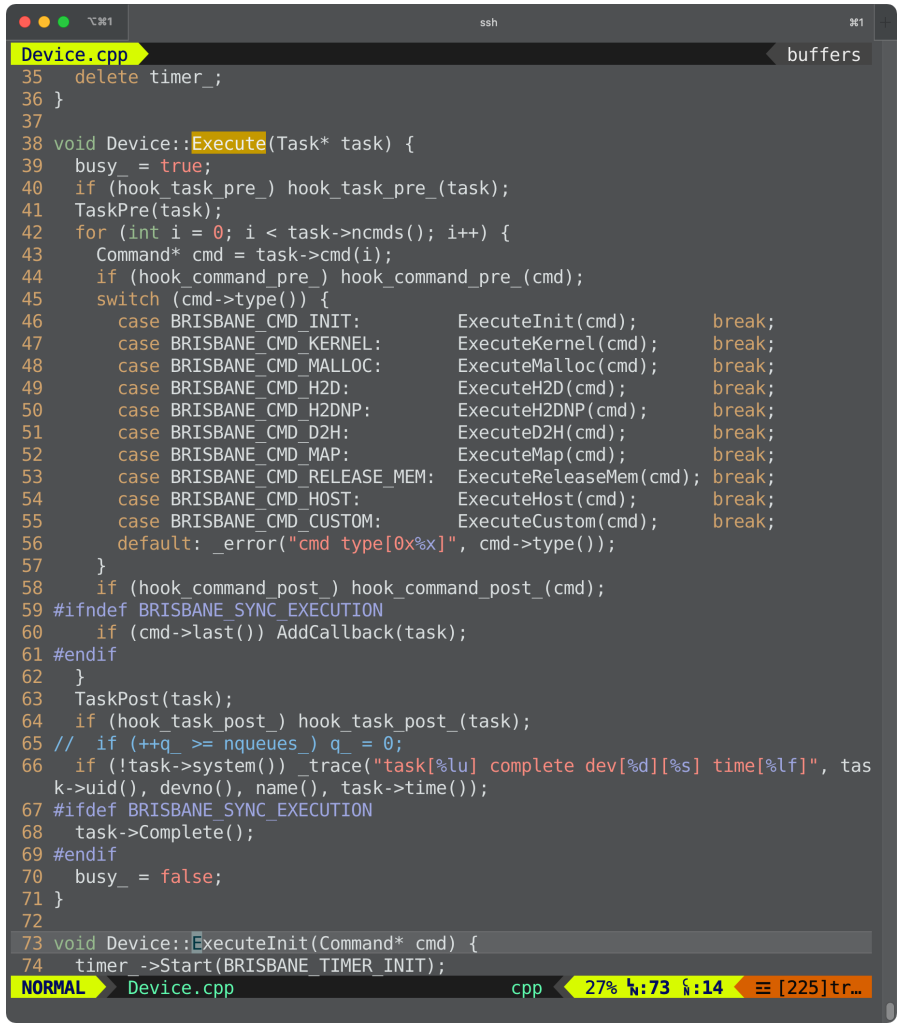
src/runtime/PolicyDevice.cpp

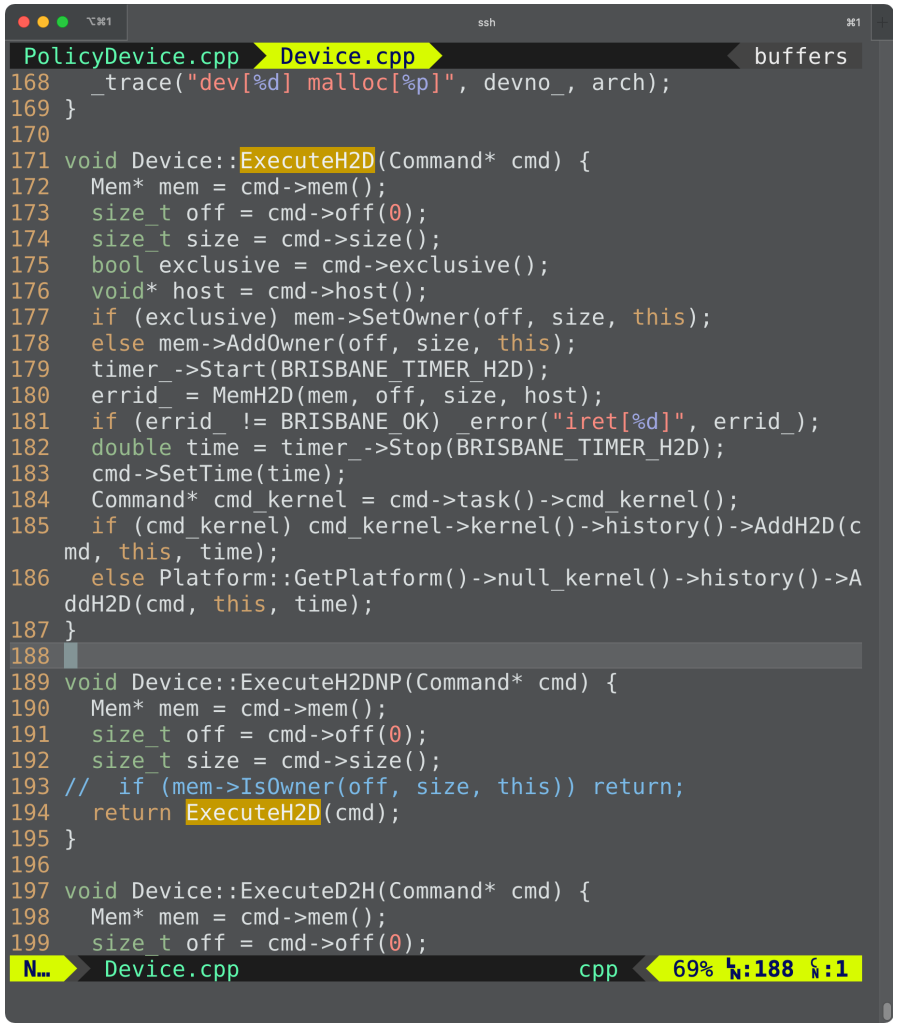


src/runtime/Device.cpp



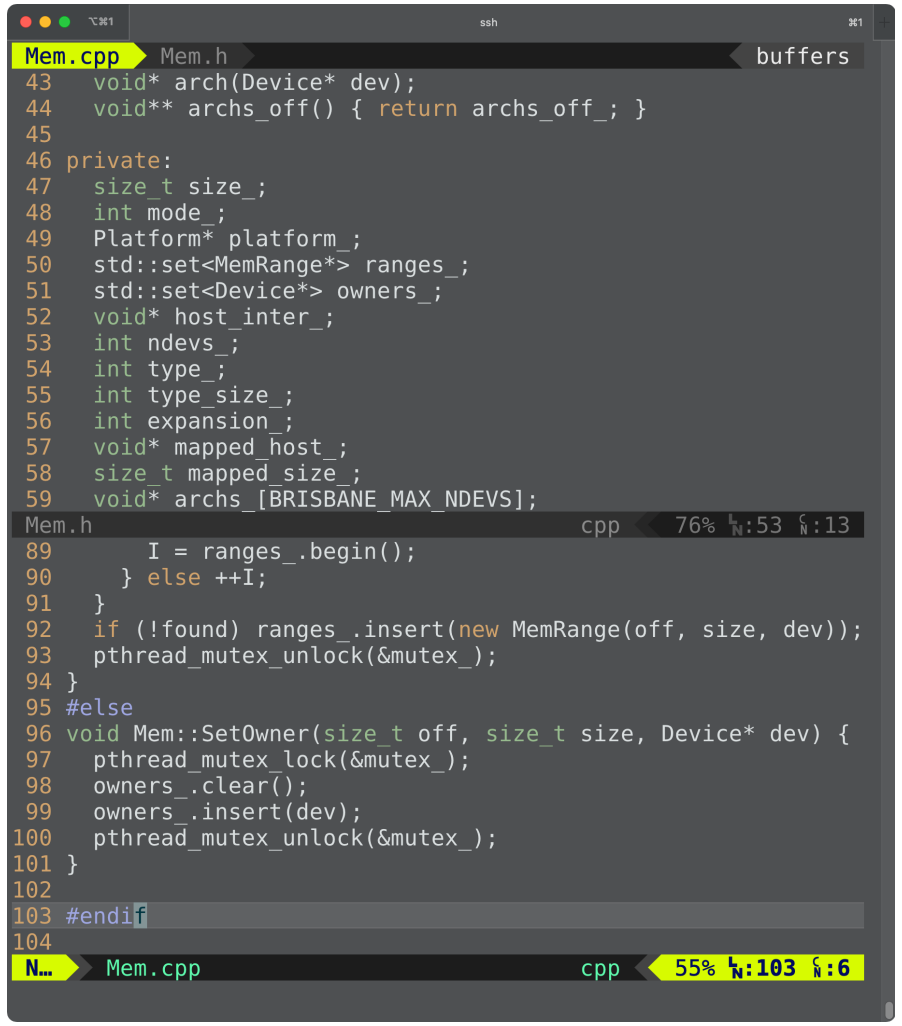
src/runtime/Device.cpp





Owner List of IRIS Memory

- An IRIS memory has an owner list
- An owner list can have zero or more owners
  - 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 in its device memory

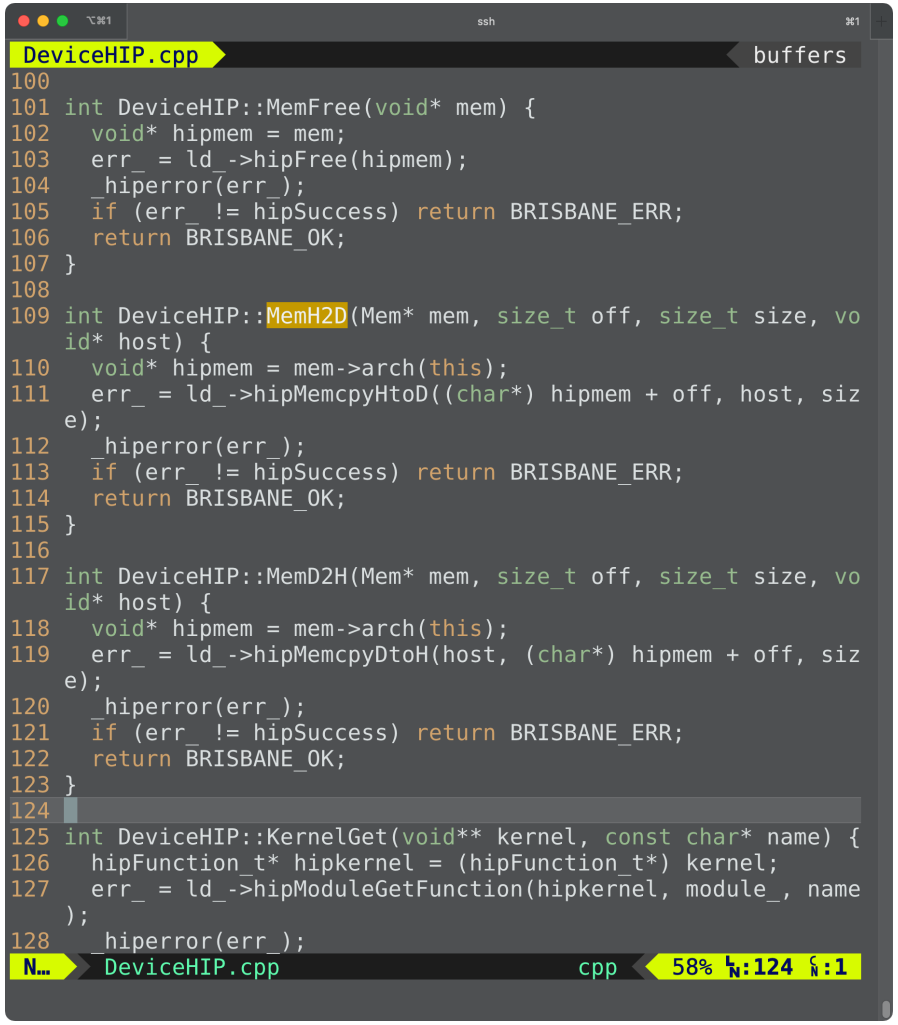


src/runtime/DeviceHIP.cpp

```
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, vo
    id* host) {
110   void* hipmem = mem->arch(this);
111   err_ = ld_->hipMemcpyHtoD((char*) hipmem + off, host, siz
    e);
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, vo
    id* host) {
118   void* hipmem = mem->arch(this);
119   err_ = ld_->hipMemcpyDtoH(host, (char*) hipmem + off, siz
    e);
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% ㏑:124 ㏇: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(c
    md, this, time);
186   else Platform::GetPlatform()->null_kernel()->history()->A
    ddH2D(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% ㏑:188 ㏇:1
```

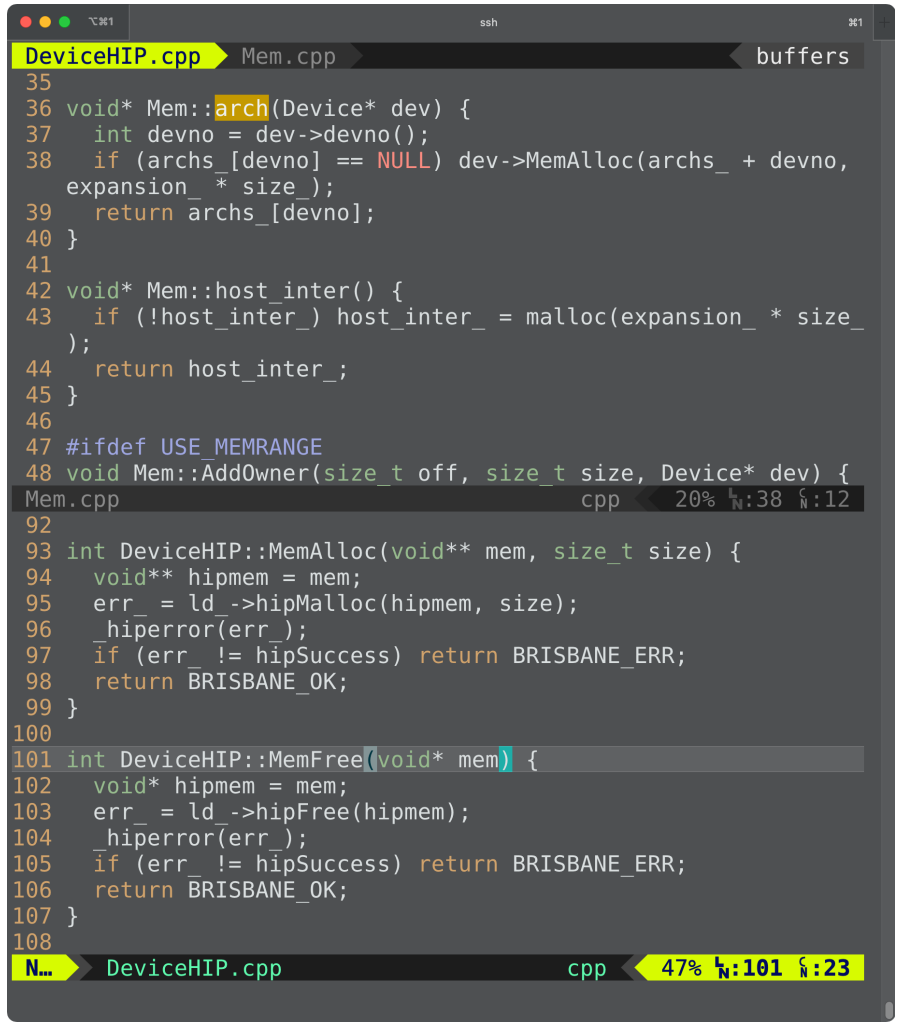
src/runtime/DeviceHIP.cpp



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]



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]

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, vo
    id* host) {
110   void* hipmem = mem->arch(this);
111   err_ = ld_->hipMemcpyHtoD((char*) hipmem + off, host, siz
    e);
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, vo
    id* host) {
118   void* hipmem = mem->arch(this);
119   err_ = ld_->hipMemcpyDtoH(host, (char*) hipmem + off, siz
    e);
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_);
```

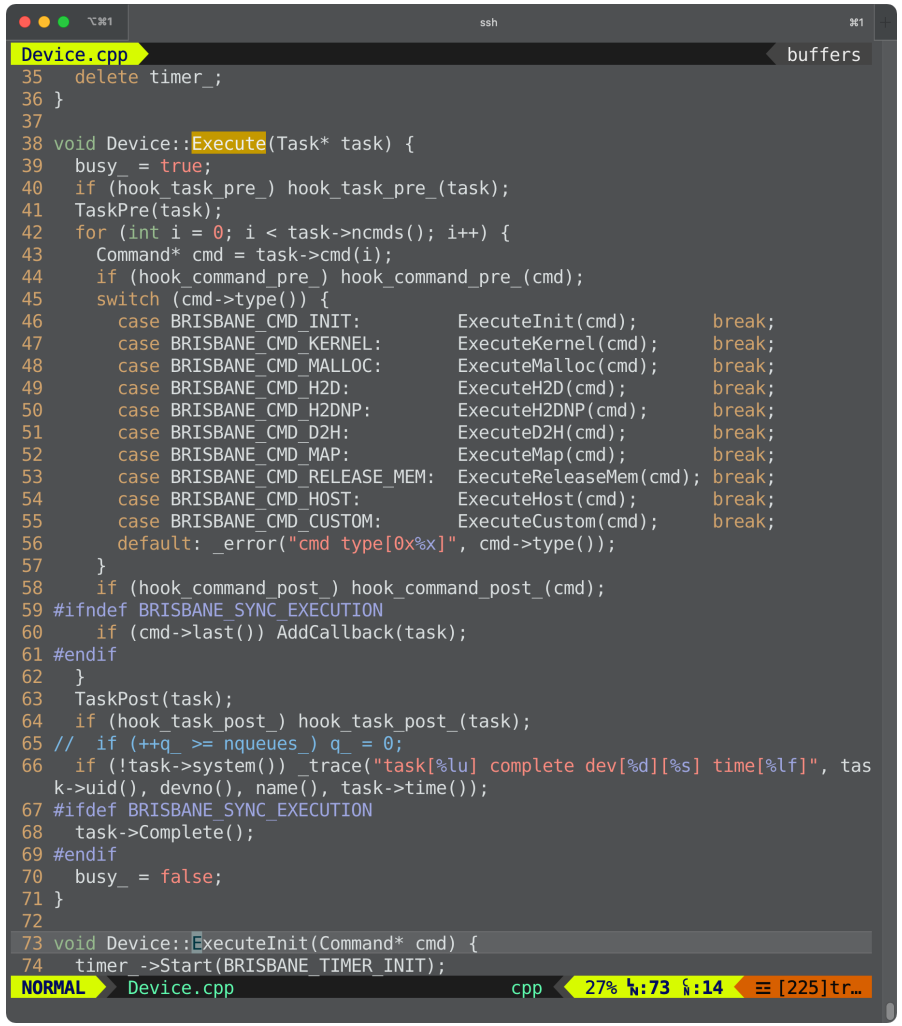
N… DeviceHIP.cpp cpp 58% ㏑:124 ☰:1

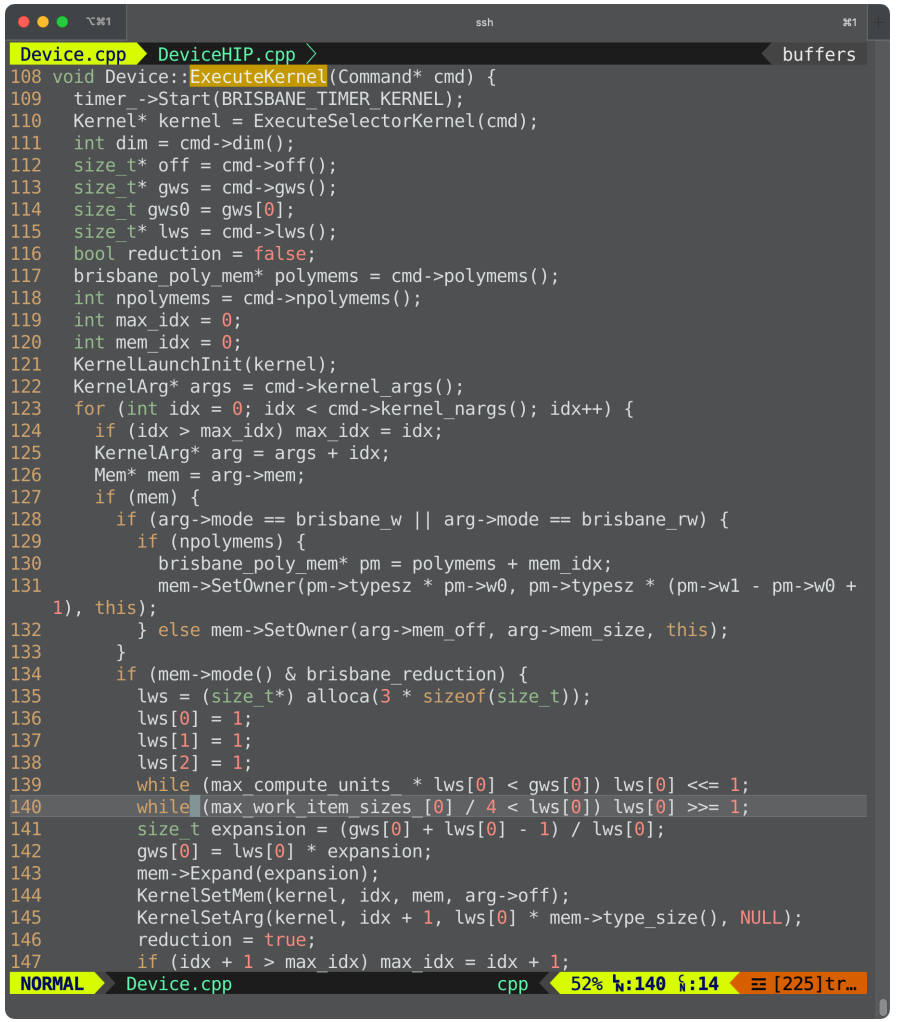
PolicyDevice.cpp Device.cpp buffers

```
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(c
    md, this, time);
186   else Platform::GetPlatform()->null_kernel()->history()->A
    ddH2D(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);
```

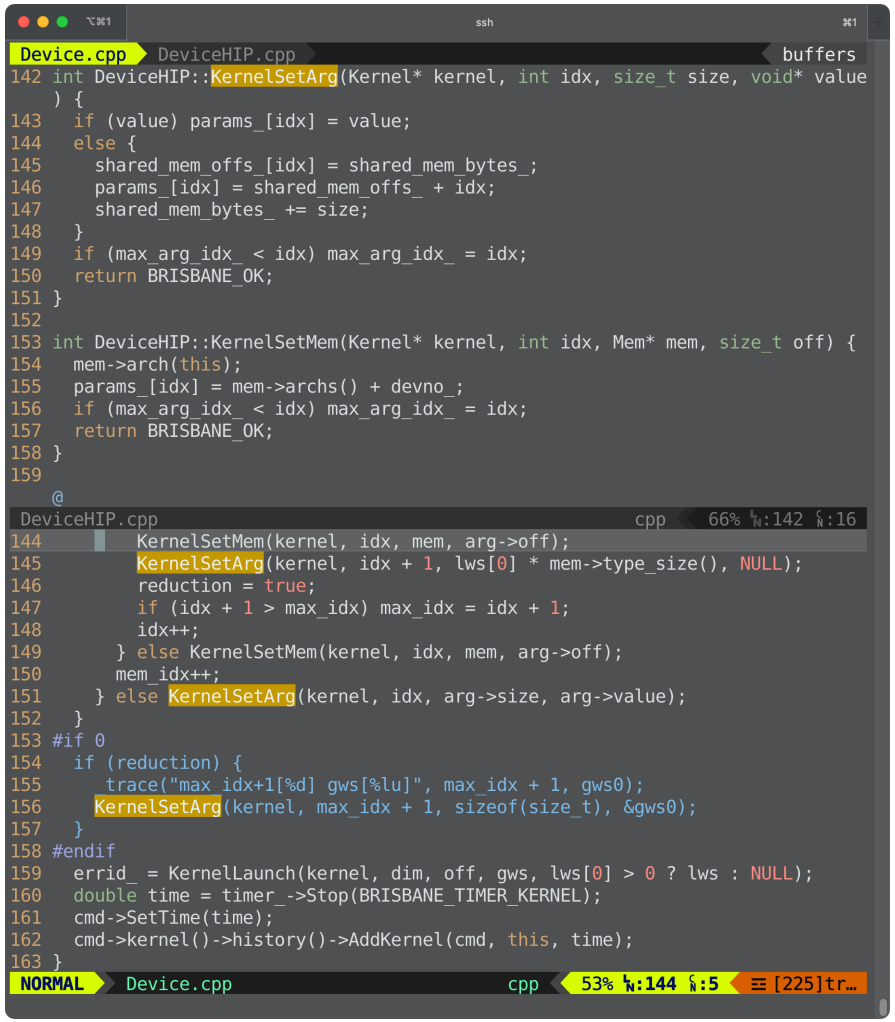
N… Device.cpp cpp 69% ㏑:188 ☰:1

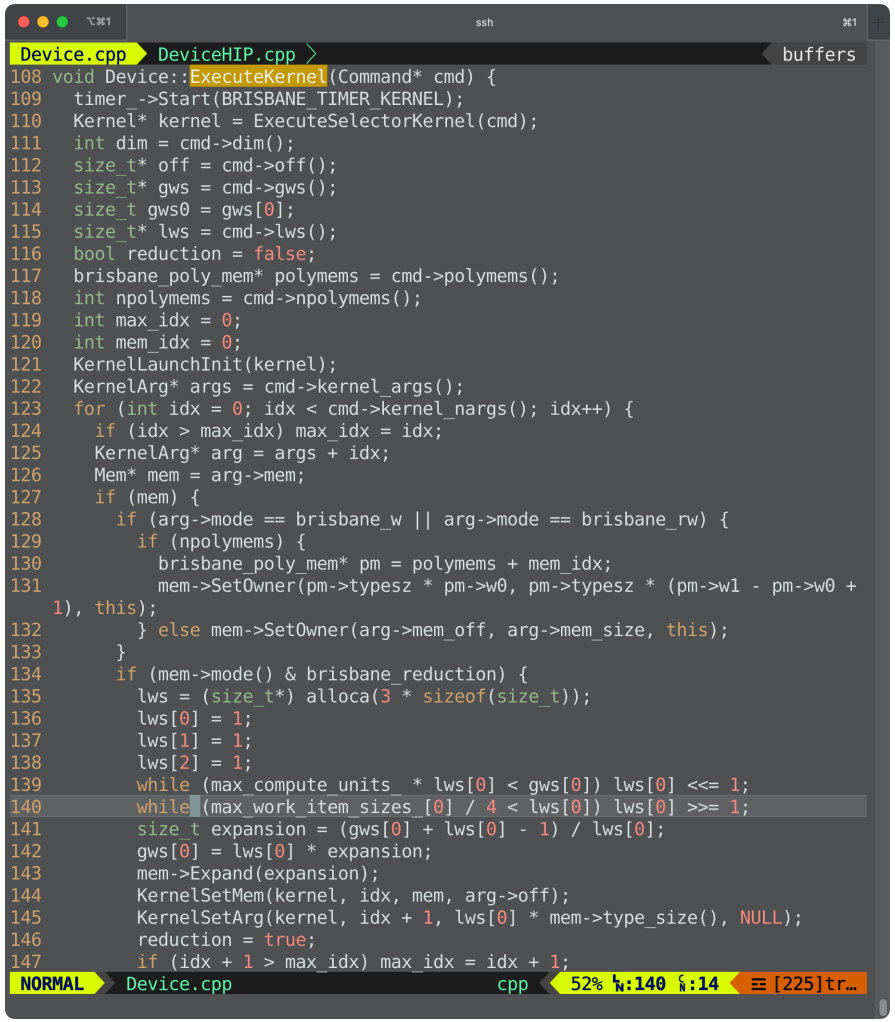
src/runtime/Device.cpp



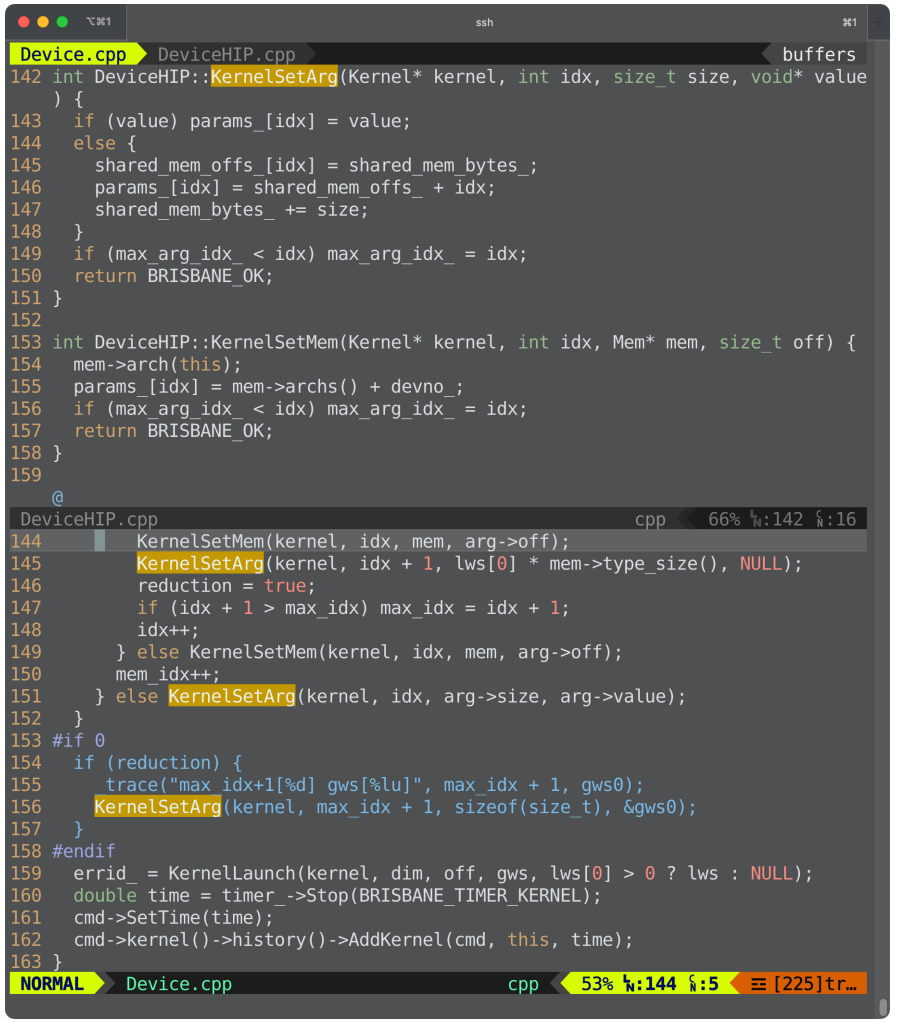


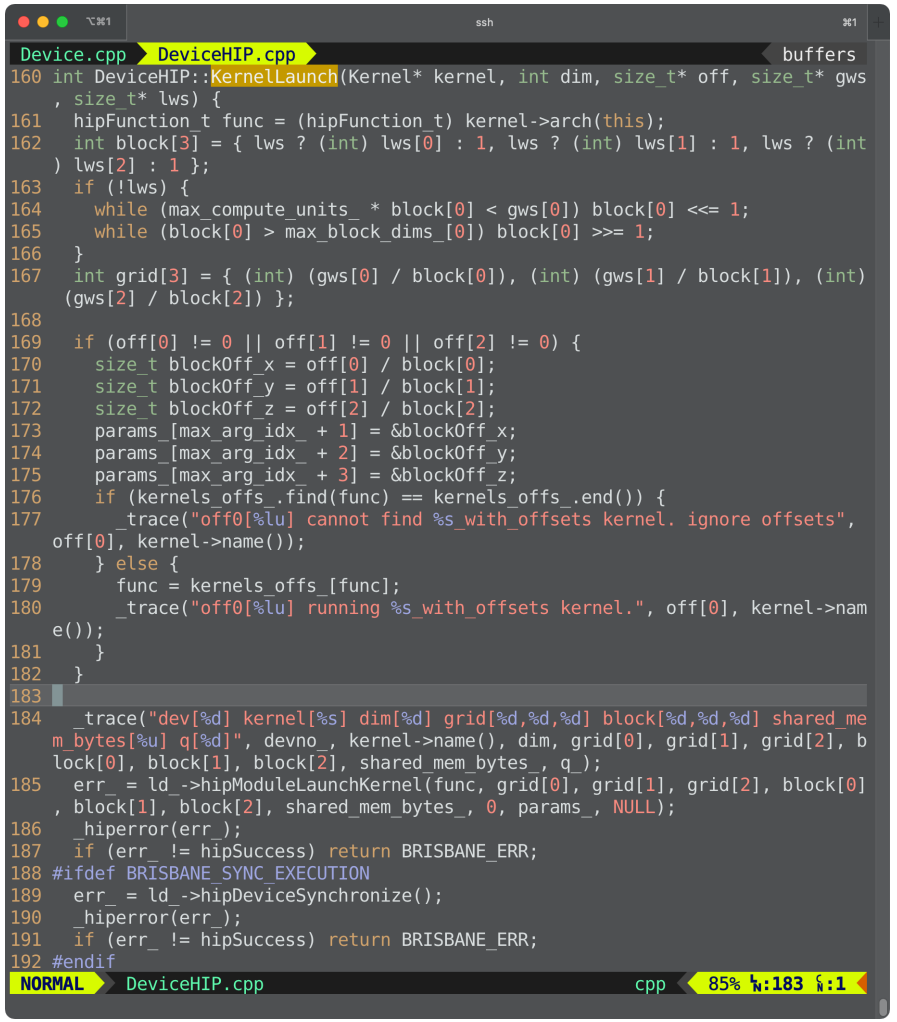
src/runtime/DeviceHIP.cpp





src/runtime/DeviceHIP.cpp

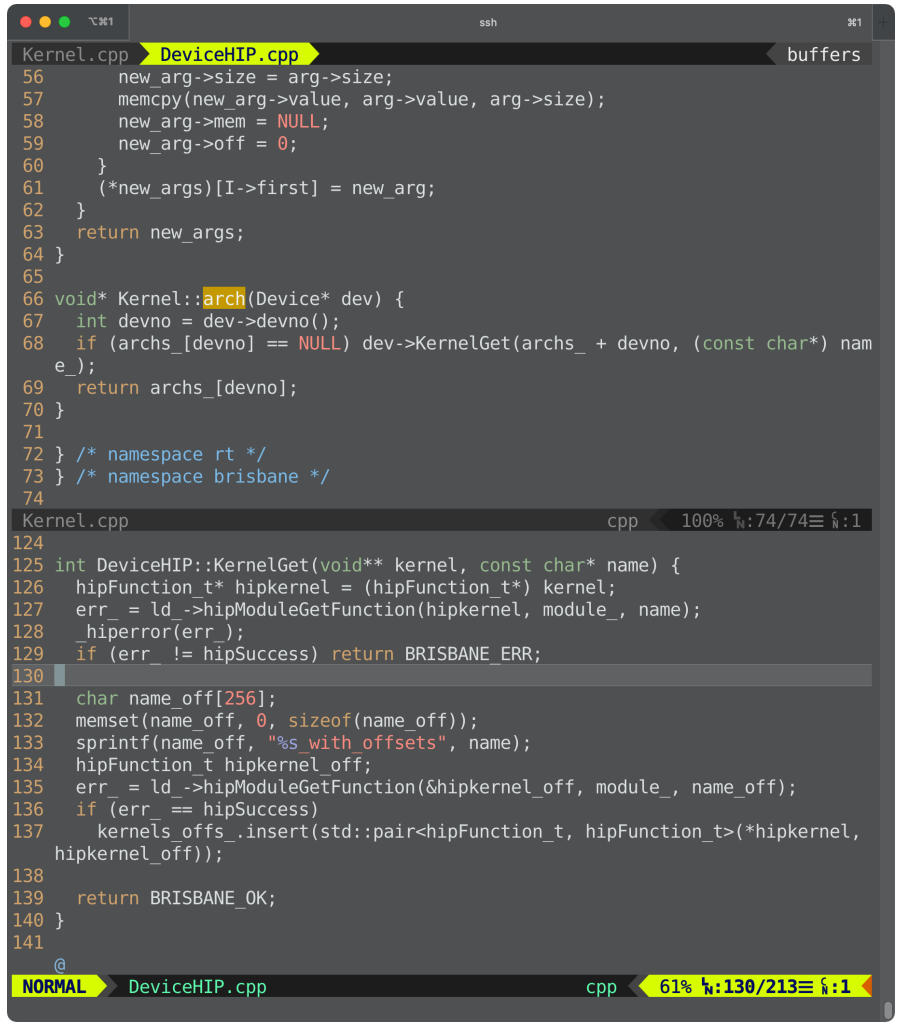




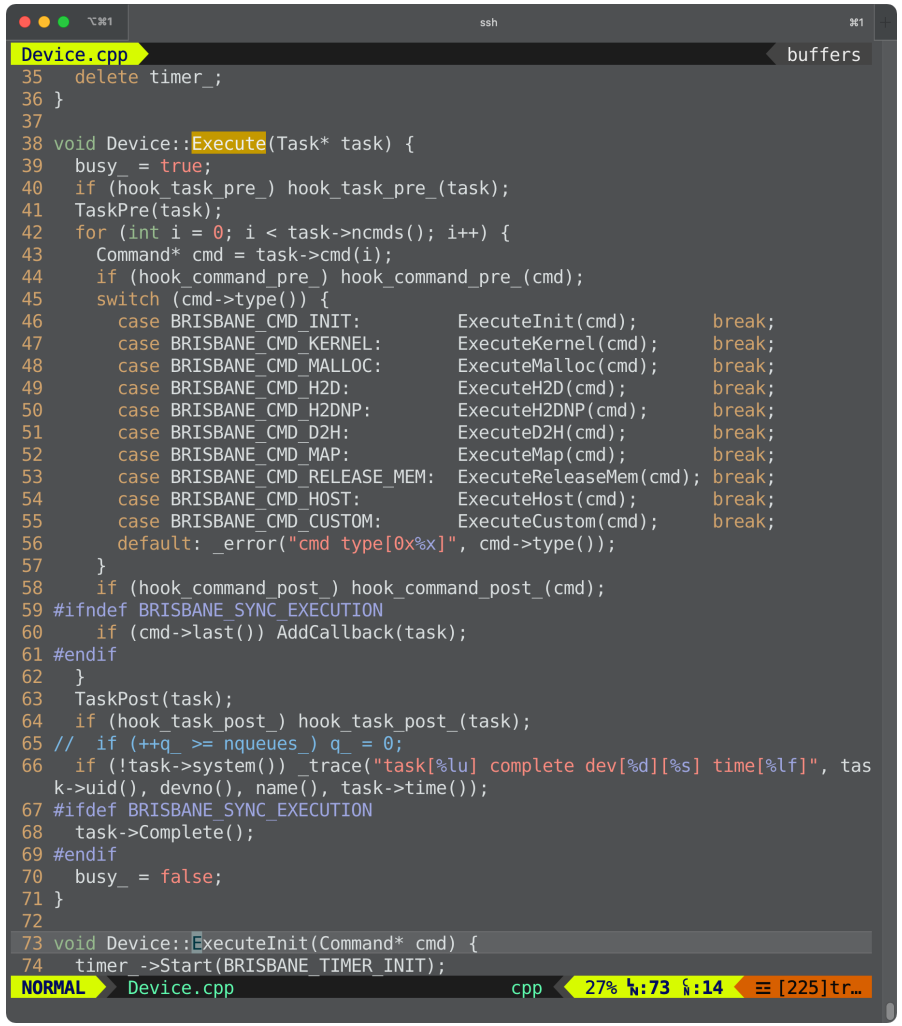
IRIS Kernel Arch Table

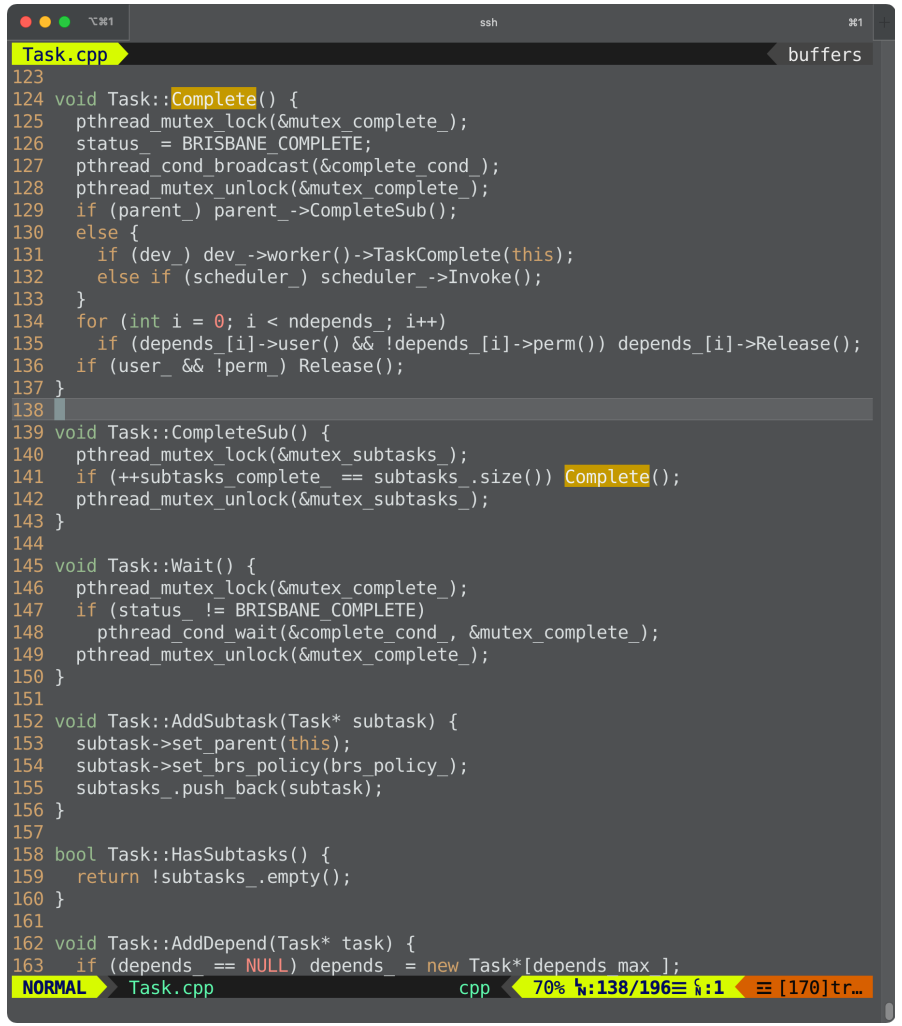
Kernel

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



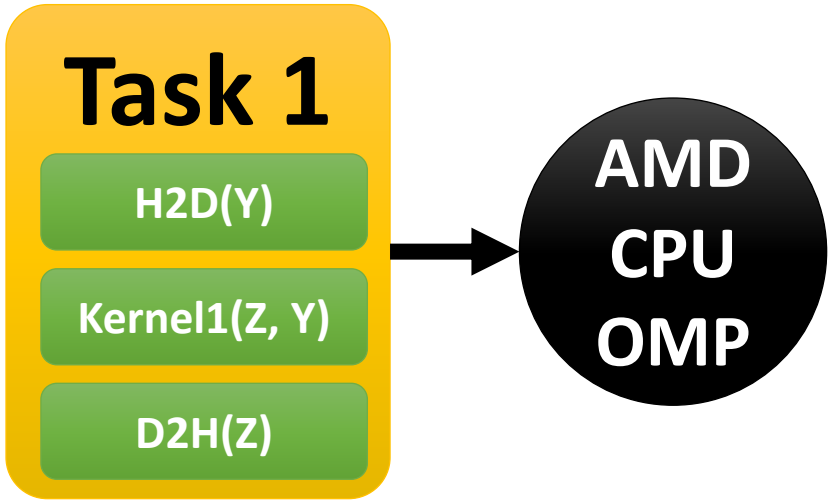
src/runtime/Device.cpp





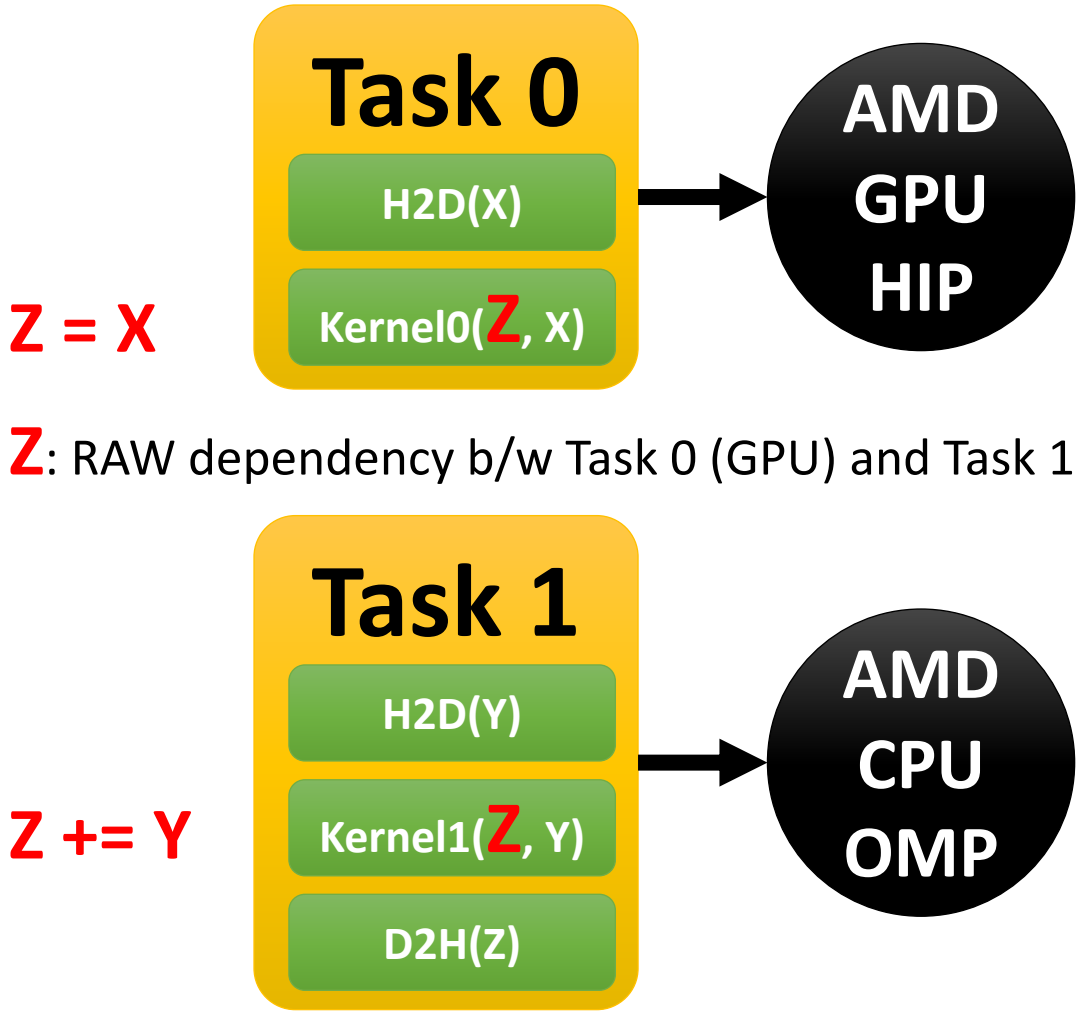
iris_task_submit()

```
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_par
   ams_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_par
   ams_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
```

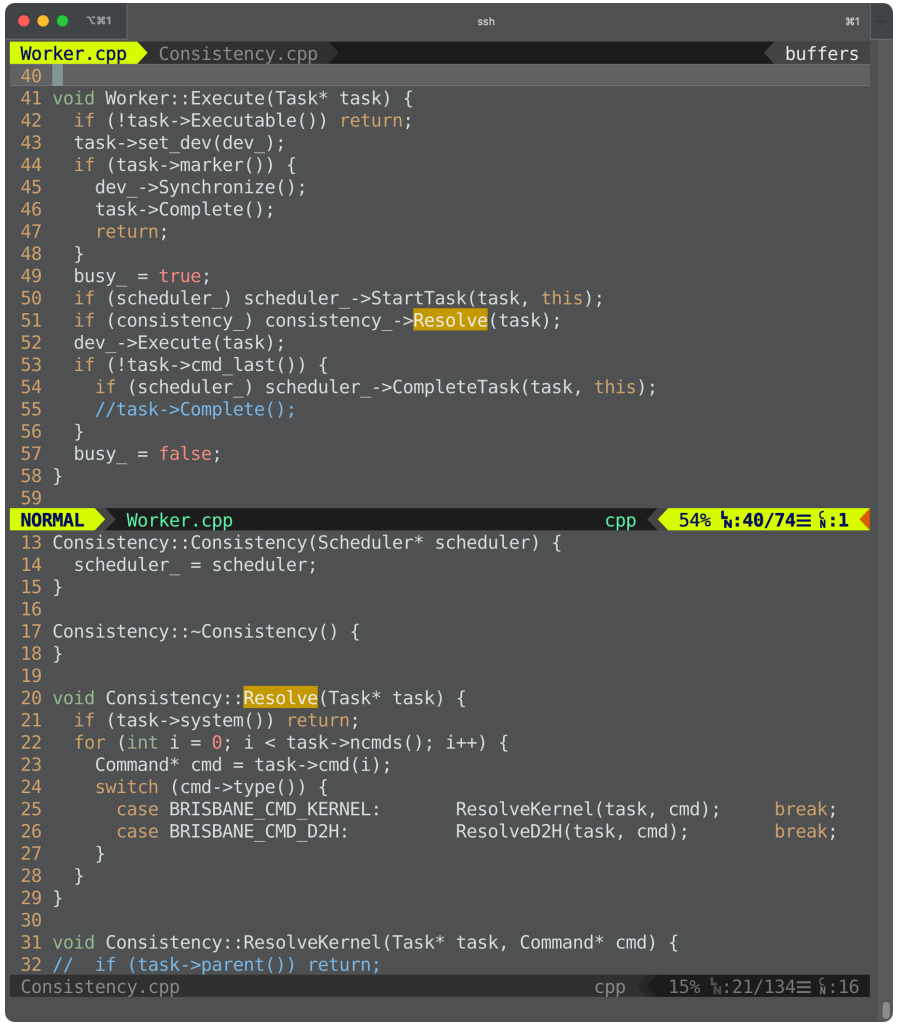


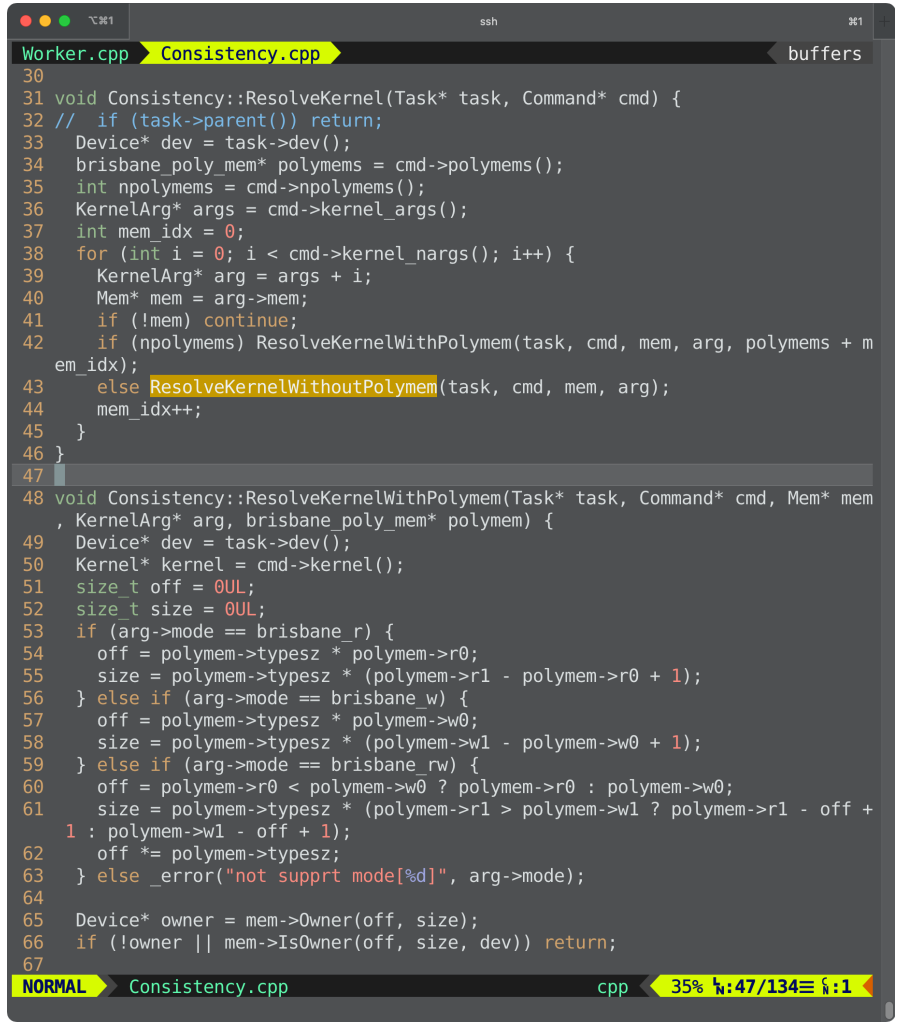
src/runtime/Consistency.cpp

```
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% ℕ:40/74≡ §: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% ℕ:21/134≡ §:16
```

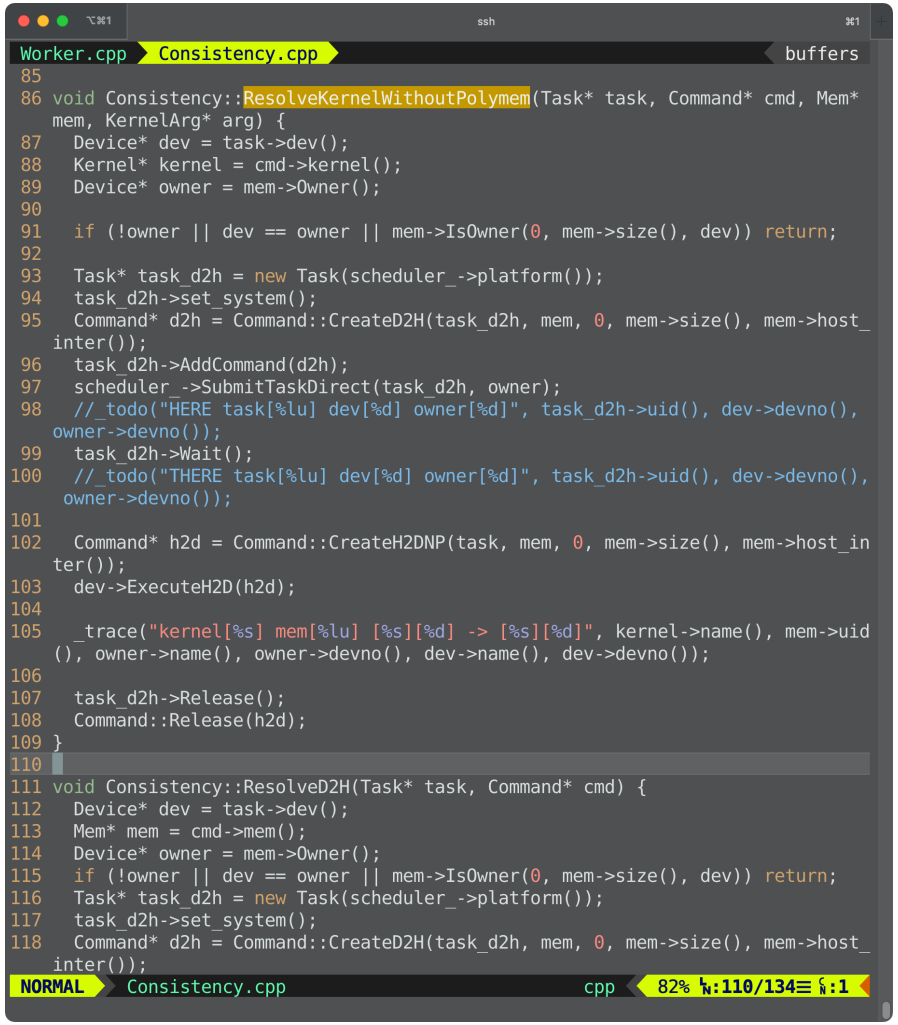


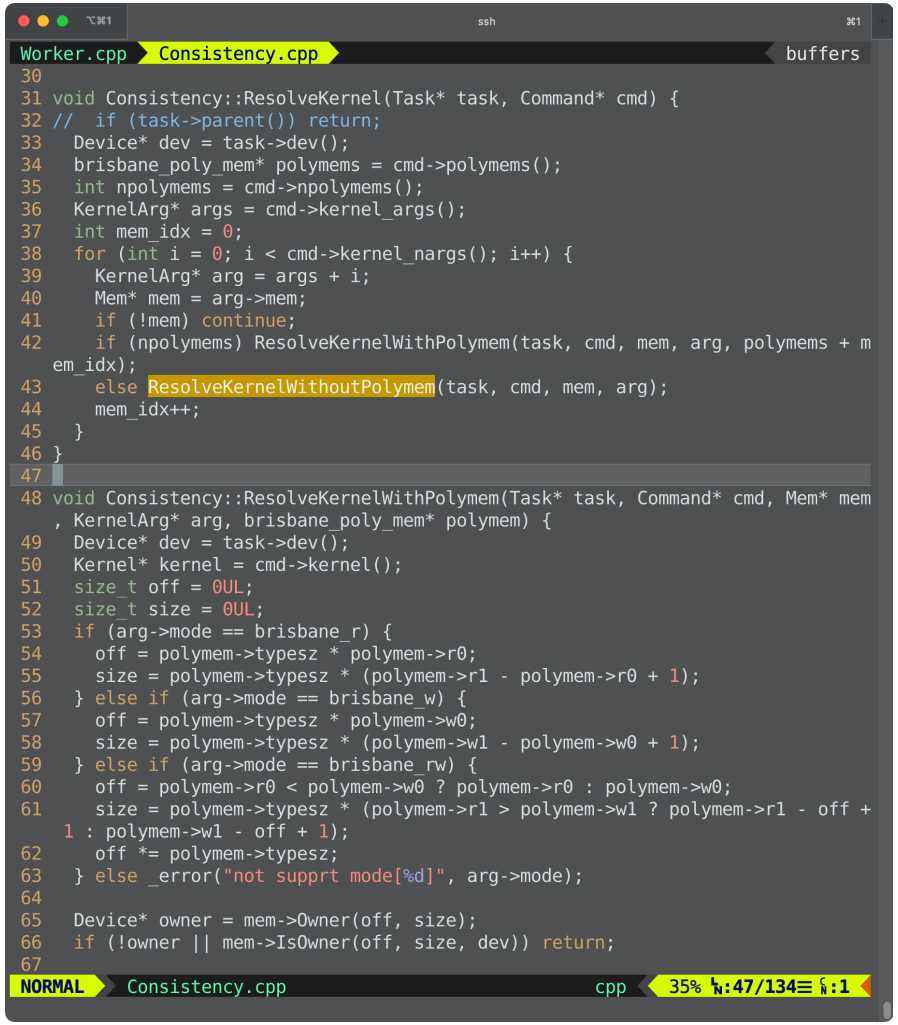
src/runtime/Consistency.cpp





src/runtime/Consistency.cpp





iris_task_finalize()

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_par
   ams_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_par
   ams_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
   ", time_app_, time_init_, time_app_ - time_init_);
1049   _info("t10[%lf] t11[%lf] t12[%lf] t13[%lf]", timer()->Total(10), timer()->
   Total(11), timer()->Total(12), timer()->Total(13));
1050   _info("t14[%lf] t15[%lf] t16[%lf] t17[%lf]", timer()->Total(14), timer()->
   Total(15), timer()->Total(16), timer()->Total(17));
1051   _info("t18[%lf] t19[%lf] t20[%lf] t21[%lf]", timer()->Total(18), timer()->
   Total(19), timer()->Total(20), timer()->Total(21));
1052   finalize_ = true;
1053   pthread_mutex_unlock(&mutex_);
1054   return ret_id;
1055 }
1056
1057 } /* namespace rt */
1058 } /* namespace brisbane */
1059
```

NORMAL Platform.cpp cpp 99% ㏑:1056/1059☰ ㏇:1 ☰[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_par
   ams_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_par
   ams_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 }
```

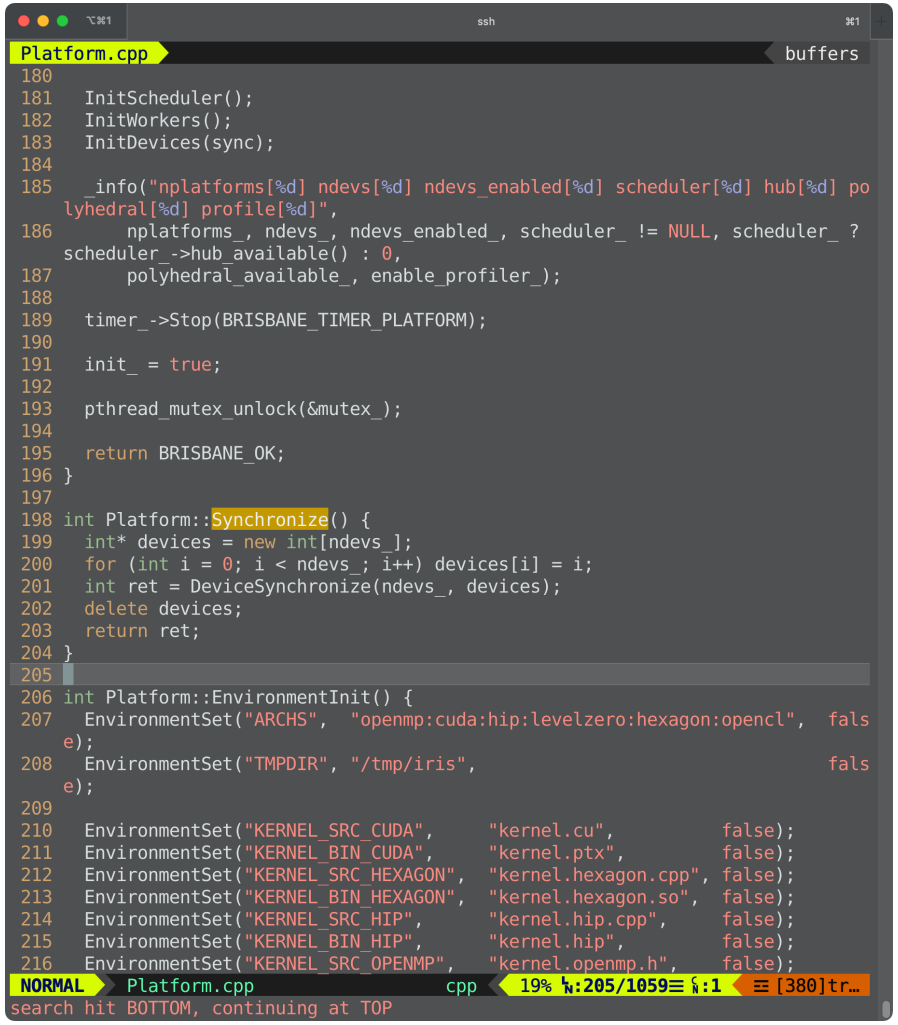
NORMAL 2tasks.c c utf-8[unix] 98% ㏑:64/65≡ ㏇:1

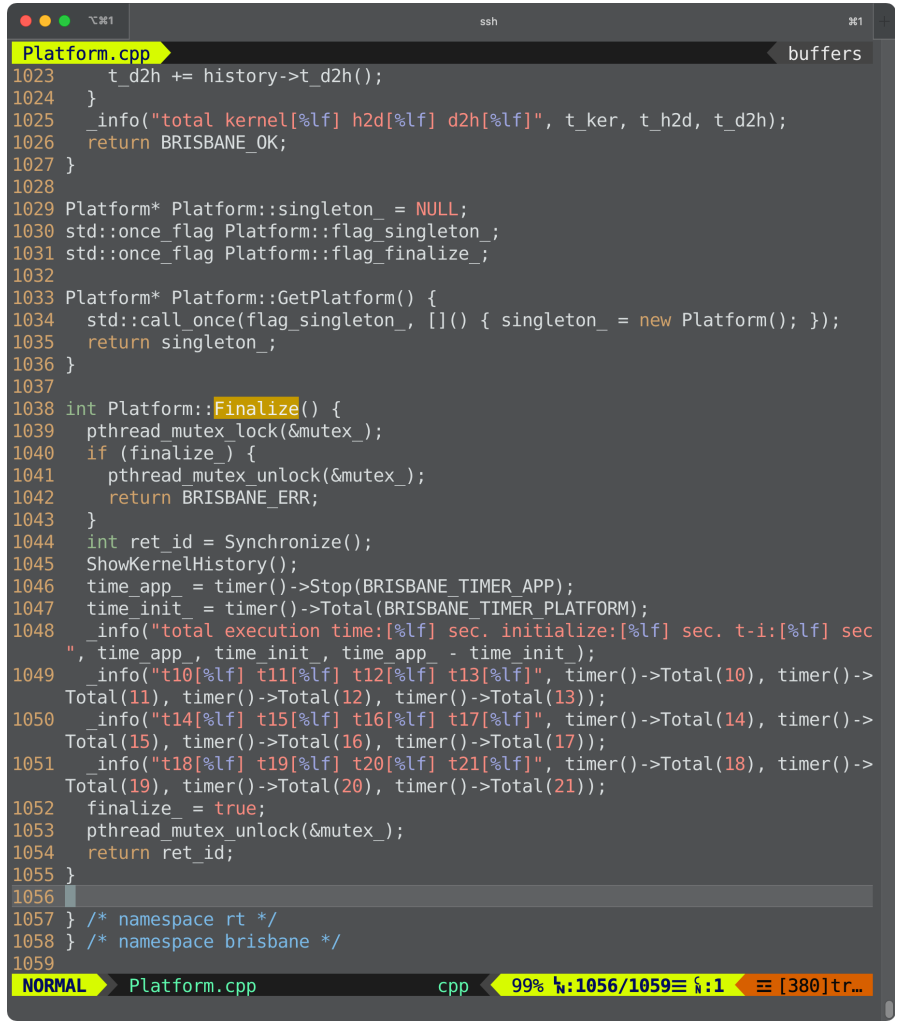
Platform.cpp — buffers

```
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
   ", time_app_, time_init_, time_app_ - time_init_);
1049   _info("t10[%lf] t11[%lf] t12[%lf] t13[%lf]", timer()->Total(10), timer()->
   Total(11), timer()->Total(12), timer()->Total(13));
1050   _info("t14[%lf] t15[%lf] t16[%lf] t17[%lf]", timer()->Total(14), timer()->
   Total(15), timer()->Total(16), timer()->Total(17));
1051   _info("t18[%lf] t19[%lf] t20[%lf] t21[%lf]", timer()->Total(18), timer()->
   Total(19), timer()->Total(20), timer()->Total(21));
1052   finalize_ = true;
1053   pthread_mutex_unlock(&mutex_);
1054   return ret_id;
1055 }
1056
1057 } /* namespace rt */
1058 } /* namespace brisbane */
1059
```

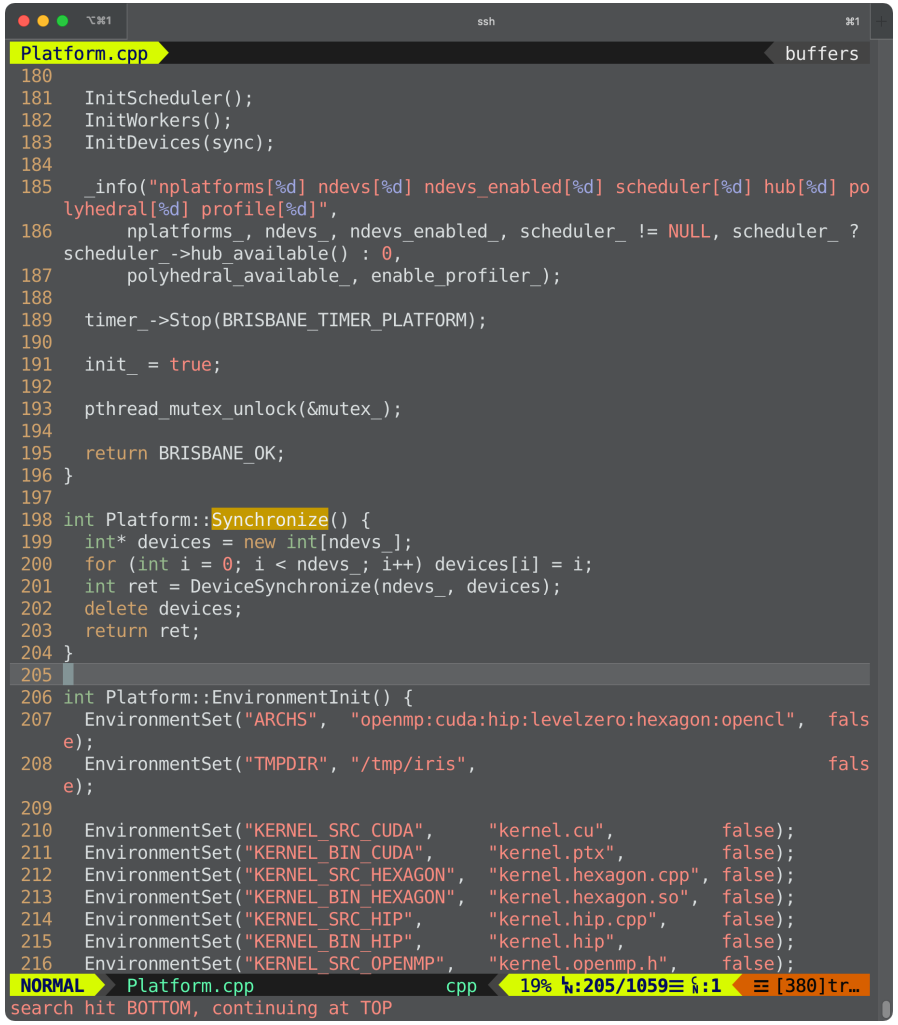
NORMAL Platform.cpp cpp 99% ㏑:1056/1059≡ ㏇:1 ≡ [380]tr…

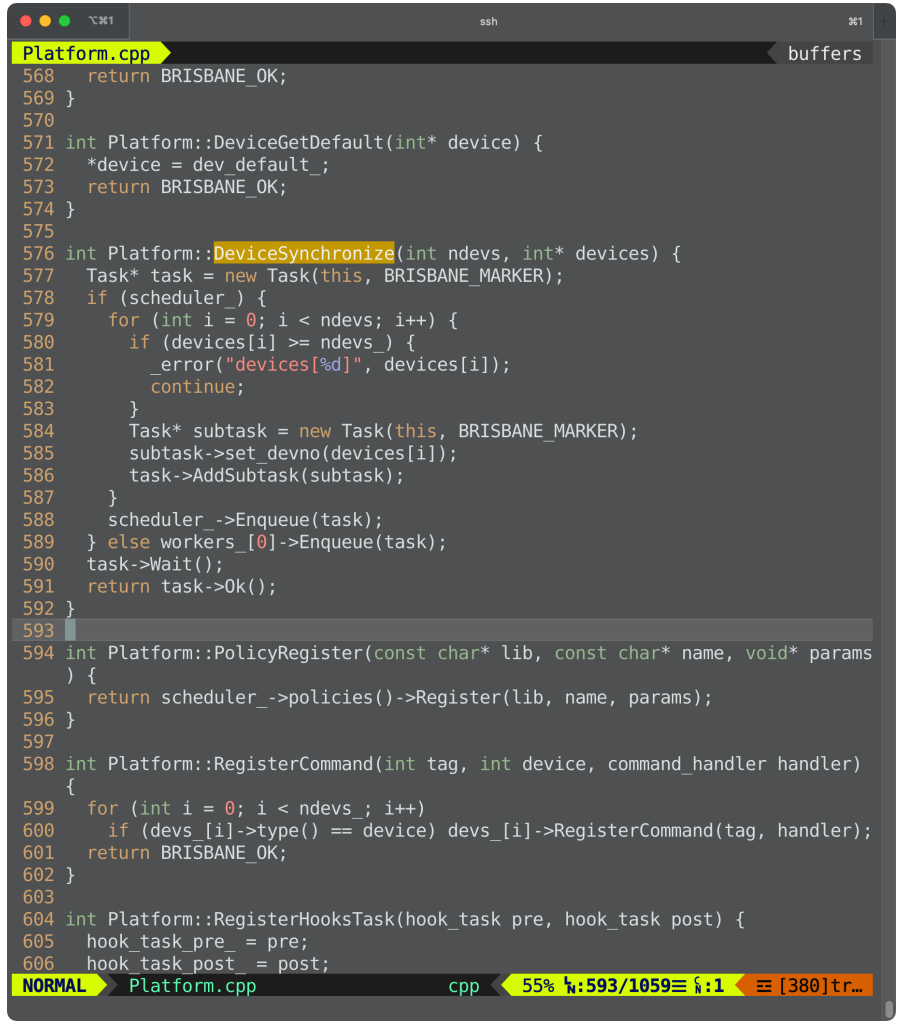
src/runtime/Platform.cpp





src/runtime/Platform.cpp





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]
[I] 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]
[I] cousteau [DeviceHIP.cpp:29:DeviceHIP] device[1] platform[1] vendor[Advanced Micro Devices] device[] ordinal[0] type[256] version[AMD HIP 40421432]
[I] 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
[I] cousteau [Platform.cpp:994:InitScheduler] Scheduler ndevs[3] ndevs_enabled[3]
[I] cousteau [DeviceHIP.cpp:70:Init] devid[0] max_compute_units[120] 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]
[I] cousteau [DeviceHIP.cpp:70:Init] devid[1] max_compute_units[120] 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]
[I] 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] [][1] -> [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.]
[I] cousteau [Platform.cpp:1020:ShowKernelHistory] kernel[brisbane_null] k[0.000000][0] h2d[0.000000][0] d2h[0.000021][1]
[I] cousteau [Platform.cpp:1020:ShowKernelHistory] kernel[kernel0] k[0.000104][1] h2d[0.510578][1] d2h[0.000000][0]
[I] cousteau [Platform.cpp:1020:ShowKernelHistory] kernel[kernel1] k[0.024058][1] h2d[0.000015][2] d2h[0.000000][1]
[I] cousteau [Platform.cpp:1025:ShowKernelHistory] total kernel[0.024162] h2d[0.510594] d2h[0.000021]
[I] cousteau [Platform.cpp:1048:Finalize] total execution time:[2.454120] sec. initialize:[1.413638] sec. t-i:[1.040482] sec
[I] cousteau [Platform.cpp:1049:Finalize] t10[0.000000] t11[0.000000] t12[0.000000] t13[0.000000]
[I] cousteau [Platform.cpp:1050:Finalize] t14[0.000000] t15[0.000000] t16[0.000000] t17[0.000000]
[I] cousteau [Platform.cpp:1051:Finalize] t18[0.000000] t19[0.000000] t20[0.000000] t21[0.000000]
```