**HETEROGENEOUS EXECUTION RUNTIME DESIGN**

**FOR**

**PROJECT ATLANTIS**

**Version 0.18**

**By Evgeny Baskakov**

**12/21/2015**



**2701 San Tomas Expressway**

**Santa Clara, CA 95050**

Table of Contents

[1 Introduction 1](#_Toc473032797)

[1.1 Revisions 1](#_Toc473032798)

[1.2 Definitions, Acronyms, Abbreviations 1](#_Toc473032799)

[1.3 User Story 1](#_Toc473032800)

[1.4 Purpose 2](#_Toc473032801)

[1.5 Objectives 2](#_Toc473032802)

[1.6 System Requirements 2](#_Toc473032803)

[1.7 Stakeholders 2](#_Toc473032804)

[1.8 Referenced Documents 2](#_Toc473032805)

[2 Description 4](#_Toc473032806)

[2.1 Perspective 4](#_Toc473032807)

[2.2 Functionality 4](#_Toc473032808)

[2.3 Dependencies 4](#_Toc473032809)

[2.3.1 Compiler 5](#_Toc473032810)

[2.3.2 Unified Memory 5](#_Toc473032811)

[2.4 Limitations and Constraints 5](#_Toc473032812)

[2.5 External Consequences and Effects 6](#_Toc473032813)

[2.6 Assumptions 6](#_Toc473032814)

[2.7 Documentation 6](#_Toc473032815)

[3 Design 7](#_Toc473032816)

[3.1 Overview 7](#_Toc473032817)

[3.2 Homogeneous compilation and execution 9](#_Toc473032818)

[3.3 Key invariants 10](#_Toc473032819)

[3.4 Runtime mechanics 11](#_Toc473032820)

[3.4.1 Runtime initialization 12](#_Toc473032821)

[3.4.2 Device image 13](#_Toc473032822)

[3.4.3 Functions and global variables registration 14](#_Toc473032823)

[3.4.4 Offload entry points 14](#_Toc473032824)

[3.4.5 Function lookup 16](#_Toc473032825)

[3.4.6 Function export tables 16](#_Toc473032826)

[3.4.7 Thread identifiers 17](#_Toc473032827)

[3.4.8 Host thread reservation 17](#_Toc473032828)

[3.4.9 Reverse offload 18](#_Toc473032829)

[3.5 Compiler requirements 18](#_Toc473032830)

[3.5.1 Device image linkage 18](#_Toc473032831)

[3.5.2 Device functions export 18](#_Toc473032832)

[3.5.3 Device variables import 18](#_Toc473032833)

[3.5.4 Function pointers 19](#_Toc473032834)

[3.5.5 Support for C++ virtual functions 19](#_Toc473032835)

[3.6 Compiler Design 20](#_Toc473032836)

[3.6.1 Data Collection 20](#_Toc473032837)

[3.6.2 Initialization 20](#_Toc473032838)

[3.6.3 Launching 20](#_Toc473032839)

[3.7 CUDA 8.0 on NVIDIA Pascal 21](#_Toc473032840)

[3.7.1 CUBIN image 21](#_Toc473032841)

[3.7.2 Device functions export 21](#_Toc473032842)

[3.7.3 Device variables import 21](#_Toc473032843)

[3.7.4 RPC stubs generation 22](#_Toc473032844)

[3.7.5 Threading model limitations 22](#_Toc473032845)

[3.8 HX API 22](#_Toc473032846)

[4 Quality 37](#_Toc473032847)

[4.1 Functional Testing 37](#_Toc473032848)

[4.1.1 Synthetic test suite 37](#_Toc473032849)

[4.1.2 Pthreads implementation 37](#_Toc473032850)

[4.2 Performance Testing 38](#_Toc473032851)

[4.3 Exceptions and Other Considerations 38](#_Toc473032852)

[4.4 Risks 38](#_Toc473032853)

[4.5 Acceptance Requirements 38](#_Toc473032854)

[5 Reviews & Approval 39](#_Toc473032855)

# Introduction

## Revisions

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Modified By | Description |
| 0.1 | 05/04/2015 | Evgeny Baskakov | Initial Draft |
| 0.2 | 05/15/2015 | Evgeny Baskakov | Sections expanded, based on review from Luke Durant |
| 0.3 | 05/19/2015 | Evgeny Baskakov | Function pointers explained, formal API updated |
| 0.4 | 05/22/2015 | Evgeny Baskakov | Changes per Philip Cuadra’s and Luke’s notes |
| 0.5 | 05/28/2015 | Evgeny Baskakov | Changes per Luke’s notes; new library API is documented |
| 0.6 | 06/01/2015 | Evgeny Baskakov | Changes per James’ notes; the final acronym “HX” is used |
| 0.7 | 06/03/2015 | Evgeny Baskakov | More HX Runtime API changes per James’ review |
| 0.8 | 06/05/2015 | Evgeny Baskakov | New device API functions documented |
| 0.9 | 06/10/2015 | Evgeny Baskakov | Homogeneous model described (new section 3.2) |
| 0.10 | 06/15/2015 | Evgeny Baskakov | Explicit function tables; lazy HX RT initialization |
| 0.11 | 06/17/2015 | Evgeny Baskakov | Document structure fixes, actual HX API described |
| 0.12 | 07/28/2015 | Evgeny Baskakov | Many minor fixes; API split into host and device parts |
| 0.13 | 09/03/2015 | Evgeny Baskakov | Mixed hxLaunch() offload mode explained; minor fixes |
| 0.14 | 11/13/2015 | Evgeny Baskakov | Added \_\_hxInitAppRuntimeCallback API description |
| 0.15 | 12/17/2015 | Evgeny Baskakov | Added more API descriptions (hxdBarrier, TLS, etc) |
| 0.16 | 12/18/2015 | Sebastian Jodlowski | Added more API descriptions (simple locks, hxdFlush) |
| 0.17 | 12/21/2015 | Evgeny Baskakov | C++ virtual tables handling described |
| 0.18 | 01/12/2016 | Sebastian Jodlowski | Added more API description (nested lock, timer) |

## Definitions, Acronyms, Abbreviations

|  |  |
| --- | --- |
| Name | Description |
| Host | Main platform that starts the user program under the CPU control; e.g. Intel x86\_64 based server |
| Device | An external device that can run portions of the program code; for example, an NVIDIA GPU |
| UVM | Unified Virtual Memory; execution model in which the device has full access to the host memory |
| HMM | Heterogeneous Memory Manager; OS extension unifying memory address space for the host and device |
| HX or HX RT | “Heterogeneous Execution” run-time library |
| Pascal | The first GPU architecture to support full Unified Memory, HBM, NVLINK, etc. |
| GP100 | The first Pascal GPU, targeted for the Tesla/HPC market. |

## User Story

[Include information regarding the User Story; identifying who will use the product or functionality, how they will benefit from it.]

## Purpose

[Provide the general description for the product or functionality including basic information such as author, intended users, and release version.]

## Objectives

[Identify the goals and non-goals of the project specific to this specification. These should be simple bulleted list items which clearly state the main objectives.]

## System Requirements

[List the target platforms, hardware and operating systems for this project.]

## Stakeholders

|  |  |  |
| --- | --- | --- |
| Group | Contact | Description |
|  |  |  |
|  |  |  |
|  |  |  |

[All stakeholders of this project MUST be listed. The Group or External Company should be included, along with the primary contact name and email. Provide a brief description of the stakeholder’s interest, involvement, and/or impact.]

## Referenced Documents

[List all documents referenced in the creation of this document. This can include other functional specifications, other technical documents, or business cases. Enough information should be provided such that a reader can obtain any of the documents listed in this section. For internal documents, the title and link to the document is sufficient. Identify associated RFEs.]

1. Project Atlantis Functional Specification <https://www.dropbox.com/s/ungcirtmp0cx7s4/Atlantis_Functional_Specification.docx?dl=0>
2. Handling Function Pointers <https://www.dropbox.com/s/jc3cwxs731dazw8/Atlantis_Handling_Function_Pointers.docx?dl=0>
3. Device to Host Function Calls <https://www.dropbox.com/s/amfc0vfejfj6o47/Atlantis_RPC_Design.docx?dl=0>

# Description

## Perspective

The primary project goal is to build a toolchain for compiling programs written in C, C++ and Fortran, with runtime managed work offloading. The term “work offload” implies that a separate computational device (e.g. GPU) or multiple devices would run parts of the program code, reducing the host CPU load and the total program execution time. It is designed such that the runtime for a program containing explicit parallel regions (e.g. based on OpenMP or OpenACC API or even Pthreads) would use this toolchain to execute the explicit parallel regions on a separate device.

The toolchain (i.e., the compiler, linker and runtime support libraries) can add optional checks for the platform capabilities to the program code. When the program runs, it implicitly checks if the current platform satisfies necessary requirements, and if so, routes the control flow to the external computational device.

The runtime API design is intended to be language agnostic, which means that it can be used for offloading code written in any language, and does not depend on a particular threading framework (OpenMP / OpenACC / etc). That said, specific API functions probably will be exposed to ease/optimize integration with certain frameworks and toolchains.

In this document, we describe the requirements, design and API of the runtime support library providing the automatic work offloading.

## Functionality

The heterogeneous runtime is intended to be a bridge between the host system and the device or devices which are used to offload work.

The primary functions of the heterogeneous runtime are:

* Provide an API for offloading parts of the target program to the device;
* Initialize the external device and decide whether it is suitable to offload work to;
* Load the device code image and bind it to the client program.

Here and below, the runtime library is often referenced by acronym HX RT (Heterogeneous Execution Runtime).

## Dependencies

The following are the major dependencies that impact the design.

### Compiler

The primary dependency of the runtime library is the compiler. The compiler services these roles:

* Generate full-program host and device code;
* Generate the runtime library and target device initialization code;
* Generate registration the program code to be offloaded;
* Replace calls of functions to be offloaded with respective runtime API calls that bring the control to the device instances of these functions.

The compiler/linker is also responsible for exposing necessary function and variable tables from the host and device object code.

These topics are covered in greater detail in the further sections of this document.

### Unified Memory

Everywhere in this document it is assumed that the platform and the external device employ a fully unified memory model. In this model, any memory location can be accessed from the host and the device, effectively referencing the same data.

#### NVIDIA CUDA Devices

The primary target platform for this project is CUDA 8.0 on NVIDIA Pascal architecture. For this platform, there is an additional dependency: a fully unified memory model is only possible on Linux x86-amd64 with HMM (a.k.a. Heterogeneous Memory Manager) kernel patch.

Changes in the CUDA driver may also be necessary. The CUDA 8.0 compilation model doesn’t allow the generated device code to directly reference non-managed host global variables. Instead, every global variable is instantiated in either managed or device memory. A solution is to add a new driver API that would allow explicit relocation of the global variables in the device code to their host instances.

## Limitations and Constraints

Major design limitations are:

* The API supports only one device image, thus it is not possible to register multiple devices separately;
* Currently, the offload scheme is static[[1]](#footnote-1), which means that the set of functions being offloaded must be prepared in advance. The compiler must compile these functions for the target device and generate registration and entry point HX API calls for each them;
* Work offload is only possible at function granularity, that is, there is no way to offload a random fragment of the program;
* Offload is not supported in dynamically linked libraries[[2]](#footnote-2).

The design also imposes some language specific restrictions:

* Exceptions are not supported across offload and reverse offload calls in the call stack – for example C++ throw/try/catch and C/C++ setjmp/longjmp;
* Thread local variables (e.g. thread\_local in C++11 ) are not supported;
* External thread libraries (e.g. POSIX Threads, C++11 std::thread, etc) are not supported, at least out-of-the box;
* Some platforms may restrict usage of variadic functions (as in C/C++).

There are also a number of platform/device specific limitations. See section 3.7, ”CUDA 8.0 on NVIDIA Pascal”.

## External Consequences and Effects

[List the impacts and repercussions this project may have on other projects or systems external to this document. If other parts of a system could be affected, those representatives must be identified in the Stakeholders section.]

## Assumptions

* PGI will provide the compiler (both the host and device backends);
* Necessary CUDA driver changes will be available with CUDA 8.0.
* Necessary NVLINK and device compiler backend changes will be available with CUDA 8.0.

## Documentation

[List all documentation which is affected or required by this project. This may include manuals, release notes, or web content. All external groups required to facilitate this documentation should be considered stakeholders and included in the appropriate section.]

# Design

## Overview

When compiling a program, the compiler must decide which of its code parts may be offloaded. Once decided, it adds the respective runtime calls to the host program code. Now, these parts of code must be compiled for both the host platform and the target device, which results in multiple separate object code instances. After the whole program is compiled this way, there will be its full host object code plus some amount of device object code. The linker combines the host and device object code plus necessary initialization routines into a single executable.

Figure 1: Toolchain flow shows such a toolchain flow, in which a GPU is used as an external device.



Figure 1: Toolchain flow

The resulting executable should be able to run on any host platform, with or without the external device. If there is no device that can handle offloaded work, the program works as if it were compiled without attempts to offload anything. If there is a device in the system, it seamlessly handles the functions that the compiler earlier decided to offload.

Function granularity is used when mapping host/device code. That is, only whole functions are offloaded, not inner parts of the program code.

Global variables always must exist in one instance per program, which imposes an additional restriction. The current design requires that all global variables must be instantiated on the host side and referenced in the device code.

The user program must not be able to directly observe locations of the device side functions. It means that if the program takes a function address on the device side, the resulting pointer must keep the address of the host function. In such design, the host code may always call function pointers directly; for the device side, the compiler will emit special calls that translate the host function address to its device value. The key requirement is that the logic of the user program is never affected by existence of device functions.

These topics are covered in greater detail in the further sections.

Figure 2: Compilation model provides a closer look at the compilation and linking process:



Figure 2: Compilation model

The runtime library key responsibilities:

* Check the platform capabilities;
* Load and prepare the device part of the compiled program code;
* Build mapping between host and device code regions;
* Build mapping between host global variables and device code;
* Provide entry points for the program control flow to offload.

The mapping between the host and device code is built on these simple concepts:

* For any host function that is going to be offloaded, there must be its device instance;
* If a device instance of function f() calls another function g(), a device instance of g() must exist, or there must be a reverse offload call to the host instance of g();
* Any global variable is instantiated once, on the host side. Device functions must always reference the host side instances.

Figure 3: Execution model outlines the high-level heterogeneous execution model:



Figure 3: Execution model

In this figure, a program fragment is shown where the compiler decided earlier to offload functions foo() and bar(). It then generated HX API calls passing host pointers to these functions. When the program is run, the runtime also decides to go ahead and let the device to run them[[3]](#footnote-3).

On the host, t1 and t2 are independent application threads. The device then uses its internal threads t1' and t2' to run the separately compiled functions code (foo\_dev() and bar\_dev(), respectively). Once they finish, the control is returned back to the originated host threads.

## Homogeneous compilation and execution

It is beneficial to consider a special case of the heterogeneous model described above, in which the CPU itself plays the role of an external device. In this model, the host side of the compiled program stays the same as in a real heterogeneous case; it still uses the HX API to offload function calls. However, there is no separate device code produced by the toolchain.

Instead, the HX runtime directly executes offloaded calls on the CPU, without involving any separate computational unit. To conform to the thread configuration requested by the user program, native OS thread means can be possibly used.

This model is convenient to refer to as “homogeneous compilation and execution model”. See Figure 4: Homogeneous compilation for an example how the homogeneous model is used in the project Atlantis.



Figure 4: Homogeneous compilation

It can be used as a “reference implementation” of the heterogeneous model for new platforms and devices:

* It is easy to implement and port to new platforms, which provides an early project starting point;
* It can help in development of test suites, before the platform-specific implementation of the HX RT library is created;
* It can help to debug higher level runtimes (e.g. OpenMP) relying on the heterogeneous runtime, by eliminating the external device from the consideration.

## Key invariants

The heterogeneous compilation and execution model must be consistent in terms of the program correctness. This implies to following invariants that the implementation must keep:

* Full program code always exists on the host side. This guarantees the program correctness when there is no available device in the system, and when functions are called directly (e.g. from a reversely offloaded code);
* The compiled program sees host function pointers only. That is, it is not allowed to see and store pointers to functions compiled for the device;
* Global variables are instantiated on the host side only. The device code references them directly through symbol relocations resolved by the dynamic loader.

The next sections cover the runtime implementation details, with these invariants kept in mind.

## Runtime mechanics

In this section, we describe how the runtime functions are called and how they work. A formal API specification is given in section 3.8, “HX API”. This section also discusses modifications to the PGI compiler necessary to permit generation of binaries suitable for the HX runtime environment.

Here is a summary of the HX API. Note that it is split in two parts – host and device. The host functions exist solely for the host-side compiler and application use. The control the code and data registration and the actual offloading. On the device side, there exist auxiliary functions that provide information about and control threads int the current launch.

**Host functions and data structures**

struct HXKernelConfig;

typedef void \*HXLaunchId;

\_\_attribute\_\_((weak)) void \_\_hxInitAppRuntimeCallback();

void \_\_hxRegisterTable(const void \*funcTable[], size\_t funcCount, const char \*deviceFuncTableName);s

void \_\_hxRegisterEntry(void \*hostFunc, const char \*name);

void \_\_hxRegisterHostVar(void \*hostVar, size\_t varSize, const char \*name);

HX\_RESULT hxLaunch(const HXThreadModel \*threadModel, void \*hostFunc, void \*args);

HX\_RESULT hxLaunchAsync(const HXThreadModel \*threadModel, HXHostFunc hostFunc, void \*args, HXLaunchId \*pLaunchId);

void hxJoin(void \*launchId);

HX\_RESULT hxLockCreate(HXLock \*lock);

void hxLockDestroy(HXLock \*lock);

void hxNestLockCreate(HXLock \*lock);

void hxNestLockDestroy(HXLock \*lock);

void hxiInitThreadIds(unsigned int maxThreadCount);

unsigned int hxiAllocateThreadIds(unsigned int threadCount, HXThreadIDRange \*ranges, unsigned int maxRanges);

void hxiReleaseThreadIds(const HXThreadIDRange \*ranges, unsigned int rangeCount);

unsigned int hxReserveThreads(unsigned int count, unsigned int \*reservedCount);

void hxReleaseThreads(unsigned int poolTicket);

**Device functions**

unsigned int hxdThreadCount();

unsigned int hxdThreadNum();

unsigned int hxdThreadId();

bool \_\_hxdCallBegin(HXKernelConfig \*config);

void \_\_hxdCallEnd();

void \*\_\_hxdGetDeviceFunc(void \*func);

void hxdBarrier();

void hxdLockSet(HXLock \*lock);

void hxdLockUnset(HXLock \*lock);

bool hxdLockTest(HXLock \*lock);

void hxdNestLockSet(HXLock \*lock);

void hxdNestLockUnset(HXLock \*lock);

int hxdNestLockTest(HXLock \*lock);

void hxdFlush();

void \*hxdAllocTLS(size\_t entrySize);

void \*hxdGetTLS();

void \*hxdGetUserData();

void hxdLaunchNested(HXHostFunc targetFunc, void \*args, void \*userData, unsigned int threadCount);

double hxdGetWallTime();

double hxdGetTimePrecision();

(Note: the specific APIs are not necessarily stable yet, but general structure should be fairly stable).

(Note: the \_\_hx prefix generally means that the function is intended to be used solely by the compiler host backend, not a higher level runtime; the \_\_hxd prefix means that the function should be used by the device compiler backend only; the hx and hxd prefixes denote functions that should be used by the higher level runtime on the host and device sides, respectively).

The following sections explain the purpose of these functions and how they should be used.

### Runtime initialization

The runtime is initialized via explicit HX API calls. The compiler/linker must provide the following information upon the runtime initialization:

* List of the host functions to offload, along with their entry names on the device side;
* List of host global variables referenced on the device side, along with their names and data region sizes;
* Tables of functions on the host side associated with their device side instances;
* Binary image containing the device code.

The initialization code in the resulting executable should be contained in a constructor function. If the program host code is compiled separately into multiple object files (translation units), each of them should contain such a constructor. The executable must invoke all the constructors at the load time.

More formally, the “constructors” in translation modules are code regions that are necessarily executed once, prior the offloaded code execution. Normally, they should be executed before the program’s entry point. For this purpose, “.init” sections as in the ELF format can be used. Their execution must be limited by a single thread. This guarantees that by the time when the first offloaded code call is reached, all necessary \_\_hxRegisterEntry, \_\_hxRegisterTable and \_\_hxRegisterHostVar calls are done, sequentially (but not necessarily in any particular order).

Here’s an example of such HX API call sequence:

\_\_hxRegisterEntry(f1, “\_\_f1v\_entry”); ...

\_\_hxRegisterEntry(fN, “\_\_fNz\_entry”);

\_\_hxRegisterHostVar(&v1, sizeof(v1), “\_v1”); ...

\_\_hxRegisterHostVar(&vK, sizeof(vK), “\_vK”);

\_\_hxRegisterTable(\_\_hxFuncTable, N, “\_\_hxFuncTable\_file\_name”);

Note – the string names used to register functions/variables are actually their mangled/relocation names (not just their original names).

The resulting global initialization sequence should first register all functions to be potentially offloaded and all host variables referenced in the device code.

#### Optional application initialization

There is also a special weak[[4]](#footnote-4) API call void \_\_hxInitAppRuntimeCallback(). This function is called automatically prior to each table, entry and variable initialization (that is, from \_\_hxRegisterTable, \_\_hxRegisterEntry, \_\_hxRegisterHostVar).

The application does not need to call it directly. Instead, the application can define it itself if it needs to initialize before the HX RT library. If\_\_hxInitAppRuntimeCallback is not defined in the application, its empty weak version will be used during HX RT initialization.

Note that the application is responsible for making sure that the logic within its locally defined \_\_hxInitAppRuntimeCallback happens once (if it needs to).

### Device image

The host linker is responsible for linking a data array containing the device image. The HX Runtime expects that there is a special symbol \_\_hxDeviceImage exported, that points to the linked array.

Upon initialization, the HX runtime will use this symbol to load the external device.

### Functions and global variables registration

In the runtime initialization, the compiler must generate the following call for each function that may be offloaded:

\_\_hxRegisterEntry(&foo, "\_\_foov\_entry");

The compiler must pass such string function name (“\_\_foov\_entry”) that it uniquely matches the entry point for the function foo() exported from the device code. An obvious way is to use a compiler-mangled function name (e.g. that is commonly used for C++ programs compilation).

Global variables used on the device side must refer to the host side data. Depending on the target device compiler backend, actions may be needed to achieve that. For that purpose, the runtime exposes an API for global variables registration:

int gBar;

...

\_\_hxRegisterHostVar(&gBar, sizeof(gBar), "\_gBar");

This function binds the given host variable (gBar) to its references on the device side. The string variable name must match its actual relocation name (i.e. mangled by the compiler) used in the compiled device image.

Static variables defined in the host code and used by offloaded functions must also be exported from the device side and registered in the runtime as well. The compiler is responsible to mangle their names in a unique way and export them along with other (conventional) global variables.

### Offload entry points

When the compiler or a high-level library decides to offload a function, the original function call is replaced with call of the hxLaunch function (see section 3.8.1.1.6 for its formal declaration).

The replacement should always be done conditionally to maintain execution when work offloading is not possible (some deployment systems may not have the target device installed, or the device doesn't have enough available resources).

Offload entry points exported from the device image must have a special form expected by the HX Runtime. The original functions should be called by these entry points.

Consider this source program code:

extern int foo(char x, double y);

...

int r = foo(x, y);

The hxLaunch() API call expects a thread model configuration and a parameter buffer for every offloaded function, so the calling host code could look like this (in C-like pseudocode):

struct HXArgsBuffer {

void \*returnValue;

void \*params[2];

};

HXThreadModel threadModel;

threadModel.deviceThreadCount = 10;

threadModel.threadCount = 10;

threadModel.lanesPerWarp = 1;

threadModel.useTicket = false;

HXArgsBuffer argsBuf;

argsBuf.params[0] = &x;

argsBuf.params[1] = &y;

argsBuf.returnValue = &r;

if(**hxLaunch**(&threadModel, &foo, &args) != HX\_SUCCESS) {

r = foo(x, y);

}

Therefore, the device instance of foo() must take the arguments from that argument buffer. Also, the device part of the Heterogeneous Execution Runtime may have to do platform specific actions upon calling every function. These actions are performed by the API calls \_\_hxdCallBegin() and \_\_hxdCallEnd(). The \_\_hxdCallBegin() function returns true if (and only if) this entry has to be executed by the current thread. Its parameter is a pointer to an opaque structure HXKernelConfig containing implementation defined device execution traits.

Hence, the offload entry point for foo() should look like this:

void \_\_foo\_entry(HXArgsBuffer \*args, HXKernelConfig \*config) {

if (**\_\_hxdCallBegin**(config)) {

\*args->returnValue =

foo(\*((char\*)args->params[0]), \*((double\*)args->params[1]));

}

**\_\_hxdCallEnd**();

}

In the example above, the \_\_foo\_entry() function must be exported from the device image and registered in the Heterogeneous Execution Runtime via the \_\_hxRegisterEntry() API call for the host foo(). The device backend is free to keep the original foo() function untransformed.

The hxLaunch() API call allows mixed heterogeneous execution of the target function. The HXThreadModel structure setup may set less device threads than its total number (see the deviceThreadCount and threadCount fields). In such case, the respective number of threads will run on the host, executing the host instance of the target function.

It is unspecified how host and device threads are numbered, i.e. how their IDs and indices (available to the user side with the hxdThreadNum() and hxdThreadId() API functions) will be assigned. See section 3.4.7 for more information on how thread identifiers are managed.

### Function lookup

The hxLaunch(&foo) runtime call looks for the "foo" function registration on the device side. The generic lookup procedure is done through a table that is initialized upon function registration (see \_\_hxRegisterEntry()).

The function table is immutable after the runtime is initialized, which means that it is only created and filled at the program startup. It is not protected by any synchronization mechanisms[[5]](#footnote-5).

Functions being offloaded must exist in a callable form on the device side of the compiled code. The runtime may need to be able to find them by name. Therefore, a function table must be exposed from the device binary image, allowing the host side runtime to find a function address by its name, e.g.:

void \_\_hxRegisterEntry(void \*hostFunc, const char \*funcName) {

DeviceFunc deviceFunc = devGetFuncByName(deviceImage, funcName);

...

}

The actual runtime implementation is free to choose another method (not based on names) of binding the device and host functions.

### Function export tables

Besides registering functions to be offloaded, the compiler must emit function association tables for each compilation unit (object file). The HX runtime will use these tables for function pointer calls (see section 3.5.4, ”Function pointers”); implementations can also use these tables to perform optimized offload calls.

In each compilation unit on the host side, the .init section must contain this call:

\_\_hxRegisterTable(\_\_hxFuncTable, N, “\_\_hxdFuncTable\_file\_name”);

The \_\_hxFuncTable symbol is statically declared per each host compilation unit, e.g.:

static const void \*\_\_hxFuncTable[] = {&foo, &bar};

Foo and bar are user functions contained in this compilation unit. N is the number of elements in \_\_hxFuncTable.

The “\_\_hxdFuncTable\_file\_name” string used in the \_\_hxRegisterTable call is a unique symbolic name of the corresponding table in the respective compilation unit on the device side. This table must contain the device addresses of the respective functions (foo and bar in the example above).

The “\_file\_name” suffix should not necessarily be a file name. The compiler is free to choose any unique identifier for the table in each compilation unit – e.g. a file name hash, a timestamp, or some plain numbering.

On the device side per each compilation unit, there must be an externally visible table containing the respective function pointers, e.g.:

const void \*\_\_hxdFuncTable\_file\_name[] = {&foo, &bar};

### Thread identifiers

The hxLaunch() and hxLaunchAsync() calls guarantee that created threads will have unique identifiers. The hxdThreadNum() function returns a *local* unique thread index within the current offload, ranging from 0, inclusively, to the number of threads per this launch, exclusively. The hxdThreadId() function returns a *global* unique thread identifier, ranging from 0, inclusively, to the maximum available number of HXRT threads, exclusively. For more information about these functions, refer to sections 3.8.1.2.5 and 3.8.1.2.6 for more formal API descriptions.

Because both numbers provided by hxdThreadNum() and hxdThreadId() are always bounded within the predicted margins, the user may use them to directly index data arrays with thread-specific data.

#### Thread ID ranges

Because global thread IDs are guaranteed to be bounded while being unique across the system, internally they become a resource that HXRT needs to allocate per user request and maintain in a consistent state.

The chosen approach is to keep all available thread IDs in a consecutive range [0…maxThreads-1]. The thread ID 0 is automatically assigned to the main program thread. Upon a top-level hxLaunch() or hxLaunchAsync() call, HXRT allocates the leftmost available subrange, [1…requestedThreadCount], which leaves the [requestedThreadCount+1…maxThreads-1] range available for further allocations.

If the user code runs nested offloads (i.e. for example, there is a host top-level hxLaunch, and some threads from that launch also call hxLaunch, performing device or host offloads), more thread ID ranges are allocated internally. When the nested offloads end, the allocated ranges are returned to the set of available IDs, but not all nested offloads may end at the same time.

It may happen that the thread ID space becomes fragmented. In this case, HXRT will allocate multiple ranges for the next offload.



Figure 5: Thread ID range allocation

In Figure 5, there is a top-level (level 0) launch with 6 threads. It has two nested level 1 launches, with 4 and 2 threads. For each of those launches, a thread ID range is assigned (the ranges are disjoined by a chance, because of some previous thread ID allocations and deallocations). For the level 2 launch consisting of 8 threads, there is no consecutive thread ID range available, so HXRT must assign it two ranges instead of one.

### Host thread reservation

TODO

### Reverse offload

Reverse Offload, or Device-to-Host function call is implemented within HXRT as a separate compiler-runtime interaction effort. Refer to the “Device to Host Function Calls” document [3] for details.

#### Memory allocation

TODO: Explain the problem; outline the solution (RPC only? replaced malloc/free heap memory allocator with a slow path through RPC?)

## Compiler requirements

The compiler is responsible for deciding which functions may be offloaded to the device. Once decided, it must generate the initialization and offload code. The compiler must also emit bindings between the host and device code.

More specifically, compiler is responsible for the following,

* For each function and variable participating in the offloading scheme, generate registration code (\_\_hxRegisterEntry, \_\_hxRegisterHostVar);
* Function export tables and the respective initialization code;
* Reverse Offload stub generation (see [3] for details).

### Device image linkage

As explained in section 3.4.2, “Device image”, the HX runtime depends on an external array containing the device image binary data. There is a linker-generated global variable containing a pointer to that data array:

extern “C” unsigned char \*\_\_hxDeviceImage[];

### Device functions export

The compiler must emit explicit function table and registration code for each of them, per compilation unit. See section 3.4.6, “Function export tables” for details.

### Device variables import

Global variables, as opposed to functions, must not be duplicated on host and device sides. Instead, the variable instances live entirely on the host, and referenced indirectly on the device side. We emphasize the importance of the unified memory model, which allows access of the same data using in the same address from the host and the device.

To achieve data uniqueness transparently for the program, the device backend must generate indirect accesses to global variables. Best way to do that is to use a relocation table, as in ELF shared objects. In this approach, the host side will be able to substitute the device addresses of global variables with their host addresses at the device image load time.

### Function pointers

Existence of two instances of each function presents a problem: taking a pointer to a function is ambiguous. To resolve it, we demand that the compiled program must always observe host function pointers, even if it takes address of a function in device code. This way, the generated host code is guaranteed to be consistent: every function pointer is always valid and can be called directly. On the device side, two challenges emerge.

#### Taking function address in device code

When taking a function address in the device code, the program must obtain the host address of that function. To achieve that, the device compiler backend must distinguish between calling functions and taking their addresses. For any function, the following rule applies:

f();

void (\*fp)() = &f;

When the “&f” construct is compiled, “f” must be replaced with a new symbol obtained from adding a unique prefix “\_\_HX\_” to the original function name. This resulting symbol “\_\_HX\_f” itself must be added to the function export table.

When the device image is dynamically loaded, the device image loader will replace (relocate) this symbol to the host instance of the “f” function.

#### Calling function pointers in device code

As any function pointer is known to hold its host address, all function pointer call sites on the device side must be made indirect. For that purpose, the HX device side API provides a special API call \_\_hxdGetDeviceFunc() which maps the given host function to its device instance.

The device compiler backend must use it to call any function through pointer:

void (\*dev\_fp)() = \_\_hxdGetDeviceFunc(fp);

(\*dev\_fp)(x);

Note that this transformation is done invisibly for the user program, to its logic is not affected.

### Support for C++ virtual functions

See the external document “Handling function pointers” [2] which discusses possible C++ virtual table implementations.

## Compiler Design

The following section discusses modifications to the PGI compiler to permit the generation of binaries suitable for the HX runtime environment.

### Data Collection

The PGI compiler must collect two sets of information during compilation. This information is required for emitting the necessary HX variable and routine initialization calls during program startup. The sets to be collected are the following:

* A set of all offloadable function symbols.
* A set of all global variables referenced by an offloadable function.

The offloadable routines can be collected in the OpenACC initialization routine, which should be called per each device image compilation.

The set of global variables can also be collected at the same point as the routine symbol name is collected. Since front-end parsing will have been completed by this point, the global variables should be known.

### Initialization

The PGI compiler is to emit code in the .ctor or .init section for each ELF object file being generated. The initialization process will make use of the two sets of information generated during earlier phases of compilation. These sets are the set of offloadable symbol names and the set of global symbol names. The symbol name of the device image must also be known at this point of compilation. With the two sets and the device image name known, the compiler can generate an initialization sequence in one of the two aforementioned ELF sections to fulfill the needs of the HX runtime.

Early PGI support will emit the expected initialization sequence to setup the HX runtime.

### Launching

The PGI compiler must detect offloadable routines to handle them appropriately. To accomplish this, offloadable routines will be apparent to the compiler via pragma notations: #pragma acc routine.

The PGI compiler also provides a compile time option which allows all routines to be considered offloadable.

When the compiler discovers a call site of an offloadable routine a kmpc call (an Intel OpenMP Runtime Library API, which will be designed implemented separately) should be executed. This call will be the result of the programmer inserting OpenMP pragmas throughout the source code. This includes:

* Determining the thread model that is to be used
* Building a compiler-generated argument buffer data structure contacting the actual arguments to the routine being called

The compiler must generate code to pass the necessary arguments to the kmpc launching routine, which will be intercepted by the HX runtime.

Early PGI support will not emit any launching code.

## CUDA 8.0 on NVIDIA Pascal

The first implementation of the heterogeneous compilation model is for CUDA 8.0 on the NVIDIA Pascal architecture. This platform imposes a number of additional requirements.

### CUBIN image

For the CUDA implementation of HX runtime, the \_\_hxDeviceImage array explained in section 3.5.1, ”Device image linkage” must contain the CUBIN image that can be loaded through the CUDA API call cuModuleLoadFatBinary().

### Device functions export

The generated CUDA binary must export entry pointer for all offloaded functions. These entry points must be visible via the CUDA Driver API cuModuleGetFunction(). Original functions are not required to be visible.

For example, for a function foo(), the device compiler backend can emit the following:

\_\_device\_\_ int foo(int x) {…};

\_\_global\_\_ void \_\_foo\_entry(HXKernelConfig \*config, void \*args) {…foo()…}

On the host, its registration will look like:

\_\_hxRegisterEntry(&foo, “\_\_foo\_entry");

### Device variables import

A new CUDA driver API function will be added for explicit relocation of these global variables after the cubin is loaded. See section 3.5.3, “Device variables import”.

### RPC stubs generation

TODO: Luke's explanation

### Threading model limitations

The Pascal execution model virtualizes thread stack memory so that:

1. GPU threads don't have access to each other's stacks;
2. The CPU doesn't have access to any GPU thread stack data; moreover, from its view point, all GPU stacks map to the same virtual address region, which makes it impossible uniquely identify GPU thread local data.

Two important implications:

1. GPU threads cannot call host functions (via RPC) while passing pointers to stack data. For example, this code won't work:

char buf[32];

sprinf(buf, "Kaboom!");

unless there is a GPU version of sprintf().

To handle this, the CUDA driver will produce a segfault with some traceback information, which will help the user to understand what call has failed.

1. Pointers to local thread data stored in global memory won't be correctly visible for other GPU and CPU threads. **TODO**: how do we handle that? Segfault, barrier with heavy data copying?..

## HX API

This section goes through HX runtime API functions. Note that this section may be slightly behind the actual implementation. Also note that the API is split info host and device parts, as explained in 3.4, “Runtime mechanics3.4.

#### Host API

##### \_\_hxRegisterTable

**void \_\_hxRegisterTable(const void \*funcTable[], size\_t funcCount, const char \*deviceFuncTableName);**

**Parameters:**

* funcTable: array of host function addresses;
* funcCount: number of function pointers in funcTable;
* deviceFuncTableName: symbol name of the respective function table on the device side.

**Description:**

This function is intended to be called once per compilation unit (e.g. in a constructor section .init). All function tables on the device side must have unique names.

For example, consider a host object file "fred.o" with two functions foo() and bar(). In this case, it would contain the following locally visible constructs:

static const void \*\_\_funcTable[] = { (void\*)&foo, (void\*)&bar };

A constructor function would register this table in such manner:

\_\_hxRegisterTable(\_\_funcTable, 2, "\_\_funcTable\_fred");

Then, the respective device object file with device implementations of foo() and bar() would contain a symmetric table, declared with global visibility:

const void \*\_\_funcTable\_fred[] = { (void\*)&foo, (void\*)&bar };

Effectively, in the example above, the \_\_hxRegisterTable call builds association between the host and device implementations of functions foo() and bar().

##### \_\_hxRegisterEntry

**void \_\_hxRegisterEntry(void \*hostFunc, const char \*name);**

**Parameters:**

* hostFunc: pointer to host function to offload;
* name: compiler-dependent mangled function name.

**Description:**

Registers the given host function for offload. The compiler must provide such mangled function name that the HX runtime would be able to locate the function code in the device image.

This API function never fails. Any possible internal failure (such as failure to locate the function in the device image) will result in program termination.

This API call is not thread safe meaning that it must be called from one thread at a time without any other concurrent HX calls. The compiler should generate call of this API function only in constructor sections of any translation module (object file). Refer to the section 3.4.1, “Runtime initialization” for clarification of the constructor sections.

##### \_\_hxRegisterHostVar

**void \_\_hxRegisterHostVar(HXHostVar hostVar, size\_t varSize, const char \*name);**

**Parameters:**

* hostVar: pointer to host global variable;
* varSize: size in bytes of the memory region occupied by the variable;
* name: compiler-dependent mangled variable name.

**Description:**

Registers the given host global variable that is accessible from the device side. The compiler must provide such mangled variable name that the HX runtime would be able to locate it in global export tables from the device image. Also, it must provide the correct variable size. It is assumed that both the host and the device allocate the same number of bytes for the variable.

This API function never fails. Any possible internal failure (such as failure to locate the variable reference in the device image export tables) will result in program termination. The compiler / upper runtime layer is responsible for providing correct data to the HX RT.

This API call is not thread safe meaning that it must be called from one thread at a time without any other concurrent HX RT calls. The compiler should generate call of this API function only in constructor sections of any translation module (object file). Refer to the section 3.4.1, “Runtime initialization” for clarification of the constructor sections.

##### \_\_hxInitAppRuntimeCallback

**\_\_attribute\_\_((weak)) void \_\_hxInitAppRuntimeCallback();**

**Description:**

This weak function is called prior to each \_\_hxRegisterTable, \_\_hxRegisterEntry, \_\_hxRegisterHostVar call. The application can benefit from this by defining its version of \_\_hxInitAppRuntimeCallback. This way, the application initializer will be called from HXRT initialization.

##### hxMaxThreadCount

**HX\_RESULT hxMaxThreadCount(unsigned int \*pMaxThreads);**

**Parameters:**

* pMaxThreads: a pointer to a user variable to contain the maximum number of threads per offload. The provided pointer must be non-zero.

**Return value:**

* HX\_SUCCESS in case of success; HX\_ERROR otherwise.

**Description:**

This function tells the maximum number of threads that the user can request for an offload. Note that there's no guarantee that hxLaunch() will necessarily satisfy requests for this number of threads. The value returned by this function tells the upper bound of threads number for the particular platform / device that can be satisfied by hxLaunch().

##### hxLaunch

**HX\_RESULT hxLaunch(const HXThreadModel \*threadModel, void \*hostFunc, void \*args);**

**Parameters:**

* threadModel: pointer to thread model configuration used to offload the target function;
* hostFunc: pointer to host function to offload;
* args: pointer to argument buffer for the given function.

**Return value:**

* HX\_SUCCESS if the host function has been offloaded successfully; HX\_ERROR otherwise, in which case it is guaranteed that the function being offloaded has not started executing at all.

**Description:**

Depending on the threadModel configuration, offloads the given the host function to the device. The host function must be previously registered by call to \_\_hxRegisterEntry(), otherwise the behavior is undefined.

Offload is done to the device if the threadModel->deviceThreadCount is non-zero. The device will run the device instance of the target function in the respective number of threads. The rest of the launched threads (their number is calculated as threadModel->deviceThreadCount - threadModel->deviceThreadCount) will run on the host, executing the host instance of the target function. Device and host threads will be numbered and assigned ids sequentially, in no particular order. That is, hxdThreadId() will return a unique id for any thread, no matter whether it is host or device. Similarly, hxdThreadNum() will return a unique thread index per this offload between 0 and threadModel->threadCount for any thread, no matter whether it is host or device.

The argument buffer is just a pointer to the original function’s parameters and, possibly, its return value. The calling code is responsible to arrange allocation and accessing the parameter buffer properly.

The threadModel, hostFunc parameters must be non-zero. Passing zero leads to a runtime crash.

See an example of a possible offload call site implementation the section 3.4.4, “Offload entry points”.

This API function is synchronous, which means that in case of successful offload, it waits until the offloaded function finishes its execution on the device.

The host and device backends, as well as the HX implementation for the particular platform will ensure that the global memory is consistent upon entering to and exiting from hxLaunch(). Both backends and the HX implementation shall guarantee the sequential consistency of the offloaded call.

This API call may fail. For example, the failure may happen because the device is busy running other offloaded code. The compiler should generate a backup path involving calling the host function directly.

This API call is thread safe, which means that multiple host threads are free to offload host functions without synchronizing with each other.

##### hxLaunchAsync

**HX\_RESULT hxLaunchAsync(const HXThreadModel \*threadModel, HXHostFunc hostFunc, void \*args, HXLaunchId \*pLaunchId);**

**Parameters:**

* threadModel: pointer to thread model configuration used to offload the target function;
* hostFunc: pointer to host function to offload;
* args: pointer to argument buffer for the given function;
* pLaunchId: pointer to memory location where an unique launch id will be stored upon success.

**Return value:**

* HX\_SUCCESS if the host function has been offloaded successfully; HX\_ERROR otherwise, in which case it is guaranteed that the function being offloaded has not started executing at all.

**Description:**

Depending on the threadModel configuration, offloads the given the host function to the device. The host function must be previously registered by call to \_\_hxRegisterEntry(), otherwise the behavior is undefined.

Offload is done to the device if the threadModel->deviceThreadCount is non-zero. The device will run the device instance of the target function in the respective number of threads. The rest of the launched threads (their number is calculated as threadModel->deviceThreadCount - threadModel->deviceThreadCount) will run on the host, executing the host instance of the target function. Device and host threads will be numbered and assigned ids sequentially, in no particular order. That is, hxdThreadId() will return a unique id for any thread, no matter whether it is host or device. Similarly, hxdThreadNum() will return a unique thread index per this offload between 0 and threadModel->threadCount for any thread, no matter whether it is host or device.

The pLaunchId parameter must a non-zero pointer to memory location where a unique launch identifier will be stored upon returning success. This identifier is necessary to pass to the hxJoin function which waits for the offloaded function to finish.

The threadModel, hostFunc and pLaunchId parameters must be non-zero. Passing zero leads to a runtime crash.

See an example of a possible offload call site implementation the section 3.4.4, “Offload entry points”.

This API function is asynchronous, which means that in case of success, it does not wait until the offloaded function finishes its execution on the device. Effectively, the successful call results in a separate simultaneous thread of execution, with entry point in the offloaded function which is started on the device.

The host and device backends, as well as the HX implementation for the particular platform will ensure that the global memory is consistent upon entering to hxLaunchAsync().

This API call may fail. For example, the failure may happen because the device is busy running other offloaded code.

This API call is thread safe, which means that multiple host threads are free to offload host functions without synchronizing with each other.

##### hxJoin

**void hxJoin(void \*launchId);**

**Parameters:**

* launchId: launch id of a asynchronously offloaded function.

**Description:**

Blocks until the function associated with the given launchId finishes.

The launchId parameter must be previously generated by a successful call of hxLaunchAsync, and there must not be two calls of hxJoin with the same launchId parameter value, otherwise the behavior is undefined.

The host and device backends, as well as the HX implementation for the particular platform will ensure that the global memory is consistent upon exiting from hxJoin().

This API call is thread safe, which means that multiple host threads are free to wait for different offload host functions without synchronizing with each other.

##### hxLockCreate

**HX\_RESULT hxLockCreate(HXLock \*lock);**

**Parameters:**

* lock: pointer to the lock structure.

**Return value:**

* returns HX\_SUCCESS upon success, respective error code otherwise.

**Description:**

Initializes the lock. The valid pointer to the pre-allocated lock structure shall be passed.

The lock shall be initialized only once before the first use of hxdLockSet() or hxdLockTest(). The lock might be re-initialized after being destroyed with hxLockDestroy().

This API call may fail. For example, failure may happen because of lack of memory to allocate internal lock structures.

##### hxLockDestroy

**void hxLockDestroy(HXLock \*lock);**

**Parameters:**

* lock: pointer to the lock structure.

**Description:**

Cleans up the lock. Only lock initialized with hxLockCreate() might be destroyed.

##### hxNestLockCreate

**void hxNestLockCreate(HXNestLock \*lock);**

**Parameters:**

* lock: pointer to the nested lock structure.

**Description:**

Initializes the nested lock. The valid pointer to the pre-allocated nested lock structure shall be passed.

The nested lock shall be initialized only once before the first use of hxdNestLockSet() or hxdNestLockTest(). The nested lock might be re-initialized after being destroyed with hxNestLockDestroy().

##### hxNestLockDestroy

**void hxNestLockDestroy(HXNestLock \*lock);**

**Parameters:**

* lock: pointer to the nested lock structure.

**Description:**

Cleans up the nested lock. Only lock initialized with hxNextLockCreate() might be destroyed.

#### Device API

##### \_\_hxdGetDeviceFunc

**void \*\_\_hxdGetDeviceFunc(void \*func);**

**Parameters:**

* hostFunc: pointer to host function to find its device instance;

**Return value:**

* pointer to the corresponding device function; NULL is never returned.

**Description:**

This function is used by the device compiler backend. The compiler should generate its call when a function is called by a pointer. It is guaranteed that any function pointer visible to the original program is valid on the host only. Therefore, when calling a function by a pointer, direct call is impossible. Instead, the corresponding device pointer must be located:

Original program code:

(\*fp)(x);

Device code:

void (\*dev\_fp)() = \_\_hxdGetDeviceFunc(fp);

(\*dev\_fp)(x);

If there is no device function instance matching fp, the \_\_hxdGetDeviceFunc() call will return a pointer to an appropriate RPC stub (see section 3.4.9, “Reverse offload”). If there is no RPC stub registered for such pointer (which is only possible if the original program used junk data as a pointer), the program behavior is undefined; Heterogeneous Execution Runtime implementations shall terminate the application, if possible.

##### \_\_hxdCallBegin

**bool \_\_hxdCallBegin(**HXKernelConfig \*config**);**

**Parameters:**

* config: pointer to offload call configuration structure;

**Return value:**

* True if the entry point is to be executed by the calling thread, false otherwise.

**Description:**

Calls of this function are generated by the device compiler backend at the beginning of every offload entry point body.  See section 3.4.4, “Offload entry points” for discussion.

The config parameter is an opaque pointer that comes from the hxLaunch() API call. An implementation of the HX Runtime may use it to do online setup of the spawned offload. Neither the higher level runtime nor the compiler should use the config parameter except passing it to \_\_hxdCallBegin().

##### \_\_hxdCallEnd

**void \_\_hxdCallEnd();**

**Description:**

Calls of this function are generated by the device compiler backend at the end of every offload entry point body.  See section 3.4.4, “Offload entry points” for discussion.

##### hxdThreadCount

**unsigned int hxdThreadCount();**

**Return value:**

* number of threads for the current offload launch.

**Description:**

This function returns the number of threads in the respective hxLaunch() call.

##### hxdThreadNum

**unsigned int hxdThreadNum();**

**Return value:**

* a number in region [0 … hxdThreadCount()-1], unique per thread, for the current offload launch.

**Description:**

This function returns an index of the current device thread in the current offload launch (hxLaunch() or hxLaunchAsync()). Threads are numbered from 0 to value returned by hxdThreadCount() minus 1.

##### hxdThreadId

**unsigned int hxdThreadId();**

**Return value:**

* thread identifier unique across all device threads.

**Description:**

This function returns a unique identifier for the current device thread. It is guaranteed that all existing device threads (across all co-existing offload launches) have different identifiers.

##### hxdBarrier

**void hxdBarrier();**

**Description:**

Blocks until all threads in the current launch execute this function. After this function exits, memory is guaranteed to be coherent across all threads in this launch (full memory barrier guarantee).

##### hxdLockSet

**void hxdLockSet(HXLock \*lock);**

**Parameters:**

* lock: pointer to the lock structure.

**Description:**

Tries to obtain the lock. Blocks until the lock is obtained.

The lock shall not be obtained by the same thread again without previously releasing it.

The lock shall be initialized with hxLockCreate() before first hxdLockSet() call on it.

##### hxdLockUnset

**void hxdLockUnset(HXLock \*lock);**

**Parameters:**

* lock: pointer to the lock structure.

**Description:**

Releases the lock.

The lock shall be previously obtained with hxdLockSet() or hxdLockTest().

The lock shall be initialized with hxLockCreate() before first hxdLockUnset() call on it.

##### hxdLockTest

**bool hxdLockTest(HXLock \*lock);**

**Parameters:**

* lock: pointer to the lock structure.

**Description:**

Tries to obtain the lock.

In contrary to hxdLockSet(), it does not block if the lock is not available.

Return true upon success, false otherwise.

The lock shall be initialized with hxLockCreate() before first hxdLockTest() call on it.

##### hxdNestLockSet

**void hxdNestLockSet(HXNestLock \*lock);**

**Parameters:**

* lock: pointer to the nested lock structure.

**Description:**

Tries to obtain the nested lock. Blocks until the lock is obtained.

If the lock is already held by the current thread, the nesting count for the lock is incremented.

The nested lock shall be initialized with hxNestLockCreate() before first hxdNestLockSet() call on it.

##### hxdNestLockUnset

**void hxdNestLockUnset(HXNestLock \*lock);**

**Parameters:**

* lock: pointer to the nested lock structure.

**Description:**

Decreases the nesting level. If the nesting level drops to 0, releases the nested lock.

The nested lock shall be previously obtained with hxdNestLockSet() or hxdNestLockTest().

The nested lock shall not be released by the thread which does not hold it.

The nested lock shall be initialized with hxNestLockCreate() before first hxdNestLockUnset() call on it.

##### hxdNestLockTest

**int hxdNestLockTest(HXNestLock \*lock);**

**Parameters:**

* lock: pointer to the nested lock structure.

**Return value:**

* the new nesting level upon success, 0 otherwise.

**Description:**

Tries to obtain the nested lock.

In contrary to hxdNestLockSet(), it does not block if the lock is not available.

Return new nesting level upon success, 0 otherwise.

The nested lock shall be initialized with hxNestLockCreate() before first hxdNestLockTest() call on it.

##### hxdFlush

**void hxdFlush();**

**Description:**

Performs full memory barrier across all threads.

After this function exits, memory is guaranteed to be coherent across all threads in this launch.

##### hxdAllocTLS

**void\* hxdAllocTLS(size\_t entrySize);**

**Parameters:**

* entrySize: size in bytes of the desired thread local storage;

**Return value:**

* a newly allocated thread local storage of the requested size, or NULL if no memory is available.

**Description:**

Allocates a thread private storage. If a previous storage was allocated, it expands or shrinks appropriately. In case if no memory is available, NULL is returned, but the current TLS storage is not affected.

##### hxdGetTLS

**void\* hxdGetTLS();**

**Return value:**

* returns a previously allocated thread local storage for the current thread, or NULL if none was allocated.

**Description:**

Returns a pointer to the thread local storage allocated by the previous hxdAllocTLS() call, or NULL if none was previously allocated.

##### hxdGetUserData

**void\* hxdGetUserData();**

**Return value:**

* returns the per-launch user data pointer provided upon hxLaunch() call.

**Description:**

Returns the per-launch user data pointer provided upon hxLaunch() call. If none was provided, NULL is returned.

##### hxdLaunchNested

**void hxdLaunchNested(HXHostFunc targetFunc, void \*args, void \*userData, unsigned int threadCount);**

**Parameters:**

* targetFunc: target function to call;
* args: target function parameters;
* userData: the pointer to substitute the userData pointer provided to the originating hxLaunch call;
* threadCount: the number of threads to launch (only 1 thread supported).

**Description:**

Launches nested device threads executing the target function. The launching thread is blocked until the target function is finished by all threads. After the target function returns, the original thread continues execution.

**Note**: at this moment, it only runs (emulates) 1 nested thread.

##### hxdGetWallTime

**double hxdGetWallTime();**

**Return value:**

* returns per-thread number of seconds elapsed from some fixed moment in the past.

**Description:**

Returns per-thread number of seconds elapsed from some fixed moment in the past.

Time returned for different threads may not correlate.

##### hxdGetTimePrecision

**double hxdGetTimePrecision();**

**Return value:**

* the smallest possible fraction of time in seconds accounted by the current device.

**Description:**

Returns the smallest possible fraction of time in seconds accounted by the current device.

# Quality

[If a formal Test Plan has been created, provide a reference to the document here and remove the remaining Quality sections from the specification. Otherwise, fill out the sections below to provide details of the quality control procedures planned for the functionality.]

## Functional Testing

[Describe the approach to testing the project’s functionality.]

Functional testing of the runtime library is complicated, because it depends on the compiler, which means that ideally they have to be tested simultaneously.

However, two compiler-independent testing frameworks have been developed.

### Synthetic test suite

TODO

### Pthreads implementation

As a separate effort, a limited POSIX Threads implementation is created on top of the HX runtime. The goal is to test the device offload functionality by exposing it to a large set of existing threaded applications and test frameworks.

The idea is to put threads created with pthread\_create to the device, that is, in effect to asynchronously offload their thread entry points. In this approach, the compiler has to compile as much of the program code as possible for the target device, but without generating the hxLaunch calls.

To support this, there is an hxLaunchAsync function in the HX API. Refer to the section 3.8, “HX API” for its formal description.

The following fundamental POSIX API functions have been implemented so far:

pthread\_create

pthread\_join

pthread\_self

pthread\_mutex\_init

pthread\_mutex\_destroy

pthread\_mutex\_lock

pthread\_mutex\_trylock

pthread\_mutex\_unlock

pthread\_cond\_signal

pthread\_cond\_broadcast

pthread\_cond\_wait

pthread\_cond\_timedwait

Currently, their basic runtime attributes are only supported: no detached threads, one runtime scheduling policy (SCHED\_OTHER), no recursive mutexes, etc. We believe that such level of support is enough to get a usable POSIX Threads implementation.

TODO: List of apps / frameworks tested.

## Performance Testing

[Describe the approach to testing the project’s performance.]

## Exceptions and Other Considerations

[Describe any exceptions to Functional testing and Performance testing which may be required.]

## Risks

[List all potential impacts of this project on other systems which may be adversely affected.]

## Acceptance Requirements

[Describe the minimum acceptable criteria which must be met in order to complete and validate this project. This may include the complete success of Functional Testing combined with meeting minimum acceptable performance.]

# Reviews & Approval

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Review Form | Result |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[Provide the Version number of the Formal Review and the Date that review occurred. Also provide a link to the Document Review Form and the Result of the Formal Review. Results should only be either “Approved”, “Conditionally Approved”, or “Not Approved”.

If the document is Conditionally Approved, a new line should be added once the document is promoted to Final. This line should include the Version and Date the final modifications were completed and recorded. The Review Form link should be left blank and the result should be set to “Approved”.]

1. A dynamic offload scheme may be employed in future, e.g. to support CUDA C++ JIT capabilities. [↑](#footnote-ref-1)
2. A subject to change in future, depending on customer needs. [↑](#footnote-ref-2)
3. The HX offload decision is always based on the device availability (that may be a dynamic trait, e.g. the device is too busy to handle another function). The initial initialization mail fail, in which case all subsequent hxLaunch() calls will fail. [↑](#footnote-ref-3)
4. See <https://en.wikipedia.org/wiki/Weak_symbol> and <https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/Function-Attributes.html> for background information about weak symbols. [↑](#footnote-ref-4)
5. This may change in future if a dynamic offload mechanism is necessary. [↑](#footnote-ref-5)