RedHawk Linux®
CUDA Persistent Threads (CuPer) API
User's Guide

Version 1.0

0898400-000
April 2018

Copyright 2018 by Concurrent Real-Time. All rights reserved. This publication or any part thereof is intended for use with Concurrent Real-Time personnel, customers, and end-users. It may not be reproduced in any form without the written permission of the publisher.

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent Real-Time makes no warranties, expressed or implied, concerning the information contained in this document.

Concurrent Real-Time and its logo are registered trademarks of Concurrent Real-Time. All other Concurrent product names are trademarks of Concurrent Real-Time while other product names are trademarks or registered trademarks of their respective owners. Linux® is used pursuant to a sublicense from the Linux Mark Institute.

Printed in the U. S. A.

Revision History:	Level:	Effective With:
April 2018	000	CUDA Persistent Threads (CuPer) API 1.0

1. Introduction

This document describes the CUDA Persistent Threads (CuPer) API operating on the ARM64 version of the RedHawk Linux operating system on the Jetson TX2 development board. These interfaces are used to perform work on a CUDA GPU device using the persistent threads programming model.

The persistent threads programming model avoids determinism problems caused by the standard CUDA launch/synchronization programming model. It does this by launching a CUDA kernel only once, at the start of an application, and causing it to run until the application ends.

Because the launch mechanism is not used to cause the CUDA device to perform a workload, another mechanism must be used. Similarly, because the synchronize mechanism is not used by the CPU to determine when the CUDA workload is complete, another mechanism must be used. The CuPer API provides those mechanisms.

Methods for utilizing this API and performance behaviors are discussed in the Concurrent Real-Time Whitepaper, "Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2", available on the Concurrent Real-Time website at URL `https://www.concurrent-rt.com/resources/whitepapers/`.

1.1. Restrictions

There exist some restrictions to this programming model:

- It is not possible to launch heterogeneous kernels throughout the application. The persistent kernel is running at all times. However, it is possible to use a switch to select from multiple pre-determined behaviors.
- The CUDA kernel must perform a busy wait when waiting for a workload from the CPU. It is likely to consume more power than allowing the CUDA GPU to become idle.
- Similarly, the CPU must perform a busy wait when waiting for completion of the GPU.
- The number of blocks and threads used by the kernel is limited to the number of resident threads supported by the device. This is 4096 on the Jetson TX2.

2. Cuper

The CuPer interfaces are provided in the `<cuper.h>` header file. They require no special link options.

All CuPer interfaces are declared within the `Cuper` namespace. There are two further namespaces declared within `Cuper`: `Std` and `DoubleBuffer`. In addition, two exception classes are defined directly within `Cuper`: `Error` and `CudaLibraryError`.

3. Cuper::Std

The `Cuper::Std` interfaces are designed for workload behavior which closely mirrors the standard CUDA launch/synchronization method.

3.1. Example Usage

The easiest way to understand this programming method is to start with a small example.

All the standard elements are declared within the `Cuper::Std` namespace. There are three classes defined therein: `Cpu`, `Cuda1Block`, and `CudaMultiBlock`. An object of the `Cpu` class is created in CPU source code. A typical usage would have a form similar to this:

```
void cpuFunction (…)
{
        Cuper::Std::Cpu  p;
        cudaHostGetDevicePointer(&d_A, h_A);
        Persistent<<<blocksPerGrid, threadsPerBlock>>>(p.token(), d_A);
        for (…) {
                … initialize h_A …
                p.startCuda();
                … possibly do unrelated CPU work …
                p.waitForCuda();
                … use results in h_A …
        }
        p.terminateCuda();
}
```

The CUDA kernel in this example is called `Persistent`. It is launched only once before entering the main loop. Any buffer(s) that will be used to pass user data back and forth during normal operation must be specified at that time. In addition, the `Cuper::Std::Cpu` object provides a `token()`, the value of which also must be passed.

The object, `p`, of the `Cuper::Std::Cpu` class is used to control the execution of workloads within the CUDA kernel. Within the main loop, `p.startCuda` informs the CUDA GPU that the input buffers are prepared and that it should begin performing its workload. This is analogous to a CUDA kernel launch. `p.waitForCuda` causes the CPU to wait for the work on the GPU to be completed. This is analogous to a CUDA synchronize.

If it is desired for the main loop ever to exit, `p.terminateCuda` may be called to request that.

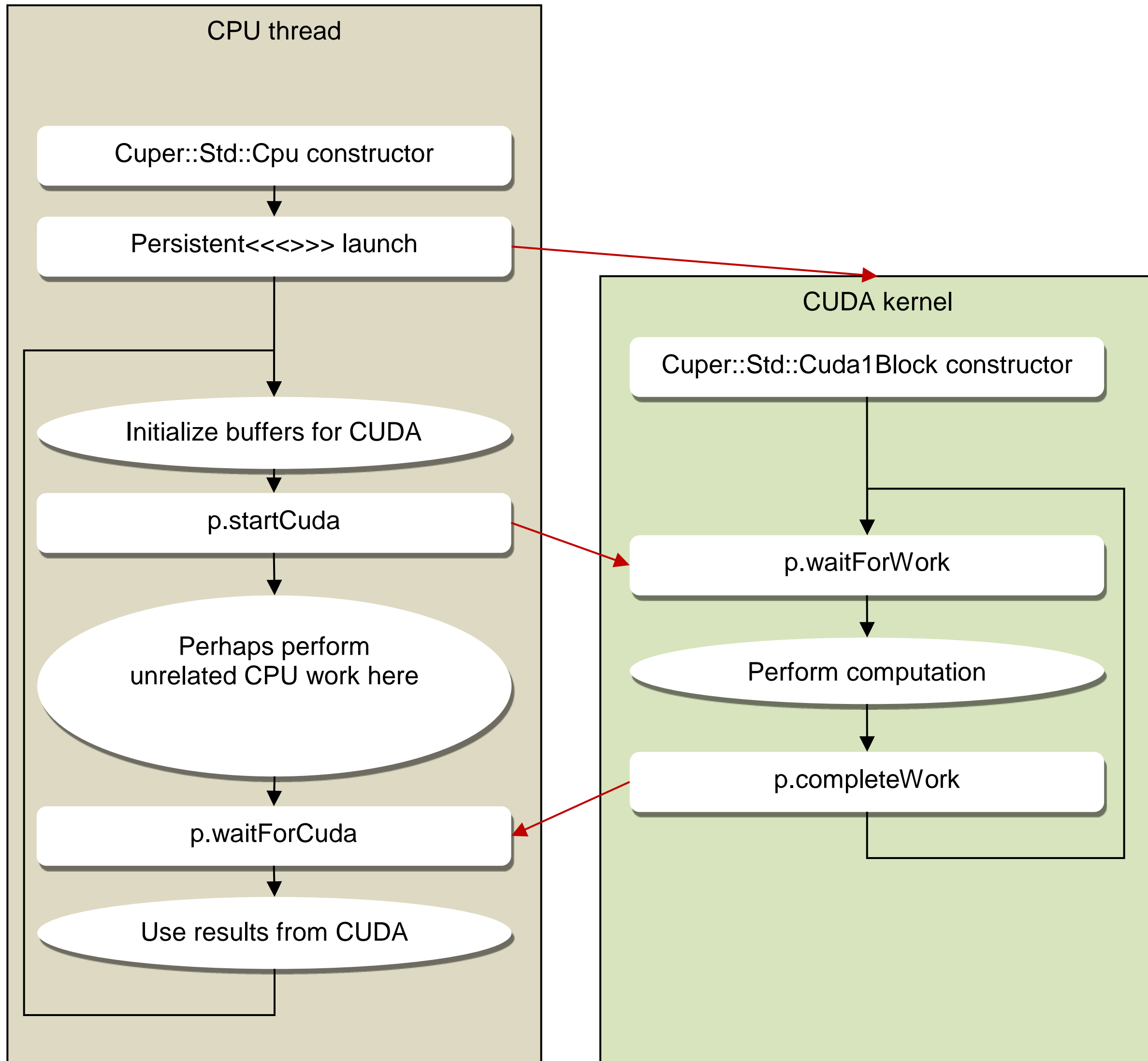
Meanwhile, the `Persistent` kernel in this example might have a form similar to this:

```
__global__ void Persistent (Cuper::Std::Token  token, float*  A)
{
        Cuper::Std::Cuda1Block  p(token);
        for (…) {
                p.waitForWork();
                if (p.isTerminated()) break;
                … perform workload …
                p.completeWork();
        }
}
```

Upon entry to the CUDA kernel, it creates object `p`, a `Cuper::Std::Cuda1Block`, and associates it with its `Cuper::Std::Cpu` counterpart using the token value passed from the CPU. This object receives commands from the `Cpu` counterpart and coordinates execution within the CUDA kernel. The kernel calls `p.waitForWork`, which causes the CUDA GPU to wait for a workload from the CPU. Upon return from that, it is safe to execute the workload with consistent user buffers. Once the computation is complete, it calls `p.completeWork` to indicate this to the CPU.

In addition, if the ability to terminate the kernel is desired, a call to `p.isTerminated` can be made to determine whether or not the CPU has requested termination.

The work flow can be visualized with this diagram:



The diagonal arrows show that:

- The `Persistent<<<>>>` launch starts the CUDA kernel
- `p.startCuda` releases the blocked `p.waitForWork`
- `p.completeWork` releases a blocked `p.waitForCuda`

This example shows the use of the `Cuda1Block` class. It is suitable for any CUDA workload which can be performed with only 1 block in the grid. The 1 block approach often is more efficient than multi-block solutions. However, if multiple blocks are required, `Cuda1Block` can be replaced with `CudaMultiBlock`.

One of the major goals of this approach is the avoidance of CUDA library calls within the main loop, including `cudaMemcpy`. So, zero-copy pinned memory is used in all cases using CuPer.

3.2. Cuper::Std::Token

`Cuper::Std::Token` is an opaque type used to bind a `Cuper::Std::Cuda1Block` or `Cuper::Std::CudaMultiBlock` object to an associated `Cuper::Std::Cpu` object.

3.3. Cuper::Std::Cpu

An object of class `Cuper::Std::Cpu` should be created from a CPU process or thread which intends to communicate with a persistent kernel. It has the following methods:

Cpu ()

The constructor expects no parameters.

~Cpu ()

The destructor expects no parameters.

Token token ()

The `token` function returns an opaque Token object intended to be passed to a CUDA kernel when it is launched. It should be passed to a `Cuda1Block` or `CudaMultiBlock` constructor to bind that object to this `Cpu` object.

void startCuda ()

The `startCuda` function is used to indicate to the associated `Cuda1Block` or `CudaMultiBlock` object that a CUDA workload is prepared. After this is called, the `Cuda*` objects' `isWorkReady` will return `true`, and their `waitForWork` calls will become unblocked.

Before this function is called, any input buffers used by the CUDA kernel should be fully initialized.

Calling this function without an intervening call to `waitForCuda`, or a call to `isCudaDone` which returns `true` is erroneous. Calling this function after calling `terminateCuda` also is erroneous.

void waitForCuda ()

The `waitForCuda` function is a blocking operation that waits for the associated `Cuda*` object's `completeWork` function to be called after the most recent `startCuda` call. Upon return, it indicates that output buffers for the workload are consistent and ready to be used.

bool isCudaDone ()

The `isCudaDone` function is a non-blocking operation that queries whether or not the associated `Cuda*` object's `completeWork` has been called after the most recent `startCuda` call. If it returns `true`, it indicates that the output buffers for the workload are consistent and ready to be used.

void terminateCuda ()

The `terminateCuda` function informs the object and its associated `Cuda*` object that the persistent kernel should terminate. This does not actually force the persistent kernel to terminate. It merely causes the `isTerminated` functions to return true.

bool isTerminated ()

The `isTerminated` function queries whether or not the given object has been terminated. Typically, this happens if its `terminateCuda` function is called, but it also can happen for a variety of detected error conditions.

3.4. Cuper::Std::Cuda1Block and Cuper::Std::CudaMultiBlock

An object of `Cuper::Std::Cuda1Block` or `Cuper::Std::CudaMultiBlock` should be created from a CUDA persistent kernel which is to be controlled by a `Cuper::Std::Cpu` object. If the grid is configured for a single block, `Cuper::Std::Cuda1Block` may be used, and will achieve higher performance. If the

grid is configured with multiple blocks, then `Cuper::Std::CudaMultiBlock` must be used. The methods for both types are the same and are as follows:

Cuda1Block (Cuper::Std::Token token)
CudaMultiBlock (Cuper::Std::Token token)

The constructor should be called from all threads in the grid, and the object must be in thread-local storage. Typically, this means it is declared as a local variable. It expects one parameter, which should be a `Token` value. That value is obtained by calling the `token` function from the `Cpu` object to which it should be bound.

~Cuda1Block () / ~CudaMultiBlock()

The destructor expects no parameters.

void waitForWork ()

The `waitForWork` function is a blocking operating that waits for a call to the associated `Cpu` object's `startCuda` function, either for the very first time, or after the most recent `completeWork` call. Upon return, it indicates that input buffers for the workload are consistent and ready to be used.

bool isWorkReady ()

The `isWorkReady` function is a non-blocking operation that queries whether or not the associated `Cpu` object's `startCuda` function has been called, either for the very first time or after the most recent `completeWork` call. If it returns `true`, it indicates that the input buffers for the workload are consistent and ready to be used.

void completeWork ()

The `completeWork` function is used to indicate to the associated `Cpu` object that a CUDA workload has been completed. After it is called, the `Cpu` object's `isCudaDone` function will return true, and its `waitForCuda` function will become unblocked.

Before this function is called, any output buffers used by the CUDA kernel should be fully prepared.

Calling this function without an intervening call to `waitForWork`, or a call to `isWorkReady` which returns `true` is erroneous. Calling this function on a terminated object – one for which the associated `Cpu` object's `terminateCuda` function was called – also is erroneous.

bool isTerminated ()

The `isTerminated` function queries whether or not the given object has been terminated. Typically, this happens if the associated `Cpu` object's `terminateCuda` function is called, but it also can happen for a variety of detected error conditions.

This function only is reliable after a return from `waitForWork` or `isWorkReady`. In either case, they will return if termination is requested, even if no workload is available.

4. Cuper::DoubleBuffer

The `DoubleBuffer` interfaces are designed for applications which intend to use double buffering to provide workloads to the CUDA persistent kernel. This approach works similarly to the `Std` interfaces, except that CUDA workloads are started in one real-time frame, but the results are used in the subsequent real-time frame. For example, in the CPU's 101st frame, it starts workload 101, and uses the

results from workload 100. This can be thought of as a 2-stage pipeline. This results in greater utilization of the CUDA GPU, if that extra delay is acceptable.

Important points about the approach:

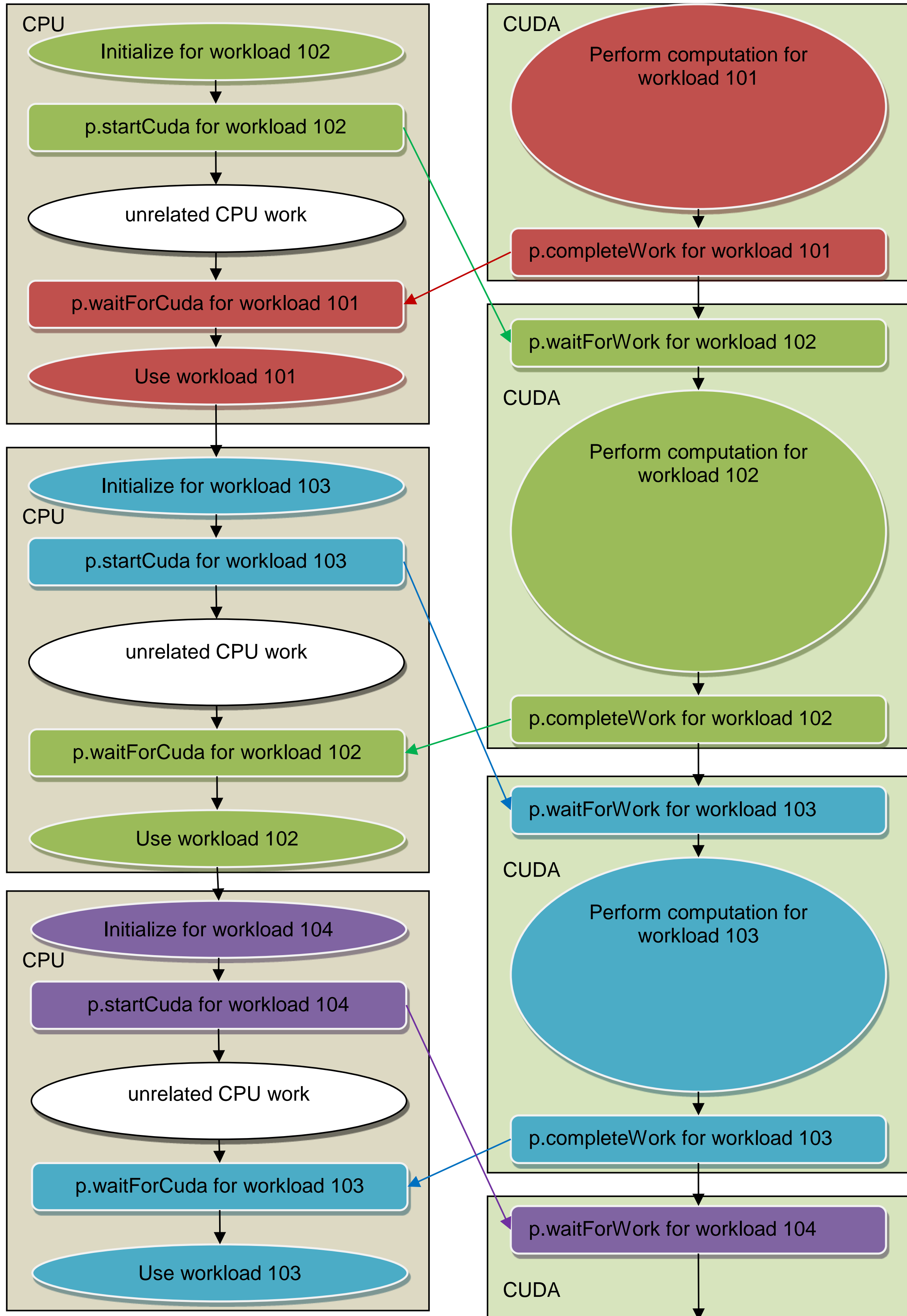
- The GPU can be kept busy, so long as the CPU is able to keep up with completed workloads.
- On the CPU, each workload is started in one frame and used in the subsequent frame.
- On the GPU, each workload is handled normally.
- To support this, there must be 2 sets of each buffer, and use of them alternates from one frame to the next. (This is the source of the term *double buffering*.) The Cuper::DoubleBuffer implementation keeps track of the correct buffer automatically.

4.1. Example Usage

An example usage of `Cuper::DoubleBuffer` might look like the following, with differences from `Cuper::Std` highlighted:

```
__global__ void Persistent (Cuper::DoubleBuffer::Token  token,
                                float* A0, float* A1)
{
        Cuper::DoubleBuffer::Cuda1Block  p(token);
        for (…) {
                p.waitForWork();
                unsigned int  which = p.claimBuffer();
                float*  A = which ? A1 : A0;
                …
                p.completeWork();
        }
}

void cpuFunction (…)
{
        Cuper::DoubleBuffer::Cpu  p;
        …
        Persistent<<<…>>>(p.token(), d_A0, d_1);

        // Start pipeline with an initial workload
        unsigned int  which = p.claimBuffer();
        float* h_A = which ? h_A1 : h_A0;
        … initialize h_A for workload 0…
        p.startCuda();

        for (unsigned int i = 1; …; i++) {
                which = p.nextBuffer();
                float* h_A = which ? h_A1 : h_A0;
                … initialize h_A for workload i…
                p.startCuda();

                … possibly do unrelated CPU work here …

                p.waitForCuda();
                which = p.claimBuffer();
                float*  h_A = which ? h_A1 : h_A0;
                … consume results in h_A for workload (i-1)…
        }

        // Flush last workload from pipeline
        p.waitForCuda();
        which = p.claimBuffer();
        float*  h_A = which ? h_A1 : h_A0;
        … consume results in h_A for workload i…

        p.terminateCuda();
}
```

The functioning of this code can be visualized with this diagram for the flow near example workloads 102 and 103:



The diagonal arrows in the diagram demonstrate the flow of work between CPU and GPU. Each workload is assigned a color, so it is possible to see how it progresses through the diagram.

For the descriptions herein, there is a concept of the number of active workloads. This is the difference between the number of workloads started and the number of workloads completed. The number of workloads started is the number of times `startCuda` has been called. The number of workloads completed is the number of times `claimBuffer` has been called following a `waitForCuda` call or an `isCudaDone` call which returned `true`.

4.2. Cuper::DoubleBuffer::Token

`Cuper::DoubleBuffer::Token` is an opaque type used to bind a `Cuper::DoubleBuffer::Cuda1Block` or `Cuper::DoubleBuffer::CudaMultiBlock` object to an associated `Cuper::DoubleBuffer::Cpu` object.

4.3. Cuper::DoubleBuffer::Cpu

An object of class `Cuper::DoubleBuffer::Cpu` should be created from a CPU process or thread which intends to communicate with a persistent kernel. It has the following methods:

Cpu ()

The constructor expects no parameters.

~Cpu ()

The destructor expects no parameters.

Token token ()

The `token` function returns an opaque Token object intended to be passed to a CUDA kernel when it is launched. It should be passed to a `Cuda1Block` or `CudaMultiBlock` constructor to bind that object to this `Cpu` object.

unsigned int nextBuffer ()

The `nextBuffer` function returns an integer which identifies the set of buffers to be initialized before calling the next `startCuda` function. It returns either 0 or 1.

Calling this function when two or more workloads are active is erroneous. Calling this function after calling `terminateCuda` also is erroneous.

void startCuda ()

The `startCuda` function is used to indicate to the associated `Cuda1Block` or `CudaMultiBlock` object that a CUDA workload is prepared. The buffers used for the workload are as returned by a previous call to `nextBuffer`. After this is called, the `Cuda*` objects' `isWorkReady` will return `true`, and their `waitForWork` calls will become unblocked.

Before this function is called, any input buffers used by the CUDA kernel should be fully initialized.

Calling this function when two or more workloads are active is erroneous. Calling this function after calling `terminateCuda` also is erroneous.

void waitForCuda ()

The `waitForCuda` function is a blocking operation that waits for the associated `Cuda*` object's `completeWork` function to be called for the earliest active workload. Upon return, it indicates that output buffers for the workload are consistent and ready to be used. The identity of the output buffers is returned by a subsequent call to `claimBuffer`.

Note that it is mandatory to call the `claimBuffer` function after this function returns.

bool isCudaDone ()

The `isCudaDone` function is a non-blocking operation that queries whether or not the associated `Cuda*` object's `completeWork` has been called for the earliest active workload. If it returns `true`, it indicates that the output buffers for the workload are consistent and ready to be used. The identity of the output buffers is returned by a subsequent call to `claimBuffer`.

Note that it is mandatory to call the `claimBuffer` function after this function returns with the value `true`.

unsigned int claimBuffer ()

The `claimBuffer` function returns an integer which identifies the set of buffers to be read for results for the more recently completed workload: the workload for which `waitForCuda` returned or for which `isCudaDone` returned true. It returns either 0 or 1.

It is mandatory to call this function after return from `waitForCuda` or a return from `isCudaDone` which returns `true`, and before the next `startCuda` call.

void terminateCuda ()

The `terminateCuda` function informs the object and its associated `Cuda*` object that the persistent kernel should terminate. This does not actually force the persistent kernel to terminate. It merely causes the `isTerminated` functions to return true.

bool isTerminated ()

The `isTerminated` function queries whether or not the given object has been terminated. Typically, this happens if its `terminateCuda` function is called, but it also can happen for a variety of detected error conditions.

4.4. Cuper::DoubleBuffer::Cuda1Block and Cuper::DoubleBuffer::CudaMultiBlock

An object of `Cuper::DoubleBuffer::Cuda1Block` or `Cuper::DoubleBuffer::CudaMultiBlock` should be created from a CUDA persistent kernel which is to be controlled by a `Cuper::DoubleBuffer::Cpu` object. If the grid is configured for a single block, `Cuper::DoubleBuffer::Cuda1Block` may be used, and will achieve higher performance. If the grid is configured with multiple blocks, then `Cuper::DoubleBuffer::CudaMultiBlock` must be used. The methods for both types are the same and are as follows:

Cuda1Block (Token token)
CudaMultiBlock (Token token)

The constructor should be called from all threads in the grid, and the object must be in thread-local storage. Typically, this means it is declared as a local variable. It expects one parameter, which should be a `Token` value. That value is obtained by calling the `token` function from the `Cpu` object to which it should be bound.

~Cuda1Block () / ~CudaMultiBlock()

The destructor expects no parameters.

void waitForWork ()

The `waitForWork` function is a blocking operating that waits for a call to the associated `Cpu` object's `startCuda` function for the next workload. Upon return, it indicates that input buffers for the workload are consistent and ready to be used.

Note that it is mandatory to call the `claimBuffer` function after this function returns.

bool isWorkReady ()

The `isWorkReady` function is a non-blocking operation that queries whether or not the associated `Cpu` object's `startCuda` function has been called for the next workload. If it returns `true`, it indicates that the input buffers for the workload are consistent and ready to be used.

Note that it is mandatory to call the `claimBuffer` function after this function returns with the value `true`.

unsigned int claimBuffer ()

The `claimBuffer` function returns an integer which identifies the set of input and output buffers which should be used for the workload. It returns either `0` or `1`.

It is mandatory to call this function after return from `waitForWork` or a return from `isWorkReady` which returns `true`, and before the next `completeWork` call.

void completeWork ()

The `completeWork` function is used to indicate to the associated `Cpu` object that a CUDA workload has been completed. After it is called, the `Cpu` object's `isCudaDone` function for the workload will return true, and its `waitForCuda` function for the workload will become unblocked.

Before this function is called, any output buffers used by the CUDA kernel should be fully prepared.

Calling this function without an intervening call to `waitForWork`, or a call to `isWorkReady` which returns `true` is erroneous. Calling this function on a terminated object – one for which the associated `Cpu` object's `terminateCuda` function was called – also is erroneous.

bool isTerminated ()

The `isTerminated` function queries whether or not the given object has been terminated. Typically, this happens if the associated `Cpu` object's `terminateCuda` function is called, but it also can happen for a variety of detected error conditions.

This function only is reliable after a return from `waitForWork` or `isWorkReady`. In either case, they will return if termination is requested, even if no workload is available.

5. Error Types

5.1. Cuper::Error

In the event that a run-time error is detected during execution of the CuPer API functions, they will throw an object of type `Cuper::Error`. This class is a subclass of `std::runtime_error`. It has the following interface, inherited from `std::runtime_error`:

const char* what () const

The `what` function returns a string describing the error.

5.2. Cuper::CudaLibraryError

In the event that a run-time error is detected by a CUDA library function during the CuPer API functions, they throw an object of type Cuper::CudaLibraryError. This class is a subclass of Cuper::Error. It has the following interfaces:

const char* what () const

The `what` function returns a string describing the error.

const char* cudaFunction () const

The `cudaFunction` function returns a string describing the simple name of the CUDA library function that failed.

cudaError_t cudaError () const

The `cudaError` function returns the error code returned by the failing CUDA library function.

6. Compilation Options

-DCUPER_DEBUG

Turn on full error checking code. Normally, this is disabled in functions expected to be called from hard real-time loops for efficiency reasons. But this compiler option can be used to enable it if a bug is suspected.

Alternately, the same functionality can be enabled by adding the following code earlier enough (before `#include <cuper.h>`) in the source:

```
#define CUPER_DEBUG
```

7. Support

If you need assistance, please contact the Concurrent Real-Time Software Support Center at our toll free number 1-800-245-6453. For calls outside the continental United States, the number is 1-954-283-1822. The Software Support Center operates Monday through Friday from 8 a.m. to 5 p.m., Eastern Standard Time.

You may also submit a request for assistance at any time by using the Concurrent Real-Time web main site at http://concurrent-rt.com/support or by sending an email to support@concurrent-rt.com.