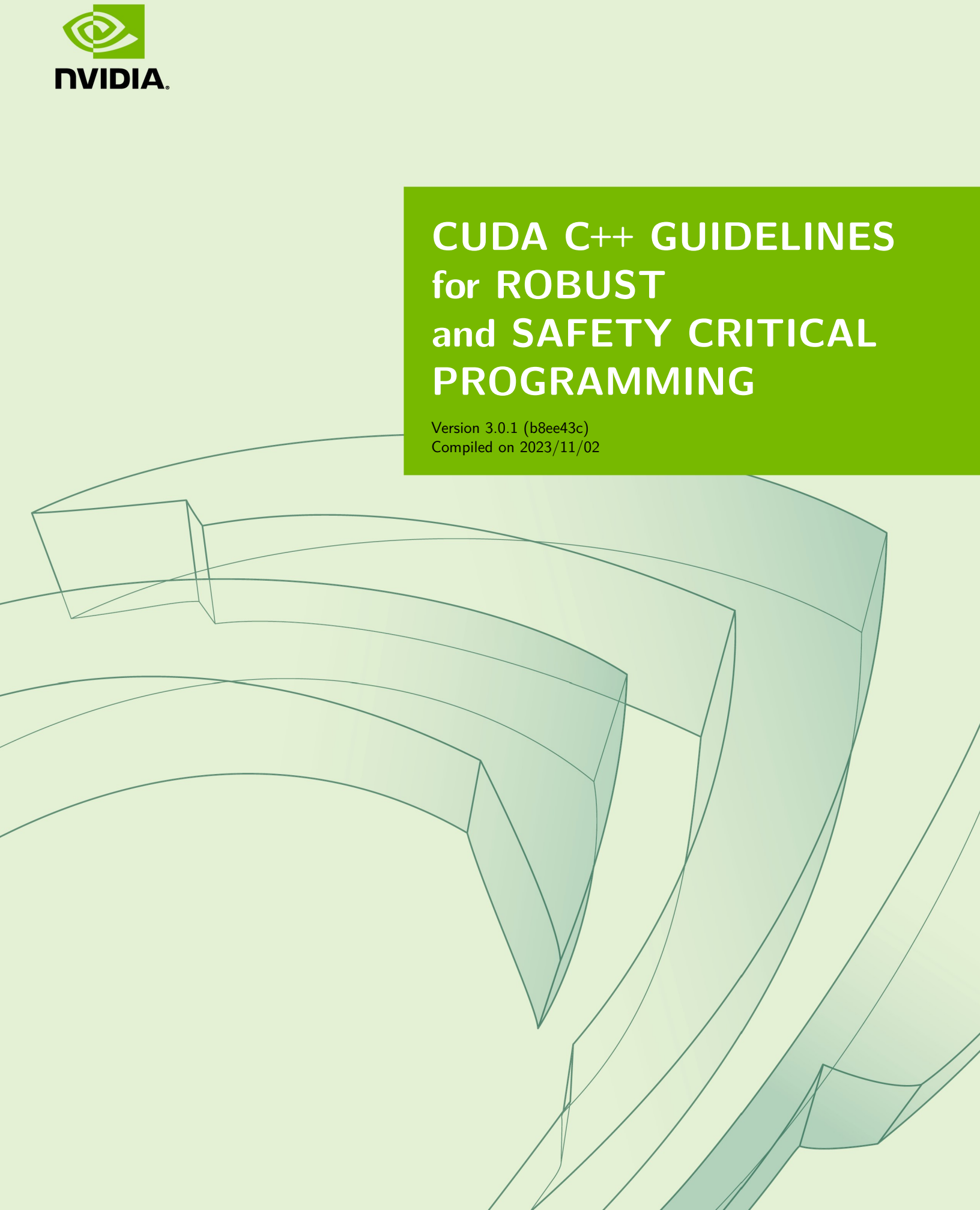

CUDA C++ GUIDELINES for ROBUST and SAFETY CRITICAL PROGRAMMING

Version 3.0.1 (b8ee43c)
Compiled on 2023/11/02

Contents

Introduction

This document describes a set of coding guidelines for:

- The CUDA C++ programming language.
- Usage of interfaceable CUDA libraries in the ISO C++ programming language.

The intention of these guidelines is to enforce practices that will increase the robustness of software that uses CUDA. These rules do not attempt to define what "good" or "performant" CUDA C++ code is. These guidelines are designed to be both readable by CUDA users and, in most cases, enforceable by automated processes through static and dynamic analysis tools.

These guidelines are not comprehensive. They only cover the aspects of the CUDA C++ programming language that differ from the ISO C++ programming language which CUDA C++ is based on. Thus, these guidelines should be used in combination with coding guidelines for the ISO C++ programming language, such as the `MISRA`, `AUTOSAR` or the `SEI CERT Coding Standards`.

These guidelines are broken down into, mostly, concrete rules. Each rule describes a principle that a program using CUDA should follow.

When using this ruleset, procedures for evaluating the correct application of the rules to a project should be designed by the team using the rules.

What is CUDA C++?

- CUDA C++ is a programming language defined by NVIDIA for the CUDA compute platform.
- CUDA C++ is distinct extension of ISO C++.
- The NVCC compiler and the CUDA C++ runtime are an implementation of CUDA C++.

CUDA C++ code consists of host entities (functions and objects) and device entities. Entities can be both host and device.

The extension for CUDA C++ source files is `.cu`.

What is Host Code?

There are two distinct types of host code:

- CUDA C++ host code.
- ISO C++ code that uses interfaceable CUDA libraries.

What are CUDA libraries?

Any software included in the official CUDA ecosystem that provides a programmatic ISO C++ interface is an interfaceable CUDA library, such as the following examples:

- The *CUDA runtime*
- The *CUDA driver*
- *cuBLAS*
- *cuFFT*
- *cuDNN*
- *NVRTC*

There are also CUDA C++ libraries which are only usable in CUDA C++:

- *Thrust*
- *Cooperative Groups*
- *CUB*
- *CUTLASS*

What is Undefined Behavior?

An operation which exhibits Undefined Behavior (UB) has no specified semantics and makes no guarantees about its behavior. When a program executes an operation that has undefined behavior, the behavior of the entire program is undefined.

Many rules in this document aim to avoid undefined behavior. In practice, undefined behavior can rarely be avoided entirely, as almost all production software relies on some undefined behavior.

Classification

As in MISRA C 2012, section 6.1, all CUDA guidelines are classified as either being a rule or a directive. The document attempts to follow a similar classification model.

A directive is a guideline that is either not fully defined, cannot be statically checked or should not be statically checked. Which of these three sets a guideline falls into is not tracked since directives are intended to be viewed by users as suggestions and not rules. As such directives should not be enforced by tools.

A rule is a guideline that has been fully defined and it should be possible to statically check for violations.

Scope

Each guideline has a scope:

- **Host:** Guidelines for CUDA C++ host code and/or ISO C++ code using interfaceable CUDA libraries.
- **Device:** Guidelines for CUDA C++ device code.

Audience

Each guideline has an audience:

- **CUDA C++:** Guidelines for CUDA C++.
- **CUDA Libraries:** Guidelines for using interfaceable CUDA libraries in ISO C++.

Category

Rules are divided into these categories: "Mandatory", "Required" or "Advisory". All rules in the same category are of equal importance and no difference in priority should be inferred by the order in which they appear in the document. The document provides a suggested definition for each category, however, projects should modify the definitions to match their needs.

The category a rule is assigned to, may change in any release of this document. However, the developer is encouraged to assess the change and decide whether to adopt the change in their process.

Mandatory Rules

These are requirements that the developer must follow without deviation. Formal deviations are not accepted and the code must be fixed. Deviations from these rules almost universally results in incorrect code.

Required Rules

These are requirements the developer should normally follow. If the rules are not followed then a formal deviation request must be submitted. These rules usually result in incorrect code, however, there may be mitigating mechanisms available to avoid the incorrect behavior.

Advisory Rules

These are rules that the developer is encouraged to follow. If the rules are not followed then no formal deviation request is required, however the developer is encouraged to file deviation reports to track the exception.

Hardware Applicability

Each rule is applicable to a certain set of CUDA Compute Capabilities. Most rules are applicable to all Compute Capabilities.

CUDA Rules

General Rules

CUDA 1.1 [device-dependent] Use CUDA after host program initiation and before host program termination

Perform the following operations only after host program initiation and before host program termination (not before or after `main`):

- Call device-dependent CUDA runtime or driver interfaces.
- Launch CUDA kernels.
- ODR-use a managed storage duration object (e.g. `__managed__`, CUDA C++ only).

Most CUDA runtime and driver interfaces are device-dependent, with some exceptions such as:

- `cudaGetErrorString` and `cudaGetErrorName`.
- `cuGetErrorString` and `cuGetErrorName`.
- `cuDriverGetVersion`.

Scope: Host.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Most CUDA runtime and driver interfaces are dependent on the existence of valid internal device state, which is lazily initialized during host program initiation and destroyed during host program termination. The CUDA runtime and driver cannot detect if this device state is invalid, so using any of these interfaces (implicitly or explicitly) during program initiation or termination will result in undefined behavior.

This means that most uses of the CUDA runtime or driver in the global constructors or destructors of C++ classes in the host program is undefined behavior.

Example 1 (Bad)

```
#include <cstdio>
#include <cassert>
#include <cuda.h>

__global__ void hello_world() {
```

(continues on next page)

(continued from previous page)

```
  printf("hello world\n");
}

struct launch_hello_world {
  __host__ launch_hello_world() {
    hello_world<<<1, 1>>>();

    cudaError_t const error0 = cudaGetLastError();
    assert(cudaSuccess == error0);
  }
  __host__ ~launch_hello_world() {
    hello_world<<<1, 1>>>();

    cudaError_t const error0 = cudaGetLastError();
    assert(cudaSuccess == error0);
  }
};

// This object's constructor is run during program initiation and attempts to
// launch a kernel, which is invalid. Likewise, the object's destructor is run
// during program termination and attempts to launch a kernel, which is invalid.
launch_hello_world launcher;

int main() {}
```

Example 2 (Bad)

```
#include <cassert>
#include <cuda.h>

struct device_storage {
  char* ptr;

  __host__ device_storage(int64_t bytes) {
    cudaError_t const error = cudaMalloc(&ptr, bytes);
    assert(cudaSuccess == error);
  }
  __host__ ~device_storage() {
    cudaError_t const error = cudaFree(ptr);
    assert(cudaSuccess == error);
  }
};

// This object's constructor is run during program initiation and attempts to
// call an initialization-dependent CUDA runtime function, which is invalid.
// Likewise, the object's destructor is run during program termination and
// attempts to call an initialization-dependent CUDA runtime function, which is
// invalid.
device_storage scratch(128);

int main() {}
```

Example 3 (Bad)

```
__managed__ int32_t global_object_count = 0;

struct A {
  __host__ A() {
    ++global_object_count;
  }

  __host__ ~A() {
    --global_object_count;
  }
};

// This object's constructor is run during program initiation and attempts to
// increment a `__managed__` object, which is invalid. Likewise, the object's
// destructor is run during program termination and attempts to decrement a
// `__managed__` object, which is invalid.
A a;

int main() {}
```

CUDA 1.2 [synchronize.termination] Synchronize with all devices before termination

Before the start of normal host program termination, synchronize with all outstanding work on all devices.

Scope: Host.

Audience: CUDA C++, CUDA Libraries.

Category: Advisory.

Hardware Applicability: All Compute Capabilities.

Rationale

Many CUDA interfaces are asynchronous and are evaluated concurrently with the host program on devices. The CUDA runtime and driver do not implicitly synchronize with any outstanding work on any devices during host program termination.

If a CUDA C++ program fails to synchronize with all outstanding work on all devices before normal host program termination (returning from `main`, calling `std::abort`, etc.), no diagnostic will be produced for any asynchronous errors that occur during the execution of that work.

Example 1 (Bad)

```
#include <cassert>


__global__ void fail() {
  *(int*)0 = 0;
}

int main() {
  fail<<<1, 1>>>();

  // This call succeeds, because the kernel launch did not have a synchronous
```

(continues on next page)

(continued from previous page)

```
  // error.
  cudaError_t const error0 = cudaGetLastError();
  assert(cudaSuccess == error0);

  // The assert in the kernel leads to an asynchronous error which is silently
  // ignored.
}
```

Example 2 (Good)

```
#include <cassert>
#include <cuda.h>
#include <stdio.h>

__global__ void fail(int n, float m, float *y) {
  y[n] = m;
}

int main() {

  fail<<<1, 1>>>(1000, 0, NULL);

  // This call succeeds, because the kernel launch did not have a synchronous
  // error.
  cudaError_t const error1 = cudaGetLastError();
  if(error1 != cudaSuccess) {
    printf("%s", "fail<<<1, 1>>>(1000, 0, d_y);");
  }

  // This call returns the sticky error from the kernel launch.
  cudaError_t const error2 = cudaDeviceSynchronize();
  assert(error2 == cudaSuccess);


}
```

Example 3 (Bad)

```
#include <cassert>
#include "testTerminate.h"

__global__ void fail(int* i) {
  *i = 17;
}

int main() {
  fail<<<1, 1>>>(nullptr); // null pointer results in a segmentation fault.

  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  // The assert in the kernel leads to an asynchronous error which is silently
```

(continues on next page)

(continued from previous page)

```
  // ignored.
}
```

Example 4 (Good)

```
#include <cassert>
#include "testTerminate.h"

__global__ void fail(int* i) {
  *i = 17;
}

int main() {
  fail<<<1, 1>>>(nullptr);

  // This call succeeds, because the kernel launch did not have a synchronous
  // error.
  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA 1.3 [error.interface] Check for errors after calling any CUDA library interface

When calling a CUDA library interface that returns a `cudaError_t` or `CUresult`, check whether the return value indicates that an error occurred.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Advisory.

Hardware Applicability: All Compute Capabilities.

Rationale

CUDA libraries report errors by returning numeric error codes from interfaces. These codes must be inspected by programs using CUDA to determine if an error has occurred. There are two classes of errors that an interface can return:

- Synchronous errors, which are the immediate result of the call to the interface that returned the error, and
- Asynchronous errors, which are the result of some unspecified previous asynchronous operation.

Asynchronous errors are "sticky", meaning that they will continue to be returned by subsequent calls to the CUDA runtime or driver.

Error codes report the class of error that occurred, but do not contain any information about which operation caused the error.

Because of the existence of asynchronous errors, almost any CUDA library interface can return an error, even if the interface itself cannot produce an error, because a "sticky" asynchronous error may be returned.

If a program fails to check whether an interface call returned an error, either:

- No diagnostic will be produced for the error if the error was synchronous.
- A subsequent interface call will return the error if the error was asynchronous.

This may make it unclear where the asynchronous error originated.

Example 1 (Bad)

```
#include <cassert>

int main() {
  int32_t device = -1;

  cudaDeviceProp prop = {};

  // The below call fails with `cudaErrorInvalidValue`, because `prop` is not
  // a valid device property. The returned error code is never checked, so the
  // error is silently ignored.
  cudaChooseDevice(&device, &prop);

  // Because the above call failed, `device` is still `-1`, which is not valid
  // as a parameter to `cudaSetDevice`, so the below call fails, returning code
  // `cudaErrorInvalidDevice`. This may be confusing, as the root of the error
  // is the previous failed call.
  cudaError_t const error0 = cudaSetDevice(device);
  assert(cudaSuccess == error0);
}
```

Example 2 (Good)

```
#include <cassert>

int main() {
  int32_t device = -1;

  cudaDeviceProp prop = {};

  // The below call fails with `cudaErrorInvalidValue`, because `prop` is not
  // a valid device property. The returned error code is checked, so the
  // failure is caught.
  cudaError_t const error0 = cudaChooseDevice(&device, &prop);
  assert(cudaSuccess == error0);

  // Because the above call failed, this code is never reached.
  cudaError_t const error1 = cudaSetDevice(device);
  assert(cudaSuccess == error1);
}
```

Example 3 (Bad)

```
#include <memory>
#include <cassert>
#include "testTerminate.h"
```

(continues on next page)

(continued from previous page)

```
__global__ void fail() {
  *(int*)0 = 0;
}

__global__ void pass() {}

int main() {
  fail<<<1, 1>>>(); // Fails.

  // Catch synchronous launch failures
  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  pass<<<1, 1>>>(); // Never executed because the above failure was fatal.

  cudaError_t const error1 = cudaGetLastError();
  testTerminate(error1);

  // This synchronization and any subsequent CUDA runtime or driver interface
  // calls fail with the asynchronous error `cudaErrorAssert`. The failure
  // reported by this call is silently ignored.
  cudaDeviceSynchronize();

  // This unrelated memory allocation and any subsequent CUDA runtime or
  // driver interface calls fail with the asynchronous error `cudaErrorAssert`.
  auto deleter = [] (int32_t* p) {
    cudaError_t const error0 = cudaFree(p);
    assert(cudaSuccess == error0);
  };

  int32_t* raw_u;
  cudaError_t const error2 = cudaMalloc(&raw_u, sizeof(int32_t));
  assert(cudaSuccess == error2);
  std::unique_ptr<int32_t, decltype(deleter)> u(raw_u, deleter);
}
```

Example 4 (Good)

```
#include <memory>
#include <cassert>
#include "testTerminate.h"

__global__ void fail() {
  *(int*)0 = 0;
}

__global__ void pass() {}

int main() {
  fail<<<1, 1>>>(); // Fails.

  // Catch synchronous launch failures
```

(continues on next page)

(continued from previous page)

```
  testTerminate(cudaGetLastError());

  // Never executed because the above failure was fatal.
  pass<<<1, 1>>>();

  testTerminate(cudaGetLastError());

  // This synchronization and any subsequent CUDA runtime or driver interface
  // calls fail with the asynchronous error `cudaErrorAssert`.
  cudaError_t const error2 = cudaDeviceSynchronize();
  assert(cudaSuccess == error2);

}
```

CUDA 1.4 [error.kernel-launch] Check for errors after launching a kernel by calling `cudaGetLastError`

After launching a kernel with the CUDA C++ kernel launch syntax, call `cudaGetLastError` and check whether its return value indicates that a synchronous error occurred before the next CUDA API call.

Scope: Host, Device.

Audience: CUDA C++.

Category: Advisory.

Hardware Applicability: All Compute Capabilities.

Rationale

CUDA C++ kernel launches return `void`, but reports errors by setting the CUDA global error state, which can be checked with `cudaGetLastError`. There are two classes of errors that an interface can return:

- Synchronous errors, which are the immediate result of the call to the interface that returned the error, and
- Asynchronous errors, which are the result of some unspecified previous asynchronous operation.

Asynchronous errors are "sticky", meaning that they will continue to be returned by subsequent calls to the CUDA runtime or driver.

Error codes report the class of error that occurred, but do not contain any information about which operation caused the error.

If a program fails to check whether a kernel launch returned an error, either:

- No diagnostic will be produced for the error if the error was synchronous.
- A subsequent interface call will return the error if the error was asynchronous.

This may make it unclear where the asynchronous error originated from.

Example 1 (Bad)

```
#include <cassert>
#include "testTerminate.h"

__global__ void fail() {}
```

(continues on next page)

(continued from previous page)

```
__global__ void pass() {}

int main() {
  fail<<<0, 1>>>(); // Fails synchronously due to invalid grid dimensions.
  pass<<<1, 0>>>(); // Succeeds, overwriting the previously error which was
                    // synchronous and thus is not "sticky".

  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  // This synchronization succeeds. The failure of the first kernel is silently
  // ignored.
  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

Example 2 (Good)

```
#include <cassert>
#include "testTerminate.h"

__global__ void fail() {}

__global__ void pass() {}

int main() {
  // Fails synchronously due to invalid grid dimensions.
  fail<<<0, 1>>>();

  // The kernel launch failure is detected by checking the CUDA global error
  // state.
  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  pass<<<1, 0>>>(); // Succeeds, overwriting the previously error which was
                    // synchronous and thus is not "sticky".

  cudaError_t const error1 = cudaGetLastError();
  testTerminate(error1);

  cudaError_t const error2 = cudaDeviceSynchronize();
  assert(cudaSuccess == error2);
}
```

CUDA 1.5 [cudevice.handles] Treat `CUdevice` objects as handles not integers

Treat `CUdevice` objects as handles to internal CUDA driver device objects, not integers.

Scope: Host.

Audience: CUDA C++ and Libraries.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

`CUdevice` is a pointer type, so they may be assigned integer values. In some historical cases, the CUDA driver interface has accepted `CUdevice` objects created by assigning an integer device ordinal to a `CUdevice` instead of calling `cuDeviceGet` with an integer device ordinal to obtain a `CUdevice`.

While this shortcut may work in certain circumstances, it is not guaranteed and should not be relied on.

Example 1 (Bad)

```
#include <cassert>
#include <cuda.h>

CUcontext init() {
  cuInit(0);

  // Create a context to device 0.
  CUcontext raw_context;
  CUresult const result0 = cuCtxCreate(&raw_context, 0, 0);
  // The third argument to `cuCtxCreate` should be a `CUdevice` handle to a
  // device obtained with `cuDeviceGet`, not an integer device ordinal.
  assert(result0 == CUDA_SUCCESS);
  return raw_context;
}
```

Example 2 (Good)

```
#include <memory>
#include <cassert>
#include <cuda.h>

CUcontext cudevice__good() {
  cuInit(0);

  // Get a handle to device 0.
  CUdevice device0;
  CUresult const result0 = cuDeviceGet(&device0, 0);
  assert(result0 == CUDA_SUCCESS);

  // Create a context to device 0.
  CUcontext raw_context;
  CUresult const result1 = cuCtxCreate(&raw_context, 0, device0);
  assert(result1 == CUDA_SUCCESS);

  return raw_context;
```

(continues on next page)

(continued from previous page)

```
}
```

CUDA 1.6 [fork] A call to `fork` must be immediately followed to a call to `exec`

In programs that duplicate themselves via the `fork` system call, a call to the `exec` system call must be made immediately following the call to `fork`. After the `fork` call and before the `exec` call, the following should be avoided:

- Calling any CUDA library interface.
- Using any objects residing in storage allocated by CUDA library interfaces.
- Using any managed storage duration object (e.g. `__managed__`).

Scope: Host.

Audience: CUDA C++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

CUDA does not duplicate any of its internal data structures or threads to other processes upon a call to `fork`. Any use of CUDA after a call to `fork` and before a subsequent call to `exec` has undefined behavior.

Example 1 (Bad)

```
#include <cuda_runtime.h>
#include <cassert>
#include <unistd.h>
#include <sys/wait.h>
#include "testTerminate.h"

int
main() {
  float *v_d;
  int gpucount;

  testTerminate(cudaGetDeviceCount(&gpucount));

  if (fork() == 0) {
    // all uses of CUDA before a call to exec have undefined behavior.
    testTerminate(cudaSetDevice(0));
    cudaError_t const error0 = cudaMalloc(&v_d, 1000*sizeof(float));
    assert(cudaSuccess == error0);
  }

  wait(NULL);
  return 0;
}
```

Example 2 (Good)

```
#include <cuda_runtime.h>
#include <cassert>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char **argv)
{
  int gpucount;

  cudaError_t const result0 = cudaGetDeviceCount(&gpucount);
  assert(result0 == cudaSuccess);

  if (fork() == 0) {
    execl("/bin/ls", "ls", "-l", "/tmp/kris", (char *) 0);
    assert(0);
  }
  wait(NULL);
  return 0;
}
```

CUDA 1.7 [specifiers.consistency] Be consistent with execution space and kernel function specifiers across all declarations

All declarations of a function should have the same execution space (e.g. `__host__` and `__device__`) and kernel (e.g. `__global__`) specifiers.

Scope: Host, Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Differing specifiers on different declarations of the same function can make it unclear which specifiers are applicable to the function and which are disregarded. No diagnostic is guaranteed for mismatches of execution space and kernel specifiers across different declarations. This can lead to subtle violations of the `sharing` guidelines, which can lead to undefined behavior.

Example 1 (Bad)

```
#include "testTerminate.h"
__device__ void foo();

__host__ void foo(){} // ill-formed, execution space specifiers don't match previous␣
↪declaration.

__host__ __device__ void bar();

void bar() {} // ill-formed

template <typename F>
```

(continues on next page)

(continued from previous page)

```
__global__ void invoke(F f) {
  f();
}

int main() {
  invoke<<<1,1>>>(foo); // This required to get any hits on issues.
  testTerminate(cudaGetLastError());
  testTerminate(cudaDeviceSynchronize());
}
```

Example 2 (Bad)

```
__global__ void foo();

void foo() { } // ill-formed, execution space specifiers don't match previous␣
↪declaration.
```

CUDA 1.8 [kernel-launch.parameter.value.trivial] Kernel parameters passed by value should be trivial

Any parameter passed to a kernel launch by value should have a trivial type.

Scope: Host, Device.

Audience: CUDA C++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

Kernel parameters passed by value are transferred to the device by the equivalent of `memcpy`. Thus, if a kernel parameter has a non-trivial copy constructor or default constructor, it will not be invoked, and the contents of the class will instead be effectively `memcpy`'d; this is undefined behavior.

Example 1 (Bad)

```
#include <memory>
#include <algorithm>
#include <cstring>
#include <cassert>
#include "testTerminate.h"
#include "deviceDynamicArray.h"


__global__ void assert_non_zero(device_dynamic_array<int32_t> u) {
  if (u[threadIdx.x] == 0) {
    *(int*)0 = 0;
  }
}

int main() {
  constexpr int32_t n = 128;
```

(continues on next page)

(continued from previous page)

```
  device_dynamic_array<int32_t> u(n);

  std::fill(u.begin(), u.end(), 1);

  // When `u` is passed by value, a temporary copy of `u` is created in this
  // host-thread and the representation of the temporary copy is `memcpy`ed into
  // the kernel parameters. The temporary copy is destroyed at the end of the
  // evaluation of the kernel launch; because kernel launches are asynchronous,
  // the kernel has not necessarily been run, so the destruction of the temporary
  // copy of `u` will race with the execution of the kernel.
  assert_non_zero<<<1, n>>>(u);
  // This call succeeds, because the kernel launch did not have a synchronous
  // error.
  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA 1.9 [kernel-launch.parameter.reference] Kernel parameters passed by reference should be managed storage duration objects

Any parameter passed by reference to a kernel launch made from the **host** should have managed storage duration (e.g. `__managed__` or `managed by unified virtual memory`).

Scope: Host, Device.

Audience: CUDA C++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

The host can only ODR-use host-private objects and objects with managed storage duration. Only objects with managed storage duration may be ODR-used from devices; passing a host-private object by reference to a device will lead to undefined behavior if that host-private object is ODR-used on the device.

CUDA 1.10 [preprocessing.guarded-declarations] Never use CUDA_ARCH to guard CUDA declarations

The declaration of a device variable or global function shall not be in a region of code guarded by a macro dependent on CUDA_ARCH.

Scope: Host, Device.

Audience: CUDA ++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

The `__CUDA_ARCH__` macro is defined when generating code for the GPU. Any declaration that is guarded by the macro will only be visible on the device. This can lead to missing declarations or objects with different types on the device and host. This can lead to confusion when objects do not behave as expected when data is transferred between the host and the device. This can also cause compilation problems for compilers that perform unified compilation, compiling both the host and device code in a single pass.

Example 1 (Bad)

```
#ifdef __CUDA_ARCH__
int x; // non-compliant: the declaration is only visible in the device program.
#endif

#ifdef __CUDA_ARCH__
int
#else
float
#endif
  y; // non-compliant: y has type int in device code and float in host code.
```

CUDA 1.11 [preprocessing.reserved-macros] Do not modify CUDA language specific macro names

The macro names defined in the CUDA language specification should not be undefined or redefined in the user code.

Scope: Host, Device.

Audience: CUDA C++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

The macro names defined in the CUDA language specification control the way the compiler behaves. The user should not modify these macros because this can have dangerous repercussions in the compiler.

Example 1 (Bad)

```
#undef __CUDA_ARCH__
// non-compliant: the macro name is defined by the implementation.
```

CUDA 1.12 [memory.constant_writes] Constant memory objects shall not be modified from the device program

An object in constant memory shall not be modified from the device program.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

An CUDA constant memory has special semantics allowing both the hardware and the software to make assumptions about how constant memory may or may not change. Changing constant memory from the device code may violate assumptions and result in unexpected behavior.

Example 1 (Bad)

```
__device__ void foo(int* rx) {
    *rx = 123;
}

__global__ void test() {
    __constant__ static int x;
    foo((int*)&x);
    //int& rx = x;
    //rx = 123;  // Non-compliant: assignment to a __constant__ memory location.
}
```

CUDA 1.13 [storage.thread_local] No thread_local storage duration objects in device code

The thread_local storage duration specifier should not be used in device code.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rational

The thread_local duration specifier is not supported on device variables.

Example 1 (Bad)

```
__device__ thread_local int x; // Non-compliant: device variable cannot have
                               // thread-local storage.
```

CUDA 1.14 [synchronize.active_threads] A implicit thread block synchronization function should be called by all active threads

An implicit thread block synchronization function is a synchronization function that has predefined behavior which cannot be changed by the programmer. Thee functions are defined to work on all active threads and must be called by all active threads or no threads; otherwise, the behavior is undefined.

The following are implicit thread block synchronization functions:

- `__syncthreads`, `__syncthreads_count`, `__syncthreads_and`, and `__syncthreads_or`.
- All variations of `this_thread_block().sync()`.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The thread block synchronization functions ensure that all active threads have reached a given point in the program. All global and shared memory accesses made by these threads before the synchronization are guaranteed to be visible to all of the threads in the block after the call.

Any active thread that skips the synchronization point violates the memory guarantees and may keep the call from returning.

Example 1 (Bad)

```
__global__ void kernel() {
  for(unsigned i = 0; i < blockDim.x; ++i) {
    if (i == threadIdx.x)
      __syncthreads(); // Non-compliant: synchronization function called with
                       // a CTA divergent condition.
  }
}
```

Example 2 (Bad)

```
__global__ void kernel(int flag) {
  if (flag)
    goto end; // Non-compliant: goto bypasses a synchronization function.
  __syncthreads();
end:
  return;
}
```

Example 3 (Bad)

```
__global__ void kernel(int flag) {
  if (flag)
    __syncthreads(); // Ok, condition is not CTA-divergent.
  if (blockIdx.x < flag)
    __syncthreads(); // Ok, condition is not CTA-divergent.
  if (threadIdx.x < 2)
    __syncthreads(); // Non-conformant: condition is CTA-divergent.

  __syncthreads();

  if (threadIdx.x < flag)
    __syncthreads(); // Ok, condition is potentially CTA-divergent but device-
                     // threads are either paritipating or terminating since
                     // this is the last statement of the kernel.
}
```

Example 4 (Bad)

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
__device__ void kernel(int flag) {
  thread_block g = this_thread_block();
  if (flag)
    g.sync(); // Ok, condition is not CTA-divergent.
```

(continues on next page)

(continued from previous page)

```
  if (blockIdx.x < flag)
    g.sync(); // Ok, condition is not CTA-divergent.
  if (g.thread_rank() < 2)
    g.sync(); // Nonconfromant: divergent code.

  g.sync();

  if (threadIdx.x < flag)
    g.sync(); // Ok, condition is potentially CTA-divergent but device-
              // threads are either paritipating or terminating since
              // this is the last statement of the kernel.
}
```

Collectives Rules

CUDA 2.1 [collective.warp.participants.active] Only include device-threads participating in a warp collective operation in the mask parameter

When a group of device-threads participating in a warp collective operation, the mask parameter for the operation should only include the device-threads that are participating. No inactive device-threads should be specified in the mask parameter.

Overlapping warp wide collective operations may occur.

The following are warp collective operations:

- `__syncwarp`.
- `__all_sync`, `__any_sync`, and `__ballot_sync`.
- `__match_any_sync`, and `__match_all_sync`.

The following are warp collective shuffle operations:

- `__shfl_sync`.
- `__shfl_up_sync`.
- `__shfl_down_sync`.
- `__shfl_xor_sync`.

Scope: Device.

Audience: CUDA C++.

Category: Advisory.

Hardware Applicability: All Compute Capabilities.

Rationale

A subset of the active device-threads may participate in a warp collective operation but specifying a superset of active device-threads via the mask parameter is undefined behavior, because that superset includes inactive device-threads.

The value returned by a warp wide collective shuffle operation is unspecified if the target device-thread is inactive.

Example 1 (Bad)

```
__global__ void kernel() {
  auto const idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(!(idx % 16)) { // diverge intentionally to show false positive problem
    __syncwarp(0b10000000000000001000000000000000); // non-compliant: wrong mask
  }
  if(!((blockIdx.x * blockDim.x + threadIdx.x) % 16)) { // same problem as above without␣
↪idx variable
    __syncwarp(0b10000000000000001000000000000000); // non-compliant: wrong mask
  }
  __syncwarp(~1u); // non-compliant: thread lane 0 is not in mask
}
```

Example 2 (Good)

```
__global__ void kernel() {
  __syncwarp(~0u); // compliant: all threads in the mask
}

__global__ void kernel2(int32_t *u) {
    auto const idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (!(idx % 2)) {
    u[idx] += u[idx + 1];
    __syncwarp(0b01010101010101010101010101010101);
  }
}

int main() {
  constexpr int32_t n = 32;

  auto deleter = [] (int32_t* ptr) {
    cudaError_t const error0 = cudaFree(ptr);
    assert(cudaSuccess == error0);
  };

  int32_t* raw_u = nullptr;
  cudaError_t const error1 = cudaMallocManaged(&raw_u, n * sizeof(int32_t));
  testTerminate(error1);

  std::unique_ptr<int32_t[], decltype(deleter)> up(raw_u, deleter);

  std::fill(up.get(), up.get() + n, 1);

  kernel<<<1,n>>>(); // this does not generate a false positive.
  kernel2<<<1, n>>>(up.get());  // this generates a false positive.

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  assert(cudaSuccess == error3);
```

(continues on next page)

(continued from previous page)

```
  assert(n == up[0]);
}
```

Example 2 (Bad)

```
#include <cassert>
#include "testTerminate.h"

__global__ void broadcast(int32_t u) {
  auto const idx = threadIdx.x & 31;

  int32_t v = 0;

  if (0 == idx)
    v = u;
  else
    // This `__shfl_sync`'s target device-thread (the third parameter) is the
    // first device-thread in the warp, which isn't participating in the
    // collective operation, and thus the value retrieved will be undefined.
    v = __shfl_sync(0b11111111111111111111111111111110, v, 0);

  *(int*)0 = 0;
}

int main() {
  constexpr int32_t n = 32;

  broadcast<<<1, n>>>(17);

  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

Example 4 (Good)

```
#include <cassert>
#include "testTerminate.h"

__global__ void broadcast(int32_t u) {
  auto const idx = threadIdx.x & 31;

  int32_t v = 0;

  if (0 == idx)
    v = u;

  v = __shfl_sync(0b11111111111111111111111111111111, v, 0);

  if(v != u) {
```

(continues on next page)

(continued from previous page)

```
    *(int*)0 = 0;
  }
}

int main() {
  constexpr int32_t n = 32;

  broadcast<<<1, n>>>(17);

  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA 2.2 [collective.warp.shuffle.width] Use a power of 2 width less than or equal to the warp size with warp collective shuffle operations

When a group of device-threads participate in a warp collective shuffle operation, the width parameter should be a power of 2 and less than or equal to the warp size.

The following are warp collective shuffle operations:

- `__shfl_sync`.
- `__shfl_up_sync`.
- `__shfl_down_sync`.
- `__shfl_xor_sync`.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Warp wide collective shuffle operations have undefined behavior if the width parameter is not a power of 2 or is greater than the warp size.

CUDA 2.3 [collective.warp.include_self] All involved threads should be included in the warp collective

The encountering thread should always be included in a mask when evaluating a warp level intrinsic. When reading data from another device thread in a warp-level intrinsic, the source thread must be active and included in the mask.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The mask is used to determine which threads need to arrive before execution can continue. Since all threads involved in a warp collective must use the same mask if a thread makes a call and is not in the mask the result will be undefined.

If the source thread for a shuffle sync operation is not present the result of the instruction is undefined.

Example 1 (Bad)

```
__global__ void test() {
  __syncwarp(1u); // Non-compliant: thread-lane 0 invokes the synchronization
                  // without being included in the mask.
}
```

Example 2 (Bad)

```
__global__ void kernel() {
  if (threadIdx.x == 0)
    return;
  else
    __shfl_sync(~1u, 1, /* srcLane = */ 0); // Non-compliant: source lane is
                                            // not included in the mask.
  int x = __shfl_sync(~0u, 1, /* srcLane = */ 0);
    // Non-compliant: the source thread has terminated.
}
```

CUDA 2.4 [collective.warp.in_convergence] All threads must execute the same `__syncwarp()` in convergence

In early systems all threads in the mask must execute the same `__syncwarp()` in convergence, and the union of all values in mask must be equal to the active mask. Otherwise, the behavior is undefined.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: Compute Capability 6.0 and below.

Rationale

In earlier systems the `__syncwarp()` behavior was more strictly defined. Since incorrect invocation can result in undefined behavior, the programmer most avoid invoking the call incorrectly.

Example 1 (Bad)

```
__global__ void test() {
    if(thread.idx == 0) {
      __syncwarp(~0u); // Non-compliant: call in divergent code.
    }
    else {
      __syncwarp(~1u); // Non-compliant: call in divergent code.
    }
}
```

Example 2 (Bad)

```
__global__ void test() {
    unsigned mask;
    if(thread.idx == 0) {
       mask = ~0u;
    }
    else if(thread.idx == 1) {
        return;
    }
    else {
       mask = ~1u;
    }

    __syncwarp(mask); // Non-compliant: Union of mask values is not equal to the active␣
↪mask.
}
```

Execution Space Rules

CUDA 3.1 [share.function] Use functions only in the execution spaces they target

Functions should only be executed in the execution space(s) they target.

- `__host__ __device__` functions target both the host and device execution space.
- `__host__` functions and functions with no execution space specifier target the host execution space.
- `__device__` functions and kernels (`__global__` functions) target the device execution space.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

Functions are only compiled for the architectures necessary for their execution space. `__device__` only functions may appear to have a definition in `__host__` code, however, these definitions are no-ops and are purely an implementation detail. Any ODR-use of a function in an execution space that it does not support results in undefined behavior.

Example 1 (Bad)

```
#include <memory>
#include <algorithm>
#include <vector>
#include <cassert>

struct square {
  template <typename T>
  __device__ T operator()(T t) {
    return t * t;
```

(continues on next page)

(continued from previous page)

```
  }
};

template <typename ForwardIt, typename Size, typename UnaryOp>
__host__ __device__
void inplace_transform_n(ForwardIt first, Size n, UnaryOp op) {
  for (Size i = 0; i < n; ++first, (void) ++i)
    *first = op(*first);
}

void vectorSquare(std::vector<int32_t> &u) {
  // This instantiates and uses `inplace_transform_n` in `__host__` code with a
  // `__device__` only `UnaryOp`, which is invalid and leads to a compile time
  // warning and a run time failure.
  inplace_transform_n(u.begin(), u.size(), square{});
}
```

Example 2 (Bad)

```
#include <memory>
#include <algorithm>
#include <vector>
#include <cassert>

struct square {
  template <typename T>
  __device__ T operator()(T t) {
    return t * t;
  }
};

// This template function can be instantiated and used in `__host__` code with
// a `__host__` only `UnaryOp`, but this will generate a vacuous warning about
// using a `__host__` only function in a `__host__ __device__` function.  This
// warning can be disabled with `#pragma nv_exec_check_disable`, but this also
// disables relevant warnings for any invalid instantiations in `__device__`
// code with `__host__` only `UnaryOp`s.
#pragma nv_exec_check_disable
template <typename ForwardIt, typename Size, typename UnaryOp>
__host__ __device__
void inplace_transform_n(ForwardIt first, Size n, UnaryOp op) {
  for (Size i = 0; i < n; ++first, (void) ++i)
    *first = op(*first);
}

void vectorSquare(std::vector<int32_t> &u)) {
  // This instantiates and uses `inplace_transform_n` in `__host__` code with a
  // `__device__` only `UnaryOp`, which is invalid and leads to a compile time
  // warning and a run time failure.
  inplace_transform_n(u.begin(), u.size(), square{});
}
```

Example 3 (Bad)

```
#include <memory>
#include <algorithm>
#include <cassert>
#include "testTerminate.h"

struct square {
  template <typename T>
  T operator()(T t) {
    return t * t;
  }
};

template <typename ForwardIt, typename Size, typename UnaryOp>
__host__ __device__
void inplace_transform_n(ForwardIt first, Size n, UnaryOp op) {
  for (Size i = 0; i < n; ++first, (void) ++i)
    *first = op(*first);
}

__global__ void square_each_element(int32_t* u, ptrdiff_t per_thread) {
  inplace_transform_n(u, per_thread, square{});
}

void ptrSquare(std::unique_ptr<int32_t> &up, int32_t n, int32_t m) {

  // This kernel instantiates and uses `inplace_transform_n` in `__device__`
  // code with a `__host__` only `UnaryOp`, which is invalid and leads to a
  // compile time warning and a run time failure.
  square_each_element<<<1, n>>>(up.get(), m);

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  assert(cudaSuccess == error3);
}
```

Example 4 (Bad)

```
#include <memory>
#include <algorithm>
#include <cassert>
#include "testTerminate.h"

struct square {
  template <typename T>
  T operator()(T t) {
    return t * t;
  }
};

// This template function can be instantiated and used in `__host__` code with
// a `__host__` only `UnaryOp`, but this will generate a vacuous warning about
```

(continues on next page)

(continued from previous page)

```
// using a `__host__` only function in a `__host__ __device__` function.  This
// warning can be disabled with `#pragma nv_exec_check_disable`, but this also
// disables relevant warnings for any invalid instantiations in `__device__`
// code with `__host__` only `UnaryOp`s.
#pragma nv_exec_check_disable
template <typename ForwardIt, typename Size, typename UnaryOp>
__host__ __device__
void inplace_transform_n(ForwardIt first, Size n, UnaryOp f) {
  for (Size i = 0; i < n; ++first, (void) ++i)
    *first = f(*first);
}

__global__ void square_each_element(int32_t* u, ptrdiff_t per_thread) {
  inplace_transform_n(u, per_thread, square{});
}

void ptrSquare(std::unique_ptr<int32_t> &up, int32_t n, int32_t m) {
  // This kernel instantiates and uses `inplace_transform_n` in `__device__`
  // code with a `__host__` only `UnaryOp`, which is invalid and leads to a run
  // time failure, but no compile time warning is produced.
  square_each_element<<<1, n>>>(up.get(), m);

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  assert(cudaSuccess == error3);
}
```

CUDA 3.2 [share.pointer.function] Use function pointers only on the host or device where their address was taken

Function pointers:

- Should only be ODR-used on the host or device its address was taken.
- Should not be copied between host and device or between devices.

This applies to all function pointers, regardless of execution space specifiers.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Advisory.

Hardware Applicability: All Compute Capabilities.

Rationale

A heterogeneous function (`__host__ __device__`) is compiled for the host and device architectures and thus has a different address in the host program and in each device programs. A device function (`__device__`) may have multiple addresses. Any ODR-use of a function pointer on a host or device where it was not created has undefined behavior.

Example 1 (Bad)

```cpp
#include <iostream>
#include <cassert>
#include "testTerminate.h"

void print(int32_t i) {
  std::cout << i << std::endl;
}

using element_function = void(*)(int32_t);

__global__ void for_each_thread(element_function ef) {
  auto const idx = blockIdx.x * blockDim.x + threadIdx.x;

  (*ef)(idx);
}

int main() {
  constexpr int32_t n = 32;

  // We incorrectly take the address of an implicit `__host__` function and
  // pass it to our kernel; the kernel will fail when it tries to call the
  // function pointer.
  for_each_thread<<<1, n>>>(&print);

  cudaError_t const error0 = cudaGetLastError();
  testTerminate(error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA 3.3 [share.object.usage] Only use objects in the execution space they are associated with

Supersedes:

- [**share.object.polymorphic**]
- [**share.object.host-private**]
- [**share.object.device-thread**]
- [**share.object.device-thread-block**]
- [**share.object.device-private**]
- [**share.object.stream-associated**]

As seen from a device, CPU or GPU, objects may be in one of two types of memory, accessible or inaccessible. Accessible memory is either directly connected to the device or is connected to another device that has been made available to the current device in some manner. Inaccessible memory is connected to another device and has not been made available to the current device in any manner.

Accessible memory can be obtained from multiple sources.

- Unified Virtual memory

- Managed memory
- Stream association
- device specific allocation mechanisms

Inaccessible memory is obtained by device specific allocation mechanisms that were not run on the current device.

Objects can be copied between memory types, however some objects like Polymorphic objects, are only properly defined on the device where they where created.

Objects may only be ODR-used on the devices where they are accessible and properly defined.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

Calling a virtual method of a polymorphic object on a host or device where it was not created has undefined behavior.

Calling a function via a pointer or reference on a host or device where the address was not captured has undefined behavior.

Inaccessible objects do not exist on the current device. Their addresses will either be invalid or will refer to different storage.

Accessible objects exist in the current device's memory space. Their addresses will always refer to the correct object.

Accessing a stream-attached region of memory from host-threads or device-threads outside of the stream that the memory is attached to has undefined behavior.

Example 1 (Bad)

```
#include <memory>
#include <cassert>
#include "testTerminate.h"

struct scalable {
  __host__ __device__ virtual ~scalable() {}

  __host__ __device__ virtual void scale(int32_t c) = 0;
};

struct point2d final : scalable {
  point2d(int32_t x_, int32_t y_) : x(x_), y(y_) {}

  __host__ __device__ void scale(int32_t c) override final {
    x = x * c;
    y = y * c;
  }

  int32_t x;
  int32_t y;
```

(continues on next page)

(continued from previous page)

```
};

__global__ void scale_object(scalable* s, int32_t c) {
  s->scale(2);
}

int main() {
  auto deleter = [] (scalable* ptr) {
    ptr->~scalable();
    cudaError_t const error0 = cudaFree(ptr);
    assert(cudaSuccess == error0);
  };

  scalable* raw_u = nullptr;
  cudaError_t const error1 = cudaMallocManaged(&raw_u, sizeof(point2d));
  testTerminate(error1);
  new (raw_u) point2d(2, 4);
  std::unique_ptr<scalable, decltype(deleter)> up(raw_u, deleter);

  // We incorrectly pass the address of a polymorphic object constructed on the
  // host to our kernel, which leads to a runtime error when the kernel attempts
  // to call a virtual function.
  scale_object<<<1, 1>>>(up.get(), 2);

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  assert(cudaSuccess == error3);

  point2d* raw_p = dynamic_cast<point2d*>(up.get());
  assert(raw_p != nullptr);
  assert(raw_p->x == 4 && raw_p->y == 8);
}
```

Example 2 (Bad)

```
#include <cassert>
#include <memory>

int main() {

  int32_t* raw_u = nullptr;
  cudaError_t const error1 = cudaMalloc(&raw_u, sizeof(int32_t));
  assert(cudaSuccess == error1);

  // `up` points to a device dynamic storage duration object, so accessing it
  // from the host below is undefined behavior.
  *raw_u = 42;

  cudaError_t const error2 = cudaFree(raw_u);
  assert(cudaSuccess == error2);
```

(continues on next page)

(continued from previous page)

```
}
```

Example 3 (Bad)

```
#include <cassert>
#include <memory>
#include "testTerminate.h"

__device__ int32_t const i = 42;

__global__ void address_of_i(int32_t const** u) {
  *u = &i;
}

int main() {
  int32_t const** raw_u = nullptr;
  cudaError_t const error1 = cudaMallocManaged(&raw_u, sizeof(int32_t const*));
  testTerminate(error1);

  address_of_i<<<1, 1>>>(raw_u);

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  testTerminate(error3);

  // The address of `i`, a device-private variable, is stored in `up`. Accessing
  // it here on the host has undefined behavior.
  assert(42 == **raw_u);

  cudaError_t const error4 = cudaFree(raw_u);
  testTerminate(error4);
}
```

Example 4 (Bad)

```
#include <cassert>
#include <memory>
#include "testTerminate.h"

__shared__ int32_t i;

__global__ void address_of_i(int32_t** u) {
  if (0 == threadIdx.x)
    i = 42;

  *u = &i;
}

int main() {
```

(continues on next page)

(continued from previous page)

```
  int32_t** raw_u = nullptr;
  cudaError_t const error1 = cudaMallocManaged(&raw_u, sizeof(int32_t*));
  testTerminate(error1);

  address_of_i<<<1, 1>>>(raw_u);

  cudaError_t const error2 = cudaGetLastError();
  testTerminate(error2);

  cudaError_t const error3 = cudaDeviceSynchronize();
  testTerminate(error3);

  // The address of `i`, a device-private variable, is stored in `up`. Accessing
  // it here on the host has undefined behavior.
  assert(42 == **raw_u);

  cudaError_t const error4 = cudaFree(raw_u);
  testTerminate(error4);
}
```

Example 5 (Bad)

```
__shared__ int x, *pz;
__device__ int* px, py;
__global__ void kernel(int *p) {
  int z;
  if (!px)
    px = &x;// Non-compliant because address of x escapes the shared scope
  *px += 1; // Non-compliant if evaluated by more than one block: the assignment
            // above makes the address of a thread-block shared variable
            // accessible to other thread-blocks.

  if (!pz)
    pz = &z; // Non-compliant because adress of z escapes block scope
  cudaDeviceSynchronize();
  *pz += 1; // Non-compiant if evaluated by more than one thread per block: the
            // address of a thread-private memory location is accessed by other
            // device threads.

  *p = 1; // Potentially non-compliant if the pointer is not referring to a
          // a memory location accessible by this device thread.
}

int main() {
  int w = x; // Non-compliant: read from device thread-block shared variable in host␣
↪code.
  x = 123; // Non-compliant: write to device thread-block shared variable in host code.
  kernel<<<2,2>>>(&w); // Compliant: If host stack objects are available to the device,␣
↪ATS or HMM.
                       // Non-compliant: host address passed to and used by
                       // device code.
}
```

Example 6 (Bad)

```
void foo() { }
__global__ void kernel(void(*pf)()) {
  pf();
}
int main() {
  kernel<<<1,1>>>(&foo); // Non-compliant: the kernel attempts to call the host
                         // function 'foo' through a function pointer.
}
```

CUDA 3.4 [shared.object.dynamic_arrays] Only use one `__shared__` array-of-unknown-bound declaration in a kernel function

At most one `__shared__` array-of-unknown-bound declaration shall be used during the evaluation of a kernel function.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Only one dynamic shared array base address can be given to a kernel. This means that only one dynamic shared array will be properly constructed, all others have undefined behavior.

Example 1 (Bad)

```
extern __shared__ int x[]; /* undefined */
extern __shared__ float y[]; /* undefined */
__global__ void kernel() {
  bool cmp = (void*)x == (void*)y; // Non-compliant: the kernel odr-uses both
                                   // 'x' and 'y'.
}
```

Examples 2 (Bad)

```
extern __shared__ int x[]; /* undefined */
__global__ void kernel() {
  float *y;
  y = (float *)&x[1]; // assumes sizeof(float) equals sizeof(int).
  bool cmp = (void*)x == (void*)y; // Compliant: the kernel derives 'y' from 'x'.
}
```

CUDA 3.5 [restricted_pointers] Do not create aliases between restricted pointer arguments

Do not create aliases when sending addresses into functions via restricted pointer arguments.

Scope: Device.

Audience: CUDA C++.

Category: Required.

Hardware Applicability: All Compute Capabilities.

Rationale

The `__restrict__` keyword gives the compiler leeway to perform optimizations that are only valid if the restricted pointer arguments are not aliases when they enter a function.

Example 1 (Bad)

```
__device__ void foo(int& a , int * __restrict__ b) {
  a = *b;
}

__device__ void bar(int a) {
  foo(a, &a); // non-compliant: the same object is accessed by a reference and
              // a restricted pointer.
}
```

Example 2 (Good)

```
__device__ void foo(int& a , int * b) {
  a = *b;
}

__device__ void bar(int a) {
  foo(a, &a);
}
```

Kernel Launch

CUDA 4.1 [kernel.unconfigured_call] Never invoke a kernel without a configuration

Direct or indirect calls to a `global` function shall always be configured.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Functions that have the `__global__` keyword are compiled for execution on a device and are not available on the host.

Example 1 (Bad)

```
__global__ void kernel();
int main() {
  auto pfn = &kernel;
  pfn(); // Non-compliant: call to a __global__ function without a configuration.
}
```

CUDA 4.2 [kernel.configured_call] Never invoke a non-kernel with a configuration

The postfix expression of configured calls must always refer to a `global` function.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Non `global` functions will not be properly compiled to execute on the device. Calling these functions using the postfix expression of configured calls will result in an incorrect kernel invocation.

Example 1 (Bad)

```
void not_a_kernel();
int main() {
  auto pfn = &not_a_kernel;
  pfn<<<1,1>>>(); // Non-compliant: configured call does not refer to a
                  // __global__ function.
}
```

CUDA 4.3 [kernel.virtual_kernel_parameters] No virtual arguments to kernel functions

No kernel argument or subobject of a kernel argument should have a virtual base class or virtual function.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The virtual tables (e.g. vtables) differ between the host and the device, since kernel arguments are captured on the host and sent to the device this can result in the wrong address being used to access a virtual function.

Example 1 (Bad)

```
struct S {
  virtual __host__ __device__ ~S(){};
};

__global__ void kernel(S s) {} // Non-compliant: kernel argument has a virtual
                               // function.
```

(continues on next page)

(continued from previous page)

```
struct T : virtual S {
  __host__ __device__ T(){};
};

__global__ void kernel(T t) {} // Non-compliant: kernel argument has a virtual
                               // base class.
```

CUDA 4.4 [kernel.launch_bounds] Do not vary launch bounds for a global function

The launch bounds specified on a global function shall be the same for all other declarations of the same functions in the same device program.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The implementation can only track one launch bounds for a function. Providing multiple bounds will result in an unknown setting of the values.

Example: 1 (Bad)

```
// translation unit a.cu:
__global__ void __launch_bounds__(1,1) f();

#if __CUDA_ARCH__ == 600
__global__ void __launch_bounds__(1,1) g();
#endif

// translation unit b.cu:
__global__ void __launch_bounds__(2,1) f();
  // non-conformant: the launch bounds of the function 'f' differ across
  // translation units.

#if __CUDA_ARCH__ == 700
__global__ void __launch_bounds__(2,1) g();
  // conformant: the launch bounds of the function 'g' differ across transation
  // units, but for different device programs.
#endif
```

Critical Safety CUDA rules

General Safety Rules

CUDA SAFETY 1.1 [safety.cuda.callback] Do not use CUDA callbacks

CUDA callbacks are not a part of the safety subset.

Scope: Host.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

CUDA callbacks are not supported in the safety subset.

Example 1 (Bad)

```
#include <cassert>

__host__ void callback(void*) {
  assert(0);
}

int main() {
  // Adding a callback is not supported in the CUDA safety subset.
  cudaError_t const error0 = cudaLaunchHostFunc(nullptr, &callback, nullptr);
  assert(cudaSuccess == error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA SAFETY 1.2 [safety.inline_ptx] Do not use ASM declarations in device code

Asm declarations and extended asm declarations shall not be used in device code.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

PTX ASM instructions are not supported in the safety subset.

Example 1 (Bad)

```
__global__ void kernel() {
  asm("trap;"); // Non-compliant: use of asm declaration in device code.
}
```

CUDA SAFETY 1.3 [safety.unsupported_type_operators] Device code should not contain unsupported type operators

Device code should not contain uses of dynamic_cast and typeid.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Dynamic_cast and typeid are not support in the safety subset.

Example 1 (Bad)

```
#include <typeinfo>

template<typename T, typename S> __global__ void kernel(T* t) {
  auto x = typeid(t); // Non-compliant: use of typeid operator in device code.
  auto p = dynamic_cast<S*>(t); // Non-compliant: use of dynamic_cast operator
                                // in device code.
}
```

CUDA SAFETY 1.4 [safety.unsupported_type] Device code should not contain unsupported types

The long double type shall not be used in device code.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The long double type is not supported in the CUDA safety subset.

Example 1 (Bad)

```
template<typename T>
__global__ void kernel(long double) { // non-compliant: use of long double in
                                      // kernel arguments.
  unsigned i = sizeof(long double); // non-compliant: use of long double in
                                    // a constant expression.
}

int main() {
  kernel<long double><<<1,1>>>(0); // non-compliant: instantiation of
                                   // a device function is dependent on
                                   // long double.
}
```

CUDA SAFETY 1.5 [safety.non_odr_use_host_variables] Const-qualified host variables shall only be used in valid contexts.

Let 'V' denote a namespace scope variable or a class static member variable that has const qualified type and does not have execution space annotations (for example, `__device__`, `__constant__`, `__shared__`). V is considered to be a host code variable.

The value of V may be directly used in device code, if

- V has been initialized with a constant expression before the point of use,
- the type of V is not volatile-qualified, and
- it has one of the following types:
  - built-in floating point type except when the Microsoft compiler is used as the host compiler,
  - built-in integral type.

Device source code cannot contain a reference to V or take the address of V.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Accessing host variables may exhibit unexpected behavior at runtime.

Example 1 (Bad/Good)

```
const int xxx = 10;
struct S1_t {  static const int yyy = 20; };

extern const int zzz;
const float www = 5.0;
__device__ void foo(void) {
  int local1[xxx];          // OK
  int local2[S1_t::yyy];    // OK

  int val1 = xxx;           // OK

  int val2 = S1_t::yyy;     // OK

  int val3 = zzz;           // error: zzz not initialized with constant
                            // expression at the point of use.

  const int &val3 = xxx;    // error: reference to host variable
  const int *val4 = &xxx;   // error: address of host variable
  const float val5 = www;   // OK except when the Microsoft compiler is used as
                            // the host compiler.
}
const int zzz = 20;
```

CUDA SAFETY 1.6 Constexpr host variables shall only be used in valid contexts.

Let 'V' denote a namespace scope variable or a class static member variable that has been marked constexpr and that does not have execution space annotations (e.g., `__device__`, `__constant__`, `__shared__`). V is considered to be a host code variable.

If V is of scalar type other than long double and the type is not volatile-qualified, the value of V can be directly used in device code. In addition, if V is of a non-scalar type then scalar elements of V can be used inside a constexpr `__device__` or `__host__ __device__` function, if the call to the function is a constant expression. Device source code cannot contain a reference to V or take the address of V.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Accessing host variables may exhibit unexpected behavior at runtime.

Example 1 (Bad/Good)

```
constexpr int xxx = 10;
constexpr int yyy = xxx + 4;
struct S1_t { static constexpr int qqq = 100; };

constexpr int host_arr[] = { 1, 2, 3};
```

(continues on next page)

(continued from previous page)

```
constexpr __device__ int get(int idx) { return host_arr[idx]; }

__device__ int foo(int idx) {
  int v1 = xxx + yyy + S1_t::qqq;  // OK
  const int &v2 = xxx;             // error: reference to host constexpr
                                   // variable
  const int *v3 = &xxx;            // error: address of host constexpr
                                   // variable
  const int &v4 = S1_t::qqq;       // error: reference to host constexpr
                                   // variable
  const int *v5 = &S1_t::qqq;      // error: address of host constexpr
                                   // variable

  v1 += get(2);                    // OK: 'get(2)' is a constant
                                   // expression.
  v1 += get(idx);                  // error: 'get(idx)' is not a constant
                                   // expression
  v1 += host_arr[2];               // error: 'host_arr' does not have
                                   // scalar type.
  return v1;
}
```

Memory and Execution Space Rules

CUDA SAFETY 2.1 [execution_space.deduced] Do not use incorrect deduced execution spaces

Use of deduced execution spaces should be done correctly.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

The use of deduced execution spaces may result in capturing function pointer on one device and invocation on a different device.

Example 1 (Bad)

```
struct S { int i; };
__device__ void g(S a, S b) {
  S& (S::*mfptr)(const S&) = &S::operator=; // address of host operator= may be captured␣
↪here.
  (b.*mfptr)(a); // non-compliant: impicit execution space is used for indirect
                 // call instead of direct invocation.
}
```

Kernel Launch

CUDA SAFETY 3.1 [safety.kernel.nested_configurations] Never invoke a kernel in a kernel launch configuration

No kernel should be launched while evaluating a kernel launch configuration.

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Launching a kernel while evaluating a kernel launch configuration could lead to unexpected data structure conflicts.

Example 1 (Bad)

```
__global__ void kernel(){}

int foo() {
  kernel<<<1,1>>>(); // May overwrite any kernel setup state that has been saved before␣
↪this invocation is encountered.
  return 1;
}

int bar() {
  kernel<<<foo(), 1>>>();  // invokes a kernel inside of foo()
  return 1;
}
```

CUDA SAFETY 3.2 [safety.kernel.device_side_launch] No configuration function calls in device code

No configuration function call shall be present in device code.

Scope: Device.

Audience: CUDA C++.

Category: Mandatory.

Hardware Applicability: All Compute Capabilities.

Rationale

Device side configuration function calls are not supported in safety critical CUDA.

Example 1 (Bad)

```
__global__ void kernel(){}

__device__ void foo() {
```

(continues on next page)

(continued from previous page)

```
  kernel<<<1,1>>>(); // non-compliant: kernel launch in device code.
}
```

CUDA Directives

This section is a collection of guidelines that are designed to help programmers avoid common mistakes.

General Directives

CUDA DIRECTIVE 1.1 [file.extensions] Use standardized CUDA file extensions

Use the standardized file extensions for all CUDA files. - `.cu` - CUDA source files. - `.cuh` - CUDA header files. (By convention) - `.ptx` - CUDA assembly files. - `.cubin` - CUDA device code binary files. - `.fatbin` - CUDA file that may contain multiple PTX and CUBIN files.

Scope: Device.

Audience: CUDA C++, CUDA Libraries.

Hardware Applicability: All Compute Capabilities.

Rationale

This rule adds the standardized set of file extensions to the safety subset to encourage developers to follow these conventions and to suggest that tools enforcing the rules should not complain about the use of these file extensions.

Examples

N/A

CUDA DIRECTIVE 1.2 [raii] Manage resource lifetimes with constructors and destructors

Use resource management types that acquire resources during construction and release resources upon destruction.

This pattern is commonly known as Resource Acquisition is Initialization (RAII).

Scope: Host, Device.

Audience: CUDA C++, CUDA Libraries.

Hardware Applicability: All Compute Capabilities.

Rationale

Manual resource management is unreliable. Manually managing resource lifetimes is prone to programmer error as it is easy to forget to release resources. Additionally, it is not exception safe.

RAII is a zero-cost alternative that is less error prone and operates correctly in the face of exceptions.

Example 1 (Bad)

```
#include <memory>
#include <cassert>

int main() {
  int32_t* raw_u = nullptr;
  cudaError_t const error0 = cudaMalloc(&raw_u, sizeof(int32_t));
  assert(cudaSuccess == error0);

  // If an exception or abnormal exit occurs, the resource will not be reclaimed.

  // There is no recourse if manual reclamation is forgotten.
  cudaError_t const error1 = cudaFree(raw_u);
  assert(cudaSuccess == error1);
}
```

Example 2 (Good)

```
#include <memory>
#include <cassert>

int main() {
  auto deleter = [] (int32_t* ptr) {
    cudaError_t const error0 = cudaFree(ptr);
    assert(cudaSuccess == error0);
  };

  int32_t* raw_u = nullptr;
  cudaError_t const error1 = cudaMalloc(&raw_u, sizeof(int32_t));
  assert(cudaSuccess == error1);
  std::unique_ptr<int32_t, decltype(deleter)> up(raw_u, deleter);

  // Reclamation occurs when `up` is destroyed, either at the natural end of
  // the scope or due to an exception or abnormal exit.
}
```

Example 3 (Bad)

```
#include <memory>
#include <cassert>

int main() {
  CUstream_st* raw_stream = nullptr;
  cudaError_t const error0 = cudaStreamCreate(&raw_stream);
  assert(cudaSuccess == error0);

  // If an exception or abnormal exit occurs, the resource will not be reclaimed.

  // There is no recourse if manual reclamation is forgotten.
  cudaError_t const error1 = cudaStreamDestroy(raw_stream);
  assert(cudaSuccess == error1);
}
```

Example 4 (Good)

```
#include <memory>
#include <cassert>

int main() {
  auto deleter = [] (CUstream_st* ptr) {
    cudaError_t const error0 = cudaStreamDestroy(ptr);
    assert(cudaSuccess == error0);
  };

  CUstream_st* raw_stream = nullptr;
  cudaError_t const error1 = cudaStreamCreate(&raw_stream);
  assert(cudaSuccess == error1);
  std::unique_ptr<CUstream_st, decltype(deleter)> stream(raw_stream, deleter);

  // Reclamation occurs when `stream` is destroyed, either at the natural end of
  // the scope or due to an exception or abnormal exit.
}
```

CUDA DIRECTIVE 1.3 [initiation.object.device-thread-block] Assign an initial value to device-thread-block storage duration objects prior to their use

A device-thread-block storage object (e.g. `__shared___` variable) should be initialized before its value is read.

Scope: Device.

Audience: CUDA C++.

Hardware Applicability: All Compute Capabilities.

Rationale

Variables with device-thread-block storage duration (e.g. `__shared__` variables) are uninitialized. This is a consequence of the lifetime of device-thread-block storage duration variables, which ends at the end of device-thread-block. Use of an uninitialized value is undefined behavior; thus, the a device-thread-block storage duration object must be initialized before its value is read.

Example 1 (Bad)

```
#include <cuda.h>

__global__ void bad_copy(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  d[t] = s[tr]; // s is ubd
}
```

Example 2 (Bad)

```
#include <cuda.h>

__global__ void bad_copy(int *d, int n)
```

(continues on next page)

(continued from previous page)

```
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  d[t] = s[tr]; // non-conformant: s has not been defined.
}
```

Example 3 (Good)

```
#include <cuda.h>

__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Example 4 (Good)

```
#include <cuda.h>

__global__ void bad_copy(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

CUDA DIRECTIVE 1.4 [device-dependent.callback] Do not use CUDA within CUDA callbacks

In callbacks which happen in stream order, e.g. those created via `cudaStreamAddCallback`, `cuStreamAddCallback`, `cudaLaunchHostFunc`, `cuLaunchHostFunc`, `cudaGraphAddHostNode`, or `cuGraphAddHostNode`, do not:

- Call device-dependent CUDA runtime or driver interfaces.
- Launch CUDA kernels.
- ODR-use a managed storage duration object (e.g. `__managed__`, CUDA C++ only).

Scope: Host.

Audience: CUDA C++, CUDA Libraries.

Hardware Applicability: All Compute Capabilities.

Rationale

Callbacks registered with CUDA which happen in stream order are executed in unspecified threads of execution which are owned by the CUDA runtime and driver. Calling a CUDA operation within one of these callbacks may potentially need to synchronize with the stream which the callback is in, which would lead to a deadlock, as the callback would be waiting for itself to complete. Thus, calling CUDA operations from stream-order callbacks has undefined behavior.

Example 1 (Bad)

```
#include <cassert>
#include <cuda.h>
#include <cuda_runtime_api.h>


__host__ void callback(void*) {
  // This call attempts to synchronize with all streams, but the callback
  // itself is executing in a stream, so this would deadlock. Thus,
  // `cudaErrorNotPermitted` is returned instead.
  cudaError_t const error = cudaDeviceSynchronize();
  assert(cudaSuccess == error);
}

int main() {
  cudaError_t const error0 = cudaLaunchHostFunc(nullptr, &callback, nullptr);
  assert(cudaSuccess == error0);

  cudaError_t const error1 = cudaDeviceSynchronize();
  assert(cudaSuccess == error1);
}
```

CUDA DIRECTIVE 1.5 [synchronize.callback] Callbacks should not wait for other callbacks

In callbacks which happen in stream order, e.g. those created via `cudaStreamAddCallback`, `cuStreamAddCallback`, `cudaLaunchHostFunc`, `cuLaunchHostFunc`, `cudaGraphAddHostNode`, or `cuGraphAddHostNode`, do not wait for a condition that will be satisfied by another callback.

Scope: Host.

Audience: CUDA C++, CUDA Libraries.

Hardware Applicability: All Compute Capabilities.

Rationale

Callbacks registered with CUDA which happen in stream order are executed in unspecified threads of execution which are owned by the CUDA runtime and driver. There execution may be serialized, even if there is not a mandated order between them. Therefore, a callback blocking on another callback may never complete, because the callback it is waiting for may not start executing until the blocking callback completes.

Example 1 (Bad)

```
#include <cassert>
#include <atomic>
#include <memory>
#include <cuda_runtime.h>
```

(continues on next page)

(continued from previous page)

```
auto make_stream() {
  auto deleter = [] (CUstream_st* ptr) {
    cudaError_t const error0 = cudaStreamDestroy(ptr);
    assert(cudaSuccess == error0);
  };

  CUstream_st* raw_stream = nullptr;
  cudaError_t const error1 = cudaStreamCreate(&raw_stream);
  assert(cudaSuccess == error1);

  return std::unique_ptr<CUstream_st, decltype(deleter)>(raw_stream, deleter);
}

std::atomic<bool> flag{false};

__host__ void poll(void*) {
  while (!flag.load(std::memory_order_acquire))
    ;
}

__host__ void signal(void*) {
  flag.store(true, std::memory_order_release);
}

int main() {
  auto stream0 = make_stream();
  // Callback waits for a global object to change status, which may never change
  // since the signal is in a different callback.
  cudaError_t const error0 = cudaLaunchHostFunc(stream0.get(), &poll, nullptr);
  assert(cudaSuccess == error0);

  auto stream1 = make_stream();
  // call back changes global object status.
  cudaError_t const error1 = cudaLaunchHostFunc(stream1.get(), &signal, nullptr);
  assert(cudaSuccess == error1);

  cudaError_t const error2 = cudaDeviceSynchronize();
  assert(cudaSuccess == error2);
}
```

Change Log

Changes from Version 3.0 to Version 3.0.1

- Rework CUDA Safety rule to correct reflect the CUDA programming guide.
  - *CUDA SAFETY 1.5 [safety.non_odr_use_host_variables] Const-qualified host variables shall only be used in valid contexts.*
- Added new rule.
  - *CUDA SAFETY 1.6 Constexpr host variables shall only be used in valid contexts.*

Changes from Version 2 to Version 3.0

- Minor formatting cleanup.
- Add reference links for change log.
- Added new rules.
  - *CUDA 1.10 [preprocessing.guarded-declarations] Never use CUDA_ARCH to guard CUDA declarations*
  - *CUDA 1.11 [preprocessing.reserved-macros] Do not modify CUDA language specific macro names*
  - *CUDA 1.12 [memory.constant_writes] Constant memory objects shall not be modified from the device program*
  - *CUDA 1.13 [storage.thread_local] No thread_local storage duration objects in device code*
  - *CUDA 1.14 [synchronize.active_threads] A implicit thread block synchronization function should be called by all active threads*
  - *CUDA 2.3 [collective.warp.include_self] All involved threads should be included in the warp collective*
  - *CUDA 2.4 [collective.warp.in_convergence] All threads must execute the same ___syncwarp() in convergence*
  - *CUDA 3.4 [shared.object.dynamic_arrays] Only use one ___shared___ array-of-unknown-bound declaration in a kernel function*
  - *CUDA 3.5 [restricted_pointers] Do not create aliases between restricted pointer arguments*
  - *CUDA 4.1 [kernel.unconfigured_call] Never invoke a kernel without a configuration*
  - *CUDA 4.2 [kernel.configured_call] Never invoke a non-kernel with a configuration*
  - *CUDA 4.3 [kernel.virtual_kernel_parameters] No virtual arguments to kernel functions*

- *CUDA 4.4 [kernel.launch_bounds] Do not vary launch bounds for a global function*
- *CUDA SAFETY 1.2 [safety.inline_ptx] Do not use ASM declarations in device code*
- *CUDA SAFETY 1.3 [safety.unsupported_type_operators] Device code should not contain unsupported type operators*
- *CUDA SAFETY 1.4 [safety.unsupported_type] Device code should not contain unsupported types*
- *CUDA SAFETY 1.5 [safety.non_odr_use_host_variables] Const-qualified host variables shall only be used in valid contexts.*
- *CUDA SAFETY 2.1 [execution_space.deduced] Do not use incorrect deduced execution spaces*
- *CUDA SAFETY 3.1 [safety.kernel.nested_configurations] Never invoke a kernel in a kernel launch configuration*
- *CUDA SAFETY 3.2 [safety.kernel.device_side_launch] No configuration function calls in device code*

Changes from Version 1 to Version 2

- Reorganized guidelines into sections.
  - CUDA Rules - Rules that must be followed by anyone wanting to write correct and well defined CUDA programs.
  - Critical Safety CUDA Rules - Rules that must be followed by anyone wanting to write correct Critical Safety CUDA programs. These rules may replace CUDA rules with more restrictive rules.
  - CUDA Directives : Guidelines that can help generate better code, that either should not or cannot be enforced by tools.
- Guidelines were given new stable identifiers that enhance ease of use.
- Rules were giving classifications.
- Added new Rules.
  - CUDA 3.3 [**share.object.usage**] was added to replace several nearly identical rules.
    - [**share.object.polymorphic**]
    - [**share.object.host-private**]
    - [**share.object.device-thread**]
    - [**share.object.device-thread-block**]
    - [**share.object.device-private**]
    - [**share.object.stream-associated**]
  - CUDA Safety 1.1 [**safety.cuda.callback**] was added to forbid the use of callbacks in safety critical codes.
  - CUDA DIRECTIVE 1.1 [**file.extensions**] was added to suggest users follow a defined file extension set.
- Several existing rules were moved into the CUDA Directives set, rules that should not be enforced by tools.
  - CUDA DIRECTIVE 1.2 [**raii**]
  - CUDA DIRECTIVE 1.3 [**initiation.object.device-thread-block**]

- CUDA DIRECTIVE 1.4 [**device-dependent.callback**]
- CUDA DIRECTIVE 1.5 [**synchronize.callback**]