__host__ __device__ - Generic programming in Cuda

Thomas Mejstrik (Dimetor GmbH, Vienna, Austria; University of Vienna, Austria),
Sebastian Woblistin (Dimetor GmbH, Vienna, Austria)

April 23, 2024

Abstract

We propose language changes for simpler writing of generic templated code which works both on the host and device side.

1 Motivation

When writing generic code in Cuda for both the device side (GPU), as well the host side (CPU), one faces easily a problem: Some generic code only works on one of the two sides. Yet, the Cuda language has no means of specifying for which target something shall be instantiated. This leads in the best case to (+) compiler errors, (∘) pages of compiler warnings (which then hide the important warnings), (−) a program which compiles but crashes when run, or in the worst case (−−) a program which has undefined behaviour. This proposal discusses solutions how to solve this problem.

This paper collects some ideas how to solve this problem. Additionally, it presents some ideas how to simplify writing code which shall run both on the host and the device side.

The presented code examples and the names of the identifiers are just for exposition. We do not propose a certain implementation, nor a specific set of names.

1.1 Terminology

In the following, a *stray function call* will be either when a `__host__` function calls a `__device__` function, or when a `__device__` or `__global__` function calls a `__host__` function, If a `__host__ __device__` function calls a `__host__` or `__device__` function, we say it is a *potential stray call*.

1.2 The problem T

In Listing 1 the function `wrap` is templatized for some user defined type `T` (e.g. a matrix class). In `main`, the function `func` is called with the type `H`. Thus both

Listing 1: Problem T

```
struct H {
    __host__ int value() { return 3; }
};

struct D {
    __device__ int value() { return 2; }
};

template< typename T >
__host__ __device__
int wrap() {
    return T{}.value();
}

int main() {
//  return D{}.call();  // error
//  return wrap< D >(); // no warning, UB
    return wrap< H >(); // warning
}
```

the `__host__` and the `__device__` version of `func` is instantiated with the type `H`. But, `H`'s function `call` is only a `__host__` function, and thus, when compiled with *nvcc* we get the warning: *"calling a __host__ function ("H::call()") from a __host__ __device__ function ("func< ::H> ") is not allowed"*.

When we would instead call `func< D >()` we provoke undefined behaviour.[1] When we would call `D.call()` nvcc 12 gives a compilation error. We call this problem as *Problem T*.

We can observe two things here:

- nvcc 12 handles stray function calls inconsistently.
- It is hard to write generic Cuda code which works both on the CPU and the GPU.

2 Current patterns to solve the problem

In this section we present patterns which solve problem T, or the problem of porting existing code to Cuda. We also discuss why these solutions are not satisfactory.

[1] a) On our test system, the value 1 is returned (instead of 2). b) Unfortunately, *nvcc* (versions 9 to 12) does not emit any compiler warning for this example.

Listing 2: Solution: `__host__ __device__`

```
struct S {
  __host__ __device__
  static void value() {}
};

template< typename T >
__host__ __device__
void func() {
  T::value();
}

int main() {
    func< S >();
}
```

2.1 `__host__ __device__` everything

2.1.1 Context

Code shall be useable with Cuda and non Cuda compilers, and the code does not contain any Cuda specific stuff.

2.1.2 Solution

We add `__host__ __device__` annotations to all functions.

2.1.3 Assessment

– Not always possible. The host and device implementation may be different or there may not be any implementation for either host or device code.

2.2 `#ifdef` blocks with `__CUDA_ARCH__`

2.2.1 Context

Code shall be useable with Cuda and non-Cuda compilers, but the necessary implementations differ.

2.2.2 Solution

Use the preprocessor macro `__CUDA_ARCH__` to determine whether we are in device or host code. If there is no sensible implementation for either host or device code, we abort the program whenever we end up in the wrong path, see Listing 3. In that listing, `S::value()` is (wrongly) called in device code, and thus a runtime error on the GPU is triggered.

Listing 3: Solution: `#ifdef` block with `__CUDA_ARCH__`

```
struct H {
    __host__ __device__      // actually only a __host__ implementation,
    static void value() {    // but annotated with __host__ __device__
        #ifdef CUDA_ARCH     // to silence compiler warnings
        __trap();
        #endif
        /* body */
    }
};

template< typename T >
__global__
void kernel( T t ) { t.value(); }

int main() {
    kernel<<< 1, 1 >>>( H{} );
    return cudaDeviceSynchronize();
}
```

2.2.3 Assessment

+ Easy to understand and use
– Source code is cluttered with preprocessor directives.
– Check for stray function calls is at runtime, not at compile time.

2.3 Pragmas `#hd_warning_disable` and `#nv_exec_check_disable`

2.3.1 Context

One “knows” that a certain code path is not possible, and thus just wants to disable compiler warnings.

2.3.2 Solution

The pragmas `#hd_warning_disable` and `#nv_exec_check_disable` can be used.

2.3.3 Assessment

+ Easy to use
○ Each function has to be annotated manually.
– These pragmas are undocumented

Listing 4: Solution: `#pragma`

```
struct S {
  static void value() {}
};

#pragma hd_warning_disable
template< typename T >
__host__ __device__
void func() { T::value(); }

int main() {
    func< S >();
}
```

- Wrong usage of these pragmas may lead to wrongly compiled code [1]
- May hide programming errors. A combination of using these pragmas with `release_assert` is thus recommended.
- Even when one "knows" that the pragmas can be used at the time when the code is written, things may change in the future.

2.3.4 Known Usages

- Thrust
- Eigen

2.4 Experimental relaxed constexpr

2.4.1 Context

One has a function which is, or can be made, `constexpr` and one compiles with *nvcc*.

2.4.2 Solution

We decorate the function with `constexpr` and compile with the option `--expt-relaxed-constexpr`. This allows device code to invoke `__host__ constexpr` functions, and host code to invoke `__device__ constexpr` functions, see Listing 5.

We assert whether the source code is compiled with `--expt-relaxed-constexpr` by checking whether the macro `__CUDACC_RELAXED_CONSTEXPR__` is defined, and produce a compilation error when not. This way, the user is informed how to correctly compile the program, when she attempts to compile it wrongly.

Listing 5: Solution: Experimental relaxed `constexpr`

```
#ifndef __CUDACC_RELAXED_CONSTEXPR__
#error "Must be compiled with:" \
    "--expt-relaxed-constexpr"
#endif

struct S {
    constexpr static int value() {
        return 42;
    }
};

template< typename T >
__global__
void kernel( T t ) {
    printf( "%i", t.value() );
}

int main() {
    kernel<<< 1, 1 >>>( S{} );
    return cudaDeviceSynchronize();
}
```

2.4.3 Assessment

++ Is also applicable to third party `constexpr` function, e.g. functions in the C++ standard library

\+ Easy to use.

\+ Needs minimal changes to the source code.

– Only applicable to `constexpr` functions.

– Is an experimental feature: It is experimental since at least Cuda 8.0 from 2016. The behaviour of this option may change in future Cuda releases.

– The source code is not self contained any more, but needs to be compiled with certain compiler flags.

– It is unclear, whether this feature is compatible with future C++ versions.[2]

– Only works with *nvcc*.

[2] Currently, Each new C++ standard softens the restrictions to `constexpr` functions E.g., C++20 allows memory allocations in `constexpr` functions.

2.4.4 Known Usages

- LBANN [2] uses a defensive strategy: Is the source compiled with `--expt-relaxed-constexpr` they annotate functions with `constexpr`, otherwise they annotate them with `__host__ __device__`.

2.5 SFINAE

This is the pattern which solves the problem mostly. But it involves ugly macros, and thus has some severe drawbacks.

2.5.1 Context

We want to make sure at *compile time* that a function, which is either usable on the host or device side, is only called on the host or device side.

2.5.2 Solution

We define three versions of the same function – a `__host__`, a `__device__`, and a `__host__ __device__` function – and make it such that always only one (the one that we need) is well formed, and thus can be called. In particular, the compiler cannot instantiate a wrong function and we do not get compiler warnings. If we would try to do a stray function call we get a compilation error.

See Listing 6 which uses the trait `hdc` (see Section 3.1.1) to determine which function needs to be called (for the macro `DEPAREN` see Listing 7).

2.5.3 Assessment

+ Easy to use
– Not straight forward to write and understand, and thus hard to maintain in the long term.
– Code is not debuggable, since the body of the function is inside of a macro.[3]
– Only works for template functions or functions of template classes.
– Only works when we can define the variable `hdc` in a sensible way, this means effectively in a class.
– Code duplication. This is mitigated by the use of macros
– Code bloat and thus increased compilation times.

[3] We devised a version with a trampoline function, even more macro stuff, some boiler plate code, and the use of the pragma `#nv_exec_check_disable` to solve this problem. But we refrain from presenting it here, since it is just an abuse of the language.

Listing 6: host device macro 3

```
template< typename T > static constexpr
HD hdc = hdc_impl< T >::hdc;

#define host_device_macro3( templateargs, hdc_, body )
\
    template< DEPAREN(templateargs), HDC hdc = DEPAREN(hdc_) >
\
        requires( hdc == HDC::Hst) >
\
    __host__ DEPAREN(body)
\
                                                                \
    template< DEPAREN(templateargs), HDC hdc = DEPAREN(hdc_) >
\
        requires( hdc == HDC::Dev) >
\
    __device__ DEPAREN(body)
\
                                                                \
    template< DEPAREN(templateargs), HDC hdc = DEPAREN(hdc_) >
\
        requires( hdc == HDC::HstDev ) >
\
    __host__ __device__ DEPAREN(body)

// Usage:
host_device_macro3( (typename T),
                    (hdc< T >),
                    (void f3( T t ) {}) )

struct HD {
  static constexpr HDC hdc = HDC::HstDev;
};

void g3() {
  f3( 0 );      // calls host version
  f3( HD{} );   // calls device version
}
```

Listing 7: DEPAREN

```
// helper macro to remove parentheses from macro arguments.
#define DEPAREN( X ) ESC( ISH X )
#define ISH( ... )   ISH __VA_ARGS__
#define ESC( ... )   ESC_( __VA_ARGS__ )
#define ESC_( ... )  VAN ## __VA_ARGS__
#define VANISH

// usage example
DEPAREN( (int) ) a;  // gives: int a;
DEPAREN( int ) b;    // gives: int b;
```

– nvcc 12 still has problems with `requires` clauses.[4] Thus currently, this pattern works only with C++11 SFINAE clauses (which make it even more ugly), and which do not work for non-template functions (like copy constructors).

2.6 Summary

From the presented patterns we infer that, there is none solution which

+ is debuggable
+ is easy to write and use
+ does not need code duplication
+ does not need lots of boiler plate code
+ is applicable to third party code
+ is easily applied to an existing code base
+ is easy to understand
+ does not use the preprocessor
+ does all stray function call checks at compile time
+ does not use undocumented features of Cuda
+ does not use experimental features of Cuda
+ does not need special compiler flags
+ does not provoke UB when used incorrectly

[4] See open bug report **XX** **id** .

3 Proposals of language changes

We now propose changes to the Cuda language to solve the problem. Most of the proposals are mutually independent from each other (in other words: orthogonal).

3.1 Proposal 1: Conditional `__host__ __device__`

This solution is inspired by pattern SFINAE. The solution partly solves the problem, needs minimal language changes, and does not introduce any breaking change to the language.

3.1.1 Host Device compatibility trait `hdc`

`hdc< T >` inspects for the given type `T`, whether a member variable of name `T::hdc` is present. If so, then `hdc< T >` returns its value. Otherwise it returns `HDC::Hst`, see Listing 8.

Controversies

- It is unclear whether `hdc` for fundamental types (`int`, pointers, ...) should return `HDC::HstDev` or `HDC::Hst`.
- It is unclear whether this trait should be provided by the cuda library, or should be written by the user.

3.1.2 Conditional `__host__ __device__`

We propose that `__device__` (and `__host__`) annotations accept a boolean parameter. If a parameter is given , and it is `false`, then the function is not compiled for `__host__` (or `__device__`).[5] See Listing 9 how to use it to solve our Problem T from the beginning.

3.1.3 Assessment

+ This is a pure language extension, such syntax is ill-formed today, and no existing code will break.
+ Easy to use.

3.2 Proposal 2: More versatile `__host__ __device__` decorators

This does not solve the problem, but allows to easily port code to Cuda.

[5] A similar approach is taken in C++ with the `noexcept`, or the `explicit` (C++20) keyword, which also takes an optional boolean argument.

Listing 8: Host Device compatibility: `hdc`

```cpp
#include <type_traits>

template< typename T >
concept has_hdc = requires( T t ) { &T::hdc; };

enum class HDC {
    Hst, Dev, HstDev
};

template< typename T >
struct hdc_impl {
    struct h {
        static constexpr HDC hdc = HD::Hst;
    };
    static constexpr HDC hdc =
        std::conditional_t< has_hdc<T>, T, h >::hdc;
};

template< typename T > static constexpr
HD hdc = hdc_impl< T >::hdc;

// Usage example
struct S {};
struct D {
    static constexpr HDC hdc = HDC::Dev;
};

static_assert( hdc<S> == HDC::Hst );
static_assert( hdc<D> == HDC::Dev );
```

Listing 9: Proposal 1

```
struct H {
    __host__ void call() {}
};

struct D {
    HDC hdc = HDC::Dev;
    __device__ void call() {}
};

template< typename T >
__host__( hdc<T> == Hst::Hst ) __device__( hdc<T> == Hst::Dev )
void wrap() {
    T{}.call();
}

int main() {
//  wrap< D >(); // error
    wrap< H >(); // OK
}
```

3.2.1 Allowing `__host__ __device__` decorations for classes and namespaces

If a class (or namespace) is decorated with `__device__` (or `__host__ __device__`), then all its undecorated member functions become automatically `__device__` (or `__host__ __device__`) member functions.

If a function cannot be implicitly decorated, then the compilation fails.

For example, the two classes `S1` and `S2` in Listing 10 are equivalent.

Controversies

- Listing 10, one could also argue that $S1$ :: *init* should be `__host__ __device__` function. But we think that the `__host__` annotations is spatially

Listing 10: `__host__ __device__` decorations for classes and namespaces

```
__device__ class S1 {
    void call();
    __host__ void init();
};
class S2 {
    __device__ void call();
    __host__ void init();
};
```

nearer to the function declaration, and thus takes precedence over `__device__`. Furthermore, if the `__host__` and `__device__` annotations would add up, it would not be possible to define a pure `__host__` function inside of class `S1` any more. This would obviously be a bad design decision, and thus users will not think that the language behaves in this way.

- One could argue, that `__global__` annotations should also be allowed before a `class` or a `namespace`. But, kernels play are very special role in Cuda programming, so we do not see a need for that feature currently For dynamic parallelism one usually needs to rewrite a lot of code until it works performant, and thus this is not a useful feature there either.
- It is unclear, whether `__host__ __device__` annotations should be allowed in forward declarations of classes or not, and if they are allowed whether they should be ignored or not.

Assessment

+ This is a pure language extension, such syntax is ill-formed today, and no existing code will break.
+ Allowing decorations on classes allows to port code more easily to Cuda.
– Apart from the syntactic problems described in *Controversies*, we could not find any downsides so far.

3.3 Proposal 3: Propagation of `__host__ __device__`

This proposal solves our problem, but also needs more language changes. It also imposes more restrictions to the programming style. In particular, a lot of functions must go to a header file.

3.3.1 Including `__host__ __device__` in the signature of the function

We propose that the function execution space specifiers are part of the signature. Then, one can get rid of using the `__CUDA_ARCH__` macro in cases where the host and device part need different code, see Listing 11.

The function `g` calls `f/*(1)*/` when `g` is compiled for `__host__` code, and calls `f/*(2)*/` when `g` is compiled for `__device__` code.[6]

Assessment

+ Adding the `__host__ __device__` decorators to the function signature simplifies learning Cuda. Often novices of Cuda/C++ think that the `__host__ __device__` annotations are part of the function signature.

[6] For the *nvcc* compiler the `__host__`, `__device__` annotations are not part of the function signature; whereas clang considers them to be part of the signature.

Listing 11: `__host__ __device__` as part of the function signature

```
__host__ void f/*(1)*/() {}
__device__ void f/*(2)*/() {}

__host__ __device__ void g() {
    func();
}
```

– Errors may not be detectable until link time (as it is now).

– Is an ABI break.

3.3.2 Propagation of `__host__ __device__`

If a `__device__` or `__global__`, (or `__host__ __device__`) function calls an undecorated function (or an undecorated member function in an undecorated class), it becomes automatically a `__device__` (or `__host__ __device__`) function. If a `__device__`, or `__global__`, (or `__host__ __device__`) function calls an undecorated templated (member) function, then this function is instantiated as a `__device__` (or `__host__ __device__`) function.

This implies,

(1) that the `__host__` and `__device__` annotations need to be part of the function signature, (otherwise the compiler cannot decide which function to call), and

(2) that the definition of the function must be present to compiler whenever the function is invoked (otherwise, the compiler does not know which functions shall be instantiated and a linker error occurs). Thus, most functions must go to header files. To mitigate this shortcoming, one could allow *explicit `__host__` and `__device__` instantiations.*[7]

Assessment

+ Needs no further work from the user.

+ Is applicable to third party code, in particular everything in `std::`.

+ Functions for which the execution space shall not get deduced are still possible, by explicitly adding annotations.

+ Speeds up compilation times, since less functions need to be compiled

∘ If an already existing program has stray function calls, this proposal may change its behaviour. For correct programs, this proposal does not change the behaviour.

[7] The explicit instantiation would be similar to explicit C++ template function instantiations.

- Functions then behave similar to template functions with all implications. This is not that of a big deal, since this feature is mostly necessary for template functions anyway.
- Needs that `__host__ __device__` annotations must be part of the function signatures with all implications.

– Could lead to bad user code, since users could assume that *just because it works* it is also efficient, whereas an efficient implementation of a function often is different on the GPU or CPU side.

3.4 Proposal 4: Strict handling of stray calls

This does not solve the problem, but leads to better compiler warnings/errors, and thus to a better developing experience.

3.4.1 Forbid stray function calls

We propose that the compiler shall handle stray function calls consistently and strictly. In particular, every stray call (`__host__` to a `__device__` function or `__device__` to a `__host__` function) shall be a compiler error. Every potential stray call (`__host__ __device__` to a `__host__` or `__device__`) shall be at least a compiler warning.[8]

Controversies It is unclear what to do with potential stray calls. Emitting a compiler warning feels like the most natural solution.

Another solution would be to make it to a compiler error, but provide two pragmas to lower the severity to a warning, or to disable the error/warning.

Assessment

\+ Consistent behaviour regarding stray function calls.

3.4.2 Solution

See Listing 12 how to use this proposal to solve our Problem T from the beginning.

[8] The authors are wondering, whether the inconsistencies in the stray function call handling (see Listing 1) are due to a special handling of cases of templated `__host__ __device__` functions, (which easily produce such warnings). This seems likely, because, if each stray function call were a compilation error, it would be nearly impossible to write generic Cuda code.

With this proposal, templated `__host__ __device__` functions can easily and cleanly be implemented. Special handling would then not be required any more, and thus the compiler then could more easily identify stray calls.

Listing 12: Proposal 2

```
struct S {
    void call() {}
};
__host__ struct H {
    void call() {}
};
__device__ struct D {
    HDC hdc = HDC::Dev;
    void call() {}
};

template< typename T >
void wrap() {
    return T{}.call();
}

__global__ void kernel() {
    wrap< S >(); // OK
    wrap< H >(); // error
}

int main() {
//  wrap< D >(); // error
    wrap< H >(); // OK
}
```

References

[1] konstantin_a, *#pragma hd_warning_disable causes nvcc to generate incorrect code (cuda 9.1).*, forums.developer.nvidia.com/t/57755.

[2] LLNL, *LBANN: Livermore Big Artificial Neural Network Toolkit*, github.com/LLNL/lbann.