`__host__ __device__` - Generic programming in Cuda

THOMAS MEJSTRIK

Writing templated functions in Cuda/C++ both for the CPU and the GPU bears the problem that in general always both `__host__` and `__device__` functions are instantiated. This easily leads to silently broken code, either on the host or device side. This paper presents patterns to solve this problem. XX **needs work**

ACM Reference Format:
Thomas Mejstrik. 2024. `__host__ __device__` - Generic programming in Cuda. 1, 1 (April 2024), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

XX **check code examples whether they compile before submitting finally**

1.1 Introduction

When writing template functions in Cuda/C++ which shall work for both the device side (GPU), as well the host side (CPU), one easily faces the following problem: Some generic code only works on one of the two sides. Yet, the Cuda language has no means of specifying for which side something shall be instantiated. This leads in the best case to compiler errors, or worse pages of compiler warnings (which then hide the important warnings), or a program which compiles but crashes on execution run, or in the worst case a program which runs and malfunctions.

This paper describes patterns to solve the problem for *templated functions and member functions*. We shortly discuss lambdas, but since Cuda has no good support for templated lambdas yet, they are of not much use to solve our problem. We do not discuss how constants can be used both in host and device code efficiently, since this is totally different topic. XX **Motivation is currently an abstract, and does not motivate much.**

Our motivating example is presented in Listing 1. There, a function is templatized for some user defined type `T` (e.g. a matrix class). In `main`, the function `func` is called with the type `H`. Although, the call originates from the host side, according to the Cuda specification both the `__host__` and the `__device__` version of `wrap` must be instantiated with the type `H`. `H`'s function `func` is only a `__host__` function. Thus the compiler notices that there is a possible stray function call, and emits a warning. *nvcc* 12 for example tells us: "`calling a __host__ function ("H::func()") from a __host__ __device__ function ("wrap< ::H> ") is not allowed`".

Note though that the program is well-formed and does what is expected — it returns `3`. Things are different, when we would instead call `func< D >()` (the line marked with `//UB`). In that case we would have really a stray function call. Unfortunately, *nvcc* still compiles this example (strangely even without a warning![1]but at runtime the result is undefined.[2]

In view of the problems described so far we would like to have

- a compilation error whenever there is a stray function call
- no compilation warnings due to wrongly instantiated function templates.

[1]A bug report was sent to Nvidia concerning this example, see Bug No. 4196685. So hopefully, *nvcc* will warn in such cases too in future.
[2]On the test system the program returned 1, instead of 2.

Author's address: Thomas Mejstrik.

Listing 1: Problem T

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

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

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

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

Readers not familiar with some intricacies of C++ or Cuda, are referred to Appendix B.

1.2 Motivation

The usecase described here, is often not on the radar of Cuda developers. Indeed, graphic cards and CPUs differ heavily in some regards. We list only some of those: CPUs only have a handful of cores nowadays whereas GPUs have thousands of them, CPUs are designed for low latency, GPUs are designed for high throughput. CPUs can handle branching quite well in contrast to GPUs, GPUs often work fastest when numbers with reduced accuracy (single / half float) are used, the memory bandwidth of graphic cards is often much much higher than that of the RAM on the host side, etc.. Thus, the problems which can efficiently be solved on a GPU differ, and thus code is often not designed to be portable between CPUs and GPUs, although it may make sense.

Yet, there are cases where it makes sense to write code which can work on both sides. One is for so-called *Embarrassingly parallel* problems. These are problems where little or no effort is needed to port a single threaded program to a multi threaded program. Such programs often are not fully suited for GPUs (e.g. they contain branches), but neither are suited for a CPU (because they lack cores). Another case is for problems which require a high memory bandwidth. XX **Search for Youtube talk**

Then there is the aspect of the available hardware of the customers. A user, which could be a developer or a customer, may not always have a compatible GPU at hand. For example for a presentation, or when the developer wants to work on the road on an old laptop.

Then there is also the aspect of developer experience. Debugging for example is much easier on the CPU than on a GPU. XX **Work on that part**

1.3 Notation and Targeted Systems

We will refer to code in `__host__` functions as *host code*, and to code in `__device__` and `__global__` functions as *device code*. We will not discuss `__global__` functions, as for the considerations of this paper, they behave the same as `__device__` functions.

We try to only use standard Cuda terminology in the paper, except for the following: We say a function call is a *stray call* if it originates (*a*) from a `__host__` (or `__host__ __device__`) function and calls a `__device__` function, or originates (*b*) from a `__device__` (or `__host__ __device__`) function and calls a `__host__` function.

Condensed code examples. In order to save space in the listings, we use a style not recommended for production code. In particular we use (very) short identifiers, omit the inclusion of headers (e.g. `<cassert>`) or headers of functions described in this paper (e.g. `"release_assert.h"`). Types which work on the host side are usually named `'H'`, types for the device side with `'D'`.

Targeted systems. We target the C++ versions 14, 17, and 20, as well as *nvcc* versions above and including 9.2[3]

We target multiple C++ versions because often (Cuda) developers do not have access to the latest C++ version, the Cuda standard lacks behind the C++ standard usually for years and/or (Cuda) compilers may have bugs.

2 THE PATTERNS

We presents eight patterns, which we deeom to be a good way to tackle the above stated, or similar, problems. Patterns and solutions which we deem to be not good are only discussed very shortly. There are different reasons why a solution can be deemed not to be good. Most of the times, this will be to a poor support of the C++ standard from Cuda compilers, or shortcomings in the Cuda language itself. These points would make some solutions unnecessary long, complicated, hard to maintain and thus bug prone. XX **Paragraph is clumsy**

2.1 Pattern: Host device everything

2.1.1 Context. You have some code without any Cuda specific language constructs. The code shall be useable both with Cuda and non-Cuda compilers, and both on the host and device side. The code (its implementation) makes sense on both on the host and device side.

2.1.2 Problem. The code does not compile under *nvcc* or the host compiler, or the code is not usable on the host or device side yet.

2.1.3 Solution. We make all functions to `__host__ __device__` functions when compiled with a Cuda compiler. When compiled with a host compiler, the functions shall have no annotations (and thus be `__host__` functions).

2.1.4 Example implementation. We annotate all of our functions with user defined preprocessor macros, which expand to `__host__ __device__` when compiled with *nvcc*, and which expand to nothing when compiled with a non-Cuda compiler, see Listing 3.[4]

2.1.5 Pitfalls to Avoid.

[3]All examples are thoroughly tested with *gcc* 11.3, *clang* 14.0 and *nvcc* 12.1. For older compiler versions *godbolt* [28] was used. We did not try to compile the Cuda code with *clang* alone, since there are notable differences between the Cuda dialect *clang* uses. [1]

[4]It is allowed by the standard to omit the `return` statement in the `main` function.

Listing 2: Easy, but *UB*, `__host__ __device__` macros

```
// cudatags_ub.h
#ifndef CUDATAGS_B
  #define CUDATAGS_B
  #ifndef __CUDACC__
    #define __host__
    #define __device__
  #endif
#endif

// Example: function definition
__host__ __device__ void func1() {}
```

Implementation differences between host and device code. In most cases, non-templated functions can directly ported to Cuda just by adding the `__device__` keyword to its declaration. Yet, this may not be the optimal approach, since the GPU and CPU may need different implementation, in order to have a "good" implementation. XX **reword**

Macro names. One has to consider the naming rules [30, global.names] in C/C++ when naming the bespoken macros. Using a forbidden identifier leads straigt to undefined behaviour, yet can have some unmatched advantages in this case.

Using an "allowed" macro name like `CUDA_HOST` and `CUDA_DEVICE`, has several disadvantages: Other developers need to know the macro name, the global namespace gets clobbered with names, and the code gets cluttered with non-standard keywords. See Listing **??** for this solution.

If one decides to use a "forbidden" macro name, its best, and safest, to boldly call them again `__host__` and `__device__`. In this case, they must only be defined when the code is compiled with a non-Cuda compiler. To decide whether the Cuda specific macros need to be defined one can check the `__CUDACC__` macro. This macro is only defined when *nvcc* steers the compilation process of Cuda source files. In particular, the macro is still defined after *nvcc* passes control over to the host compiler (which than compiles the preprocessed source files by *nvcc* without any Cuda specific language constructs). XX **rephrase** The big plus of it is, that it does not require any additional knowledge by other developers, and it may be also be applicable to third party Cuda code. Unfortunately, this name incurs *UB* since identifiers starting with two underscores are reserved to be used by the compiler. Yet, it is very unlikely that Solution B will break any code. Indeed, if a compiler shall be a compatible host compiler for *nvcc*, then the identifiers `__host__`, `__device__`, etc... must not be used by the host compiler. Thus, a potential host compiler will not use these identifiers for its own purposes. Furthermore, legacy code has lots of identifiers starting with two underscores. So, a compiler will do nothing bad to code with such identifiers whenever possible.

See Listing 2 for this solution.

2.1.6 *Assessment.*

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

2.1.7 *Known Usages.*

- Boost/Utility [31] uses a combination

Listing 3: Host device everything

```
#include "cudatags_ub.h"

struct HD {
  __host__ __device__
  static void value() {}
};

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

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

```
#define BOOST_GPU_ENABLED \
    __host__ __device__
```

- Eigen [32] uses a variant

```
#ifdef /* some Eigen macro */
  #define EIGEN_DEVICE_FUNC \
    __host__
#else
  #define EIGEN_DEVICE_FUNC \
    __host__ __device__
#endif
```

- Dimetor [35] uses a solution similar to the code in Listing 2.

2.2 Pattern: Conditional function body

2.2.1 Context. You have some "functionality" which shall be useable both with Cuda and non-Cuda compilers, both on the host and device side. To implement the "functionality", you need to use host or device side specific constructs or the necessary implementations differ.

2.2.2 Problem. XX **Symptom: suboptimal implementations (functions), code duplications**

2.2.3 Solution. Use preprocessor macros to determine whether we are in device or host code, and to guide the compilation process.

Listing 4: Conditional compilation: release_assert

```
#include <cstdlib>

#ifdef __CUDA_ARCH__
static constexpr bool cuda_arch = true;
#else
static constexpr bool cuda_arch = false;
#endif

__host__ __device__
void release_assert( bool flag ) {
  if( !flag ) {
    #ifdef __CUDA_ARCH__
    __trap();
    #else
    std::abort();
    #endif
  }
}
```

2.2.4 Example implementation. See Listing 4. In order to make different function bodies for host and device code, we use the macro `__CUDA_ARCH__`. `__CUDA_ARCH__` is defined when device code is compiled, i.e. after *nvcc* passes control over to the host compiler `__CUDA_ARCH__` is not defined.[5]

Listing 4 presents the function `release_assert`, taking an argument of type `bool`. If the argument is `false` then a runtime error is triggered. In device code this is done by calling `__trap`, which aborts the kernel execution. in host code this is done by calling `std::abort`. Note that, in production code one may want to use less severe means, and/or want to print out debug info to `stderr` beforehand [13, 14].

2.2.5 Assessment.

+ Easy to understand and use
+ Given that there is both a sensible host and a device implementation, this solution is often the last resort.
∘ Source code is cluttered with preprocessor directives. which is usually not a problem when the function body is short. Otherwise, the patterns described in [26] may help.
–

2.2.6 Known usages.

- Eigen

[5]See Appendix B for the compilation process.

Listing 5: Wrong solution for Conditional compilation

```
#include "cudatags.h"

struct H {
  __host__
  static void value() {}
};

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

int main() {
    #ifndef __CUDA_ARCH__  //
    func< H >();           // UB
    #endif                 //
}
```

```
static float infinity() {
  #if defined(EIGEN_CUDA_ARCH)
    return CUDART_INF_F;
  #else
    return HIPRT_INF_F;
  #endif
}
```

- Most likely in every larger Cuda codebase

2.2.7 Pitfalls to Avoid. There are some restrictions on how `__CUDA_ARCH__` can be used. For this papers scope, the most important are: *The signature of functions, function templates and instantiated function templates, as well as the arguments used to instantiate function templates must not depend on whether* `__CUDA_ARCH__` *is defined or not* [5]. This implies that the solution in Listing 5 is not a valid one, since `func< H >` is only instantiated when `__CUDA_ARCH__` is undefined.[6]

One may be tempted to use `if constexpr` (introduced in C++17) to get rid of the preprocessor. Unfortunately, this is a bad idea due to C++ rules. `if constexpr` works only in templated functions, and for arguments which depend on the functions template types (so-called dependent types). If `if constexpr` is used differently one faces undefined behaviour.[stmt.if][30]

2.2.8 Notes. The patterns Host device everything and Conditional function body are related. The difference is, that the former one only makes changes to the function declaration, whereas the latter one additionally introduces changes in the function body.

[6]Although the program in Listing 5 is not a valid one, nvcc 12.1 gives no warning when compiling it.

Listing 6: CONSTEXPR EVERYTHING

```
#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();
}
```

2.3 Pattern: CONSTEXPR EVERYTHING

2.3.1 Context. You have some "functionality" which shall be be compilable both with Cuda and non-Cuda compilers, and both on the host and device side. Its implementation makes sense both on the host and the device side. Furthermore, the "functionality" is implemented as a `constexpr` function and no code changes can be made to the function, e.g. it is a third party function, and you compile with *nvcc*.

2.3.2 Problem. The code currently does not compile, or the compiler spits out compiler warnings, or it is not known whether stray fubnction calls happen.

2.3.3 Solution. Since the function is decorated with `constexpr`, we add the compilation option `--expt-relaxed-constexpr` [6]. This allows device code to invoke `__host__ constexpr` functions, and host code to invoke `__device__ constexpr` functions.

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, see Listing 6.[7]

Be aware that `--expt-relaxed-constexpr` is an experimental feature and only works with *nvcc*. In fact, it is experimental since at least Cuda 8.0 from 2016, which is quite a long time for a feature to stay experimental. The behaviour of this option may change in future Cuda releases.

[7]The triple chevron syntax `<<< ... >>>` is used to call a `__global__` function from a `__host__` function, and thus to start a computation on the device. The numbers between the brackets determine how many threads are started, for our toy example we are content with one thread.

2.3.4 Pitfalls to avoid. In order that a function can be made `constexpr`, it must satisfy some requirements. [23] In general it is not allowed to call some intrinsics. Thus, a `constexpr` function may be slower at runtime, or need more ressources, than a non-`constexpr` function. In order to accomodate for this, a combination of this pattern with Pattern: Conditional function body may be advantageous.

2.3.5 Assessment.

++ Is also applicable to third party `constexpr` functions, 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 unclear, whether this feature is compatible with future C++ versions.
– The source code is not self contained any more, but needs to be compiled with certain compiler flags and with a certain compiler (*nvcc*)

2.3.6 Known Usages.

- LBANN [33] uses a defensive strategy: If the source is compiled with `--expt-relaxed-constexpr`, then functions are annotated with `constexpr`, otherwise with `__host__ __device__`.

2.4 Pattern: Disable the warnings

2.4.1 Context. You have some code, which is compiled for both host and device side. You know stray calls are not possible for some reasons. Yet, the compiler does not and thus spits out compilation warnings.

2.4.2 Problem. The compiler spits out compiler warnings which can be ignored and which mask other important warnings.

2.4.3 Solution. Disable the compiler warnings of the stray function calls. Since disabling warnings is usually a bad idea, it should happen in a the most fine grained fashion possible.

2.4.4 Example implementations. *nvcc* provides various ways to disable warnings regarding stray calls. Since *clang* emits much less warnings than *nvcc* in this regard, we do not discuss *clang* here, which has similar ways to disabled warnings.

The most fine grained control is acchieved by using the pragmas `nv_diagnostic push`, `nv_diag_suppress`, and `nv_diagnostic pop`. [11] They disable warnings for a specific part of the code.

The second set of pragmas consists of `#hd_warning_disable` and `#nv_exec_check_disable`. If one of the pragmas is placed in front of a function, then the compiler does not spill out warnings for bad function calls originating from that function The pragmas only affect the function they are placed in front of, in particular there is no need for a pragma which enables the warnings again. The pragmas also do not affect function calls in subfunctions.

If one wants to disable warnings regarding stray calls globally one can use the compiler options `-diag-suppress 20011,20014`.[8] If one wants to disable *all* warnings, then one can use `-disable-warnings`.[9]

[8]There must not be whitespace between the numbers.
[9]The numbers 20011 and 20014 in the listings and the compiler option examples, are the warning ids regarding stray function calls. They are not fixed, and may be different in a new *nvcc* version. To obtain the warnings ids, one can use the compiler option `-display-error-number`.

Listing 7: Disable the warnings

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

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

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

2.4.5 Pitfalls to avoid. Care should be taken with the pragmas `#hd_warning_disable` and `#nv_exec_check_disable`, which are undocumented; Yet Nvidia uses them itself in open sourced code. Their exact behaviour is not clear. In some cases one pragma works, in some cases the other, in some cases none. Even worse, it is reported that a wrong usage of the pragmas may lead to wrongly compiled code [12].

The biggest problem though is *maintanance*. Even if disabling the warning is correct currently, a code change in future may change this situation, and then the disabled warnings may hide severe programming errors.

2.4.6 Assessment.

+ Easy to use
∘ Each function has to be annotated manually.
– These pragmas are undocumented
– Wrong usage of these pragmas may lead to wrongly compiled code [12]
– May hide programming errors. A combination of using these pragmas with `release_assert` (in non-performance-critical code) should be considered
– Even when one "knows" that the pragmas can be used at the time when the code is written, things may change in the future.

2.4.7 Known Usages.

- Thrust [34]

```
#pragma nv_exec_check_disable
template< typename DerivedPolicy,
          typename ForwardIt,
          typename LessThanComp >
__host__ __device__
ForwardIt lower_bound(/* ... */);
```

- Eigen [32] XX **Code example would be nice**

2.4.8 Notes. The Pattern: Constexpr everything can be seen as an application of the Pattern: Disable the warnings, because by making everything `constexpr` we implicitely disable the

Listing 8: Solution: `#ifdef` block with `__CUDA_ARCH__`

```
#include "release_assert.h"
struct H {
  __host__ __device__
  static void func() {
    release_assert( !cuda_arch );
    /* body */
  }
};

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

int main() {
  kernel<<< 1, 1 >>>( H{} );
  return cudaDeviceSynchronize();
    // returns non-zero code
}
```

compiler warnings. A notable difference between these patterns is, that the functions in the Pattern: Constexpr everything should all qualify to be called on the host and the device side. Wherease, for Pattern: Disable the warnings we have some functions which must not be called from one of the sides. XX **This note concerns also other patterns. Check where it is best to put it to.**

2.5 Pattern: Defensive Programming

XX **Is this is an original pattern?**

2.5.1 Context. You have some "functionality" which cannot run on host or device side for certain inputs, because there exists no sensible implementation for one of the sides for certain inputs. At compile time you do not have enough knowledge to decide whether stray function calls are possible, or you cannot cast your knowledge into a shape which can be used at compile time.

2.5.2 Problem. You want to make sure at runtime that no stray function calls happen.

2.5.3 Solution. We use function dispatching at runtime and take measures if stray calls still happen, both by exploiting runtime information. To get rid of compiler warnings, we can use the patterns Pattern: Disable the warnings, Pattern: Constexpr everything, or Pattern: Host device everything.

The macro `__CUDA_ARCH__` can be used to decide whether one is in `__host__` or `__device__` code.

But, bugs always creep in, and thus it is vital to recognize such errors. How to handle such an error depends on the application, one can either try to continue - which is typically the need in availability, safety, or security critical applications, or to error out as fast as possible - which is (if possible) usually the better approach, and known as *offensive programming*. XX **citations?**

2.5.4 Implementation example for offensive programming. The simplest solution is to just abort the program whenever we end up in the wrong path, see Listing 8.[10] In that listing, `S::value()` is (wrongly) called in device code, and thus a runtime error on the GPU is triggered. The runtime error is not reported back to host side, until one checks the Cuda error code in host code,[11] here again using `cudaDeviceSynchronize`.

2.5.5 Assessment.

+ In most cases applicable
– Check of execution space is done at runtime.

2.6 Pattern: Conditional Host Device Template

XX Christoph: Das Pattern HOST DEVICE TEMPLATE finde ich sehr gut. Es beschreibt eine gut verständliche Lösung. Nach der Solution wird jedoch eine Variante beschrieben und es gibt auch eine eigene Assessment-Section für die Variante. Das macht das Pattern wieder komplizierter. Ist es wichtig, dass diese Variante so detailiert abgehandelt wird? Würde nicht einfach nur ein Satz in der Solution reichen, welcher erwähnt, dass man mit Macros der Code Duplication entgegenwirken könnte?

2.6.1 Context. You have some code, which is compiled for both host and device side, and the compiler complains about stray function calls. Yet, at compile time at the call side it is known whether a host and/or device function can be called, even when the caller does not know its own execution space at compile time. Furthermore, the code can be cast into a template function.

2.6.2 Problem. You want to make sure at compile time, that no stray calls can happen. **XX Symptom: Code compiles, although it should not**

2.6.3 Solution. We define three versions of the same function (in this section called *triplet*) – a `__host__`, a `__device__`, a `__host__ __device__` function – and make it such that always only one can be called, depending on the information passed by the caller (in general the passed types of the arguments). Subsequently, the compiler cannot instantiate a wrong function and we cannot get stray function calls. If we would try to do a stray function call, then we would get a compilation error.

More precisely, we define a compile time function `hdc<>`[12], which translates the passed types to an enum `HDC`, see Listing 9. `HDC` comprises three values: `HDC::Hst`, `HDC::Dev` or `HDC::HstDev`.[13] Our triplet of functions has an additional template parameter, which defaults to the output of `hdc<>`. Only if this output equals `HDC::Hst`, `HDC::Dev` or `HDC::HstDev`, the corresponding `__host__`, `__device__` or `__host__ __device__` function can be called.

To guide the compiler subsequently to the correct overload of our triplet, we add an additional template parameter `hst_` to all of them. The parameter `hst_` is in general not meant to be set by the user, but shall always take the return value from `hdc<>` (see [19] for a way how to achieve this). Finally we use a `requires` clause, which compares the value of `hst_` with the return value of `hdc<>`.[14] See Listing 10 for an example.

[10] The variable `cuda_arch` is usable in device code, although not being marked with `__constant__` or `__shared__`, due to [8].
[11] This is not entirely true: `__trap` raises an interrupt on the host side which in theory could be handled.
[12] Short for: *Host Device Compatibility*
[13] We note again that, only due to restricted space we use very short names here. In production code, some more descriptive names should be used.
[14] Up to C++17 we cannot use a `requires` clause, but have to resort to SFINAE, see Listing 102 for a not-totally-ugly approach.

Listing 9: Host Device Compatibility

```
enum class HDC { Hst, Dev, HstDev };
```

Listing 10: Conditional Template Instantiation

```
template< typename T, HDC x = hdc<T> >
  requires( x == HDC::Hst )
__host__ void func( T ) {/*body*/}
template< typename T, HDC x = hdc<T>  >
  requires( x == HDC::Dev )
__device__ void func( T ) {/*body*/}
template< typename T, HDC x = hdc<T>  >
  requires( x == HDC::HstDev )
__host__ __device__ void func( T ) {/*body*/}

// Usage:
template< typename T >
__host__ __device__
void wrapper( T t ) {
    func( t );
}
```

`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 101. XX **Merge texts**

2.6.4 Assessment.

- This solution only works for template functions or functions of template classes. This has some severe implications:
  - \+ More possibilities for compiler optimizations.
  - – Source code is physically coupled tighter, potentially leading to longer compilation times.
  - – The definition of the function must go into a header file.
  - – More code in header files and thus compilation times may increase.
- – Code bloat and thus increased compilation times.
- – Code duplication

2.6.5 Variant with macros. XX **Listing 11 als usecase bei dimetor**

The above solution, has the big drawback that one has to duplicate code manually three times. This can be automated using macros, see Listing **??**.[15]

But using macros does not solve all problems. Even worse it introduces a sever new one: Since the function body is in a macro now, debugging gets hard to impossible.[16]

XX **merge texts**

[15] A backslash as last character on a line is a line continuation. Since macros must be written in exactly one line, using '\' is used to split long macros into multiple lines.

[16] Actually, we do not know of any compiler which allows debugging a function defined in a macro.

2.6.6 Assessment of variant with macros. Apart from the points in the solution before, we have:

+ Duplicated code is automatically generated.
+ Very easy to use for the user
∘ Reasonably easy to use for the implementer.
– Since the body of the function is in a macro, debugging is hard.

2.6.7 Known usages.

- Dimetor (Closed source code) [35]

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

// Usage:
struct D {
  static constexpr HDC hdc = HDC::Dev;
  __device__ void func() {}
};

struct H {
  __host__ void func() {}
};

host_device_macro(
  (typename T),
  (hdc< T >),
  (void func( T t ) { t.func(); }) )

__device__ void d() {
  // func( H{} );  // does not compile
  func( D{} );
}

__host__ void h() {
  func( H{} );
  // func( D{} );  // does not compile
}
}
```

2.6.8 Pitfalls to avoid. XX **rephrase** The program is ill-formed, no diagnostic required, if: (6.1) no valid specialization can be generated for a template or a substatement of a constexpr if statement within a template and the template is not instantiated, or

A similar, also invalid, idea is to use a constant dependent on `__CUDA_ARCH__` as a non-type template parameter to some function. Whether using a `if constexpr` clause with a constant

dependent on `__CUDA_ARCH__` in the body is unclear. We have no authoritative answer from Nvidia yet.

2.7 Pattern: Function dispatching

2.7.1 Context. You applied Pattern: Conditional Host Device Template.

2.7.2 Problem. You want to get rid of code duplication. XX **This is not the problem, but the context** .

2.7.3 Solution. This can be acchieved using a combination of the patterns above. Firstly, we use the Pattern: Conditional Host Device Template to guide the compiler to the correct function of the dispatcher `__host__`, `__device__` and `__host__ __device__` triplet. This function than forwards its argument to *one* `__host__ __device__` function. The latter call is (for the compiler) a stray call, but due to our first indiraction, we know that the stray call cannot happen. Indeed, the dispatching function is never compiled for a case where a stray function call could happen.

Secondly, we can safely disable the compiler warnings emerging from the (unhappening) stray calls.

2.7.4 Implementation example. See Listing 11[17] for the solution. One can see, that this solution again needs boilerplate code – One has to write the code to forward the arguments to the actual function (here: `func_imp`) by hand. As for the **??**, one can use a macro to generate the boilerplate code. But in this situation, the body of the function is not part of the macro, and thus still debuggable.

2.7.5 Assessment.

+ No code duplication
+ Code is mostly debuggable.
+ Easy to use for the user.
∘ Library implementer again has to write some boilerplate code.
– Not straight forward to write and understand, and thus hard to maintain in the long term.
– One indirection for each function call in debug mode via `std::forward`. For implications of this, see [27, Item 30]. In Release mode this indirection most likely is optimized out.

2.7.6 Known usages. No known usages.

2.8 Anti Pattern: Link Pattern

XX **Other name: ch for cu Pattern, The Device Wrapper, Link Pattern** Do not know how to make sourcefiles for gcc and clang (.cu) withou code/file duplications.

2.8.1 Context. You have code which must be in header files. Yet you also want to call cuda or host intrinsics.

2.8.2 Problem. Directly calling those functions leads to compilation error, since the functions are not always defined during the compilation process.

2.8.3 Solution. You introduce an abstraction layer: You put the intrinsic into its own library, and link against this library during compilation. To be able to compile the library, you use a suitable pattern described above, most likely the Pattern: Conditional function body, Pattern: Disable the warnings, or Pattern: Function dispatching.

[17] nvcc implicitly considers `std::move` and `std::forward` to have `__host__ __device__` annotations. Therefore, the code in Listing 11 is valid. [7]

Listing 11: Function dispatching

```
struct H {
  void func() {}
};

struct D {
  static constexpr HDC hdc = HDC::Dev;
  __device__ void func() {}
};

#pragma nv_exec_check_disable
template< typename T >
__host__ __device__
void func_impl( T && t ) {
  t.func();
}

/* macro generated: start */
template< typename T, HDC hdc = hdc<T> >
requires( hdc == HDC::Hst )
__host__ void func( T && t ) {
  return func_impl( std::forward<T>(t) );
}
template< typename T, HDC hdc = hdc<T> >
requires( hdc == HDC::Dev )
__device__ void func( T && t ) {
  return func_impl( std::forward<T>(t) );
}
template< typename T, HDC hdc = hdc<T> >
requires( hdc == HDC::HstDev )
__host__ __device__ void func( T && t ) {
  return func_impl( std::forward<T>(t) );
}
/* macro generated: end */

// Usage:
__global__ void kernel() {
//func( H{} );  // error
  func( D{} );
}

int main() {
  func( H{} );
//func( D{} );  // error
}
```

Listing 101: Host Device compatibility: hdc

```
#include <type_traits>

HAS_MAKE( hdc )

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 );
```

APPENDIX

The appendix present the helper functions and macros used in the paper in Appendix A, elaborates on some C++/Cuda terms in Appendix B, and referes to a proposal to language changes to Cuda in Appendix C.

A APPENDIX: HELPER FUNCTIONS AND MACROS

This section collects the helper functions used in the listings above. They are by itself interesting, but none of them are new.

A.1 Host Device compatibility (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 101.

Depending on the use case, one may want to alter the code such that `hdc` returns `HDC::HstDev` for fundamental types like `int`,... and trivial C-style structs. To this end, the type traits `std::is_fundamental` and `std::is_trivial` can be used, `std::is_pod` should not be used, since it is deprecated since C++20.

Listing 102: REQUIRES (C++11)

```
#include <type_traits>

#define REQUIRES( ... )     \
  typename std::enable_if< \
    __VA_ARGS__, bool       \
  >::type = false

// usage example
template< typename T,
    REQUIRES( std::is_integral_v<T> ) >
T inc( T i ) { return i + 1; }
```

Listing 103: has_xxx (C++20)

```
#define HAS_MAKE( name )  \
  template< typename T >  \
  concept has_ ## name =  \
  requires( T t ) { &T::name; };

HAS_MAKE( foo );
```

A.2 REQUIRES

If one cannot use a `requires` clause (because C++20 is not available), one has to fall back to the `std::enable_if` pattern [22]. Unfortunately, this pattern is mostly incomprehensible for people who did not use it before. The macro `REQUIRES`, given in Listing 102[18] generates all the needed boilerplate code, and nearly resembles the syntax of the `requires` clause. XX **rephrase**

There is minor issure with the `REQUIRES` macro. It is only applicable to template functions. Or said in another way, it cannot be used when the function in question cannot be made into a template. In particular, it is not[19] applicable to some special member functions of template functions, e.g. the copy constructor.[20]

A.3 has_xxx type trait

The trait `has_xxx` returns whether a class has a member function or variable named xxx. We present a version for C++17, and one for C++20.

A C++20 version is given in Listing in 103. For each name one has to generate a trait using the macro `HAS_MAKE`.

[18]The macro uses `__VA_ARGS__` in order to be able to handle arguments which include commata.
[19]To overcome this problem, the standard approach up to C++17 is to inherit from a templated class which has non-templated copy constructors. This is so messy, that we refrain from giving a code example.
[20]Sadly, *nvcc* 12.2 seems to have bug and does not handle `requires` clauses on copy constructors correctly in all cases, see Bug No. 4212135. Thus, it may currently not be possible to apply the pattern Function dispatching to the copy constructor easily.

Listing 104: has (C++17)

```
#include <type_traits>

template< typename T, typename Lambda >
constexpr auto has_( Lambda && la )
 -> decltype( la(std::declval<T>()), true )
{ return true; }

template< typename >
constexpr auto has_( ... ) -> bool
{ return false; }

#define has( T, EXPR )       \
 has_< T >(                  \
 []( auto && obj )           \
 -> decltype( obj.EXPR){} )

// For compatibility to 20 version
#define HAS_MAKE( name )                \
 template<typename T> static constexpr \
 bool has_ ## name = has( T, name );

// usage example
struct X {
  int foo;
};
static_assert( has(X, foo) );

HAS_MAKE( foo )
static_assert(  has_foo<X> );
static_assert( !has_foo<int> );
```

The C++17 version in Listing 104 [21] has the advantage, that one does not need to generate a type trait for each name. For compatibility reasons to our code listing, we nevertheless define such a helper trait.

A C++14 version (with slightly different properties) can be found at [20].

B APPENDIX: CUDA/C++ BACKGROUND

XX Introductory text missing

nvcc. The nvidia Cuda compiler *nvcc* processes Cuda code in two steps. First, it processes all device (GPU) specific code sections by itself, and then passes a processed version of the code to the host compiler, usually *gcc* or *clang*.

Function execution space specifiers. Cuda distinguishes between `__host__`, `__device__`, and `__global__` functions [2]: `__host__` functions run on the CPU, `__device__` and `__global__` functions run on the GPU. `__global__` function are the entry point for `__host__` functions to start something on

the GPU. Unsurprisingly, `__host__` functions are allowed to call `__host__` and `__global__` functions, `__global__` and `__device__` functions are allowed to call `__global__` and `__device__` functions.

Functions without annotations are implicitly considered as `__host__` functions by the compiler.

Cuda error checking. Since Cuda has no support for exceptions in device code, error handling is done by checking return error codes. Furthermore, an error on the GPU usually never leads to a termination of the program, just to undefined behaviour if the GPU is used subsequently. Thus, after each (potentially failing) Cuda call, one should check the error code. We will (ab)use `cudaDeviceSynchronize` for this task.[21] A return code of `0` means success.

*Undefined Behaviour (*UB*).* C++ has a feature called *undefined behaviour* (*UB*). Whenever an operation results in *UB*, the behaviour of the program (starting from that particular operation) is not defined any more. In other words: Anything is allowed to happen. For an example of UB see Listing 1.

The compiler is not mandated to, nor is it able to, diagnose all potential appearances of *UB*. On the other hand, just because something is *UB*, does not mean that the compiler is mandated to generate broken code.

Novices in C++ usually are usually not aware of the dangers of UB, until they get more proficient with the language, up to some point where they greatly overestimate the danger of it.

Further information. Information about restrictions in device code can be found in the *Cuda Documentation* [9, Section 14]. Compilers, and different versions of it, can be tested using *godbolt* [28].

C APPENDIX: PATTERNS WHICH NEED LANGUAGE CHANGES

A proposal for solving the described issue elegantly is submitted to Nvidia [18]. XX **Remove this?**

REFERENCES

[1] LLVM, *Compiling CUDA with clang* llvm.org/docs/CompileCudaWithLLVM.html

[2] Nvidia, *CUDA C++ Programming Guide - Function Execution Space Specifiers*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#function-execution-space-specifiers.

[3] Nvidia, *CUDA C++ Programming Guide - Kernels*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#-kernels.

[4] Nvidia, *CUDA C++ Programming Guide - __global__ Function Argument Processing*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-function-argument-processing.

[5] Nvidia, *CUDA C++ Programming Guide - Preprocessor Symbols*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#preprocessor-symbols.

[6] Nvidia, *CUDA C++ Programming Guide - Constexpr functions and function templates*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#constexpr-functions-and-function-templates.

[7] Nvidia, *CUDA C++ Programming Guide - Constexpr functions and function templates*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#rvalue-references.

[8] Nvidia, *CUDA C++ Programming Guide - Const-qualified variables*, docs.nvidia.com/cuda/cuda-c-programming-guide/-index.html#const-qualified-variables.

[9] Nvidia, *CUDA C++ Programming Guide - C++ Language Support*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-support.

[10] Nvidia, *CUDA C++ Programming Guide - Notes on __host__ __device__ lambdas*, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#notes-on-host-device-lambdas.

[11] Nvidia, *Reducing Application Build Times Using CUDA C++ Compilation Aids*, developer.nvidia.com/blog/reducing-application-build-times-using-cuda-c-compilation-aids/

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

[21] In production code error checking should be done differently, especially since `cudaDeviceSynchronize` is not a reliable way to check for all possible Cuda errors. [14, 15]

[13] Richard, *Is it possible to get assertion info from within a CUDA kernel?*, stackoverflow.com/q/72565993/8052809.
[14] talonmies, *What is the canonical way to check for errors using the CUDA runtime API?*, stackoverflow.com/q/14038589/-8052809.
[15] Robert Crovella, *States of memory data after cuda exceptions* stackoverflow.com/a/31642573/8052809.
[16] Jan Zapletal, *calling a __host__ function from a __host__ __device__ function*, stackoverflow.com/q/62810274/8052809.
[17] WilliamKF, Jakub Klinkovský, *How to disable Cuda host device warning for just one function?*, stackoverflow.com/a/-57641289/8052809.
[18] Thomas Mejstrik, *__host__ __device__ - Generic programming in Cuda*, developer.nvidia.com/bugs/4258859.
[19] HolyBlackCat, *How can I prevent user from specifying a function template parameter, forcing it to be deduced?*, stackoverflow.com//a/58771661/8052809.
[20] Kirill V. Lyadvinsky, Johannes Schaub - litb, *How to detect whether there is a specific member variable in class?*, stackoverflow.com/a/1007175/8052809.
[21] andy, Dmytro Ovdiienko, *Templated check for the existence of a class member function?*, stackoverflow.com/a/-62292282/8052809.
[22] cppreference.com, *std::enable_if*, cppreference.com/w/cpp/lan/guage/constexpr.
[23] cppreference.com, *constexpr*, cppreference.com/w/cpp/types/enable_if.
[24] cppreference.com, *SFINAE - Substitution Failure Is Not An Error*, cppreference.com/w/cpp/language/sfinae.
[25] Wikipedia, *Substitution failure is not an error*, en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.
[26] Christopher Preschern. 2019. *Patterns to escape the #ifdef hell*, Proc. of the 24th European Conference on Pattern Languages of Programs (EuroPLop '19) 2, 1–12, ACM, doi: 10.1145/3361149.3361151.
[27] Scott Meyers, *Effective Modern C++*, O'Reilly Media, 2014,
[28] Matt Godbolt, *Compiler Explorer*, godbolt.org.
[29] *cppreference.com*, cppreference.com/.
[30] *Working Draft, Programming Languages – C++*, eel.is/c++draft/.
[31] Beman Dawes, *Boost.Utility* [Computer Software], boost.org/doc/libs/1_82_0/libs/utility/doc/html/index.html
[32] Benoît Jacob, Gaël Guennebaud, et al., *Eigen* [Computer Software], gitlab.com/libeigen/eigen.
[33] LLNL, *LBANN: Livermore Big Artificial Neural Network Toolkit* [Computer Software], github.com/LLNL/lbann.
[34] Nvidia, *Thrust* [Computer Software] docs.nvidia.com/cuda/thrust/index.html.
[35] Dimetor, *AirborneRF* [Computer Software], (2020), www.dimetor.com.