Bridging the Gap:
Writing Portable Programs for CPU and GPU

THOMAS MEJSTRIK

universität wien

FWF

Bridging the Gap: Writing Portable Programs for CPU and GPU using CUDA

Thomas Mejstrik
Sebastian Woblistin

Content

Motivation

Motivation:

Audience etc..

Audience etc..

- Ask questions during the talk
  - Or after the talk
- Audience?
- Only Cuda

Audience etc..

- Ask questions during the talk
- Audience?
  - Target audience: Cuda/Nvidia
  - Other heterogeneous systems
  - Liked the title / Disliked other titles
  - Do not know where they are
- Only Cuda

Audience etc..

- Ask questions during the talk
- Audience?
- Only Cuda
  - What is Cuda
  - Do not ask me about SYCL, ROCm, Vulkan, ...
  - You can tell me about afterwards

Why write programs for CPU and GPU

- Difference CPU/GPU
  Algorithms are designed differently
  - Latency/Throughput
  - Memory bandwidth
  - Number of cores
  - Handling of branches
  - Cache sizes
  - number formats

  "Under the radar"-Problem
- Why it makes sense?
- Scope of the talk

Why write programs for CPU and GPU

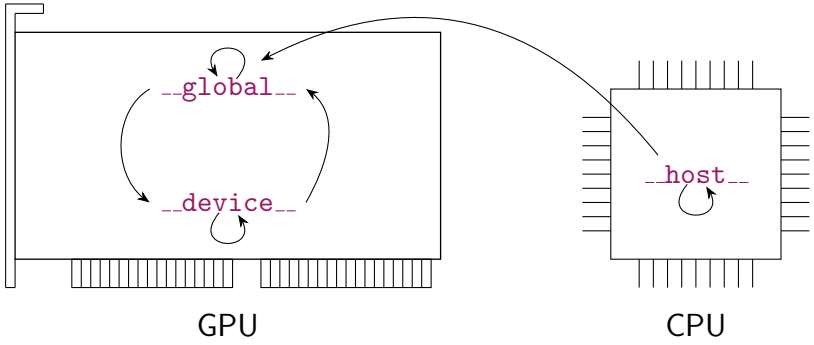
- Difference CPU/GPU
- Why it makes sense?
  - Library/Framework developers
  - Embarrassingly parallel algorithms
  - User experience
  - Developer experience
  - Debugging / Testing - No performance issues
- Scope of the talk

Why write programs for CPU and GPU

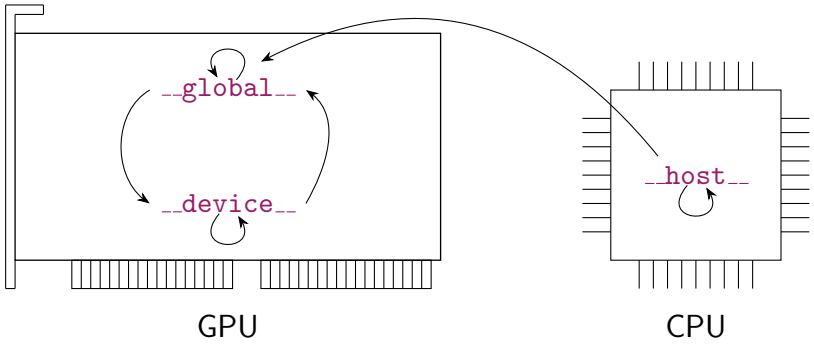
- Difference CPU/GPU
- Why it makes sense?
- Scope of the talk
  - Functions - Helper functions, containers,
  - ~~Variables~~, ~~Data handling~~
  - No performance considerations
  - No silver bullet©
  - Navigate around Cuda problems

Motivation:
Cuda crash course

Allowed function calls in Cuda



Allowed function calls in Cuda



- `__global__` $\simeq$ `__device__`

Cuda "Hello World" without world

```
1  #include <cstdio>
2
3  __device__ int  print()  { return 0; }
4  __global__ void kernel() { printf( "%i", print() ); }
5  __host__   void start()  { kernel<<< 2, 3 >>>(); }
6
7  int main() {  // implicitly __host__
8    start();
9    return cudaDeviceSynchronize();
10 }
```

Cuda "Hello World" without world

```
1  #include <cstdio>
2
3  __device__ int  print()  { return 0; }
4  __global__ void kernel() { printf( "%i", print() ); }
5  __host__   void start()  { kernel<<< 2, 3 >>>(); }
6
7  int main() {  // implicitly __host__
8    start();
9    return cudaDeviceSynchronize();
10 }
```

- nvcc + host compiler
- clang
- HIP / nvc / gpucc / ???

Cuda "Hello World" without world

```cpp
#include <cstdio>

__device__ int  print()  { return 0; }
__global__ void kernel() { printf( "%i", print() ); }
__host__   void start()  { kernel<<< 2, 3 >>>(); }

int main() {  // implicitly __host__
  start();
  return cudaDeviceSynchronize();
}
```

- nvcc + host compiler / clang

```
stdout: 000000
return code: 0
```

Motivation:
Quiz time

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    // return H.func();
10   // return D.func();
11   // return wrap< H >();
12   // return wrap< D >();
13 }
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    return H.func();
10   // return D.func();
11   // return wrap< H >();
12   // return wrap< D >();
13 }
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    return H.func();      // OK
10   // return D.func();
11   // return wrap< H >();
12   // return wrap< D >();
13 }
```

[[No compiler messages]]

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   return D.func();
11   // return wrap< H >();
12   // return wrap< D >();
13 }
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   return D.func();     // compilation error
11   // return wrap< H >();
12   // return wrap< D >();
13 }
```

```
<source>(12): error: calling a __device__ function("D::func()") from a __host__
function("main") is not allowed return D.func();
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }  // __host__ int wrap(){ H.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   // return D.func();      // compilation error
11   return wrap< H >();
12   // return wrap< D >();
13 }
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }  // __host__ int wrap(){ H.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   // return D.func();      // compilation error
11   return wrap< H >();  // compilation warning
12   // return wrap< D >();
13 }
```

```
<source>(8): warning #20014-D: calling a __host__ function from a __host__ __device__
function is not allowed. detected during instantiation of "int wrap<T>() [with T=H]"
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }   // __host__ int wrap(){ D.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   // return D.func();      // compilation error
11   // return wrap< H >();   // compilation warning
12   return wrap< D >();
13 }
```

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }   // __host__ int wrap(){ D.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   // return D.func();      // compilation error
11   // return wrap< H >();   // compilation warning
12   return wrap< D >();   // no warning, UB at runtime
13 }
```

[[No compiler messages]]

nvcc 12.6

Quiz time: Bad cross function calls

```
1  struct H { __host__   int func() { return 42; } };
2
3  struct D { __device__ int func() { return 666; } };
4
5  template< typename T > __host__ __device__
6  int wrap() { return T{}.func(); }
7
8  int main() {
9    // return H.func();      // OK
10   // return D.func();      // compilation error
11   // return wrap< H >();   // compilation warning
12   // return wrap< D >();   // no warning, UB at runtime
13 }
```

nvcc 12.6
clang: *it depends*

Patterns

Patterns:
Oldschool

Oldschool - Solution

- Use very old Cuda version

Patterns:

__host__ __device__ all the things©

__host__ __device__ all the things - Problem

```
1 inline constexpr
2 float MHzToWavelength( int frequency_in_MHz )
3 {
4     return 299792458.f / (frequency_in_MHz * 1000.f * 1000.f);
5 }
```

__host__ __device__ all the things - Solution

```
1 __host__ __device__ inline constexpr
2 float MHzToWavelength( int frequency_in_MHz )
3 {
4     return 299792458.f / (frequency_in_MHz * 1000.f * 1000.f);
5 }
```

__host__ __device__ all the things - Solution

```
1  struct H {
2    __host__ __device__ int func() { return 42; }
3  };
4  struct D {
5    __host__ __device__ int func() { return 666; }
6  };
7  template< typename T > __host__ __device__
8  int wrap() { return T{}.func(); }
9
10 int main() {
11   H{}.func();    // OK
12   D{}.func();    // OK
13   wrap< H >();   // OK
14   wrap< D >();   // OK
15 }
```

`__host__ __device__` *all the things* - Consequences

+ Easy to use
◦ Code bloat
◦ Not always possible
– May hide logical errors

Patterns:

`__host__ __device__` *everywhere*

__host__ __device__ everywhere - Problem

```
1  // mhz.h
2  __host__ __device__ inline constexpr
3  float MHzToWavelength( int frequency_in_MHz ) {
4      return 299792458.f / (frequency_in_MHz * 1000.f * 1000.f);
5  }
6
7  // mhz.cpp
8  #include "mhz.h"
9  #include <cstdio>
10
11 int main( int argc, char ** ) {
12     printf( "%f", MHzToWavelength( argc ) );
13 }
```

Compile with: `gcc mhz.cpp`

__host__ __device__ everywhere - Problem

```
1  // mhz.h
2  __host__ __device__ inline constexpr
3  float MHzToWavelength( int frequency_in_MHz ) {
4      return 299792458.f / (frequency_in_MHz * 1000.f * 1000.f);
5  }
6
7  // mhz.cpp
8  #include "mhz.h"
9  #include <cstdio>
10
11 int main( int argc, char ** ) {
12     printf( "%f", MHzToWavelength( argc ) );
13 }
```

Compile with: `gcc mhz.cpp`

`error: '__host__' does not name a type` gcc 14.2

__host__ __device__ everywhere - Solution

```
1  #ifndef CUDATAGS
2    #define CUDATAGS
3    #ifndef __CUDACC__
4      #define __host__
5      #define __device__
6    #endif
7  #endif
8
9  __host__ __device__
10 void func() {}
```

__host__ __device__ everywhere - Solution

```
#ifndef CUDATAGS
  #define CUDATAGS
  #ifndef __CUDACC__
    #define __host__
    #define __device__
  #endif
#endif

__host__ __device__
void func() {}
```

```
#ifndef CUDATAGS
  #define CUDATAGS
  #ifndef
    #define HST
    #define DEV
  #else
    #define HST __host__
    #define DEV __device__
  #endif
#endif

HST DEV
void func() {}
```

Patterns:
Conditional function body

Conditional function body - Problem

2-Norm of vector in $\mathbb{R}^3$:

- Manually
- Cuda: `norm3df`

Conditional function body

- Compilation of Cuda
  - nvcc + host compiler
  - clang
- Language differences:
- `__CUDA_ARCH__`
- `__NVCC__`

Conditional function body

- Compilation of Cuda
- Language differences:
  - Function signatures
  - [Some others]
- `__CUDA_ARCH__`
- `__NVCC__`

Conditional function body

- Compilation of Cuda
- Language differences:
- `__CUDA_ARCH__`
  - Defined when device code is compiled
  - Restrictions (later)
- `__NVCC__`

Conditional function body

- Compilation of Cuda
- Language differences:
- `__CUDA_ARCH__`
- `__NVCC__`

```
1 #if defined(__clang__) && defined(__CUDA__) && !defined(__CUDA_ARCH__)
2   // clang compiling CUDA code, host mode.
3 #endif
4
5 #if defined(__clang__) && defined(__CUDA__) && defined(__CUDA_ARCH__)
6   // clang compiling CUDA code, device mode.
7 #endif
```

Conditional function body - Solution

clang

```
1  __host__
2  float norm( const Vec3f & v ) {
3    double sum = 0;
4    for ( int i = 0; i < 3; ++i )
5      sum += v[i]*v[i];
6    return sqrtf( (float)sum );
7  }
8
9  __device__
10 float norm( const Vec3f & v ) {
11   return norm3df(v[0],v[1],v[2]);
12 }
```

Conditional function body - Solution

clang

```
1  __host__
2  float norm( const Vec3f & v ) {
3    double sum = 0;
4    for ( int i = 0; i < 3; ++i )
5      sum += v[i]*v[i];
6    return sqrtf( (float)sum );
7  }
8
9  __device__
10 float norm( const Vec3f & v ) {
11   return norm3df(v[0],v[1],v[2]);
12 }
```

nvcc + host compiler / *clang*

```
1  __host__ __device__
2  float norm( const Vec3f & v ) {
3    #ifndef __CUDA_ARCH__
4      double sum = 0;
5      for ( int i = 0; i < 3; ++i )
6        sum += v[i]*v[i];
7      return sqrtf( (float)sum );
8    #else
9      return norm3df(v[0],v[1],v[2]);
10   #endif
11 }
```

Preferred

Conditional function body - __CUDA_ARCH__

- *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.*
- `if constexpr` ??

Conditional function body - __CUDA_ARCH__

Not a solution!
Warning before the spell©

Conditional function body - __CUDA_ARCH__

```
1  struct H {
2    __host__ int func() { return 42}
3  };
4
5  template< typename T >
6  __host__ __device__
7  void wrap() { T{}.func(); }
8
9  int main() {
10   #ifndef __CUDA_ARCH__      //
11     wrap< H >();             // UB
12   #endif                     //
13 }
```

Not a solution!
Warning before the spell©

Conditional return type - Solution

clang

```
1  struct H {};
2  struct D {};
3
4  __host__   H func() {
5    return H{};
6  }
7  __device__ D func() {
8    return D{};
9  }
```

Conditional return type - Solution

clang

```
1  struct H {};
2  struct D {};
3
4  __host__   H func() {
5    return H{};
6  }
7  __device__ D func() {
8    return D{};
9  }
```

nvcc: UB

```
1  struct H {};
2  struct D {};
3
4  #ifndef __CUDA_ARCH__
5    __host__   H func() {
6      return H{};
7    }
8  #else
9    __device__ D func() {
10     return D{};
11   }
12 #endif
```

Patterns:

constexpr everything

constexpr everything - Problem

```
1 constexpr int func() {
2     std::array< int, 5 > a;
3     /* ... */
4     return std::accumulate( a.begin(), a.end(), 0 );
5 }
```

constexpr everything - Problem

```
1 constexpr int func() {
2     std::array< int, 5 > a;
3     /* ... */
4     return std::accumulate( a.begin(), a.end(), 0 );
5 }
```

Compile with nvcc 12.5

```
error: calling a __host__ function("func") from a __device__ function("???") is not allowed

error: calling a __host__ function("T2 ::std::accumulate<int *, int> (T1, T1, T2)") from a __device__ function("func") is not allowed

...

The experimental flag '--expt-relaxed-constexpr' can be used to allow this.
```

`constexpr` everything

- Context: Function ··· Cuda and non-Cuda compilers ··· host and device side ··· implementation ok ··· `constexpr` ···

`constexpr` everything

- Context: Function $\cdots$ Cuda and non-Cuda compilers $\cdots$ host and device side $\cdots$ implementation ok $\cdots$ `constexpr` $\cdots$
- Problem: Cannot make changes to code

constexpr everything

- Context: Function ··· Cuda and non-Cuda compilers ··· host and device side ··· implementation ok ··· `constexpr` ···
- Problem: Cannot make changes to code
- Solution: Compile with *nvcc* and `--expt-relaxed-constexpr`

`constexpr` *everything* - 3 known uses

- RAPIDS[1] by Nvidia: Discussed whether to use `--expt-relaxed-constexpr`, but eventually decided against it

[1] Data science and AI library
[2]

`constexpr` *everything* - 3 known uses

- RAPIDS[1] by Nvidia: Discussed whether to use `--expt-relaxed-constexpr`, but eventually decided against it
- MatX[2] by Nvidia: Uses `--expt-relaxed-constexpr`

[1] Data science and AI library
[2] Matrix functions

`constexpr` *everything* - 3 known uses

- RAPIDS[1] by Nvidia: Discussed whether to use `--expt-relaxed-constexpr`, but eventually decided against it
- MatX[2] by Nvidia: Uses `--expt-relaxed-constexpr`
- Dimetor: Under consideration

[1] Data science and AI library
[2] Matrix functions

constexpr everything - Failing examples

```
1 constexpr int foo( int j ) {
2     if( j < 0 ) throw;
3     return 42;
4 }
```

becomes on nvcc 12.2 (without compiler warnings)

```
1 __device__ constexpr int foo( int j ) {
2     return 42;
3 }
```

Jake Hemstad

constexpr everything - Failing examples

```
1 constexpr int foo( int j ) {
2     if( j < 0 ) throw;
3     return 42;
4 }
```

becomes on nvcc 12.2 (without compiler warnings)

```
1 __device__ constexpr int foo( int j ) {
2     return 42;
3 }
```

Jake Hemstad

constexpr everything - Failing

```
1  int bar( int i ) {
2      return i * 2;
3  }
4
5  constexpr int foo( int j ) {
6      if( j < 0 ) return bar( j );
7      return 42;
8  }
```

becomes on nvcc 12.2 (without compiler warnings)

```
1  __device__ constexpr int foo( int j ) {
2      return 42;
3  }
```

Jake Hemstad

constexpr everything - Failing

```
1  int bar( int i ) {
2      return i * 2;
3  }
4
5  constexpr int foo( int j ) {
6      if( j < 0 ) return bar( j );
7      return 42;
8  }
```

becomes on nvcc 12.2 (without compiler warnings)

```
1  __device__ constexpr int foo( int j ) {
2      return 42;
3  }
```

Jake Hemstad

constexpr everything - Failing

```
1 constexpr int set() {
2     auto i = (int*) malloc( sizeof(int) );
3     *i = 42;
4     int y = *i;
5     free( i );
6     return y;
7 }
```

constexpr everything - Failing

```
1 constexpr int set() {
2     auto i = (int*) malloc( sizeof(int) );
3     *i = 42;
4     int y = *i;
5     free( i );
6     return y;
7 }
```

- May work – or not
- Depends on Compiler, OS, Driver, GPU, ...

constexpr everything - Failing

```
1 constexpr int set() {
2     auto i = (int*) malloc( sizeof(int) );
3     *i = 42;
4     int y = *i;
5     free( i );
6     return y;
7 }
```

- May work – or not
- Depends on Compiler, OS, Driver, GPU, ...
- Not what you want

`constexpr` *everything* - Consequences

++ Is also applicable to third party `constexpr` functions
- `std::array`
- `std::optional`
- `std::upper_bound`
- ...

\+ Easy to use

\+ Needs minimal changes to the source code

– Only applicable to `constexpr` functions

– Bad if used in a library

– Is an experimental feature ($\leq$ 2016)

– Future C++ versions?

–– May lead to subtle bugs[3]

[3] *github.com/rapidsai/cudf/issues/7795*

`constexpr` *everything* - Consequences

++ Is also applicable to third party `constexpr` functions
- `std::array`
- `std::optional`
- `std::upper_bound`
- ...

\+ Easy to use

\+ Needs minimal c

– Only applicable

– Bad if used in a library

– Is an experimental feature ($\leq$ 2016)

– Future C++ versions?

–– May lead to subtle bugs[3]

Takeaway:
Consider© using
`--expt-relaxed-constexpr`

[3] *github.com/rapidsai/cudf/issues/7795*

Patterns:

Disable Cuda warnings

Disable Cuda warnings - Problem

```
1
2  template< typename Container >
3  __host__ __device__ constexpr
4  void fill_ones( Container & ct ) {
5      for ( auto & x : ct ) x = 1;
6  }
7
8  #include <vector>
9  int main() {
10     std::vector< int > v{1,2,3,4,5};
11     fill_ones( v );
12 }
```

- Code equivalent to Quiz example: `return wrap< H >();`

nvcc 12.5

Disable Cuda warnings - Solution

```
1
2  template< typename Container >
3  __host__ __device__ constexpr
4  void fill_ones( Container & ct ) {
5      for ( auto & x : ct ) x = 1;
6  }
7
8  #include <vector>
9  int main() {
10     std::vector< int > v{1,2,3,4,5};
11     fill_ones( v );
12 }
```

- Code equivalent to Quiz example: `return wrap< H >();`

```
<source>(5): warning #20014-D: calling a __host__ function
from a __host__ __device__ function is not allowed
```

nvcc 12.5

Disable Cuda warnings - Solution

```
1  #pragma nv_exec_check_disable
2  template< typename Container >
3  __host__ __device__ constexpr
4  void fill_ones( Container & ct ) {
5      for ( auto & x : ct ) x = 1;
6  }
7
8  #include <vector>
9  int main() {
10     std::vector< int > v{1,2,3,4,5};
11     fill_ones( v );
12 }
```

- Code equivalent to Quiz example: `return wrap< H >();`

`[[No compiler messages]]`

nvcc 12.5

Disable Cuda warnings - How?

- function scope pragmas
  - `#pragma hd_warning_disable` and
  - `#pragma nv_exec_check_disable`
- line scope pragmas
- global scope: compiler flags
- Patterns from before

Disable Cuda warnings - How?

- function scope pragmas
- line scope pragmas
  - `#pragma nv_diagnostic push`,
    `#pragma nv_diag_suppress`,
    `#pragma nv_diagnostic pop`, ...
  - Similar to `#pragma GCC diagnostic pop`, ...
- global scope: compiler flags
- Patterns from before

Disable Cuda warnings - How?

- function scope pragmas
- line scope pragmas
- global scope: compiler flags
  - `--diag-suppress 20011,20014`
- Patterns from before

Disable Cuda warnings - How?

- function scope pragmas
- line scope pragmas
- global scope: compiler flags
- Patterns from before
  - `constexpr` *everything*
  - `__host__ __device__` *everything*

Disable Cuda warnings - 3 known uses

- Thrust (Nvidia): `nv_exec_check_disable`

```
1 #pragma nv_exec_check_disable
2 template< typename Policy, typename Iter, typename Comp >
3 __host__ __device__ Iter lower_bound( /* ... */ );
```

Not recommended

Disable Cuda warnings - 3 known uses

- Thrust (Nvidia): `nv_exec_check_disable`
- Eigen: `nv_exec_check_disable` and `--expt-relaxed-constexpr`

Disable Cuda warnings - 3 known uses

- Thrust (Nvidia): `nv_exec_check_disable`
- Eigen: `nv_exec_check_disable` and `--expt-relaxed-constexpr`
- Dimetor

```
1 #pragma nv_diag_suppress 20011,20014
2 #include <Eigen/Core>
3 #pragma nv_diag_warning 20011,20014
```

Disable Cuda warnings - Consequences

\+ Easy to use

◦ Each function has to be annotated manually

– Pragmas `hd_warning_disable` and `nv_exec_check_disable` are undocumented, Wrong usage may lead to wrongly compiled code[4]

−− Future?

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

Disable Cuda warnings - Consequences

\+ Easy to use

◦ Each function has to be annotated manually

– Pragmas `hd_warning_disable` and `nv_exec_check_disable` are undocumented, Wrong usage may lead to wrongly compiled code[4]

−− Future?

Disable Cuda warnings:
Not recommended
but sometimes needed

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

Patterns:

`__host__ __device__` *template*

__host__ __device__ template - Problem

```
1  template< typename Container >
2  __host__ __device__ constexpr
3  void fill_ones( Container & ct ) {
4      for ( auto & e : ct ) {  // std::vector::begin:  __host__
5          e = 1;
6      }
7  }
8
9  #include <vector>
10 int main() {
11     std::vector< int > v{1,2,3,4,5};
12     fill_ones( v );
13 }
```

```
warning #20014-D: calling a __host__ function
from a __host__ __device__ function is not allowed
```

nvcc 12.5

__host__ __device__ template - Solution

```
1  enum class HDC { Hst, Dev, HstDev };
2
3  #define MACRO( targ_, hdc_, func_ )  \
4      template< targ_, HDC x = hdc_ >  requires( x == HDC::Hst )    \
5      __host__              func_                                    \
6      template< targ_, HDC x = hdc_ > requires( x == HDC::Dev )     \
7      __device__ func_                                               \
8      template< targ_, HDC x = hdc_ > requires( x == HDC::HstDev ) \
9      __host__ __device__ func_
10
11 MACRO( typename Container, hdc< Container >,
12    void fill_ones( Container & ct ) { for( auto & e : ct ){ e = 1; } } )
13
14 #include <vector>
15 int main() {
16     std::vector< int > v{1,2,3,4,5};
17     fill_ones( v );
18 }
```

`__host__ __device__` template

- Context: Template functions $\cdots$ can encode target in types
- Problem: Compiler instantiates functions for wrong sides
- Solution: Three versions of the same function: `__host__`, `__device__`, `__host__ __device__` $\cdots$ only one can be called

__host__ __device__ template

```
1  enum class HDC { Hst, Dev, HstDev };  // Host-Device-Compability
2
3  template< typename T, HDC x = hdc<T> >  // helper trait: hdc<> -> HDC value
4    requires( x == HDC::Hst )
5    __host__              void func( T ) { /*body*/ }
6  template< typename T, HDC x = hdc<T> >
7    requires( x == HDC::Dev )
8              __device__ void func(  T ) { /*body*/ }
9  template< typename T, HDC x = hdc<T> >
10   requires( x == HDC::HstDev )
11   __host__ __device__ void func( T ) { /*body*/ }
12
13 // Usage example
14 template< typename T > __host__ __device__
15 void wrap( T t ) {
16     func( t );
17 }
```

__host__ __device__ template

```
1  enum class HDC { Hst, Dev, HstDev };  // Host-Device-Compability
2
3  template< typename T, HDC x = hdc<T> >  // helper trait: hdc<> -> HDC value
4    requires( x == HDC::Hst )
5    __host__              void func( T ) { /*body*/ }
6  template< typename T, HDC x = hdc<T> >
7    requires( x == HDC::Dev )
8              __device__ void func(  T ) { /*body*/ }
9  template< typename T, HDC x = hdc<T> >
10   requires( x == HDC::HstDev )
11   __host__ __device__ void func( T ) { /*body*/ }
12
13 // Usage example
14 template< typename T > __host__ __device__
15 void wrap( T t ) {
16     func( t );
17 }
```

__host__ __device__ template

```
1  enum class HDC { Hst, Dev, HstDev };  // Host-Device-Compability
2
3  template< typename T, HDC x = hdc<T> >  // helper trait: hdc<> -> HDC value
4    requires( x == HDC::Hst )
5    __host__              void func( T ) { /*body*/ }
6  template< typename T, HDC x = hdc<T> >
7    requires( x == HDC::Dev )
8              __device__ void func(  T ) { /*body*/ }
9  template< typename T, HDC x = hdc<T> >
10   requires( x == HDC::HstDev )
11   __host__ __device__ void func( T ) { /*body*/ }
12
13 // Usage example
14 template< typename T > __host__ __device__
15 void wrap( T t ) {
16     func( t );
17 }
```

__host__ __device__ template

```
1  enum class HDC { Hst, Dev, HstDev };  // Host-Device-Compability
2
3  template< typename T, HDC x = hdc<T> >  // helper trait: hdc<> -> HDC value
4    requires( x == HDC::Hst )
5    __host__              void func( T ) { /*body*/ }
6  template< typename T, HDC x = hdc<T> >
7    requires( x == HDC::Dev )
8              __device__ void func(  T ) { /*body*/ }
9  template< typename T, HDC x = hdc<T> >
10   requires( x == HDC::HstDev )
11   __host__ __device__ void func( T ) { /*body*/ }
12
13 // Usage example
14 template< typename T > __host__ __device__
15 void wrap( T t ) {
16     func( t );
17 }
```

__host__ __device__ template

```
1  enum class HDC { Hst, Dev, HstDev };  // Host-Device-Compability
2
3  template< typename T, HDC x = hdc<T> >  // helper trait: hdc<> -> HDC value
4    requires( x == HDC::Hst )
5    __host__              void func( T ) { /*body*/ }
6  template< typename T, HDC x = hdc<T> >
7    requires( x == HDC::Dev )
8              __device__ void func(  T ) { /*body*/ }
9  template< typename T, HDC x = hdc<T> >
10   requires( x == HDC::HstDev )
11   __host__ __device__ void func( T ) { /*body*/ }
12
13 // Usage example
14 template< typename T > __host__ __device__
15 void wrap( T t ) {
16     func( t );
17 }
```

__host__ __device__ template macro - Extra points

```
1  __global__ void kernel() {
2      //fill_ones( H{} );  // compilation error
3      fill_ones( D{} );
4  }
5
6  int main() {
7      fill_ones( H{} );
8      //fill_ones( D{} );  // compilation error
9  }
```

`__host__ __device__` *template* - Consequences

+ No bad cross function calls possible
+ No wrong compiler warnings
+ Easy for the user
◦ Only for template functions
– Difficult for the maintainer
– Code duplication

__host__ __device__ template macro - Solution

```
1  #define MACRO( targ_, hdc_, func_ )   \
2      template< targ_, HDC x = hdc_ >  \
3        requires( x == HDC::Hst )      \
4        __host__              func_    \
5      template< targ_, HDC x = hdc_ >  \
6        requires( x == HDC::Dev )      \
7                 __device__ func_      \
8      template< targ_, HDC x = hdc_ >  \
9        requires( x == HDC::HstDev )   \
10       __host__ __device__ func_
11
12 // Usage example
13 MACRO( typename Container,
14        hdc< Container >,
15        void fill_ones( Container & ct ) { for( auto & x : ct ){ x = 1; } }
16      )
```

__host__ __device__ template macro - Example

```
1  template< typename T, int N >
2  struct hstdev_array {
3    static constexpr HDC hdc = HDC::HstDev;
4    __host__ __device__ T& operator[]( int idx ){ return data_[ idx + 1 ]; }
5     /* ... */
6  };
7
8  __global__ void kernel() {
9    hstdev_array< int, 5 > hd_a{1,2,3,4,5}
10   fill_ones( hd_a );  // hdc< hstdev_array > == HDC::HstDev
11 }
12
13 int main() {
14   std::array< int, 5 > std_a{1,2,3,4,5};
15   fill_ones( std_a );  // hdc< std::array > == HDC::Hst
16 }
```

__host__ __device__ template macro - Example

```
1  template< typename T, int N >
2  struct hstdev_array {
3    static constexpr HDC hdc = HDC::HstDev;
4    __host__ __device__ T& operator[]( int idx ){ return data_[ idx + 1 ]; }
5     /* ... */
6  };
7
8  __global__ void kernel() {
9    hstdev_array< int, 5 > hd_a{1,2,3,4,5}
10   fill_ones( hd_a );  // hdc< hstdev_array > == HDC::HstDev
11 }
12
13 int main() {
14   std::array< int, 5 > std_a{1,2,3,4,5};
15   fill_ones( std_a );  // hdc< std::array > == HDC::Hst
16 }
```

__host__ __device__ template macro - Example

```
1  template< typename T, int N >
2  struct hstdev_array {
3    static constexpr HDC hdc = HDC::HstDev;
4    __host__ __device__ T& operator[]( int idx ){ return data_[ idx + 1 ]; }
5     /* ... */
6  };
7
8  __global__ void kernel() {
9    hstdev_array< int, 5 > hd_a{1,2,3,4,5}
10   fill_ones( hd_a );  // hdc< hstdev_array > == HDC::HstDev
11 }
12
13 int main() {
14   std::array< int, 5 > std_a{1,2,3,4,5};
15   fill_ones( std_a );  // hdc< std::array > == HDC::Hst
16 }
```

__host__ __device__ template macro - Example

```
1  template< typename T, int N >
2  struct hstdev_array {
3    static constexpr HDC hdc = HDC::HstDev;
4    __host__ __device__ T& operator[]( int idx ){ return data_[ idx + 1 ]; }
5     /* ... */
6  };
7
8  __global__ void kernel() {
9    hstdev_array< int, 5 > hd_a{1,2,3,4,5}
10   fill_ones( hd_a );  // hdc< hstdev_array > == HDC::HstDev
11 }
12
13 int main() {
14   std::array< int, 5 > std_a{1,2,3,4,5};
15   fill_ones( std_a );  // hdc< std::array > == HDC::Hst
16 }
```

__host__ __device__ template macro - Example

```
1  template< typename T, int N >
2  struct hstdev_array {
3    static constexpr HDC hdc = HDC::HstDev;
4    __host__ __device__ T& operator[]( int idx ){ return data_[ idx + 1 ]; }
5     /* ... */
6  };
7
8  __global__ void kernel() {
9    hstdev_array< int, 5 > hd_a{1,2,3,4,5}
10   fill_ones( hd_a );  // hdc< hstdev_array > == HDC::HstDev
11 }
12
13 int main() {
14   std::array< int, 5 > std_a{1,2,3,4,5};
15   fill_ones( std_a );  // hdc< std::array > == HDC::Hst
16 }
```

`__host__ __device__` *template macro* - Consequences

- Debugging is hard

The dark path

The dark path:
Function dispatch triple

Function dispatch triple - Problem

```
1  template< typename Container >
2  __host__ __device__ constexpr
3  void fill_ones( Container & ct ) {
4      for ( auto & x : ct ) {
5          x = 1;
6      }
7  }
8
9  #include <vector>
10 int main() {
11     std::vector< int > v{1,2,3,4,5};
12     fill_ones( v );
13 }
```

```
<source>(4): warning #20014-D: calling a __host__ function
from a __host__ __device__ function is not allowed
```

nvcc 12.5

Function dispatch triple - Solution

1. `__host__ __device__` *template* for dispatcher

Function dispatch triple - Solution

1. `__host__ __device__` *template* for dispatcher
2. Dispatcher forwards arguments to *one* `__host__ __device__` function.

Function dispatch triple - Solution

1. `__host__ __device__` *template* for dispatcher
2. Dispatcher forwards arguments to *one* `__host__ __device__` function.
3. *Disable cuda warnings*

Function dispatch triple - Solution

1. *`__host__ __device__ template`* for dispatcher
2. Dispatcher forwards arguments to *one* `__host__ __device__` function.
3. *Disable cuda warnings*
4. Take care of parentheses and commata

Function dispatch triple - Solution

1. `__host__ __device__` *template* for dispatcher
2. Dispatcher forwards arguments to *one* `__host__ __device__` function.
3. *Disable cuda warnings*
4. Take care of parentheses and commata
5. Enable if trickery with C++17

Function dispatch triple - Solution

1. `__host__ __device__` *template* for dispatcher
2. Dispatcher forwards arguments to *one* `__host__ __device__` function.
3. *Disable cuda warnings*
4. Take care of parentheses and commata
5. Enable if trickery with C++17
6. Empty arguments in macros

Function dispatch triple - Example

```
#define function_dispatch_macro( ... )
    OVERLOADED_MACRO( host_device_pragma_macro, __VA_ARGS__ )
#define function_dispatch_macro_2 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_3 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_4( TemplArg, hd_, signature, wrapper )                \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Hst )  \
      __host__             DEPAREN(signature) { DEPAREN(wrapper) }                    \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev )  \
               __device__ DEPAREN(signature) { DEPAREN(wrapper) }                     \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev )  \
      __host__ __device__ DEPAREN(signature) { DEPAREN(wrapper) }
```

Function dispatch triple - Example

```
#define function_dispatch_macro( ... )
    OVERLOADED_MACRO( host_device_pragma_macro, __VA_ARGS__ )
#define function_dispatch_macro_2 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_3 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_4( TemplArg, hd_, signature, wrapper )                \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Hst )  \
      __host__             DEPAREN(signature) { DEPAREN(wrapper) }                    \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev )  \
               __device__ DEPAREN(signature) { DEPAREN(wrapper) }                     \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev )  \
      __host__ __device__ DEPAREN(signature) { DEPAREN(wrapper) }
```

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

Function dispatch triple - Example

```
#define function_dispatch_macro( ... )
    OVERLOADED_MACRO( host_device_pragma_macro, __VA_ARGS__ )
#define function_dispatch_macro_2 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_3 /* ... */ \
                                  /* ... */
#define function_dispatch_macro_4( TemplArg, hd_, signature, wrapper )                \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Hst ) \
      __host__              DEPAREN(signature) { DEPAREN(wrapper) }                   \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev ) \
              __device__ DEPAREN(signature) { DEPAREN(wrapper) }                      \
    template< DEPAREN(TemplArg), HDC hdc = DEPAREN(hd_) > requires( hdc == HDC::Dev ) \
      __host__ __device__ DEPAREN(signature) { DEPAREN(wrapper) }
```

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

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

function_dispatch_macro( (typename T), (hdc< T >),
                         (void wrap( T && t )),
                             (return wrap_impl( std::forward< T >( t ) );) )
```

Function dispatch triple - Consequences

\+ Solves the former problem

−− Much too complicated

? Other realizations?

Cuda proposal

Cuda proposal:

Conditional `__host__ __device__`

Proposal: *Conditional __host__ __device__*

- `__host__ __device__` annotations accept a boolean parameter.

```
1 template< typename T >
2 __host__( hdc<T> == HDC::Hst )
3 __device__( hdc<T> == HDC::Dev )
4 void wrap() {
5     T{}.func();
6 }
```

Proposal: *Conditional* `__host__ __device__`

- `__host__ __device__` annotations accept a boolean parameter.

```
1 template< typename T >
2 __host__( hdc<T> == HDC::Hst )
3 __device__( hdc<T> == HDC::Dev )
4 void wrap() {
5     T{}.func();
6 }
```

+ pure language extension, currently ill-formed today, no existing code will break.
+ Easy to use.

Cuda proposal:

Forbid bad cross function calls

Proposal: *Forbid bad cross function calls*

- A bad cross function call is always a compilation error
- No special casing of template and non-template functions needed

Thank you

universität wien

DIMETOR

FWF

Paper: Thomas Mejstrik, Sebastian Woblistin, *__host__ __device__ – Generic programming in Cuda*, arxiv.org/abs/2309.03912

Slides: `tommsch.com`

Dimetor: Looking for EU-based CUDA developers