Published under CC BY-SA 4.0 DOI:10.5281/zenodo.4021877

HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

Adding CUDA® Support to Cling:JIT Compile to GPUs

S. Ehrig[1], A. Huebl[1,2], A. Naumann[3] and V. Vassilev[3]

[1] Helmholtz-Zentrum Dresden – Rossendorf

[2] Lawrence Berkeley National Laboratory

[3] CERN

2020 Virtual LLVM Developers' Meeting

October 6th-8th 2020

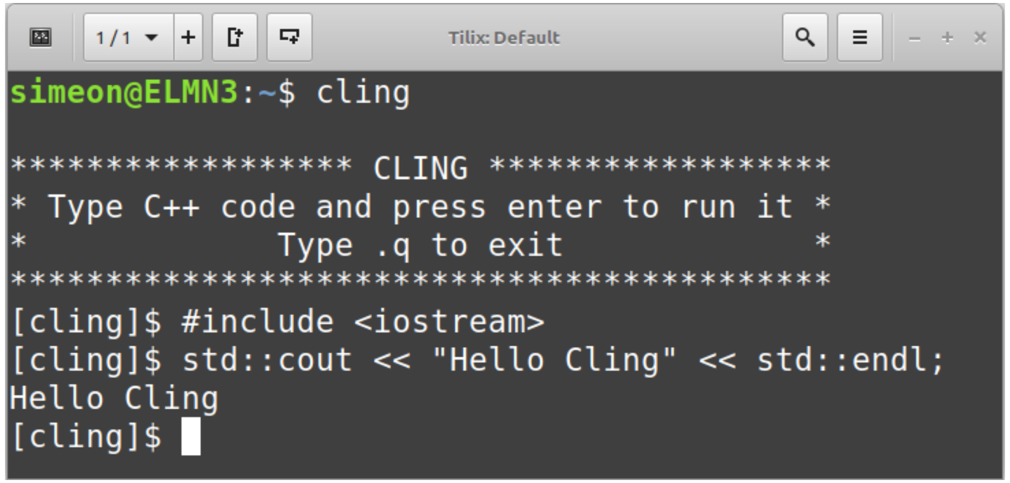
Research Group Computer Assisted Radiation Physics · FWKT · Simeon Ehrig · s.ehrig@hzdr.de · www.hzdr.de

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

DRESDEN
concept

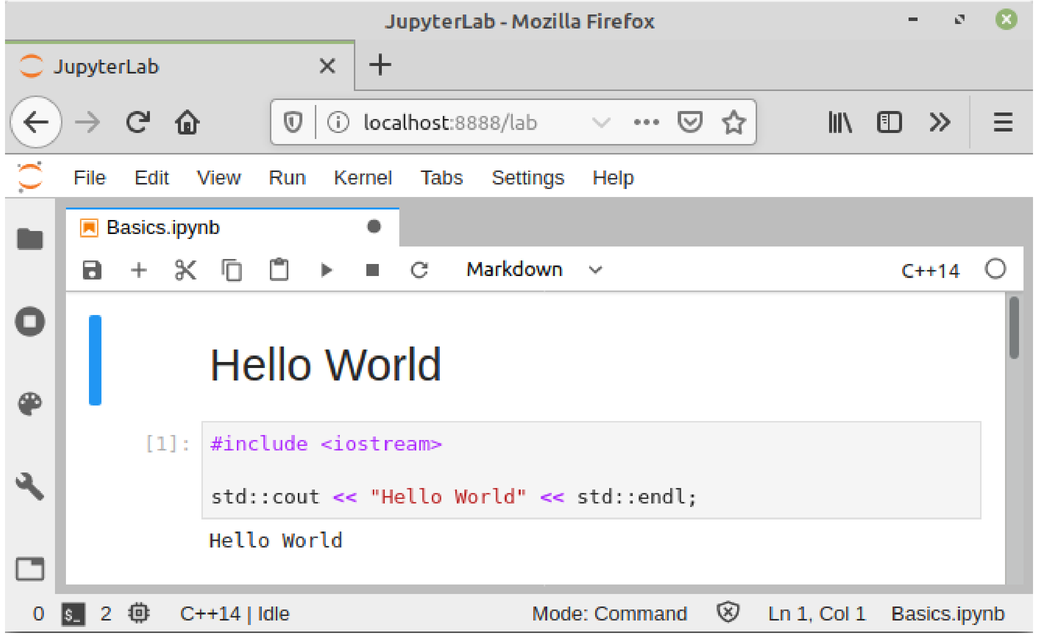
Introduction

Using Cling




```
#include "cling/Interpreter/Interpreter.h"

int main(int argc, char *argv){
        auto cling = cling::Interpreter(argc, argv);
        return 0;
}
```

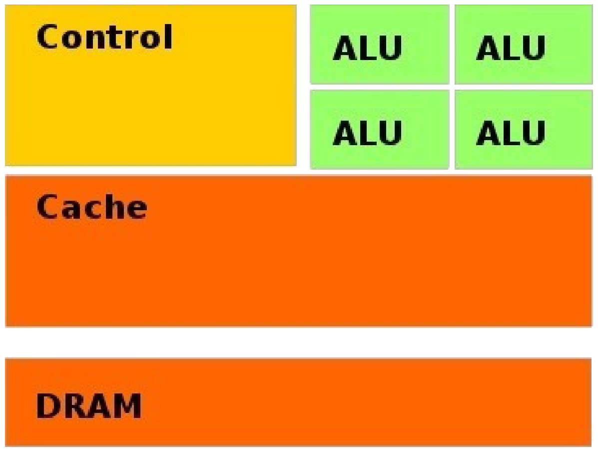


Properties

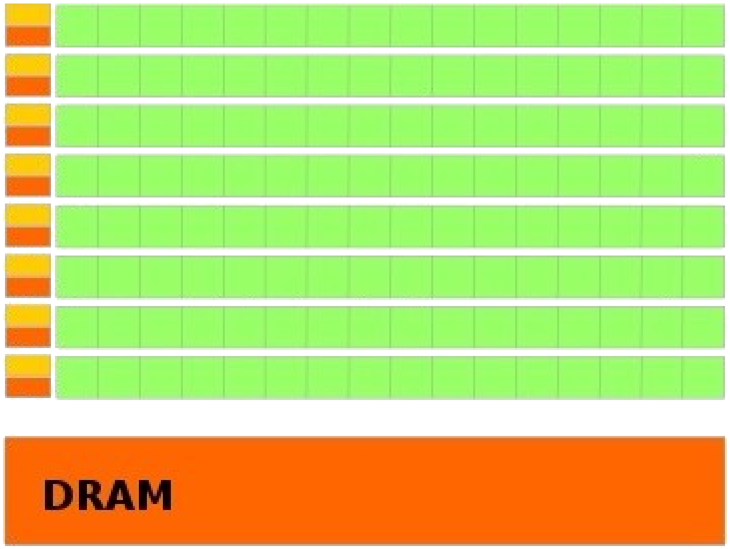
- Read-Eval-Print loop principle
- Does not interpret → the code is JIT compiled
- Fully compatible to existing libraries
  - Can include header files, load unmodified shared libraries and JIT compile C++ source code
- Modifications on syntax and semantic of C++
  - No main() function → everything in global space
  - Missing semicolon at the end of the statement will print the return value
  - Just allowed in the Cling terminal interface or Jupyter Notebook

CPU/GPU Model

CPU

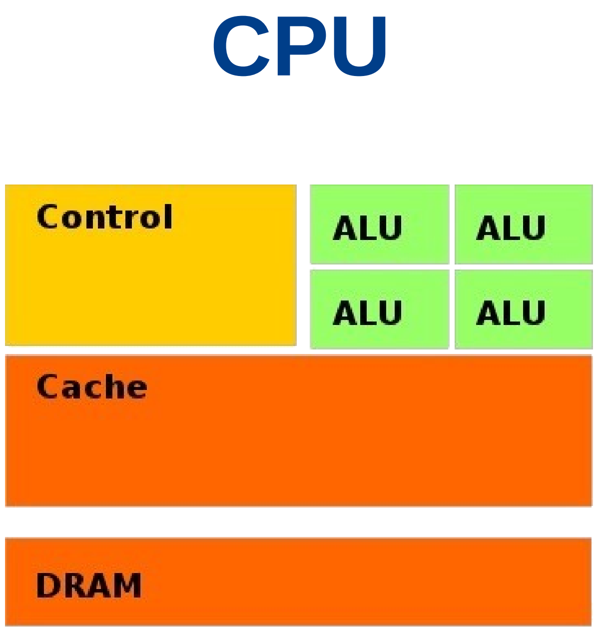


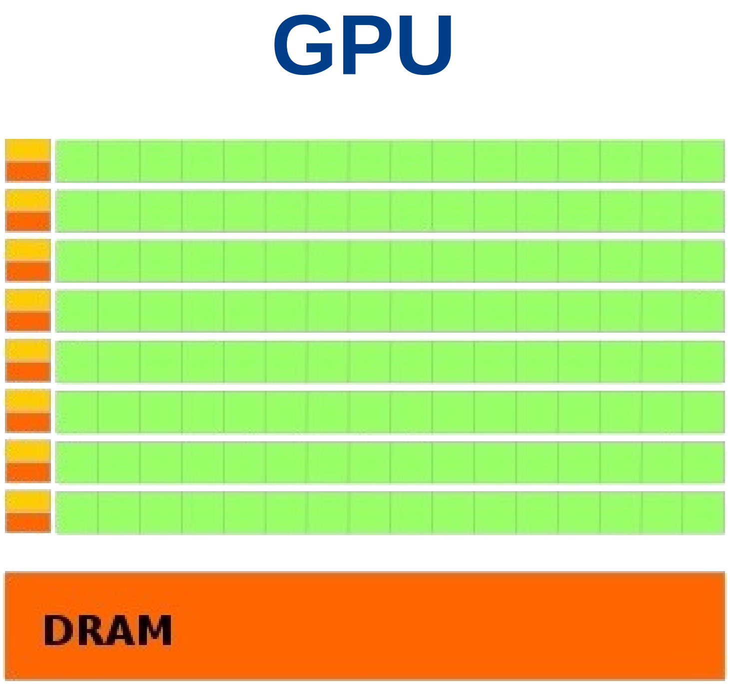
GPU



Sources: Nvidia. *CUDA Reference Guide*

CPU/GPU Model





- Why GPU: Better performance for certain algorithms
- Why CUDA: existing algorithms and widest distribution

Sources: Nvidia. *CUDA Reference Guide*

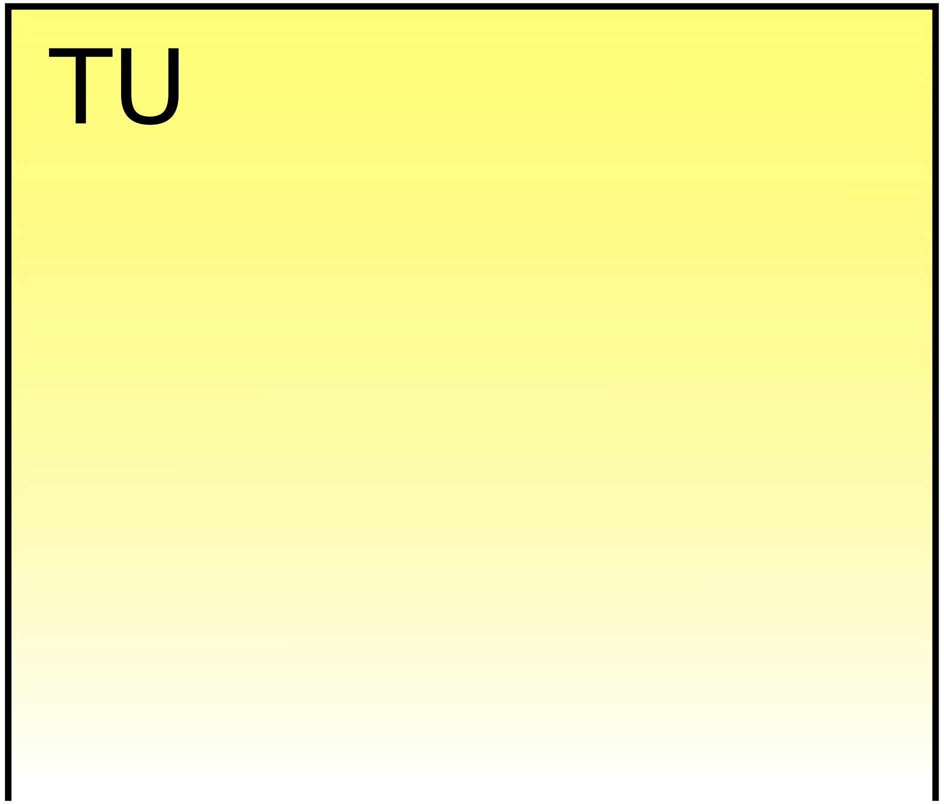
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Basic concept

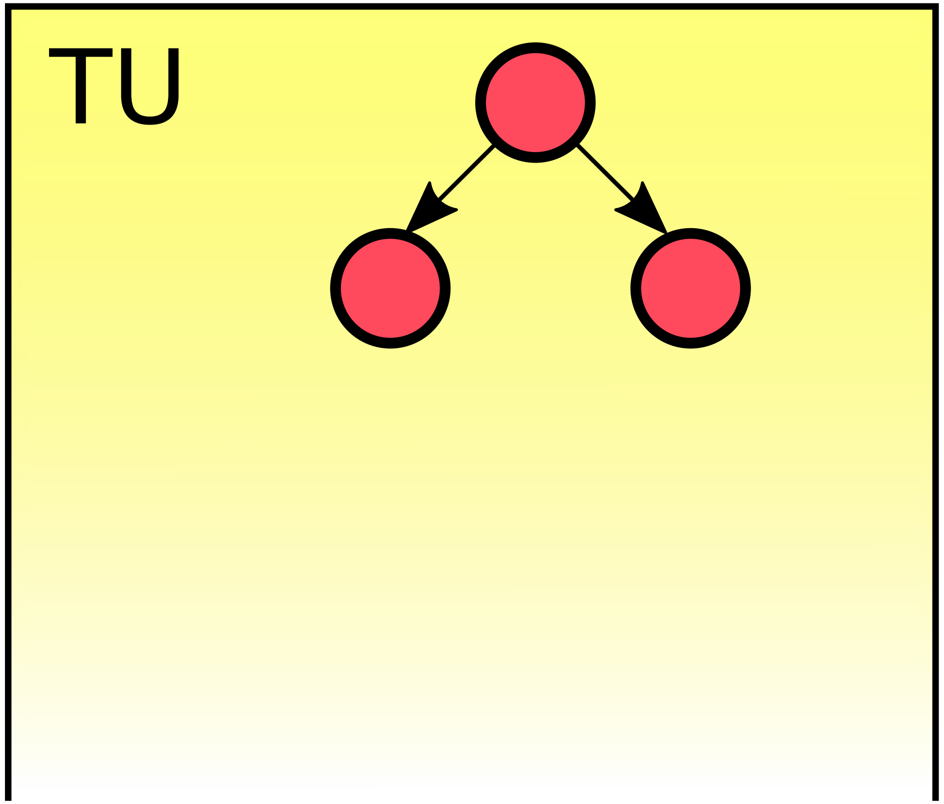
CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

DRESDEN concept

Extendable application flow

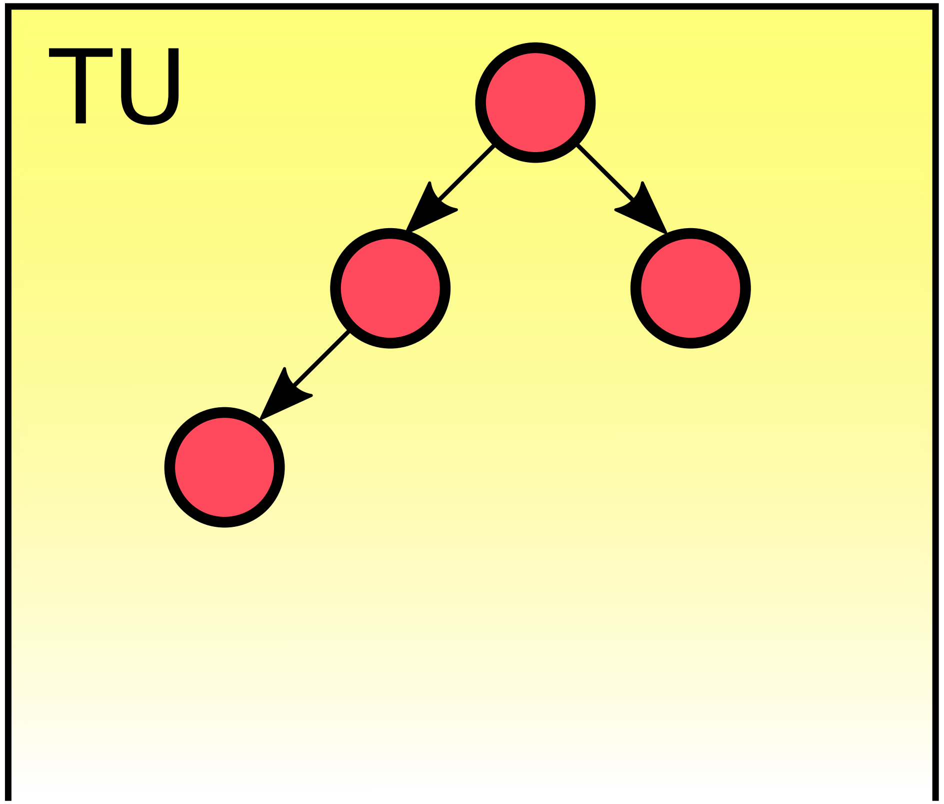


Extendable application flow



Transaction 1
(initial state)

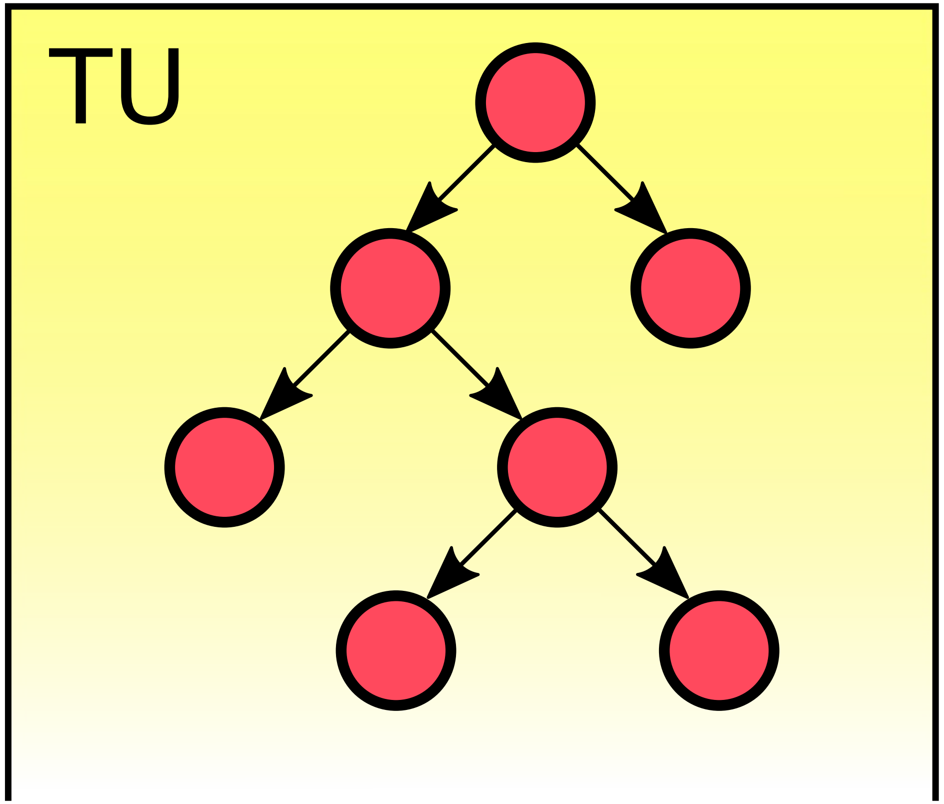
Extendable application flow



Transaction 1
(initial state)

Transaction 2
int i = 3;

Extendable application flow



Transaction 1
(initial state)

Transaction 2
int i = 3;

Transaction 3
i = i + 3;

Creating a single transaction

Input

Metaparser

Parser

AST-Transformer

Code Generator

Executor

Creating a single transaction

Input

foo()

```
****************** CLING ******************
* Type C++ code and press enter to run it *
*             Type .q to exit             *
*******************************************
[cling]$ int foo() { return 3;}
[cling]$ foo()
```

Class references:
cling::UserInterface

Creating a single transaction

Input

Metaparser

```
void __cling_Un1Qu32(void* vpClingValue)
{
   foo();
}
```

Tasks of the Metaparser

- Transforms source code
- Detects meta commands
  - e.g.: .L libz.so
  - Linking the shared library z

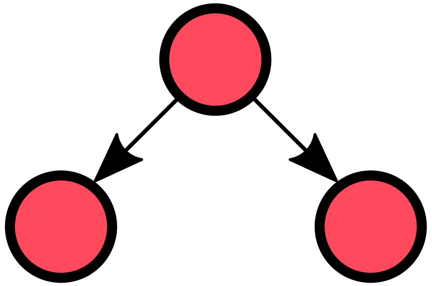
Class references:
cling::Metaprocessor
cling::utils::getWrapPoint

Creating a single transaction

Input

Metaparser

Parser



Properties of the Parser
- Non-modified Clang parser
- Needs valid C++ code

Class references:
cling::IncrementalParser
clang::Parser
clang::ASTConsumer

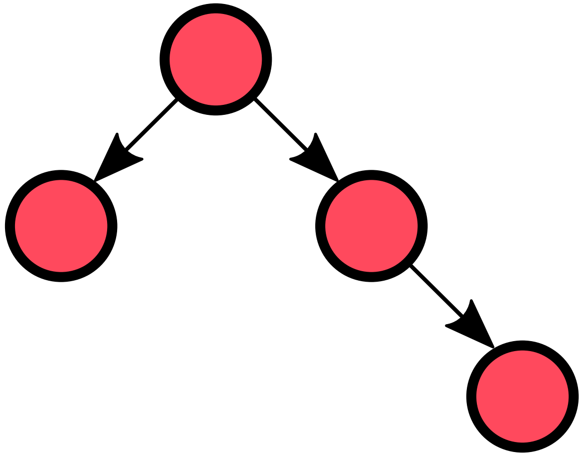
Creating a single transaction

Input

Metaparser

Parser

AST-Transformer



Tasks of the AST-Transformer

- Enables functionality
  - e.g. CUDA device kernel inliner
- Adds error protection
  - e.g. nullptr access
- Adds cling specific features
  - Shadow namespaces for redefinition

Class references:
cling::ASTTransformer
llvm::legacy::PassManager

Creating a single transaction

Input

Metaparser

Parser

AST-Transformer

Code Generator

```
push    rbp
mov     rbp, rsp
sub     rsp, 8
mov     QWORD PTR [rbp-8], rdi
call    foo()
nop
leave
ret
```

Class references:
- cling::IncrementalJIT
- llvm::orc

Creating a single transaction

Input

Metaparser

Parser

AST-Transformer

Code Generator

Executor

foo()
(int) 3

```
****************** CLING ******************
* Type C++ code and press enter to run it *
*             Type .q to exit             *
*******************************************
[cling]$ int foo() { return 3;}
[cling]$ foo()
(int) 3
[cling]$
```

Class references:
cling::IncrementalExecutor

Challenges

Challenges

1) Is interactive CUDA C++ possible?
   - The driver API allows it, but we want to use the runtime API
   - Answered with many experiments with modified LLVM IR and prototypes

Challenges

1) Is interactive CUDA C++ possible?
   - The driver API allows it, but we want to use the runtime API
   - Answered with many experiments with modified LLVM IR and prototypes
2) How does Cling understand CUDA C++?
   - CUDA C++ is not valid C/C++ → e.g. foo<<<1,1>>>();
   - Google‘s GPUCC project solved the problem for the compiler pipeline → only needed to be activated in Cling
   - Metaparser does not use the Clang parser

Sources: Google. gpucc: *An Open-Source GPGPU Compiler*

Challenges

1) Is interactive CUDA C++ possible?
   - The driver API allows it, but we want to use the runtime API
   - Answered with many experiments with modified LLVM IR and prototypes
2) How does Cling understand CUDA C++?
   - CUDA C++ is not valid C/C++ → e.g. foo<<<1,1>>>();
   - Google's GPUCC project solved the problem for the compiler pipeline → only needed to be activated in Cling
   - Metaparser does not use the Clang parser
3) How to integrate the device pipeline?
   - Cling was not designed for a second compiler pipeline
   - Solved a lot of different implementation tasks

Sources: Google. gpucc: *An Open-Source GPGPU Compiler*

General Problems

- CUDA is proprietary
  - In general, the documentation is good …
  - … but some details are not documented → black box testing

General Problems

- CUDA is proprietary
  - In general, the documentation is good …
  - … but some details are not documented → black box testing
- Documentation
  - The whole software stack containing Cling, Clang and LLVM is really complex and I had to learn a lot
  - The LLVM documentation is really good
  - The Clang documentation was okay
  - The Cling documentation is rudimentary and there are no other similar projects

General Problems

- CUDA is proprietary
  - In general, the documentation is good …
  - … but some details are not documented → black box testing
- Documentation
  - The whole software stack containing Cling, Clang and LLVM is really complex and I had to learn a lot
  - The LLVM documentation is really good
  - The Clang documentation was okay
  - The Cling documentation is rudimentary and there are no other similar projects
- The CUDA Runtime API was not used interactively until now
  - No experience
  - Some workarounds necessary

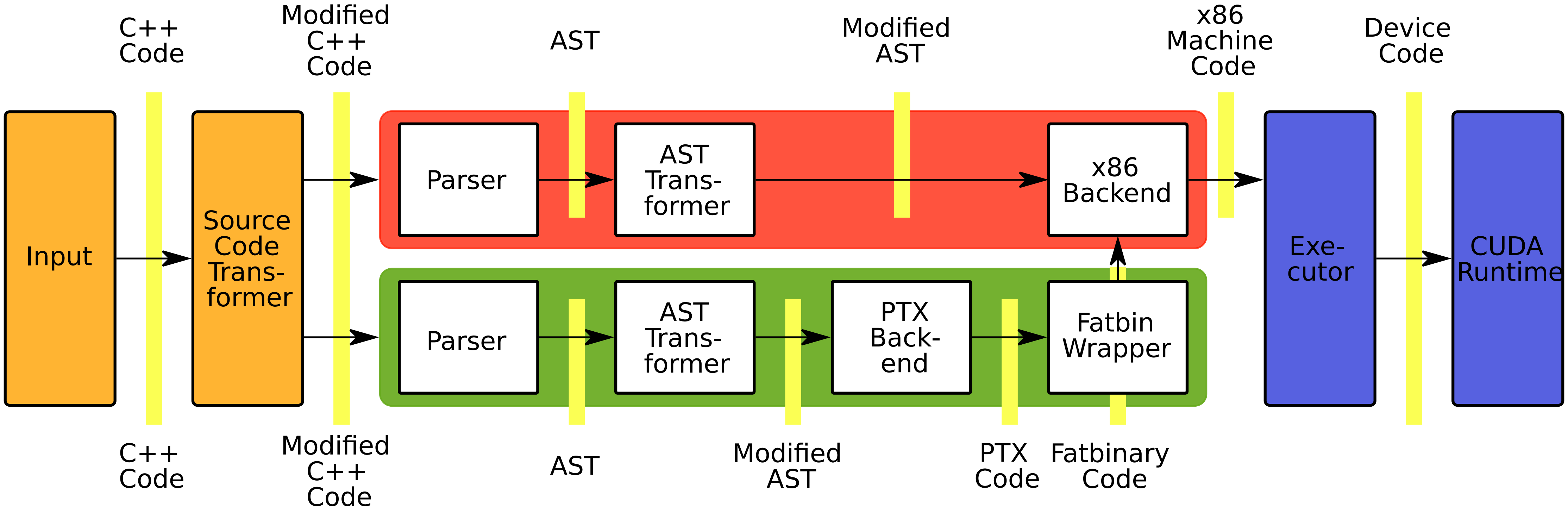
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Implementation

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

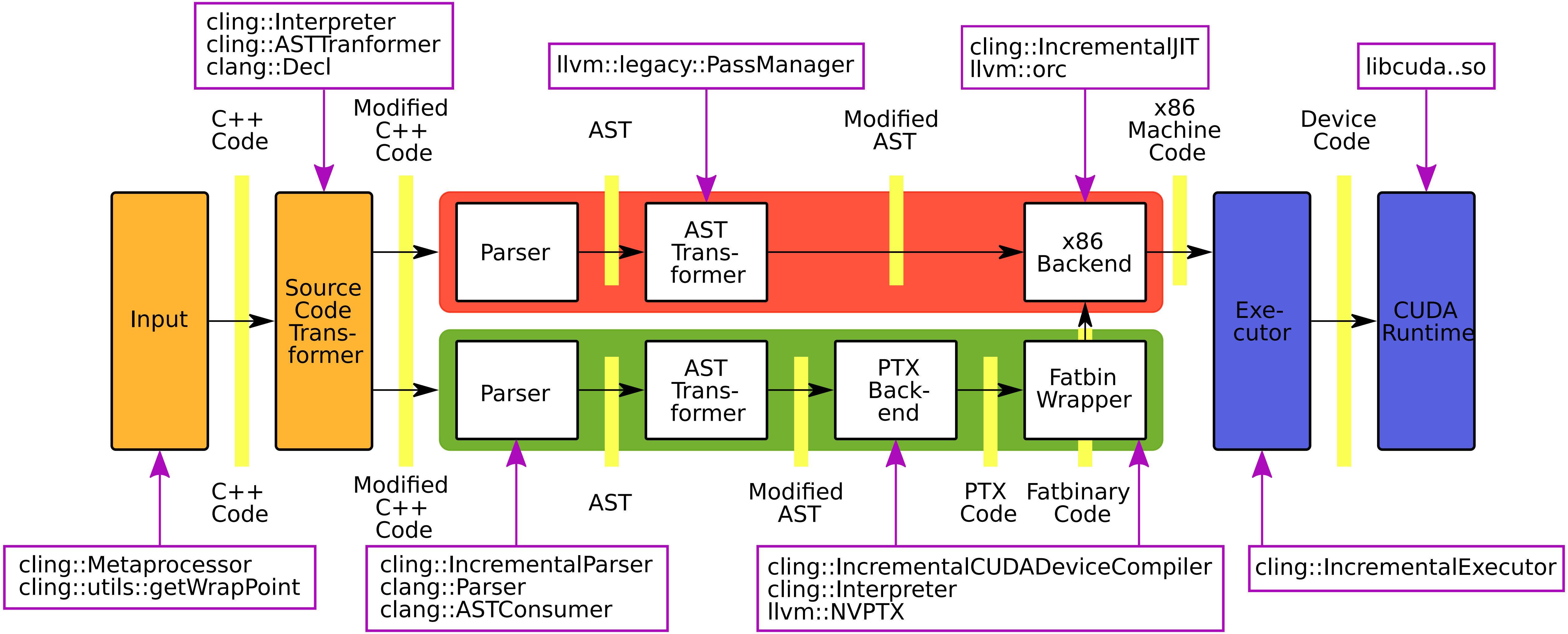
DRESDEN concept

General Implementation



Versions:
- Cling 0.7
- Clang/LLVM 5.0

General Implementation



Detail Problem: Metaparser + CUDA

- Problem
  - The Metaparser is completely self-written and parses the “interactive” C++ semantic and the meta commands of Cling
  - The semantic of C++ is complex, the Cling extension makes it even more complex and the CUDA extension too
  - A lot of implementation work is necessary to cover all cases
- Solution
  - Still looking for an optimum solution
  - The most important cases are covered
  - Raw input mode as workaround
- Possible improvements
  - Modifying the Clang parser to handle the “interactive” C++ semantic of Cling

Function references:
cling::utils::getWrapPoint

Detail Problem: Catching errors

- Problem
  - The interpreter runtime and the user code use the same process and memory space. If a segmentation fault occurs in the user code, the entire interpreter crashes.
- Solution
  - Catch the errors with code analysis before the code is executed.
  - Current solution is not generally applicable
    - e.g. Segmentation faults via indirect pointers

Detail Problem: Updating the Clang/LLVM base

- Problems
  - Each new Clang/LLVM version supports new CUDA versions, C++ features and has a lot of bug fixes especially with respect to CUDA.
  - The C++ API is not stable and changes continuously. The JIT backend is also continuously developed further.
  - Cling requires a patched version of Clang/LLVM.
  - Updating the Clang specific patches causes a lot of work.
- Possible Solution
  - RFC for simple Clang REPL by Vassil Vassilev (August 2020)
  - Move as many REPL specific patches as possible upstream to Clang

What is still missing

- Some C++ and CUDA statements, although supported by Clang 5.0 on CUDA 8.0
  - e.g. CUDA `__constant__` memory
  - and CUDA global `__device__` memory
- Not all Cling features work with CUDA yet
  - e.g. redefinition of kernels via namespace shadowing
- Metaparser does not detect all valid CUDA C++ statements
- Error catching needs to be improved

HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

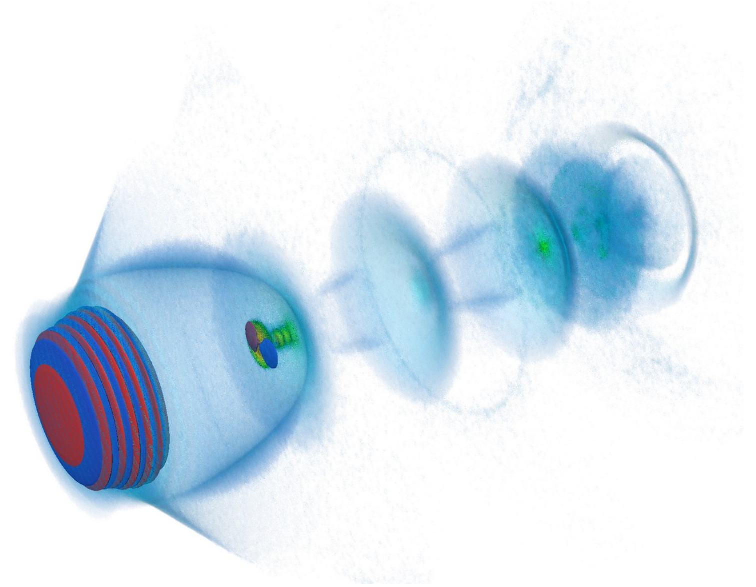
Application Areas

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

DRESDEN concept

Application areas

- Cling was initially developed for **large data analysis** in HEP physics
- Big, **interactive simulation** with GPUs
- **Teaching** GPU programming
- Easing **development** and debugging



https://github.com/alpaka-group/alpaka

https://github.com/ComputationalRadiationPhysics/picongpu/

Summary

Summary

- First interactive C++ JIT compiler for the CUDA runtime API
- Added a dual compiler instance concept to Cling, which can be used for other GPU APIs (AMD, Intel)
- Most features already upstream in cling master
- Interactive CUDA C++ in Jupyter Notebook enables new areas of application
  - Data analysis in notebooks with GPUs
  - Big, interactive simulations with GPUs
  - Teaching GPU programming
  - Easing development and debugging

Versions:
Cling 0.7
Clang/LLVM 5.0

Detail Problem: Clang CUDA expected a completed TU

- Problem
  - How does CUDA register kernels? No official documentation.
  - The Compiler generates the __cuda_module_ctor and __cuda_module_dtor functions which register and unregister the kernels and register the functions in the global constructor and destructor.
  - Cling creates the functions for each transaction. But Cling is lazy and only translates the first occurrence of __cuda_module_ctor into machine code and reuses it for each transaction. So you can only register one kernel in each cling instance.
- Solution
  - Make the function names __cuda_module_ctor and __cuda_module_dtor unique.

Class references:
UnqiueCUDACtorDtorName

Detail Problem: Embedding the Fatbin Generator

- Problem
  - The LLVM IR code of the device compiler pipeline is translated into Nvidia PTX code (a kind of assembler) and embedded in a fatbinary file (struct with meta data and ptx code).
  - Compared to the PTX code, the fatbin struct is not officially specified. Only Nvidia’s external fatbin tool is available for embedding PTX code in the fatbin struct.
- Solution
  - Reimplementation of the fatbin tool based on a header file from the CUDA SDK in “llvm-project-cxxjit”
  - Thanks to Hal Finkel

Class references:
cling::IncrementalCUDADeviceCompiler

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING DRESDEN concept HZDR

demo

September 9, 2020

1 Hello World

[1]:
```
#include <iostream>

std::cout << "Hello World" << std::endl;
```

```
Hello World
```

2 Global and Local Variables

[2]:
```
// global variable
int g1 = 1;
```

[3]:
```
// local variable
{
    int l1 = 2;
}
```

[4]:
```
std::cout << l1 << std::endl;
```

```
input_line_11:2:15: error: use of undeclared identifier
'l1'
 std::cout << l1 << std::endl;
              ^

input_line_11:2:15: error: use of undeclared
identifier 'l1'
 std::cout << l1 << std::endl;
              ^



        Interpreter Error:
```

[5]:
```
std::cout << g1 << std::endl;
{
    // hide global variable
    int g1 = 3;
    std::cout << g1 << std::endl;
}
std::cout << g1 << std::endl;
```

```
1
3
1
```

3 Including and Linking

[6]:
```
%%file foo.hpp
#pragma once

namespace foo {
    int bar();
}
```

```
Writing foo.hpp
```

[7]:
```
%%file foo.cpp
#include "foo.hpp"

int foo::bar() { return 42; }
```

```
Writing foo.cpp
```

[8]:
```
!gcc -shared foo.cpp -o foo.so
```

[9]:
```
#include "foo.hpp"
```

[10]:
```
#pragma cling(load "foo.so")
```

[11]:
```
foo::bar()
```

[11]: 42

4 CUDA

[12]:
```
template <int A, int B>
class CUDA {
    int host;
    int *device;
```

```
    public:
    static __global__ void kernel(int *out){
        *out = A + B;
    }

    CUDA(){
        cudaMalloc((void**)&device, sizeof(int));
    }

    ~CUDA(){
        cudaFree(device);
    }

    int compute(){
        kernel<<<1,1>>>(device);
        cudaMemcpy(&host, device, sizeof(int), cudaMemcpyDeviceToHost);
        return host;
    }
};
```

[13]:

```
CUDA<19,23> c;
```

[14]:

```
c.compute()
```

[14]: 42

[ ]:

```

```