HZDR

HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Cling's CUDA Backend: Interactive GPU development with CUDA C++

Compiler as a Service project @ Princeton University

Simeon Ehrig

s.ehrig@hdzr.de

March 4th 2021

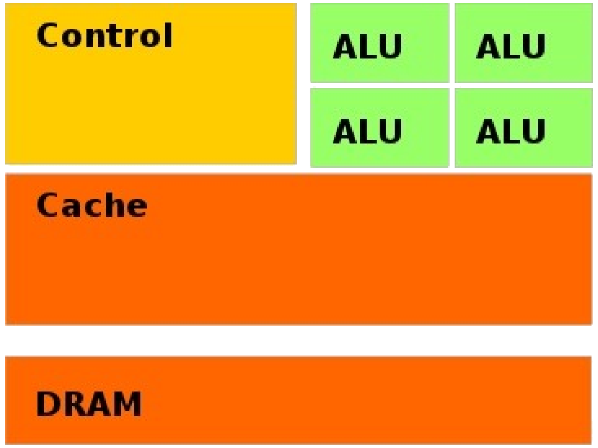
CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

DRESDEN concept

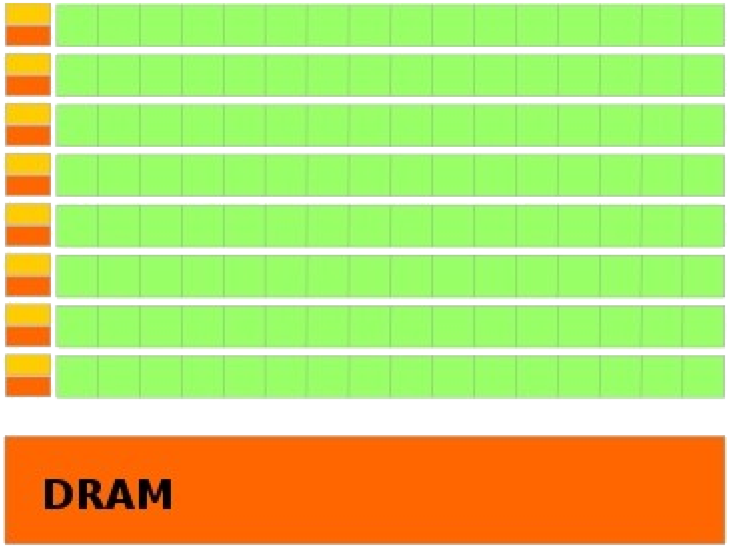
Developing GPU applications

CPU/GPU Model

CPU

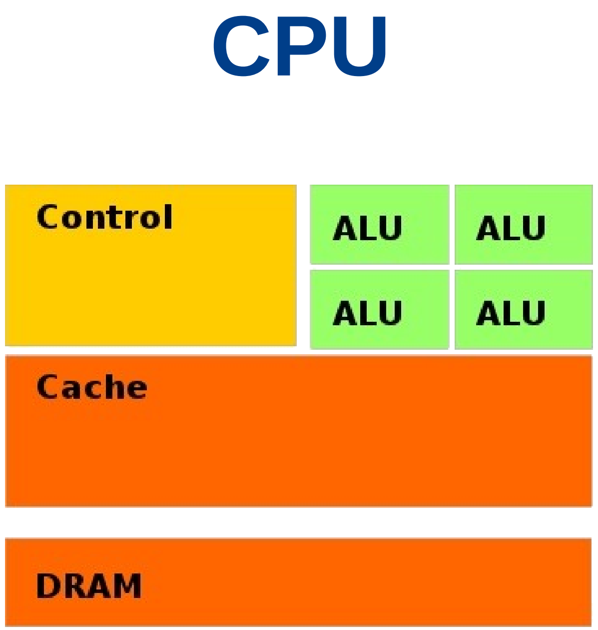


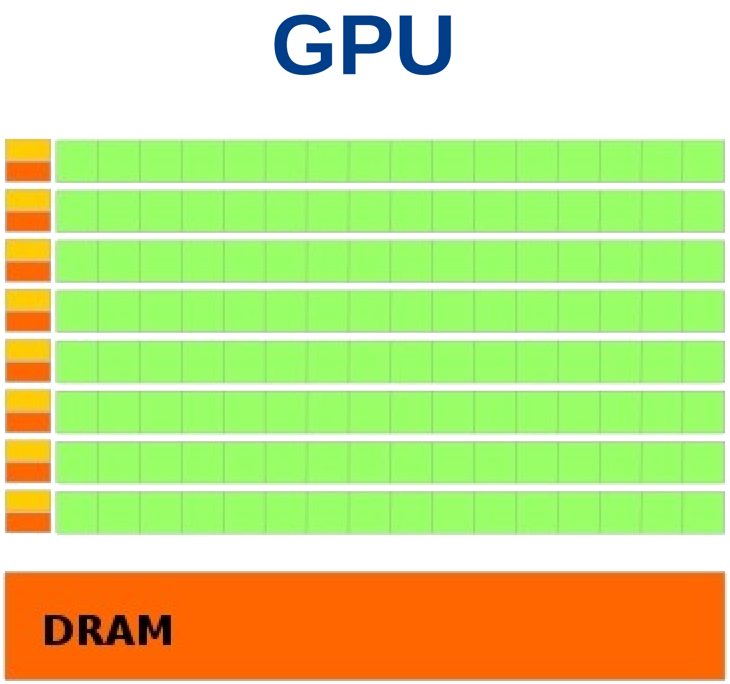
GPU



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*

Writing CUDA code

```
//function, which will run on GPU
template <typename T>
__global__ void copy_kernel(T * in, T * out, unsigned int N){
	int id = blockIdx.x * gridDim.x + threadIdx.x;
	if(id < N)
		out[id] = in[id];
}

int main(){
	// …
	// copy memory from cpu to gpu
	cudaMemcpy(device_in, host_in, sizeof(int) * N, cudaMemcpyHostToDevice);

	// start function on GPU with 32 threads an 10 blocks
	copy_kernel<int><<<32, 10>>>(a, b, c);

	// copy memory from gpu to cpu
	cudaMemcpy(host_out, device_out, sizeof(int) * N, cudaMemcpyDeviceToHost);
	// …
}
```

Writing CUDA code

Device-Code

```
//function, which will run on GPU
template <typename T>
__global__ void copy_kernel(T * in, T * out, unsigned int N){
	int id = blockIdx.x * gridDim.x + threadIdx.x;
	if(id < N)
		out[id] = in[id];
}
```

Host-Code

```
int main(){
	// …
	// copy memory from cpu to gpu
	cudaMemcpy(device_in, host_in, sizeof(int) * N, cudaMemcpyHostToDevice);

	// start function on GPU with 32 threads an 10 blocks
	copy_kernel<int><<<32, 10>>>(a, b, c);

	// copy memory from gpu to cpu
	cudaMemcpy(host_out, device_out, sizeof(int) * N, cudaMemcpyDeviceToHost);
	// …
}
```

HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Compiler pipeline

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

DRESDEN concept

Parsing and executing a statement

Input

Metaparser

Parser

AST-Transformer

Code Generator

Executor

Parsing and executing a statement

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

Parsing and executing a statement

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

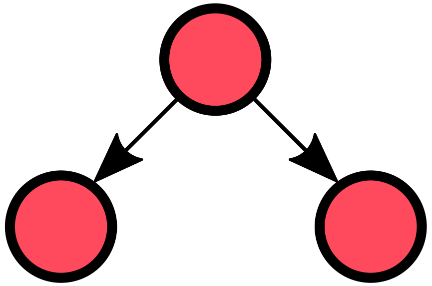
Class references:
  cling::Metaprocessor
  cling::utils::getWrapPoint

Parsing and executing a statement

Input

Metaparser

Parser



Properties of the Parser

- Non-modified Clang parser
- Needs valid C++ code

Class references:

cling::IncrementalParser
clang::Parser
clang::ASTConsumer

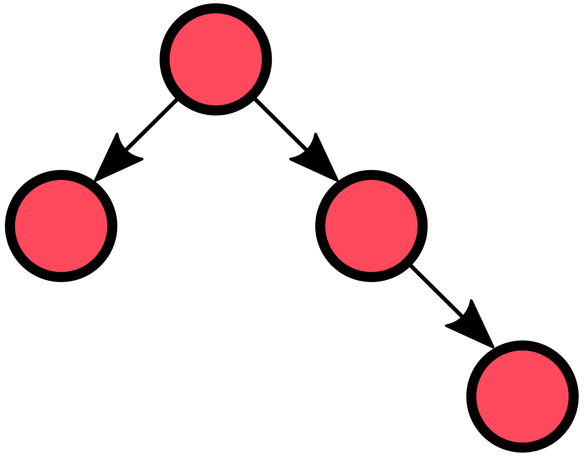
Parsing and executing a statement

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

Parsing and executing a statement

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

Parsing and executing a statement

- 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

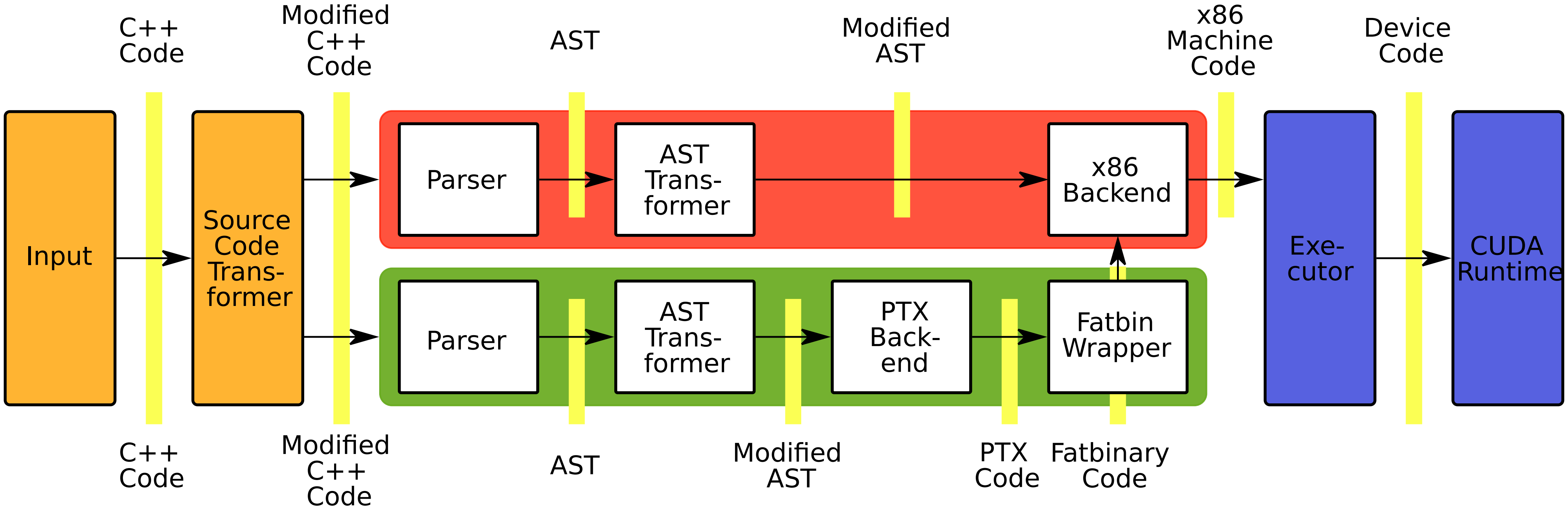
HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

Implementation

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

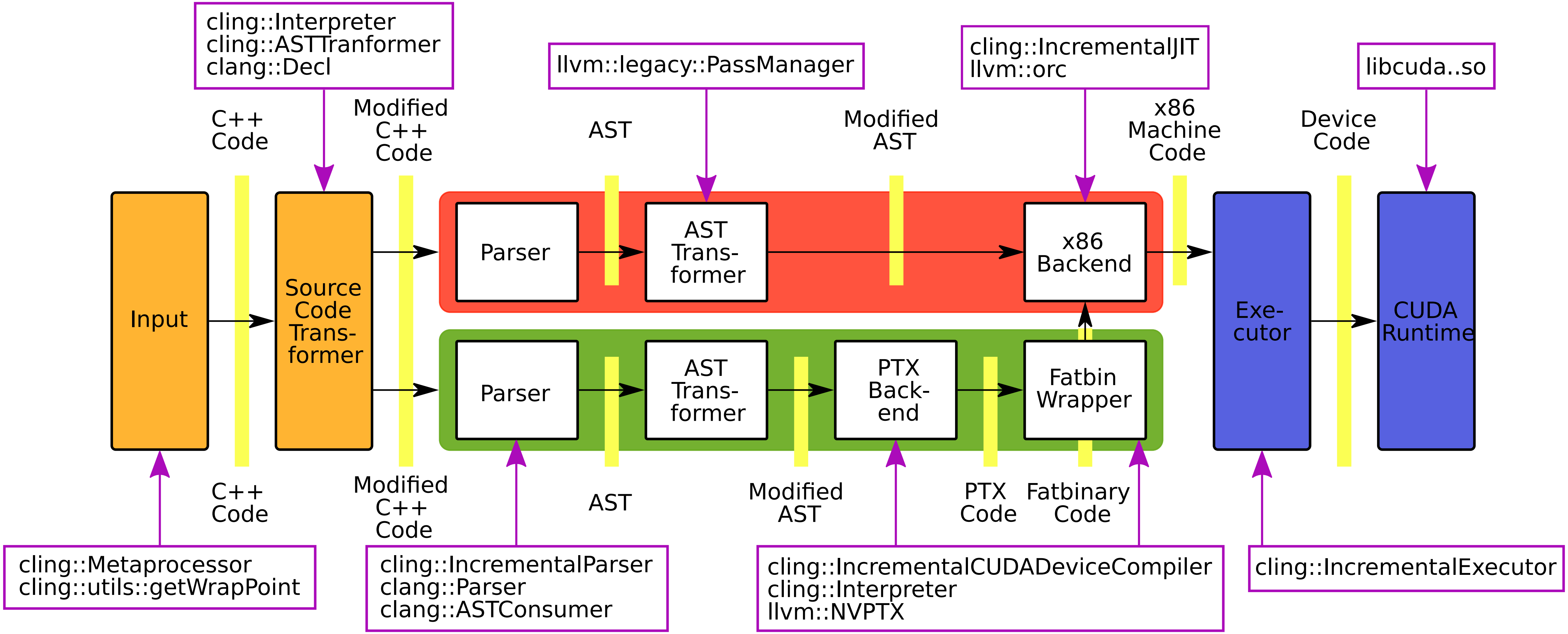
DRESDEN concept

General Implementation – Data flow

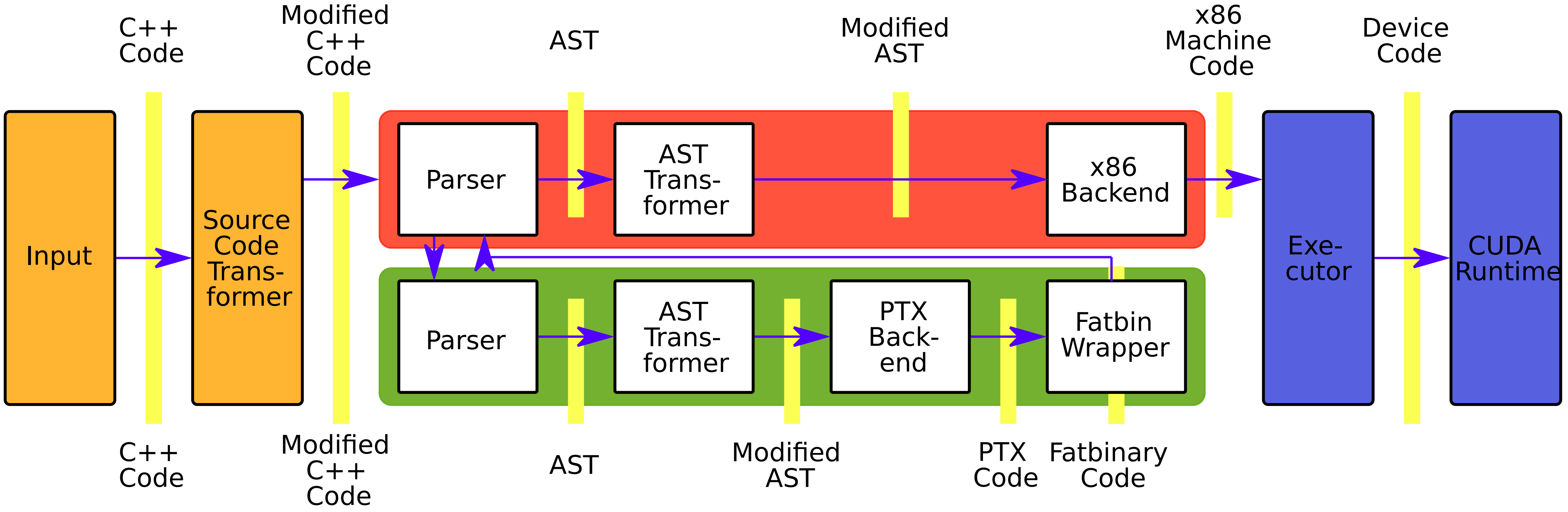


Versions:
- Cling 0.8
- Clang/LLVM 5.0

General Implementation – Data flow



General Implementation – Program flow



Versions:
- Cling 0.8
- Clang/LLVM 5.0

Implementation: Compiling Device Code

- In the begin of cling::Interpreter::process(), parse(), declare() the device compiler pipeline is executed

```
Interpreter::CompilationResult
Interpreter::declare(const std::string& input, Transaction** T/*=0 */) {
  if (!isInSyntaxOnlyMode() && m_Opts.CompilerOpts.CUDAHost)
    m_CUDACompiler->declare(input);

  CompilationOptions CO = makeDefaultCompilationOpts();
  CO.DeclarationExtraction = 0;
  CO.ValuePrinting = 0;
  CO.ResultEvaluation = 0;
  CO.CheckPointerValidity = 0;

  return DeclareInternal(input, CO, T);
}
```

Implementation: Compiling Device Code

- cling::IncrementalCUDADeviceCompiler contains device compiler pipeline
- Uses modified cling::Interpreter for parsing and transforming code and use custom back-end to generate PTX and Fatbin code
- Device compiler pipeline stages
  - Parsing code
  - AST transformations
  - Generating PTX code
  - Wrapping PTX code in Fatbin wrapper
  - Writing to file
  - Return to host compiler pipeline
- The x86 CUDA code generator reads the Fatbin code from file

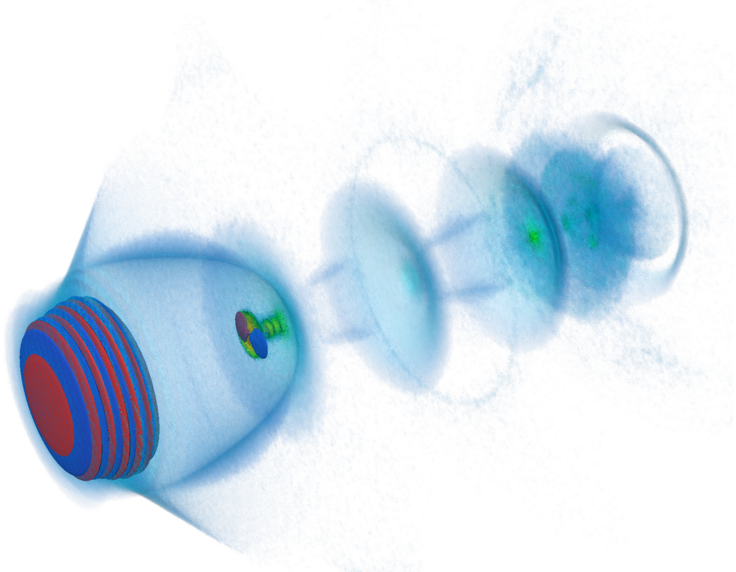
What is still missing

- Some C++ and CUDA statements, although supported by Clang 9.0 on CUDA 10.1
  - 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

Application Areas

Application areas

- **Teaching** GPU programming
- Big, **interactive simulation** with GPUs
- Easing **development** and debugging



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

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

Outlook

Outlook

- Enable CUDA mode in ROOT
- Fixes bugs to run matmul<alpaka::AccGpuCudaRt>
- Add support for __constant__ memory (source code transformer)
- Refactor device compiler to inherited class of cling::Interpreter
- GSoC project: add redefinition support for CUDA mode

Versions:
Cling 0.8
Clang/LLVM 9.0

Wish list

- GPU CI
- Documentation of concepts with linkage to the code (sphinx-doc, llvm user documentation)

Backup

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: 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