A Compute Graph Simulation and Implementation Framework Targeting Versal AI Engines

Jonathan Strobl, Leonardo Solis-Vasquez, Yannick Lavan, Andreas Koch
Presenter: Torben Kalkhof

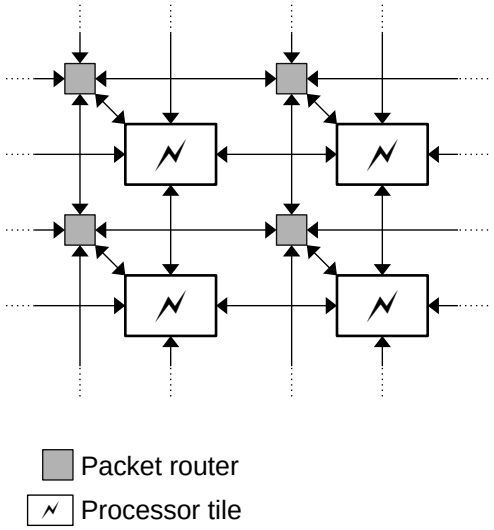
Embedded Systems and Applications Group
Technical University of Darmstadt, Germany

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  COMPUTE_GRAPH constexpr auto the_graph = make_compute_graph_v<[] (
2      IoConnector<float> in1, IoConnector<float> in2
3  ) {
4      IoConnector<float> squared, out;
5
6      squaring_kernel(in1, squared);
7      adder_kernel(squared, in2, out);
8
9      return std::make_tuple(squared, out);
10 }>;
```

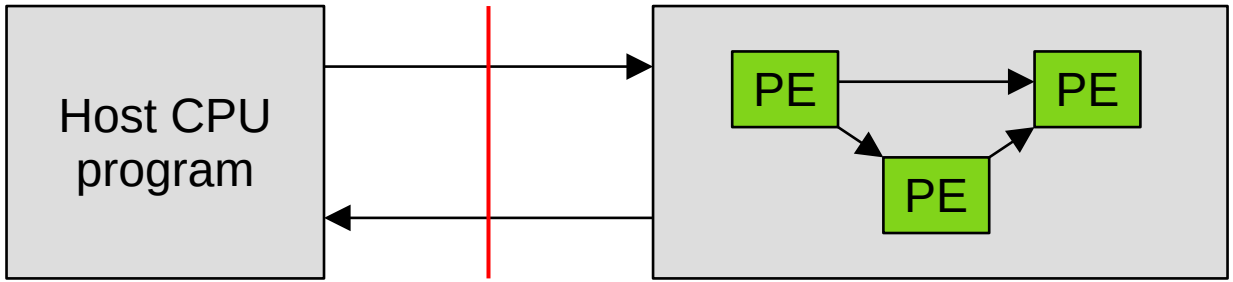
Background

- Specialized accelerators are becoming more prevalent in HPC
  - GPUs, FPGAs, **dataflow architectures**, …
- Example: AMD Versal AI Engines
  - **2D Grid** of VLIW processors
  - Data streaming connections
  - Integration with Versal FPGAs
- Programming usually requires vendor-specific toolchains



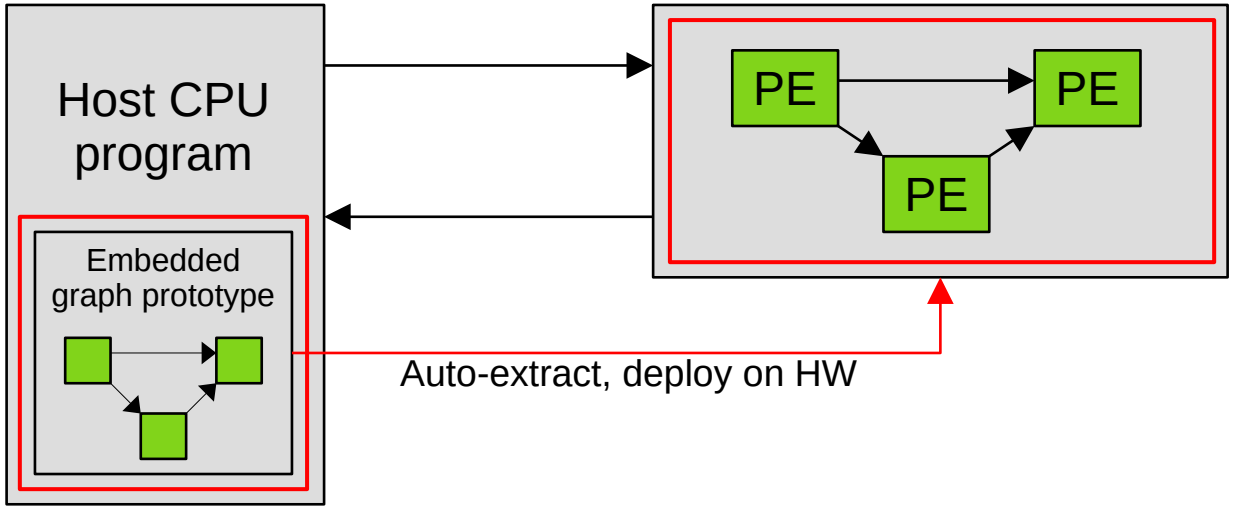
Challenge

- Host and accelerator codebases are separate
  - For AIEs, every compute kernel is a separate program
- High barrier to entry
  - Existing applications must be **split up**
  - **Different toolchains** and debuggers for each part



Contribution

- Let users prototype graphs within an existing application
- Extract graphs from application source code **programmatically**



Previous Work: „Graphtoy“

- **Compute graph simulator** based on C++20 coroutines[1]
  - For simulating **AI Engine** dataflow graphs (and other architectures)

+ Can be **integrated into existing applications**

- **Rapid prototyping** of graphs
- No early codebase split needed

– Graphs must be translated to target architecture **manually**

- Code split still necessary, just delayed

[1] J. Strobl, L. Solis-Vasquez, Y. Lavan, and A. Koch. “Graphtoy: Fast Software Simulation of Applications for AMD’s AI Engines”, ARC 2024

High-Level Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Rework Graphtoy constructs** to bring them closer to AIE syntax
- Use **source-to-source translation** to generate AIE code
  - Utilize **constexpr code execution** to **off-load** graph construction / analysis to the **compiler**
- Requires rewrite of most parts of Graphtoy
  - Now called **cgsim** (Compute Graph Simulator)

Embedded Systems & Applications

Compile-time Code Execution in C++

TECHNISCHE UNIVERSITÄT DARMSTADT

- C++ code is usually compiled to machine code
  - Executes at runtime
- Since C++11, code can be run at **compile time (constexpr)**
  - Results can influence the compilation of other code

```
1  constexpr auto compile_time_fibonacci = [] {
2      std::array<int, 20> result = {1, 1};
3
4      for (int i = 2; i < result.size(); ++i) {
5          result[i] = result[i - 1] + result[i - 2];
6      }
7
8      return result;
9  }();
```


Embedded Systems & Applications

Compile-time Code Execution in C++

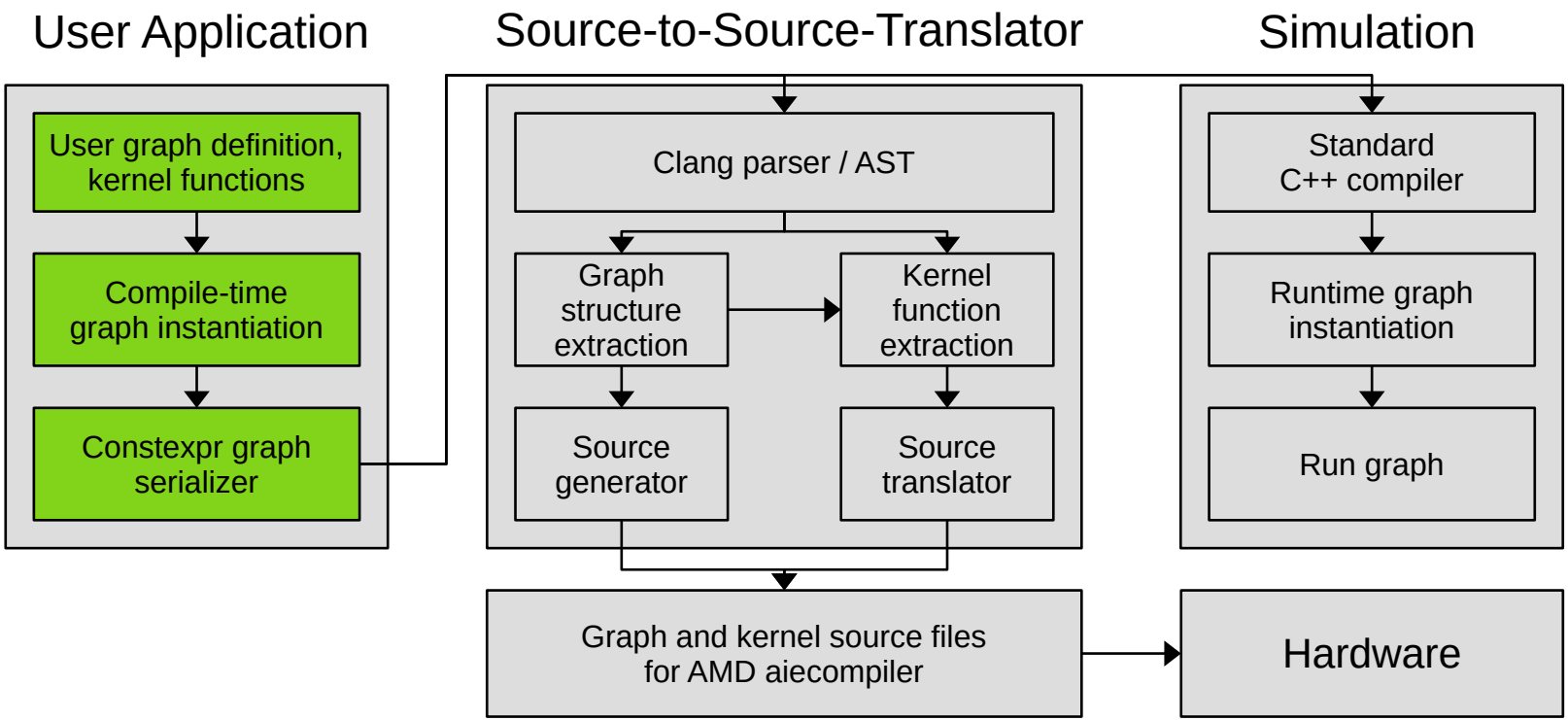
TECHNISCHE UNIVERSITÄT DARMSTADT

- Only a subset of C++ constructs can be executed at compile time
- Dynamic memory allocations are limited
  - Memory can be allocated at compile time
  - But only if it is also **deallocated** at compile time
  - Compile-time allocations **cannot escape compile-time execution**

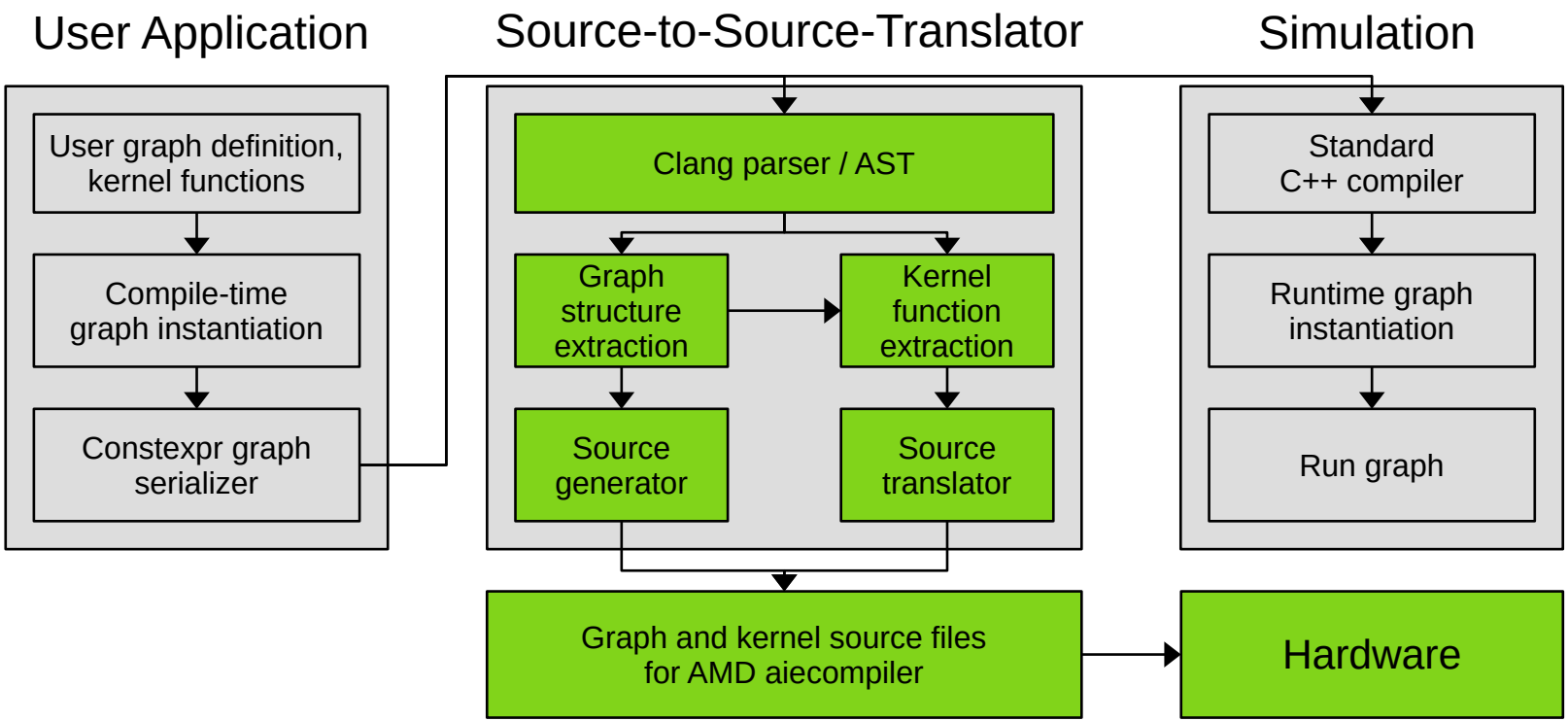
```
1  constexpr auto bad_code = [] {
2      int *result = new int[20];
3
4      /* compute fibonacci numbers */
5
6      return result; // Compile error!
7  }();
```


Embedded Systems & Applications

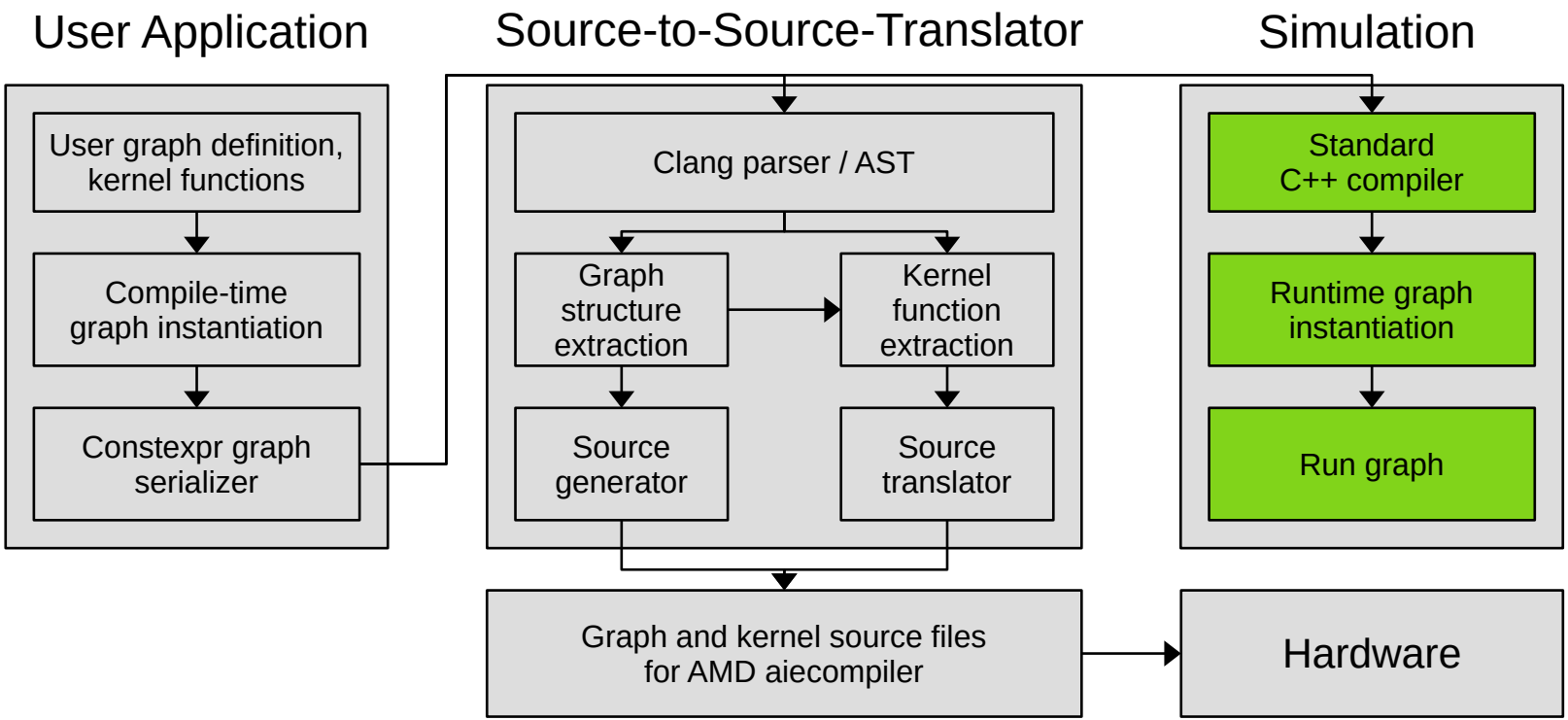
High-Level Overview



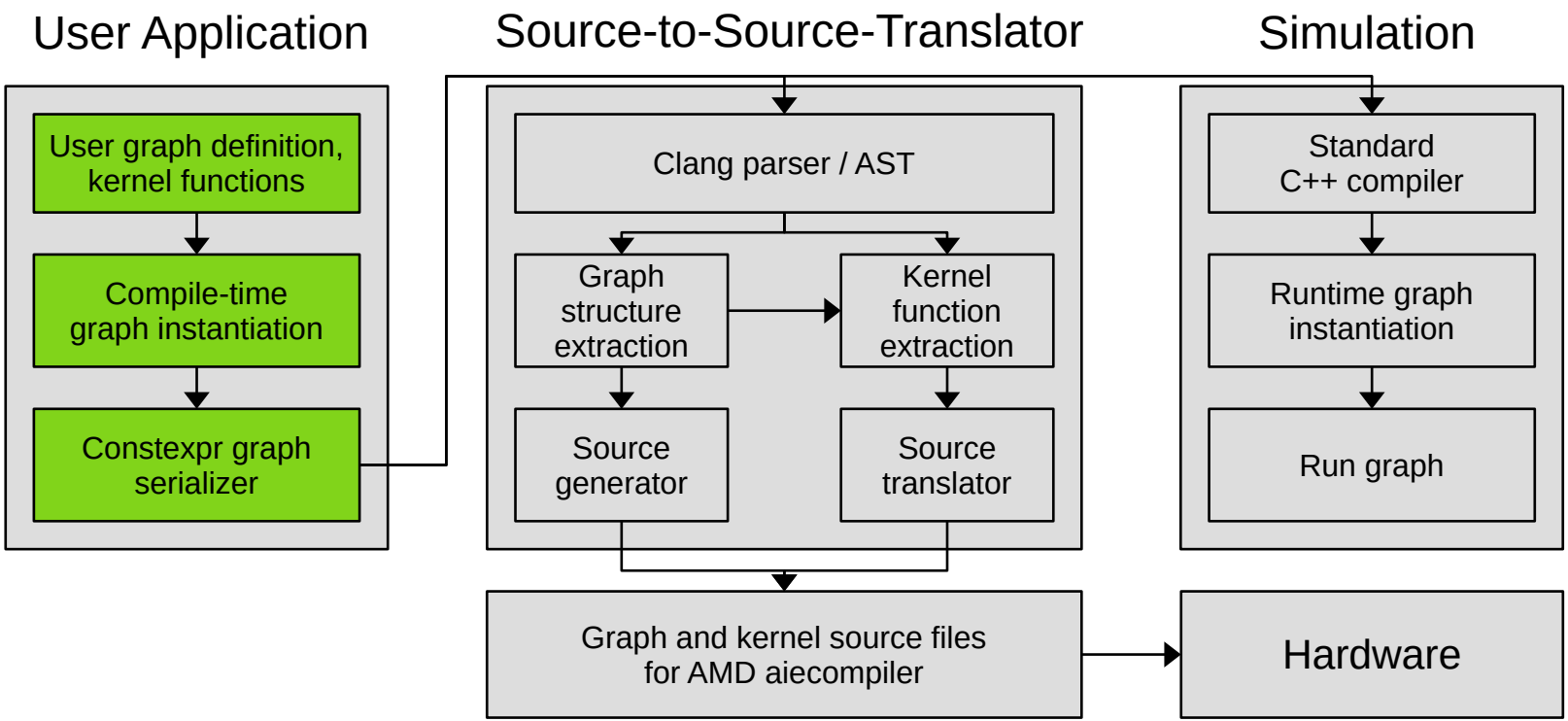
High-Level Overview



High-Level Overview



High-Level Overview



Compute Kernels in cgsim

TECHNISCHE UNIVERSITÄT DARMSTADT

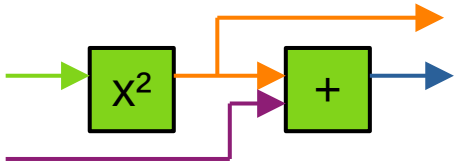
```
1  COMPUTE_KERNEL(aie, adder_kernel,
2                 KernelReadPort<float> in1, KernelReadPort<float> in2,
3                 KernelWritePort<float> out)
4  {
5      while (true) {
6          const float val = (co_await in1.get()) + (co_await in2.get());
7          co_await out.put(val);
8      }
9  }
10
11 COMPUTE_KERNEL(aie, squaring_kernel,
12                KernelReadPort<float> in,
13                KernelWritePort<float> out)
14 {
15     while (true) {
16         const float val = co_await in.get();
17         co_await out.put(val * val);
18     }
19 }
```


Embedded Systems & Applications

Compute Graphs in cgsim

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  COMPUTE_GRAPH constexpr auto the_graph = make_compute_graph_v<[] (
2      IoConnector<float> in1, IoConnector<float> in2
3  ) {
4      IoConnector<float> squared, out;
5
6      squaring_kernel(in1, squared);
7      adder_kernel(squared, in2, out);
8
9      return std::make_tuple(squared, out);
10 }>;
```



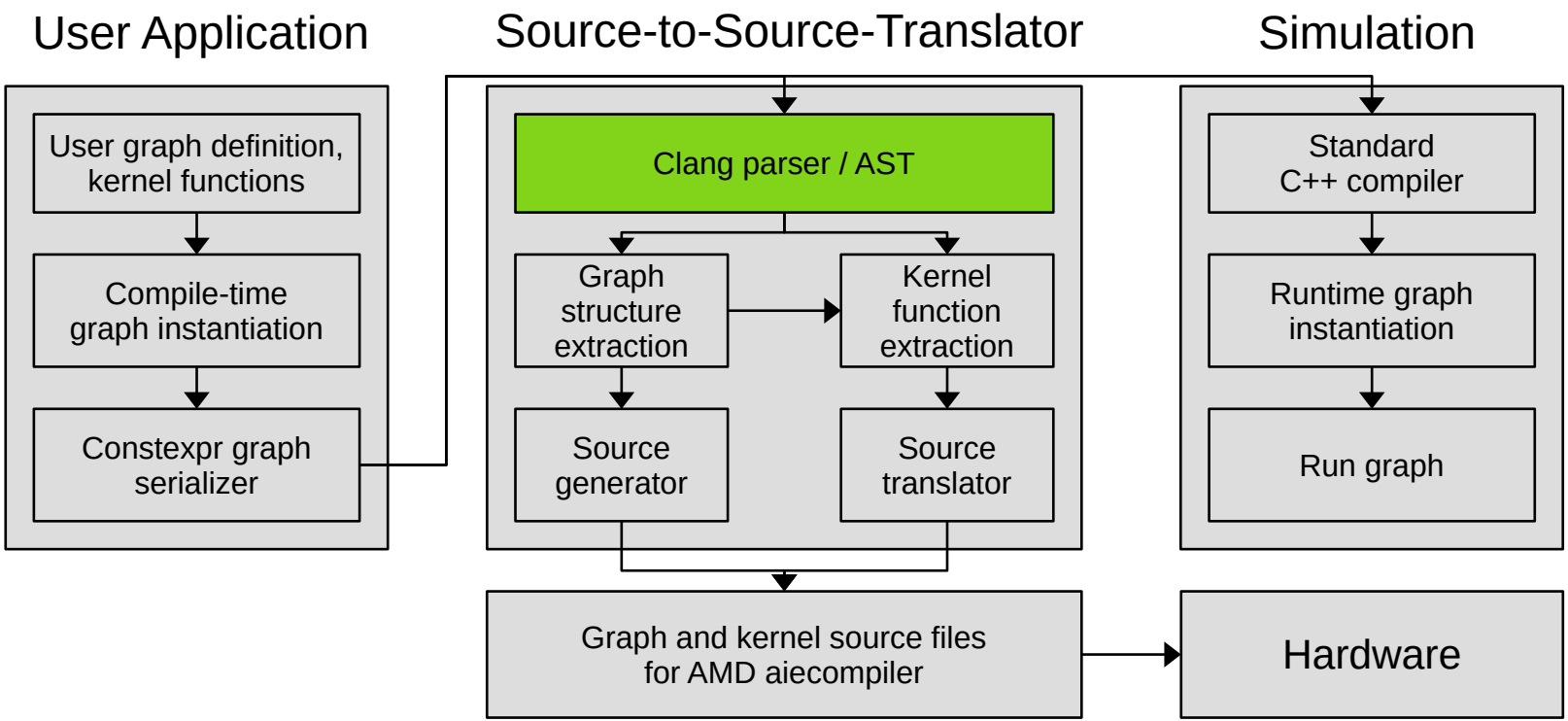
Graph Serialization

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parsing arbitrary user-provided code is hard
- **Compile-time code execution** can make parsing easier
  - Instantiate the compute graph at compile time
  - Gather information with traits / template metaprogramming
  - **Serialize** the graph into a **constexpr variable**
- Result: Problem reduced to parsing a **flat data structure**

Embedded Systems & Applications

High-Level Overview

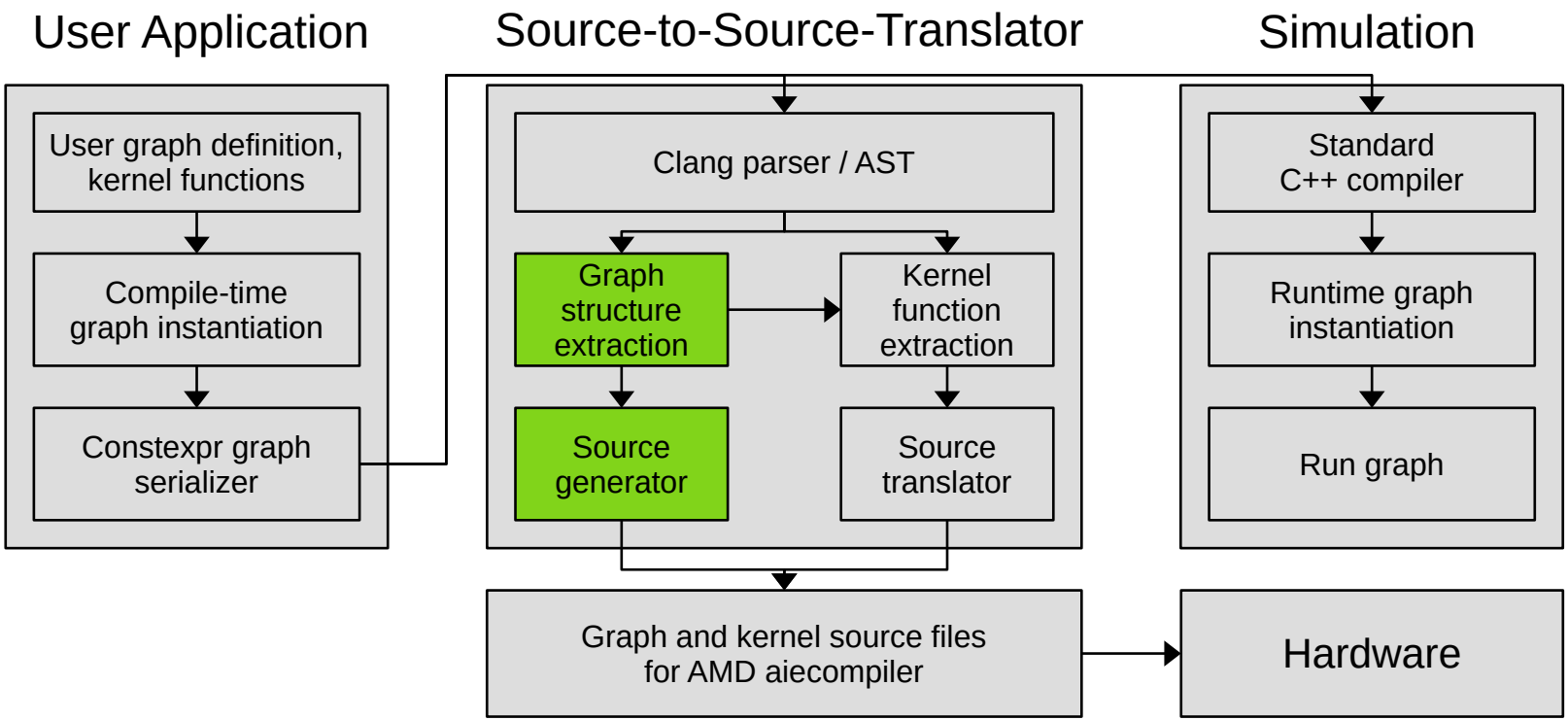


Graph Ingestion

TECHNISCHE UNIVERSITÄT DARMSTADT

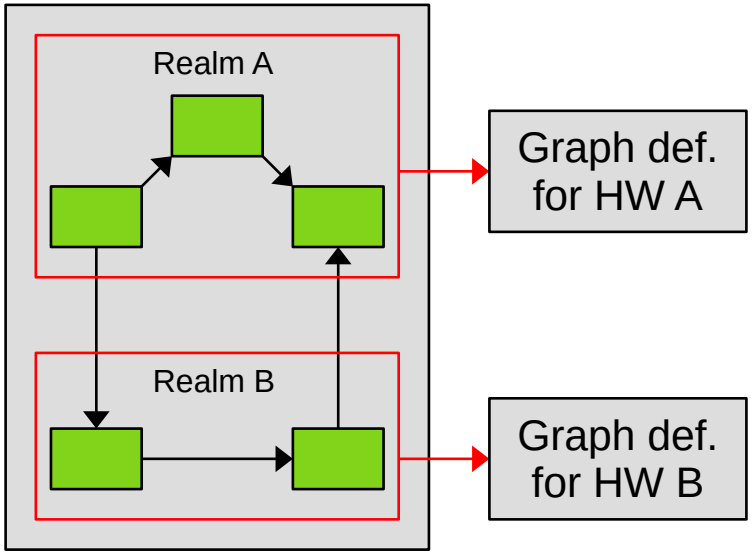
1. Read source files, build Clang AST
2. Scan for compute graphs (marked by **attribute**)
3. Evaluate each graph expression
   - Clang runs the user-provided constexpr code
   - Returns **serialized form of the graph**
4. Deserialize (unflatten) the graph

High-Level Overview

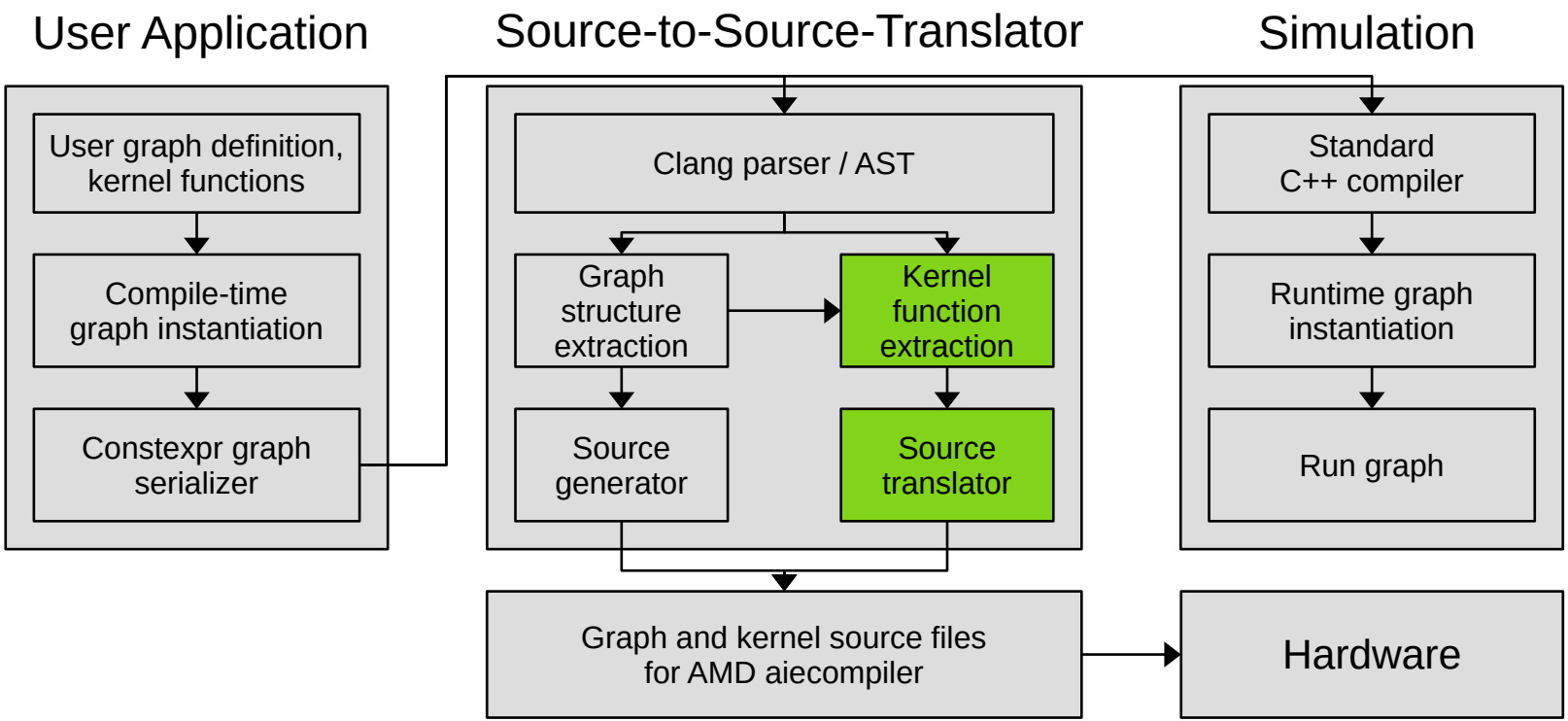


Graph Source Generation

- **Partition** the graph into multiple **kernel realms**
  - **AIE**, HLS, CPU, …
- **Emit graph definition** source file(s) for each realm
  - Realm-specific **source generators**
- AIE generator:
  - Kernel declarations & instantiations
  - Graph connectivity



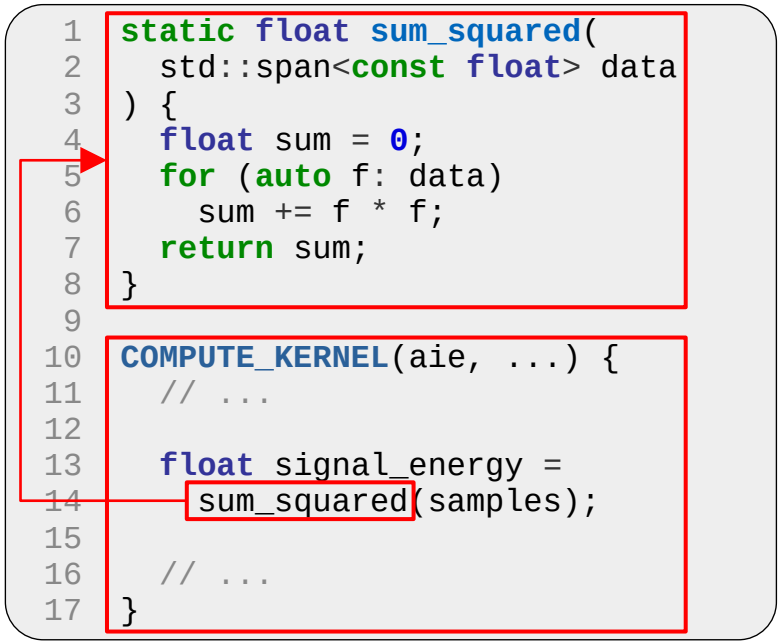
High-Level Overview



Kernel Function Extraction

TECHNISCHE UNIVERSITÄT DARMSTADT

- Determine **source range** of each used kernel
- **Transform** each kernel in a **realm-specific** way
- Copy **referenced source snippets** into the kernel source file
  - Functions, variables, types, include directives, ...



AIE Source Transformation

- Remove all **co_await** tokens (used in cgsim port I/O)
  - Asynchronous calls → synchronous calls
- Generate **kernel entry thunk** function
  - Wrapper classes translating generic **get()/put()** to AIE-specific calls

```
1  COMPUTE_KERNEL(aie, adder_kernel,
2      KernelReadPort<float> in1, KernelReadPort<float> in2,
```

```
1  void adder_kernel_thunk(
2      adf::input_async_buffer<...>  xlat_kparam_0,
3      adf::input_async_buffer<...>  xlat_kparam_1,
4      adf::output_async_buffer<...> xlat_kparam_2
5  ) {
6      adder_kernel_impl(
7          KernelReadPort<...> (xlat_kparam_0),
8          KernelReadPort<...> (xlat_kparam_1),
9          KernelWritePort<...>(xlat_kparam_2)
10     );
11 }
```

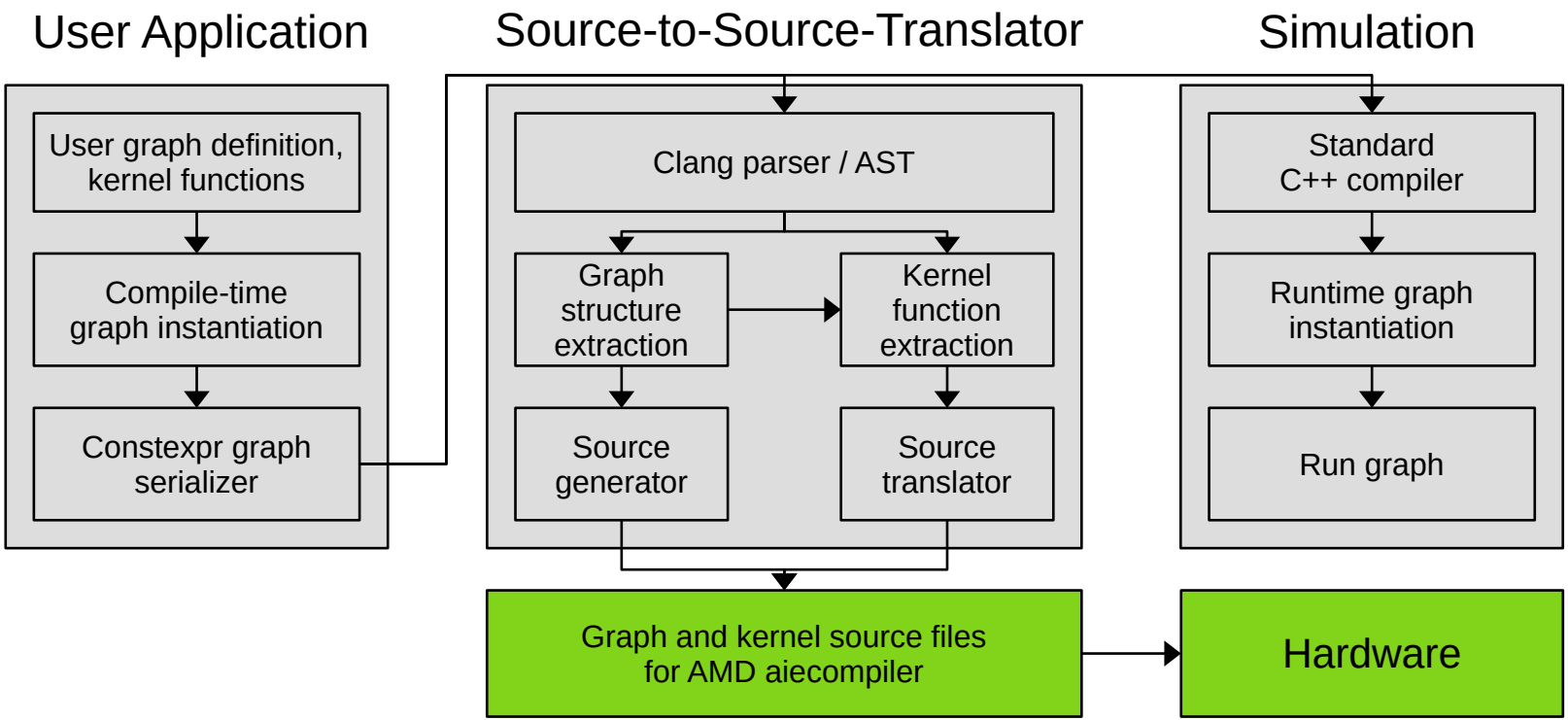
AIE API & Intrinsics in cgsim

TECHNISCHE UNIVERSITÄT DARMSTADT

- AIE intrinsics and API calls required to fully leverage AIEs
- AMD provides an x86 implementation of AIE intrinsics
  - Required for **x86sim** simulator
  - C++ headers and static libraries
- cgsim can use these headers too
  - Enables writing **AIE SIMD code in cgsim**
  - Syntax and semantics identical to Vitis

Embedded Systems & Applications

High-Level Overview



Evaluation Strategy: AIE → cgsim → AIE

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Port **example AIE applications** (from AMD) to cgsim
   - Small performance-focused demos
   - **Hand-optimized** AIE code
2. Then **extract the graphs** again
   - Turns cgsim code back into AIE code
3. Profile the graphs in the AIE simulator
   - **Throughput and latency**
   - Simulator performance

Embedded Systems & Applications

Evaluation Strategy: Selection of Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

- Examples selected to stress different aspects of cgsim
- Bitonic sort
  - Relies heavily on **AIE intrinsics**
- Farrow filter
  - Two-kernel graph, optimized for throughput
  - Gets close to theoretical performance limits of AIE tiles
  - Stresses **kernel-to-kernel communication**
- Others: IIR filter (throughput), Bilinear Interpolation (intrinsics)

Embedded Systems & Applications

Results: Graph Correctness (cgsim → AIE)

- Auto-extracted graphs compile with AMD toolchain
- Ported graphs produce **correct outputs**
  - **Bit-for-bit identical** in cgsim and AMD's simulators
  - Validated with test vector files

Results: Graph Performance (cgsim → AIE)[1]

TECHNISCHE UNIVERSITÄT DARMSTADT

Graph	Input block size	AMD (ns / block)	This work (ns / block)	Relative performance
Bitonic	16 floats	3557	4169	85.32 %
Farrow	1024 samples	913	1019	89.58 %
IIR	2048 samples	5410	5385	100.46 %
Bilinear	256 pixels	484	567	85.33 %

[1]Determined via cycle-approximate AIE architecture simulator (aiesim)

Embedded Systems & Applications

Results: Graph Performance (cgsim → AIE)

TECHNISCHE UNIVERSITÄT DARMSTADT

- **cgsim** generated kernels achieve 85 – 100 % of original performance
- Performance differences due to code changes
  - Wrapper for AIE to **cgsim** type conversion
  - Impact on VLIW instruction scheduling
    - Insertion of NOPs during kernel initialization
  - Amortizing for larger block sizes
  - In IIR case cgsim achieves slight **performance improvement**
- Hand-optimization **through cgsim** possible

Embedded Systems & Applications

Results: Simulator Performance

TECHNISCHE UNIVERSITÄT DARMSTADT

Graph	Repetitions	cgsim (seconds)	AMD x86sim (seconds)	AMD aiesim (seconds)
Bitonic	1024	14.3	22.9	5826
Farrow	512	22.3	20.7	4287
IIR	256	18.2	21.4	4346
Bilinear	1	15.0	15.6	3536

Embedded Systems & Applications

Results: Simulator Performance

- cgsim has **very little overhead**
  - Simulation runtime dominated by AIE intrinsics
  - **< 0.1%** of time spent in cgsim library
  - Fast despite being single-threaded
- AMD x86sim is slightly faster in 1 of 4 cases (Farrow)
  - Multiple compute heavy-kernels
  - Leverages multi-threading
  - Suffers from high synchronization overhead

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

- In-application graph simulation and code extraction framework targeting AMD Versal AIEs **cgsim**
- Hybrid source-to-source translator approach
  - Compile-time preprocessing of C++ AST
  - Lowers complexity of AST introspection
- **Comparable graph performance** to hand-optimized kernels
  - Optimizations can be applied **through cgsim**
- Foundation for **multi-target code generation** already built

Embedded Systems & Applications

Future Work

- Only AIE code-generation backend implemented
  - We are currently adding support for **HLS** kernels
- No support for **templated kernels** yet
  - Requires complex template metaprogramming / type traits
  - Work in progress

Thank you for your attention!

TECHNISCHE UNIVERSITÄT DARMSTADT

- Any questions? Contact us: jonathan.strobl@gmx.de
- We release cgsim as open-source software at:

  https://github.com/esa-tu-darmstadt/cgsim

Embedded Systems & Applications