A Compute Graph Simulation and Implementation Framework Targeting AMD Versal AI Engines

Jonathan Strobl
Jonathan.Strobl@gmx.de
Technical University of Darmstadt
Darmstadt, Germany

Leonardo Solis-Vasquez
solis@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Yannick Lavan
lavan@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Andreas Koch
koch@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

We present a framework for developing compute graph-based applications targeting the AI Engine (AIE) array of AMD Versal SoCs. This framework enables users to embed AIE-based dataflow graph prototypes directly within existing C++ applications and automatically transform them into deployable AIE graph projects. It thereby eliminates the need to manually separate host and accelerator codebases, as required by the standard AMD Vitis workflow. The framework comprises two core components: (1) a compute graph simulation library that can be linked into existing C++ programs, and (2) a Clang-based source-to-source translator that extracts simulator-defined graphs and prepares them for compilation with AMD's AIE toolchain. We evaluate our framework using AMD's official example graphs and show that our generated AIE code achieves performance comparable to hand-optimized Vitis implementations. Additionally, we demonstrate how C++ compile-time code execution can be leveraged to simplify the implementation of source-to-source translation and static source analysis.

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques*; **Source code generation**; • **Hardware** → **Reconfigurable logic and FPGAs**; • **Computing methodologies** → **Vector / streaming algorithms**.

KEYWORDS

AMD AI Engine, AMD Versal, Clang, source-to-source translation

ACM Reference Format:
Jonathan Strobl, Leonardo Solis-Vasquez, Yannick Lavan, and Andreas Koch. YYYY. A Compute Graph Simulation and Implementation Framework Targeting AMD Versal AI Engines. In *Preprint: Accepted for presentation at H2RC 2025 (Held with SC'25). The final version will appear in the official workshop proceedings.*. ACM, New York, NY, USA, 9 pages. https://doi.org/WW.XXXX/XXXXXX.YYYYYYY

SC Workshops '25, November 16–21, 2025, St Louis, MO, USA

ACM ISBN XXXXXXXX/YYYY/ZZ
https://doi.org/WW.XXXX/XXXXXX.YYYYYYY

1 INTRODUCTION

Specialized compute accelerators are playing an increasingly important role in HPC, offering significantly higher performance per watt and compute density compared to general-purpose CPUs. In addition to GPUs, streaming MIMD compute graph accelerators have gained traction in recent years [6][12]. However, programming these architectures remains challenging due to their complexity and heavy reliance on vendor-specific toolchains.

One example of such an accelerator architecture is the AI Engine (AIE) array integrated into AMD Versal SoCs. This array consists of a two-dimensional grid of VLIW processors, each equipped with SIMD vector units [2]. Each processor contains local code SRAM and multiple banks of data SRAM, allowing them to execute independent compute kernels while communicating through data streaming interfaces. AMD provides the *aiecompiler* toolchain for AIEs as part of the Vitis software suite [3]. To use this toolchain, developers must express their application as a *compute graph*, which is a collection of kernels that exchange data streams among themselves and with the array's external interfaces. Since each kernel runs as an independent program on its own processor, the toolchain requires kernels to be defined in *separate* source files and compiled *individually*. Thus, developers must *manually* partition their HPC application into *separate* host and device codebases, each with distinct toolchains, simulators, and debuggers—significantly increasing the complexity of porting to the AIE architecture.

To lower the barrier to entry for AIE programming, we introduce a C++ library, `cgsim`, for building and simulating compute graph prototypes, along with an associated source-to-source translator that *automatically* converts these prototypes into deployable AIE projects. Our primary goal is to provide an alternative development workflow for AIE graphs in which developers can *embed* graph prototypes directly within their existing applications using our simulation library, rather than *separating* host and device components already in the early stages of development. Compared to AMD's standard AIE development flow, our approach ensures a fully functional application throughout the graph development process and eliminates the need for vendor-specific tools or build environments during early-stage simulation and debugging.

Our contributions are summarized as follows:

- We design and implement a compute graph simulation library, **cgsim**, which allows users to embed AIE-based dataflow graph prototypes directly within C++ applications. In contrast to the existing simulation library *Graphtoy* [15], **cgsim** uses a *compile-time* graph construction approach, which enables both efficient simulation and automated extraction of graphs from a larger source file.
- We develop a corresponding **compute graph extractor**, which leverages Clang's semantic analysis capabilities to transform **cgsim** graphs into AIE projects through a combination of source-to-source translation and code generation.
- We evaluate our approach by porting several official AMD AIE examples to **cgsim**, demonstrating that the automatically extracted graphs achieve at least 85% of the performance of hand-optimized Vitis implementations.
- We release our framework as open source at: https://github.com/esa-tu-darmstadt/cgsim.

The remainder of this paper is organized as follows: Section 2 reviews related work on AIE programming frameworks and source-to-source translation tools. Section 3 and Section 4 present the architecture of the **cgsim** simulator and its associated compute graph extractor, respectively. Section 5 evaluates the proposed approach. Finally, Section 6 concludes with a discussion of current limitations and future work.

2 RELATED WORK

Several efforts have focused on improving the programmability of AIEs. For example, PyAIE [18] is a Python-based framework that allows developers to run **numpy**-based machine learning algorithms on Versal SoCs. It utilizes PyLog [9] to translate Python functions to HLS kernels and off-loads matrix multiplication operations to the AIEs using code templates. Another example is Vyasa [7], which extends the Halide DSL compiler [13] to automatically generate vectorized Conv2D loops for the AIE architecture. Vyasa emits C code that invokes AIE vector intrinsics and exploits both temporal and spatial memory locality through vector register reuse and the merging of adjacent accesses into wider vector operations. Moreover, Vyasa can automatically explore the design space of possible execution schedules by leveraging feedback from AMD's cycle-approximate AIE simulator [3].

In contrast to the high-level, code-generating abstractions of PyAIE and Vyasa that are focused on matrix operations for machine learning applications, Graphtoy [15] adopts a low-level, developer-oriented approach. It provides a coroutine-based compute graph simulation library designed for integration into existing C++ codebases, rather than as a standalone application development framework. Graphtoy does not perform automatic code generation; instead, it facilitates rapid prototyping and iterative refinement of AIE kernels and graph structures. However, it lacks an automated path for converting prototypes into deployable AIE projects, requiring users to manually reimplement their application in Vitis once their prototype is validated. In this work, we address this limitation by introducing a source-to-source translation tool, which requires a redesign of the simulation library to support automated source code analysis and transformation.

To enable source-to-source translation of compute graphs embedded in C++ programs, an appropriate analysis and transformation framework must be first selected. We evaluated two candidate frameworks for this purpose: Clang LibTooling and Clava.

Clang [16] is a standards-compliant C++ compiler that uses the LLVM compiler infrastructure [11] as its backend for code generation. The LibTooling project extends Clang's modular architecture to support static analysis and source-to-source translation [17]. It does so by exposing the Clang frontend as a library, allowing external tools to access the AST representations of C++ source files for analysis and transformation. LibTooling further enhances this functionality with APIs for AST pattern matching and source-level rewriting. Because Clang enriches the AST with semantic information before it is passed to user-defined tools, LibTooling enables not only syntactic but also semantic analysis. This includes support for higher-level C++ constructs such as template instantiations, constant evaluation, and compile-time code execution.

Clava [4] [5] is a source-to-source compiler that supports C, C++, CUDA, and OpenCL. It is structured as a pipeline of tools and leverages the Clang frontend to convert source code into a custom AST representation. This simplified AST can be analyzed and transformed using user-defined JavaScript code via the Clava Weaver Engine. The modified AST is then reconstructed into source code and written to disk for downstream compilation. Although Clava offers a simpler API for source-to-source translation, it does not expose the full range of AST analysis and transformation capabilities available in the underlying Clang frontend.

In this work, we choose to interface directly with the Clang frontend using the *LibTooling* framework [17] rather than using Clava. While this decision involves managing the added complexity of Clang's infrastructure, it provides access to its advanced semantic analysis capabilities, which are essential for our translation goals. Our rationale for selecting Clang over Clava is discussed in greater detail in Section 4.

3 THE COMPUTE GRAPH SIMULATOR

The compute graph simulation library developed in this work, **cgsim**, builds upon concepts introduced in the earlier Graphtoy simulator [15], but has been completely rewritten to support programmatic post-processing of compute graphs. Like Graphtoy, it employs C++20 coroutines to simulate concurrently executing kernels through cooperative multitasking, and uses multi-producer, multi-consumer (MPMC) queues for data streaming between kernels. However, **cgsim** differs fundamentally in its implementation from Graphtoy due to the need for its compute graph constructs to be readily parseable by a source-to-source translation tool. The following sections describe the architecture of **cgsim** in detail.

3.1 Architecture rationale

In the prior Graphtoy toolflow, compute graphs are constructed *dynamically* at runtime when invoked by the host application. This dynamic construction poses a major challenge for source-to-source translation, as graph construction may be interleaved with and be dependent on arbitrary user code. Thus, the structure of the compute graph can vary based on runtime input data. To extract all possible compute graphs from such a program, one would need

to determine whether the program generates a finite set of graph variants for all possible inputs—in other words, whether the graph generation process *halts*. This problem reduces to the *halting problem* [14], which is undecidable, making it infeasible to enumerate and extract all compute graphs in the general case for that approach.

To overcome this fundamental limitation, we considered three alternative approaches:

(1) *Runtime serialization of graph variants*: Retain the original runtime construction approach, but *serialize* all instantiated compute graphs to disk during program execution. This strategy requires users to provide a wide range of test inputs that trigger *all* possible graph variants that the program might generate.
(2) *Restricted runtime construction with interpretation*: Maintain the runtime graph construction, but restrict graph construction code to a well-defined subset of the C++ language and disallow dependencies on runtime data. The compute graph extractor can then parse and interpret this limited subset. However, this approach would require an additional C++ *interpreter* alongside the compiler.
(3) *Compile-time graph construction*: Shift graph construction *entirely* to compile time, requiring a complete redesign of the simulator. While this departs from the original Graphtoy's runtime-based design, it enables—in the extractor—the use of C++'s standardized **`constexpr`** *compile-time execution mechanism* [10], which is supported by all conforming compilers. This approach also aligns with the programming paradigm of AIEs and similar accelerator architectures, which do not even support dynamic graph construction at runtime.

Thus, we adopt the third approach for our **`cgsim`** because it better matches the AIE programming model and offers the potential to significantly reduce the complexity of the compute graph extraction by leveraging existing, well-maintained compiler infrastructure.

The following sections will give an overview over the use of the advanced C++ features that enable us to transparently (for the user) leverage the existing Clang infrastructure by compile-time code execution. This novel approach spares us from having to implement a custom compiler front-end, which would be the usual solution in such a tool flow.

3.2 Architecture overview

The compute graph simulator provides the compile-time **`constexpr`** functions and macros that enable users to define compute graphs using standard C++ syntax. These definitions are serialized into a flattened, array-based data structure and stored in a **`constexpr`** variable. This structure embeds references to the graph's kernels and I/O port types, enabling the graph runtime to reconstruct a complete copy of the compute graph at program execution. The same data structure can be parsed by the compute graph extractor (described in Section 4) and transformed into a representation suitable for cross-compilation targeting real hardware. Figure 1 illustrates the architecture of compile-time graph construction, while Figure 2 shows how the **`cgsim`** library, which implements this compile-time functionality, integrates into the overall graph prototyping workflow.

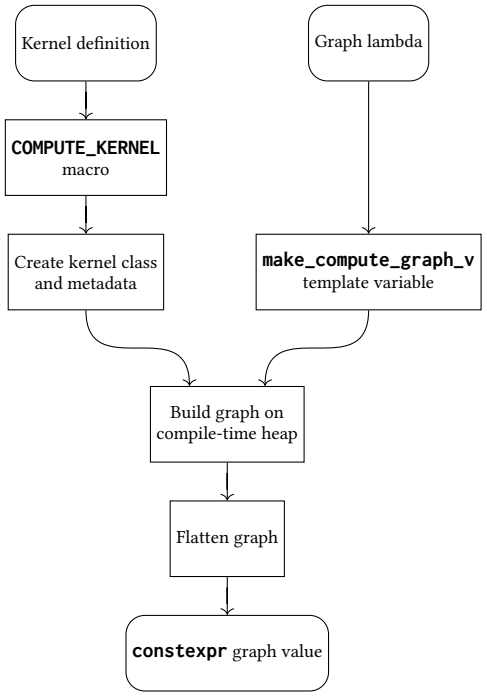


Figure 1: A compute graph in `cgsim` consists of individual kernels, defined by the user using the `COMPUTE_KERNEL` macro, and a graph definition specified as a lambda function that describes graph's connectivity when invoked. The compile-time logic of `cgsim`, invoked via the `make_compute_graph_v` template variable, combines these kernel and connectivity definitions through postprocessing and flattening steps to produce a complete compute graph. This graph can then be stored in a `constexpr` variable for later execution or extraction.

3.3 Kernel definition

To enable compile-time inspection of external interfaces—such as stream connections and parameters—via template metaprogramming, kernels in a **`cgsim`**-based compute graph must be defined as static functions using the **`COMPUTE_KERNEL`** macro. Figure 3 illustrates this approach. Internally, the macro transforms the kernel into a member function of a newly generated class. This class also stores metadata, including the kernel's execution realm (see Section 4.3) and I/O port type information, which is collected using C++ type traits.

3.4 Graph construction

Once all compute graph kernels have been defined, they must be instantiated and connected to form a complete graph. **`cgsim`** performs this construction at compile time using **`constexpr new`**, a C++20 feature that enables dynamic memory allocation within **`constexpr`** contexts, as long as all compile-time allocations are properly deallocated before the context concludes [10][1].

[1]While the C++ standard does not explicitly state that dynamic memory allocation is allowed in **`constexpr`** context, it requires that compilers elide calls to the replaceable global allocation functions in such contexts, while preserving the semantics of the allocation *operator*. As a consequence, conforming implementations must provide an

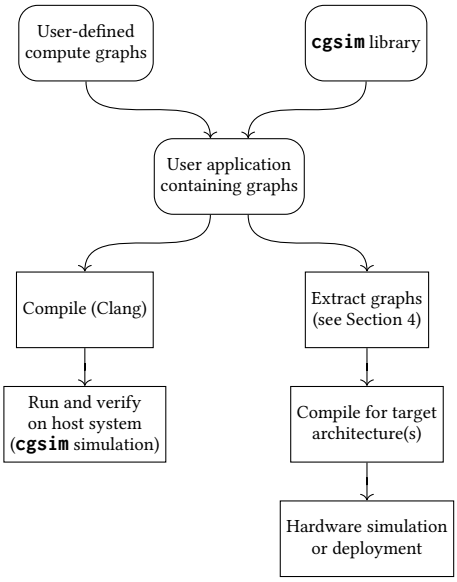


Figure 2: High-level overview of the `cgsim` workflow for prototyping compute graphs. Developers define kernels and graph structure insidet their application using the `cgsim` library, which they can then either simulate on their workstation (left), or extract deployable graphs from via source-to-source translation (right).

```
COMPUTE_KERNEL(
    aie,            // Realm (target HW)
    adder_kernel,   // Kernel name

    // I/O ports
    KernelReadPort<float> in1,
    KernelReadPort<float> in2,
    KernelWritePort<float> out
) {
    while (true) {
        const float val = (co_await in1.get())
                        + (co_await in2.get());

        co_await out.put(val);
    }
};
```

Figure 3: Example of a compute kernel intended for execution on the AIE array. The kernel reads pairs of values from two input streams, computes their sum, and writes the result to an output stream.

Users initiate graph construction by passing a lambda function as a non-type template argument to the **`make_compute_graph_v`** template variable. This lambda is executed at compile time and defines graph connectivity using **`IoConnector`** objects. These objects are passed as parameters to kernel functions; when multiple inputs or outputs reference the same **`IoConnector`**, implicit stream broadcast and merge operations are created, respectively.

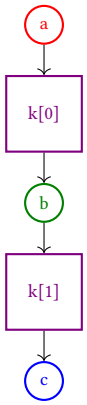
alternative, allocation-like mechanism for compile time use. This mechanism is not required to perform actual memory allocation; it must simply behave *as-if* it did.

```
constexpr auto the_graph =
make_compute_graph_v<[](
    // External graph inputs
    IoConnector<int> a
) {
    // Internal connections
    IoConnector<int> b, c;

    // Kernels
    k(a, b);
    k(b, c);

    // External graph outputs
    return std::make_tuple(c);
}>;
```

(a) Compute graph definition.



(b) Resulting graph.

Figure 4: Side-by-side comparison of a compute graph definition and the resulting in-memory graph after `constexpr` evaluation. Kernels and `IoConnector` instances are color-coded to highlight how graph elements correspond to their defining code sections.

The lambda's **`IoConnector`** parameters become graph's global inputs, while **`IoConnector`** objects returned from the lambda become global outputs. Figure 4 illustrates this construction pattern.

In addition, users can attach a list of *attributes* to each I/O connection in the graph. These attributes are key-value pairs with string keys and either string or integer values. While these attributes do not affect **`cgsim`**'s runtime behavior, they provide auxiliary information, such as PLIO port names and buffering modes, to the graph extractor running later in the flow. This auxiliary information cannot be inferred automatically. In contrast, settings that do influence graph behavior are specified as non-type template arguments to the **`KernelReadPort`** and **`KernelWritePort`** parameters within kernel functions. These settings include options such as marking a port as a runtime parameter or specifying the beat size of the underlying bus (e.g., AXI) for streaming interfaces. When two parameterized ports are connected via an **`IoConnector`**, **`cgsim`** checks for compatibility and merges their configurations into a unified setting shared by all connected endpoints. If the settings are incompatible, a compile-time error is generated. Again, note that this functionality would typically need to be realized in a separate compiler (pass), but, using compile-time code execution, can be moved forward to the *unmodified* Clang front-end.

3.5 Graph serialization

When the compute graph definition lambda is executed, it generates a graph structure composed of dynamically allocated objects in the compile-time heap using **`constexpr new`**. Graph edges are represented as raw pointers between these objects. However, the C++ standard requires that all compile-time allocations be deallocated before the end of **`constexpr`** evaluation (see Section 3.4). As a result, this pointer-based representation cannot persist beyond the **`constexpr`** context, making it unavailable for use by the graph runtime or the extractor.

To overcome this limitation, **cgsim** applies an additional compile-time processing step that transforms the graph into a flattened, array-based data structure, eliminating the need for dynamic memory. This representation *can* be stored in a **constexpr** variable, enabling information to be transferred from compile time to runtime. Type information for kernels and ports is preserved through template functions, which reconstruct objects of the appropriate type when invoked.

3.6 Runtime graph instantiation

To execute compute graphs defined and serialized using the mechanisms described above, **cgsim** provides a *runtime deserializer* that can be invoked via the function call operator of the serialized graph object. This deserializer takes as input the serialized graph data structure, which was produced at compile time, and reconstructs a copy of the compute graph from it on the runtime heap. The instantiation process begins by recreating all graph I/O ports from the serialized descriptors detailed in Section 3.5. Subsequently, **cgsim** instantiates all kernels in the same way and establishes connections between them via the previously constructed port objects. A **RuntimeContext** object encapsulates and manages all reconstructed components, representing a single execution instance of the compute graph. Finally, global graph inputs and outputs are connected to external data sources and sinks, as described in Section 3.7.

At runtime, kernels exchange data through fixed-capacity MPMC queues with broadcast semantics, ensuring that each consumer receives a complete copy of all data written to the buffer. While the queue preserves the order of data from *individual* producers, data from *multiple* producers may be interleaved.

3.7 Global I/O

In addition to managing data transfers between kernels within a compute graph, **cgsim** supports streaming data into and out of the graph's global I/O ports. This functionality is implemented via runtime data sources and sinks—specialized kernel coroutines that the **RuntimeContext** attaches to the graph after instantiating it from its **constexpr** representation (see Section 3.6). During graph execution, each coroutine either produces or consumes a single data stream by accessing standard C++ container objects supplied by the user. The framework also supports passing scalar values and variables through Runtime Parameter sources and sinks, respectively.

Users specify data sources and sinks as positional arguments when invoking the graph, with sources (graph inputs) listed first, followed by sinks (graph outputs).

3.8 Running a graph

After the compute graph is constructed and connected to its data sources and sinks, **cgsim** executes it by starting the coroutine task scheduler embedded within the **RuntimeContext**. Execution proceeds in two steps: First, the runtime context creates all compute kernel coroutines in a *suspended* state and registers them with the scheduler as pending tasks. This initialization step establishes the set of cooperatively multitasked user-mode threads required for graph simulation. Second, the context activates the scheduling algorithm, which invokes pending tasks until no coroutines can continue execution.[2]

Once graph execution completes, the **RuntimeContext** terminates all kernel coroutines and deallocates their associated heap-allocated context objects. The results of the computation remain available in the data sinks connected to the graph.

3.9 Using the AMD AIE intrinsics and API

Although **cgsim** does not directly emulate the AMD AIE intrinsics or API, AMD provides C++ headers and libraries that implement these functions accurately for x86 host CPUs [3]. The **cgsim** framework can import these definitions through an adapter header, allowing compute graph prototypes to leverage AIE SIMD intrinsics outside the Vitis environment. Due to licensing constraints, the **cgsim** repository does not include these libraries; users must manually supply a copy of the **aietools** directory from their Vitis installation to enable AIE intrinsic support in the simulator.

4 THE GRAPH EXTRACTOR

After the compute graph-based application has been shown to be functionally correct using the **cgsim** flow, it needs to be transformed for mapping to the actual AIE hardware. The graph source code extractor is a source-to-source translation tool that processes C++ source files containing **cgsim** graphs and converts them into one or more AMD Vitis-compatible AIE projects through a combination of automated source rewriting and code generation. Although the current implementation targets AIEs exclusively, the tool's architecture is designed to support additional backends, such as high-level synthesis (HLS). A special **noextract** target is also provided, allowing users to exclude specific kernels from the extraction process. The following sections describe the rationale, design decisions, and implementation details of the graph extractor. Figure 5 presents an overview of the extraction workflow.

4.1 Selecting a source-to-source translation framework

To implement a source-to-source translator capable of extracting compute graphs from C++ programs, an appropriate underlying framework must be first selected. Based on the design decisions made for the graph simulator (see Section 3.2), the framework must provide the following capabilities:

- Support for C++20 or higher, including modern language features required by the simulator.
- Semantic analysis, including access to type information and template instantiations.
- Evaluation of **constexpr** expressions, through a built-in C++ interpreter capable of compile-time execution.

We select Clang LibTooling [11][17] as the foundation for the compute graph extractor, primarily due to its comprehensive semantic analysis capabilities, as explained in Section 2.

4.2 Graph ingestion

The compute graph extractor begins by invoking Clang's frontend to parse the input source file into an AST. It then scans the AST

[2]There is no explicit termination condition.

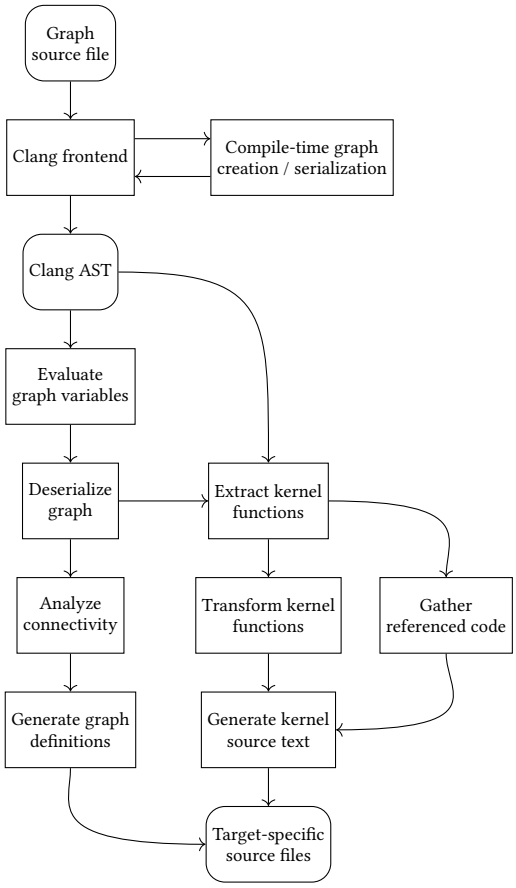


Figure 5: Overview of the graph extraction flow. Starting with a user-provided source file containing one or more compute graphs, the extractor first builds an AST using the Clang frontend. At this stage, `cgsim`'s compile-time graph preprocessing logic executes, producing serialized compute graph variables that the extractor can evaluate later to recover all graph definitions. Finally, the extractor applies a sequence of source transformations to the graphs and their associated kernel functions, writing the results to disk as individual source files.

for global variables with static storage duration that are annotated with the custom `extract_compute_graph` attribute, which is added to Clang to mark extractable graphs. Since `cgsim` represents compute graphs as `constexpr` global variables (see Section 3.5), the extractor can leverage Clang's built-in constant expression evaluator to retrieve the graph definition in serialized form. This approach significantly simplifies the extraction process: instead of traversing the entire AST, the extractor operates on a flattened data structure to reconstruct the graph's connectivity. The complexity of the actual interpretation is offloaded to Clang's well-tested `constexpr` interpreter, simplifying the implementation of our tool flow considerably. After obtaining the serialized data, the extractor deserializes it, converting index-based vertex references back into a pointer-based graph representation.

The serialized compute graph includes function pointers to template functions instantiated with the specific kernel and I/O port types used in the graph. While the `cgsim` library uses these pointers to reconstruct the graph at runtime (see Section 3.6), the extractor follows them to recover the graph's type information from their template arguments.

4.3 Graph partitioning

After deserialization, the extractor partitions the compute graph into subgraphs based on their target hardware for execution, referred to as *realms*. Each kernel is annotated by the user with its intended realm; currently supported values include AIE and `noextract`. Based on these annotations, the extractor classifies all connections between kernels into three categories:

- *Intra-realm* ports: Connections entirely contained within a single realm.
- *Inter-realm* ports: Connections that transfer data between different realms.
- *Global* ports: Connections that move data into or out of the graph.

The extractor assigns this classification to each port in the graph, enabling realm-specific backends to generate the appropriate internal connections and external interfaces.

4.4 Extraction of kernels

The deserialized and partitioned compute graph retains references to AST nodes corresponding to each kernel's function declaration. To generate standalone source files for each kernel, the extractor must isolate their source text from the original file and adapt it to conform to the programming model of the target realm. This is accomplished using a `clang::Rewriter`, which operates on the *expansion range* of each function declaration[3]. The extractor processes each unique kernel function twice: once to generate a forward declaration (call signature only) and once to generate the full function definition (including the body).

Each realm can define custom transformation routines for its associated kernels. Additionally, the extractor provides a set of standard transformation functions that can be reused by realm-specific backends, eliminating the need to duplicate common operations such as removing `co_await` tokens or extracting declarations from definitions.

When the extractor removes `co_await` tokens from a kernel, the coroutine-based asynchronous stream operations are transformed into synchronous, blocking calls. This conversion eliminates the dependency on `cgsim`'s cooperative multithreading framework. However, the streaming I/O port types (`KernelReadPort` and `KernelWritePort`) remain unchanged in the kernel source. Consequently, each realm must provide its own implementations of these types that adapt the `cgsim` API to the native streaming I/O interface of the target realm.

[3]The `clang::Rewriter` must operate on the full macro expansion range, rather than the function declaration's source range, because `cgsim` kernel functions are defined using preprocessor macros.

4.5 AIE kernel transformation

In addition to the standard (realm-independent) transformations described above, the AIE realm augments each kernel declaration with an adapter thunk that converts AIE-specific kernel parameters—such as buffer ports and runtime parameters—into the generic `KernelReadPort` and `KernelWritePort` types expected by the kernel implementation. This thunk serves as the entry point for the kernel on AIE hardware.

4.6 Co-extraction of referenced code

In addition to extracting the source text of individual kernels, the extractor can identify and co-extract other declarations from the input source file that are required for successful kernel compilation and execution. This capability enables users to define custom data types, constant lookup tables, and helper functions at global scope within the `cgsim` graph prototype, which the extractor automatically includes in the appropriate kernel source files. The extraction process captures not only direct dependencies of each kernel function but also transitive dependencies and include directives. To prevent simulation-specific helpers from being included in hardware builds, the extractor allows each realm to blacklist specific headers.

4.7 Graph code generation

In addition to kernel functions, the extractor must generate a graph definition for each realm to instruct the target hardware toolchain on how to instantiate and connect the individual kernels. This step requires a dedicated source generator, rather than source-to-source translation, because the original graph definition is no longer available in source code form when the extractor processes it (see Section 4.2). To accommodate structural differences across hardware targets, each realm-specific code generator may emit multiple source files.

For the AIE realm, the code generator emits two header files per compute graph: `kernel_decls.hpp` and `graph.hpp`. This structure follows AMD's AIE graph programming guide recommendations [2]. The `kernel_decls.hpp` header contains declarations for all kernel functions within the AIE realm, generated during the kernel transformation step (see Section 4.5). The `graph.hpp` header defines the compute graph itself, specifying kernel instantiations, external I/O ports, connectivity, and user-defined attributes.

5 EVALUATION

To demonstrate that the graph simulator and extractor produce correct results without incurring excessive overhead, we port several demo applications from AMD's *Vitis-Tutorials* repository [1] to `cgsim`. We select applications that can be implemented entirely on the AIE array, as the current version of the compute graph extractor does not yet support HLS targets (see Section 4). The selected applications are listed below:

- `Bilinear_Interpolation`: Performs bilinear interpolation on image data using AIE vector intrinsics.
- `bitonic-sorting`: A single-kernel graph that implements a 16-wide bitonic sort on 32-bit floating-point values using the AIE vector intrinsics and API. Its extensive use of the AIE API makes it a suitable test case for evaluating `cgsim`'s API compatibility.

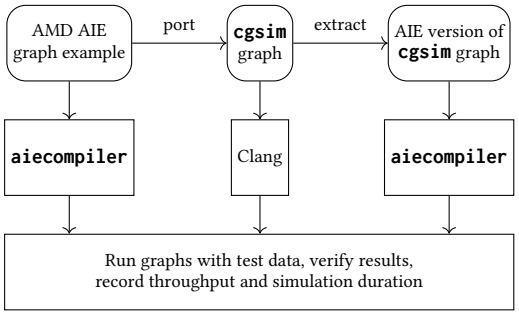


Figure 6: High-level overview of the graph porting and performance evaluation workflow.

- `farrow_filter`: Implements a fractional delay Farrow filter [8] with throughput exceeding 1 Gsps. It uses two kernels with ping-pong buffer I/O, VLIW loop pipelining, and hand-optimized fixed-point SIMD convolution. Its heavily optimized nature makes it ideal for revealing overhead introduced by the automatic translation enabled using our flow.
- `implementing-iir-filter` (part 2b): Implements an IIR filter using SIMD intrinsics, with a focus on maximizing system throughput.

Figure 6 illustrates our graph porting and performance evaluation workflow, which extends the `cgsim` graph prototyping workflow previously shown in Figure 2.

5.1 Graph porting

The example compute graphs provided by AMD rely exclusively on standard C++, AIE intrinsics, and the AIE vector API within their kernel code. Since `cgsim` provides full support for these APIs, we were able to reuse the core algorithm implementations without modification. As a result, we were also able to retain the AIE-specific optimizations that AMD had applied to the example kernels in their respective `cgsim` versions, which include manual vectorization and optimized memory access ordering. Only the streaming I/O interfaces of the kernels required adaptation. Specifically, we updated the parameter declarations, which in the original examples use AIE-specific stream and runtime parameter types. This manual replacement was straightforward, as the corresponding `cgsim` I/O types preserve the same semantics as their AIE framework counterparts. Our interface even improved type safety: `cgsim` supports custom stream data types, including user-defined structs, whereas the AIE framework requires flat, unstructured data buffers. Moreover, we successfully ported the *graph definitions*, which instantiate and connect the kernels (see Section 3.4), without encountering significant issues.

5.2 Performance

To ensure comparability, we measured the performance of the ported graphs using the methodology also employed by AMD for the original AIE examples:

- For the `bilinear`, `IIR`, and `farrow` examples, the primary performance metric is the *time between iterations* as reported

by the execution trace of AMD's AIE cycle-approximate simulator—**aiesim**.
- For the **bitonic** graph, performance data was obtained using the Vitis AIE profiler.

We assume that all **cgsim**-based graphs operate at an AIE clock frequency of 1250 MHz and a PL frequency of 625 MHz in the cycle-approximate simulation, matching the configuration used in the original AMD examples.

Table 1 presents the performance evaluation results. All ported examples achieved at least 85 % of the throughput of the original hand-tuned AMD implementations. The observed performance reduction is primarily attributed to differences in code generation around I/O stream access in the **aiecompiler** (see Section 4.4). Notably, the **cgsim** port of the **IIR** example achieves performance parity with its hand-optimized counterpart.

Table 1: Processing time per input block for original AMD examples vs. cgsim implementations, measured using aiesim on simulated AIE hardware.

Graph	Block size (bytes)	AMD (ns)	This work (ns)	Rel. throughput (percent)
bitonic	64	3556.8	4168.8	85.32
farrow	4096	912.8	1019.0	89.58
IIR	8192	5410.0	5385.0	100.46
bilinear	2048	484.0	567.2	85.33

We also evaluated simulator performance by measuring the wall-clock time required to execute each example graph. Since most graphs complete in under 100 ms, we repeated the input test vectors to extend the execution time of AMD's AIE functional simulator—**x86sim**—to approximately 20 s. This approach minimizes distortions caused by process startup overhead. As shown in Table 2, **cgsim** and **x86sim** exhibit comparable performance on compute-intensive examples such as **bilinear** and **IIR**, where large data blocks are transferred in bulk. This reduces the impact of synchronization overhead in kernel-to-kernel data transfers. In contrast, the **bitonic** graph processes smaller data blocks, leading to more frequent synchronization. In this case, **cgsim** outperforms **x86sim** due to its low-overhead cooperative multitasking architecture. We verified this by profiling with **perf**, which revealed that **cgsim** spends 99.94 % of its total runtime executing the **bitonic** kernel and only 0.06 % on synchronization and data transfer. Profiling the remaining examples confirmed that synchronization overhead in **cgsim** remains negligible across all cases.

The modest performance advantage that **x86sim** demonstrates over **cgsim** when simulating the **farrow** graph stems from their different execution models: **x86sim** assigns each kernel to a dedicated OS thread, whereas **cgsim** employs cooperative multitasking to execute all kernels on a single shared thread. Since the **farrow** graph consists of two kernels, **x86sim** utilizes two CPU cores fully for 20.70 s, while **cgsim** uses a single CPU core for 22.26 s. This result highlights **cgsim**'s efficiency despite its single-threaded design. However, this design choice may lead to slower performance in scenarios where graphs contain many compute-intensive kernels with limited inter-kernel communication. Furthermore, for completeness, we also report the runtime of the cycle-approximate simulator, **aiesim**.

Table 2: Wall-clock simulation time comparison between cgsim and the AMD's AIE simulators.

Graph	Repetitions	Simulation runtime (seconds)		
		cgsim	**x86sim**	**aiesim**
bitonic	1024	14.32	22.90	5825.96
farrow	512	22.26	20.70	4287.03
IIR	256	18.20	21.37	4346.19
bilinear	1	14.95	15.57	3534.90

6 CONCLUSION AND FUTURE WORK

The compute graph simulator (**cgsim**) and source-to-source translator (graph extractor) introduced in this work enable developers to prototype AI Engine (AIE) graphs directly within existing codebases. This approach eliminates the need to separate applications into distinct host and accelerator components during the prototyping phase, leading to simpler and faster design iterations. Our performance evaluation shows that AIE projects generated from **cgsim** prototypes achieve competitive throughput—reaching at least 85 % of the performance of hand-optimized AIE implementations across selected AMD benchmark examples. Importantly, it is possible to apply hardware-specific optimizations to AIE kernels in the **cgsim** version of a compute graph, avoiding the need to manually modify code generated by the graph extractor.

The combination of the graph simulator and extractor demonstrates the potential of hybrid source-to-source translation architectures that leverage C++ compile-time code execution. By exploiting template metaprogramming and Clang's built-in capabilities, this approach substantially reduces the complexity of AST analysis and source transformation. This design also minimizes code duplication and lowers the risk of missing edge cases, as the extractor's analysis phase benefits directly from Clang's mature support for template metaprogramming.

Although **cgsim** currently supports many features of the AIE array—including access to AMD's AIE processor emulation library outside the Vitis environment—several hardware capabilities remain unexposed. These include advanced DMA operations such as corner-turning, templated kernel support, and Global Memory I/O functionality. Furthermore, the current implementation of the graph extractor generates code only for the AIE hardware target. However, its realm-based architecture already provides a foundation for future extensions by automatically partitioning graph prototypes into hardware-specific subgraphs. This design will enable the development of code generators for additional targets, including FPGAs via HLS.

REFERENCES

[1] Advanced Micro Devices, Inc. 2024. *Vitis In-Depth Tutorials.* https://github.com/Xilinx/Vitis-Tutorials/tree/2024.2
[2] Advanced Micro Devices, Inc. 2025. *AI Engine Kernel and Graph Programming Guide (UG1079).* https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding

[3] Advanced Micro Devices, Inc. 2025. *AI Engine Tools and Flows User Guide (UG1076).* https://docs.amd.com/r/en-US/ug1076-ai-engine-environment/

[4] Joao Bispo and João M.P. Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565. https://doi.org/10.1016/j.softx.2020.100565

[5] Joao Bispo, João M.P. Cardoso, et al. 2025. *Clava GitHub repository.* University of Porto, Faculty of Engineering (FEUP). https://github.com/specs-feup/clava

[6] Nick Brown. 2023. Exploring the Versal AI Engines for Accelerating Stencil-based Atmospheric Advection Simulation. In *FPGA '23: Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 91–97. https://doi.org/10.1145/3543622.3573047

[7] Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar. 2020. Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine. In *IEEE High Performance Extreme Computing Conference (HPEC)*. Institute of Electrical and Electronics Engineers (IEEE), New York City, USA. https://doi.org/10.1109/HPEC43674.2020.9286183

[8] C. W. Farrow. 1988. A Continuously Variable Digital Delay Element. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. Institute of Electrical and Electronics Engineers (IEEE), New York City, USA, 2641–2645. https://doi.org/10.1109/ISCAS.1988.15483

[9] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. In *IEEE Transactions on Computers*. Institute of Electrical and Electronics Engineers (IEEE), New York City, USA, 2015 – 2028. https://doi.org/10.1109/TC.2021.3123465

[10] ISO/IEC. 2020. *Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland. https://www.iso.org/standard/79358.html

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Institute of Electrical and Electronics Engineers (IEEE), New York City, USA. https://doi.org/10.1109/CGO.2004.1281665

[12] Samuel Maddrell-Mander, Lakshan Ram Madhan Mohan, Alexander Marshall, Daniel O'Hanlon, Konstantinos Petridis, Jonas Rademacker, Victoria Rege, and Alexander Titterton. 2021. Studying the Potential of Graphcore IPUs for Applications in Particle Physics. In *Computing and Software for Big Science*, Vol. 5. Springer, Cham, Switzerland. https://doi.org/10.1007/s41781-021-00057-z

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 519 – 530. https://doi.org/10.1145/2491956.2462176

[14] Michael Sipser. 2012. *Introduction To The Theory Of Computation* (3rd ed.). Cengage Learning, Independence, Kentucky, USA.

[15] Jonathan Strobl, Leonardo Solis-Vasquez, Yannick Lavan, and Andreas Koch. 2024. Graphtoy: Fast Software Simulation of Applications for AMD's AI Engines. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 20th International Symposium, ARC 2024*. Springer, Cham, Switzerland, 166–180. https://doi.org/10.1007/978-3-031-55673-9_12

[16] The Clang project contributors. 2025. *Clang C language family frontend for LLVM.* https://clang.llvm.org/

[17] The Clang project contributors. 2025. *Clang documentation: LibTooling.* https://clang.llvm.org/docs/LibTooling.html

[18] Hongzheng Tian, Shining Yang, Yoonha Cha, and Sitao Huang. 2023. Late Breaking Results: PyAIE: A Python-based Programming Framework for Versal ACAP Platforms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. Institute of Electrical and Electronics Engineers (IEEE), New York City, USA. https://doi.org/10.1109/DAC56929.2023.10247843