Graphtoy: Fast Software Simulation of Applications for AMD's AI Engines

Jonathan Strobl[1] [0000-0003-0442-7485],
Leonardo Solis-Vasquez[1] [0000-0001-6896-9879],
Yannick Lavan[1] [0000-0001-5309-4141], and Andreas Koch[1] [0000-0002-1164-3082]

Embedded Systems and Applications, Technical University of Darmstadt, Darmstadt, Germany
`Jonathan.Strobl@gmx.de, {solis, lavan, koch}@esa.tu-darmstadt.de`

Abstract. This work[1] presents Graphtoy, a coroutine-based compute graph simulator built in C++20, which can be embedded into a target application for rapid step-by-step prototyping of graphs targeting AMD's AI Engines, as used in Versal FPGAs and Ryzen 7040 CPUs. By using a molecular docking application as a case study, we demonstrate: 1) how compute graphs developed using Graphtoy can be ported to the AI Engines with no modifications to the graph structure, and 2) that C++20 coroutines are well suited for simulating many-core systems with complex inter-core communication schemes. Furthermore, our set of molecular docking graphs ported to Graphtoy achieves an order-of-magnitude increase in simulation speed compared to AMD's AI Engine graph simulators. The corresponding code is released as open source under: https://github.com/esa-tu-darmstadt/graphtoy.

Keywords: Versal FPGA · AI-Engines · C++20 coroutines · prototyping · simulation · compute graphs · molecular docking.

1 Introduction

AI Engines (AIEs) are a new kind of compute element AMD has introduced in its Versal series of FPGAs [6] and recently added to mobile CPUs, such as its Ryzen AI 7040 processors. For the discussion here, we will focus on the use of AIEs on the Versal platform. But the techniques are also applicable to the processor-integrated units.

The AIEs consist of a parallel set of tiled Very Long Instruction Word (VLIW) vector processors, providing a truly Multiple Instruction Multiple Data (MIMD) processing model. These processors are connected through a configurable routing fabric that enables stream-based communication between them, which makes the architecture well suited for executing compute graphs or pipelines.

The AIE toolchain provided by AMD imposes a rigid development methodology when porting an application to AIE graphs. In particular, graphs must be

[1] **This is a pre-print version. The final version is available on the publisher website.**

separated from the application's host code, running on the conventional processor(s), while the entire application must be ported to the Versal SoC framework (i.e., Vitis [4]) for testing the functional correctness across the conventional and AIE portions of the code. This means that there is a large up-front porting effort required before it is even possible to execute an AIE graph design in conjunction with the host code for a particular application. Furthermore, AIE compile and simulation times are often lengthy when compared to traditional software, and debugging is again complicated due to the separation of host and AIE graph code.

To tackle this, we propose an alternative approach to AIE graph prototyping: instead of porting an application to the compute graph framework, a graph simulator can be *embedded* into the application. For development and debugging purposes, this allows the use of a traditional software-only compile and debug flow, which will also be much faster and easier to use than the actual AIE tools. Our solution is called *Graphtoy* and provides a fast architecture-independent AIE compute graph simulator that heavily exploits C++20 coroutines in its internal architecture.

Our contributions are summarized as follows:

– We present Graphtoy, our embeddable coroutine-based compute graph simulator built in C++20, and discuss its design, usage, and overall porting methodology.
– We compare the structure of graphs implemented in Graphtoy and the AIE framework.
– We benchmark Graphtoy's performance against that of AMD's AIE simulators. For this purpose, we use a molecular docking application as a case study.

The remainder of this paper is structured as follows: Section 2 compares our work to previous studies. Section 3 describes in detail the architecture of the Graphtoy compute graph simulator, while Section 4 shows how we use Graphtoy to port a molecular docking application: first, to generic compute graphs; and thereafter, to the actual AIEs. Finally, Section 5 concludes this paper with a summary and outlook to future work.

2 Related Work

Other efforts to make AIEs easier to use exist. For instance, PyAIE [13] is a Python-based programming framework that performs the following: 1) it enables users to implement algorithms in Python instead of C/C++, and 2) it maps these user-written functions into host code as well as to Versal compute units (PL, AIEs) by automatically translating Python into C/C++ code. In contrast to PyAIE, which is a high-level abstraction, Graphtoy does not perform automatic code-generation but rather enables users to iterate more quickly on low-level AIE kernel and graph designs, *without* having to rely on the vendor-provided AIE toolchains and simulators. Moreover, Graphtoy is explicitly designed as a library to be integrated into pre-existing C++ codebases, and not as a framework to write applications in from scratch.

Versal SoCs are currently examined with great interest for High Performance Computing (HPC) applications, e.g., [7] leverages AIEs for accelerating atmospheric advection computations. Our work aims to speed up and simplify the development of compute graphs for such applications by decoupling graph design from vendor-provided frameworks and boosting the simulation speed. Similarly to the above work, we attempt to port an HPC application (namely, molecular docking) to the AIEs, but with a focus on easy-to-use, rapid turnaround simulation, instead of optimizing the performance of executing on actual hardware AIEs.

3 The Graphtoy library

3.1 Design

Graphtoy uses C++20 coroutines [8] and a simple task scheduler to implement cooperative multitasking, which enables the high-throughput simulation of a multi-core system on a single host CPU thread *without* the overhead of actual thread context switches. Listing 1.1 shows an example of a compute kernel implemented in Graphtoy. Each kernel in a compute graph is represented by a potentially infinitely-running coroutine, which can receive and emit data via an arbitrary number of typed input and output streams. The rate at which data is transferred via these streams is also arbitrary: a kernel coroutine can read from and write to any of its connected streams at any point during its execution.

```cpp
struct ExampleKernel: GtKernelBase {
    ExampleKernel(GtContext *ctx): GtKernelBase(ctx) {}

    GtKernelIoStream<int> *m_input1 = addIoStream<int>();
    GtKernelIoStream<int> *m_input2 = addIoStream<int>();
    GtKernelIoStream<int> *m_output = addIoStream<int>();

    GtKernelCoro kernelMain() override {
        while (true) {
            int a = co_await m_input1->read();
            int b = co_await m_input2->read();
            co_await m_output->write(a + b);
        }
    }
}
```

Listing 1.1: Complete source code of a compute kernel implemented in Graphtoy. This kernel reads integers from two input streams and writes their sums to an output stream.

Internally, these streams are backed by multi-producer multi-consumer FIFOs with per-stream configurable sizes. When a stream read or write can not be fulfilled immediately (because the FIFO is empty or full, respectively), the calling kernel coroutine is suspended by saving its execution state to a pre-allocated memory arena (the *coroutine frame*). Graphtoy's scheduler can then resume another coroutine from its *ready list*. Once the condition that caused a kernel coroutine to suspend is alleviated, the coroutine is placed on the scheduler's ready list for eventual resumption. The order in which pending coroutines are resumed is unspecified, as Graphtoy does not implement any kind of scheduling priority system.

Figure 1 indicates that Graphtoy stores the kernels of a graph and their connections in a `context` object. This object must be reconstructed for each invocation of a particular graph, as it also contains management information. Examples of management information includes the kernel coroutine instances, the scheduler's ready list, and the input/output connections that push data into and drain data out of the graph, respectively. As (re)construction of the `context` can be performed very quickly, this requirement does not hamper the speed of Graphtoy-based development flows.

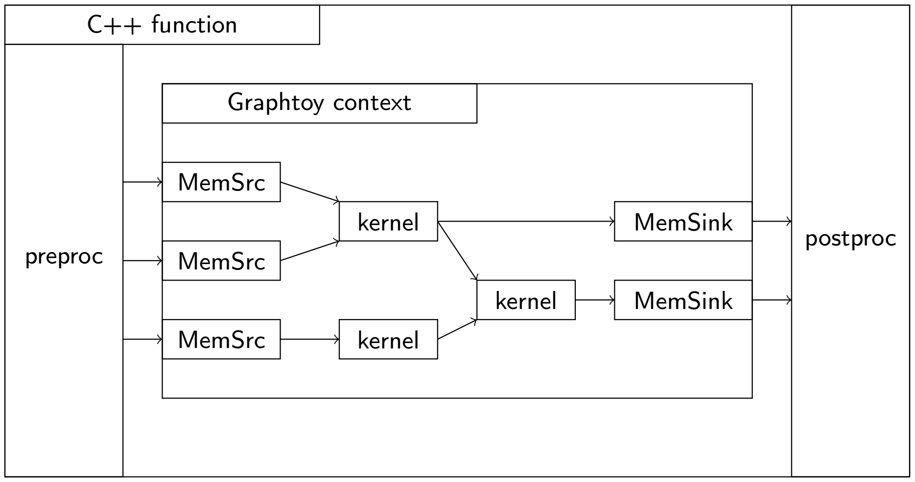

Fig. 1: Scheme of a simulated compute graph embedded into a plain C++ function using Graphtoy.

One particularity of Graphtoy is the way that graph execution is terminated: there is no explicit termination condition, and the graph execution cannot be terminated early. Instead, the scheduler continuously resumes coroutines until its ready list is empty, at which point the simulation ends[2]. As a result, Graphtoy executes a graph until all of its coroutines are halted and the graph can therefore no longer make forward progress. This happens when a) all input data has been

[2] C++20 coroutines can be destroyed cleanly whenever they are suspended, which makes this graph termination scheme possible. Any local variables that are alive within the coroutine at its point of destruction will be destroyed and deallocated as well.

```
WARNING: Graphtoy detected data stuck in one or more I/O buffers of kernel 4 (an instance
      of 31Kernel_InterE_InterpolateEnergy).
=> Input stream 2 (of type d) contains unprocessed data: 1 of 1 FIFO entries in use.
=> Currently active coroutine frames in this kernel (frame trace), deepest last:
   #1 in virtual graphtoy::GtKernelCoro Kernel_InterE_InterpolateEnergy::kernelMain() (./
     processligand_graphtoy.cpp, line 611, col 60)
=> Note: This kernel is likely the cause of the deadlock because none of its output
     streams are blocked.
```

Listing 1.2: Example of Graphtoy's deadlock detection trace (truncated). We encountered this deadlock while implementing the `Inter_E` compute graph of the molecular docking algorithms (detailed in Section 4 and Figure 4).

fully processed (i.e., all kernels are idle and blocked waiting for data), or b) the graph encounters a deadlock (i.e., cyclic wait).

We considered but ultimately rejected adding an explicit termination call for the following reason: if a graph is terminated when it can still make forward progress, it has not fully processed all input data yet and the graph's output will be incorrect[3]. Conversely, once a graph has processed all input data, it can no longer make forward progress and will terminate automatically anyway.

To ease graph debugging, Graphtoy detects deadlocks in the simulated graph using a simple algorithm: if any stream FIFO still contains data after the completion of a graph's execution, a deadlock likely occurred as some input data has not been fully processed in that case. Conversely, if all FIFOs are empty when the graph terminates, no deadlock occurred (assuming the simulated graph is a DAG, which is not checked by Graphtoy[4]). Whenever a deadlock is detected, Graphtoy prints debug information containing the state of all involved kernel coroutines and FIFOs.

For this purpose, we implement a tracer that maintains a call stack for each kernel coroutine as it suspends, resumes, and recurses into sub-coroutines. This is necessary because the call stack of a C++20 coroutine is lost between suspension and resumption, as only the coroutine's immediate execution state is saved. Using this information, Graphtoy can print the source location at which each involved kernel is suspended, even if it is in a deeply nested stack of sub-coroutines. Listing 1.2 shows an example of such a trace.

Other debugging tasks, such as single-stepping and placing breakpoints, can be performed using a standard C++ debugger such as *gdb* (the GNU debugger).

Besides connections between kernels, Graphtoy also supports simulated stream-based DMA, which can read data from a linear memory region (MemSrc) and stream it into the compute graph, or take a stream of data from the graph and write it out to memory (MemSink). This functionality is implemented as a pair

[3] Due to the unspecified kernel scheduling order, the output might also be unpredictable as some pending kernel iterations may or may not have been run yet at the time of early graph termination.

[4] Simulation of cyclic compute graphs is possible in Graphtoy, but may lead to deadlocks not being detected when they occur.

of helper kernels and emulates the AI Engines' GMIO. Additionally, Graphtoy supports basic packet routing (stream splitting, merging, and broadcasting), as well as input overlap windows, which are also implemented using helper kernels and coroutines.

3.2 Usage

In order to use Graphtoy, there are two prerequisites: 1) a C++ compiler supporting coroutines, and 2) the addition of Graphtoy's source files into the target application project (i.e., an HPC program). Once this is done, the user can *incrementally* add compute kernels and graphs to the traditional application, as depicted in Figure 1, and does not have to perform a "big bang" port to a new tool flow and execution environment.

Graphtoy represents kernels as heap-allocated objects. As indicated in Listing 1.1, to create a new kernel, the user creates a class that derives publicly from `GtKernelBase`. This class must provide a constructor taking a `GtContext *` as its first argument, which in turn must be forwarded to the constructor of `GtKernelBase`. The "contents" of the kernel are specified by overriding the `GtKernelBase::kernelMain()` function, which is actual the kernel coroutine. I/O streams are instantiated as needed within the kernel by using the `GtKernelBase::addIoStream` function template. The three main characteristics of Graphtoy's kernels are the following: 1) kernel classes can be instantiated and added to graphs as often as desired. 2) Kernels can be parameterized (analogously to runtime parameters in the AIE framework) as well as preloaded with look-up tables, both by adding further arguments to the kernel's constructor. 3) As kernels are just regular C++ classes, they can be templated if desired.

Once the kernel(s) have been defined, the actual *compute graph* can be constructed. On each invocation of the graph, a new `GtContext` object has to be created, which can then be populated with its constituent kernels as well as memory sources and sinks (including the definition of memory regions read by the sources). Once incorporated into the context, all of these components can be connected to each other via graph edges. Finally, the graph can be executed and the results can be read from the memory sinks.

Listing 1.3 shows the source code of such a function, which contains and simulates a simple one-kernel graph that adds two arrays of integers. It is important to note that the `GtContext` object cannot be reused once the graph has been run. Instead, it is necessary to re-create the graph on every invocation, as implemented in the example.

As graphs formulated using Graphtoy are completely encapsulated within regular C++ functions, it is possible to easily replace pre-existing functions of the target application – one at a time – with graph versions, without having to change the application's overall software architecture. This means that the user retains a running application at *all times* during their graph prototyping effort. Additionally, since Graphtoy kernels are regular C++ code, which is compiled and linked together with the rest of the user application, it is possible to call pre-existing library and helper functions from within those kernels.

```cpp
auto addIntsWithGraph(
    std::span<const int> a,
    std::span<const int> b)
{
    GtContext ctx{};

    auto& srcA  = ctx.addKernel<GtMemStreamSource<int>>(a);
    auto& srcB  = ctx.addKernel<GtMemStreamSource<int>>(b);
    auto& adder = ctx.addKernel<ExampleKernel>();
    auto& sink  = ctx.addKernel<GtMemStreamSink<int>>();

    ctx.connect(srcA.output(), adder.m_input1);
    ctx.connect(srcB.output(), adder.m_input2);
    ctx.connect(adder.m_output, sink.input());

    ctx.runToCompletion();

    return sink.data();
}
```

Listing 1.3: Complete source code of a compute graph instantiation and simulation using Graphtoy. This compute graph is based on the compute kernel from Listing 1.1.

3.3 Porting Graphs from Graphtoy to the AI Engines

Up to this step, compute graphs written in Graphtoy are generic (target-independent). While this level of description already is very helpful when incrementally moving over an application into a graph-parallel form, it is necessary to perform an additional porting step in order to run them on physical hardware.

Therefore, once the user is satisfied with the functionality of the Graphtoy-simulated compute graph, the user must modify these graphs to address two key differences between Graphtoy and the actual AIEs. First, in Graphtoy, kernels do not have to explicitly terminate. Instead, the execution of a graph stops when all of its kernels are blocked. For the actual AIEs, kernels are terminated explicitly. Second, Graphtoy does not distinguish between different types of streams between kernels, while the AIEs do (e.g., AXI4 streams, cascade accumulator connections, and direct inter-tile local memory accesses). When targeting actual AIE hardware, the nature of the streams must be specified.

In general, this refinement process starts with: 1) copying the kernel source code (`kernelMain` function) into the AIE project, and naming it appropriately. 2) The kernel's I/O streams and runtime parameters must be added to its function signature, choosing an appropriate I/O mechanism for each one. 3) All I/O operations in the kernel, which use `co_await` in Graphtoy, must be replaced with the corresponding AIE intrinsics, i.e., `readincr`, `writeincr`, and asynchronous window or buffer lock/unlock[5]. 4) For the explicit kernel termination used by the AIEs,

[5] It is important to note that the I/O windows or buffers must be asynchronous to allow the kernel *itself* to acquire and release them when desired, which is not possible with their synchronous versions, as those are managed automatically between kernel invocations by the AIE framework.

a termination condition must be added to the kernel's main loop. E.g., a sentinel value to be pushed through the entire graph after all data has been processed. 5) The kernels must be connected appropriately in a graph class, with the AIE window and buffer I/O ports being marked as asynchronous at connection time. 6) The entire graph can be started, generally with an iteration count of one for a single execution.

The addition of manual looping and termination to the kernels is necessary because the AIE framework only supports running a graph either indefinitely, or for a fixed number of iterations that is predetermined (and ideally equal) for all of its kernels [3]. Graphtoy does not impose such a restriction and allows kernel repetition counts to be determined *dynamically* at runtime. In fact, with Graphtoy, it is possible to write kernels with arbitrary control flow, including non-repeating kernels, dynamic dispatch, and kernels with multiple main loops. By manually implementing kernel repetition and termination for the AIEs and setting an iteration count of one, it is possible to replicate Graphtoy's more generic behavior on the actual platform. To some extent, some of these transformations could be automated, either by suitable C++ abstractions or external tooling. However, we believe that even in its current form, Graphtoy is an extremely useful tool due to the significantly improved development productivity we have observed, compared to using the AIE tools directly.

4 Case Study: Molecular Docking

As a sample use-case of Graphtoy, we discuss some of the steps required when porting a full-scale HPC scientific computing code to the AIE platform. Note that this use-case is *not* intended for benchmarking AIE hardware performance, as we have observed a number of issues with the AMD AIE toolchain that currently lead to the binaries being non-functional in AIE native mode. However, all of the graphs we discuss below *do* run correctly when executed in the AMD toolchain's x86 simulation mode.

Our use-case is AutoDock, a widely used molecular docking application that can employ, among others, a genetic algorithm to predict a chemical ligand's orientation and conformation as it docks to a biological receptor [9]. A real world application for AutoDock is drug discovery, where the interactions of many chemical compounds, e.g., with a bacterial or viral surface, is examined.

The core algorithms in AutoDock have been previously ported to run on GPUs [10], FPGAs [12], and even vector processors [11]. This part of our work explores the process of porting a mini-version of AutoDock to the AIE architecture, using Graphtoy as a rapid-turnaround prototyping tool.

4.1 Algorithm Overview

The mini-version of AutoDock that we employ for our porting efforts implements only the genetic algorithm variant. Based on that, genotypes produced by the genetic algorithm encode the ligand's bond rotation angles, as well as its absolute position and orientation in space. The fitness function is the ligand conformation's energy, which must be minimized for a high quality docking (ligand and receptor molecules "fit" well together in the computed conformation).

Figure 2 shows the phases of the core docking algorithm. First, new genotypes (solution representations) are created via the genetic algorithm and a local search heuristic. Then, the genotype is interpreted to derive the conformation of the ligand in space, which in turn, is used to compute the intramolecular (atom interactions *within* the ligand) and intermolecular (*between* ligand and receptor atoms) energies. During these computations, the ligand is represented as a list of atoms (each with type, coordinates, and charge). On the other hand, the receptor is modeled as a spatial grid with smoothing via trilinear interpolation.

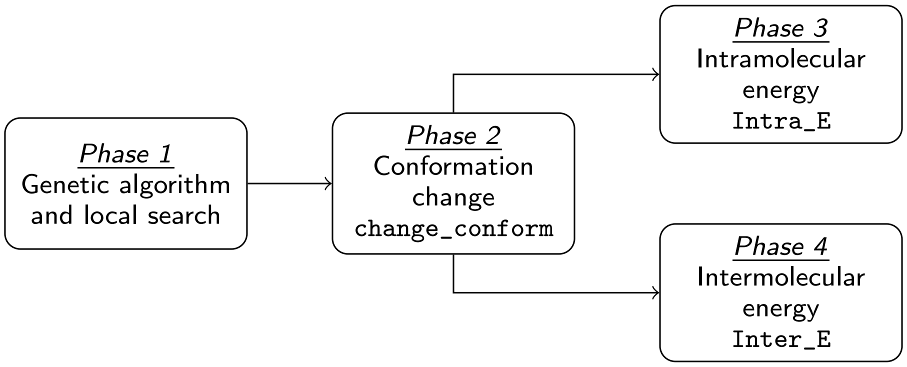

Fig. 2: Pipeline processing in our molecular docking case study.

4.2 Porting Methodology

To port this mini-version of AutoDock to the AIE array, we first include the Graphtoy library into the AutoDock source code. Then, by using Graphtoy, we develop compute graphs for three out of the four core docking algorithms: the conformation change, as well as the two energy calculation functions. In particular, we do not attempt to port the genetic algorithm nor the local search portions to the AIE array, as their code is branch intensive, rather than compute focused, making them more suitable for execution on a CPU core than on a graph processing array.

The conformation change algorithm (`change_conform`) needs to rotate each of the ligand atoms around a varying number of rotational bonds to determine their final positions. For each ligand atom, the order of rotations must remain the same, while all of these atoms are also streamed into and out of the AIE array in order. To keep the design of the compute graph simple, we use a *dynamic* pipelining approach.

Figure 3 illustrates the corresponding graph including its dynamic pipeline, which is based on a management kernel (Build rotate input) that determines the number of rotation operations that each one of a chain of downstream compute kernels (Partial rotation) should perform on atoms as they pass through the pipeline. Atom data is passed from one pipeline kernel to the next using asynchronous window I/O (win).

In order to evenly distribute workload in `change_conform`, the above number of rotations performed by each pipeline kernel varies for each processed atom. The genotype data is uploaded via global memory I/O (GMIO) and broadcast to all pipeline kernels once the graph is invoked, which requires only a single DMA channel, as the broadcast operation happens *within* the AIE array. Each pipeline kernel stores a copy of the genotype in local SRAM for fast lookup of the required bond rotation angles as atoms subsequently pass through.

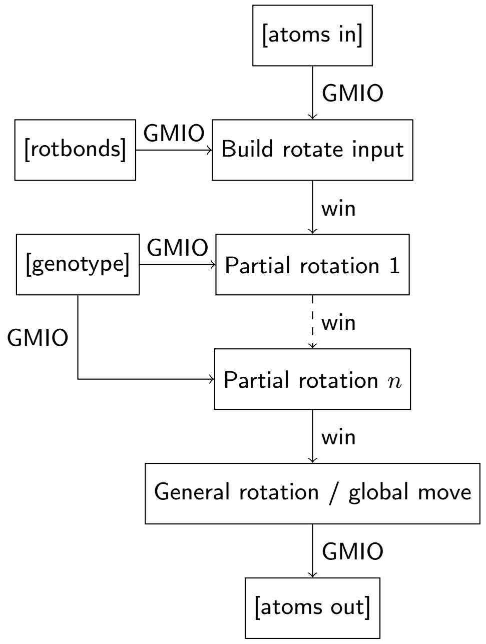

Fig. 3: `change_conform` compute graph. The dashed edge represents a chain of n identical *partial rotation* kernels.

The intramolecular energy calculation function (`Intra_E`) computes the distance-dependent energy contribution of pairs of atoms within the ligand. To speed this up, the algorithm uses a precomputed table that indicates which atom pairs must be processed and which can be skipped in the calculations. Transforming this algorithm into a compute graph was relatively straightforward, as it is suitable for *static* pipelining. The graph takes the precomputed atom pair table as input, and returns the final energy values via a runtime parameter (RTP) after they have been accumulated at the end of the pipeline. Additionally, the ligand-atoms' data is uploaded into one of the kernels' local SRAM using another input stream.

The main challenge of porting `Intra_E` was the large number of look-up tables that must be kept within the graph. To address this, we partition the algorithm into multiple kernels, so that none of these kernels requires more memory to hold look-up tables than the maximum available on an AIE tile (i.e., 32 KiB, including stack and management data).

Finally, the intermolecular energy calculation function (`Inter_E`) computes the energy contribution of the interactions between the ligand atoms and the receptor. The resulting energy value is again returned to the host system via a runtime parameter. Since the receptor is represented as a spatial grid, the algorithm has to perform lookups in the grid (based on the atoms' positions), leading to a random memory access pattern.

This pattern is problematic because the AIE array lacks the ability to autonomously perform random accesses on system DRAM [2], which means that `Inter_E` cannot be fully implemented on the AIE array alone. To actually perform such DRAM accesses, we will use a separate HLS kernel executing on the Programmable Logic (PL) of the Versal device, which we simulate by using a host-side helper in the AIE port of the graph. In the hardware implementation, this kernel will be connected to the rest of the graph using the programmable logic I/O (PLIO) interfaces of the SoC. Figure 4 illustrates this host-side helper as the DRAM reader (PL) kernel. In Graphtoy, modelling this HLS kernel does not need any special handling, as indicated in Listing 1.4.

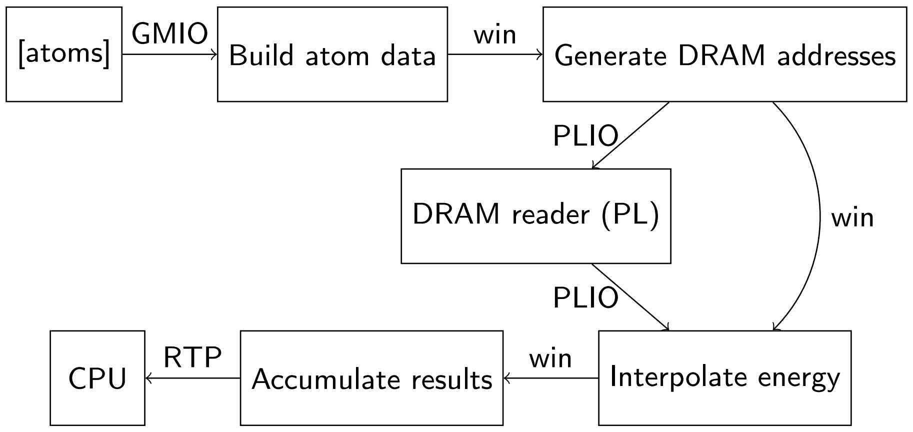

Fig. 4: `Inter_E` compute graph.

Once these three compute graphs are implemented and tested using Graphtoy, we are able to port them to the AIEs with no changes to their overall

```cpp
template<typename T>
struct DramReader: GtKernelBase {
    DramReader(GtContext *ctx, std::span<const T> mem):
        GtKernelBase(ctx), m_memoryRegion(mem) {}

    std::span<const T> m_memoryRegion;

    GtKernelIoStream<uint32_t> * m_addrIn  = addIoStream<uint32_t>();
    GtKernelIoStream<T>        * m_dataOut = addIoStream<T>();

    GtKernelCoro kernelMain() override {
        while (true) {
            uint32_t addr = co_await m_addrIn->read();
            co_await m_dataOut->write(m_memoryRegion[addr]);
        }
    }
}
```

Listing 1.4: A DRAM reader kernel, as used in the `Inter_E` compute graph, modelled for Graphtoy simulation. An HLS implementation of this kernel will be later used on the actual Versal hardware.

architecture. In addition, no large-scale debugging is required using the more complex AIE development flows, as the architecture has already been shown to be functionally correct using Graphtoy. Moreover, a minor change that does need to be made is switching the algorithms from double-precision floats to a mix of single-precision and fixed-point arithmetic on the AIEs. The latter is needed since trigonometric functions are only available as fixed-point variants on the AIE hardware [1]. A comparison between a Graphtoy kernel and its corresponding AIE version is shown in Listing 1.5.

4.3 Benchmarking Simulation Speed

As Graphtoy is designed to simplify compute graph prototyping as well as to speed-up simulation, we benchmark the simulation runtimes achieved by Graphtoy against those by AMD's AIE graph simulators (*x86sim* and *aiesim*) [5]. For this purpose, we employ the AutoDock molecular docking mini-version already discussed configured with a precomputed set of 10^4 ligand genotypes as input data for all experiments. Moreover, all evaluations are performed on an Intel Xeon W-3265 CPU, using Vitis 2022.2 for *x86sim* and *aiesim* versions, and GCC 12 with the `-O2` optimization setting for the native (the original non-graph AutoDock code) and Graphtoy versions[6].

Table 1 shows the corresponding simulation runtimes. As AMD's *aiesim* needed about 45 min to complete a *single* iteration of `Intra_E`, we canceled the simulation run after two iterations and extrapolated from the above baseline runtime. Hence, the estimated value for *aiesim* (i.e., 10+ months of simulation time) is only a *lower* bound. The actual time required to complete the simulation

[6] We also tried `-O3`, which led to a slight performance degradation for Graphtoy, and no change for the non-graph version.

on *aiesim* is likely higher due to the additional time that would be required for simulating `change_conform` and `Inter_E`.

The overall results can be interpreted as follows: the Graphtoy simulator introduces significant overhead over the native (non-graph) version of AutoDock, due to the additional simulation of kernel-to-kernel communication. This was determined via profiling with `perf`. However, Graphtoy is around an order of magnitude faster than AMD's *x86sim*.

```
// Graphtoy version.

// Runtime parameters, set in kernel constructor
double m_genrot_unitvec[3] = {};
double m_genrot_angle;
double m_globalmove_xyz[3] = {};

GtKernelCoro kernelMain() override {
    while (true) {
        auto data = co_await m_inputStream->read();
        double *atom_xyz = &data.m_atomdata.m_atom_idxyzq[1];

        const double genrot_movvec[3] = {0, 0, 0};
        rotate(atom_xyz, genrot_movvec, m_genrot_unitvec, &m_genrot_angle, 0);
        vec3_accum(atom_xyz, m_globalmove_xyz);

        co_await m_outputStream->write(data.m_atomdata);
    }
}


// AIE version.

void changeConform_GeneralRotation_GlobalMove(
        input_window<uint8> *atomData_in,

        // Runtime parameters, set via graph API
        const float (&genrotUnitvec)[3],
        const float genrotAngle,
        const float (&globalMoveXyz)[3],

        output_stream<float> *idxyzq_out)
{
    while (true) {
        auto data = readTypedWindowData<ChangeConform_AtomData>(atomData_in);
        if (data.m_isTerminationSentinel) break;

        float * const atom_xyz = &data.m_atom_idxyzq[1];

        static constexpr float genrotMovvec[3] = {0, 0, 0};
        rotate(atom_xyz, genrotMovvec, genrotUnitvec, genrotAngle);
        vec3_accum(atom_xyz, globalMoveXyz);

        for (uint32 i = 0; i < 5; ++i) {
            writeincr(idxyzq_out, data.m_atom_idxyzq[i]);
        }
    }
}
```

Listing 1.5: Comparison of the Graphtoy and AIE versions of the General rotation / global move kernel, as used in the `change_conform` compute graph.

Both of the aforementioned simulators are directly comparable, since they both compile compute graphs natively for the x86 host architecture and do not emulate microarchitectural details of the AI Engines. AMD's *aiesim* is the slowest among all tested simulators. This is because *aiesim* does simulate AIEs on a micro-architectural level, unlike the others.

Table 1: Graph simulation runtime benchmarks.

AMD x86sim	**AMD aiesim**	**Native (non-graph)**	**Graphtoy**
299 s	10+ months	0.311 s	27.5 s

5 Conclusions and Future Work

In this work, we have used C++20 coroutines as a building block for Graphtoy, which is a new compute graph simulator optimized for ease of development and low simulation overhead. For our case study consisting of compute graphs for a mini-AutoDock, Graphtoy achieves simulation times *at least an order of magnitude faster* compared to AMD's AI Engine simulators (*x86sim* and *aiesim*).

Our Graphtoy-based port of molecular docking algorithms into compute graphs showed that the kernel and graph constructs provided by Graphtoy map well to the real AIEs. As we have also shown, Graphtoy easily enables a straightforward prototyping of algorithms requiring co-processing between the AIEs and the FPGA PL, e.g., for custom memory accesses. For instance, in cases where random accesses to large memory blocks were needed, we were able to prototype such PL interfaces by simply adding a Graphtoy kernel, without the need to fully develop an HLS or RTL hardware kernel for the PL in advance.

As mentioned above, we will consider to extend Graphtoy to further simplify/automate the Graphtoy-to-AIE porting process in a future refinement of the system.

Acknowledgments. We would like to thank AMD/Xilinx for donations of AIE hardware and design tools.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. AMD: AI Engine Intrinsics: Elementary Functions (2021), https://www.xilinx.com/htmldocs/xilinx2021_2/aiengine_intrinsics/\intrinsics/group__intr__elem.html

2. AMD: AI Engine Kernel and Graph Programming Guide (UG1079): Graph Programming Model (2023), https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/Graph-Programming-Model
3. AMD: AI Engine Kernel and Graph Programming Guide (UG1079): Run-Time Graph Control API (2023), https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/Run-Time-Graph-Control-API
4. AMD: AMD AI Engine Technology: AI Engine Development Flows (2023), https://www.xilinx.com/products/technology/ai-engine.html#developers
5. AMD: Versal Adaptive SoC Design Guide (UG1273): Design Flows (2023), https://docs.xilinx.com/r/en-US/ug1273-versal-acap-design/Design-Flows
6. AMD: Versal Adaptive SoC Design Guide (UG1273): System Architecture (2023), https://docs.xilinx.com/r/en-US/ug1273-versal-acap-design/System-Architecture
7. Brown, N.: Exploring the Versal AI Engines for Accelerating Stencil-Based Atmospheric Advection Simulation. In: ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays (FPGA). p. 91–97. ACM (2023). https://doi.org/10.1145/3543622.3573047
8. cppreference.com: Coroutines (C++20) (2023), https://en.cppreference.com/w/cpp/language/coroutines
9. Morris, G.M., Goodsell, D.S., Huey, R., Hart, W.E., Halliday, S., Belew, R., Olson, A.J.: AutoDock User's Guide, Version 3.0.5 (2001), https://autodock.scripps.edu/wp-content/uploads/sites/56/2022/04/AutoDock3.0.5_UserGuide.pdf
10. Santos-Martins, Diogo and Solis-Vasquez, Leonardo and Tillack, Andreas F and Sanner, Michel F and Koch, Andreas and Forli, Stefano: Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. J. Chem. Theory Comput. (JCTC) **17**(2), 1060 – 1073 (2021). https://doi.org/10.1021/acs.jctc.0c01006
11. Solis-Vasquez, L., Focht, E., Koch, A.: Mapping Irregular Computations for Molecular Docking to the SX-Aurora TSUBASA Vector Engine. In: 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3). pp. 1–10. IEEE (2021). https://doi.org/10.1109/IA354616.2021.00008
12. Solis-Vasquez, L., Koch, A.: A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software. In: 5th Int. Workshop on FPGAs for Software Programmers (FSP). pp. 1–10. VDE Verlag (2018), https://ieeexplore.ieee.org/document/8470463
13. Tian, H., Yang, S., Cha, Y., Huang, S.: Late Breaking Results: PyAIE: A Python-based Programming Framework for Versal ACAP Platforms. In: 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–2. IEEE (2023). https://doi.org/10.1109/DAC56929.2023.10247843