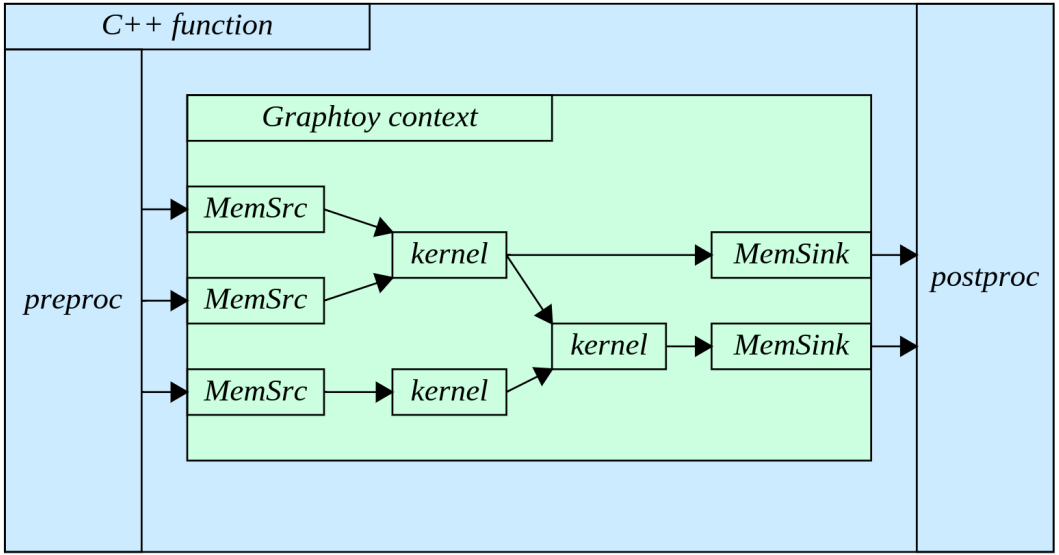
Graphtoy: Fast Software Simulation of Applications for AMD's AI Engines

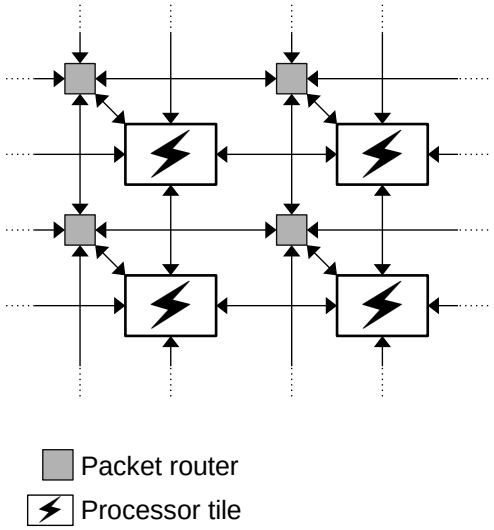
Jonathan Strobl, Leonardo Solis-Vasquez, Yannick Lavan, Andreas Koch
Embedded Systems and Applications Group

Technical University of Darmstadt



Graphtoy: AIE Introduction

- AMD Versal adaptive SoC
  - Heterogeneous accelerator
  - Integrates CPUs, FPGA, **AI Engines**, DSP Engines, …
- AI Engine array
  - 2D array of VLIW SIMD processors
  - Streaming connections

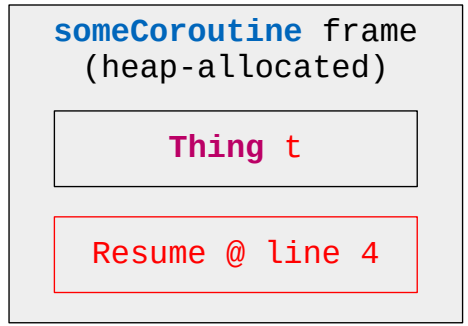


Graphtoy: C++20 Coroutine Introduction

TECHNISCHE UNIVERSITÄT DARMSTADT

- **C++20 coroutines = suspendable functions**
  - Based on stackless **coroutine frames**
  - Not threads!

```
1  CoroutineType someCoroutine() {
2      Thing t = getStuff();
3
4      bool success = co_await doStuff(t);
5
6      return success ? t : {};
7  }
```



Graphtoy: C++20 Coroutine Introduction

TECHNISCHE UNIVERSITÄT DARMSTADT

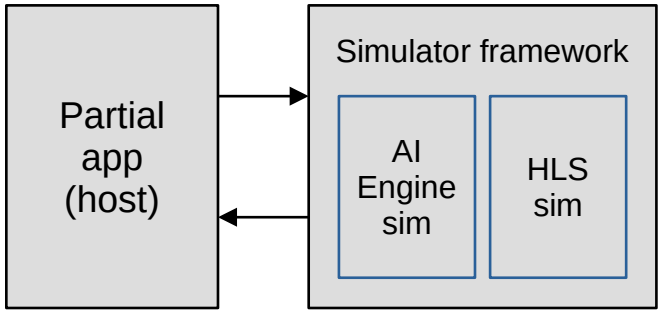
- Coroutines enable new programming patterns
  - Async / await
  - Generators
  - **Cooperative multitasking**
- **Bare-bones compiler support**
  - Semantics user-defined via callbacks

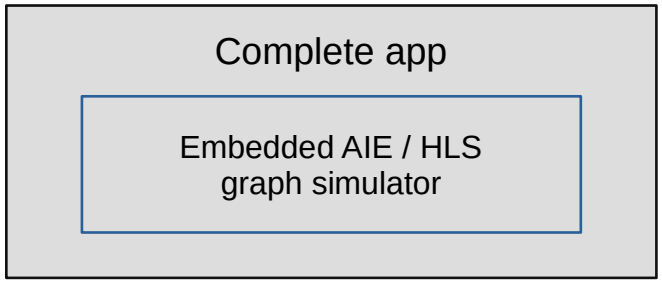
Embedded Systems & Applications

Graphtoy: Motivation

- **Vendor simulators are difficult to use**
  - Tedious set-up process
  - Limited debugging capabilities
  - Often quirky, buggy, and **slow**

- Explore the approach of **embedding a simulator into an application**





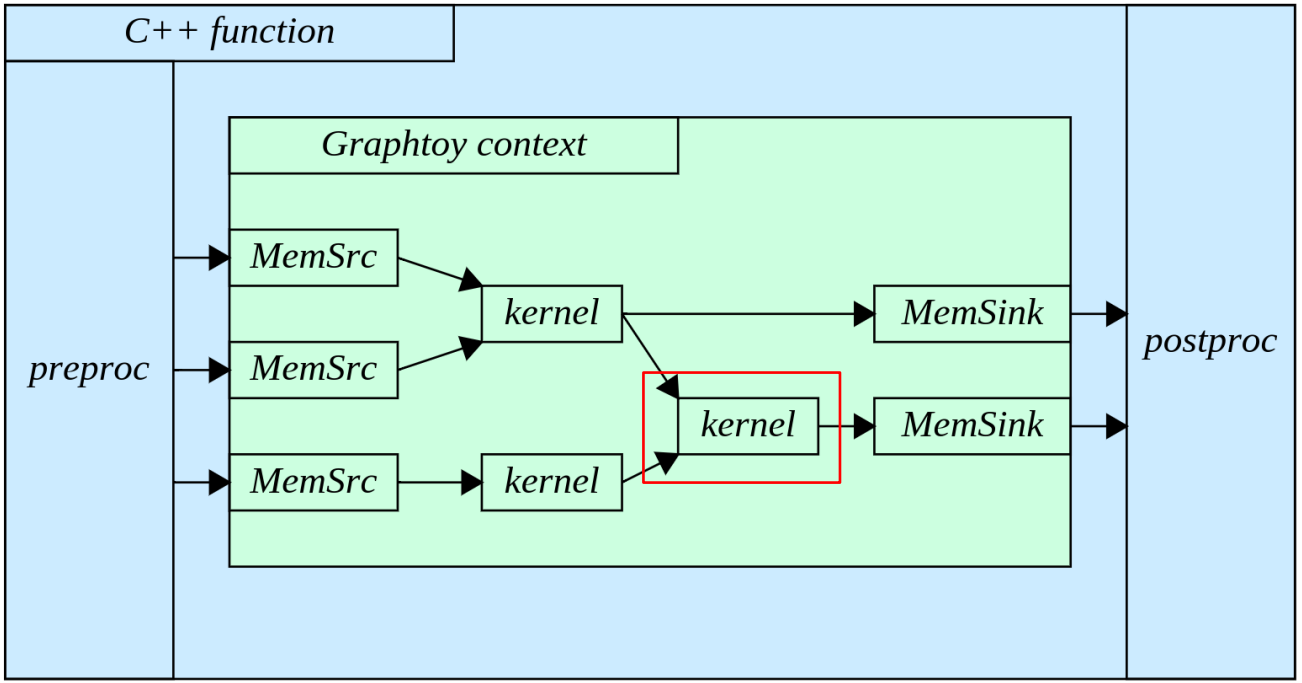
Graphtoy: Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

- Compute **graph simulator based on coroutines**
  - For simulating **AI Engine** graphs (and other architectures)
- Pure C++
  - Architecture independent
  - Can leverage **existing debuggers**
- Can be **integrated into existing applications**
  - **Rapid prototyping** of graphs

Embedded Systems & Applications

Graphtoy: Overview



Graphtoy: Kernel Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  struct ExampleKernel: GtKernelBase {
2      ExampleKernel(GtContext *ctx): GtKernelBase(ctx) {}
3
4      GtKernelIoStream<int> *m_input1 = addIoStream<int>();
5      GtKernelIoStream<int> *m_input2 = addIoStream<int>();
6      GtKernelIoStream<int> *m_output = addIoStream<int>();
7
8      GtKernelCoro kernelMain() override {
9          while (true) {
10             int a = co_await m_input1->read();
11             int b = co_await m_input2->read();
12             co_await m_output->write(a + b);
13         }
14     }
15 }
```

Embedded Systems & Applications

Graphtoy: Graph Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  auto addIntsWithGraph(
2      std::span<const int> a,
3      std::span<const int> b)
4  {
5      GtContext ctx{};
6
7      auto& srcA  = ctx.addKernel<GtMemStreamSource<int>>(a);
8      auto& srcB  = ctx.addKernel<GtMemStreamSource<int>>(b);
9      auto& adder = ctx.addKernel<ExampleKernel>();
10     auto& sink  = ctx.addKernel<GtMemStreamSink<int>>();
11
12     ctx.connect(srcA.output(), adder.m_input1);
13     ctx.connect(srcB.output(), adder.m_input2);
14     ctx.connect(adder.m_output, sink.input());
15
16     ctx.runToCompletion();
17
18     return sink.data();
19 }
```

Embedded Systems & Applications

Graphtoy: Implementation

TECHNISCHE UNIVERSITÄT DARMSTADT

- **GtContext holds a graph** as it's built and executed
- Graphtoy's **main class**

```
1  auto addIntsWithGraph(
2      std::span<const int> a,
3      std::span<const int> b)
4  {
5      GtContext ctx{};
6
7      auto& srcA  = ctx.addKernel<GtMemStreamSource<int>>(a);
8      auto& srcB  = ctx.addKernel<GtMemStreamSource<int>>(b);
9      auto& adder = ctx.addKernel<ExampleKernel>();
10     auto& sink  = ctx.addKernel<GtMemStreamSink<int>>();
11
12     ctx.connect(srcA.output(), adder.m_input1);
13     ctx.connect(srcB.output(), adder.m_input2);
14     ctx.connect(adder.m_output, sink.input());
15
16     ctx.runToCompletion();
17
18     return sink.data();
19 }
```


Embedded Systems & Applications

Graphtoy: Implementation

TECHNISCHE UNIVERSITÄT DARMSTADT

- The **coroutine scheduler** is implemented in GtContext
  - Run a graph via **runToCompletion()**
- Keeps running until **all kernels are blocked on I/O**
  - **No explicit termination condition!**

```
1  auto addIntsWithGraph(
2      std::span<const int> a,
3      std::span<const int> b)
4  {
5      GtContext ctx{};
6
7      auto& srcA  = ctx.addKernel<GtMemStreamSource<int>>(a);
8      auto& srcB  = ctx.addKernel<GtMemStreamSource<int>>(b);
9      auto& adder = ctx.addKernel<ExampleKernel>();
10     auto& sink  = ctx.addKernel<GtMemStreamSink<int>>();
11
12     ctx.connect(srcA.output(), adder.m_input1);
13     ctx.connect(srcB.output(), adder.m_input2);
14     ctx.connect(adder.m_output, sink.input());
15
16     ctx.runToCompletion();
17
18     return sink.data();
19 }
```


Embedded Systems & Applications

Graphtoy: Implementation

- The **coroutine scheduler** is implemented in GtContext
  - Run a graph via **runToCompletion()**
- Keeps running until **all kernels are blocked on I/O**
  - **No explicit termination condition!**

```
 8  GtKernelCoro kernelMain() override {
 9      while (true) {
10          int a = co_await m_input1->read();
11          int b = co_await m_input2->read();
12          co_await m_output->write(a + b);
13      }
14  }
```

Graphtoy: Stream Interfaces

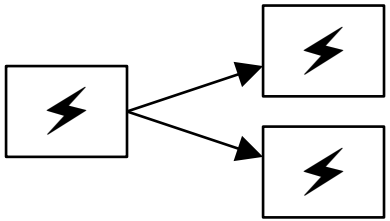
- Kernel I/O streams
  - **MPMC FIFOs**
- Data transfer:
  - **read(), write()**
  - Stream not ready?
    → Kernel **suspends**
- **Cooperative multitasking**

```
 8  GtKernelCoro kernelMain() override {
 9      while (true) {
10          int a = co_await m_input1->read();
11          int b = co_await m_input2->read();
12          co_await m_output->write(a + b);
13      }
14  }
```

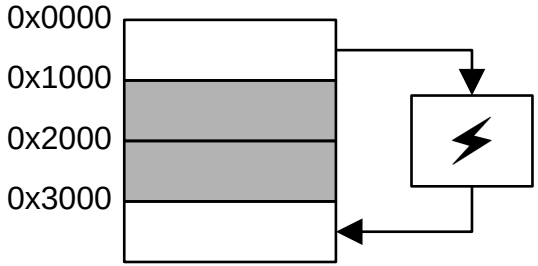
Graphtoy: Stream Connections

TECHNISCHE UNIVERSITÄT DARMSTADT

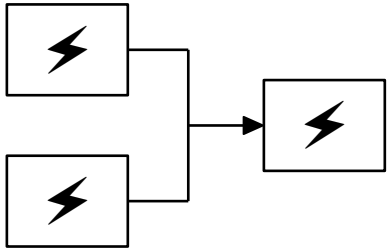
- Kernel I/O mechanisms **match the AI Engine** functionality



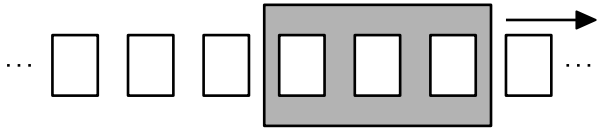
Broadcast



DMA source & sink



Packet split & merge

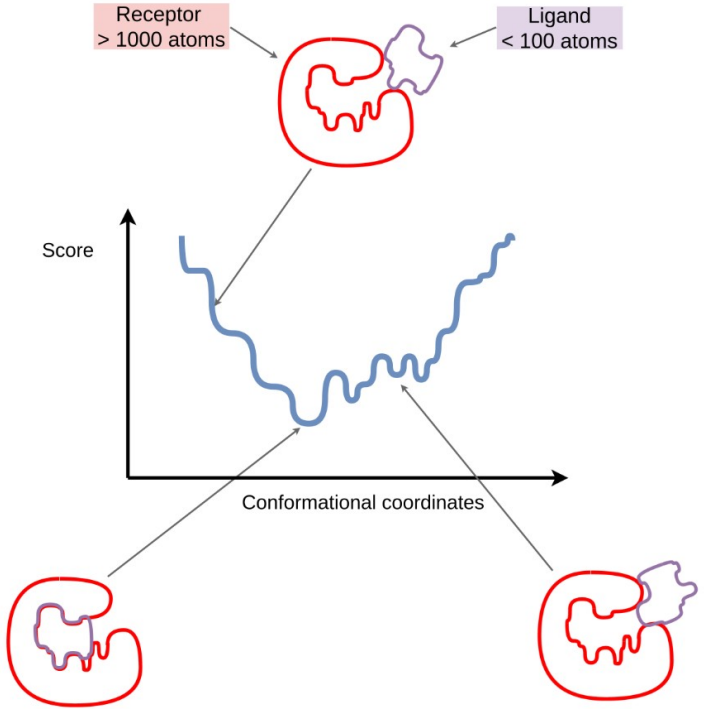


Sliding windows

Embedded Systems & Applications

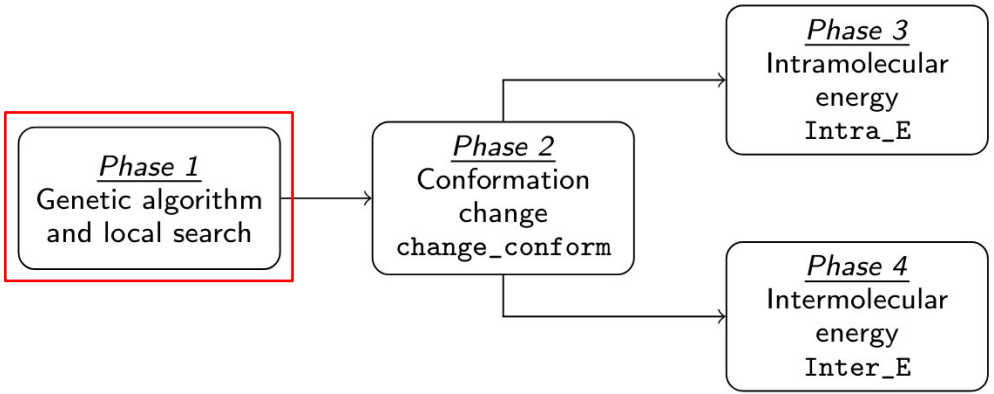
Molecular Docking: AutoDock Introduction

- Simulates **interaction between a ligand and a receptor**
- Uses a **genetic algorithm**
- Fitness function: **energy of a given ligand conformation**



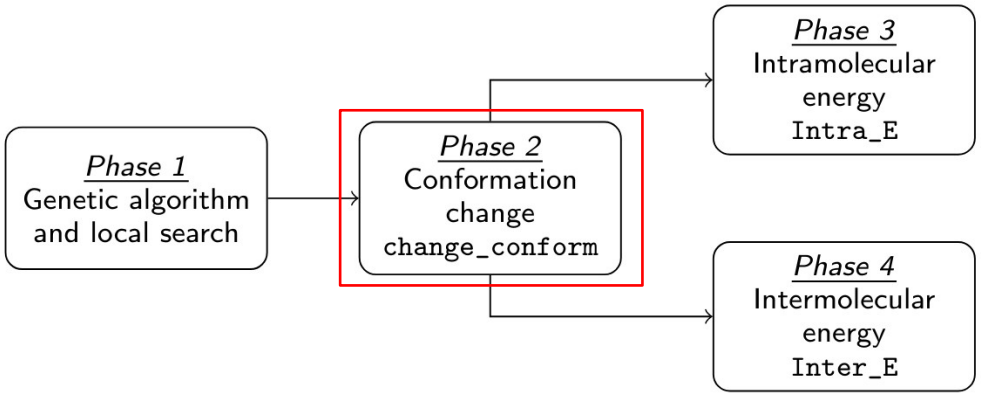
Molecular Docking: Porting Goals

- Port core algorithms to AI Engines
  - Ligand conformation change, energy calculation
- Approach: Prototype with Graphtoy, then switch to AIE



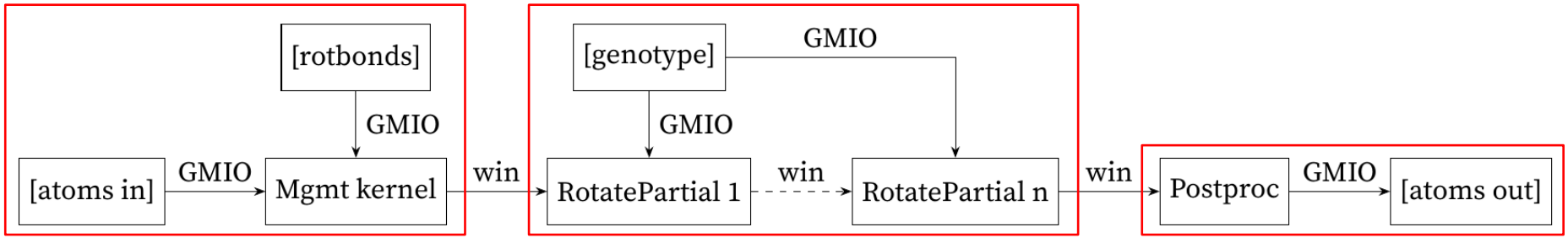
Molecular Docking: Conformation Change

- Take a **genotype** from the genetic algorithm
- **Reposition** the atoms in the ligand
  - Ligand „pose“ from genotype



Molecular Docking: Conformation change on AIE

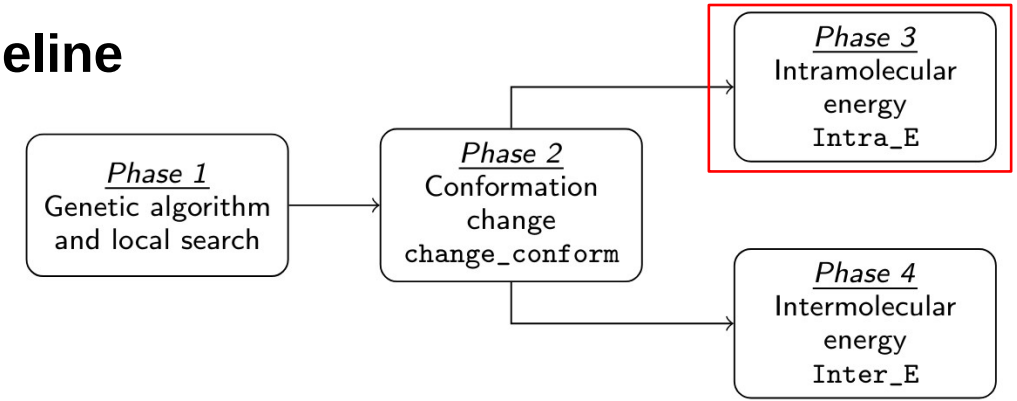
- **Dynamic** pipeline



- GMIO: Global Memory I/O
- win: I/O window

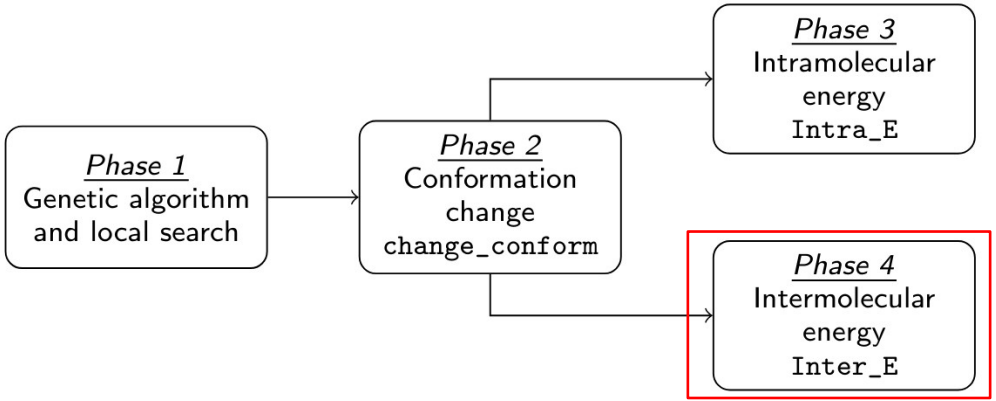
Molecular Docking: Intramolecular Energy

- Interactions between **pairs of atoms** of the ligand
- Needs many large constant tables
  - Don’t fit into local memory (32KiB per kernel)
  - Split over multiple AIE kernels
- **Static pipeline**



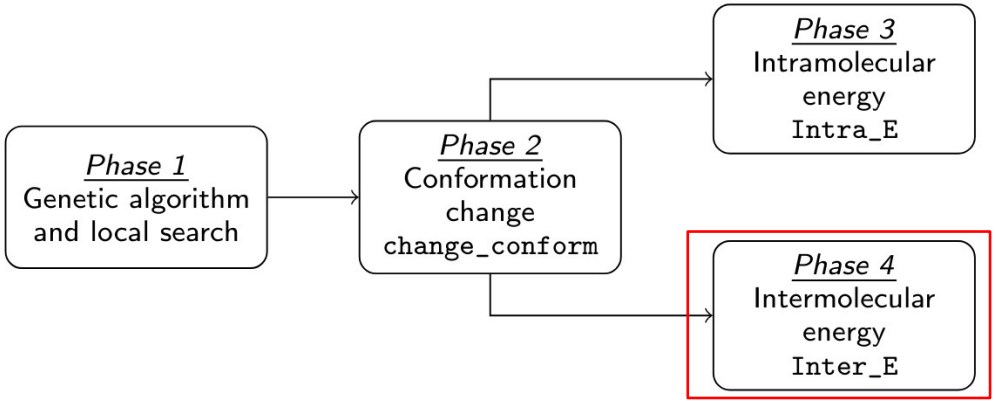
Molecular Docking: Intermolecular Energy

- Interactions between ligand atoms and the receptor
- Receptor represented as spatial grid
  - Too large to fit in kernel SRAM (~20MiB)

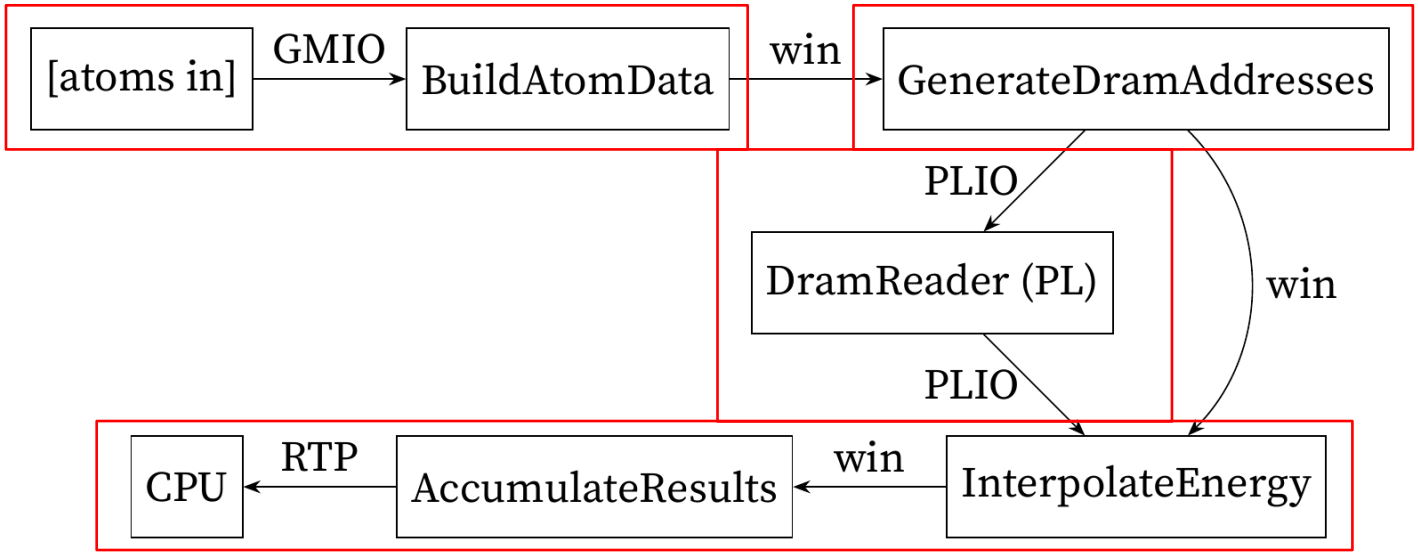


Molecular Docking: Intermolecular Energy

- Place grid in system DRAM
- **No random DRAM access** from AI Engine kernels
- **HLS kernel in FPGA fabric** for grid lookups in DRAM
  - Simulated in Graphtoy



Molecular Docking: InterE on AIE



- GMIO: Global Memory I/O
- win: I/O window
- PLIO: Programmable Logic I/O
- RTP: Runtime Parameter

Molecular Docking: Porting to AI Engines

- Graphs were **developed and validated in Graphtoy**
- Then **ported onto the actual AI engines**
  - Choose I/O mechanisms: AXI4-stream or windows
  - Use built-in AIE intrinsics to access streams

```
1  void kernelMain(
2      input_window<uint8_t> dataIn,
3      output_window<uint8_t> dataOut)
4  {
5      while (true) {
6          auto data = read(dataIn);
7
8          if (data == SENTINEL_IN) {
9              write(dataOut, SENTINEL_OUT);
10             break;
11         }
12
13         auto result = process(data);
14         write(dataOut, result);
15     }
16 }
```

Molecular Docking: Kernel Termination on AIE

TECHNISCHE UNIVERSITÄT DARMSTADT

- AIE kernels can run either **indefinitely**, or for a **fixed number of iterations**
- **Manually terminate kernels**
  - Push a **termination sentinel** through the graph
- Invoke the entire graph for **one iteration**
  - Kernels loop over input data on their own

```
1  void kernelMain(
2      input_window<uint8_t> dataIn,
3      output_window<uint8_t> dataOut)
4  {
5      while (true) {
6          auto data = read(dataIn);
7
8          if (data == SENTINEL_IN) {
9              write(dataOut, SENTINEL_OUT);
10             break;
11         }
12
13         auto result = process(data);
14         write(dataOut, result);
15     }
16 }
```


Embedded Systems & Applications

Molecular Docking: AIE Port Status

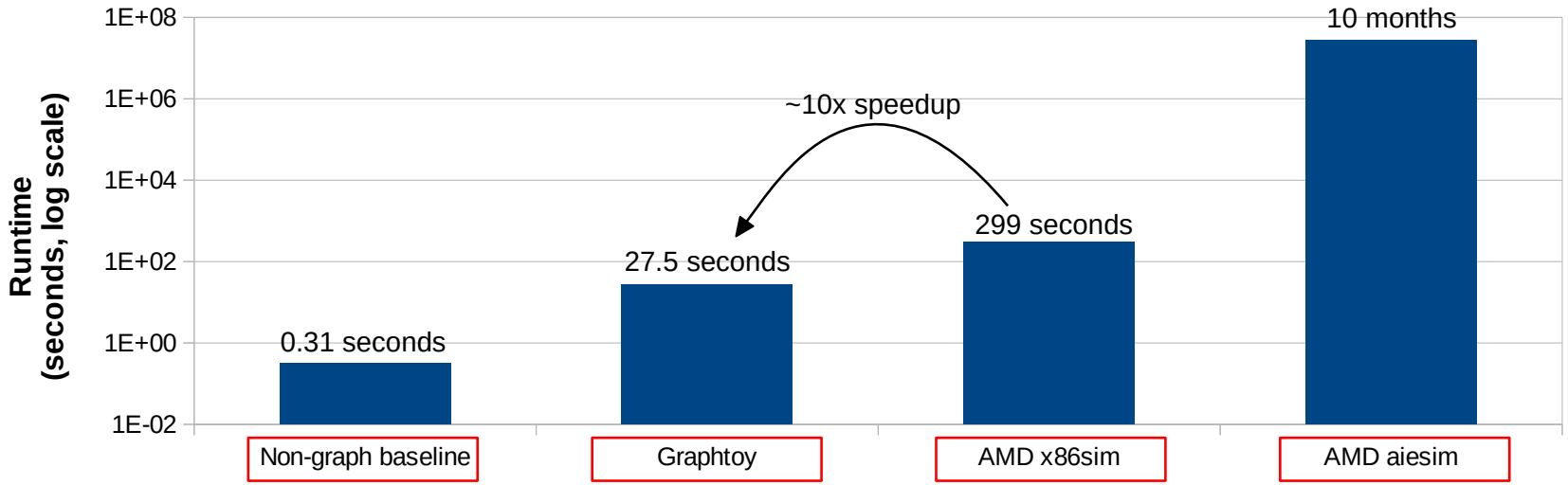
TECHNISCHE UNIVERSITÄT DARMSTADT

- Software emulation (x86sim) ✓
  - Imprecise results (15-bit sine)
- Hardware emulation (aiesim) ~
  - change_conform, IntraE ✓
  - InterE: hits a compiler bug ✘
- Not tested on real hardware yet
  - Waiting for aiecompiler bugfixes

Embedded Systems & Applications

Molecular Docking: Graph Sim Performance

- Graph simulator benchmark: 10k AutoDock iterations
  - x86sim / aiesim: Vitis 2022.2
  - aiesim result extrapolated: 1 iteration = 45min



Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

- A coroutine-based graph simulator was developed (Graphtoy)
  - Easy to use, quick to get started with
  - Embeddable into target application
- Case study: Molecular docking
  - Ported to compute graphs with Graphtoy
    - Successful
  - Then ported to real AIEs
    - Partially successful

Embedded Systems & Applications

Future Work

- Automate Graphtoy-to-AIE translation
- Or eliminate translation completely (source-compatibility)

Thank You For Your Attention!

TECHNISCHE UNIVERSITÄT DARMSTADT

- Graphtoy is released as open-source (MIT) at:
  https://github.com/esa-tu-darmstadt/graphtoy

Embedded Systems & Applications

C++20 Coroutines: Lifetime

TECHNISCHE UNIVERSITÄT DARMSTADT

- Initial call to coroutine
  - **Creates coroutine frame**
  - Optionally starts coroutine
  - Multiple calls create multiple **independent instances** of the coroutine
- Coroutine **suspends**
  - **Saves state** to coroutine frame
  - Optionally arranges for eventual resumption (up to the implementer!)
  - **Returns to the caller**
- To **resume** a suspended coroutine…
  - **Call the coroutine_handle**
  - Returns when the coroutine suspends again

Embedded Systems & Applications

C++20 Coroutines: A Coroutine from the Outside

TECHNISCHE UNIVERSITÄT DARMSTADT

- For the caller…
  - There's **no difference between a coroutine and a regular function!**
  - When called, the coroutine will eventually return, just like any other function.
    - Calling the coroutine creates the coroutine frame.
    - The coroutine might still live, in suspended state, after it has returned!
- A suspended coroutine from the „outside"…
  - Is just a **callable object** (functor): **coroutine_handle**.
  - **Calling it resumes the coroutine**.
  - It returns when the coroutine suspends again (or terminates).

Embedded Systems & Applications