The LEGOSim User Manual

July 29, 2025

1 Introduction

1.1 Background

In recent years, multi-chiplet architectures have been widely adopted in high-performance computing (HPC) and AI chips due to their superior scalability and energy efficiency. In the context of the development of multi-chiplet systems, there is a lack of heterogeneous multi-chiplet simulators. Numerous simulators have been developed to simulate individual components/chiplets such as CPUs, GPUs, and NoIs, which unfortunately lack the flexibility to be integrated to simulate heterogenous multi-chiplet systems. For example, gem5 are designed specifically to simulate CPU or GPU. The simulation speed of gem5 is extremely slow. SimBricks can integrate multiple simulators, but its complex integration mechanism results in low simulation speed, and it cannot model inter-chiplet network transmission. ZSim can efficiently simulate large-scale systems, but it has accuracy issues in simulating multi-chiplet interconnection networks. To address these challenges and shortcomings, it is essential to develop a simulator for multi-chiplet heterogeneous integrated systems.

1.2 Features of the Simulator

LEGOSim can simulate multi-chiplet heterogeneous integrated systems. The chiplets include CPU, GPU, NPU and CiM, etc. LEGOSim has two key features:

1) adaptive and accurate time quantum, where the synchronization only occurs when there are inter-chiplet network communications.

2) non-global fencing, where only the communication chiplets/simlets are involved in synchronization instead of stalling all the simlets. This avoids global stalling and significantly reduces synchronization overhead without compromising accuracy.

2 Installation and Usage

The installation environment consists of one CPU and three GPUs, which are connected through a mesh 4*4 inter-chiplet network. The CPU and GPU configuration detail information is shown in Table 1.

Snipersim is selected to simulate CPU, and GPGPU-Sim for GPU. Matrix multiplication is used as an example. The Matmul_c executable file will load and run on the CPU, which is compiled from the matmul.cpp C++ format file. The Matmul_cu executable file will load and run on the GPU compiled from matmul.cu, which is a CUDA format file. The corresponding relationship of the running programs is shown in Figure 1.

2.1 Installing the Simulator

The installation steps are listed below:

1) Download LEGOSim code repository.

```
git clone --single-branch --branch master_v2 https://github.com/
FCAS-LAB/Chiplet_Heterogeneous_newVersion.git
cd Chiplet_Heterogeneous_newVersion/
```

Table 1: Target Chiplets Parameters

Chiplet	Properties	Values
CPU	Architecture :	x86_64
	Total Cores :	1
	Logical CPUs :	1
	Core Model :	nehalem
	Frequency(G) :	2.66
	L1d Cache :	32
	L1i Cache :	32
	L2 Cache :	256
	L3 Cache :	8192
GPU	Simd Model :	1
	Clusters :	15
	Cores PER Cluster :	1
	Num SP Units :	2
	Num SFU Units :	1
	Shader Registers :	32768
	Shmem Size :	49152
	Flush L1 Cache :	1
	Cache:dl1 :	N:32:128:4,L:L:m:N:H,S:64:8,8
	Cache:il1 :	N:4:128:4,L:R:f:N:L,S:2:32,4
	Cache:dl2 :	S:64:128:8,L:B:m:L:L,A:256:4,4:0,32
NETWORK	Bandwidth(GB/s) :	25.6
	Memory Model 1 :	bus
	Memory Model 2 :	bus

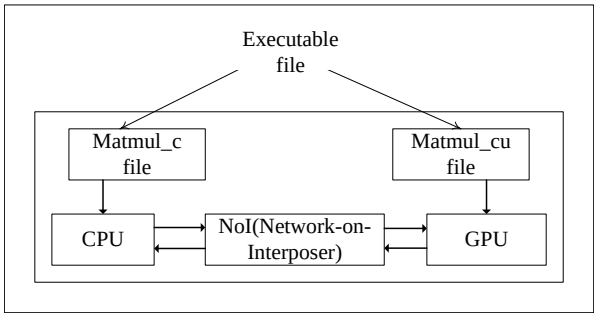


Figure 1: Mapping relationship of executable file in LEGOSim.

The following commands are executed under the root folder of the repository. The root of the repository is referenced as $SIMULATOR_ROOT

2) Initialize and update the submodule:

```
$SIMULATOR_ROOT/git submodule init
$SIMULATOR_ROOT/git submodule update
```

3) Run shell script, It will initialize the environment variables.

```
$SIMULATOR_ROOT/source setup_env.sh
```

If it runs successfully, it will occur: Setup_environment succeeded;

4) Change the code of Snipersim and GPGPU-Sim:

```
$SIMULATOR_ROOT/apply_patch.sh
```

2.1.1 Dependencies Installation

In order to install LEGOSim, the following conditions must be met for the environment:

1) GPGPU-Sim requires cuda to be installed. The new version of gpgpu-sim can support any version from cuda4 to cuda11, see GPGPU-Sim README for details.

2) GPGPU-Sim has requirements for the GCC version, and GCC7 is recommended.

2.1.2 Compiling the Simulator

Compile Snipersim, Gem5, GPGPU-Sim, popnet and inter-chiplet interconnection network.

1) Compiling Snipersim:

```
cd $SIMULATOR_ROOT/snipersim
make -j4
```

2) Compiling Gem5, which can be executed on X86 and ARM architecture:

```
cd $SIMULATOR_ROOT/gem5
scons build/X86/gem5.opt
```

or

```
cd $SIMULATOR_ROOT/gem5
scons build/ARM/gem5.opt
```

3) Compiling GPGPU-Sim:

```
cd $SIMULATOR_ROOT/gpgpu-sim
make -j4
```

4) Compiling popnet:

```
cd $SIMULATOR_ROOT/popnet_chiplet
mkdir build
cd build
cmake ..
make -j4
```

5) Compiling inter-chiplet interconnection network:

```
cd $SIMULATOR_ROOT/interchiplet
mkdir build
cd build
cmake ..
make
```

After compiling has been completed, the lib file libinterchiplet_app.a will in the interchiplet/lib directory.

2.2 Using the Simulator

2.2.1 Benchmark Compilation

This section will take matrix multiplication as an example to compile and run a benchmark.

1) Set up the environment variables.

```
$SIMULATOR_ROOT/source setup_env.sh
```

2) Compiling the source file matmul.cpp and matmul.cu, the matmul.cpp file is used to the CPU, the same as the matmul.cu to the GPU.

```
cd $SIMULATOR_ROOT/benchmark/matmul
make
```

The path $SIMULATOR_ROOT/benchmark/matmul is referenced as $BENCHMARK_ROOT

2.2.2 Test Examples

Run matrix multiplication benchmark:

1) The Matmul benchmark included four processes, one CPU process, and three GPU processes. It must be executed in the $BENCHMARK_ROOT directory.

```
$BENCHMARK_ROOT/../../interchiplet/bin/interchiplet ./matmul.yml
```

2) After the benchmark execution is finished, a group of proc_r{R}_p{P}_t{T} directory will be found, which mapped the execution about round R phase P thread T, This log files in the directory: i. GPGPU-Sim temporary file and log: gpgpusim_X_X.log. ii. Snipersim temporary file and log: sniper.log. iii. popnet log file: popnet.log.

3) The YAML file as follows:

```
# Phase 1 configuration.
phase1:
  # Process 0
  - cmd: "$BENCHMARK_ROOT/bin/matmul_cu"
    args: ["0", "1"]
    log: "gpgpusim.0.1.log"
    is_to_stdout: false
    pre_copy: "$SIMULATOR_ROOT/gpgpu-sim/configs/tested-cfgs/SM2_GTX480/*"
  # Process 1
  - cmd: "$BENCHMARK_ROOT/bin/matmul_cu"
    args: ["1", "0"]
    log: "gpgpusim.1.0.log"
    is_to_stdout: false
    pre_copy: "$SIMULATOR_ROOT/gpgpu-sim/configs/tested-cfgs/SM2_GTX480/*"
  ......

# Phase 2 configuration.
phase2:
  # Process 0
  - cmd: "$SIMULATOR_ROOT/popnet/popnet"
    args: ["-A", "2", "-c", "2", "-V", "3", "-B", "12", "-O", "12", "-F",
    "4", "-L", "1000", "-T", "10000000", "-r", "1", "-I", "../bench.txt",
    "-R", "0"]
    log: "popnet.log"
    is_to_stdout: false
```

The first level key words in the YAML configuration file is:

- phase1: Config the phase 1 simulator process.
- phase2: Config the phase 2 simulator process.

The two phases all support multiple simulator processes. The supported key words for simulator process is below:

- cmd: The simulator command.
- args: The simulator arguments.it is stored in a string array.
- log: The log file name.
- is_to_stdout: To redirect the stdout/stderr to the stdout of inter-chiplet or not.
- pre_copy: The prefix command, if some simulator need to copy environment files, It can be listed within double quotes, with spaces separating them.

3 Implementation

3.1 Overall Architecture

LEGOSim supports parallel simulation and breaks down the simulation of a multi-chiplet system into the following three components, as shown in Figure 2:

1) **Heterogeneous Chiplet Simulation Units (Simlets)**: Simlets refer to independent simulation processes used to simulate various chiplets such as CPU, GPU, NPU, CiM, etc. They can be existing open-source simulators (e.g., gem5 or Snipersim for CPU, GPGPU-Sim for GPU, MNSIM for compute-in-memory chiplets, etc.) that interact through a unified integrated interface (UII), which will be described in Section 4.1.

2) **Network-on-Interposer (NoI) Simulator**: Used for modeling the interconnection topologies of inter-chiplet network, to accurately simulate inter-chiplet communication latency.

3) **Global Manager (GM)**: Responsible for coordinating inter-chiplet data synchronization, scheduling NoI simulation, and executing synchronization strategies. The GM employs on-demand synchronization to minimize synchronization overhead while ensuring simulation accuracy. The global manager, serving as the system control, is implemented in interchiplet.cpp and cmd_handler.cpp in LEGOSim. The global manager has two responsibilities: creating new threads to launch each chiplet and handling protocol commands from different chiplets. When LEGOSim is started, the global manager initiates chiplet simulators to run as separate processes based on the configuration file (*YAML* file). This initialization is handled by the bridge_thread function in interchiplet.cpp. Functional model and timing model synchronizations are achieved through the command handling function parse_command in interchiplet.cpp. The parse_command function calls functions defined in cmd_handler.cpp, depending on the commands it receives from the chiplets.

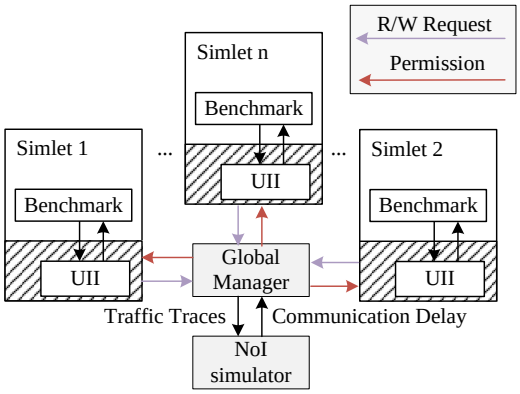


Figure 2: Overview of LEGOSim architecture and its components.

Simlets perform their respective chiplet simulation in parallel, communicate and synchronize with the GM through the UII, while the GM coordinates these simlets' synchronization and data transfers, ensuring the accuracy of the simulation. The NoI simulator simulates the communication behavior between chiplets and provides the GM with communication delay of inter-chiplet data transfer, thus enabling the GM to make correct synchronization decisions.

3.2 Workflow

An application is partitioned into multiple threads to run on each chiplet. They are compiled using compilers according to the ISA of the chiplets. APIs in Section 4.1 are used for inter-chiplet communication and data access. Each simlet runs the application threads and upon inter-chiplet communications, calling of application level APIs are captured by the simlets. Simlets then request the global manager (GM) for synchronization and data transfer. LEGOSim runs in three stages, as shown in Figure 3. In stage 1, both timing and functional models are simulated and inter-chiplet communication latency is estimated by zero load latency in NoI, and all inter-chiplet communication traffic traces are recorded. In stage 2, these traces are simulated by a separate NoI simulator to obtain accurate latency results. In stage 3, LEGOSim runs with the accurate inter-chiplet communication latencies

integrated. In this manner, inter-chiplet communication latency can be obtained separately which is used to compute clock cycles to safely advance for each simlet so as to avoid per-cycle synchronization and improve simulation efficiency. As a comparison, the previous parallel simulation schemes use per-cycle synchronization upon inter-chiplet communication to ensure temporal causality, leading to high synchronization overhead.

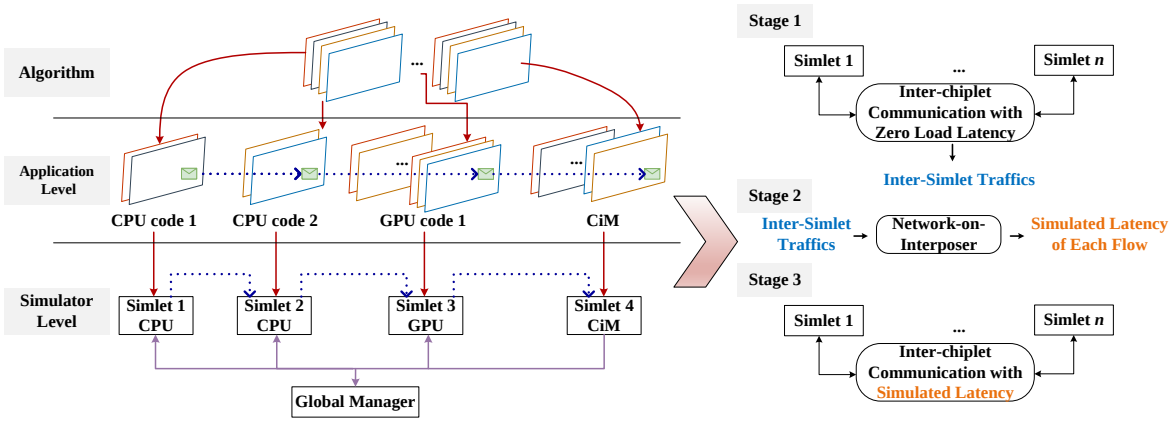


Figure 3: Workflow of the three-stage decoupled simulation of LEGOSim. Left hand side: The overall flow of an application running in LEGOSIM. Right hand side: The three-stage simulation

3.3 Communication and Synchronization

As shown in Figure 4, inter-simlet synchronization is coordinated by the GM, which operates as a centralized controller thread/process. The simulation workflow involves the following four steps:

① **Simlet Requesting**: A simlet i generates a *send/receive* or shared memory/cache chiplet access request and sends it to the GM. This request includes timing information such as the simlet's local clock cycle τ_i. Upon submission, this simlet halts local clock progression and waits for the response from the GM.

② **Request Handling by the Global Manager**: The GM handles requests as follows:

1) For *send/receive* requests, the GM matches the sender and responder simlets using a producer-consumer model to ensure ordered inter-chiplet communication. The GM calculates the next target clock cycle to be advanced for the simlet, coordinating with other active simlets to maintain consistent timing across the system.

After identifying communication pairs, the GM computes the next admissible simulation cycle for the requesting simlet to avoid timing violation. In stage 1, where accurate inter-simlet traffic delays are

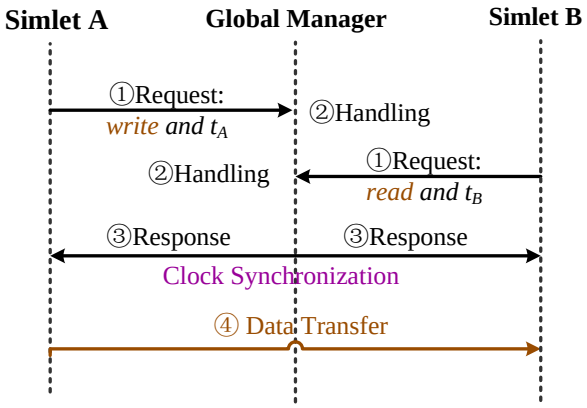


Figure 4: Workflow of inter-simlet *send/receive*. Response in Step ③ includes permission (for Step ④ data transfer) and the clock cycle to be advanced.

not known yet, the simlet's clock is advanced to the maximum of the two participating simlets' cycles plus the zero load inter-chiplet transmission latency. In stage 3, in contrast, the GM takes into account the actual latency obtained from stage 2: the sender simlet advances to the maximum cycle between the two, while the receiver simlet's advancement is offset by the corresponding NoI transmission delay.

2) For shared memory access requests, the GM identifies conflicting accesses. In stage 1, conflicts can be detected by setting a sliding time window ρ whereby each simlet j checks whether t_j falls within the range $\tau_i - \rho$ cycles, indicating a potential overlap in memory address. In stage 3, conflict information is directly derived from the recorded traces of Stage 1.

For shared memory access, a simlet i is only permitted to proceed once all other conflicting simlets j, accessing the same address with earlier simulation time (t_j ¡ t_i), have advanced to time t_i at least. This ensures strict temporal causality. The GM keeps an ordered request list for each shared memory address a as $L_a =< a, \{\alpha_i, \tau_i + l_i\} >$, where τ_i is the simlet's request timestamp, l_i is NoI transmission latency (which is zero load latency in NoI in Stage 1 and accurate NoI latency in Stage 3), and α_i represents the effective clock cycle when simlet i's request arrives at the shared memory. These requests are sorted by α_i in ascending order to ensure that earlier simulation events are processed first, even if their corresponding simlets progress more slowly in real time (wall clock time).

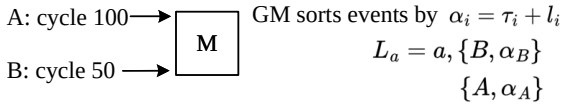
③ **Request Response**: After processing the request, the GM returns a response to the originating simlet.

1) For *send/receive* requests, this response authorizes the data transfer to proceed and specifies the clock cycle to be advanced.

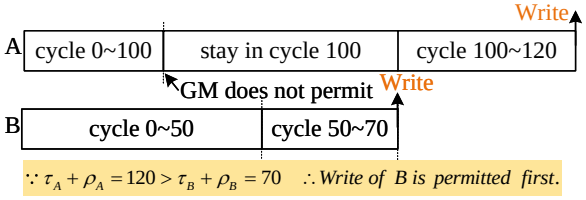
2) For shared memory access requests, the response contains: (a) a valid cycle value α_i if i is the earliest requester in L_a, granting it access to address a; (b) a zero advancement value for all other simlets in L_a, indicating they must wait until the earlier access completes. This mechanism preserves memory access order and prevents causality violations across simlets.

④ **Data Transfer Execution/Shared Memory Access**: Upon receiving the synchronization response from the GM, the simlet advances its local clock to the designated cycle and performs the requested operations, either data transfer or shared memory access.

In stage 3, discrepancies between the observed memory access order and that of stage 1 may lead to timing violations. In such cases, an optimistic execution approach can be used. The simulation uses checkpointing and rollback to resolve conflicts, ensuring that both the functional and temporal correctness are preserved. Our experiments show that such violations are rare, especially for dataflow-dominated workloads, making this approach practical and efficient. By dynamically adjusting the synchronization points based on actual communication behavior and accurate data latency, LEGOSim achieves a favorable trade-off between simulation efficiency and model accuracy.



(a) The system model where chiplets A and B write to shared memory chiplet M.



(b) Timing diagram. Assume inter-chiplet transmission latency is 20 cycles for packets from both chiplets.

Figure 5: Simlets A and B writing to shared memory chiplet M.

Figure 5 shows a case where both simlet A and B need to write to shared memory chiplet M at their respective clock cycles of 100 and 50. Although simlet A runs faster, it is prevented from advancing past cycle 100 by the GM, as B has an earlier scheduled access at cycle 50. The GM ensures

that B writes to M first, after which A is allowed to proceed, which preserves correct execution order and maintains causal consistency.

3.4 NoI Network Configuration

The inter-chiplet interconnection network plays a vital role in the multi-chiplet system in that, there are various inter-chiplet interconnection topologies and interconnection configurations that should be explored. However, network-on-chip simulators like bookism or Noxim only support the simulation of regular topologies like mesh or torus. Supporting an arbitrary topology needs to redesign the simulator. Therefore, in LEGOSim, the inter-chiplet network is designed to be configurable, that is, a connection file and a routing table file are input to the network such that it can generate the corresponding topology. The connection file is defined as follows. "$a \rightarrow b$" represents an edge connecting node "a" to node "b". For example, Figure 6 shows an example of the connection file which is written in .dot file format and the corresponding topology.

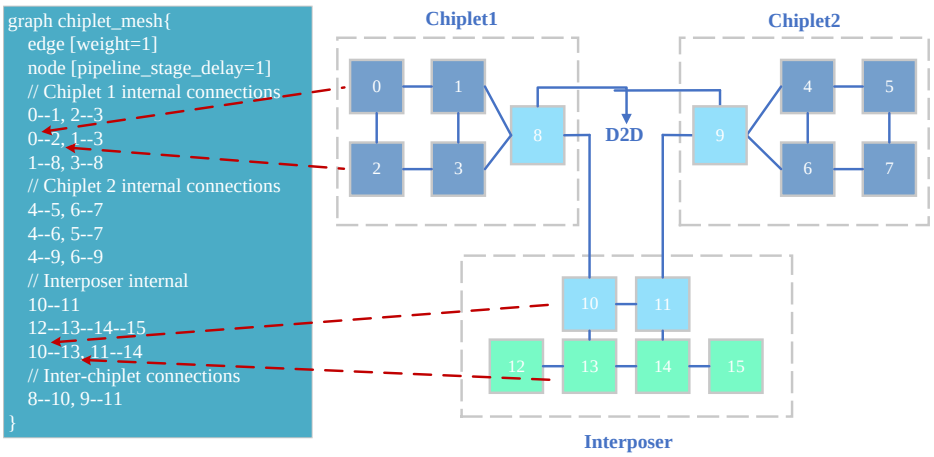


Figure 6: The example of the connection file and the corresponding topology.

The NoI simulator needs to establish a routing table and the port mapping table for each router based on the topology file. The routing table can be computed using the Floyd-Warshall algorithm, which contains the current address, destination address, and next-hop address in each item. A new function, "calRoutingTable", has been incorporated into the "sim_router.cc" file. This function is designed to enable the simulator to traverse each edge within the topology file, thereby establishing port connection relationships. These relationships are then added to the port mapping tables of the interconnected routers.

As shown in Figure 7, based on the edges related to vertex 0 in the topology graph, we can obtain the port mapping table of vertex 0, and thus connect it to vertex 1 and vertex 2 through the corresponding ports. We allow routers to have different delays by configuring the edges with varying weights.

4 Adding more simlets

4.1 Unified Integration Interface APIs

The Unified Integration Interface (UII) is a foundational component of the LEGOSim framework to support modular parallel simulation of heterogeneous multi-chiplet systems. It is designed to provide a standardized framework to integrate diverse simulators—whether they model CPUs, GPUs, DRAMs, or domain-specific accelerators (DSAs)— into a cohesive simulation platform. UII abstracts simulator-specific interfaces and harmonizes them under a unified API, and supports benchmark/application-level APIs, system call mapping, data transfer management, and clock synchronization. Figure 8 outlines its modules, which include three modules:

1) Benchmark/Application-Level APIs and System Call Definition: The UII defines a standard set of benchmark-level APIs used by chiplets for inter-chiplet communication and synchronization:

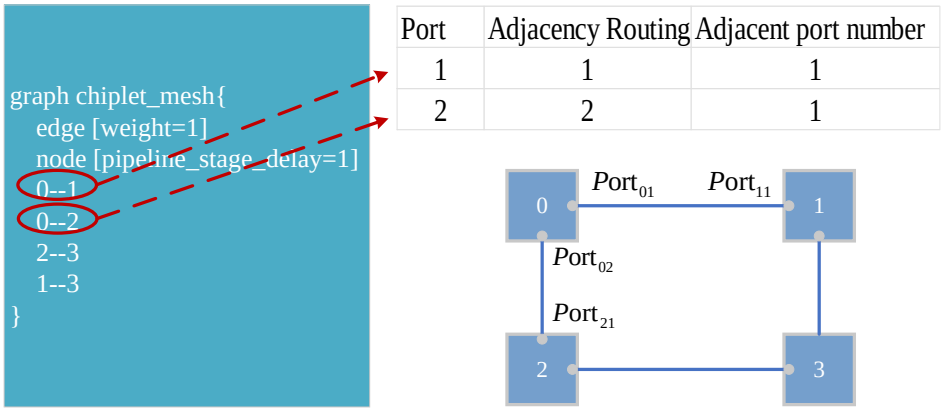


Figure 7: The router port connection table for R1.

sendMessage() and *receiveMessage*(). When integrating a new simlet, these APIs must be mapped to the internal mechanisms of the simulator as follows.

- For system-call based simulators (e.g., gem5, Sniper), these APIs are implemented as custom syscalls (e.g., *SYSCALL_REMOTE_READ* and *SYSCALL_REMOTE_WRITE*) and processed by the syscall handling routine.
- For runtime-library-based simulators (e.g., GPGPU-Sim), these APIs are mapped to existing functions (e.g., *cudaMemcpy*()).
- For DSA simulators (e.g., Scale-sim), these APIs are embedded as function calls or files within the simulation script.

Each simlets have the following application/benchmark level APIs for the programmer to call to issue inter-chiplet events.

- *sendMessage(dst_x, dst_y, src_x, src_y, addr, nbyte)* is used to send *nbyte* byte messages located at *addr* from chiplet *(src_x, src_y)* to chiplet *(dst_x, dst_y)*.
- *receiveMessage(dst_x, dst_y, src_x, src_y, addr, nbyte)* is used to receive *nbyte* byte messages from chiplet *(src_x, src_y)* to chiplet *(dst_x, dst_y)*.
- *barrier(id_list)* is used to synchronize execution among chiplets in *id_list*. All participating chiplets must reach the barrier before any of them can continue execution.
- *lock(lock_id)* is used to acquire a lock identified by *lock_id* across all chiplets. Only one chiplet can hold the lock at a time, enabling mutually exclusive access to shared resources. *unlock(lock_id)* is used to release the lock identified by *lock_id*,
- *read(dst_x, dst_y, src_x, src_y, addr, nbyte)* and *write(dst_x, dst_y, src_x, src_y, addr, nbyte)* are used to read *nbyte* bytes of data from address *addr* or write *nbyte* bytes to *addr* on chiplet *(src_x, src_y)*.

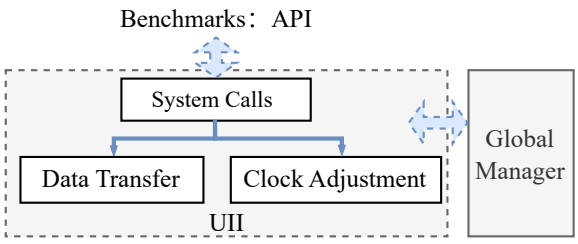


Figure 8: Modules of the UII.

2) Data Transfer Implementation: Data transfer between chiplets in the UII is managed by functions such as *sendSync*(), *receiveSync*(), *write_data*(), and *read_data*(). These functions coordinate data transfer protocols with the GM and enable data transmission through dedicated channels as follows.

- For CPU simulators (e.g., gem5, Sniper) , *sendMessage*() / *receiveMessage*() are translated to be inter-simlet data transmission in the syscall handling routines by file exchange, pipes, or shared memory in the host machine. Data is transferred from and to this simlet's internal simulated memory.
- For GPU simulators (e.g., GPGPU-Sim), additional memory copy operations (e.g., *cudaMemcpy*()) are inserted before/after calling *sendSync*() and *receiveSync*() to move data between this simlet and others. These wrappers ensure that the GPU's memory space remains consistent with LEGOSim's global model.
- For DSA simulators (e.g., Scale-sim), UII writes inputs to an interface file, executes the DSA script, then reads output. *sendMessage*() is implemented by writing input data to this file, or passing arguments to the Python configuration function for the simlet. *receiveMessage*() reads output data after simulation completes.

3) Clock Control: Given the diversity of simulation timing models, UII supports a flexible synchronization model to ensure that heterogeneous simlets advance their respective local clock tick correctly as follows.

- For cycle-accurate simulators (e.g., gem5, GPGPU-Sim), their clock cycles are controlled. For example, gem5 uses an event-driven model of clock tick granularity, and synchronization is managed by controlling *tick*. In GPGPU-Sim, simulation progress is tracked using *gpu_sim_cycle* and *gpu_tot_sim_cycle*. Clock ticking is controlled by these variables in such simulators.
- For non-cycle-driven simulators (e.g., Sniper), UII inserts pseudo operations to artificially delay execution, such as *Sleep*() to adjust the clock delay according to the synchronization events.
- For DSA simulators (e.g., Scale-Sim), which have no native clock or with simplified execution timeline: A block of operations/computations is performed to obtain the execution time, which is reported to the GM for synchronization.

4.2 UII examples

Below are examples of how it facilitates integration in Sniper , GPGPU-Sim and Scale-Sim:

Integration of Sniper. Sniper, a CPU simulator, required additional adaptation due to its non-cycle-driven execution. Custom system calls are defined (*SYSCALL_REMOTE_READ* and *SYSCALL_REMOTE_WRITE*) to map Sniper's remote read/write operations to UII's *sendMessage*() and *receiveMessage*() functions. In the functional model, these system call handling routine translates *receiveSync*(), *read_data*(), *sendSync*(), and *write_data*() into inter-simlet message passing. In the timing model, *readSync*() and *writeSync*() are used for synchronization. However, since Sniper does not advance by discrete clock cycles, a *Sleep*() function is inserted to adjust its execution timing according to the target clock cycles to be advanced, ensuring accurate synchronization.

Integration of GPGPU-Sim. GPGPU-Sim is used to simulate NVIDIA GPU architectures and relies on the CUDA runtime environment. Within the LEGOSim framework, its *sendMessage*() and *receiveMessage*() functions are mapped to CUDA *cudaMemcpy*(), facilitating data transfer between this simlet and others. In terms of timing synchronization, GPGPU-Sim records local clock by *gpu_sim_cycle* and *gpu_tot_sim_cycle* and updates them according to the target clock cycles to be advanced.

Integration of Scale-Sim. Scale-Sim is integrated into LEGOsim as simlet through executing the corresponding python script with designated chiplet identifiers, topology, the workload of NPU as input parameters. As Scale-Sim has only timing model, the functional model is implemented in a dedicated C++ model. It receives input through *receiveMessage*() and transmits output via *sendMessage*() as wrappers. Upon completion of the simulation, the wrapper proceeds reading the execution time from Scale-Sim's output logs. This execution time will be added to the time get from *readSync*() and sent to other chiplets through *writeSync*(). Data is received using *receiveSync*() and *read_data*() and sent using *sendSync*() and *write_data*().

By standardizing APIs, inter-chiplet communication data management, and clock synchronization, the UII enables seamless interoperability between diverse simlets, reducing integration complexity.

5 Example

In this section, matrix multiplication is used as an example. Matrix multiplication is one of the basic operations in neural networks and has a wide range of applications. This chapter will take matrix multiplication as an example to illustrate the programming model, load mapping and precautions in the process of designing multi-chiplet programs that can run this simulator.

5.1 Analyzing how to parallelize the tasks of the analysis program

The first step in designing a program is to analyze how its tasks can be parallelized. In systems composed of one CPU and multiple GPUs, the CPU typically handles task distribution and data allocation, as well as receiving computational results from GPUs. GPU chiplets are responsible for performing most computations, which should be optimized through parallelization to align with their architectural characteristics. Parallelization involves two levels: how to parallelize the program for multi-chiplet system operation, and how to distribute computational load across each GPU chiplet.

5.1.1 How to parallelize the program for multi-chiplet system operation

According to the architecture characteristics of GPU, we need to divide a program into many small threads with the same operation, and these threads are preferably free of data dependencies in time sequence. Take the analysis of matrix multiplication as an example:

- System architecture: One CPU chiplet (responsible for task distribution and result aggregation) and four GPU chiplets (responsible for matrix calculation).
- Input matrix: Input matrix Z with size of $m \times n$ and matrix W with size of $n \times k$.
- Output matrix: Result matrix E.
- The operation process of matrix multiplication is listed in Figure. 9.

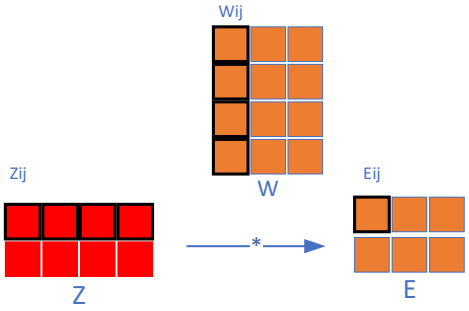


Figure 9: The calculation process of matrix multiplication.

The operation process has two key characteristics: First, for the result matrix E, the values at position (i,j) are not dependent on data in other positions such as $(i+1,j)$. Second, the calculation process remains identical for each element in the result matrix. Therefore, we can determine the number of threads based on the size of result matrix E, and map the task of calculating each value to a specific thread. This results in $m \times k$ threads, with each thread's responsibility being to compute a single value within the result matrix.

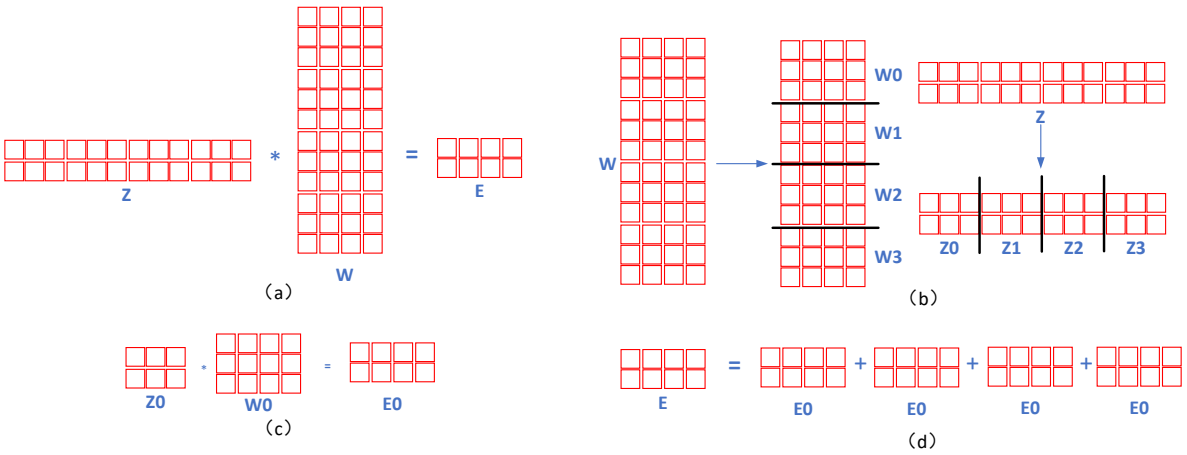


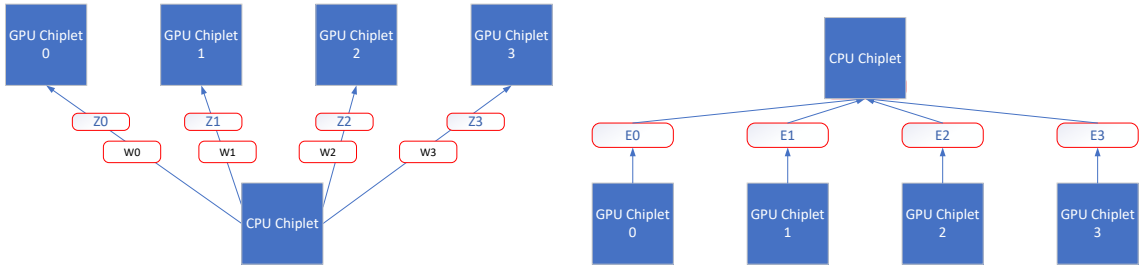
Figure 10: (a) The original matrix multiplication is not decomposed, and the data scale is large; (b) Cutting the input matrix; (c) After decomposition, the smaller matrices are multiplied.

5.1.2 How to distribute computational load across each GPU chiplet

There are two primary approaches for decomposition: scope decomposition and functional decomposition.

- Scope decomposition involves breaking down the required data into manageable chunks, where different chiplets process distinct data using the same program. This method is typically employed when handling massive datasets.
- Functional decomposition divides the problem into multiple tasks, each performing specific operations on available data. The chiplets executing these tasks can operate like a pipeline system.

Through analyzing matrix multiplication, we observe the following characteristics: As shown in Figure. 10, when dealing with two large metrices, the method illustrated in Figure. 10(b) can be applied to divide them into several smaller segments. By calculating and summing the products of these smaller segments. By calculating and summing the products of these smaller matrices, the final result matches that obtained by directly multiplying the original matrices.



(a) The CPU chiplet distributes data to the GPU chiplets

(b) The GPU chiplet completes the computing task and feeds back the results to the CPU core

Figure 11: The simulation process of matrix multiplication.

In conclusion, matrix multiplication is a good fit for programs that are decomposed by range. Figure. 11 shows the mapping work flow for matrix multiplication. In this strategy, We map the smaller matrix multiplication tasks after calculation division to different GPU chiplets and finally add their results to gather at the CPU chiplet.

5.2 Programming model

We categorize the code required for each chiplet's tasks into code segments. A complete program should consist of one or multiple code segments that handle most computational tasks. The programming model comprises two key components: 1) how to structure code segments in each chiplet, and 2) how to organize code segments between different chiplets.

5.2.1 Internal code segment construction

The work flow of an internal code segment is listed in Figure. 12

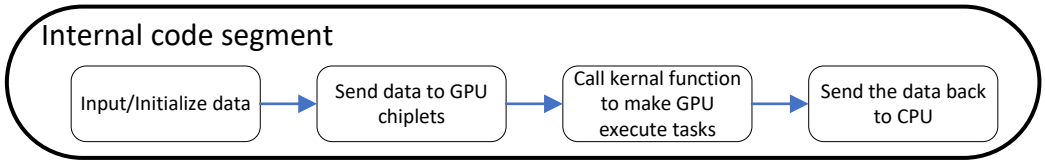


Figure 12: Internal code segment work flow.

1) Input/Initialize data

The primary task in the data loading phase is to prepare input data, which involves declaring, initializing, and partitioning data within the CPU's memory space. In the example matrix multiplication program, we skip the original matrix partitioning process during data loading. Instead, we directly initialize a smaller partitioned input matrix using random numbers.

```
// Declare data
int64_t *A = (int64_t *)malloc(sizeof(int64_t) * Row * Col);
int64_t *B = (int64_t *)malloc(sizeof(int64_t) * Row * Col);
int64_t *C1 = (int64_t *)malloc(sizeof(int64_t) * Col);
int64_t *C2 = (int64_t *)malloc(sizeof(int64_t) * Col);
int64_t *C3 = (int64_t *)malloc(sizeof(int64_t) * Col);
//Initialize data
for (int i = 0; i < Row * Col; i++) {
    A[i] = rand() % 51;
    B[i] = rand() % 51;
}
```

2) Send data to GPU chiplets

In this part, the major work is to send data from CPU chiplet to GPU chiplets through *sendMessage*() and *receiveMessage*() API.

```
// CPU chiplet side
InterChiplet::sendMessage(0, 1, idX, idY, A, 10000 * sizeof(int64_t));
InterChiplet::sendMessage(1, 0, idX, idY, A, 10000 * sizeof(int64_t));
InterChiplet::sendMessage(1, 1, idX, idY, A, 10000 * sizeof(int64_t));

InterChiplet::sendMessage(0, 1, idX, idY, B, 10000 * sizeof(int64_t));
InterChiplet::sendMessage(1, 0, idX, idY, B, 10000 * sizeof(int64_t));
InterChiplet::sendMessage(1, 1, idX, idY, B, 10000 * sizeof(int64_t));

//GPU chiplet side
int64_t *d_dataA, *d_dataB, *d_dataC;
cudaMalloc((void**)&d_dataA, sizeof(int64_t) * Row * Col);
cudaMalloc((void**)&d_dataB, sizeof(int64_t) * Row * Col);
cudaMalloc((void**)&d_dataC, sizeof(int64_t) * Col);
```

```
receiveMessage(idX, idY, 0, 0, d_dataA, sizeof(int64_t) * Row * Col);
receiveMessage(idX, idY, 0, 0, d_dataB, sizeof(int64_t) * Row * Col);
```

3) Call kernal function to make GPU execute tasks

This part is the core part of program design. The kernel function refers to the execution content of each thread in GPU runtime. The design of kernel function mainly depends on the analysis results of "how to parallelize the program to make it suitable for GPU operation" in the process of program analysis.

According to the Section 5.1.1, the kernal function is prepared as follows:

```
__global__ void matrix_mul_gpu(int64_t* M, int64_t* N,
                               int64_t* P, int width) {
    int sumNum = threadIdx.x + threadIdx.y * 10;
    int i = threadIdx.x;
    int j = threadIdx.y;
    int64_t sum = 0;
    for (int k = 0; k < width; k++) {
        int64_t a = M[j * width + k];
        int64_t b = N[k * width + i];
        sum += a * b;
    }
    P[sumNum] = sum;
}
```

4) Send the data back to CPU

The data results mainly include two types of results: transferring the results to CPU memory and sending the calculated results to other chiplets. The following example program shows how this part of the code should be written.

```
//GPU chiplets side
sendMessage(0, 0, idX, idY, d_dataC, 100 * sizeof(int64_t));

//CPU chiplets side
InterChiplet::receiveMessage(idX, idY, 0, 1, C1, 100 * sizeof(int64_t));
InterChiplet::receiveMessage(idX, idY, 1, 0, C2, 100 * sizeof(int64_t));
InterChiplet::receiveMessage(idX, idY, 1, 1, C3, 100 * sizeof(int64_t));

for (int i = 0; i < 100; i++) {
    C1[i] += C2[i];
    C1[i] += C3[i];
}
```

5.2.2 Organize code segments between different chiplets

Completing the code segment means that we have programmed the tasks to be performed the tasks to be performed by a single core. Based on the analysis results obtained in Section 5.1.2, we can map the code segments to different cores using the following methods.

1) Construct the configuration files.

In the configuration file, we need to define the configuration of each chiplet and the mapping relationships between simlets and chiplets. An 1CPU-3GPU chiplet system is configured as follows:

```
# Phase 1 configuration.
phase1:
  # Process 0
  - cmd: "$BENCHMARK_ROOT/bin/matmul_cu"
    args: ["0", "1"]
    log: "gpgpusim.0.1.log"
    is_to_stdout: false
    clock_rate: 1
```

```
    pre_copy: "$SIMULATOR_ROOT/gpgpu-sim/configs/tested-cfgs/SM7_TITANV/*"
  # Process 1
  - cmd: "$BENCHMARK_ROOT/bin/matmul_cu"
    args: ["1", "0"]
    log: "gpgpusim.1.0.log"
    is_to_stdout: false
    clock_rate: 1
    pre_copy: "$SIMULATOR_ROOT/gpgpu-sim/configs/tested-cfgs/SM7_TITANV/*"
  # Process 2
  - cmd: "$BENCHMARK_ROOT/bin/matmul_cu"
    args: ["1", "1"]
    log: "gpgpusim.1.1.log"
    is_to_stdout: false
    clock_rate: 1
    pre_copy: "$SIMULATOR_ROOT/gpgpu-sim/configs/tested-cfgs/SM7_TITANV/*"
  # Process 3
  - cmd: "$SIMULATOR_ROOT/snipersim/run-sniper"
    args: ["--", "$BENCHMARK_ROOT/bin/matmul_c", "0", "0"]
    log: "sniper.0.0.log"
    is_to_stdout: false
    clock_rate: 1

# Phase 2 configuration.
phase2:
  # Process 0
  - cmd: "$SIMULATOR_ROOT/popnet_chiplet/build/popnet"
    args: ["-A", "100", "-c", "1", "-V", "3", "-B", "12", "-O", "12",
    "-F", "2", "-L", "1000", "-T", "1000000000", "-r", "1", "-I",
    "../bench.txt", "-R", "4", "-G", "test/mesh_6_6.gv", "-D",
    "../delayInfo.txt", "-P"]
    log: "popnet_0.log"
    is_to_stdout: false
    clock_rate: 1
```

2) Receiving coordinates of each chiplet in the code segments.

For each simlet, it will run all the contents of our code file in full. Therefore, we set the chiplet number as a parameter required by the startup program, so that the emulator process can determine to execute based on its chiplet number.

```
int idX, idY;

int main(int argc, char **argv) {
    idX = atoi(argv[1]);
    idY = atoi(argv[2]);
```