# Description

Title of Invention: A Distributed Parallel Device and Method for Multi-AI Processors Based on a High-Speed Interconnect Bus

# Technical Field

[0001] The present invention relates to a distributed multi-AI processor parallel architecture based on a high-speed interconnect bus and a multi-core load-balanced scheduling method. By leveraging multiple AI processors in parallel to infer a single AI network, the invention effectively addresses the computational power bottleneck caused by heat dissipation limitations of single AI processors, which is constrained by semiconductor manufacturing processes. It avoids the use of expensive HBM technology and server-level cooling solutions, and instead utilizes multiple AI processor cores in a cascaded manner to achieve a near-linear expansion of the overall system's memory bandwidth and computational scale, enabling large AI models to be deployed in edge-side or consumer terminal domains.

# Background Art

[0002] Currently, mainstream LLM/VLM applications are still primarily based on cloud computing power. However, cloud computing faces issues such as large network transmission latency and high costs, making edge-side AI inference a practical need. A single core's effective computational power has an upper limit due to several factors: limitations in chip manufacturing processes, the inability of edge devices to handle server-level cooling costs, and the memory bandwidth and capacity constraints of single-core, large-bandwidth memory technology. For example, with a 12nm chip manufacturing process, the computational power of a single general-purpose AI inference core is typically limited to within 16 TOPs. To support real-time inference for larger AI LLM/VLM models on the edge, we need to leverage distributed AI inference technology to distribute the massive computations required for single-word inference of large models onto multiple cascaded AI inference chips. This approach disperses the computational load, memory access bandwidth requirements, and heat dissipation issues across multiple AI inference chips, allowing for real-time edge-side computing of LLM/VLM using consumer-grade memory costs and cooling structures.

[0003] However, communication between multiple AI processors faces the following challenges:

1. The communication bus between AI processors has an order of magnitude lower transmission bandwidth and an order of magnitude higher transmission latency compared to the on-chip bus within a System-on-Chip (SoC). In addition, inter-core data exchange often requires inefficient access to off-chip memory, which consumes additional memory bandwidth. Therefore, it is necessary to reduce the transmission startup latency on the inter-core connection bus and minimize unnecessary read/write access to external memory related to transmission, thereby reducing the additional memory bandwidth consumed by inter-core transmission.
2. To improve the utilization of expensive on-die high-speed memory, the scheduler of a single-core AI processor often adopts a Depth-First Search (DFS) scheduling strategy to reduce the frequency of data exchange between compute core via off-chip memory. This can lead to fine-grained out-of-order execution between layers of the AI network. When an AI network layer is partitioned for execution across multiple SoC compute cores, the computational tasks may not be perfectly balanced. This means the fine-grained DFS scheduling on each core cannot be uniform. However, when multiple cores are inferring the same AI network, we must ensure that for operations with data dependencies, the producer on core A completes its computation earlier than the consumer on core B starts, to avoid

wasting core B's computational power while waiting for core A's result. Therefore, the scheduler must not only optimize external memory bandwidth by using DFS but also guarantee the execution order of all producer/consumer pairs, ensuring the producer completes before the consumer starts. Achieving both internal DFS for all cores and the ordered execution of producer completion and consumer start times is an NP-hard problem.

# Summary of the Invention

In this document, the terms **"compute core"** and **"compute node"** refer to distinct levels of the computing system. For clarity, they are defined as follows:

- **Compute Core**: A dedicated AI computation module or unit implemented on a chip.
- **Compute Node**: An integrated unit comprising an AI circuit, its peripheral support circuitry, and external memory, functioning as a single, organized computing entity.

[0004] To address the above challenges, the present invention provides a scalable computational architecture. This architecture extends processing capability either by deploying multiple compute nodes or by integrating multiple compute cores within a single semiconductor die interconnected via a high-speed internal bus (e.g., AXI). Managed by a compiler-driven scheduling strategy, the architecture distributes and balances compute tasks across nodes or cores while accommodating cross-node/core communication latency. It increases both computational power and memory bandwidth through node/core cascading. Furthermore, by combining dedicated hardware circuits with compiler support, the invention enables multiple on-die high-speed memories (such as SRAM or Cube Memory) to communicate and synchronize data directly via the cascaded hardware bus, thereby significantly reducing off-chip memory access traditionally caused by inter-chip data transfers.

[0005] Another objective of the present invention is to provide a scheduling strategy and compiler implementation method that reconciles two inherently conflicting requirements:

- **Parallelism Across Nodes and Cores:** Maximizing throughput by distributing AI computation tasks across multiple compute cores, which may themselves be deployed across multiple compute nodes, while strictly preserving the timing and execution order of data-dependent operations between cores. This approach produces a wheeling-level scheduling strategy that coordinates task execution across cores and nodes and balances the task load of each execution layer.
- **Depth-First Scheduling Within Each Core:** Applying a Depth-First Search (DFS)–based scheduling strategy inside each compute core to enhance data reuse within limited on-die high-speed memory and to minimize the frequency of external memory accesses. This approach results in a DFS-style scheduling policy within each core.

This approach enables asymmetric AI execution tasks to comply with the local operator DFS order while simultaneously meeting the execution timing constraints imposed by inter-core data dependencies.

# Device Composition and Functionality

### System Overview

[0006] The distributed AI inference system described in the present invention consists of multiple compute cores. These compute cores can be implemented in two main forms:

1) Multi-Die Interconnect: Each compute core is an independent semiconductor die, interconnected by a high-speed inter-die communication bus (such as UCIe, PCIe).
2) Single-Die Integration: Multiple compute cores are integrated within a single semiconductor die, interconnected by an efficient on-chip bus (such as AXI).

[0007] The key design focus of each compute core is establishing a direct data path between the on-chip high-speed memories of each core using a Direct Memory Access (DMA) engine and an inter-core synchronization controller. This design provides a solid hardware foundation for the efficient scheduling algorithms at the software layer.

**Embodiment 1: Multi-Die Interconnect System Structure (as shown in Figure 1)**

[0008] Figure 1 shows an implementation where two independent compute nodes (Dies) are cascaded via an inter-die communication bus (such as PCIe or UCIe). The system consists of two identical compute nodes, each composed of the following core components:

- Compute Chip (110)
- External Memory (170)

[0009] Instances of the two compute chips (110) are connected via the inter-die communication bus (180).

[0010] Internal Module Composition of Compute Chip (110):

- AI Processor Module (IPU) (120), the compute core inside compute chip 110.
- Main CPU Module (140)
- Memory Controller (150)
- Inter-die Cascaded High-Speed Bus Controller (160)
- Internal Bus Module (130): Used to connect modules 120, 140, 150, 160, and others.

[0011] Internal Sub-Module Composition of AI Processor Module (120):

- Data Transfer Module (DMA Engine) (121)
- On-die High-Speed Memory Module (122)
- Compute Unit (123)
- Inter-core Synchronization Module (124)
- Controller Module (125)

[0012] Content of External Memory (170):

- Main CPU Instructions and Data (171): Required for the Main CPU Module (140) to operate.
- AI Instructions (172): For the Controller Module (125) within the AI Processor Module (120) to execute.
- AI Weights and Activation Data (173): Model weights and activation data required for AI processing.

[0013] Main Module Functions and Workflow

1. **Function of Main CPU Module (140)**: The Main CPU Module (140) executes high-level control tasks by running instructions and data stored in (171), including: a. **Data Preparation**: Prepares the required AI instructions (172) and AI data (173) for the AI

Processor (120). b. **Initiate AI Processing**: Starts the AI Processor (120) to begin executing the AI instructions (172).

2. **Function of AI Processor Module (120)**: After being started by the Main CPU, its internal Controller Module (125) begins executing AI instructions (172) and coordinates its sub-modules to complete the following specific tasks: a. **Data Loading**: Controls the DMA Engine (121) to efficiently read AI weights and activation data from the (173) region of the External Memory (170) into the On-die High-Speed Memory Module (122). This allows the compute cores to perform multiple high-speed read/write accesses, thereby reducing wait times and the frequency of access to external memory. b. **AI Computation**: Controls the AI Compute unit (123) to access data in the On-die High-Speed Memory Module (122) to perform AI computation tasks. c. **Result Write-back**: Controls the DMA Engine (121) to write the computation results of the AI Processor Module (120) back to the (173) region of the External Memory (170). d. **Inter-node Data Exchange**: Controls the DMA Engine (121) to efficiently exchange intermediate computation results with other compute cores via the Inter-die Communication Bus (180). e. **Multi-node Synchronization**: Controls the Inter-core Synchronization Module (124) to send synchronization signals (fences) to other compute cores, or to wait for and receive synchronization signals from other cores, to achieve precise data and timing synchronization between multiple compute cores.

**Embodiment 2: Single-Die Integrated System Structure (as shown in Figure 2)**

[0014] Figure 2 shows another embodiment where multiple compute cores 201, 202, 203, 204 are integrated within a single semiconductor die, interconnected by an on-chip bus (such as AXI). The internal structure of compute cores 201, 202, 203, 204 is consistent with the compute core in Embodiment 1. This structure is particularly suitable for scenarios using Cube Memory technology. In this scenario, multiple compute cores are distributed across different areas of the same large die, with each node having its dedicated local Cube Memory bank serving as its on-die high-speed memory. In this embodiment, the communication bus (205) is implemented as an on-chip bus (such as an AXI bus).

**Description of Drawings and Definitions of Terms**

[0015] Fig 1 shows the structure diagram of two die instances cascaded via an inter-die communication bus, as well as the internal block diagram of the die, in one embodiment of the present invention.

[0016] Fig 2 shows the structure diagram of multiple compute cores connected via an on-chip bus within a single die, in another embodiment of the present invention.

[0017] Fig 3 shows the flow of the software compiler's process for disassembling an AI network for computation on multiple compute cores, in one embodiment of the present invention.

[0018] Fig 4 shows an example flow of the software compiler's process for tiling an AI network, in one embodiment of the present invention.

[0019] Fig 5 an example flow of the software compiler's process for distributing the tiled AI network to multiple compute cores and identifying data that needs to be transmitted across nodes, in one embodiment of the present invention.

[0020] Fig 6 shows an example flow of the software compiler's process for inserting SendOps and RecvOps node pairs into the cross-node data transfer process, in one embodiment of the present invention.

[0021] Fig 7 shows the flow of the software compiler's process for compiling and generating instructions for the SendOps and RecvOps node pairs, in one embodiment of the present invention.

| | | |
|---|---|---|
| [0022] | 100 | The two AI processor cascading devices. |
| [0023] | 110 | Computational chip. |
| [0024] | 120 | The AI processor module. |
| [0025] | 121 | The DMA module |
| [0026] | 122 | The local high-speed storage module. |
| [0027] | 123 | The computation unit. |
| [0028] | 124 | The inter-core synchronization module. |
| [0029] | 125 | The controller unit. |
| [0030] | 130 | The internal bus |
| [0031] | 140 | The CPU module. |
| [0032] | 150 | The memory controller Module. |
| [0033] | 160 | The inter-core interconnect interface module. |
| [0034] | 170 | The external memory. |
| [0035] | 171 | The instructions and data for CPU. |
| [0036] | 172 | The AI instruction area |
| [0037] | 173 | The AI weights and activity data area |
| [0038] | 180 | High-speed communication bus between compute nodes. |
| [0039] | 201～204 | Four compute cores within a single die. |
| [0040] | 300 | Flow of the software compiler's process for disassembling an AI network for computation on multiple compute cores. |
| [0041] | 400 | Example diagram of partitioning an AI network into software tiles. |
| [0042] | 500 | Example diagram of assigning software tiles to different compute cores. |
| [0043] | 600 | Example diagram of establishing SendOps/RecvOps between software tiles across compute cores for inter-node data dependencies. |
| [0044] | 700 | Compilation method 700 for SendOps and RecvOps. |
| [0045] | S301~304 | Steps of software compiler for partitioning AI networks across multiple compute cores and inserting synchronization instructions. |
| [0046] | S701~706 | Steps of Software Compiler Handling SendOps and RecvOps. |
| [0047] | DFS | Depth-First Search |
| [0048] | MILP | Mixed-Integer Linear Programming |
| [0049] | Fence | A **fence** is a synchronization or ordering mechanism used in computer systems to ensure that certain memory operations or instructions complete before others begin. |

| | | |
|---|---|---|
| [0050] | DMA | DMA enables fast, efficient, CPU-offloaded data movement between peripherals and memory. |
| [0051] | IPU | Intelligent Processing Unit, nickname for AI processor circuit |
| [0052] | AI | Artificial Intelligence |
| [0053] | Software tile | A software tile refers to a subgraph or partition of a computational graph (e.g., an AI network) that is treated as a unit of computation by a compiler or runtime system. |
| [0054] | CPLEX | IBM ILOG CPLEX Optimizer |
| [0055] | Gurobi | Gurobi Optimizer |
| [0056] | SRAM | Static Random-Access Memory |
| [0057] | Cube Memory | Cube Memory is a stacked memory technology |
| [0058] | Chiplet | Chiplets are small, modular chips that are combined using advanced packaging to build a larger, more powerful system |
| [0059] | Big-M method | The big-M algorithm is a technique for solving linear programming (LP) problems. It's especially useful for models containing "greater than or equal to" or equality constraints, or even "less than or equal to" constraints, allowing these constraints to be represented in a linear programming format. |

## Specific Embodiments

[0060] The structural and operational principles of the present invention will be described in detail below with reference to the accompanying drawings.

[0061] The objective of the present invention is to provide a distributed multi-AI processor parallel architecture and a multi-core balanced scheduling method based on a high-speed interconnect bus. By interconnecting multiple AI processors via a high-speed bus, the invention enables efficient parallel inference for a single AI network, thereby solving the problem where a single AI processor struggles to meet the demands for computational power, storage capacity, and storage bandwidth in large model inference.

[0062] Referring to FIG. 1, FIG. 1 shows a block diagram of two **AI processor cascading devices** 100 in one embodiment of the present invention. The two AI processor cascading devices 100 perform data exchange and transmit synchronous fence instructions via a **high-speed cascading bus** to collectively execute an inference task for the same artificial intelligence model.
The following technical effects can be achieved through the aforementioned cascading method:

- **Computational Power and Storage Expansion**: By performing task division and data sharing between two AI processors, the pressure on a single processor in terms of computational power and die-internal high-speed memory bandwidth can be effectively reduced. This allows the total external storage bandwidth and computational power of the entire cascaded system to increase approximately linearly.
- **Avoidance of Process Limitations**: Since the computational power of a single core is limited by semiconductor manufacturing processes, there exists a physical upper limit to its computational capability. The present invention, through a multi-processor cascading method, effectively breaks through this limitation to achieve higher-scale parallel inference performance.
- **Cost Advantages and Application Deployment**: The present invention does not rely on high-cost **HBM memory** or advanced packaging processes. Instead, it achieves large computational power requirements through a cascaded architecture of multiple AI processor nodes. This not only reduces the concentration of computational power on a single chip but also avoids the use of server-grade cooling systems, enabling the deployment and application of generative artificial intelligence large models on **edge computing devices** and **consumer-grade terminal products**.

[0063] In one embodiment, the AI processor cascading device 100 includes two **compute nodes** (compute node 1 and compute node 2). The two compute nodes are structurally identical and are connected via a **high-speed interconnect bus** 180. In this embodiment, the interconnect bus 180 is a Peripheral Component Interconnect Express (PCIe) bus. Since computer node 1 and computer node 2 have the same structure, the following description will use computer node1 as an example. Computer node1 consists of a **computational chip** 110 and an **external storage module** 170. The computational chip 110 primarily includes the following internal units: an **AI processor module** 120, a **CPU module** 140, an **external storage access controller** 150, an **inter-core cascading high-speed bus interface controller** 160, and an **on-chip bus** 130 used to interconnect the various units.

[0064] In one embodiment, the external storage module 170 is used to store, but is not limited to: AI network weight parameters, intermediate results of AI computation, AI instruction sets, and the operating system space required for the CPU module 140 to run. The external storage 170 preferably uses high-speed memory, such as Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM). Its stored content includes: instructions and data 171 required for

the CPU module 140, AI instructions 172 required for the AI processor module 120, and the weights and activity data 173 of the AI network.

[0065] In one embodiment, a **local high-speed storage module** 122 inside the AI processor module 120 is used to cache AI network weights, intermediate AI computation results, and AI instructions. This reduces the frequency of access to the external memory 170 and shortens the access latency during computation. The local high-speed storage module 122 preferably adopts a low-latency, high-bandwidth memory structure, such as Static Random Access Memory (SRAM), Cube Memory, or other memory types with high bandwidth and low latency characteristics. The present invention is not limited in this regard.

[0066] In one embodiment, the interconnect bus 180 between compute nodes must satisfy the characteristics of low latency and high bandwidth. Its implementation methods may include, but are not limited to, a PCIe bus, a Universal Chiplet Interconnect Express (UCIe) bus, etc. In a single-die, multi-computing-node solution adopting a Cube Memory structure, the interconnection can also be achieved through an on-chip interconnect bus (e.g., AXI).

[0067] In one embodiment, the CPU module 140 accesses the instructions and data 171 in the external memory 170 through the external storage controller 150 and schedules the AI processor module 120 to perform computation tasks. A **controller unit** 125 inside the AI processor module 120 is responsible for parsing and executing the instructions in the AI instruction area 172 and controlling a **DMA module** 121 to move data between the local high-speed storage module 122 and the weights and activity data area 173 of the external memory 170. The purpose of pre-moving the weights and activity data to the local high-speed storage module 122 is to concentrate multiple data accesses of a **computation unit** 123 on the low-latency die-internal high-speed memory, thereby significantly reducing the amount of data access to the external memory 170.

[0068] In one embodiment, when the scheduling algorithm of the software compiler determines that the current computation result needs to be transferred to a remote computing node, the AI instructions generated by the compiler will cause the controller unit 125 to schedule the DMA module 121 to perform a data movement operation. The data can be stored in the local high-speed storage module 122 or the external memory 170. The DMA module 121 is connected to the inter-core interconnect interface module 160 via the internal bus 130 and transfers data to the local high-speed storage module 122 or the external memory 170 of the target computing node via the interconnect bus 180. The selection of the source and destination for data transfer is determined by the memory scheduling strategy of the software compiler.

[0069] In one embodiment, the controller unit 125 can control the **inter-core synchronization module** 124 of the AI processor module 120. When it is necessary to wait for a synchronous fence signal from another computing node, the controller unit 125 can pause the execution of local AI instructions until the corresponding fence signal is received before resuming operation. At the same time, the controller unit 125 can also send a fence signal to the inter-core synchronization module 124 of other compute nodes via the internal bus 130, the inter-core interconnect interface module 160, and the interconnect bus 180, to achieve execution synchronization and data consistency control among multiple nodes.

[0070] Please also refer to FIG. 2. FIG. 2 shows another embodiment of the present invention. In this embodiment, based on a Cube Memory chiplet solution, multiple compute cores are integrated within a single logical chip die. Each computing node is interconnected via a die-internal **Advanced eXtensible Interface (AXI)** bus. Specifically, FIG. 2 shows four compute nodes 201, 202, 203, and 204, which are interconnected via an AXI ring bus. Each computing node is an instance of the computing node structure shown in FIG. 1 and is independently configured with its local Cube Memory storage partition as the die-internal high-speed memory resource for that node.

[0071] Please also refer to FIG. 3, FIG. 4, FIG. 5, FIG. 6, and FIG. 7, which illustrate the process in one embodiment of the present invention where an AI software compiler breaks down the computation task of an AI network into multiple sub-networks, allocates them to multiple compute cores for execution, and inserts data transfer instructions and synchronization instructions between the compute cores. FIG. 3 shows a flowchart of a method 300 for a software compiler to perform computation breakdown of an AI network and map it to multiple compute cores, which includes the following steps.

[0072] **At step S301**: In one embodiment, the **software compiler** can divide the tensor of each layer of a neural network into several **software tiles** (SW tiles), where each SW tile corresponds to the computation task of a portion of the network layers. At the same time, the software compiler re-establishes data dependencies based on the SW tile granularity to ensure the correct order of computation. The granularity of the SW tile division can be determined based on the optimal efficiency that the hardware can achieve in a single computation, so as to fully utilize computational resources.

[0073] Through this division method, the following technical effects can be achieved:

- **Improved Parallelism**: After network layers are divided into multiple SW tiles, they can be executed in parallel across multiple compute cores, thereby improving overall computational power utilization.
- **Optimized Memory Access**: The data for each SW tile can be scheduled and cached in the die-internal high-speed memory module. This reduces the number of accesses to external memory, lowers access latency, and alleviates memory bandwidth bottlenecks.
- **Flexible Scheduling**: The division of SW tiles and their execution order can be dynamically adjusted by the compiler based on the hardware resource configuration and memory layout. This allows the compiler to strategically schedule the execution of SW tiles, ensuring the output of a preceding SW tile is immediately used for the subsequent computation. This process significantly reduces the need to transfer network results in and out of the local high-speed memory, thereby lowering the probability of accessing off-chip memory and minimizing both storage access latency and the demand for off-chip memory bandwidth.

[0074] The present invention does not limit the method of SW tile division, scheduling strategy, and execution order described above. Please refer to FIG. 4. FIG. 4 shows an example of a four-layer AI network after SW tile division, where the re-established data dependencies based on SW tile granularity clearly indicate the dependency order of each SW tile. This is followed by step S302.

[0075] **At step S302**: In one embodiment, after completing tensor division in step S301, the software compiler further performs SW tile division along the highest dimension or the highest and second-highest dimensions of each tensor. This division method aims to distribute the computation tasks as evenly as possible among different compute nodes to improve load balancing. The chosen division dimensions are typically the highest dimension or the highest and second-highest dimensions of the tensor, which enhances the locality of data within the compute nodes, thereby reducing cross-node data transfers.

[0076] After completing SW tile allocation, the software compiler identifies which SW tiles and their corresponding data areas are depended upon by other compute nodes based on the data dependencies between SW tiles. FIG. 5 shows the allocation result after the SW tiles from FIG. 4 have been divided and allocated to two compute nodes, where the data areas requiring cross-node access (shaded parts) are marked to intuitively show the cross-node dependency relationship. This is followed by step S303.

[0077] **At step S303**: In one embodiment, the software compiler processes the SW tiles that have cross-computing-node dependencies. A **SendOps** node is inserted after each dependency source SW tile to indicate that after the source SW tile's computation is complete, the DMA module needs to be initiated to transfer the relevant data to the target computing node and to send a transfer-complete synchronous fence signal.

[0078] Correspondingly, a **RecvOps** node is inserted before each dependency target SW tile to indicate that before starting the computation task, the target SW tile must wait for the data transfer from the SendOps node of the dependency source SW tile to complete and receive the corresponding synchronous fence signal, thereby ensuring data consistency and execution order for cross-node computation.

[0079] Please refer to FIG. 6. FIG. 6 shows an example of Layer 0 and Layer 1 from FIG. 5, where the target SW tile for SW tile L0S2 is L1S3, and the target SW tile for SW tile L0S3 is L1S2. In these cross-node dependencies, the software compiler has inserted SendOps and RecvOps nodes, respectively. This is followed by step S304.

[0080] **At step S304**: In one embodiment, the software compiler processes the SendOps and RecvOps nodes inserted in step S303 to allocate the necessary receive memory for cross-node data transfer and to generate the corresponding data transfer and synchronization instructions. The specific process is shown in FIG. 7.

[0081] FIG. 7 shows a compilation method 700 for SendOps and RecvOps, which includes the following steps:

- **Step S701**: For each pair of SendOps and RecvOps, a receive storage area is allocated to receive the data. The local high-speed storage module 122 of the target computing node is prioritized. If it cannot be satisfied, the external memory 170 is used to store the data sent by the SendOps. Prioritizing the local high-speed storage module reduces external memory access and lowers bandwidth consumption.
- **Step S702**: Because the receive storage area allocated in step S701 might be occupied by other computation instructions of the receiving node during execution, the compiler inserts a "Wait buffer available fence" instruction at the data sending node to wait for the receiving node to notify the sending node of the availability of the receive storage area.
- **Step S703**: At the receiving node, instructions are inserted to wait for the data receive storage area to become available and then send a buffer available fence to the sending node, signaling it to begin the data transfer.
- **Step S704**: At the sending node, an instruction to start the data transfer is inserted, which transfers the data to the pre-allocated storage area of the receiving node via DMA or the internal interconnect bus.
- **Step S705**: At the receiving node, an instruction to "wait for the done fence from the sending node" is inserted to ensure that subsequent computations can only continue after the data transfer is complete.
- **Step S706**: At the sending node, a data transfer complete instruction is inserted, and a done fence signal is sent to the receiving node, notifying it that the transferred data can be safely used.

[0082] Through the above process, the software compiler can ensure the correctness and orderliness of cross-computing-node data transfer in terms of storage space allocation, transfer initiation, and synchronization confirmation, thereby guaranteeing efficient execution and data consistency for multi-node AI computation tasks.

[0083] In one embodiment of the present invention, the AI software compiler employs a method based on a depth-first parallel scheduling strategy at the multi-computing-node level. This scheduling method is based on a **Mixed-Integer Linear Programming (MILP)** global optimization framework. It formalizes the multi-objective scheduling problem into a solvable mathematical model and obtains a globally optimal solution. This method converts a complex, multi-objective scheduling problem into a solvable mathematical model to obtain a globally optimal solution.

[0084] The approach integrates the following two mutually restrictive optimization objectives:

- **Maximizing data reuse** in the die-internal high-speed memory by using a depth-first strategy to prioritize computations within the die-internal high-speed memory, thereby reducing access to external memory.
- **Minimizing inter-compute-core waiting latency** by optimizing the timing of data transfers and execution order to ensure efficient communication and synchronization between different compute cores.

[0085] To achieve these objectives, the method is implemented through the following three main stages:

- **Modeling Stage**: The optimization objectives, computation task dependencies, memory capacity limitations, and communication latency are collectively modeled as an MILP problem. This model accurately describes the overall operation of the distributed system and captures the complex interactions between compute cores.
- **Solving Stage**: A mature mathematical programming solver (e.g., Gurobi, CPLEX) is used to perform global optimization on the MILP model, obtaining a globally optimal scheduling strategy that avoids getting stuck in local optima, which is a common issue with traditional heuristic methods.
- **Execution Stage**: Based on the globally optimal scheduling strategy, the AI software compiler generates offline IPU instructions. The IPU controller then executes these instructions across the compute cores of the distributed system. Within each compute core, a depth-first execution method is applied to maximize data locality, while the ordering of synchronization operators (sender and receiver operators) is strictly controlled in the global time domain to minimize ineffective waiting and improve overall efficiency.

## 1.Mathematical Modeling Approach

### *1.1 Input Parameters*

| **Parameter** | **Symbol** | **Description** |
|---|---|---|
| Subgraph set | $\mathcal{K}$ | Each $k \in \mathcal{K}$ contains $G_k = (V_k, E_k)$ |
| Node execution time | $E_t(i)$ | $\forall i \in \bigcup_{k\in\mathcal{K}} V_k$ |
| Maximum layer count | $\max_L$ | Upper bound for scheduling layers |
| Cross-subgraph dependencies | $\mathcal{D}$ | $\{d = (X \in G_a \rightarrow Y \in G_b)\}$ |

| Parameter | Symbol | Description |
|---|---|---|
| Dependency penalty weight | $W_{\text{publish}}^{(d)}$ | Penalty coefficient for dependency violation |
| DFS weight | $\alpha$ | Depth-first weight |
| Violation upper bound | $\overline{\Delta}_d$ | Upper Bound on Dependency Violation |

### *1.2 Decision Variables*

| Main Variables | |
|---|---|
| $x_{k,i,\ell} \in \{0,1\}$ | Node $i$ in subgraph $k$ executes at layer $\ell$ |
| $y_{k,i,\ell} \in \{0,1\}$ | Node $i$ executes at layer $\leq \ell$ (prefix indicator) |
| $S_{k,\ell} \geq 0$ | Cumulative execution time for subgraph $k$ at layer $\ell$ |
| **Auxiliary Variables** | |
| $T_a^X, T_b^Y \geq 0$ | Prefix time of dependent nodes |
| $\delta_d \geq 0$ | Violation amount for dependency $d$ |
| **DFS Control Variables** | |
| $u_{k,\ell} \in \{0,1\}$ | Whether subgraph $k$ is active at layer $\ell$ |
| $p_{k,\ell} \geq 0$ | The number of tasks executed simultaneously in subgraph $k$ at level $\ell$. |

### *1.3 Constraint System*

**Fundamental Constraints (C1-C7)**

- C1: Node Layer Uniqueness Constraint

$$\sum_{\ell=0}^{\max_L} x_{k,i,\ell} = 1 \quad \forall k \in \mathcal{K}, \forall i \in V_k \qquad \text{(C1)}$$

Meaning: Each node must be allocated to one and only one layer for execution.

- C2: Prefix Indicator Definition Constraint

$$y_{k,i,\ell} = \sum_{t=0}^{\ell} x_{k,i,t} \quad \forall k \in \mathcal{K}, \forall i \in V_k, \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C2)}$$

Meaning: $y_{k,i,\ell} = 1$ if and only if node $i$ executes at layer $\ell$ or an earlier layer.

- C3: Cumulative Execution Time Calculation Constraint

$$S_{k,\ell} = \sum_{i \in V_k} E_t(i) \cdot y_{k,i,\ell} \quad \forall k \in \mathcal{K}, \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C3)}$$

Meaning: The cumulative execution time of subgraph $k$ at layer $\ell$ is equal to the sum of the execution times of all nodes that execute at or before layer $\ell$.

- C4: Define the parallelism at each level within a subgraph

$$p_{k,\ell} = \sum_{i \in V_k} x_{k,i,\ell} \quad \forall k \in \mathcal{K}, \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C4)}$$

Meaning: $p_{k,\ell}$ directly defines the number of tasks executed simultaneously by subgraph $k$ at level $\ell$.

- C5: Topological Dependency Constraint

$$y_{k,v,\ell} \leq y_{k,u,\ell} \quad \forall k \in \mathcal{K}, \forall (u,v) \in E_k, \forall \ell \in \{1, \dots, \max_L\} \qquad \text{(C5)}$$

Meaning: For an edge $(u, v)$ within a subgraph, the predecessor node $u$ executes before the successor node $v$. Additionally, it is generally allowed that dependent nodes execute within the same layer in the correct topological order.

- C6: Cross-Graph Dependency Time Binding Constraint (Big-M Linearization)

For each dependency $d = (X \in G_a \rightarrow Y \in G_b)$::

$$T_a^X \geq S_{a,\ell} - M_{a,\ell}(1 - x_{a,X,\ell}) \quad \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C6a)}$$

$$T_a^X \leq S_{a,\ell} + M_{a,\ell}(1 - x_{a,X,\ell}) \quad \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C6b)}$$

$$T_b^Y \geq S_{b,m} - M_{b,m}(1 - x_{b,Y,m}) \quad \forall m \in \{0, \dots, \max_L\} \qquad \text{(C6c)}$$

$$T_b^Y \leq S_{b,m} + M_{b,m}(1 - x_{b,Y,m}) \quad \forall m \in \{0, \dots, \max_L\} \qquad \text{(C6d)}$$

Meaning: Use the Big-M formulation to express '$T_a^X = S_{a,\ell}$ when $x_{a,X,\ell} = 1$ and $T_b^Y = S_{b,m}$ when $x_{b,Y,m} = 1$' in a form acceptable to a linear solver, where $M_{a,\ell}$ and $M_{b,m}$ are sufficiently large positive constants..

- C7: Cross-Subgraph Dependency Constraint (Soft Constraint)

$$T_a^X \leq T_b^Y + \delta_d \quad \forall d = (X \in G_a \rightarrow Y \in G_b) \qquad \text{(C7a)}$$

$$0 \leq \delta_d \leq \overline{\Delta}_d \quad \forall d \in \mathcal{D} \qquad \text{(C7b)}$$

Meaning: Allows for a violation of the dependency relationship, with the violation amount being $\delta_d$.

**DFS Forcing Constraints (C8-C9)**

- C8: Subgraph Layer Activity Constraint

$$u_{k,\ell} \geq x_{k,i,\ell} \quad \forall k \in \mathcal{K}, \forall i \in V_k, \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C8a)}$$

$$u_{k,\ell} \le \sum_{i \in V_k} x_{k,i,\ell} \quad \forall k \in \mathcal{K}, \forall \ell \in \{0, \dots, \max_L\} \qquad \text{(C8b)}$$

Meaning: $u_{k,\ell} = 1$ if and only if subgraph k has at least one task executing at layer $\ell$.

- C9: Subgraph Continuity Constraint

$$u_{k,\ell} + u_{k,\ell+2} - 1 \le u_{k,\ell+1} \quad \forall k \in \mathcal{K}, \forall \ell \in \{0, \dots, \max_L - 2\} \qquad \text{(C9)}$$

Meaning: Prevents a subgraph from "skipping layers" during execution, ensuring continuous execution.

### *1.4 Objective Function*

Pure DFS Optimization Objective:

$$\min \quad \alpha \cdot \sum_{k \in \mathcal{K}} \sum_{\ell=0}^{\max_L} p_{k,\ell}^2 + \sum_{d \in \mathcal{D}} W_{\text{publish}}^{(d)} \cdot \delta_d$$

**Composition of the Objective Function**:

- **First term**: $\alpha \cdot \sum_{k \in \mathcal{K}} \sum_{\ell=0}^{\max_L} p_{k,\ell}^2$ - Minimize the sum of squares of intra-subgraph task parallelism to achieve the first objective — performing depth-first scheduling within each subgraph.
- **Second term**: $\sum_{d \in \mathcal{D}} W_{\text{publish}}^{(d)} \cdot \delta_d$ - It penalizes cross-subgraph constraint violations (the dependency-satisfaction term) to prevent wasted computation time when a producer node finishes later than its corresponding consumer node becomes ready for execution.

### *1.5 Variable Domain*

$$x_{k,i,\ell}, y_{k,i,\ell}, u_{k,\ell} \in \{0,1\}$$

$$S_{k,\ell}, T_a^X, T_b^Y, \delta_d, p_{k,\ell} \ge 0$$

## 2.Solver Integration and Compiler Usage

After building the complete MILP model, the AI software compiler exports the model and passes it to a mature mathematical programming solver (such as Gurobi or CPLEX) for a solution. The solver internally utilizes efficient optimization algorithms to traverse a vast solution space, ultimately finding a globally optimal solution that satisfies all constraints and minimizes the objective function value.

## 3.Interpretation and Use of Solver Results by the Compiler:

The optimal solution returned by the solver is a set of specific numerical values for the decision variables. For example, the values of the variable $x_{k,i,\ell}$ will explicitly indicate which layer $\ell$ each node $i$ executes on within which subgraph $k$. The compiler uses these specific variable values to generate the final, executable AI instruction set.

1. **Task Assignment and Scheduling**: Based on the values of $x_{k,i,\ell}$, the compiler generates a scheduling list for each AI processor (i.e., each computing node), precisely specifying the execution order and location of each computation task.
2. **Memory Allocation**: The compiler determines which data should be stored in the local high-speed memory and which needs to be transferred across nodes, based on the model's constraints and variable values. It then generates the corresponding memory allocation and data movement instructions.

3. **Synchronization and Communication**: Based on the values of the cross-graph dependency variables (e.g., $T_a^X$ and $T_b^Y$) returned by the solver, the compiler can determine the optimal timing for cross-node data transfers. It precisely inserts "send operations" (SendOps) and "receive operations" (RecvOps), along with their corresponding synchronization signals (fences), into the instruction stream to ensure that data is available at the correct time, avoiding unnecessary waiting.

In this way, the present invention transforms a complex scheduling problem involving multiple mutually restrictive objectives into a solvable mathematical model. The solver provides the optimal global scheduling strategy, while the compiler is responsible for translating this theoretical solution into practical, executable hardware instructions, ensuring that the multi-AI processor system can complete large model inference tasks efficiently and collaboratively.

# Claims

**Claim 1: A distributed AI inference system, comprising:**

- **A plurality of compute cores**, each comprising an on-die high-speed memory module and a data movement engine (DMA);
- **A high-speed communication bus** directly connecting said compute cores, wherein said DMA is configured to transfer data directly between the on-die high-speed memory modules of different compute cores, without necessarily relying on external memory for inter-core data exchange; and
- **An AI software compiler**, configured to:
    - identify data dependencies across compute cores and insert SendOps and RecvOps into the instruction stream to facilitate inter-core data transfer, and
    - preferentially allocate data exchanges via the on-die high-speed memory modules, thereby minimizing latency and reducing bandwidth requirements for accessing external memory.

**Claim 2: A method for achieving globally optimal task scheduling in a distributed AI inference system, comprising:**

- **Modeling:** Transforming a multi-objective scheduling problem into a Mixed-Integer Linear Programming (MILP) model;
- **Objective function definition:** Defining a multi-objective function with hierarchical priorities, comprising:
    - **Cross-node dependency preservation (high-weight term):** Any violation of the producer-consumer execution order across compute cores incurs a large penalty, ensuring dependent tasks execute in the correct sequence and minimizing idle time caused by waiting for remote data; and
    - **Depth-First Search (DFS) data reuse (low-weight term):** Within each compute core, a DFS-based scheduling strategy is applied to maximize the reuse of on-die high-speed memory, improving local computation efficiency;
- **Solving:** Using a MILP solver to obtain a globally optimal scheduling solution that satisfies both objectives under the hierarchical weighting scheme; and
- **Execution:** Generating executable instructions based on the optimal solution, enabling each compute core to implement DFS execution locally while precisely controlling the timing of synchronization operators in the global time domain.

**Claim 3:**The method according to claim 2, wherein cross-subgraph dependencies are linearized using a Big-M method, treating such dependencies as soft constraints in the MILP model.

**Claim 4:**The method according to claim 2, wherein the Mixed-Integer Linear Programming (MILP) model further comprises:

- **Variables representing** the Node-to-Layer Assignment of nodes in active subgraphs at each layer, the Subgraph Layer Activity at each layer, and the Layer Concurrency of active subgraphs at each layer;
- **Node layer uniqueness constraint**, ensuring that each node is assigned to only one execution layer;
- **Topological dependency constraint**, ensuring that a preceding node completes execution before a succeeding node;
- **Cross-subgraph dependency constraint**, allowing a controllable amount of violation in dependency relationships, wherein any violation of the time completion order between

nodes with data dependencies across compute cores incurs a penalty to reduce idle computational time caused by waiting for remote data;

- **Subgraph continuity constraint**, preventing sub-graphs from skipping layers during execution; and
- **A global soft concurrency constraint,** applied per subgraph and per execution layer, configured to penalize the concurrency of nodes in each layer of each subgraph, the target being to limit concurrency in each layer of each subgraph to enforce a depth-first execution approach, the penalty being implemented as a squared-difference term to improve on-chip high-speed cache utilization.