rebellions_

Understanding RBLN Compiler

Oct 01, 2024

Compilers in AI Inference

A compiler translates high-level code into low-level machine code, but its role becomes especially crucial for deep learning models. Before delving into the role of the RBLN Compiler in accelerated AI inference, we need to grasp some characteristics unique to AI compilers.

Optimizations with Global Data

During inference, AI compilers optimize the execution of machine learning models based on the **global** data and computations. That is, AI compilers analyze the entire computational graph and data dependencies across all operations in the model, enabling global optimization strategies such as scheduling, memory allocation, and parallel execution. By doing so, AI compilers generate highly optimized code that minimizes latency, maximizes throughput, and efficiently utilizes hardware resources like Rebellions' ATOM™.

Compile-Time Memory and Dependency Management

During AI inference, memory allocation, cache (SRAM) utilization, and dependency management are handled at compile-time, allowing for precise control over resource allocation and data flow. Unlike traditional compilers, where memory allocation is dynamic, and caching and dependency management are left to the hardware during runtime, AI compilers pre-allocate memory and optimize data placement in SRAM based on the entire computational graph. This approach minimizes latency and maximizes efficiency by ensuring that all memory and dependency considerations are tightly integrated into the compilation process, tailored specifically for the AI model's execution.

RBLN Compiler

Optimization Goals

With the unique demands of AI compilers in focus, their high-level goals can be distilled into two key objectives:

- **Achieving fast computation with minimal direct memory access**
- **Maximizing parallelization of compute and memory operations**

The rationale behind these goals is clear. Fast AI inference relies on the efficient parallel execution of Direct Memory Access (DMA) and computational tasks. DMA facilitates swift data transfer between DRAM and SRAM, which is crucial for maintaining the data flow

needed to keep compute units operating at maximum utilization. Concurrently, the parallel execution of compute tasks across multiple cores—such as ATOM™'s Neural Engines—enables simultaneous processing of different segments of the neural network.

By intelligently scheduling DMA and compute tasks, the system can minimize idle time, reduce latency, and achieve higher throughput, ensuring that data is continuously available for processing without bottlenecks. This parallelism is critical for maximizing the performance of AI models, especially in real-time and low-latency applications.

Compilation Steps

1. Graph optimization and Op fusion

AI models are represented as computational graphs consisting of nodes and edges that dictate the flow of data. To accelerate inference, these graphs undergo rigorous optimization processes. Techniques such as common subexpression elimination (CSE) and dead code elimination (DCE) are employed to streamline the graph by removing redundant or unnecessary operations, effectively reducing its execution time, or latency.

Op fusion is the process by which multiple operators are fused into a single operator so that tensor and vector computations can be executed in parallel. It also reduces data transfers from shared memory and Scratch Pad in the Neural Engines.

2. Op splitting and grouping

In some instances, operations are simply too big to be handled efficiently by the system. When an operation exceeds the storage capacity of SRAM, it must be meticulously broken down into smaller, more manageable pieces—a process known as **op splitting**. Splitting the operations involves technical fine-tuning to maximize efficiency by taking into consideration the adjacent operations so that they can be scheduled optimally. This optimal scheduling is referred to as **grouping,** which minimizes unnecessary data transfers. Overall, **op splitting** and **grouping** lead to reduced overhead and maximized hardware utilization, ensuring that the system's capabilities are leveraged to their fullest potential.

3. Op tiling

Computation within the Neural Engines is further optimized through **op tiling**, which divides operations into optimally manageable segments across the multiple Neural Engines to enhance parallelism. Depending on the operation types—such as matrix multiplications

or activations—and the shapes of the tensors, Rebellions' Compute Library generates a tailored program, based on Rebellions' RISC ISA for Neural Engines. During the generation, the Compute Library determines computation details such as tiling, required SRAM size, and estimated compute time, ensuring precise and efficient computation. The information determined by the Compute Library is used by the Compiler in further compiler passes such as SRAM allocation and command scheduling.

4. Op scheduling

The scheduling of operations within the Neural Engines is of paramount importance. The Compiler must ensure that SRAM is utilized as efficiently as possible, adhering to the underlying principle of maximizing hardware utilization through parallelism.

5. Bufferization

In the early stages of compilation, a model is represented by a computational graph consisting of pure tensor operations that do not reveal memory operations. The tensor operations are abstract and do not directly correspond to specific memory locations like DRAM or SRAM. **Bufferization** bridges this gap by converting abstract data structures into concrete memory buffers—contiguous blocks of memory where data is stored. This conversion allows the Compiler to manage memory more effectively, ensuring that data is stored, accessed, and reused in the most efficient way possible, thereby reducing redundant SRAM usage.

Bufferization also facilitates more direct interaction with hardware, making it essential for efficient execution. Mapping operations to specific buffers enables the Compiler to generate code that is tightly integrated with the hardware's capabilities.

6. Memory allocation

During **memory allocation**, SRAM is allocated to buffers based on their lifetimes—ensuring that each buffer occupies memory only when necessary. By managing these lifetimes effectively, the system can parallelize compute and memory operations. This means that while some parts of the hardware are performing computations, others are simultaneously loading or storing data involved with previous or next computation tasks. This parallelism maximizes utilization, improves processing speed, and ensures that the computational resources are continuously engaged, avoiding unnecessary delays.

7. Dependency analysis

During **dependency analysis**, the Compiler analyzes memory (DRAM/SRAM) spaces

accessed by each compute and memory operation and finds dependencies between the operations that should be preserved during the model execution. It is crucial for ensuring the correct execution order and for optimizing performance through parallelization and other compiler optimizations. By understanding and managing dependencies, systems can execute programs more efficiently, especially in environments where parallelism is key to achieving high performance.

8. Command scheduling

The RBLN Compiler also plays a crucial role in determining the execution order of instructions. The primary objectives of command scheduling are to maximize parallelism for memory accesses and compute operations while respecting memory dependencies, ensuring that the system operates at peak efficiency.

9. Code generation

In the final stage, the RBLN Compiler generates optimized machine code for the Command Processor, which oversees the execution of the workload. Rebellions' AI accelerator chip excels at handling dependencies and memory management, allowing the Compiler to produce streamlined, performance-optimized code that leverages the hardware's capabilities to manage execution intricacies. Additionally, the Compute Library generates a program binary for the Neural Engines, resulting in an executable file that is ready for deployment.

Conclusion

The RBLN Compiler is a highly specialized tool designed to optimize the inference of AI models by leveraging the unique characteristics of neural processing hardware, like Rebellions' ATOM™. Through techniques such as graph optimization, op fusion, tiling, and memory allocation, the Compiler ensures that every aspect of the model's execution is fine-tuned for maximum performance. By effectively managing dependencies and leveraging hardware-based memory management at compile-time, the RBLN Compiler can generate highly optimized machine code that fully utilizes the capabilities of the underlying hardware, ensuring fast, efficient, and reliable AI inference.