AI Processor Basics

AI Systems Architecture Series
John Lakness, AI Architect, Intel Federal

Introduction

In this article, we will introduce the most common choices of core processor architectures that are used in AI systems in three categories: Scalar, Vector, and Spatial. For each, we will give some generalizations about the performance characteristics and the types of algorithms that they are optimized for. In later articles, we will go into more depth about how they are implemented, and their performance on different types of AI workloads.

Flynn's Taxonomy

Any exposition on processor architecture would be incomplete without the rather popular "Flynn's Taxonomy" simply because the nomenclature is common. Its original intent was to describe how a Harvard Architecture computer might ingest instruction and data streams, and probably makes the most sense in that context. Still, modern processors are often closer to one characterization than others, so we often refer to them in this way, but we should note that it would be a gross oversimplification to assume that any modern processor fits neatly into one of these categories. Presented here, with some liberties taken, is the taxonomy in a slightly more modern context.

SISD: Single Instruction Single-Data

The simplest form of CPU fits into this category. Each cycle of the CPU ingests an instruction and a data element and processes them in order to modify a global state. This concept is fundamental to computer science, and therefore most programming languages compile to a set of instructions that target this architecture. Most modern CPUs also emulate SISD operation, although very different concepts may be employed in software and hardware around it.

SIMD: Single Instruction Multiple Data

The simplest SIMD architecture is a vector processor, which is similar to a SISD architecture with a wider data type so that each instruction operates on multiple contiguous data elements. Slightly more complex is thread parallelism, in which a single instruction operates on multiple thread states, which is a more general programming model.

MISD: Multiple Instruction Single Data

There is no general consensus about what a MISD processor is, so I will take some liberties here. Consider an architecture that is able to execute multiple arbitrary instructions in sequence in a single cycle on a single data input. This would essentially entail multiplexing from output to input without storing intermediate results. Later, we shall see the advantages of this advanced architecture.

MIMD: Multiple Instruction Multiple Data

Again, I'm taking some liberties by saying that a Very Long Instruction Word (VLIW) processor best fits this category. The intent of such processors is to expose a programming model that more accurately fits the resources available to the processor. A VLIW instruction is able to send data to all of the execution units simultaneously, which has great performance benefits through instruction-level parallelism (ILP), but the compiler must be architecture-aware and do all of the scheduling optimization. This often proves challenging.

Scalar (CPUs)

Hybrid Performance

A modern CPU is an incredibly complex system designed to perform well at a huge variety of tasks. It has elements that span every class of Flynn's taxonomy. For instance, you can of course program it as a SISD machine and it will give you the output as if the program had been computed in the order you gave it. However, it is common for each CISC instruction to be converted to a chain of multiple RISC instructions for execution on a single data element (MISD). It will also look at all the instructions and data that you give it and it will line them up in parallel to execute data on many different execution units (MIMD). There are also many operations, such as in the AVX instruction set, that performs the same computation on many aligned data elements in parallel (SIMD). Also, with multiple cores and multiple threads running in parallel to use resources simultaneously on a single core, almost any type of parallelism from Flynn's taxonomy can be implemented.

A Code Optimizer

If a CPU were to operate in a simple SISD mode, grabbing each instruction and data element one at a time from memory, it would be exceptionally slow, no matter how high the frequency is clocked at. In a modern processor, only a relatively small portion of the die area is dedicated to actually performing arithmetic and logic. The rest is dedicated to predicting what the program will do next, and lining up the instructions and data for efficient execution without violating any causality constraints. Perhaps the most relevant to the CPU's performance versus other architectures is the handling of conditional branching. Instead of waiting to resolve a branch, it predicts which direction to take, and then completely reverts the processor state if it was wrong. There are hundreds of tricks like this etched into the silicon, which are tested on a wide variety of workloads to give a great advantage at executing highly complex arbitrary code.

Philosophy of Moore's Law

In one of my first jobs, I was given the assignment of integrating a very expensive ASIC which was considered necessary for the real-time decoding of satellite imagery. I noticed that the design was a few years old, and did some calculations which told me that by then I could have almost the same compute power on an Intel processor. I wrote the algorithm in C and demonstrated the system on a Pentium III CPU before the ASIC was available. At that time, the pace of 'Dennard Scaling' was so rapid that given a small amount of time, the performance

advance of a general-purpose processor eclipsed the need for a specialized processor. Perhaps the biggest advantage of choosing a general-purpose processor is that it is easy to program on, which makes it the platform of choice for both algorithm development and system integration. It is possible to optimize your algorithm to a more specialized processor, but the CPU is already very good at doing this for you. In my particular case, the first version of the satellites used Reed-Solomon codes, but Turbo Codes were considered for later designs. The downlink sites that had used the ASIC would have to replace their entire system, and our sites would work with a simple software update and a regular CPU upgrade. So, you can either spend your time optimizing code, or you can spend your time innovating applications. The corollary to Moore's Law is that soon enough it will be fast enough.

Vector (GPUs and TPUs)

Simple and Parallel

In many ways, a vector processor is the simplest modern architecture to talk about: a very limited computation unit that is repeated many times over the chip to perform the same operation over a wide array of data. These were first popularized in graphics, hence the term GPU. In general, a GPU does none of the predictive gymnastics that a CPU does to optimize complex arbitrary code, and specifically has a limited instruction set to only support certain types of computation. Most of the advances in GPU performance have come through basic technological scaling of density, area, frequency, and memory bandwidth.

GPGPU

Somewhat recently there has been a trend to expand the GPU instruction set to support general-purpose computing. These GP instructions must be adapted to run on the SIMD architecture, which exposes some advantages and disadvantages depending on the algorithm. Many algorithms that are programmed to run as a repeated loop on a CPU are actually just performing the same operation on each adjacent data element of an array for each cycle. These can easily be parallelized, sometimes massively on a GPU, with some programmer effort. There are some caveats. If there are any conditionals on any of the elements, then all of the branches must be run on all of the elements. For complex code, this can mean exponential increase in computation time versus the CPU. GPUs have very wide memory busses that provide excellent streaming data performance, but if the memory accesses are not aligned with the vector processor elements, then each data element requires a separate request from the memory bus, while a CPU has a very complex predictive caching mechanism that compensates greatly for this. The memory itself is also very fast, but small, and depends on transfers over the PCIe bus for data access. GPGPU algorithm development is, for the general case, much more difficult than for a CPU. This challenge, however, is somewhat addressed by the discovery and optimization of efficient parallel algorithms which achieve the same result with uniform execution branching and aligned memory access. Often these algorithms are less efficient in terms of raw operations, but much faster to execute across a parallelized architecture.

AI Operations

Many popular algorithms in Artificial Intelligence are based on linear algebra, and a great deal of advancement in the field has been simply due to massively expanding the size of parameter matrices. The parallelism of a GPU allows for massive acceleration of the most basic linear algebra, so it has been a fit for AI researchers, so long as they stay within the confines of dense linear algebra on matrices that are large enough to occupy a large portion of the processing elements, and small enough to fit in the memory of the GPU. The acceleration is so great, however, that a great deal of the progress in deep learning up to today has occurred within these limitations. The two main thrusts of modern development in GPUs have been toward tensor processing units (TPUs), which perform full matrix operations in a single cycle, and Multi-GPU interconnects to handle larger networks. Today, we are experiencing greater divergence between the hardware architectures for dedicated graphics, and hardware designed for AI. The simplest divergence to talk about is in precision, where AI is developing techniques based on low-precision floating point and integer operations. Slightly more obtuse are the shortcuts that a graphics processor takes to render convincing recreations of an impossibly complex scene in real-time, often with very specialized computation units. Thus, the similarity between architectures ends at the highest levels of optimization for either.

Spatial (FPGAs and ASICS)

Systolic Arrays

An ASIC or FPGA can be designed for any type of computing architecture, but here we focus on a specific type of architecture which is somewhat distinct from the other choices, and relevant to artificial intelligence. In a clocked architecture such as a CPU or GPU, each clock cycle loads a data element from a register, moves the data to a processing element, waits for the operation to complete, and then stores the result back to the register for the next operation. In a spatial dataflow, the operations are physically connected on the processor so that the next operation executes as soon as the result is computed, and there is no storage of the result in a register. When moderately complex units that may contain their own state in registers local to the processing elements are chained together in this way, we call it a ‘systolic array’.

Power, Latency, and Throughput

There are some immediate advantages that are easily realized. In a register-based processor, power consumption is mostly due to data storage and transport to and from the registers. This is eliminated, and the only energy expended is in the processing elements, and transporting data to the next stage. The other main advantage is in latency between elements, which is no longer limited to the clock cycle. There are also some potential advantages in throughput, as the data can be clocked into the systolic array at the rate limited only by the slowest processing stage. The data clocks out at the other end at the same rate, with some delay in-between, which establishes the dataflow. Compared to the synchronous clock-execute-store loop, this can be orders of magnitude more energy efficient and/or faster, depending on the goals of the architecture.

Dataflow Design

If a CPU is the easiest to program, and a GPU presents a greater challenge, then FPGA requires a very significant effort with a great deal of skill, and an ASIC requires a massive investment of cost and engineering. Still, the benefits can be tremendous for specific algorithms. To get an idea of how much advantage there can be, consider that the 'normalized delay' of one inverter driving another in a modern silicon process is measured in single picoseconds, whereas a clock cycle is closer to nanoseconds. Similarly, the transport energy is a function of resistance and capacitance, which can be computed from the length of interconnects, and the distances between processing elements can be orders of magnitude shorter than distance to registers that hold data between clock cycles. The FPGA does not get quite as much of an advantage because there are additional fan-out, switching, and transport delays between elements, but it gives the flexibility advantage of being able to adapt to multiple dataflow architectures with one chip. Although any type of algorithm can be implemented, there are limitations to complexity because conditionals require layout of both branches, which massively increases the area and decreases utilization efficiency. FPGAs and ASICS may also employ a mix of synchronous and systolic architectures to optimize the tradeoff between layout efficiency and speed.

Dataflow Systems

The most common type of systolic array for AI implementations is the tensor core, which has been integrated into a synchronous architecture as a TPU or part of a GPU. Many different types of convolution cores have also been proposed. Full dataflow implementations of entire deep learning architectures like ResNet-50 have been implemented in FPGA systems, which achieved state-of-the-art performance in both latency and power efficiency for inference. Customizability also allows for arbitrary bit-length precision, which decreases layout size and processing delay, but must be carefully tuned to the statistical performance requirements of the system. The major unique capability, however, is that the real-time nature of the processing allows integration of AI with other signal processing components in real-time systems.

Conclusion

When choosing an AI processor for a particular system, it is important to understand the relative advantages of each within the context of the algorithms used and the system requirements and performance objectives. In later chapters, we will look at some considerations and examples of this. We will see each one of these processor architectures become greatly advantaged over the others in various system-level considerations.

References

1. Fowers, J., K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, D. Burger, et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI." *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018. doi:10.1109/isca.2018.00012.

2. Hennessy, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Burlington, MA: Morgan Kaufmann, 2017.
3. Sze, V., Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." *Proceedings of the IEEE* 105, no. 12 (2017), 2295-2329. doi:10.1109/jproc.2017.2761740.
4. Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter: “Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning”, 2018; arXiv:1801.08058.