Model Compute Co-design for Composable Models

A Possible Path to Artificial General Intelligence

Fei Sun
fsun@feisun.org

July 2022

Contents

1 Prologue

I have always been a big fan of artificial general intelligence (AGI). In fact, I'm among the optimists who believe that AGI may be achievable in our lifetime. Since the beginning of last year, I have spent quite some time pondering on the directions of AI research that may lead us to AGI. It resulted in some power points here, here, and here. This article is an extension of those works, and condenses my thinking in words. Please note, this is not a technical paper, not a survey paper, not even a position paper. Rather, it is a blog style article with many hypotheses, which records my understanding on this topic at this specific time. I have discussed this topic with many people and got many suggestions and feedbacks. Some of the contents in this article reflect their contribution. The citations in the paper are not meant to be comprehensive, but to give readers a few leads to dig deeper.

I seem to learn quite a bit every day. My current way of thinking may well be the result of lacking the big picture, and having superficial understanding in certain domains. With this article, I hope to induce some discussions and critiques, which will help me dive deeper on this topic. Please drop me a message if you are interested in discussing more.

2 Current Status of AI

The current AI boom started in 2012 with AlexNet [1], which revolutionized computer vision (CV) tasks with CNN. Deepmind's AlphaGo [2] in 2016 breathtakingly beat human champions in the game of Go using reinforcement learning. The self-attention modules in the transformer architecture [3] dominated the NLP domain, and later penetrated to CV and many other domains. Needless to say, our lives have been fundamentally reshaped in the last decade from the applications of deep neural networks (DNN), a new branch of AI.

In 2018, OpenAI contributed the driving forces of AI advancement to three factors: algorithmic innovation, data, and the amount of compute available for training [4]. The discovered transformer architecture exhibits superb scaling capabilities: the loss is reduced smoothly as we increase the model size, dataset size and the amount of compute [5]. As a result, researchers compete against each other on their capabilities to end-to-end train larger models in a shorter period of time. It is very easy to scale model sizes, and relatively easy to scale data in selected domains due to the unlabeled self-supervised approaches, but it is very costly to scale compute. The price tags of training large models are so high that only the companies and research institutions with the deepest pockets can afford to train a few highly profitable models [6, 7].

Sometimes, we wonder: will we achieve AGI by simply scaling the models, collecting more data, and monolithically training larger models end-to-end? We are less optimistic on these approaches, and would like to explore alternative means. In our opinion, the ability to compose is essential to any general intelligence. In the next section, we will explain our thinking process. Then, we will

describe how composition will change the way we design models, collect data, and build efficient computers.

3 Composition

The ability to use tools separates humans from the less intelligent animals. As an example, if we want to build a house (or whatever complicated task), we do not build the house using our bare hands directly. Instead, we first build different kinds of tools to make building houses easier. Then we use those tools to build houses. Better yet, if other people have previously built similar tools for other purposes, we could leverage their work and purchase the tools instead of building them. Thus, we actually divide a complicated task (building a house) into two relatively independent tasks: building tools, and using the tools to build houses. Each task may leverage previous works and do not need to start from scratch. Solving these two problems is actually simpler than solving the combined problem directly.

This is just an example of decomposition: a complicated task is divided into several simpler sub-tasks, and solving the sub-tasks obtains the solution for the complicated task. This is an important skill that people use to conquer difficult problems. In recent history, both computer software and hardware designs are built on top of this technique.

On the software side, all high level languages have the concept of procedures and functions, which instantiate the same computation multiple times with different arguments. This is called function composition [8]. Yet a higher level abstraction called object composition [9] is often used in object oriented programming to cluster the closely related objects while separating the loosely coupled ones. They serve as the cornerstones of modern software infrastructure.

The importance of composition may be more profound on computer hardware. The electronic analog computers [10], which were driven by the physics of current and voltage, were popular in the early 20th century. But nowadays, all computers are digital, built on the symbolic Boolean zero and one values. Analog electronics only exist to interface the physical world (*e.g.* sensors, long range communication). Analog signals are converted to digital domain via ADCs whenever possible. One may wonder, why?

Please note, analog computing is far more efficient than digital computing for most basic operations. For example, analog addition may be achieved by carefully regulating multiple current sources going into the same wire. The difficulty of analog computing is its scalability. Because it deals with current and voltage in a continuous domain, a slight change in one area of the circuit may affect the circuitry precision far away. Due to process variations, errors in the continuous domain are unavoidable, but it is difficult to constrain such errors locally. Most analog systems spend the vast majority of the design effort to reduce the error and error propagation instead of designing for functionality. This butterfly effect makes it very difficult to design large analog systems. Thus, limited reuse opportunities exist in the analog domain and new systems need to

be designed end-to-end [1]. It is difficult to abstract the design rules in the general sense, and the electronic design automation (EDA) tools are still very immature. This means, composing simple analog systems to build more complicated ones is hard to automate. The scale of the system is determined by the smartest designer.

On the contrary, digital computing is much more complicated for basic operations, as it cannot leverage the voltage-current curve in the physical world. Instead, everything is built on top of zeros and ones based on Boolean algebra. Process variation errors cannot pass clock boundaries (Flip-flops reset zeros and ones). Thus, digital circuits for different tasks can be independently built. When a more complicated system needs to be designed, it can be **composed** from the already designed simple systems. This significantly improves the design efficiency as we do not need to reinvent a slightly different wheel for every new design. Due to the same reason, design rules have local implications. They are clear and easy to conclude. Then, we may raise the abstraction hierarchy. High level designs do not need to worry about the low level details, and EDA tools can concretize the high level designs automatically. From lowest to highest levels of abstraction, the current digital systems span from device, circuit, gate, register transfer level (RTL), intellectual property (IP) to system on chip (SoC) [2]. Each higher level of abstraction is a ten fold increase in productivity. The entire digital empire is built on its capability of composition.

In the early 20th century, people understood analog circuits better than digital circuits. Small analog devices were faster and more efficient than digital counterparts. But the analog proponents faced a difficult problem of designing larger analog systems. Fast forward to current days, the DNN researchers face similar problems. DNN models are monolithic and without hierarchy. End-to-end training using raw data usually provides better quality and is strongly preferred. Researchers strive for larger models but computation barely keeps up. Those DNN characteristics are more similar to analog computing than digital computing: no composition. If history repeats, will we see some future "digital DNN" based on composition to overtake the current monolithic "analog DNN"?

We consider it highly probable that finding the ability to compose is an essential step towards AGI, as AGI is a complicated system and is likely to be built on many simpler systems. This idea is also shared by some AI researchers. For example, the world model proposed by Yann LeCun [11] requires some degree of composition to raise the abstraction level and design a hierarchical system.

Researching the future composable DNNs may face some challenges in the current state:

[1] Some design templates still exist, but they may not be used as is most of the time. Careful adjustments need to be made for different systems or even the same system in different technology nodes. Designers' experience still plays a big role and experienced designers are hard to find.

[2] Instead of raising the abstraction level, SoC leverages IP reuse to increase scalability. A system may be further scaled to system in package (SIP) and multi-chip module (MCM) using the same method.

- Composition may relieve the designers' effort to re-design (re-train) everything from scratch. DNN blocks designed in one context may be used without fine grained training to improve design productivity. However, it means that a composable system may not be best designed for the task, and an end-to-end designed system at the same scale may beat its performance.

- The way to overcome the above mentioned issue is to further increase the system complexity: to a scale that the existing end-to-end training method cannot handle. The benefit of composition becomes obvious once the increased complexity outweighs the loss of efficiency. However, it imposes significant pressure on inference: inference time is much worse. This may require some paradigm shift: merging training and inference.

- With composition, the design abstraction level may be increased. However, efficient composition mechanisms still need to be researched, especially on higher levels. It is unclear how much the gradient descent mechanism may be used in such composable systems.

Since the research on composable DNNs is still at its infancy, it may be challenging to compare the quality of such models with the existing DNN models. It may require some leap of faith to stick to this direction. In the following sections, we will describe what we view as the important areas to research on, divided between algorithm, data, and compute. Fortunately, many of the research areas may benefit conventional DNNs as well.

4 Algorithm

A paradigm shift is the result of many contributing factors. Redesigning the computing hardware is very costly and may not be justified without strong application needs. Collecting more data does not change the trajectory of model capabilities, but more data reduces the likelihood of overfitting and enables larger models to be efficiently trained. Re-architecting the models and inventing new training approaches are relatively cheap methods to demonstrate the potential benefits of composition [3].

Thus, algorithm trend is a forward indicator of future paradigm shifts. The following subsections describe a few algorithm research directions that may increase our understanding on composition.

[3]Model exploration is highly constrained by the capabilities of the underlying hardware. It is possible that some novel model architectures do not show benefits simply because the hardware is inefficient to those models and cannot train them to the same degree as the models more efficient on the hardware. Thus, model, data, and compute are highly intertwined, and they are advanced in an iterative manner.

4.1 Model compression

Model compression is an important research topic since DNN models are deployed in production. One may improve the model accuracy by increasing the model size and training longer time in the lab setting. But the slight accuracy improvement may not justify the cost of the increased resource consumption. In some real time systems, the latency or throughput are hard requirements and the accuracy is optimized based on such constraints. Many works have shown that redundancies exist in popular models and various methods may be applied to squeeze them out with minimal impact on accuracy.

Model compression is even more important for composable DNN. The benefits of composition may be obvious on ultra large models that the conventional end-to-end training methods fail to train them. Such models are unlikely to satisfy the production deployment requirements. One may use those models as teachers and distill smaller student models. Alternatively, each component of the model may be compressed to make the model leaner.

Many model compression approaches exist, mostly targeting unidirectional compression from training to inference. Below are a few examples:

- **Quantization**. Models are trained using floating points (FP32, FP16, or BF16) because they represent real values with minimal gaps. However, floating points are costly in hardware and many DNN inference accelerators only implement integer operators. The weights in the floating point models may be quantized to the nearest integer values so the models may run on such hardwares. The size of the models may also be reduced by replacing floating point values (32 or 16 bits) with 8-bit or less integers. This is a well studied area and many training tricks may be used to reduce the accuracy loss due to quantization.

- **Factorization**. A matrix may be factorized into products of multiple matrices, with the total number of parameters less than the original matrix. Similarly, convolutions may also be factorized. A notable example is the depthwise separable convolution, which divides a regular convolution into two convolutions: one convolves the spatial dimensions and the other convolves the channel dimension. Factorization reduces both storage and computation, but they are usually harder to train.

- **Sparsity**. DNN is built on cascaded matrix multiply accumulation operations of the weights and activations. In this process, all weights and activations are treated equally, consuming the same amount of computation. Sparsity introduces zeros to the weights and activations so that multiplications on such values do not affect the output, thus such computation may be avoided. Section 4.2 describes this approach in detail.

In such large composed models, the boundary between training and inference is blurred. It is possible that some components are relatively stable and are compressed for computation efficiency, while some other components are actively

exploring better architectures with more redundant structures to reduce the exploration difficulty. In such scenarios, compression is not a one way street from training to inference. It may be necessary to expand the compressed modules easily to the complete form to improve exploration efficiency. This, however, is a less studied area, which may reveal its importance as composition becomes more mainstream.

4.2 Sparsity

Most existing DNN models are based on dense matrix matrix multiplications. In such a computation pattern, every weight and activation consumes the same amount of computation. However, they have different magnitudes and their contribution to the output values are different. This discrepancy results in an inefficient use of compute resources. This may not be a major issue for small models running on hardware optimized for dense computation. But it is still not sustainable for ultra large models when the hardware resource is already the bottleneck.

Sparsity is a way to resolve such a conflict. By introducing zeros or groups of zeros (explicit or implicit) to the weights and activations, portions of the dense computation may not affect the output and thus may be skipped. In some sense, *sparsity is a mechanism to only compute the relevant features and weights in a model*. It balances the resource consumption and model output contribution for each weight.

We may be reaching the tipping point that sparsity will be a first order design consideration in DNN models. It will enlarge the design space significantly and will broaden researchers' imaginations. I compare the current situation on sparsity with the time that "Dennard scaling" was about to be broken in the early 2000.

"Dennard scaling" [12] refers to the observation that every newer technology node in very large scale integration (VLSI) increases the logic frequency by 40%, which translates to 40% higher performance. In the 80s and 90s, the CPUs were all single core, but every product generation would increase application performance by 40%. In those golden times, software was optimized for a single core and everyone's mindset was to increase frequency to boost performance. But frequency could not be increased infinitely. In the early 2000, frequency plateaued, so was the performance. At that time, designers were forced to think out-of-the-box, and designed multi-core CPUs. The performance improvement due to multiple cores far outpaced the previous improvement from frequency increase.

The story of the "Dennard scaling" [4] tells us that people have the tendency to spend the least amount of effort to get reasonable benefits (increasing frequency). Architecture enhancements are constrained to single core improving instructions per cycle (IPC). Major breakthroughs (and larger investments) are only possible when the easier approaches stop working.

[4] The story is simplified to emphasize the points.

In DNN, we are reaching the time that training large models is no longer economically feasible. For example, training the GPT-3 model with 174 billion parameters requires a supercomputer and costs $12M [13]. We are about to reach an era where the current horizontal scaling approach is no longer affordable. We may be forced to think out-of-the-box and invest heavily to solve more difficult problems. Sparsity has the potential to deliver higher speedup than horizontal scaling at a fraction of the cost.

The industry has already started the process of exploring sparsity. The mixture-of-experts (MoE) models [14] dynamically select a small subset of the experts during runtime. As a result, much larger models may be trained without increasing computation [15, 16]. NVIDIA's A100 GPU implements sparse tensor cores [17] to support fine grained structured weight sparsity, which is the first mainstream hardware vendor supporting sparsity.

So far, the shift to sparse computation is mostly the algorithm's response to plateaued hardware performance. But sparsity also introduces a lot of hardware challenges. They will be discussed in detail in the subsections of section 6. The paradigm shift to composable models may drive sparse model designs from the application needs directly.

4.2.1 Static sparsity

Static sparsity, also called weight sparsity, refers to a fixed sparse pattern in the weight matrices that does not change for different input data. It is a widely studied sparsity to improve the computation efficiency.

The idea of static sparsity is inspired from biological brains, which is known to be ultra sparse at a fine grained level [18]. Human brains experience a period of dense connections in the early years, then are followed by massive pruning. Still, even full-grown brains change a large percentage of their synapses every day.

This is similar to the DNN pruning process, starting from a dense model and pruning to sparse. Some works also repeatedly grow and prune the models to explore better sparse patterns [19, 20]. However, the DNN's achievable sparsity without significant degrading model accuracy is far less than the biological brains. As the activations remain dense, the runtime speedup is less than many researchers have hoped for due to hardware limitations (details see section 6). Also, people view sparsity the same as quantization: a mechanism only to improve computation efficiency. All of them limit the wide adoption of static sparsity.

In our opinion, sparse patterns are first level model structures that need to be explored. The pruning techniques to sparse models is comparable to the neural architecture search (NAS) techniques to dense models. As the model tends to go sparse when viewed in a top-down approach, the temporary hurdles may be overcome by leveraging dynamic sparsity, significantly changing the training methodology, and drastically changing the underlying hardware.

4.2.2 Dynamic sparsity

Dynamic sparsity, as its name indicates, refers to the activation values or groups of activation values becoming zero during model inference when feeding in different inputs. Such activations do not need to be computed (output sparsity). They also do not affect the activations of the next layer and that computation may also be skipped (activation sparsity).

To most researchers, it is still a mechanism to improve computation efficiency. Depending on the inputs, a different portion of the non-zero weights do not participate in the computation. Unlike static sparsity, whose accuracy is usually compared with the dense model the sparse model is pruned from, the model with dynamic sparsity is usually compared with the dense model with equivalent average FLOPs (or number of parameters). [21] surveys various dynamic sparsity techniques in this context.

At some coarse grained level, researchers propose model architectures that leverage dynamic sparsity without explicitly mentioning it. For example, in mixture of experts (MoE) models [14], different tokens flow to different experts based on the output of a gating network. This selective routing mechanism may be viewed as dynamic sparsity: a token is not routed to all experts, but to a few selected experts, which may be equivalently expressed as passing an expanded token to all experts with the selected experts obtaining the replicated original token values but all other experts receiving tokens with zero values.

MoE models are examples of models exhibiting control flow behaviors. The gating network serves as the control path, while the tokens are in the data path. In this sense, any control flow model may be viewed as a data flow model with dynamic sparsity. As an extension, the sparsely-activated models [22] advocated by Jeff Dean and later the Pathways system [23] fall in the same category.

In fact, MoE models may be viewed as simple composable models. Each expert is a component and the MoE model is decomposed to many experts, each serving a subset of tokens. Everything is done via back propagation in an end-to-end fashion. The composition features missing from the MoE models are training each component separately and sharing the components for different tasks.

As more large models resort to the MoE mechanism, if we do not view it as a well studied standalone methodology, but to view it as a first step in exploring a much broader scope composable model, would it make it easier to advance the model exploration research?

4.3 Memory

A composable model is a heterogeneous model. Each component serves a different objective, ideally orthogonal to other components. In this sense, ResNet is unlikely to be composable, as the same layer structure is stacked back to back. But a transformer model contains more component structures: the architectures for the self-attention block and the feedforward block are different and they serve drastically different purposes.

Trying to figure out the composition mechanism directly is a daunting task. It may be more realistic to start from a case study, conclude its characteristics, and extrapolate to other areas. Memory may be a good candidate for the case study, whose significance is overlooked.

In current deep learning models, weights serve dual purposes: 1) store the methods to solve the training task, and 2) store the experience learned from the training data. Current models are indifferent to the nuances of the two purposes, which contribute to the ever increasing model sizes. Memory may be able to decouple such two objectives: the model is only used to learn methods, while experiences are saved in memory.

Memory is a piece of storage external to the model. It is the start and end points of back propagation. In the simplest sense, it is an embedding table. In a conventional embedding table, the embedding indices are generated via a fixed mechanism using human's prior knowledge or via comparing all entries with the input based on similarity. However, in memory, the embedding entries are extracted from the location based addressing mechanism generated from the model.

The content of a memory may be generated in training and used in inference. It recently became a heated topic, exemplified by DeepMind's work RETRO [24], which achieved the state-of-the-art results using 25$\times$ fewer parameters. However, accessing its 2 trillion token memory still relied on content based addressing.

On the other hand, data in the time domain are usually processed using recurrent neural networks (RNNs). The LSTM or GRU models store sequential information in the activations requiring evaluation in every time step, which may be easily forgotten. The memory augmented neural networks (MANN) try to address this issue by incorporating memory in the model architecture. Early works include neural turing machine [25] and differential neural computer [26] from DeepMind and memory networks [27, 28] from Facebook. This line of research, however, was overshadowed by the self-attention based architectures. Transformer models no longer need to *remember* the previous tokens. Instead, the models access all previous tokens within a certain limit, which incur no information loss and become vastly popular on solving problems with short sequences.

However, the transformer architecture is fundamentally difficult in solving long sequence problems. The relatively low accuracy on the long range arena benchmark [29] may reveal such challenges. For example, the best transformer family model may only achieve 37.27% accuracy on the simple ListOps dataset [29], while a simple memory network achieves near perfect accuracy [30], even with extended sequence length in our experiments. We hypothesize that a feed forward network consuming many input tokens at once unnecessarily incurs complicated logic, while it may be easily solved when consuming one input token at a time. For example, it is difficult to compute the result of the sum of ten three-digit variables at once. But it is easy to compute the sum of two three-digit variables ten times in a loop.

Thus, MANN may be suitable for problems with long sequences. We believe

this is an overlooked high potential research direction. An efficient location based addressing mechanism is a key to reduce memory access computation [5]. Recently, we see more active research in this area. For example, [31] introduces control paths in memory address logic and further improves the state-of-the-arts. We hope this trend continues.

4.4 Neural symbolic computing

Symbolic AI and connectionist AI are long standing two schools of thought. It became a heated debated topic recently because of Gary Marcus's article "deep learning is hitting a wall" [32], which discounted the potential of DNN and advocated an integration of symbolic computation. Later Yann LeCun countered it in "what AI can tell us about intelligence" [33], which admitted that symbolic computation was only a hurdle to overcome. Yann LeCun also briefly discussed the relations of symbols and the world model in [11], but without a concrete solution.

Symbols are discrete, less informative than continuous real values. But it is more robust, as small variations are clipped to the closest symbol. In the analogy of the analog and digital circuit described in section 3, existing DNNs are considered closer to analog circuits, with continuous values and propagated errors. If the analogy holds, maybe it is possible to build higher level architectures using existing DNN models, whose interfaces are discrete and less error prone. That said, existing DNNs are similar to combinational logic, which are merely components in a much larger sequential logic.

In this setting, symbols are interfaces between DNN modules. The discrete nature of symbols serves two purposes. 1) It prevents the prediction error propagated from one DNN module to the next, such that the next DNN always receives the standardized inputs. It enables training at component level as output variations in one module are clipped and do not affect the other modules. 2) Less information is passed from one DNN to the next. It forces the content of the interface (*a.k.a.* symbol) to preserve the most valuable information, and the noise is stripped out.

As described in [11], a hierarchical system may require some discontinuous action space at high levels of abstractions. In a composed system, the discrete symbols at component boundaries would also extend the scalability by stabilizing the system. Language is an example of such a component interface already developed by people.

Language is composed of alphabets (*i.e.* symbols). But it doesn't exist on its own naturally. It is either printed in a book (fixed patterns of black ink on white paper), or from someone's voice (sound waveforms following specific patterns). People interpret languages in their minds to understand the meaning. Significant noises are filtered out in the process of generating and receiving languages. For example, it doesn't matter whether a sentence is printed in

[5] Location based addressing sparsely accesses entries in the entire memory at different time steps. It may be viewed as a kind of dynamic sparsity in the time domain.

bold, italic, underline formats, or a sentence is spoken by person A or person B, the same sentence is received and understood. Also, the discrete nature of language limits the communication bandwidth. It forces a person to distill the most important, high level information to convey in language. Starting from it, the receiving person would fill in the missing information based on prior experience.

Current DNN models are naturally divided at language boundaries. The OCR models recognize letters in different formats. The ASR models listen to various human voices and translate to text. The TTS models create voice waveforms from texts. Starting from language, various NLP models perform many different tasks, either to understand it, or to generate more of it. This is already a kind of composition, based on human prior knowledge.

Language is an example. Would it be possible to create more of such discrete symbolic interfaces among DNN components? Connectionist AI and symbolic AI are not disjoint schools of thought. We've seen a growing trend of research integrating them together recently, which is called neural symbolic computing [34, 35]. Many still rely on prior knowledge to decompose a system [36].

Ideally, we would like the system to be decomposed as a side effect of end-to-end training, and the symbols emerged in the process. It may require solving multiple complicated problems and sharing their components. This is a non-trivial problem, but as we hope to scale to much larger systems, this is a promising direction.

5 Data

Prior to the deep learning era, model sizes were relatively small due to algorithmic and computational limitations. A small model with slightly larger dataset did not show overwhelming benefits. As labeling data was costly, most researchers focused on small datasets. Yet Feifei Li still managed to crowdsource the ImageNet dataset [37], which was over ten times larger than the previous ones. The ImageNet challenge further set a grand target and attracted researchers to explore innovative algorithms. Up to now, ImageNet dataset is still the golden standard to compare the quality of various models.

More labeled training data reduce the likelyhood of overfitting, which is critical to increase the accuracy of larger models. Much larger datasets [38] are created by a few companies having easy access to vast amounts of data. However, manually or semi-automatically labeling data was still very costly and error prone, which became the bottleneck of increasing dataset size even further. With more algorithmic innovations, useful features could be extracted from unlabeled data via self-supervised training methods [39]. It first became popular in the NLP domain due to ubiquitous availability of text data on the internet. Models with hundreds of billions or even trillions number of parameters may only be trained with unlimited amounts of data [6, 15, 16] and still see noticeable accuracy improvement. The vast number of image-caption pairs available in the internet also found their uses due to the recent advances in multi-modal

models [40, 41].

From the advances of deep learning in the last decade, we can see that dataset creation and standardization usually come before algorithm innovation. A large, challenging, and easily obtainable standard dataset reduces the entry barrier to the domain, and attracts many researchers advancing algorithms by making them easily comparable.

As resources pile into training large models, the need for large training dataset is ever increasing. However, in the real world, data may only be cheaply retrieved in a few domains, which are already widely studied. There exists a long tail of specialized domains lacking significant amounts of data. Some of them are in the neighborhood of the domains with sufficient amounts of data, thus transfer learning methods may be adopted to pretrain the model with large amounts of data in the neighborhood domain, then fine-tune (or even linear probe) the model in the target domain with limited data. This may suffer from data distribution shift issues and decrease accuracy. If some domains are not in the neighborhood, they are lagged behind and no efficient solutions exist.

Thus, current DNN models are narrowly successful in areas with abundant data. The monolithic large models, which require large amounts of data to train on, prevent them from penetrating more domains. If, however, a large model is decomposable to several smaller modules, each trained separately, we may reduce the required training data for each module. A composable model may also be trained on a few related domains, each lacking sufficient data. The components may emerge during joint end-to-end training. Compared with transfer learning, the composition approach may be more effective in domains with limited data.

Thus, composition may revert the current appetite for large amounts of data, and spread DNN models to broader areas.

6 Compute

Exploring novel model architectures requires repeated experiments. In each experiment, the model architecture and training data are already fixed. The experiment turnaround time is largely determined by the amount of compute used in training, which impacts the speed of exploration.

Training speed may be improved via two ways: 1) scale up, by making individual training nodes more powerful, and 2) scale out, by utilizing more hardware servers jointly training the same task. OpenAI has studied that over the six years between 2012 and 2018, the amount of compute increase due to Moore's law is only 8× (*i.e.* scale up), but the amount of compute increase due to parallelization is 37,500× (*i.e.* scale out) [42] [6]. Such a large increase is backed by heavy capital spending on the server clusters, which has already

[6] The contribution of scale up is probably underestimated. Between 2012 and 2018, CUDA cores were the main compute units in GPUs. Later, with the introduction of tensor cores, the peak compute is jumped several folds. Thus, architecture and microarchitecture changes may scale up the compute beyond Moore's law.

reached the point that few companies may afford to train ultra large models. Sparse models discussed in section 4.2 is an effort to address the training cost issue from the algorithm side.

Since the start of the current deep learning boom, many companies have built specialized hardware to speed up DNN model inference and training. The mainstream ones include GPU (NVIDIA, AMD), CPU (Intel, AMD), TPU (Google). A lot of startup companies explore some exotic architectures, hoping to cut a piece of the pie. Notable examples are: Tenstorrent [43], SambaNova [44], GraphCore [45], Cerebras [46], and Groq [47]. The successes of their architecture choices are yet to be proven by time.

For most hardware architectures, their design goal is to balance three aspects.

- Compute per processing element (PE). It sets the peak performance a PE may achieve. This timing the number of PEs in a chip is the peak performance of the chip.
- Bandwidth to/from PEs. It is the rate of feeding data to PEs, so that PEs may maintain their peak performance. Ideally, the data transfer time is entirely hidden by the compute time (*i.e.* in the compute bound region of a roofline model).
- Local memory size. It is used to hide the long latency of transferring data to PEs in some architectures (*e.g.* GPU), where the transfer local memory addresses are determined in advance. For some other architectures, local memories need to account for the latency variations between different computations. Memory hierarchy complicates the decisions.

The three aspects are constrained by chip area and total cost, and need to be coordinated to maximize the performance on a set of typical workloads. Among all the decisions, the choice of memory transfer mechanism may be the most important one: should the system primarily use a shared memory architecture with a unified address, or a message passing mechanism with multiple disjoint private addresses?

The current hardware architecture is designed for the current data intensive deep learning workloads. If, in the near future, the model paradigm is shifted to be composable, will we see design shifts in software and hardware as well?

6.1 Hardware challenges to support composition

We summarize the characteristics of composable models to the following three aspects:

1. A composable model is extremely large. When processing individual input, only a small fraction of the model is activated (*i.e.* sparsely activated). Since the activated portion is input dependent, besides the data path that processes the input, there exists a control path that determines which data path to take.

2. The stability of a composable model largely depends on its capability of constraining its component prediction errors within the component.

3. A composable model is unlikely to solve one task, it may take inputs from multiple modalities, and output solutions to multiple tasks. Thus, the components in the model may be shared across different tasks and are trained separately. In this way, the boundary of training and inference is blurred.

Aspect 3 is a training methodology and deployment change. It may be achieved via enhancing the deep learning framework software stacks. In addition, it may indicate that a unified hardware solution for training and inference is preferred, which means a quantized inference only hardware may not prevail.

The proposed solution to aspect 2 is neural symbolic computing (section 4.4). This is still a less explored area and the algorithms are in flux. Our current estimate is that such functionalities may be achieved via software enhancements for most part, supplemented by some hardware acceleration on specific computation.

Aspect 1 is relatively better thought through. It is also along the line of other researches [11, 22]. We will discuss the challenges around this aspect in the next section.

6.1.1 Hardware challenges for sparse computation

Exploring data locality is an important technique to get around the "memory wall" issue [48]. It is much easier to pile up PEs than bringing in data to them. Thus, computation is repeatedly done on the limited data already brought close to the PEs to increase compute utilization. It is also represented as the data reuse factor, OP/byte, in the roofline model.

Increasing batch size is a simple method to increase data reuse, since the same weights are computed on different input data. Thus, it is widely used in many dense DNN operator kernels (*e.g.* MM, CNN) on bandwidth bounded hardware (*e.g.* GPU). In a composed model, however, different inputs activate different components of the model. Packing those inputs to a minibatch no longer increases data reuse. Thus, a composed model is easier to be hit by the memory wall.

In the current systems training MoE models, such an issue is partially addressed by limiting the model and software flexibility: the number of experts and the amount of data parallelism match; the capacity of each expert is limited with the over-capacity tokens pruned; and the model computation is globally synchronized before and after the expert layers. Even with those changes, the MoE models are known to be inefficient on existing hardware.

We do not expect the current model and software patches for MoE models to be generalized to future composable models. Instead, we expect the future systems to be able to perform **single batch training** efficiently.

As a sparsely activated model requires a control path to route data to the activated components, the routing destination is unknown until the control module

finishes its computation. It may incur significant latency in a throughput optimized system. Such a long latency may translate to high local memory pressure. In addition, control path computation may include complicated data structures and random memory accesses, which is not suitable for PEs with wide data paths. As an example, the memory module (discussed in section 4.3) requires location based address calculation, and once the address is calculated, the corresponding memory entry is fetched or updated. That is a random access in a large array. Unlike the current embedding table accesses for recommendation models, which are clustered at the beginning of the model and can be performed in CPU, random accesses in memory may occur throughout model execution. Thus, native support for **latency optimized random accesses** is desired.

With single batch training and dynamic routing, the computation pattern is fragmented and unpredictable. The conventional static scheduling and synchronous programming model may not be efficient. It would be interesting to see whether a **dataflow architecture with asynchronous programming model** better suits this computation pattern.

6.2 Composition readiness on existing hardware

We project that the possible paradigm shift to composable models will trigger a series of enhancements in software, programming model, and hardware. Intentionally or not, some of the newer features in some mainstream hardware providers already embrace such changes. In the next few sections, we will discuss such features on selective hardware [7].

6.2.1 GPU

NVIDIA GPU is the de facto training platform for DNN models. Its massive number of cores and mature SIMT ecosystem fit the fine grained parallel computation pattern in deep learning models perfectly. It was an obvious choice against CPU in the early days. Over the years, NVIDIA GPU has included more features to strengthen its dominant position in deep learning.

The tensor core introduced in Volta architecture [49] and the sparse tensor core introduced in Ampere architecture [17] sped up matrix multiply-accumulation operation by several folds. However, it deviated from the SIMT programming model, separately implemented an ASIC block supported by several slightly more coarse grained instructions with stricter data access patterns. The performance gain far outweighed the little "inconvenience" in the more complicated programming model.

To speedup data transfer, per-thread asynchronous copy and asynchronous barrier mechanisms were introduced in Ampere architecture [17]. It was then enhanced in Hopper architecture [50] with block copying, electing one thread configuring a DMA like hardware. It further strayed from the SIMT model, making the programming more ASIC-like. The newly introduced thread block

[7] The discussion is based on our limited understanding of the hardware and application. Please critique and correct our mistakes.

clusters in Hopper architecture [50] maintained a unified shared memory address space within a cluster. One SM may access data from another SM directly, which further complicates programming targeting high performance.

Traditionally, GPU schedules one kernel after another. This is sufficient for large kernels that occupy all SMs for long periods of time. However, composable models may contain many small kernels due to single batch training, promoting the need to schedule multiple kernels concurrently. The multi-stream solution is known to be inefficient. Compiler solution based on persistent threads [51] only deals with static scheduling. The CUDA graph feature introduced by NVIDIA currently only targets reducing kernel launching overhead. But it will possibly be extended to perform inter-kernel optimizations in the future, which may alleviate the low utilization issue for small kernels.

The control paths in composable models contain complicated data structures and random memory accesses, which are inefficient in the current GPU architecture. However, if such computation is performed on CPU, the low bandwidth and long latency of PCIe channels become the bottleneck. The Grace Hopper superchip brings CPU and GPU next to each other. The increased bandwidth, reduced latency, and unified memory space all benefit composable model execution [8]. Nevertheless, it is unclear whether this type of coarse grained separation is sufficient for the future models. Or, such two types of computation are better implemented in the same hardware compute unit. More research on the model side may bring clarity to the algorithm needs.

NVIDIA GPU adopts a shared memory mechanism. But the separate address spaces in global memory and shared memory makes it possible to perform some message passing strategy. This hybrid approach incur significant hardware cost to cater both techniques. For example, the L2 cache may not be necessary for the message passing mechanism. It is natural to use shared memory for graphics workloads, but it is not as obvious for deep learning workloads.

On the GPU software programming side, many performance related decisions are passed to the programmers. It is relatively straightforward for the SIMT programming model, but shared memory and two address spaces complicates it. Features like tensor cores, async copies, and thread block clusters expect software programmers to be microarchitecture experts, which is unlikely for programmers in the community. Thus, most efficient kernels may only be written by NVIDIA engineers. They provide black box libraries and external engineers may only call those libraries, which may negatively affect the CUDA ecosystem.

6.2.2 CPU

CPU is specifically designed for control intensive applications. Many hardware features target on reducing latency instead of improving throughput. For the same reason, a large percentage of the core area is not related to either compute

[8] Bringing CPU and GPU closer benefits current DNN workload as well. It removes host CPUs from the system, increases data preprocessing (augmentation) pipeline throughput, and benefits models requiring large memory (*e.g.* recommendation models.)

or data movement. It results in relatively few cores in a chip and narrow parallel compute execution units in a core. Lacking sufficient compute capability is the main limitation of a CPU. To address this problem, the Sapphire Rapids processor from Intel introduced advanced matrix extensions (AMX) to speed up matrix matrix multiply accumulation operations. However, even with increased computation, it may be difficult for CPUs in the current architecture to achieve top performance.

First, all CPU cores in a chip are identical, adopting a symmetric multiprocessing (SMP) mechanism [52]. In this setting, all cores are symmetric, containing AMX. In the current CPU-GPU execution model, CPU is responsible for data loading, preprocessing, and preparing and issuing kernels to GPUs. When the kernel is executed in CPU with AMX, CPU still needs to spare a portion of the cores to do data loading and data preprocessing tasks. The AMX blocks in those cores are idle, which lowers utilization. Similar to GPU, SMP in CPU is more suitable to compute large kernels. The smaller kernel size in composable models may not utilize all cores. Software workarounds may be complicated and difficult to maintain.

Second, all cores share a unified memory address space. With a complicated memory hierarchy, hardware makes many data load, prefetch, and invalidation decisions. Those decisions are based on local context and may not be optimal. When those decisions are performed by software, features like cache line locking, software prefetch, and block prefetch are implemented. Message passing data transfer may also be emulated in software. In this aspect, the CPU provides all the necessary features to execute deep learning workloads, but they may not be natively designed for deep learning. It results in extra hardware overhead and more complicated software programming.

Third, the CPU's context switching overhead is significantly higher than the GPU's. With one or a few cycle context switching penalty, GPU may fill in any stall with independent computation in other contexts. In CPUs, however, bubbles due to long memory loads and control flow divergence are filled with computation in the same context via out-of-order execution or branch prediction. With regular access patterns in the deep learning workloads, such opportunities may be limited.

If we consider GPU an ideal platform for large and data intensive workloads, CPU goes to the other extreme, excelling control intensive workloads with limited available parallelism. The composable models fill in the middle, with a mixture of control flow and data flow computation. NVIDIA's Grace-Hopper architecture is an effort to cater both ends. But maybe it is more natural to design one architecture for both compute patterns. Research in this direction is highly desired.

6.2.3 TPU

TPU is the first commercially successful architecture designed for deep learning workloads from scratch [53, 54]. It is best known for its re-introduction of the systolic array [55] to compute matrix multiplication. The simple and ele-

gant hardware design moves much of the complexity to the compiler side: the large systolic array makes matrix tiling very difficult. It is challenging to find strategies to utilize all multiply accumulators.

With less data reuse in the composable models, TPU compiler's job may be insurmountable. The Pathways [23] partitions the sparsely activated models at the scheduling level. It deals with large components that can saturate the computation in one node. Data reuse may also be exploited at runtime with asynchronous computation: component runtime may collect sufficient input data dynamically before starting the computation.

Those methods are beneficial when the granularity of the sparsity is above a certain threshold, possibly at the component level. Sparsity may also be leveraged to speed up computation within a component. Such fine grained sparsity is difficult to exploit without a significant redesign of the systolic array architecture.

6.2.4 CGRA

Coarse grained reconfigurable array (CGRA) [56] is a non-processor-like architecture that intermixes small memories and arithmetic logic units (ALUs) in a chip. With each reconfiguration, data can be rerouted between memories and ALUs post silicon. The reconfiguration nature enables it to perform different computation by routing data differently. Compared with processors, it no longer needs instructions and the related instruction fetch/issue logic; replaces register files with pipeline registers and small memories; and packs more compute and higher bandwidth in a small chip area.

The conventional wisdom of exploiting data reuse in a shared memory processor is to build many layers of memory hierarchy, from DRAM, HBM, LLC, L2, L1/shared memory, to register files. The higher the level, the higher the bandwidth is. At the same time, the same data may be repeatedly loaded multiple times from the same source to the same destination in a temporal segmented manner. In comparison, a CGRA architecture loads data to different ALUs in a spatial manner and performs computation simultaneously. Data dependencies are ensured by routing data to different memories and different ALUs. In this sense, CGRA is a replacement of the blocks from L1 cache to register file to ALU, while leaving the lower levels untouched.

One challenge in the CGRA architecture is its compiler design. Computation at this granularity can be precisely determined cycle by cycle. A CGRA compiler needs to precisely determine which data flow to which ALU/memory at which cycle and perform which computation. It needs to build up the routing paths for all the data, containing numerous data joining and forking, constrained by the total area. In some cases, the stall logic to wait for data is not implemented, so any prediction error not only results in performance loss, but also means incorrect functionality.

Since all decisions are determined by a compiler ahead of the time, the computation is statically scheduled and cannot be adjusted based on runtime conditions dynamically. For composable models with dense components, some

external logic (*e.g.* CPU), may determine the datapath logic, and the CGRA only implements the logic. However, it creates a control-data dependency, with the data logic reconfigured based on the control output. Such reconfiguration may take thousands of cycles and may easily be the bottleneck. The static scheduling mechanism in CGRA also sets a lower bound for the components, making it suffer the same design challenge as TPU.

In addition, CGRA mainly optimizes on-chip data reuse, only indirectly influencing off-chip data transfers. One question to ask is whether the on-chip memory bandwidth is the bottleneck or the off-chip bandwidth is. The answer is not a clear cut, as it depends on chip size, microarchitecture design, and target application. But off-chip data transfer is orders of magnitude lower bandwidth, higher latency, and higher power. It is worthwhile to prioritize optimizing it.

SambaNova [44] is a rising star among AI startups based on CGRA architecture. Time will tell whether this choice has a definite advantage over a GPU or CPU plus accelerator architecture.

6.2.5 Tenstorrent like dataflow processors

Tenstorrent [43] is an AI startup led by the legendary Jim Keller. Prior to Tenstorrent, he was at Tesla, designing an AI training chip with a similar architecture: DOJO. Compared with other architectures, dataflow processors are better prepared for the future composable models. In the following, we will discuss their strength and compare DOJO and Tenstorrent processors.

One important design decision is to lower the shared memory to message passing boundary to the tile level. Shared memory excels at fine grained data accesses with hardware reactively or proactively initiating data transfers. Message passing defers all data transfer decisions to software, with hardware acceleration. It is especially efficient for burst data transfers in volume. By restricting shared memory accesses to within a PE, its memory hierarchy becomes very simple. At the global scale, the message passing architecture replicates data in a PE spatially and temporally, controlled by software. Thus, the area for lower level caches may be replaced with more PEs. This design choice reduces hardware complexity and cost. Its simple programming model also reduces software complexity: the same functionality may be achieved via fewer ways. Even with simplified hardware and software designs, the runtime performance may not be negatively affected.

Please note, GPUs are primarily a shared memory architecture, where multi-level large inclusive caches are preferred. Many features in recent generations also enable the software to perform message passing data transfers at the tile level. This hybrid solution, however, still wastes hardware area and complicates programming, which in our opinion is sub-optimal.

Message passing also simplifies interconnect design: network-on-chip (NoC) only needs to optimize for burst transfers. Standard network protocol is used in chip-to-chip communication. The same software stack can be used for any PE-to-PE communication, regardless whether the PEs are in the same chip or not. This is strikingly different from GPUs, whose intra-chip, inter-chip but

intra-server, and inter-server communication protocols are all different.

In the dataflow architecture, each chip contains hundreds of PEs connected via a mesh or folded torus topology. In DNN computation, one operator requires a cluster of PEs collaboratively producing the output. Each PE only computes a small tile of the output. Breaking down computation at tile level and communicating data via message passing effectively utilize the limited chip area to maximize computation.

If we consider CGRA optimizing for on-chip data reuse at extreme fine granularity, and GPU optimizing for very coarse grained large kernels on a single chip, this dataflow architecture optimizes off-chip data reuse for a cluster of chips. With a tight packing of the direct connected chips in a mesh topology, data can be transferred from one chip to the next without going through off-chip memories. In this way, if all the weights and activations can be flattened to the SRAMs of numerous PEs, entire training may be done on compute chips alone. Compared with CGRA, it may be able to reduce even more off-chip data accesses.

At this granularity, it is difficult to maintain cycle by cycle prediction of computation and data movement. Thus, solely relying on compilers to statically schedule operators without a stalling mechanism is not feasible. The scheduling burden is split between the compiler and firmware, with the compiler making best effort guesses and firmware adjusting dynamically based on runtime conditions. This significantly reduces its compiler's complexity compared with CGRA.

Optimizing computation at tile level is also friendly to the future composable models with sparsely activated modules. This fine grained computation significantly reduces the module size that may benefit from this computation pattern. It is small enough for efficient single batch training. Such a granularity may also help reduce the control-data latency in some scenarios. Asynchronous programming models may be used to adjust the dynamic nature of sparse workloads and further reduce compiler's performance prediction accuracy requirement.

The exact implementation inside a PE is not critical. Tenstorrent chooses a multi-core implementation augmented with dedicated ASIC engines. DOJO adopts a multi-thread processor with vector/matrix extensions. The Mozart from SimpleMachines [57] uses a CGRA to replace an ASIC. The implementation choice of a PE is encapsulated inside the PE. It is considered sufficient if the design spec is satisfied.

Tenstorrent and DOJO processors still have subtle architecture differences, mostly driven by their different target applications [9]. Tenstorrent targets general DNN applications while DOJO targets CV applications.

Within a PE, Tenstorrent chooses a multi-core architecture. Each core has its own process, which collaboratively computes one kernel, or independently computes different kernels. Even though it is unlikely to run multi-tenancy workloads in one PE simultaneously, it is relatively easy to overlap compu-

[9] This comparison is based on limited public information, which may be significantly deviated from reality.

tation for multiple micro-batches. Since each process is single thread and is responsible for one dedicated micro-kernel, its programming model is straightforward. On the other hand, DOJO chooses a multi-thread architecture. This is sufficient for single training, but would be challenging for multiple independent workloads. Mixing multiple collaborating micro-kernels in one pipeline may be too fine grained, possibly making the instruction fetch the bottleneck. Its design complexity for both hardware and software may be higher than a multi-core architecture.

One surprising decision made by DOJO is that its chip does not connect to any DRAM. Instead, 25 chips from an MCM and three MCMs stack together before accessing the nearest DRAM. Such a long distance to DRAM means SRAM needs to be large enough to store all weights and activations. It is unclear whether it is large enough to hold them across forward and backward passes, considering many micro-batches in a mini-batch. Alternatively, they may still flow to DRAMs between passes, but that creates very unbalanced traffic. PEs closer to DRAMs require much higher bandwidth while PEs far away from DRAMs need less bandwidth. This design decision seems to be more appealing to CNN based models, where the size of the weights and activations is relatively small compared with the amount of compute. However, as matrix multiplication based transformer models become more popular recently, it is unclear whether that creates additional bottlenecks in this architecture [10].

On the other hand, each Tenstorrent chip connects to large DRAMs in addition to connecting to other Tenstorrent chips. In this way, the weights and activations may be flowed to DRAMs between passes without incurring unbalanced traffic. It is a more conservative design that may be suitable for more applications. The price to pay is the chip area dedicated for DDR controllers, which is not negligible.

A Tenstorrent chip also contains several high performance cores outside its PE mesh. The purpose of them is to cover uncommon operators, whose computation is not implemented in the PEs. Some operators need to manipulate tensor shapes in a way difficult on PEs and existing tiling patterns. The high performance cores provide a back door to use the shared memory method to implement them. This way, PEs may only cover frequent operators.

In contrast, a DOJO chip is exclusively composed of the same PEs. Thus, a lot more PEs may be squeezed in the same chip area and increasing parallelism. However, each PE needs to effectively compute more operators, whose area is larger. This decision is also due to the decision of not connecting DOJO chips to DRAMs. After all, the difference in the target applications determines those design choices.

Compared with other hardware solutions, Tenstorrent processors may achieve the state of the art performance on existing workloads, though consume less power due to more efficient dataflow management. Its architecture is even better prepared for future composable workloads with dynamic sparsity, at least from the information collected so far. We hope more research is devoted to

[10] This analysis may be completely off. We will update it when new information is available.

dataflow processors, targeting composable models.

7 Call for Research on Composable Models

In this article, we explain our view to approach an essential technical capability in the path towards artificial general intelligence: the ability to decompose a large model to several smaller independently solvable models. We reason the importance of composition, and divide the technical advancements to three aspects: algorithm, data, and compute. These three aspects are intertwined and need to advance coordinately.

Full understanding of composition is challenging and may take many years of research. We describe several research directions in algorithms and compute aspects that we consider important, as first steps to apprehend the potential of composition and explore its methodologies.

In recent years, many researchers have started looking into those areas and have made many achievements. Yet many questions remain unanswered. We call for more research along the line of composition, regardless whether the directions are mentioned in this article or not. We hope, in three to five years, composable models will become a popular model choice. This model paradigm shift will lead to significant improvements on the software and hardware side. That will continue driving our current third AI wave, or even better, start a new AI boom.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Conf. Neural Information Processing Systems*, Dec. 2012, pp. 1097–1105.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Proc. Conf. Neural Information Processing Systems*, vol. 30, 2017.

[4] OpenAI, "Ai and compute," 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/

[5] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[7] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[8] Wikipedia, "Function composition," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Function_composition

[9] ——, "Analog computer," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Object_composition

[10] ——, "Object composition," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Analog_computer

[11] Y. LeCun, "A path towards autonomous machine intelligence," Jun. 2022.

[12] Wikipedia, "Dennard scaling," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Dennard_scaling

[13] VentureBeat, "OpenAI's massive GPT-3 model is impressive, but size isn't everything," 2020. [Online]. Available: https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/

[14] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *Proc. Int. Conf. Learning Representations*, 2017.

[15] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *J. Machine Learning Research*, 2022.

[16] J. Lin, R. Men, A. Yang, C. Zhou, M. Ding, Y. Zhang, P. Wang, A. Wang, L. Jiang, X. Jia *et al.*, "M6: A chinese multimodal pretrainer," *arXiv preprint arXiv:2103.00823*, 2021.

[17] NVIDIA, "NVIDIA A100 tensor core GPU architecture," Tech. Rep., 2020.

[18] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.

[19] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," in *Proc. Nature Communications*, vol. 9, no. 1, 2018, pp. 1–12.

[20] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in *Proc. Int. Conf. Machine Learning*, 2020, pp. 2943–2952.

[21] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 2021.

[22] J. Dean, "The deep learning revolution and its implications for computer architecture and chip design," in *Int. Solid-State Circuits Conf.*, 2020, pp. 8–14.

[23] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy *et al.*, "Pathways: Asynchronous distributed dataflow for ml," *Proc. Machine Learning and Systems*, vol. 4, pp. 430–449, 2022.

[24] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. v. d. Driessche, J.-B. Lespiau, B. Damoc, A. Clark *et al.*, "Improving language models by retrieving from trillions of tokens," *arXiv preprint arXiv:2112.04426*, 2021.

[25] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[26] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, pp. 471–476, 2016.

[27] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014.

[28] S. Sukhbaatar, J. Weston, R. Fergus *et al.*, "End-to-end memory networks," *Proc. Conf. Neural Information Processing Systems*, vol. 28, 2015.

[29] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long range arena: A benchmark for efficient transformers," *arXiv preprint arXiv:2011.04006*, 2020.

[30] Y. Shen, S. Tan, A. Hosseini, Z. Lin, A. Sordoni, and A. C. Courville, "Ordered memory," *Proc. Conf. Neural Information Processing Systems*.

[31] D. Tanneberg, E. Rueckert, and J. Peters, "Evolutionary training and abstraction yields algorithmic generalization of neural computers," *Proc. Nature Machine Intelligence*, vol. 2, no. 12, pp. 753–763, 2020.

[32] G. Marcus, "Deep learning is hitting a wall," 2022. [Online]. Available: https://nautil.us/deep-learning-is-hitting-a-wall-14467/

[33] Y. LeCun and J. Browning, "What AI can tell us about intelligence," 2022. [Online]. Available: https://www.noemamag.com/what-ai-can-tell-us-about-intelligence/

[34] A. d. Garcez, M. Gori, L. C. Lamb, L. Serafini, M. Spranger, and S. N. Tran, "Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning," *arXiv preprint arXiv:1905.06088*, 2019.

[35] A. d. Garcez and L. C. Lamb, "Neurosymbolic ai: the 3rd wave," *arXiv preprint arXiv:2012.05876*, 2020.

[36] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Neural module networks," in *Proc. Conf. Computer Vision and Pattern Recognition*, 2016, pp. 39–48.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. Conf. Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[38] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling vision transformers," in *Proc. Conf. Computer Vision and Pattern Recognition*, 2022, pp. 12 104–12 113.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[40] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *Proc. Int. Conf. Machine Learning*, 2021, pp. 8748–8763.

[41] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents," *arXiv preprint arXiv:2204.06125*, 2022.

[42] D. Hernandez and T. B. Brown, "Measuring the algorithmic efficiency of neural networks," *arXiv preprint arXiv:2005.04305*, 2020.

[43] Tenstorrent, https://tenstorrent.com/.

[44] SambaNova, https://sambanova.ai/.

[45] Graphcore, https://www.graphcore.ai/.

[46] Cerebras, https://www.cerebras.net/.

[47] Groq, https://groq.com/.

[48] Wikipedia, "Memory wall," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Random-access_memory#Memory_wall

[49] NVIDIA, "Nvidia Tesla V100 GPU architecture," Tech. Rep., 2017.

[50] ——, "NVIDIA H100 tensor core GPU architecture," Tech. Rep., 2022.

[51] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}," in *Symp. Operating Systems Design and Implementation*, 2020, pp. 881–897.

[52] Wikipedia, "Symmetric multiprocessing," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Symmetric_multiprocessing

[53] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. Int. Symp. Computer Architecture*, 2017, pp. 1–12.

[54] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: TPUv2 and TPUv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[55] Wikipedia, "Systolic array," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Systolic_array

[56] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[57] K. Sankaralingam, T. Nowatzki, V. Gangadhar, P. Shah, M. Davies, W. Galliher, Z. Guo, J. Khare, D. Vijay, P. Palamuttam *et al.*, "The mozart reuse exposed dataflow processor for ai and beyond," in *Proc. Int. Symp. Computer Architecture*, 2022.