SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2023, Vol. 66 192102:1–192102:17
https://doi.org/10.1007/s11432-021-3596-x

Rescue to the *Curse* of universality

Yongwei ZHAO[1], Zidong DU[1,3], Qi GUO[1], Zhiwei XU[1,2] & Yunji CHEN[1,2*]

[1]*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190*, China;*
[2]*University of Chinese Academy of Sciences, Beijing* 100049*, China;*
[3]*Beijing Academy of Artificial Intelligence, Beijing* 100084*, China*

Received 21 October 2021/Revised 1 June 2022/Accepted 9 October 2022/Published online 2 August 2023

Abstract From the very beginning of computers, universality has been the core focus in the building of computing machines, such as the universal Turing machine, von Neumann architecture, random-access machines, and universal circuits. Academia has taken universality as the primary principle ever since. However, the *Curse* of universality, implied from L. G. Valiant's Universal Circuit, states that computers based on logic circuits cannot be both universal and efficient, as the cost of universality is $\Omega(n \log_2 n)$. Though the *Curse* has been hidden by the rapid advancement of semiconductor technologies, it has been wielding its effects noticeably in recent years. Due to the ending of Dennard scaling and Moore's law, general-purpose processors leave less room for improvement. Therefore, domain-specific architectures (DSAs), such as deep learning processors, have been exploding, leading to the new golden age of computer architectures. For DSAs, universality is traded off for optimal efficiency. However, we predict that universality will once again be a major concern for post-golden-age computers. In this paper, we discuss how much universality could an efficient computer keep. As a rescue to the *Curse*, we define and discuss quasi-universal architectures. Quasi-universal architectures can solve any computable problem and are efficient for a wide range of problems. The proposed Recursive-Encapsulated (RENC) architecture achieves maximal universality while keeping optimal efficiency as found in specialized architectures. The discovery of RENC suggests that current golden-age architectures are not Pareto optimal.

Keywords universality, general-purpose architecture, specialized architecture, deep learning processor, universal circuit

Citation Zhao Y W, Du Z D, Guo Q, et al. Rescue to the curse of universality. Sci China Inf Sci, 2023, 66(9): 192102, https://doi.org/10.1007/s11432-021-3596-x

1 Introduction

High-level, domain-specific languages and architectures, freeing architects from the chains of proprietary instruction sets · · · will usher in a new golden age for computer architects.
——*John L. Hennessy and David A. Patterson, Turing lecture [1]*

From the very beginning of computer science, universality has been the primary principle for building computing machines. In 1937, Turing [2] constructed the universal Turing machine to show that there are undecidable problems. In 1945, von Neumann [3] summarized the stored-program principle for designing variable (universal) computers. In 1976, Valiant [4] proposed the universal circuits, whose cost is optimal to the informational lower bound. The philosophy behind, i.e., one machine for all tasks, is driving computer scientists to pursue universality.

Universality is a *Curse*, nevertheless, since it contradicts efficiency. The contradiction is well-defined under the circuit model by Valiant's aforementioned work: due to the informational bound, there is an inevitable efficiency gap between the specialized circuits and the universal circuit, namely the *Curse* of universality that is discussed in this paper. However, the *Curse* had been hidden for several decades. As predicted under the Dennard scaling and Moore's law, the performance of computers was increasing rapidly even without much architectural optimization thanks to the advances in semiconductor technology.

* Corresponding author (email: cyj@ict.ac.cn)

Therefore, computer scientists reached a universality-first consensus, where universality comes first, and then efficiency is discussed and improved.

Eventually, most of the low-hanging fruits implied from Dennard's scaling and Moore's law have been picked. In this century, computer scientists have proposed several techniques to maintain the rate of improvement in efficiency. In 2003, the frequency of processors stopped growing and parallel computing became popular. In 2010, the multicore scaling also flattened due to the Dark Silicon [5] and heterogeneous computing (e.g., GPGPU) was favored. However, the GPU also hit walls: the NVIDIA CUDA cores increased $\sim 70\%$/year in the five years before 2014, but the ratio flattened to $\sim 9\%$/year in the five years after 2014.

The *Curse* prominently wields its effects nowadays, at which time computer scientists are forced to trade universality for efficiency. Domain-specific architectures (DSAs), rather than general-purpose ones, have been extensively studied in recent studies. In 2014, researchers proposed DianNao [6] to accelerate deep neural networks, providing $\sim 100\times$ speedup and $\sim 20\times$ energy savings over CPUs. In 2017, NVIDIA released the Volta architecture with TensorCores, substantially increasing the GPU peak performance from 18.7 to 125 TFlops under deep learning scenarios. Due to the great explosion of deep learning applications, a variety of deep learning processors (DLPs) are proposed, racing towards the extreme of efficiency. After discarding the yoke from universality requirements, the computer scientists opened a new golden age, according to Patterson and Hennessy's Turing lecture [1] in 2017.

It is not a secret that universality contradicts efficiency. Trade-offs were extensively studied in the old age, when computer scientists asked how efficient could a universal computer be. Then the new golden age researchers need to understand how efficient could a computer be, even if not necessarily universal. There is hardly any study discussing how much universality could an efficient computer keep, from the perspective of efficiency-first, i.e., efficiency comes first, and then universality is discussed and improved.

This paper discusses such trade-offs from the aforementioned perspective. As a rescue to the *Curse*, we define several quasi-universal architectures under the circuit model, including typical old-age/new-golden-age architectures. New efficient architectures with better universality have been discovered, hence current new-golden-age architectures are not Pareto optimal. The main result of this paper is threefold:

• We clarify the *Curse* of universality as an important challenge to the post-golden-age computer scientists, as universality will once again become a major concern in the new age.

• We formally define the quasi-universal architectures, which is somewhere in between the universal architectures and the specialized architectures. Several quasi-universal architectures are defined and discussed in terms of implementation costs and computational powers.

• Among the discussed quasi-universal architectures, we come to our solution: the recursive-encapsulated (RENC) architecture, which is efficient, and universal to the maximum extent. We find that many practical algorithms used in real-life computers are RENC compatible, thus efficiently computable in the proposed RENC architecture.

In this study, we also argue that the future of computer architectures, including both the general-purpose architectures (e.g., GPGPU) and the domain-specific architectures (e.g., DLP), should pay more attention to the efficiency-first perspective.

2 Preliminaries

2.1 Notation

Asymptotic notations. As asymptotic notations have ambiguous definitions by different scholars, we follow the convention of Knuth [7] of "Ω" in this paper. Precisely, for two functions $f : \mathbb{N} \mapsto \mathbb{N}$ and $g : \mathbb{N} \mapsto \mathbb{N}$, we use the following definitions.

• $f(n) = o(g(n))$ means for every $\epsilon > 0$, there exists a constant N such that $\forall n \geqslant N$, $f(n) \leqslant \epsilon g(n)$.

• $f(n) = O(g(n))$ means there exist $\epsilon > 0$ and a constant N, such that $\forall n \geqslant N$, $f(n) \leqslant \epsilon g(n)$.

• $f(n) = \omega(g(n))$ means for every $\epsilon > 0$, there exists a constant N such that $\forall n \geqslant N$, $f(n) > \epsilon g(n)$, i.e., $g(n) = o(f(n))$.

• $f(n) = \Omega(g(n))$ means there exist $\epsilon > 0$ and a constant N, such that $\forall n \geqslant N$, $f(n) \geqslant \epsilon g(n)$, i.e., $g(n) = O(f(n))$.

• $f(n) = \Theta(g(n)) \Leftrightarrow$ both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold.

Table 1 Symbols used in this paper

Symbol	Definition	Symbol	Definition
G	A graph	X	Input bits $x_0x_1\cdots x_{s-1}$
V	A set of vertices	Y	Output bits $y_0y_1\cdots y_{d-1}$
E	A set of edges	Prog	Programming bits $p_0p_1\cdots p_{n-1}$
C	A circuit	v	A function, the number of gates in each capsule
$\{C_0, C_1\}$	A family of circuits[a)]	w	A function, the number of terminals in each capsule
$C \overset{u}{\twoheadrightarrow} C_*$	Circuit C is universal to the family C_*	Γ^n	The universal family, $\Gamma^n = \{C \mid \lvert C\rvert \leqslant n\}$
$\lvert C\rvert$	The size of circuit C	U_n	Edge universal graph of size n
n	An independent variable, $n \in \mathbb{N}$	UC_n	Universal circuit of size n
s	Number of input bits	A	An architecture (Definition 2)*
d	Number of output bits	I	Implementation of the architecture, $I_n \overset{u}{\twoheadrightarrow} A^n$
Ω_C	The basis of circuit/family C	R	A reference circuit family
Ω_0	The standard basis $\{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$	$\textsc{Static}_C$	The Static architecture (Definition 6)
P	A logic predicate	$\textsc{Packed}_w$	The Packed architecture (Definition 7)
$\top$	Tautology, a predicate that always holds	$\textsc{Slacked}_{v,w,R}$	The Slacked architecture (Definition 9)
S_R	The static predicate (Definition 5)	$\textsc{Enc}_{v,w}$	The Encapsulated architecture (Definition 10)
$K_{v,w,P}$	The capsule predicate (Definition 8)	$\textsc{REnc}_{v,w}$	The Recursive-Encapsulated architecture (Definition 11)

a) We number the circuits in a circuit family with subscripts, and number the circuit families in an architecture with superscripts.

Iterated function. We define iterated function f^* as the number of times, and the function $f : \mathbb{N} \mapsto \mathbb{N}$ must be iteratively applied to reduce the result less or equal to 1. A formal definition could be given in recurrence relation:

$$f^*(n) = \begin{cases} 0, & \text{if } n \leqslant 1; \\ f^*(f(n)) + 1, & \text{otherwise.} \end{cases} \tag{1}$$

We list common correspondences between f and f^*: $f^*(n) = n - 1$ when $f(n) = n - 1$, $f^*(n) = \log_2 n$ when $f(n) = \frac{n}{2}$, $f^*(n) = \log_2 \log_2 n$ when $f(n) = \sqrt{n}$, and $f^*(n) = \log^* n$ when $f(n) = \log n$.

2.2 Circuit model

Definition. Circuits can be taken as directed acyclic graphs (DAGs), where the input and output vertices carry the labels of Boolean variables, and other vertices carry the labels of gates [8]. For a circuit C, its corresponding DAG can be represented as $G = \langle V, E\rangle$, where the meanings of symbols are explained in Table 1. For conciseness, we also write $C = \langle V, E\rangle$ alternatively. Specifically, we have the following definitions.

• **Inputs.** The inputs are the s vertices without incoming edges. Input variables are labeled on these vertices. Each input variable represents one bit of signals fed into the circuit.

• **Outputs.** The outputs are the d vertices without outgoing edges. Output variables are labeled on these vertices which the result bits of the circuit are assigned to. The output vertices may also carry gates.

• **Gates.** The gates are all the vertices except inputs. Types of gates are labeled on these vertices. Each gate represents a logic function computed over the values from incoming edges.

• **Basis.** The basis of the circuit C, denoted by Ω_C, is the set of gate types used in the circuit DAGs. Unless otherwise specified, we assume the standard basis is used, i.e., $\Omega_0 = \{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$ and a degree (maximal gate fan-in and fan-out) of 2.

• **Size.** The size of the circuit C, denoted by $|C|$, is the number of vertices $|V|$.

Please note that the size of a circuit C can be taken as an energy measurement of real machines. Roughly, the energy of a real digital circuit is largely decided by the energy costs of gate charging/discharging, which is proportional to the number of gates ($|C|$). For a combinational circuit, its energy is directly proportional to $|C|$. For a sequential circuit, we first convert it into a combinational circuit C by repeating the combinational part $\hat{C}$ in the sequential circut running cycle ($N_{\text{cycles}} < \infty$) times. Thus, the energy can be estimated with Energy $\sim |C| \sim N_{\text{cycles}} \times |\hat{C}|$.

Power of circuits. Unlike other universal computing models such as finite-state-machine (FSM) or Turing machines, circuit model is non-uniform. Precisely, for the same problem, the circuit model builds individual circuits C_n for each size n of the problem. Therefore, a family of circuits $\{C_1, C_2, \ldots\}$ is built

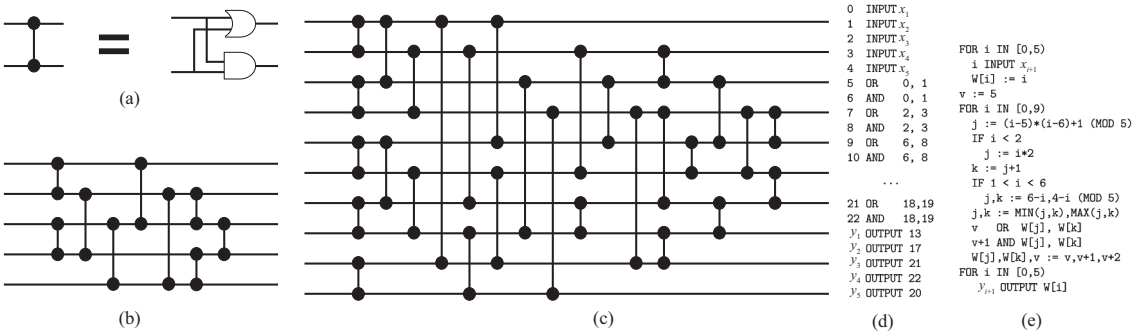

Figure 1 Sorting networks as examples of the circuit model. (a) `SWAP` comparator built over the standard basis Ω_0; (b) size-specialized circuit for 5 inputs, where each vertical line indicates a `SWAP` comparator; (c) size-specialized circuit for 10 inputs; (d) corresponding straight-line program description for (b); (e) branching program description of the same circuit as (b); after the execution of (e), (d) is instantiated.

for solving one problem. Specially, the family of all circuits under the size limit n is defined as Γ^n, i.e., $\Gamma^n = \{C \,|\, |C| \leqslant n\}$.

For example, for the Sorting problem, the SortingNetwork circuit family is built to address the same problem but with different input sizes. Figures 1(b) and (c) show the individual sorting network circuits for input sizes 5 and 10, respectively, where the vertical lines denote the 2-input, 2-output `SWAP` comparators as shown in Figure 1(a).

The non-uniformity suggests that the circuit model is more powerful than Turing machines, and the latter must decide the problem of any size in one instance. Given circuits of size n, let $\mathsf{SIZE}(n)$ denote the decidable problem set of the circuit model. In fact, any decision problem of n input bits is in $\mathsf{SIZE}(O(2^n/n))$, including those even undecidable by Turing machines (e.g., the Halting problem). For more details, please refer to Example 6.4 in [9].

Descriptions. To describe a circuit, people write programs. Every circuit has a corresponding straight-line program describing it [8]. A straight-line program is a sequence of computing steps. Each step is formatted as (y `OP` $x_1, \ldots x_k$), where `OP` identifies the operation (the type of a gate) performed on variables $x_1, \ldots x_k$, and y defines a new variable which stores the result of the current computing step. An example of converting the circuit of Figure 1(b) into a straight-line program is shown in Figure 1(d). The number of lines of a straight-line program always equals the number of vertices in the circuit graph. Therefore, the size of the straight-line program would be $O(n \log_2 n)$ bits, since each of the computing steps requires at least $k \log_2 n + O(1)$ bits to encode.

The straight-line programs are not convenient for describing large but regular circuits, since the program must write out every gate even if the gates are repeatedly constructed in trivial patterns. To describe large circuits in short notations, people write branching programs. Branching programs have high-level control flows (variables, branches, and loops), take no inputs (or only a constant number of inputs, e.g., the problem size), and instantiate a straight-line program after execution. A branching program for the circuit in Figure 1(b) is shown in Figure 1(e).

Architecture. In computer science, the architecture is often viewed as a set of constraints. We adopt this perspective. In this paper, since we are discussing the circuit model, the word "architecture" is interchangeable with "the circuit family following a set of constraints". The formal definition comes in Subsection 3.4.

2.3 Universal circuit

Definition of the universal circuit (UC). A circuit C is universal to a circuit family $\{C_0, C_1, \ldots\}$, denoted by $C \overset{\boldsymbol{u}}{\rightarrow} \{C_0, C_1, \ldots\}$, if C can simulate the function of any circuit in the family $\{C_0, C_1, \ldots\}$, by properly assigning input/output variables. Let C_i denote the circuit under simulation. The circuit C receives a specification of the circuit under simulation ($\mathrm{Prog}(C_i) = p_0 p_1 p_2 \cdots p_{n-1}$) and the input bits to the circuit under simulation ($X = x_0 x_1 x_2 \cdots x_{s-1}$), and then outputs the output bits of the circuit under simulation ($Y = y_0 y_1 y_2 \cdots y_{d-1}$), i.e., $\forall C_i \in \{C_0, C_1, \ldots\}, C(\mathrm{Prog}(C_i), X) \equiv C_i(X)$. $\mathrm{Prog}(C_i)$ is called the programming bits of C_i as it programs the circuit under simulation C_i. Specially, universal circuit UC_n is a circuit that can simulate the function of any circuit with a size up to n, i.e., $\mathrm{UC}_n \overset{\boldsymbol{u}}{\rightarrow} \Gamma^n$. From the perspective of the graph theory, we have the following definitions.

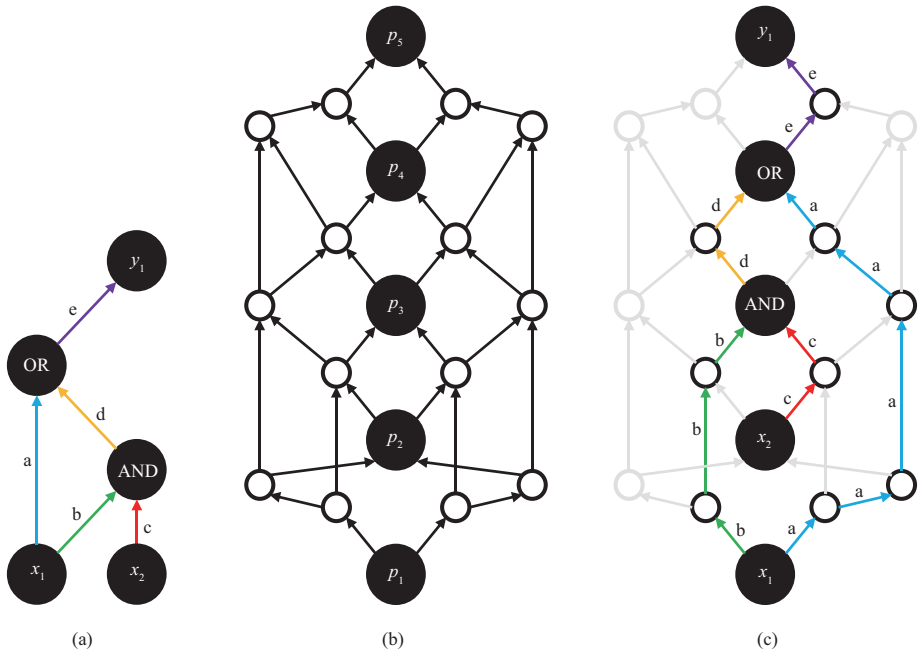

Figure 2 (Color online) Universal circuit construction by the method of Valiant [4, 10]. (a) An example circuit C with 2 inputs, 1 output, and 2 gates. (b) U_5, where the filled nodes denote poles, the small circles denote switches. (c) Embedding C into U_5. When universal gates are placed at poles, the universal circuit is constructed.

• **Γ^n.** Γ^n is the set of all directed acyclic graphs with up to n vertices, where inputs, outputs, and gates are arbitrarily labeled, and at most 2 incoming edges and 2 outgoing edges are allowed for each vertice.

• **Edge-embedding.** Edge-embedding is a mapping from directed acyclic graph $G = \langle V, E\rangle$ into $G' = \langle V', E'\rangle$, where V is mapped into V' one-to-one, E is mapped into directed paths in E' (all paths are pairwise edge-disjoint).

• **Edge-universal.** A graph U_n is an edge-universal graph (EUG) for Γ^n, if every graph $G = \langle V, E\rangle$ in Γ^n can be edge-embedded into U_n.

• **Universal circuit.** (UC is a logic circuit that can be programmed to compute any circuit up to a given size n by defining a set of programming bits Prog, i.e., $\forall C \in \Gamma^n, \mathrm{UC}_n\left(\mathrm{Prog}(C), X\right) \equiv C(X)$.

UC construction. Figure 2 shows the method proposed by Valiant to build an universal circuit [4]. We explain the procedure of construction using the same simple circuit as a driving example [10]. The example circuit is shown in Figure 2(a), which has two inputs, one output, and two gates.

A minimalist EUG, U_5, is shown in Figure 2(b). The EUG graph will be further translated into UC_5 by placing universal gates (i.e., the universal 2-to-1 circuits) at poles. By feeding programming bits to the switches and universal gates, any circuit in Γ^5 with the same number of inputs and outputs can be embedded into the constructed UC. As a driving example, the embedding of the example circuit is shown in Figure 2(c). Larger EUGs are built recursively on smaller EUGs. Please refer to [4, 10–12] for more details.

3 The *Curse* of universality

3.1 A brief history of universal computers

In the 1930s, Church and Turing laid out the fundamental definition of computation. λ-calculus and Turing machine are designed to be general abstractions for any machinery computation. Turing first used the concept of the universal Turing machine (UTM) to show that there are undecidable problems. By definition, UTM is general to any Turing machine, just as a universal circuit is to any circuit: the specifications and inputs of the Turing machine are fed to the UTM, and the UTM simulates that Turing machine. So any machinery computation can be performed by the same UTM, just given different specifications of Turing machines, which are called programs today. In the subsequent practice of building

real computers, e.g., EDVAC, von Neumann summarized the von Neumann architecture, which is still the golden model for today's computer technologies. The first principle of the von Neumann architecture is "stored-program", which makes the machine an imitation of UTM. However, von Neumann giveth, von Neumann taketh away. The inefficiency of such computers is also named after him, i.e., the von Neumann Bottleneck [13], which states the relative inability of data accesses brought about by the von Neumann's programming style.

A more commonly used general-purpose computing model is the random access machine (RAM). At each step of the computation, the RAM model allows one or more accesses to arbitrary locations in memory. To implement a RAM with n-bit storage in logic circuits, each of the n-bit cells must be connected to some gates to be accessible, implying a $O(n)$ energy cost per computational step. Therefore, the energy cost of the RAM model is much higher. For example, comparison-based sorting of n elements requires $\Theta(n \log_2 n)$ steps under the RAM model, so the energy cost will be $O(n^2 \log_2 n)$, which is very inefficient.

There are more efficient computing models and computers. Since we are concerned with the efficiency of the computation (in terms of energy cost), we discuss the circuit model because it directly reflects the cost of its size. The computers under this model are also proposed. In 1979, Kung [14] implemented algorithms directly on very large scale integrated (VLSI) circuits, namely systolic arrays, which have recently gained renewed attention for their emerging applications in deep learning. As UTM and RAM, it is also possible to build circuits that are universal to any circuit up to size n, such as the circuit that solves the CircuitValueProblem (CVP, a problem well known for its P-completeness). In terms of computational power, n-universal circuits can solve any problem in $\mathsf{SIZE}(n)$.

In 1976, Valiant proposed an asymptotically size-optimal construction for universal circuits of size $O(n \log_2 n)$ [4]. Valiant's universal circuit can simulate the functionality of any circuit up to n in size by embedding the circuit's wires (edges) and gates (vertices) into EUGs. Other following studies keep optimizing Valiant's construction in sizes [10,15,16]. However, the construction given by Valiant is already optimal within a constant multiplicative factor of the informational lower bound. It can be seen as the optimal computing architecture implementing universality, despite the constant factor hidden in big-O is continuously improved.

We select the universal circuit as the representative universal computer in this paper, because its proven optimal efficiency is superior to other universal computers: UTM, RAM, etc. Note that being efficient is different from being simple. There are extremely simple abstract machines and dynamic systems that prove to be universal [17], built with just one instruction or a few simple state transition rules. However, these machines are often inefficient. They take much more computing steps to simulate necessary functionalities, resulting in a high power-delay-product (energy cost) when implemented in logic circuits.

3.2 The *Curse*

Figure 2 illustrates how the selected universal computer (i.e., Valiant's UC) works conceptually. In his original 1976 paper, Valiant was optimistic about the ability to construct a universal circuit of size only $O(n \log_2 n)$, which is quite efficient, considering the naive approach of putting all possible circuits together without structural sharing costs roughly $\Omega(n!)$.

In this paper, however, we take a pessimistic view. We illustrate our point with an intuitive example: Figure 3 implements Valiant's UC in a logic circuit diagram, together with the example circuit in Figure 2(a) with two gates, as shown in the upper right box. Note how an incredible number of gates are paid to universally embed circuits with at most two gates. We annotate the gates corresponding to the function of the original gates from the upper right box, and all other gates are just (as P. Kelly phrased) Turing tariffs [18] — excessive costs paid for universality.

This example shows a huge constant factor hidden in Valiant's big-O notation. The real problem, however, is that the proportion of Turing tariffs grows with size, as UC scales asymptotically worse than its specialized counterparts, i.e., $\Theta(n \log_2 n)$ vs. n. There is a huge performance-per-Watt headroom in the exa-scale, post-Moore's-law era [19], which was paid for unnecessary universalities.

UC is already an optimal universal computer, and its asymptotic size cannot be further improved. An informational lower bound can simply be derived from the uncertainty of the circuits to be embedded. For a random circuit with n gates, each gate provides a new variable for the input to the subsequent gates, so there are more than $\Omega(n!)$ circuits to be embedded. UC must first decide which to embed,

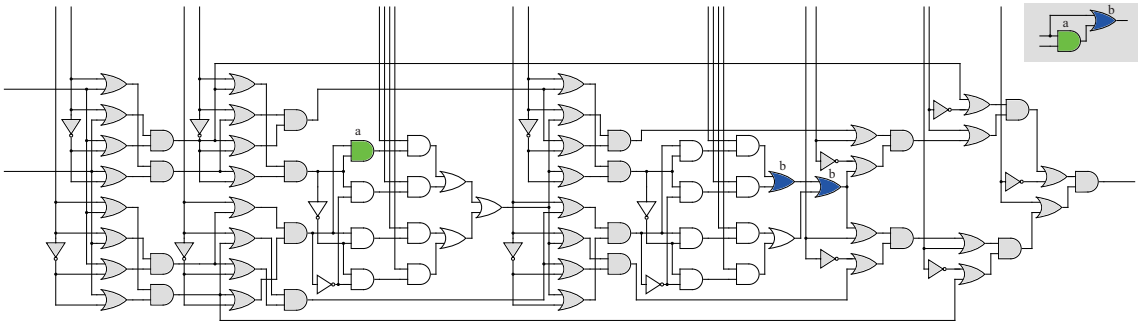

Figure 3 (Color online) Implementation of UC_5 ($s = 2$, $d = 1$) using Valiant's construction, embedding the example from Figure 2(a) shown in the upper right box. $X = x_0x_1$ is from the left and $\mathrm{Prog} = p_0p_1 \cdots p_{18}$ is from the top, $Y = y_0$ is to the right. Gates forwarding bits are annotated in grey, while others constitute universal gates. The gates performing the actual computation from the original circuit (gate a and gate b) are annotated in colors.

encoding possible circuits requires $\log_2 \Omega(n!) \simeq \Omega(n \log_2 n)$ bits of information, and each bit has to be connected to some gates. Therefore, the size of UC is lower-bounded by $\Omega(n \log_2 n)$, which cannot be improved asymptotically.

As analyzed above, computing functions through UC is much less efficient than using specialized circuits. This gap between UC and specialized circuits is due to the reduction in informational entropy required to implement universality, and is inevitable for any architecture that implements universality (not just for Valiant's UC), hence the term "the *Curse* of universality". The *Curse* implies that it is impossible to build a computer in logic circuits that is both efficient and universal. By saying "universal", it means "able to compute any problem in $\mathsf{SIZE}(n)$"; by saying "efficient", it means "cost-optimal within a constant multiplicative factor of n".

Definition 1 (The *Curse*)**.** It is impossible to build a computer in logic circuits that is both universal and efficient.

3.3 The unnecessity in the universality

The main reason for the *Curse* is unnecessary universality. Roughly speaking, the universality of the previous definition requires that the circuit C implements all the functions of the circuits in Γ^n, i.e., $C \twoheadrightarrow \Gamma^n = \{C' \mid |C'| \leqslant n\}$. However, we argue that most circuits in Γ^n are not necessary, since they are usually not produced by human.

From a programming point of view, most circuits are not programmable; i.e., they cannot be built by writing short branching programs because there are not enough short programs. Here a short program is defined as having a length within a constant multiple of the logarithmic circuit size, i.e., $c \log_2 n$ bits, where c is a constant factor and n is the circuit size. We always assume that programs written by human are short and within a constant length in terms of lines of source code. Since each step of the straight-line program requires $k \log_2 n + O(1)$ bits to encode, the size requirement of the short program we define is set to $c \log_2 n$ bits (approximately corresponding to c/k lines of source code). We formalize the argument as follows.

Theorem 1 (Almost all circuits are not programmable)**.** The fraction of circuits in Γ^n that can be described by short programs is at most $2^{-n+1} n^{c+2} n!^{-2}$, where c is a constant factor.

Proof. (1) The total number of circuits. Each computing step in a straight-line program (over the standard basis Ω_0) can have one input (the `NOT` gate) or two inputs (the `AND` gate and the `OR` gate). The input can either be the output of a previous computing step, or one of the s input variables. The operation can either be `AND`, `OR` or `NOT`. We fix the output to be the result of the last gate, and doing so yields a conservative estimate of the total number of circuits. There are at least $2(i+s)^2 + (i+s)$ possibilities for the $(i+1)$-th computing step. For a circuit of size n, the corresponding straight-line program must contain $n - s$ steps. The possible combinations of the steps are the product of the possibilities in each step, that is at least

$$\prod_{i=0}^{n-s-1} 2(i+s)^2 + (i+s) = \prod_{i=s}^{n-1} 2i^2 + i \geqslant \prod_{i=s}^{n-1} 2i^2 = 2^{n-s} \left(\frac{(n-1)!}{(s-1)!} \right)^2 .$$

Therefore, the number of circuits of size n with s input variables is at least $2^{n-s}(n-1)!^2(s-1)!^{-2}$. Since s can be any number from 1 to n, the size of Γ^n can be approximated as the sum over different choices of s:

$$|\Gamma^n| \geqslant \sum_{s=1}^{n} 2^{n-s}\left(\frac{(n-1)!}{(s-1)!}\right)^2 = 2^{n-1}(n-1)!^2\sum_{s=0}^{n-1}\frac{1}{2^s s!^2} = 2^{n-1}(n-1)!^2\times[1, 1.566082929756350537292\ldots].$$

The series $\sum_{s=0}^{n-1}\frac{1}{2^s s!^2}$ equals 1 if only the first term is evaluated. When $n\to\infty$, the infinite series is equal to $I_0(\sqrt{2})\approx 1.566$, where $I_\alpha(x)$ is the modified Bessel function of the first kind. We conservatively choose 1 as the lower bound, while it shows that our approximation at this step is quite tight.

(2) Fraction of programmable circuits. However, at most $2^{c\log_2 n} = n^c$ programs can be expressed by no more than $c\log_2 n$ bits. Due to the pigeonhole principle, the fraction of the circuits in Γ^n can be expressed as $\mathrm{Prob}(n,c)$ where

$$\mathrm{Prob}(n,c) = n^c\left|\Gamma^n\right|^{-1} \leqslant 2^{-n+1}n^c(n-1)!^{-2} = 2^{-n+1}n^{c+2}n!^{-2}.$$

The resulting approximation of the fraction decreases very fast with n, for example when $n=10, c=10$ the approximated fraction is about 0.015%, but increasing n to 12 reduces the fraction to 1.9×10^{-8}. As proved that almost none of the circuits in Γ^n are useful, the conventional definition of universality is not helpful in studying the relationship between universality and efficiency. Therefore, in Section 4, we apply appropriate constraints to focus only on necessary universality, i.e., quasi-universal architectures.

3.4 Architectural universality and quasi-universality

Before diving into specific architectures, we formalize the concepts of architectural universality and quasi-universality that this paper refers to. We say the architecture is universal if the n-th circuit implementation embeds Γ^n for all n, such as Valiant's UC. Since an architecture is a series of circuit families subject to certain constraints, the definition can be formalized as follows.

Definition 2 (Architecture)**.** Architecture $A=\langle B,P\rangle$ is a series of circuit families $A^0, A^1, A^2, \ldots$, where each circuit family is a subset of the base B^n subject to a set of constraints P, i.e., $A^n=\{C\in B^n \mid P(C)\}$. A circuit family $I=\{I_0, I_1, I_2, \ldots\}$ is an implementation of architecture A if, for every instance $I_n\in I, I_n \overset{\boldsymbol{u}}{\rightarrow} A^n$.

Definition 3 (Universal architecture)**.** A circuit family $C=\{C_0, C_1, C_2, \ldots\}$ is universal if it implements Γ, i.e., $\forall n\in\mathbb{N}, C_n \overset{\boldsymbol{u}}{\rightarrow} \Gamma^n$. An architecture A is universal if all implementations of A are universal.

As for quasi-universal architectures, the requirements are relaxed. For Γ^n to be embedded, it is indeed embedded in some instantiated circuit implementation of the architecture, but possibly latter than the n-th instance. The universality of this relaxed form allows some circuits in Γ^n to be embedded in subsequent instances of the implementation (the $\Omega(n)$-th, larger instances). The definition formalized is as follows.

Definition 4 (Quasi-universal architecture)**.** A circuit family $C=\{C_0, C_1, C_2, \ldots\}$ is quasi-universal if $\exists f:\mathbb{N}\mapsto\mathbb{N}, \forall n\in\mathbb{N}, C_{f(n)} \overset{\boldsymbol{u}}{\rightarrow} \Gamma^n$. An architecture A is quasi-universal if all implementations of A are quasi-universal.

We refer to the function f as the completion function. For simplicity, we additionally require the completion function to be monotonic. The completion function describes the size-increments required for a quasi-universal architecture to support a complete universality, showing a lower bound on the power of the architecture: the $f(n)$-th instance of the architecture is at least as powerful as UC_n.

The completion function describes the size increment required for a quasi-universal architecture to support full universality, showing a lower bound of the architecture's power: the $f(n)$-th instance of the architecture implementation is at least as powerful as UC_n.

By definition, universal architectures are also quasi-universal, and the completion functions are the identity function. If an architecture is not quasi-universal by definition, then it is said to be specialized.

4 Quasi-universal architectures

Unlike universal architectures, quasi-universal architectures prioritize one particular or a small set of functions rather than treating all a priori possibilities of functions uniform. Therefore, these architectures can provide better efficiency on those prioritized functions while maintaining comparable efficiency on

Table 2 Discussed architectures and their projections in real-world architectures

Type	Constraint	Projected real-world architecture
Universal circuit	–	Programmable logic (CPLD, FPGA)
The Static	Fixed interconnections	Systolic array [14]
The Packed	Packed interconnections	Vector machine (Cray), SIMD, SIMT (CUDA)
The Slacked	Fixed non-trivial basic operations	Accelerator (DaDianNao [20], TensorCore)
The Encapsulated (Enc)	Non-trivial basic operations	–
The Recursive-Encapsulated (REnc)	Recursive non-trivial basic operations	Cambricon-FR [21]

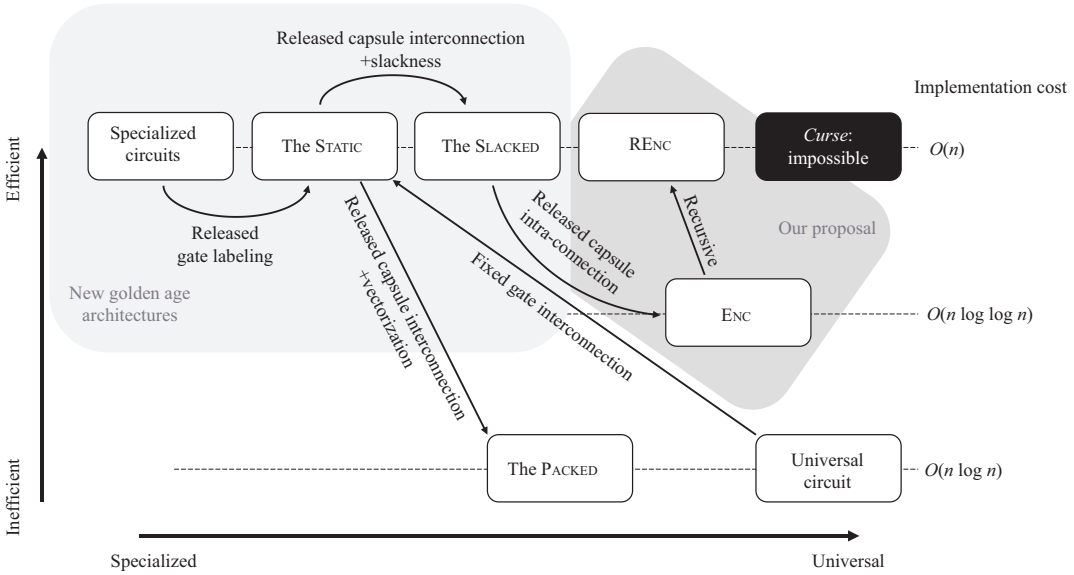

Figure 4 Overview of the discussed architectures.

other functions. However, the design of architectures, i.e., deciding which part of the universality is prioritized to portray real-world problems, is highly dependent on human intellect. In this section, we discuss the design and cost of several architectures formed by applying different constraints to Γ^n, as summarized in Table 2 [20, 21] and Figure 4.

4.1 The Static

The first type of constraints we apply is fixed interconnects. Since the uncertainty mainly comes from the undetermined gate-to-gate interconnections, we fix the interconnections in each embedded circuit to the same pattern, e.g., the same as the reference circuit family R, but the gate names are still variable. Therefore, this architecture is denoted by the Static_R architecture. We formally define Static_R as follows.

Definition 5 (The Static predicate)**.** Define the Static predicate as $S_R(C)$, where $R = \{R_0, R_1, R_2, \ldots\}$ is the reference circuit family, and $C = \langle V, E\rangle$ is the circuit to be examined. Let $n = |C|$ denote the size of C. Let V_{R_n} and E_{R_n} denote the vertices and edges of R_n, respectively. The value of the predicate $S_R(C)$ is whether there exists a bijective vertice mapping $f : V \mapsto V_{R_n}$, such that $\{(f(v_1), f(v_2)) \,|\, (v_1, v_2) \in E\} = E_{R_n}$, i.e., C and R_n are isomorphic.

Definition 6 (The Static architecture)**.** $\text{Static}_R = \langle \Gamma, S_R \rangle$.

Then we show that the implementation cost of Static_R is $\Theta(n)$.

Theorem 2 (Cost of the Static)**.** $\exists I_n : |I_n| = \Theta(n) \wedge I_n \overset{\mathbf{u}}{\rightarrow} \text{Static}_R^n$.

Proof. See Figure 5. Construct the circuit I_n by replacing the gates in R_n with universal gates, which could be implemented as multiplexers plus all types of gates in the basis Ω_R. Then I_n can simulate any circuit in Static_R^n. Since Ω_R always contains only a limited types of gates (e.g., $|\Omega_0| = 3$ gates), the size of I_n is with in a constant multiplicative factor of R_n, i.e., $O(n)$. Trivially, $|I_n|$ is $\Omega(n)$ by definition.

The resulting Static_R architecture is only quasi-universal if the reference circuit family has certain properties, which are beyond the scope of this paper. We assume these conditions are met.

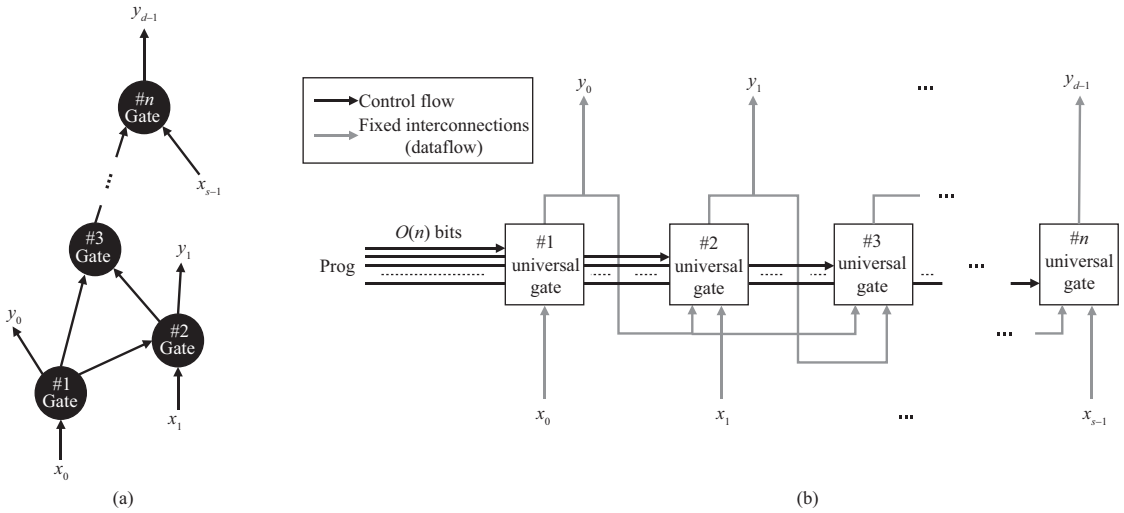

Figure 5 The $\textsc{Static}_R$ architecture fixes the interconnections of universal gates, as the gate interconnections are in R. (a) Example reference circuit R_n; (b) construction of $I_n : I_n \overset{\boldsymbol{u}}{\mapsto} \textsc{Static}_R^n$.

With fixed interconnects, the STATIC architecture is very efficient. It improves the cost from $O(n \log_2 n)$ of UC to $O(n)$, thus removing the *Curse*. The price is a very narrow prioritizing set, since any circuit interconnected differently than R_n will not be embedded in the n-th instance.

The best possible completion function is $\Theta(n \log_2 n)$, which occurs when the reference circuit family is universal circuits, i.e., $\forall i, R_i \equiv \mathrm{UC}_i$. However, despite providing a theoretical lower bound for the completion function, an architecture that prioritizes UC may not actually make sense. A more representative example of a reference circuit family is meshes, and such circuits can also be seen as temporally unrolled systolic architectures. When reference circuits are 2D-meshes, the best completion function is $\Theta(n^2)$: $O(n^2)$ is achieved using orthogonal grid drawing techniques such as [22], and $\Omega(n^2)$ is required due to [23].

4.2 The PACKED

Since fixing interconnects is a too strong constraint, we apply the second constraint only on the patterns of interconnects, rather than fixing them completely. Vectorization is the most common way to constrain interconnections in practice. The $\textsc{Packed}_w$ architecture requires that each $w(n)$-bit of data is packed as an vector, and the data in each vector is always forwarded together. From the information lower bound, for UC_n, the free interconnection between n gates brings about $O(n!)$ uncertainty, so it requires $O(n \log_2 n)$ information bits to determine the embedded circuit. But for $\textsc{Packed}_w^n$, the uncertainty is reduced from $O(n!)$ to $O(^n/_{w(n)}!)$ for free forwarding $^n/_{w(n)}$ vectors. Therefore, the informational bound is relaxed from $\Omega(n \log_2 n)$ to $\Omega(^n/_{w(n)} \log_2 {}^n/_{w(n)})$. If $w(n) = \log_2 n$, then the bound will be $\Omega(n)$; i.e., the *Curse* is no longer guaranteed by the information bound. We formally define the $\textsc{Packed}_w$ architecture as follows.

Definition 7 (The PACKED architecture)**.** $\textsc{Packed}_w = \langle \Gamma, P \rangle$, where $w : \mathbb{N} \mapsto \mathbb{N}$ is the function that decides the length of the vector given the circuit size, and the $P(C)$ predicate is defined as the existence of $C' \subseteq C$, such that C is a disjoint union of $w(|C|)$ isomorphic copies of C'.

Unfortunately, despite the relaxation of information bounds by vectorization, the PACKED architecture still cannot remove the *Curse*, since when implemented in logic circuits, the asymptotic cost of vector-wise forwarding is no less than bit-wise forwarding. We formalize the argument as the following theorem.

Theorem 3 (The PACKED is not efficient)**.** $\forall I_n : I_n \overset{\boldsymbol{u}}{\mapsto} \textsc{Packed}_w^n \wedge w(n) = o(n) \to |I_n| = \Omega(n \log_2 n)$.

Proof. Without loss of generality, we consider the implementation as aligned single instruction, multiple data (SIMD) execution units that perform bitwise operations on input variables. The n vertices in the circuit graph are packed into $^n/_{w(n)}$ vectors, $w(n)$ in each. The implementation can decompose along intra-vector offsets, since bits from different intra-vector offsets never interact. Figure 6(a) shows such a decomposition, where the implementation is divided into $w(n)$ lanes, and in each lane a universal circuit $\mathrm{UC}_{O(^n/_{w(n)})}$ is required. Since in each lane there are $|\mathrm{UC}_{O(^n/_{w(n)})}| = \Omega\left(^n/_{w(n)} \log_2 {}^n/_{w(n)}\right)$ gates, $w(n)$ lanes have a total of $\Omega\left(n \log_2 {}^n/_{w(n)}\right)$ gates. Since $w(n) = o(n)$, the $w(n)$ in the denominator of the

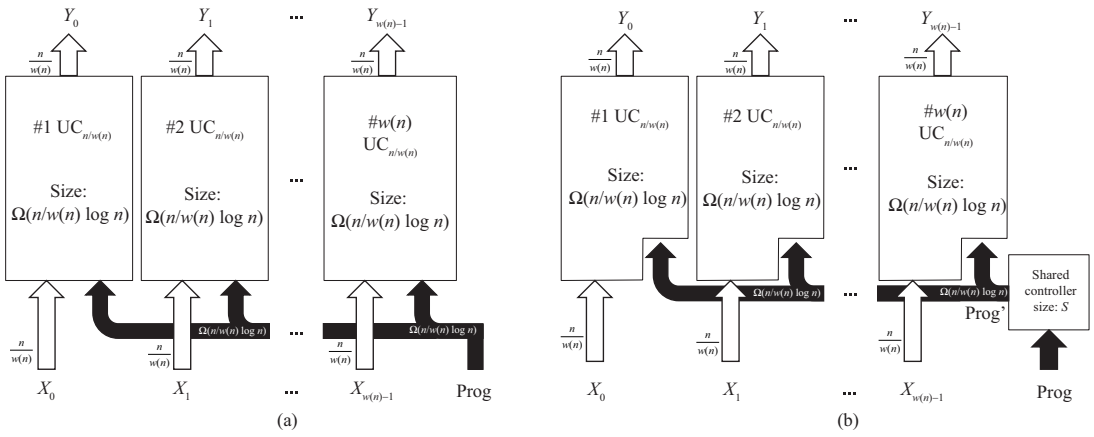

Figure 6 Vectorization does not remove the *Curse*. (a) Implementation via decomposition into $w(n)$ independent lanes; (b) saving some gates by building a shared controller, while the informational bound still applies to each lane.

logarithm will be hidden by the asymptotic notation, i.e., $\Omega\left(n \log_2 {}^{n}/_{w(n)}\right) = \Omega\left(n\left(\log_2 n - \log_2 w(n)\right)\right) = \Omega\left(n \log_2 n\right)$.

The implementation shown by decomposition is optimal within a constant multiplicative factor, since putting independent variables together does not save anything. The only gates that can be saved are those that depend only on shared inputs (i.e., programming bits). One can extract the saved gates from lanes, and combine them into a shared controller. However, this does not change the asymptotic size of each lane. As shown in Figure 6(b), after the shared controller is extracted, the processed programming bits Prog' connected to each lane are still bounded by the informational entropy, i.e., $\Omega\left({}^{n}/_{w(n)} \log_2 {}^{n}/_{w(n)}\right)$. Each processed programming bit must connect to some gates, hence the size bound $\Omega\left({}^{n}/_{w(n)} \log_2 {}^{n}/_{w(n)}\right)$ for each lane still applies.

The completion function for the Packed architecture is obviously $nw(n)$: by using only the first bit in the vector, an instance of $\textsc{Packed}^{nw(n)}$ degenerates to UC_n.

The Packed architecture in the industry has been developed since the 1970s when vector machines (e.g., Cray-1) were created for supercomputing. SIMD and SWAR instructions (e.g., Intel MMX) have been implemented in desktop processors since the 1990s. It has also been at the heart of GPGPU efficiency, starting with the release of CUDA by NVIDIA in 2007, until it was succeeded by TensorCore. Today, Intel is still releasing packed instructions to accelerate deep learning workloads, such as the AVX512VNNI extension. However, since packed instructions only perform trivial operations, the cost of PRF access becomes a major bottleneck for efficiency. Thus, the industry has moved to the next, ushering in a new golden age of computer architecture [1].

4.3 The Slacked

Since vectorization cannot reduce the overall size of the EUG, the number of bits forwarded in the EUG must actually be reduced in order to build an efficient architecture. We introduce slackness; i.e., for each $w(n)$ bits forwarded in the EUG, $v(n) = \omega\left(w\left(n\right)\right)$ gates are required to compute over the $w(n) = o\left(v\left(n\right)\right)$ bits, without any other incoming/outgoing wires. Let $\textsc{Slacked}_{v,w,R}$ denote the slacked architecture. We formally define the architecture as follows.

Definition 8 (The capsule predicate)**.** Define the capsule predicate as $K_{v,w,P}(C)$, where $C = \langle V, E\rangle$ is the circuit to be examined, $v : \mathbb{N} \mapsto \mathbb{N}$ is the function deciding the maximum size of each capsule given the size of C, $w : \mathbb{N} \mapsto \mathbb{N}$ is the function deciding the maximum number of terminals of each capsule given the size of C, and P is the sub-predicate to examine over each capsule. Let $n = |C|$ and $k = {}^{n}/_{v(n)}$. The value of the predicate $K_{v,w,P}(C)$ is whether there exists a set of sub-circuits $C_* = \{\langle V_1, E_1\rangle, \langle V_2, E_2\rangle, \ldots, \langle V_k, E_k\rangle\}$, such that the following conditions are satisfied.

(1) Properly partitioned into capsules. C_* is a $(k, 1)$ graph partition of C.

(2) Each capsule has v-limited gates. $\bigwedge_{C_i \in C_*} |C_i| \leqslant v(n)$ holds.

(3) Each capsule has w-limited in-/outgoing terminals. Let $E_{(i,j)} = \{(v_1, v_2) \in V_i \times V_j \,|\, (v_1, v_2) \in E\}$, $\forall i \leqslant k, \sum_{i \neq j=1}^{k} |E_{(i,j)}| + |E_{(j,i)}| \leqslant w(n)$.

(4) Each capsule meets the sub-predicate. $\bigwedge_{C_i \in C_*} P(C_i)$ holds.

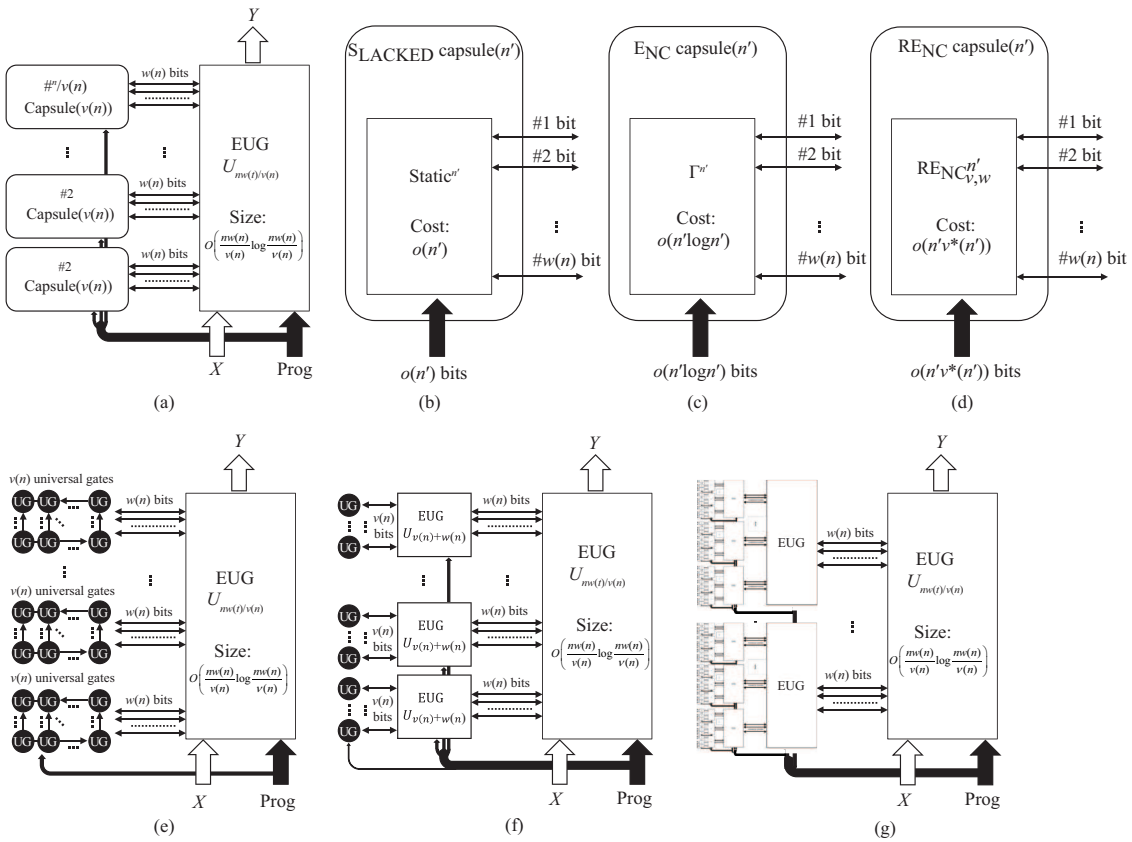

Figure 7 Efficient quasi-universal architectures: the SLACKED ((a) and (b)), the Encapsulated (ENC, (a) and (c)), and the Recursive-Encapsulated (RENC, (a) and (d)). (e), (f), and (g) show the detailed implementation in the capsules, respectively.

For completeness, $K_{v,w,P}$ is defined as tautology $\top$ when $k = 1$ and $P = K_{v,w,P}$, i.e., when the capsule predicate evaluates to itself.

Definition 9 (The SLACKED architecture)**.** $\text{SLACKED}_{v,w,R} = \langle \Gamma, K_{v,w,S_R} \rangle$.

Roughly, the uncertainty in the $\text{SLACKED}_{v,w,R}$ architecture is reduced from $O(n!)$ to $O(^{nw(n)}/_{v(n)}!)$, when $v(n) = \Omega(w(n)\log_2 n)$, the informational bound is relaxed from $O(n\log_2 n)$ to $O(n)$. Different from the PACKED, the SLACKED architecture can actually save gates since the size of EUG is reduced. We formalize the argument as follows.

Theorem 4 (Cost of the SLACKED)**.** $v(n) = \Omega(w(n)\log_2 n) \to \exists I_n : I_n \underline{\boldsymbol{u}} \text{SLACKED}^n_{v,w,R} \wedge |I_n| = O(n)$.

Proof. Figures 7(a) and (b) show an implementation. The n gates to be embedded are divided and organized as $^n/_{v(n)}$ capsules, and each capsule contains $v(n)$ gates organized as an instance of $\text{STATIC}^{v(n)}_R$. Each capsule has $w(n)$ terminals for incoming/outgoing bits. The universal interconnects between all terminals are provided through a EUG $U_{nw(n)/v(n)}$, which provides $^{nw(n)}/_{v(n)}$ terminals. The size of each capsule is $O(v(n))$, and the size of $U_{nw(n)/v(n)}$ is $O(^{nw(n)}/_{v(n)}\log_2 {}^{nw(n)}/_{v(n)})$. The total cost is $O(n)$ if $v(n) = \Omega(w(n)\log_2 n)$.

For simplicity, we omit the parameters v and w in the notation when $w(n) = \Theta(\log_2 n)$ and $v(n) = \Theta(w(n)\log_2 n)$, and also omit the capsule reference R when not discussing a particular reference circuit family.

On the downside, the SLACKED increases the cost of trivial operations, because most gate embeddings in the capsules are wasted when only $o(v(n))$ gates are used per $w(n)$ bits. For example, imagine a situation where scalar/vector arithmetic needs to be padded with zeros to take advantage of a processor with only matrix units. But for circuits built primarily of non-trivial operations, once the capabilities of the capsules are fully utilized, the SLACKED architecture will be much more efficient than the PACKED. In the industry, the most notable instance of the SLACKED architecture is the TensorCore introduced in the NVIDIA Volta, 2017. Compared to conventional CUDA units, TensorCore units (capsules) perform non-trivial computations on low-precision input data, delivering state-of-the-art performance for deep learning workloads.

So far we have shown that the SLACKED prioritizes circuits built mainly with non-trivial operations and can be implemented at $O(n)$ cost, thus removing the *Curse*. The completion function for the SLACKED architecture is $nv(n)$ in the worst case, similar to the deduction for the PACKED by using only one gate per capsule. However, better completion functions are possible if a detailed specification is given. For example, $\text{SLACKED}^{n}_{\log_2{}^2 n,\log_2 n,\text{2DMESH}}$ (SLACKED^{n} with $\text{STATIC}^{\log_2{}^2 n}_{\text{2DMESH}}$ capsule) has a completion function $O(n\log_2 n)$, since each capsule is known to be $\Theta(\log_2 n)$-universal as discussed earlier.

4.4 The Encapsulated (ENC)

Here, we propose the Encapsulated (ENC) architecture that further releases constraints. Unlike the SLACKED architecture that fixes interconnects inside capsules, ENC releases them. ENC only encapsulates, requiring $v(n)$ gates in each capsule to share $w(n)$ terminals, i.e., preserving the wire locality requirement. $\text{SLACKED}_{v,w,R}$ is transformed to $\text{ENC}_{v,w}$ by releasing interconnects inside capsules. As shown in Figures 7(a) and (c), the capsule of $\text{ENC}_{v,w}$ is $\Gamma^{v(n)}$ instead of $\text{STATIC}^{v(n)}_{R}$. Therefore, each $v(n)$ gate can connect to each other without restriction, but any incoming/outgoing wires beyond these gates are limited by the $w(n)$ terminals per capsule. The ENC enables more flexible connections. We give a formal definition as follows.

Definition 10 (The ENC architecture)**.** $\text{ENC}_{v,w} = \langle \Gamma, K_{v,w,\top} \rangle$.

The added cost per capsule increases the total asymptotic cost by a factor of $O(\log_2 v(n))$. Specifically, when $v(n) = \Omega(w(n)\log_2 n)$, it is $O(\log_2\log_2 n)$ times. Although $\text{ENC}_{v,w}$ does not completely remove the *Curse*, it still reduces the cost to $O(n\log_2\log_2 n)$ gates and imposes the most relaxed constraints. The cost can be formalized as follows.

Theorem 5 (Cost of the ENC)**.** $v(n) = \Omega(w(n)\log_2 n) \to \exists I_n : I_n \overset{\mathbf{u}}{\rightarrow} \text{ENC}^n_{v,w} \wedge |I_n| = O(n\log_2 v(n))$.

Proof. Trivial after Theorem 4. The EUG part keeps the same. Capsule size increased by $O(\log_2 v(n))$ times, hence the total size increased to $O(n\log_2 v(n))$.

The completion function for the $\text{ENC}_{v,w}$ architecture is ${}^{nv(n)}/_{w(n)}$, by using each capsule as a $\text{UC}_{w(n)}$.

4.5 The Recursive-Encapsulated (RENC)

Since the extra double-logarithmic factor in the cost formula of ENC is due to the fact that ENC allows free connections within capsules, we can recursively apply the encapsulating constraint in the capsules, then in the capsules of capsules, and then in the capsules of capsules of capsules, and so on. In the process of recursive encapsulation, the *Curse* is divided into smaller and smaller granularities, eventually reducing the Turing tariff to almost arbitrarily small asymptotically, i.e., down to a triple-logarithmic factor, then a quadruple-logarithmic factor, and then a quintuple-logarithmic factor, and so on. We refer to the infinitely recursively encapsulated architecture as RENC. As shown in Figures 7(a) and (d), the RENC capsule is a smaller RENC instance, instead of a UC in ENC. We formalize the architecture as follows.

Definition 11 (The RENC architecture)**.** $\text{RENC}_{v,w} = \langle \Gamma, P \rangle$ where $P = K_{v,w,P}$.

Theorem 6 (Cost of the RENC)**.** $v(n) = \Theta(w(n)\log_2 n) \to \exists I_n : I_n \overset{\mathbf{u}}{\rightarrow} \text{RENC}^n_{v,w} \wedge |I_n| = O(nv^*(n))$.

Proof. The encapsulation recursively applies $v^*(n)$ times, each time introducing a level of EUG, as shown in Figure 7(a). Following Theorem 4, each EUG costs $\Theta(n)$. The recurrence formula of $|I_n|$ can be written as

$$|I_1| = O(1), \quad \text{(a)}$$
$$|I_n| = \frac{n}{v(n)}\left|I_{v(n)}\right| + \Theta(n). \quad \text{(b)}$$

The recurrence formula can be expanded by formula (b) $v^*(n)$ times before reaching (a), each time bringing out an additive term $\Theta(n)$. Therefore, $|I_n| = O(nv^*(n))$.

Specifically, when $w(n) = \Theta(\log_2 n)$ and $v(n) = \Theta(w(n)\log_2 n)$, the cost to implement $\text{RENC}^n_{v,w}$ is $O(n\log^* n)$. We claim that RENC can remove the *Curse* because $O(\log^* n)$ grows so slowly. For any circuit that can be constructed in the known universe, $\log^*_2 n \leqslant 5$, and thus can be safely treated as a constant factor in practice. *Curse* can also be perfectly removed by letting $v(n)$ grow slightly faster w.r.t. $w(n)$, say $v(n) = \Omega(w(n)\log_2 n\log^* n)$; alternatively, by letting $w(n)$ be polylogarithm, say $w(n) = \log_2{}^b n$ and $v(n) = \log_2{}^a n$, $a > b+1$. The cost per EUG will be slightly reduced to $O({}^n/_{\log^* n})$, so the cost of RENC will be exactly $O(n)$.

The completion function of the RENC architecture remains the same as the ENC, i.e., ${}^{nv(n)}/_{w(n)}$.

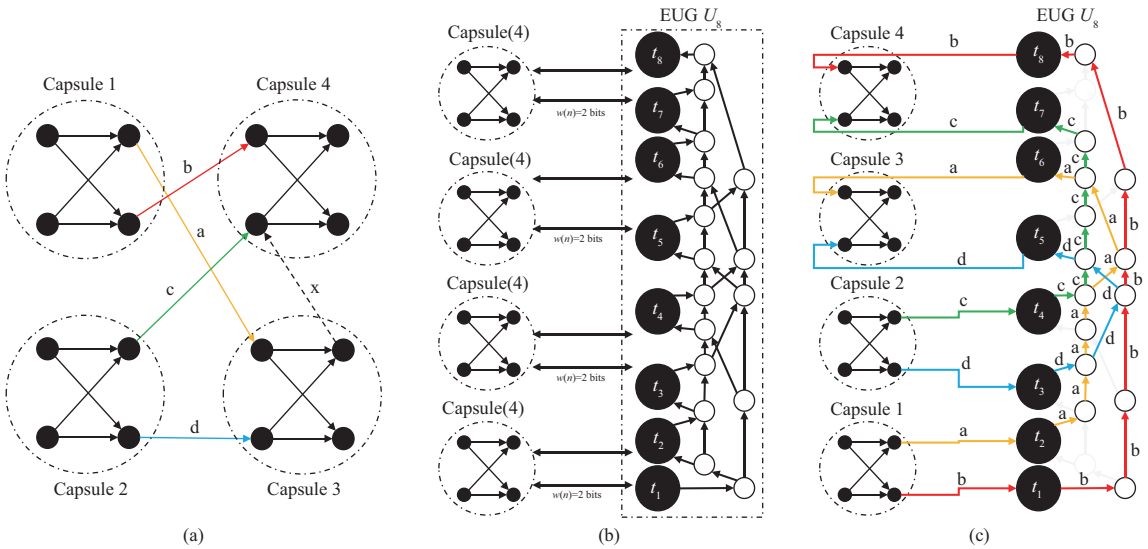

Figure 8 (Color online) A minimal example showing the power of encapsulation. (a) Circuit following the capsule predicate $K_{4,2,\top}$ (Definition 8) w/o the dotted wire x, but adding the dotted wire x will inevitably violate $K_{4,2,\top}$, because no partitions can keep the crossing wires within $w(n) = 2$ for every capsule. (b) The implementation of a capsule-based quasi-universal architecture that constrains $K_{4,2,\top}$ (the $\textsc{Slacked}^{16}_{4,2}$, the $\textsc{Enc}^{16}_{4,2}$, and the $\textsc{REnc}^{16}_{4,2}$), corresponding to Figure 7(a). (c) Embedding (a) into (b), the embedded wires are annotated in colors a, b, c, d.

4.6 Power of the quasi-universal architectures

In this subsection we discuss the power of the proposed quasi-universal architectures. Quasi-universal architectures are modified a priori probability distributions over Γ. They injected human intelligence to better characterize real-world problems, rather than treating them equally as randomized problems like the proof of Theorem 1. Theorem 1 is an imitation of Kolmogorov's incompressibility theorem, which states that almost all strings are incompressible. However, compression algorithms like LZW do work because they capture the characteristics behind real-world data. Similarly, we expect the quasi-universal architectures to work like "compressors" of circuits by prioritizing real-world circuits. For example, the $\textsc{Static}_{\textsc{2DMesh}}$ architecture prioritizes circuits where each gate is only locally connected to its nearest neighbors. The Slacked, the Enc, and the REnc are capsule-based quasi-universal architectures that prioritize circuits with dense local connections but sparse global connections, i.e., locality. The Slacked shows that building the circuits with non-trivial basic operations can be a rescue, as it implies locality in the computing scheme. The Enc further releases the connections inside capsules but keeps the locality, then the REnc extends locality to a scale-invariant property.

The power of the Static and the Packed architectures is straightforward, as many practical architectures follow their constraints. To understand the power of capsule-based architectures, the capsule predicate (Definition 8) is the key. It is the capsule predicate that implies locality, and it is real, since it has been observed in VLSI engineering as Rent's Rule [24].

4.6.1 *The capsule predicate*

We illustrate the capsule predicate (Definition 8) $K_{v,w,P}$. Figure 8(a) shows a circuit ($n = 16$). Without the dotted wire x, we show how to partition the vertices (gates or inputs) into capsules, where each capsule has at most $v = 4$ vertices, $w = 2$ incoming/outgoing wires (a, b, c, d). As the partition exists, we say the shown circuit follows $K_{4,2,\top}$. However, if we add the dotted wire x into the circuit, it will violate $K_{4,2,\top}$ since such a partition will not exist anymore.

$K_{4,2,\top}$ is both necessary and sufficient for the membership in $\textsc{Enc}^n_{4,2}$, and is necessary for the membership in $\textsc{Slacked}^n_{4,2}$ and $\textsc{REnc}^n_{4,2}$. The latter two architectures additionally require every capsule to follow a sub-predicate, i.e., S_R for $\textsc{Slacked}^n_{4,2}$ and $K_{4,2,K_{4,2,K_{\dots}}}$ for $\textsc{REnc}^n_{4,2}$. These additional requirements are to be understood separately per capsule, hence here we do not dive into it.

Figure 8(b) shows how to implement a capsule-based architecture, please also compare with Figure 7. The EUG U_8 provides $w = 2$ terminals for each capsule, each terminal can embed either an incoming wire or an outgoing wire. U_8 also provides universal interconnection between terminals. Therefore, for

Table 3 Asymptotic comparison of quasi-universal architectures

Architecture	Cost	Completion function	Complete cost
Universal			
Γ	$\Theta(n \log_2 n)$	n	$\Theta(n \log_2 n)$
The Static			
$\textsc{Static}_{\textsc{UC}}$	$\Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$
$\textsc{Static}_{\textsc{2DMesh}}$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
The Packed			
$\textsc{Packed}_{\log_2 n}$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2{}^2 n)$
The Slacked			
$\textsc{Slacked}_{\log_2{}^2 n, \log_2 n}$	$\Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$
The Encapsulated			
$\textsc{Enc}_{\log_2{}^2 n, \log_2 n}$	$O(n \log_2 \log_2 n)$	$\Theta(n \log_2 n)$	$O(n \log_2 n \log_2 \log_2 n)$
The Recursive-Encapsulated			
$\textsc{REnc}_{\log_2{}^2 n, \log_2 n}$	$O(n \log^* n)$	$\Theta(n \log_2 n)$	$O(n \log_2 n \log^* n)$
$\textsc{REnc}_{\log_2{}^a n, \log_2{}^b n}, a > b+1$	$\Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$

any circuit that follows $K_{4,2,\top}$, U_8 can embed all interconnects across the capsule. Figure 8(c) shows the embedding of Figure 8(a), and it can be easily observed that there is no more space to embed the dotted wire x. Capsules will be implemented separately based on their required sub-predicates, so we will not show the implementation inside capsules here either. Specifically, to understand the Enc architecture, imagine a Valiant's UC built inside each capsule; to understand the REnc architecture, imagine an implementation similar to Figure 8(c) but of different sizes built recursively in each capsule.

4.6.2 *Bounds of powers*

Table 3 summarizes the costs and completion functions of the discussed architectures. The complete cost is the composite of the two, indicating the worst-case cost when conventional universality is required.

REnc preserves universality to the greatest extent, since relaxing any constraints in $\textsc{REnc}^n$ will result in a worse cost. However, there are still some common problems that cannot be efficiently computed in the REnc. We define a new complexity class $\mathsf{REnc}(n)$ as the set of problems that circuits in $\textsc{REnc}^n$ can compute, and show some basic relations to several complexity classes.

Theorem 7. There exists a sorting network for $O(n \log_2 \log_2 n / \log_2{}^2 n)$ bits in $\textsc{REnc}^n$.

This is due to Beigel et al. [25] that an n-sorter can be built from $O(n/k \log_k n)$ k-sorters for any n and k. As a corollary, for $k = w(n) = \log_2 n$, there are $O(n/\log_2 \log_2 n)$ k-sorters each embedded in an REnc capsule. This result is comparable to, when embedded in $\textsc{UC}_n$, the $O(n/\log_2{}^2 n)$-bit bitonic sorters or Batcher's sorting networks, which are considered the best practical sorting circuits. The result shows that REnc is quite efficient for sorting.

However, it is unlikely one can embed larger sorting networks into $\textsc{REnc}^n$ through AKS-like constructions. Although we still leave this question open. AKS sorting networks [26] (and its variants) are asymptotically optimal, but impractical due to huge constant factors. The difficultly is that sorting only has a decomposability factor of $O(\log_2 n)$, so most gate-embeddings in the REnc capsules are certainly wasted due to the lack of locality, assuming the capsules perform sorting as well (either the even weaker halving operations used in AKS).

Theorem 8. $\mathsf{SIZE}(n/\log_2 n) \subset \mathsf{REnc}(n)$.

Trivial lower bound due to the completion function. The containment can be shown by the embedding of a smaller UC (i.e., $\textsc{UC}_{n/\log_2 n}$) in $\textsc{REnc}^n$. Despite the capsules in REnc can embed up to $v(n)$ gates, we use only $w(n)$ of them, and fill other gate-embeddings and switches with padding. Afterwards, the $\textsc{REnc}^n$ instance degenerates to $\textsc{UC}_{\frac{nw(n)}{v(n)}}$ [16].

Finding one problem in $\mathsf{REnc}(n)$ but not in $\mathsf{SIZE}(n/\log_2 n)$ is sufficient to show that the containment is proper. For example Theorem 7 applied here, sorting $O(n \log_2 \log_2 n / \log_2{}^2 n)$ variables is in $\mathsf{REnc}(n)$ but not in $\mathsf{SIZE}(n/\log_2 n)$.

Theorem 9. $\mathsf{REnc}(n) \subset \mathsf{SIZE}(n)$.

This is obvious since REnc is obtained by applying constraints on UC, its functionality is a subset of UC's. The *Curse* says the containment is proper due to the $o(n \log_2 n)$ cost of REnc.

5 Discussion

One of the major challenges for efficiency-first research is to quantify universality. Hennessy and Patterson, who claimed the new golden age in their Turing lecture, provided a systematic quantitative approach for computer architecture research (certainly, quantifying efficiency for universality-first research). Although this paper defines the completion function for quasi-universal architectures, herein, we only describe a trivial lower bound for universality. The quantification of universality is still an important open question.

Several deep learning processors are initiated by the quasi-universal architectures (especially the REnc), including the fractal von Neumann architecture (FvNA) [27] and the functional instruction set computers (FISC) (Zhiwei Xu, personal communication). Cambricon-FR [21] is an implementation of FvNA. Initiated by the REnc, Cambricon-FR decomposes non-trivial instructions recursively and allows universal control over each hierarchy. Such a design has led to great reductions in the size of programs required as the REnc does. To further cut down the size of programs, Cambricon-FR requires the decomposed instructions to keep their name and are therefore fractal. As a result, Cambricon-FR can address the programming productivity issue encountered in the industry of deep learning processors. However, applying the REnc architecture directly in real processors could be challenging. There are two major difficulties. The first is due to the fact that the circuit model we used corresponded only to combinational logic circuits. It is left for future work to make the REnc effectively implemented in sequential logic circuits. The second is due to the programming limitations. In this work, we are only able to prove that programming the implementation of the REnc is possible, but it is still not clear how to program it under a reasonable time budget.

The discussed architectures are not only a set of processor design guidelines. We expect the possible application of quasi-universal architectures in the field of inductive inference.

6 Conclusion

In the new golden age of computer architecture, computer scientists traded off universality for optimal efficiency. However, we predict that for post-golden-age computers, universality will once again become a major concern. The *Curse* says universal computers cannot be efficient, hence we define quasi-universal architectures as a rescue. Quasi-universal architectures (such as the proposed REnc) can solve any computable problem and are efficient for a wide range of problems. The discovery of the REnc architecture suggests that current new-golden-age architectures are not Pareto optimal.

Acknowledgements This work was partially supported by National Key Research and Development Program of China (Grant No. 2018AAA0103300), National Natural Science Foundation of China (Grant Nos. 61925208, 62102398, U19B2019), Strategic Priority Research Program of Chinese Academy of Science (Grant No. XDB32050200), Beijing Academy of Artificial Intelligence (BAAI) and Beijing Nova Program of Science and Technology (Grant No. Z191100001119093), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-029), and Youth Innovation Promotion Association CAS and Xplore Prize. The authors would like to thank Qian LI from Institute of Computing Technology, Chinese Academy of Sciences, and Yu XIA from Peking University for their constructive comments in relation to this work.

References

1 Hennessy J L, Patterson D A. A new golden age for computer architecture. Commun ACM, 2019, 62: 48–60
2 Turing A M. On computable numbers, with an application to the entscheidungsproblem. Proc London Math Soc, 1937, s2-42: 230–265
3 von Neumann J. First draft of a report on the EDVAC. IEEE Ann Hist Comput, 1993, 15: 27–75
4 Valiant L G. Universal circuits (preliminary report). In: Proceedings of the 8th Annual ACM Symposium on Theory of Computing, Hershey, 1976. 196–203
5 Esmaeilzadeh H, Blem E, Amant R S, et al. Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA), 2011. 365–376
6 Chen T, Du Z, Sun N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014. 269–284
7 Knuth D E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd ed. Redwood City: Addison Wesley Longman Publishing Co., Inc., 1997
8 Savage J E. Models of Computation: Exploring the Power of Computing. Redwood City: Addison-Wesley Longman Publishing Co., Inc., 1997
9 Arora S, Barak B. Computational Complexity: A Modern Approach. Cambridge: Cambridge University Press, 2009
10 Alhassan M Y, Günther D, Kiss Á, et al. Efficient and scalable universal circuits. J Cryptol, 2020, 33: 1216–1271
11 Zhao S, Yu Y, Zhang J, et al. Valiant's Universal Circuits Revisited: An Overall Improvement and a Lower Bound. Technical Report 943, 2018

12 Liu H, Yu Y, Zhao S, et al. Pushing the limits of valiant's universal circuits: simpler, tighter and more compact. In: Proceedings of Annual International Cryptology Conference, 2021. 365–394
13 Backus J. Can programming be liberated from the von Neumann style? Commun ACM, 1978, 21: 613–641
14 Kung H T. Why systolic architectures? Computer, 1982, 15: 37–46
15 Kiss Á, Schneider T. Valiant's universal circuit is practical. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques. Berlin: Springer, 2016. 699–728
16 Lipmaa H, Mohassel P, Sadeghian S. Valiant's universal circuit: improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017, 2016. https://ia.cr/2016/017
17 Neary T, Woods D. The complexity of small universal Turing machines: a survey. In: Proceedings of Theory and Practice of Computer Science, 2012. 385–405
18 Kelly P H J. Advanced Computer Architecture, Chapter 1.3: the stored program concept and the Turing Tax. 2020. https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/Lectures/pdfs/Ch01-part4-TuringTaxDiscussion.pdf
19 Edwards C. Moore's law: what comes next? Commun ACM, 2021, 64: 12–14
20 Chen Y, Luo T, Liu S, et al. DaDianNao: a machine-learning supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014. 609–622
21 Zhao Y, Fan Z, Du Z, et al. Machine learning computers with fractal von Neumann architecture. IEEE Trans Comput, 2020, 69: 998–1014
22 Kornaropoulos E M, Tollis I G. Algorithms and bounds for overloaded orthogonal drawings. J Graph Algorithms Appl, 2016, 20: 217–246
23 Valiant L G. Universality considerations in VLSI circuits. IEEE Trans Comput, 1981, C-30: 135–140
24 Lanzerotti M Y, Fiorenza G, Rand R A. Microminiature packaging and integrated circuitry: the work of E. F. Rent, with an application to on-chip interconnection requirements. IBM J Res Dev, 2005, 49: 777–803
25 Beigel R, Gill J. Sorting n objects with a k-sorter. IEEE Trans Comput, 1990, 39: 714–716
26 Ajtai M, Komlós J, Szemerédi E. An $O(n \log n)$ sorting network. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 1983. 1–9
27 Zhao Y, Du Z, Guo Q, et al. Cambricon-F: machine learning computers with fractal von Neumann architecture. In: Proceedings of ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019. 788–801