OSCAR: An Open-Source Flexible and Hierarchical AI Accelerator Generator with Accurate Power Model

JAY ZHE-AN MOK, The Hong Kong University of Science and Technology Department of Electronic & Computer Engineering, Hong Kong
QIJUN ZHANG, The Hong Kong University of Science and Technology Department of Electronic & Computer Engineering, Hong Kong
DI PANG, AI Chip Center for Emerging Smart Systems, Hong Kong
XUEJIAO LIU, AI Chip Center for Emerging Smart Systems, Hong Kong
YAO LU, The Hong Kong University of Science and Technology, Hong Kong
DONGBO WANG, Univista, Hong Kong
ZHIYAO XIE, The Hong Kong University of Science and Technology, Hong Kong

With the growing demand for artificial intelligence (AI) applications, high-performance and energy-efficient AI chips are needed to support the computation. However, architectural-level AI chip design, PPA evaluation, and power estimation remain challenging due to the exponential set of possible designs and the difficulty of accurately modeling the impact of diverse dataflows and workflows on the underlying hardware at the architectural-design stage. We propose a novel open-source framework named OSCAR, which, given a set of hardware and workload specifications, provides architecture-level power estimation and can also automatically generate Chisel and synthesizable RTL of the custom AI chip. Our contributions include (1) a flexible and hierarchical AI chip design space, software and hardware stack, and an RTL generator supporting dense, Transformer, Winograd, systolic, and reconfigurable architectures in one unified framework, (2) hierarchy-based data-sensitive power model using architectural-level toggling features, achieving 3.8% error and correlation coefficient R > 0.99 to post-synthesis power, outperforming state-of-the-art power estimation methods, (3) validation of our power model by performing design space exploration, finding designs with better Pareto-optimality, 2.5× lower power or 2× better runtime metrics, compared with using prior art power models, and (4) a tape-out of an AI chip based on DSE results, with OSCAR modeling its power with over 90% accuracy.

CCS Concepts: • **Hardware → Very large scale integration design**; • **Computing methodologies → Machine learning**.

This work is supported by National Natural Science Foundation of China (NSFC) 62304192, 92364102, Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, 16200724, and T46-415/25-R. It is also supported by RGC JLFS-YSF/E-605/26. It was partially conducted by ACCESS - AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.
Authors' Contact Information: Jay Zhe-An Mok, The Hong Kong University of Science and Technology Department of Electronic & Computer Engineering, Hong Kong, Hong Kong; e-mail: jzmok@connect.ust.hk; Qijun Zhang, The Hong Kong Uni- versity of Science and Technology Department of Electronic & Computer Engineering, Hong Kong, Hong Kong; e-mail: qzhangcs@ connect.ust.hk; Di Pang, AI Chip Center for Emerging Smart Systems, Hong Kong, Hong Kong; e-mail: brandonpang@ust.hk; Xuejiao Liu, AI Chip Center for Emerging Smart Systems, Hong Kong, Hong Kong; e-mail: xuejiaoliu@ust.hk; Yao Lu, The Hong Kong University of Science and Technology, Hong Kong, Hong Kong; e-mail: yludf@connect.ust.hk; Dongbo Wang, Univista, Hong Kong, Hong Kong; e-mail: wdb@univista-isg.com; Zhiyao Xie (Corresponding Author), The Hong Kong University of Science and Technology, Hong Kong, Hong Kong; e-mail: eezhiyao@ust.hk.

ACM 1084-4309/2026/09-ART00
https://doi.org/10.1145/3841479

Additional Key Words and Phrases: Large Language Models, Design Space Exploration, Agile Hardware Design, Hardware Acceleration

ACM Reference Format:
Jay Zhe-An Mok, Qijun Zhang, Di Pang, Xuejiao Liu, Yao Lu, Dongbo Wang, and Zhiyao Xie. 2026. OSCAR: An Open-Source Flexible and Hierarchical AI Accelerator Generator with Accurate Power Model. *ACM Trans. Des. Autom. Electron. Syst.* 00, JA, Article 00 (September 2026), 33 pages. https://doi.org/10.1145/3841479

1 Introduction

As AI applications continue to grow, the demand for high-performance and energy-efficient AI accelerators has become increasingly evident. Designing AI chips requires navigating a vast design space encompassing diverse architectures, including spatial, Winograd, and reconfigurable architectures [1]. Finding optimal designs within this immense space is a challenging and time-consuming task.

To simplify architecture exploration, prior works often reduce the AI chip design space by focusing on the matrix multiply or convolution unit of the accelerator core. The design space is typically formulated as a combination of memory hierarchy, custom loop order, and tilings [2–6]. This formulation can cover many well-known designs, including weight-stationary, row-stationary, and systolic architectures [7–10]. However, these exploration tools suffer from several limitations, as summarized below.

Limited Design Space. The design space of accelerators is limited: (1) The exploration of reconfigurable accelerators, meaning a given operation can be mapped to different dataflows, is largely neglected. They play an important role in the performance and power tradeoffs of modern AI chips [1, 11]. (2) The important categories of bit-serial and Winograd architectures, often used in industrial AI chips [12, 13], are not considered. (3) Many essential components, such as interconnect, network circuitry, crossbars, steering logic, and accumulators, are either ignored or inadequately modelled. These modules can contribute significant power (e.g., 30–40% of power is consumed by network components such as the sparse crossbar [14]).

Insufficient Support to Implementation. Many architecture exploration tools [3, 5, 6, 15, 16] only support architecture-level performance or power estimators, leaving hardware implementation to the user. In other words, the tools are unable to generate hardware RTL, preventing users from verifying their explored designs in FPGAs or implementing the design with EDA flows straightforwardly.

Limited Early-Stage Estimator Accuracy Due to Simplification. Existing architecture exploration tools suffer from oversimplified power and energy models. They assume that the per-operation power and energy are constant or limited to several states (e.g., idle/active power or named zero/random power [5, 6]). Moreover, they require users to provide these energy values for each operation. In reality, the energy per operation is highly sensitive to input activities (i.e., the number of 0 bits or the toggle rate of input signals). Hence, data-sensitive power models are vital for characterizing AI chips. Furthermore, many ML power models target CPU, GPU and HLS applications [17, 18], but are limited in their ability to model general accelerators.

The three aforementioned observations highlight two important aspects of AI chip design: *(1) early-stage evaluation* and *(2) agile design implementation. Early-stage evaluation* is critical to quickly understand tradeoffs between different architectures without implementation and running full EDA flows, and accurate evaluation is necessary, especially for sparse neural networks and sparse architectures, where runtime and power heavily depend on complex data patterns. *Agile design implementation* is important for users to quickly and flexibly prototype new designs in a push-button manner, thereby enabling users to validate designs at the system-level or

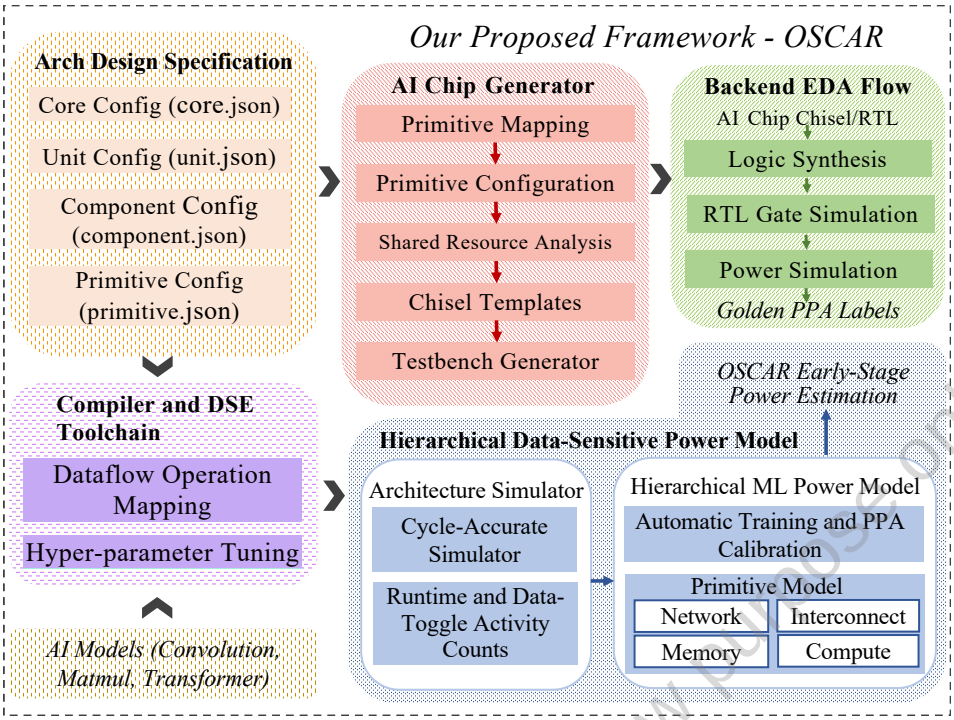


Fig. 1. Overview of OSCAR. It supports a flexible and hierarchical design space for AI chips. An RTL generator converts each design configuration automatically into the RTL implementation. An early architectural-stage power model provides accurate early-stage power estimations.

post-synthesis/physical design level, and reduce the time for engineering implementation and tape-out.

As shown in Figure 1, we propose an open-source[1] AI accelerator estimation and generation framework titled *OSCAR*. Our work's main contribution is **an intricate and accurate architecture-level power model** and **an automated hierarchical-level RTL generator for AI accelerators**. OSCAR's early power models allow designers to quickly evaluate the tradeoffs in power footprints of different AI architectures and neural networks, enabling checking the power overhead and feasibility of new designs and allowing rapid **design space exploration** (**DSE**). The generator allows designers to map high-level architectures to RTL, allowing found designs to be quickly implemented, prototyped, and verified. The advantages of having both a power model and a generator include consistency between modelling and implementation, simplification of DSE and the optimization process. Generators also allow designs to be interfaced in larger systems, allowing for system-level evaluation, debug and verification.

Externally for the user, OSCAR can efficiently explore, evaluate, and generate many diverse dataflows, such as systolic, different loop-order and tiling dataflows (i.e., different data-stationary flows) and support many operations (algorithms), such as dense, Winograd convolutions and transformer workloads. OSCAR also supports optimized reconfigurable or multi-dataflow architectures, leveraging hardware configurability to achieve better performance with minimal hardware area overhead compared with single-dataflow architectures.

[1]Our complete generator and power model development flow can be found at https://github.com/hkust-zhiyao/OSCAR

Internally, to efficiently model and generate AI chips, OSCAR proposes a four-level hierarchy (i.e., core, unit, component, and primitive levels) to model and generate AI accelerators, ultimately decomposing the entire design of any configuration into a set of common hardware primitives. In this hierarchical manner, a comprehensive and wide range of AI accelerators can be both modelled and generated by only using a small set of common primitives.

In order to boost power model accuracy and generalize to diverse designs, separate power models are built for each primitive, utilizing data-toggle features to achieve high accuracy. Unlike previous power models, which neglect toggle features or simplify toggle variations [6, 19, 20], our power model uses *data-sensitive*, or fine-grained toggle information, as features that are extracted in the architectural stage through a software model. Thus, our early architectural stage power model has better prediction accuracy. Our model also proposes a more comprehensive and diverse set of primitives, including networks and a diverse selection of **processing-element** (**PE**) primitives, compared with prior works, which focused on multiplier or memory units. OSCAR can hence be trained efficiently and enables better generality in modelling a variety of architectures.

Our key contributions of OSCAR are summarized below.

- **Hierarchical-level design space for AI Accelerators.** We define the AI chip design space to be flexible and general with a novel systematic 4-level hierarchical structure, supporting many types of dataflows, such as Winograd convolution and transformer architectures, and customization of component- and primitive-level parameters. We also propose an algorithm for multi-dataflow architectures to identify shareable hardware components to improve the PPA of the design.
- **AI Chip Generator from Design Space to Implementation.** Our AI chip generator consists of full-stack software and RTL-generation scripts. After a user selects a point in the design space and an AI workload in ONNX format to map, OSCAR will automatically schedule, memory map, and bind the opcodes to the hardware, and then provide a synthesis-ready hardware in Verilog along with testbenches and power evaluation scripts.
- **AI Chip Hierarchical Decomposition with Novel ML-based Data-Sensitive Power Models.** We propose power models based on the minimum primitives (e.g., multiplier, adder, SRAM, etc.), enabling efficient model training and accurate power evaluation. These primitive models consider fine-grained data and toggle features, which are easily obtained from an early-stage architectural simulator. These features improve power prediction accuracy.
- **A Unified Power Model and RTL Generator for AI Architectures.** The power model and generated RTL are calibrated and consistent, meaning that the power model is correlated with the actual power of generated RTL designs. This synergy between model and implementation enables users to rapidly evaluate and validate designs both early-stage and post-synthesis.
- **Efficient Exploration of AI Accelerators and Validation.** OSCAR supports efficient design space exploration with its comprehensive design space, automated RTL generator, and accurate power model. OSCAR explores millions of accelerators in hours, with a Pareto-optimal design being adopted and validated through the tape-out of a real AI chip.

In the remaining parts of this article, we will first introduce the AI design space supported by OSCAR in Section 3, then we will cover the AI chip generator and power model in Sections 4 and 5, respectively. Finally, we will discuss our design space exploration methodology in Section 6 and our work's experimental results in Section 7.

Table 1. Comparison of AI Chip Architecture Models and Generators

Method	Framework Support		Dataflow Support & Operation Configurability							Hardware Configurability	
	RTL Generator	Early Power Model	Dataflows: Loop Order	Dataflows: Tiling	Dataflows: Systolic	Operation: Reconfigurable	Operation: Dense	Operation: Winograd	Operation: Transformer	PE Array	Memory
Interstellar [16]		✓	✓	✓			✓			✓	✓
Maestro [3, 21]		✓	✓	✓	✓		✓			✓	✓
Accelergy [6]		✓	✓				✓		✓	✓	✓
NeuroMeter [20]		✓	✓	✓	✓		✓		✓	✓	✓
AutoDSE [22]	✓		✓	✓	✓		✓				✓
WinoGen [23]	✓			✓				✓			✓
AutoAI2C [24]	✓	✓	✓	✓	✓		✓		✓		✓
PowerGear [17]		✓	✓	✓	✓		✓			✓	✓
Panda [18]		✓					✓			✓	✓
Gemmini [10]	✓		✓	✓	✓		✓			✓	✓
NVDLA [12]	✓			✓		✓	✓	✓			✓
This Work	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

2 Related Work

2.1 AI Chip Design Space

As summarized in Table 1, AI chips are defined by two main aspects: (1) their operation and dataflow and (2) their hardware parameters. *Operation and dataflow* are defined as the algorithms supported and how a given algorithm is mapped to hardware, respectively. *Hardware parameters* refer to configurations, such as precisions, bank widths, modes, buffer sizes and other parameters that are hardware-specific. Existing works propose different formulations of the dataflow design space, such as a fixed set of loop orders and tilings for convolution [25], flexible tiling for Winograd dataflows [23], and output or weight-stationary dataflows for systolic arrays [10]. Current works have several limitations: (1) *Design spaces* are not complete, cannot tune the detailed configurations of the PE, and only focus on a subset of dataflows and rarely consider reconfigurable or multi-dataflow architectures, (2) Design spaces may *not have hardware generation*, for example, works like Maestro [3] and Sparseloop [26] analyze architectures but do not generate RTL, and (3) Generators *cannot easily extend to new design spaces*, because the design space and generator are tightly coupled.

OSCAR *addresses these gaps* by creating a hierarchical and flexible design space covering the commonly found dataflows used in AI accelerators along with an AI compiler. Furthermore, by providing a common library of hardware primitives that include memory, arithmetic, and networking modules, OSCAR is flexible and able to generate and model a wide range of AI accelerators and extend to architecture exploration.

2.2 AI Chip Generators

NVDLA [12] is an industrial chip generator from NVIDIA corporation, featuring a reconfigurable multi-dataflow architecture with dense and Winograd convolutions. The main bottlenecks of the work include limited configurability of the tiling and loop order (i.e., is weight stationary) and limited sparsity and systolic support.

Gemmini [10] is a recent AI chip generator that can generate a systolic array accelerator and be evaluated at the system-level with a CPU. They provide templates for systolic and spatial dataflows, but did not consider general reconfigurable and Winograd architectures. Furthermore, the hardware's pipeline is similar to that of a RISC-CPU with execute, load and store instructions, which leads to bottlenecks due to the increased bandwidth requirement of instruction fetching and decoding. Both of these limitations cause Gemmini's performance to be lower than the industrial-grade NVDLA accelerator under similar hardware settings [10, 12]. OSCAR addresses these challenges by providing generation for Winograd and reconfigurable architectures.

2.3 AI Chip Power Estimation Tools

Power models for AI accelerators and generators are characterized by the level of abstraction and the features used. Low-level abstraction power models include Maestro [3], Accelergy [6], Neurometer [20], and Zigzag [15], where the total power is a sum of the AI accelerator primitive powers. On the other hand, high-level power models such as AutoAI2C [24] characterize power at the architectural level using graph models, using high-level parameters such as tilings, loop orders, and precision. The latter approach is simpler and does not require separate models for many subcomponents, but it is less flexible and cannot easily extend to new architectures.

Maestro [3] and fixed-cost energy models take a coarse-grained approach, estimating the overall energy of the PE array and SRAM based on fixed energies per operation. Neurometer [20] and Zigzag [15] are similar to Maestro. Accelergy [6] and **Lookup Table** (**LUT**)-based energy models are based on fine-grained models, using different operation energies for different incoming data. For example, different energies are used for zero-toggling versus random-toggling inputs. The energy is thus formulated as a weighted sum, where each type of energy is counted and then summed to get the overall power. PowerGear [17] is a graph-learning-based model for HLS-based accelerators, where they use graph neural networks to estimate the power based on the HLS intermediate state. They use HLS simulation to get accurate toggle information at each hardware block. Panda [18] is a hybrid analytical-ML power model methodology originally for CPUs. It decomposes the architecture design into multiple components, which use an analytical model with an ML calibrator to tune the model. The component powers are then summed together to get the total power.

A major limitation of current power models is that *they require users to provide the energy per operation*. In other words, these energy models are calculators that rely on energy data from third-party tools such as Aladdin [5] or datasheets such as those commonly found for SRAM components. There are several limitations to this approach. (1) The task of characterizing energy per operation is left to the user, which can be a tedious, repetitive, and manual task requiring either writing HDL and running EDA flows to get accurate energy values or using analytical/empirical models, which are not data-sensitive, such as CACTI [27]. (2) Inaccurate energy per operation because of the high dependency of power on data. A LUT-based power model storing the energy of all combinations of the incoming data is unfortunately impractical. For example, for a 32-input 8-bit module, a LUT-based power model approach would require 8^{32} entries, which is intractable.

Another limitation is that some current power models, such as Maestro [3] and Panda [18], do not consider fine-grained power. Others such as PowerGear [17] do not have a robust set of features for power calibration, as they only consider toggle. There are cases where the power is a complex function of the input and outputs, such as for multipliers, network components, which are multi-cycle or have different modes. As an illustrative example, bit-serial and multi-cycle multiplier accelerators [28] are especially difficult to model with prior works because the number of cycles and thus energy is a function of the number of zero-bits in the multiplier and not the toggle.

Our work OSCAR *addresses these issues* by providing an automatic object-oriented framework in Chisel and Verilog for characterizing the energy per operation of a wide range of common low-level primitives. Based on accurate primitive-level powers, OSCAR's hierarchical power model allows characterization of a wide range of AI chips in an extensible and agile manner.

3 AI Chip Design Space For Early-Stage Model and Generator

In this section, we define our AI chip design space (Section 3.1) across four levels of hierarchy, which include core (Section 3.2), unit (Section 3.3), component (Section 3.4), and primitive levels (Section 3.5). We highlight the key user-configurations at each level, leaving the details of how configurations are transformed into RTL designs in the next section.

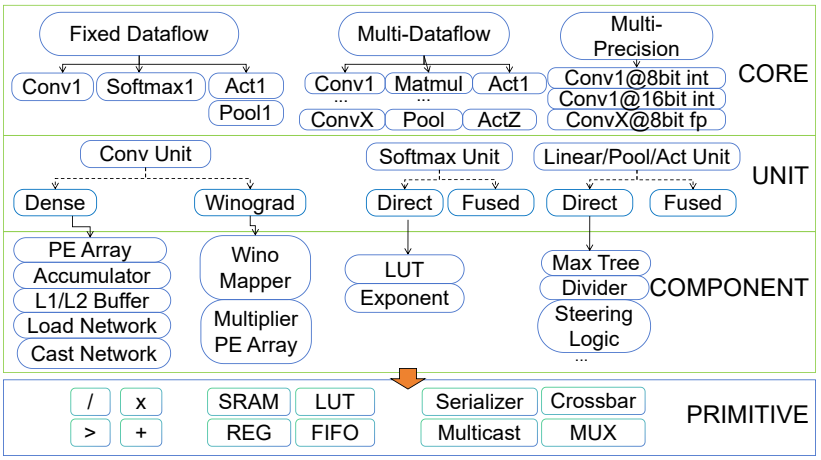


Fig. 2. OSCAR's 4-level AI chip design space hierarchy, covering core, unit, component, and primitive levels. The structure enables modular hardware generation and easy identification of any primitives that can be shared in reconfigurable and multi-dataflow architectures, greatly reducing chip area.

3.1 Overall Design Space Hierarchy

As shown in Figure 2, we model the AI chip design space through four levels of hierarchy: (1) The *core* is the top-level accelerator (Section 3.2), (2) *Unit* is the specific implementation for a given operation (Section 3.3), (3) *Components* are child modules of a given unit (Section 3.4), and (4) *Primitives* are functional units that form the basic building blocks of the design (Section 3.5). An accelerator is thus a configuration over all four levels of hierarchy. The four-level hierarchy enables efficient neural network mapping, shared resource identification, and RTL hardware generation. Our design space is thus flexible and hierarchical, providing the ability to configure the accelerator from top-level architectural parameters to detailed-level hardware primitive parameters.

3.2 Core-Level Design Space

Core-level configuration determines the supported operations (e.g., convolution, pooling, activation, matmul, softmax) and the set of units for each operation. Referring to Figure 2, a fixed dataflow core has one unit for each type of operation. For a reconfigurable or multi-dataflow core, each operation has multiple units, for example, as denoted by Conv1 to ConvX for a convolution unit. For a multi-precision core, the set of units per operation has different precisions. The top of Figure 3 shows a reconfigurable and multi-dataflow core configuration similar to NVDLA. Other configurations include operation fusion, which allows two or more dataflows to be run in a pipeline fashion for faster inference. Common fused operators include convolution + activation and matrix multiply + softmax. The detailed hardware support is in Section 4.5.

3.3 Unit-Level Design Space

Unit-level configuration determines the algorithmic dataflow and the set of child components. Convolution units support dense and Winograd. Child components consist of PE arrays, buffers, crossbars, and networks. Configurable parameters include loop order, tiling, fusion/pipeline support and parameters that affect all child components, such as clock frequency and technology node. The bottom of Figure 3 shows an example configuration for a convolution unit. This work focuses on

```
NVDLAReconfigFlow = {
  "Conv": ["Dense", "Winograd"]}

MultiPrec = {"Conv": ["Dense8", "Dense16"]}
```

```
MatMulUnit = {
"GENERAL_CONFIG": {
    "LOOP":["B","I","N"],"B":1,"N":16,"I":16,
    "CLOCK": 1,"tech":"tsmc40"}
"Components": ["PEs", "L1", ...]}
```

Fig. 3. (Top) Core-level JSON for NVDLA and reconfigurable/multi-dataflow cores. (Bottom) Unit-level JSON for a dense dataflow Matrix Multiply Unit. B, N and I refer to the batch, output and input channel tilings, and $[B, I, N]$ refers to the loop order.

dense architectures but we plan to explore the space of sparse and compression architectures in the future.

3.4 Component-Level Design Space

Each component consists of a set of primitives. We highlight common components below.

3.4.1 PE Array Components. The PE array is a set of computation primitives. The key configurations include the number and type of each compute primitive.

3.4.2 Accumulator Components. The accumulator performs reduction on the computed results from the PE Array. The accumulator's configurations include the number and type of each reduction primitive (e.g., addition, maximum, tree-like, or cycle-based primitive).

3.4.3 Buffer Components. Buffers store data and serve as pipeline elements. The main configuration is the number and type of memory primitives for each component.

3.4.4 Load Network Components. The load network is the interfacing logic between two different components, which includes deserializer and serializer components. The loading ratio is the ratio of adjacent component bandwidths. For example, a 64-bit SRAM followed by eight 8-bit PEs has a ratio of 1, whereas if the SRAM is 32- or 128-bit, the ratio is 1/2 and 2, respectively. If the ratio is 1, no load network is needed; if the ratio is larger than 1, a deserializer, i.e., parallel-in serial-out, primitive is needed; and if the ratio is less than 1, a serializer (i.e., serial-in parallel-out) primitive is needed. This component is used to interface DMA and computing components, and for systolic arrays to improve timing.

3.4.5 Casting Network Components. Casting networks broadcast a value to multiple nets. The component can be a simple register primitive with high fanout, or be comprised of a deserializer primitive, where the broadcasted data are shifted to each net over multiple cycles.

3.4.6 Winograd-Related Components. Winograd convolution includes an additional constant multiplication for the transformed weight primitive and transformed output. Other primitives are similar to those of dense convolution. Constant multiplication generally uses less power and area compared with general multiplication primitives.

3.4.7 Transformer-Related Components. Transformer operations are comprised of matrix multiplication and softmax operations. Matrix multiplication is already covered in convolution. The

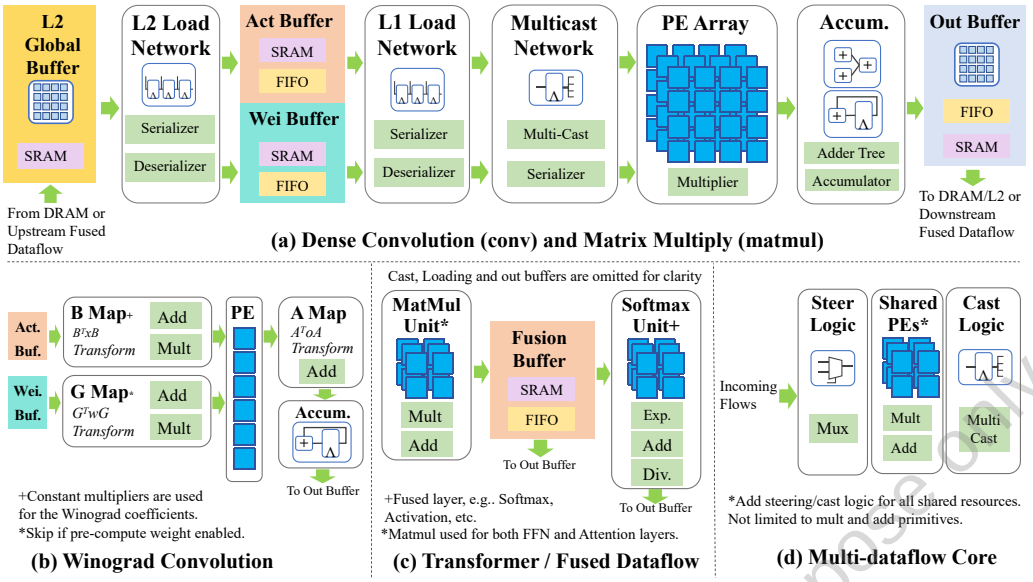


Fig. 4. **OSCAR's flexible and hierarchical accelerator dataflows.** The figure shows the primitive mapping and analysis stage, from AI chip design space configuration to RTL generation. Given an accelerator design configuration, OSCAR maps the design into hardware primitives, which are shown as green boxes.

softmax component is decomposed into exponent, addition, and division primitives. Exponents are either LUT-based or exactly calculated using polynomial approximation, similar to [29].

3.4.8 Other Components. For components not listed above, users can define their components based on OSCAR's primitives or even create their own primitive library and mapping in OSCAR's framework.

3.5 Primitive-Level Design Space

Primitives are the fundamental building blocks of the AI chip architecture. We categorize primitives into compute, memory, and network primitives. Compute primitives include multipliers, adders, and other arithmetic hardware. Memory primitives include registers, SRAMs, and ROMs. Network primitives include serializers, deserializers, multicast modules, multiplexers, and crossbars. Referring to Table 2, Each hardware primitive has their own set of configurations. We leave discussion of the primitive-level design space to the primitive-modelling section in Section 5.

4 Automated AI Chip Generator Framework

Once an architecture design space is given, OSCAR will map the design into primitives for RTL generation and power estimation. The output of our generator is a set of Chisel primitives and configurations, which will then generate RTL. OSCAR's AI Chip generation process consists of four main stages: (1) primitive mapping and analysis, which identifies the required number of each primitive from the user-given architectural description; (2) primitive configuration, which sets the detailed configurations of each primitive; (3) shared resource analysis, which identifies any resources that can be shared and adds reconfiguration logic; and (4) generation of RTL from the mapped architecture. We cover each stage in one subsection below.

4.1 Primitive Mapping and Analysis

Given the user-defined architectural description, primitive mapping and analysis identify the number of primitives and their type. This procedure is done for each unit. Figure 4 highlights the architecture of the three main units for convolution and transformers. Other unit mappings are similar, and we encourage interested readers to view the open-source code for further details.

4.1.1 Dense Convolution Units. Dense units are configured by weight/input/output precisions, size of weight/input/output buffers, tiling, and loop-order. The tilings are along the 7 dimensions I, X, Y, N, K_x, K_y, and B, corresponding to input channel, X-dim and Y-dim, output channel, kernel X-dim and Y-dim, and batch. As seen in Figure 4 part (a), a dense convolution is matched to a load network, act/wei/out buffers, multicast network, PE array, and accumulator.

The PE array consists of $I \cdot X \cdot Y \cdot N \cdot K_x \cdot K_y \cdot B$ multiplier primitives. The PE also includes an adder tree that combines inner product terms along the filter and input channel dimensions, so there are $B \cdot X \cdot Y \cdot N$ adder tree primitives each adding $K_x \cdot K_y \cdot I$ terms. To improve data reuse, weight and input tiles are broadcast into the PE array, therefore, in the multicasting network, $I \cdot N \cdot K_x \cdot K_y$ weights are each broadcasted (fanned-out) to $X \cdot Y \cdot B$ PE elements, and $(X + K_x) \cdot (Y + K_y) \cdot I \cdot B$ inputs are broadcasted to $K_x \cdot K_y \cdot N$ PE elements.

To support systolic architectures, we observe that changing the broadcasting network into a multi-cycle shift register primitive leads to systolic data movement. For spatial architectures, the shift register is 1 cycle, with fan-out equal to the broadcast amount, and for systolic architectures, the shift register cycle corresponds to the broadcast amount, and fan-out is 1. For architectures in between, the fan-out and cycle is a mix, but the fan-out and cycle number must equal the broadcast amount.

The DMA or load network components move data from main memory/global memory, or from the L1 weight/input buffers into the PE. Each has serializers or parallel to serial primitives to support different tiles. These network primitives are configured so that the correct tile size of weights, inputs enter the PE.

4.1.2 Winograd Convolution Units. Winograd is faster than dense convolution because the data is transformed into the Winograd space, resulting in fewer multiplications [23]. The overall formulation is, $O = A(GWG^T \cdot BIB^T)A^T$, where A, G, and B are constant matrices, W, I, and O are the weight, inputs, and tiled outputs, respectively [23]. The G, B, T, and their transposes' matrix multiplications are mapped into constant multipliers and adder primitives.

Figure 4 part (b) depicts the Winograd convolution unit's architecture. The architecture is similar to the dense convolution unit except with additional units for interpolation and mapping. The broadcasting and PE array components are also different in order to facilitate the Hadamard product. The PE array consists of $I \cdot N \cdot (X + K_x - 1) \cdot (Y + K_y - 1) \cdot B$ multipliers, which is fewer than that of a Dense convolution unit.

Figure 5 shows the general mapping between the algorithm and the hardware. The Winograd weight map corresponds to the G, G^T matrices, the activation map component to the B, B^T matrices, and the output map corresponds to the A and A^T matrices. The maps are implemented as a two-stage pipeline. For the G weight mapper, in the first stage, to calculate $G \cdot W$, where W is the weight tile, $I \cdot N \cdot K_y \cdot K_x$ weights are multiplied by the constant weights in the $K_y \cdot K_x$ G matrix (implemented as constant multipliers or shifter primitives), then the partial sums are added with $(X + K_x - 1) \cdot K_y \cdot N \cdot I$ adder trees summing Kx terms. The second stage is to calculate $(\cdot)G^T$, then perform the G^T matrix multiplication similarly, yielding the final $I \cdot N \cdot (X + K_x - 1) \cdot (Y + K_y - 1)$ weights. The weights are then broadcast into the PE-array with a fan-out of B. The inputs and output Winograd mappers are similar.

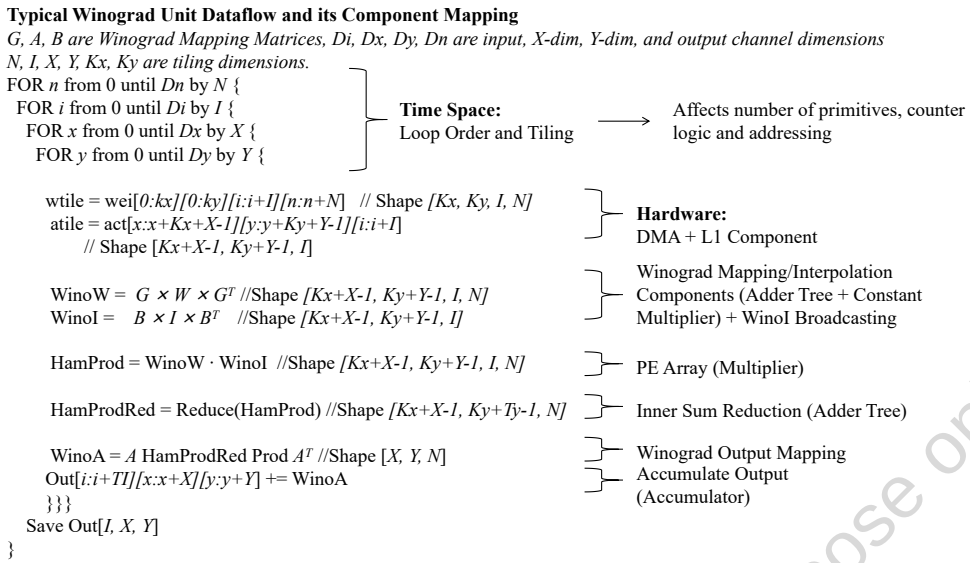


Fig. 5. Algorithm to hardware mapping. Example of a Winograd Unit. The left-side pseudo-code is mapped into hardware. Dense and other units are similarly done.

4.1.3 Transformer Units. *Transformer-based models* are composed of attention operators, which themselves consist of matrix multiplication and softmax operators. *Transformers have two stages*, namely prefill (encoder) and decoder mode [30]. Briefly, prefill computes the key-value pairs and next token for a given initial sequence, and decode uses the prefill stage's key-value pairs stored in the KV Cache and calculates only the next token in an autoregressive manner [30]. The *two stages* are different in the input sequence lengths and matrix multiplication sizes. These different modes are converted into matrix multiplication operations by translation, i.e., attention heads and sequences are merged into the output and batch dimensions, respectively.

Figure 4 part (c) depicts the transformer-based unit's architecture. There is a softmax and matrix multiply core, which is configured similarly to the convolution core. There are now only three main loop dimensions, namely the batch, input dimension, and output dimension. We have added softmax units, similar to [31], and their primitive decomposition to OSCAR. To adapt convolution kernels for transformer workloads, the convolution kernels are converted into matrix multiply units by setting the kernels to 1×1. The original convolution's kernel tiles are set to 1, and the output tilings are also set to 1. To support both feed-forward networks, decode and prefill modes, and fused attention, i.e., softmax followed by matrix multiply, the tensor calculated after the matrix multiply can be: (1) given to the softmax unit through a fusion buffer in a pipelined manner, or (2) directly stored as an activation for feed-forward network and other non-fused matmul-softmax layers.

4.2 Primitive Configuration

Once the number and types of primitives are identified, the primitive parameters are assigned. The assignment can be homogeneous or heterogeneous. Parameters are primitive-level, which will be introduced in Section 5 and Table 2. Parameters are either explicitly assigned by the user or implicitly determined during analysis. Figure 4 shows the relationship between architecture and primitives. Each primitive's primitives will be computed and stored for further processing.

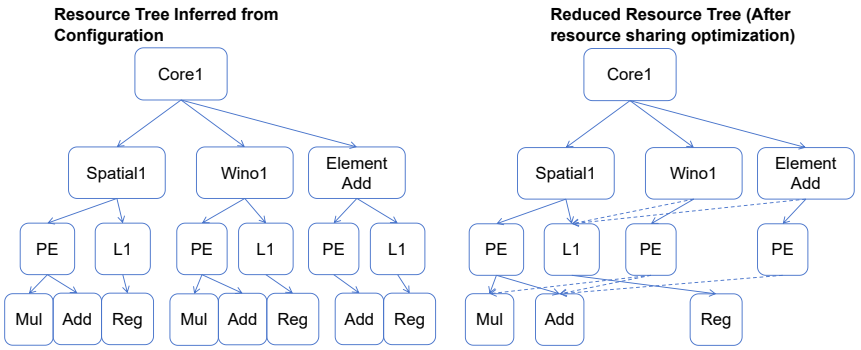


Fig. 6. Generation of resource tree and reduction of resource tree for shared resources.

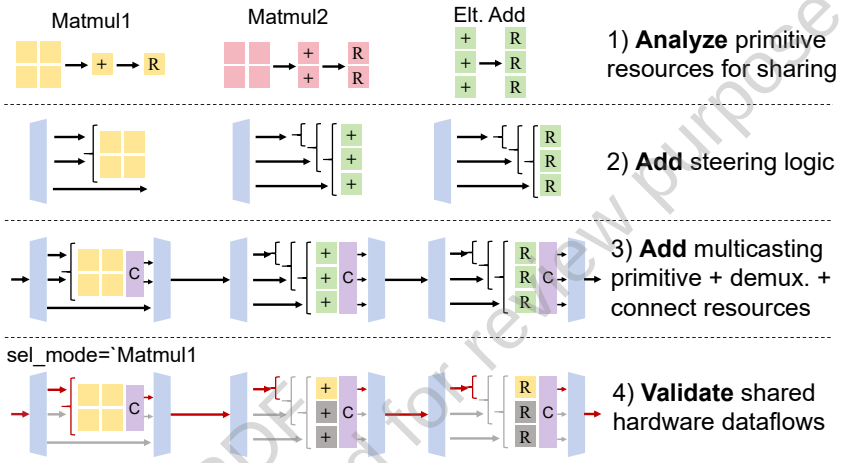


Fig. 7. Reconfigurable/multi-dataflow example of combining multiple resources together and adding corresponding steering and multicasting logic. Square blocks are multipliers unless otherwise indicated by the symbols, i.e., + for addition, R for buffers.

4.3 Shared Resource Analysis

For multiple-dataflow architectures, resource sharing needs to be considered, and steering logic needs to be added to the primitives. The shared resource algorithm identifies any primitives that are (1) the same, (2) subsets (e.g., a 32- and 16-input adder), and (3) inherently reconfigurable (e.g., bit-fusion and reconfigurable adders). Steering logic is also added. For each shared resource, at the input, a multiplexer primitive with N paths, where N is the number of dataflows going through a shared resource, is added, and at the output, a multicast primitive with M fanout, where M is the number of dataflows the shared resource broadcasts to, is added. The resource sharing idea is shown in Figure 6. Here, there is resource sharing between similar units (Winograd and Spatial Convolution), as well as between different units (Element Add Vector Unit and Convolution Units).

Generally, the transformation is functionally correct because by setting the correct select signal, the steering units will choose the desired datapath. Moreover, we perform functional testing using a similar testbench that is used to test the pre-shared architecture to ensure correctness. For example,

Figure 7 illustrates the process of adding steering logic to support multi-dataflows. Steps 1–3 add new hardware to support the sharing of hardware between two matrix multiply and one element-wise addition units. Step 4 shows that when the select mode is matmul1, the correct and desired dataflow is selected. Other dataflows are similarly selected.

During inference, each operation in the AI model is mapped to a dataflow in the multi-dataflow accelerator, which is done using the fast architectural performance and PPA models, which are discussed in Sections 4.6 and 5. The correct select signals will then be given to the architecture to select the desired dataflow for each operation.

4.4 Generation of Design RTL and Workload

Once the architecture is mapped onto primitives and configured, we generate the hardware code via Chisel. Each primitive has readily available Chisel templates, which are then parameterized according to the configurations mapped earlier. OSCAR provides model weight loading, testbenches and power evaluation scripts for the generated primitives and overall accelerator.

We employ a bottom-up approach based on the 4-level hierarchy and template-based coding structure for integrating the primitives into the overall architecture. First, each component has Chisel-based templates with "slots", or sub-module instantiations, which correspond to the required primitives. For example, the PE component instantiates an array of multiplier primitives. For shared resources, multiplexers and casting networks are generated to connect different dataflows to the shared resource primitive. Second, once each component is generated, OSCAR generates each unit, instantiating the required primitives and components. Each unit has several configuration registers, corresponding to unit-specific global parameters; for example, convolution units have input, output, and kernel size. There is also a start signal which initiates the unit once its configuration registers are all set. For reconfigurable architectures, the configuration register also includes the dataflow id/type, which controls the select signals for the multiplexers that determine the different dataflows. Each unit also has a common port interface, with memory read/write and configuration read/write signals similar to the AXI handshaking protocol. Finally, at the core-level, each unit is connected to a common bus with arbitration. The memory units, such as the L2 buffer and Main Memory, including any secondary main memories, are connected to the bus and arbitration is conducted for memory requests to the same address space. Each unit's configuration registers are memory-mapped, with addresses being calculated during generation.

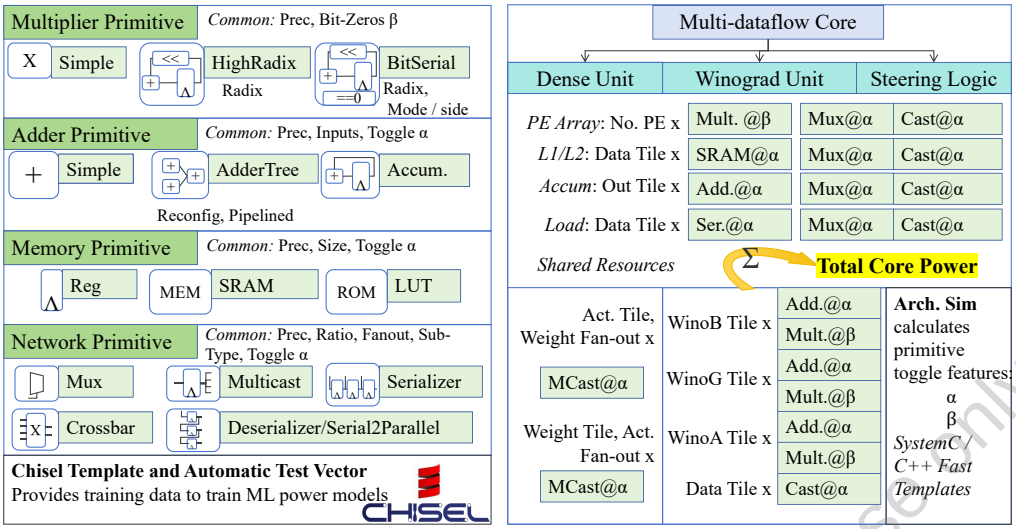
4.5 Fused Operator and Pipelined Execution

OSCAR supports fusion operators. Instead of executing operations serially and accessing main memory every time for input/output data, operations can be executed in a more efficient manner by storing the output activations in cache and then reading the activations in cache for the next operation. Costly memory accesses are reduced, and thereby the execution time is also reduced.

The main hardware needed to support fusion includes:

(1) Multiplexer primitives to steer the output to either another unit's load buffer or to the global buffer. See part (c) of *Figure 4 Transformer/Fused Dataflow* for an example of such steering.
(2) Control circuitry to control the pipeline and indicate when a unit's output tile is ready to be processed by the next unit.

During compilation, any fused operator identified in the operation tree of the AI model will be mapped to fused operator hardware, thereby improving performance. The correct fusion dataflow path will then be selected during inference.



(a) Power Model Training (b) Power Model Inference (NVDLA-like example)

Fig. 8. Hierarchical data-sensitive power model. Primitive powers are trained using ML models on various data. (a) shows the training of primitives, and the (b) at inference for an overall accelerator.

4.6 AI Architecture Performance Simulator

OSCAR has a built-in cycle-accurate performance simulator based on analytical models in SystemC and C++. The simulator parses the high-level parameters, estimates the ideal compute-bound and memory-bound latencies based on throughput and bandwidth analysis, and picks the maximum of the two as the estimated latency. For example, for convolution units, we first compute the throughput and latencies: (1) the compute-bound throughput equals the multiplier PE's throughput, and the latency equals the total MAC count divided by the tile times the PE latency, and (2) the memory-bound throughput equals the DMA's throughput, and the latency equals the total size of input and weights divided by the bandwidth. Convolution units are generally compute-bound, so the overall latency equals the compute-bound latency. Our simulator yields latencies accurate to 2% compared with RTL simulations.

The simulator also allows calculating important toggle features for each workload because the simulator is cycle-accurate and can estimate the toggling at each tensor. These features are critical for OSCAR's power modelling, which we discuss in the following Section 5.

5 Hierarchical Power Modeling

Our framework provides a hierarchical and accurate early-stage power model. Figure 8 shows the training and inference stages. Figure 9 shows how OSCAR takes a bottom-up approach to build the power model of the top design from primitives. OSCAR's power model is early-stage and can be used at the architectural design stage. Furthermore, the features are based on data-sensitive toggle statistics, which are readily available at the architectural level. Finally, we have independent power models for each primitive type. We now introduce OSCAR's power model and the training procedure.

5.1 Hierarchical Power Model

OSCAR estimates the power hierarchically, beginning at the primitive level as shown in Figure 9. After primitive mapping, OSCAR estimates the power of each primitive with an independent model and then sums them to get the overall power. The advantages of the bottom-up approach to power estimation include: (1) *Fast power model training and calibration*, because only primitive-level modules need power models, obviating the need for slow power simulations of the whole design to gather higher-level training labels. (2) *Easy-to-leverage data-sensitive power features*. One can quickly estimate the data-toggle of different primitives and get accurate power estimations. (3) Improved extrapolation capability to better generalize and analyze new accelerators.

5.2 Primitive Model Features

Primitives are characterized by their type, hardware features, and data-toggle features, as shown in Table 2. Type (t) refers to the specific implementation of the primitive, hardware feature (h) relates to any parameterizations, and data-toggle feature refers to data-sensitive information. We identify two key metrics for capturing toggling activity, namely signal toggle (α) and zero-bit count (β). Signal toggle α is the number of bit toggles of a variable between cycles. Zero-bit count β is the number of zero bits of a signal at a given cycle. We observed that these two features are critical in modelling the power of various primitives: (1) α captures power variations across time and thus are important for low or single-cycle primitives, such as adders and registers, because signal toggle directly reflects capacitance switching and thus dynamic power at the output and input ports, which comprises most of the overall power, whereas (2) β captures power variation across space, and are thus important for multi-cycle primitives such as multipliers, because zero-bit counts can reflect a primitive's internal switching behavior over multiple cycles. An important example of how zero-bit counts reflect power is for shift-based or bit-serial multipliers, i.e., a high β means more terms are accumulated, computation is more frequent, thus power is higher. α cannot capture this behaviour because there can be little toggle between cycles, i.e., 1111 changing to 1110 has $\alpha = 1$, but the β is still high at 4 and 3. Hence, both types of toggling activity α and β are important for accurate power modelling at the primitive level.

To estimate these toggle features during inference, because an accelerator's loop order and tiling are known, the task of calculating the zero-bits and signal toggles is straightforward: (1) traverse the algorithm in loop order, and in the inner loop read out the required data, (2) calculate the zero-bits β by computing the population count of the data, (3) calculate the signal toggle α by performing an XOR of the data against itself in the previous cycle and then compute its population count. For signals that are not directly available from activations or weights, such as the accumulator inputs, our templates provide an "accumulate" variable, which first accumulates the data and then performs data-toggle calculations. We implement the simulator in C++ for high efficiency with a template structure for different dataflows. The toggle features are all computed and readily available at the architecture stage, and thus can be used for architectural-level power estimation.

5.3 Overview of Power Model—Primitive Training

We collect a comprehensive training set for each primitive. The training features are listed for each primitive in Table 2 and the detailed values in Table 3. Given a set of hardware and type features, our script generates a set of input vectors that maximally vary the data-toggle and bit zeros at both the inputs and outputs. For example, for an 8-bit primitive, the script automatically generates input data with zero-bits or toggling from 0 to 8, i.e., set the input value from 0 to 255. The vectors are then used to simulate the primitive and generate traces, which are used to collect ground-truth power labels via a conventional EDA flow. Because power calibration is done at the

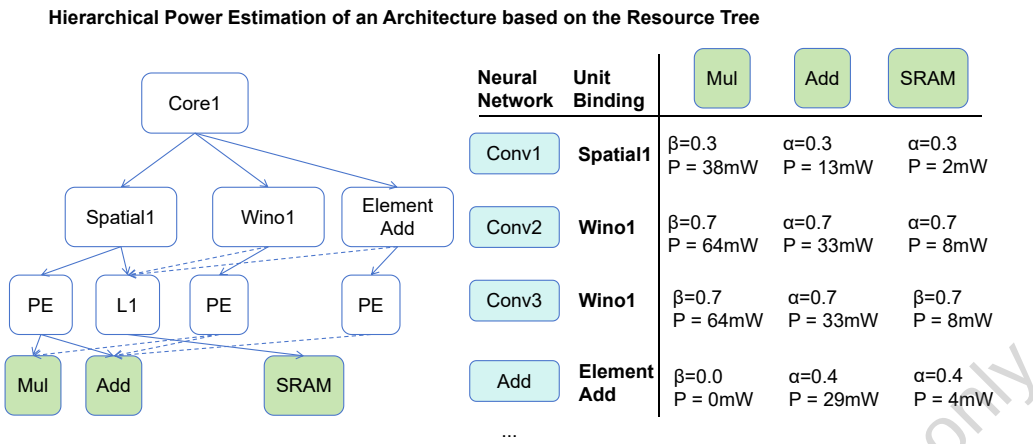


Fig. 9. *Power estimation example of a reconfigurable architecture* based on the primitive hierarchy power model. The left side shows the architecture represented in the 4-level hierarchy after resource sharing optimization, and the right side shows the power estimation calculation. For each operation and its hardware binding, OSCAR estimates the power by finding the active primitives, estimating toggle features, and summing the powers based on OSCAR's pre-trained power models. Interconnect and the MLP calibration for total power are omitted for clarity.

Table 2. Primitives Parameters and Data-Sensitive Features

Primitive	**Primitive Parameters**		**Toggle Feature**
	Type (t)	Hardware Feature (h)	
Adder	Simple	prec, pipelined	Toggle α
	Tree	prec, #input signals	Toggle sum $\sum \alpha$
	Accumulate	prec, #input signals	Out toggle α_{out}
Multiplier	Simple	prec	Zero bits β
	High-radix	prec, radix	Zero bits β
	Bit Serial	prec, radix, mode	Zero bits β
	Bit Fusion	prec, radix	Zero bits β
Memory	Reg	prec	Toggle α
	SRAM	sub-type, bank width, mode	Toggle α
	ROM/LUT	sub-type, bank width	Toggle α
Network	Mux	prec, #input signals	Toggle sum $\sum \alpha$
	Crossbar	prec, #inputs/outputs	Toggle α
	Multicast	prec, fanout	Toggle α
	Serializer	prec, sub-type, ratio	Toggle α
	Deserializer	prec, sub-type, ratio	Toggle α
Other	Max	prec, #input signals	Toggle α

Primitives are characterized by their type (t), hardware features (h), and toggle features (α and β). The zero-bit count (β) is the averaged bit-level zeros of a signal. The signal toggle (α) is the sum of bit toggles between cycles, i.e., hamming distance of the signal between adjacent cycles.

primitive-level, the data feature and power label collection process is fast, modular and easy to automate.

The power model f of each primitive can be generally formulated as below, with P being output power:

$$P = f(t, h, \alpha, \beta), \tag{1}$$

with inputs including primitive type t, parameters h, signal toggle α and zero-bit count β. We explored a wide range of ML models for f, including MLP, random forest, XGBoost [32], and so on. We adopt XGBoost, a gradient tree model for the power model of all primitives, which yields the best accuracy and training time during validation.

5.4 Detailed Primitive Models

Adder primitives support three main types of adders(t): simple, tree-based, and accumulation-based. "Simple" is the Chisel-native adder implementation using the "+" operator. "Tree" is an adder tree. "Accumulate" is a cycle-based adder. Each primitive type can be customized based on the input/output precisions, number of input signals, and whether each stage is pipelined. In the power model of adder primitives, the adder type t is encoded using one-hot encoding, and the other parameters h are integers. To address the issue of multiple input signals of an adder, instead of assigning a toggle feature for each input, we use the sum of the input signals' toggles along with the output signal's toggle. The power model of an N-input adder is

$$P_{\text{adder}} = P\left(h, t, \sum_{i=1}^{I} \alpha_{\text{inputs}}(i), \alpha_{\text{outputs}}\right), \tag{2}$$

with each ith input signal's toggle denoted as $\alpha_{\text{inputs}}(i)$.

Multiplier primitives support four types of multipliers(t): simple, high-radix, bit serial, and bit-fusion-like. "Simple" is the Chisel-native implementation using the "*" operator. High-radix is a shift-based accumulate multiplier like the Booth multiplier. Bit serial is also a shift-based accumulator, but can complete once the shifted multiplicand is zero or has zeros, like in Pragmatic [33]. Bit fusion is a reconfigurable multiplier supporting user-specified precisions [28]. Using bit-level zero counts β is sufficient for estimating power, because the number of zero bits in the multiplicand determines the effective number of shifts and additions. Each multiplier supports full customization of the two input signals, precision, radix, and shifting side. For further implementation details, we point our readers to Ref. [34]. The multiplier type and mode are encoded using one-hot encoding; the other parameters are integers. The power model for a multiplier primitive is given below, where the two input signals are the neural network's weights and activations:

$$P_{\text{multiplier}} = P(h, t, \beta_{\text{weights}}, \beta_{\text{act.}}). \tag{3}$$

Memory primitives support four main types(t): register/register files, 2-port SRAMs, 1-port read-only LUTs, and ROMs. SRAMs are configured based on bank size (i.e., the unit bank size in a banked SRAM component) and the mode (i.e., whether it is read or write). Sub-type refers to the type of SRAM, since a memory compiler supports different versions of SRAM. The SRAM type and sub-types are one-hot encoded, the other parameters are integers. We do not directly use the address as a feature and instead use the read and write data's toggle as a feature. The power model for a memory is

$$P_{\text{memory}} = P(h, t, \alpha_{\text{inputs}}). \tag{4}$$

Networking primitives support five main types(t): multiplexers, crossbars, multicast, serializers, and deserializers. Multiplexers are configured based on the number of inputs and precision. Crossbars, or N-M networks, are implemented as several multiplexers in our work, but can be extended to incorporate other kinds, such as partial crossbars, butterfly networks, and other networking logic. Multicasting units are registers with high fanout. Serializers, similar to input queues or FIFOs, serve as a bridge between high-throughput to low-throughput components, and deserializers are

vice versa. They are critical in systolic architectures. Serializer and deserializers have multiplexer and shifting implementations, which we refer the user to [35] for details. All networking primitives are characterized by precision, fanout, and the number of input/output signals. Serializer and deserializers have a ratio feature, which refers to the ratio between input and outputs. In the case of 1, they are simply multicasting primitives. All features are integers except for type and sub-types, which are one-hot encoded. For multiplexers and crossbars, we do not directly use the select signal as a feature and instead use the input and output data's toggle as features. The power model for a network primitive with I-inputs and O-outputs is

$$P_{\text{network}} = P\left(h, t, \sum_{i=1}^{I} \alpha_{\text{inputs}}(i), \sum_{o=1}^{O} \alpha_{\text{outputs}}(o)\right), \tag{5}$$

where input and output toggles are $\alpha_{\text{inputs}}(n)$ and $\alpha_{\text{outputs}}(m)$.

Other primitives are modelled similarly, and we encourage the reader to view the models in our open-source code base. Our architectural-level power model and generator support DSE well, thus we can generate high-quality accelerator designs in a short time.

5.5 Interconnect Modelling and Primitive Summation Calibration

Apart from primitive power modelling, interconnect power and calibration between the primitive summation to the real golden truth label is required for power models to work for real designs. The need to model interconnect and also consider total power calibration was observed in many prior works [17, 18, 20]. We now describe how OSCAR addresses these two issues.

As shown in part (b) of Figure 12, OSCAR models interconnect power, i.e., the power of wires and RC parasitics, through implicit and explicit interconnect:

(1) **Implicit interconnect.** Interconnect power is included in the ground truth label implicitly. Implicit interconnect power is automatically included in the label, either as a simple wireload model for post-synthesis ground truth power or as an RC parasitic power model. Our model will then be calibrated against the power label, which includes implicit interconnect by default.
(2) **Explicit interconnect.** Interconnect power also exists between primitives, such as the fan-out, fan-in nets, long nets, and so on. We use primitives as a proxy to estimate explicit interconnect power by using buffers and network primitives. The primitives are configured based on the topology of the dataflow and are used effectively to fill the gap between the total power and the primitive summation power.

To further reduce the gap between total power and the primitive summation power, an MLP calibrator, similar to that used in works [18], is used. This ensures that our power model can capture any non-linear effects and power optimizations done by the EDA tools. We find that a simple MLP calibrator with 1-2 hidden layers and input and hidden size equal to the number of primitive models is sufficient for performing MLP calibration.

5.6 Power Model Automation and EDA tools

Our power modelling framework is highly automated and suitable for users to introduce custom primitives. The user essentially provides the hardware code in Chisel (or as a blackbox Verilog module) and a sample testbench that drives the test vectors into the primitive's input ports using Chisel tests. The test vectors are configured to be either random toggling, fixed bit switching, or custom vector patterns that drive the DUT under different toggling conditions. Afterwards, OSCAR provides a user-friendly "FullTester" class that automatically generates the simulation, synthesis,

and power characterization scripts to drive EDA tools. We provide the capability to customize the technology node and EDA scripts for users without commercial tools.

As an illustrative example, we highlight the steps for characterizing the power model for a new primitive implementation or on a new technology node.

(1) Add the new primitive into the Chisel framework. For Verilog designs, the primitive can be added via Chisel blackboxes, using the Verilog as the implementation.
(2) Set the input patterns for characterizing the power. The framework provides several pre-made templates, such as random data, low to high toggle input patterns, bit flip patterns and provide a good coverage of the power of the primitive in an automated fashion. We have added a figure to show these patterns in a more clearer manner.

5.7 Toggling Activity Simulator for Inference and Real Workloads

OSCAR provides a fast architecture simulator that can calculate toggling activities at the *per-cycle granularity if needed, as mentioned earlier in Section 4.6*. Given a *hardware configuration space and a workload*, our architecture simulator not only calculates the total cycles and runtime statistics but is also able to *calculate the toggling activities at each tensor and primitive input/output accurately*. In theory, the per-cycle power or energy can be calculated, and according to our observation, this power can be accurately captured by our power models. Thus, overall power can also be calculated for any workload. Generally, this architectural simulator is fast and can be completed within minutes for larger networks, such as ResNet50. In practice, average power is calculated to reduce architectural simulation time.

We *highlight the steps for toggling activity and power characterization* for a new workload.

(1) The input and weight data are propagated through the accelerator. A cycle-accurate simulator allows the data that enters and leaves each hardware module to be simulated accurately.
(2) The value's toggling and zero-bit parameters are then calculated rapidly at each hardware module's inputs and outputs.
(3) The toggle and zero bit features are fed into the corresponding power models, and the primitive powers are combined together to yield the total power.

Both weights and inputs are expected to toggle for different workloads. The weight toggling can be pre-calculated easily. To estimate the input toggling of new workloads, one has to run the toggling simulator on the new benchmark and input, which will generate the corresponding toggling of the input signals of the hardware. This toggling feature can then be used in the power model. The toggling simulator is fast because it is purely software-level based and does not require any hardware or RTL simulation.

Real workloads generally do not have the maximum amount of toggling, because real workloads have much more sparsity, which can be a result of zero weights, pruning, or activation functions [36]. As a result, for our DSE experiments, which we use to run real workloads after tape-out, we use real workloads and average and not maximum toggle patterns to get the average power in the experimental studies.

6 Design Space Exploration

We introduce one application well-supported by OSCAR. For a given set of workloads and design criteria, design space exploration involves finding a set of architecture design parameters that achieve Pareto-optimality of PPA. The brute-force approach is to use the PPA model for all possible designs. Below is the algorithm used for hierarchical design space exploration.

The algorithm seeks to find the Pareto-optimal designs within the design space of hyperparameters (*ds*) evaluated over a benchmark neural network (*nn*). The goal is to find the best designs with

ALGORITHM 1: Hierarchical Design Space Exploration

```
nn = full neural network
ss = simple sample
ds = design space hyper-parameters
// 1 - Component-Level Exploration
for c ∈ AllUniqueComponents(ds) do
    CachedCompPPA ← save ComponentPPA(c, ss);
// 2 - Unit-Level Exploration and Pareto
for unit ∈ AllUniqueUnits(ds) do
    for c ∈ unit.comp do
        UnitPPA += CachedCompPPA[c]
    CachedUnitPPA ← save UnitPPA
u_op ← GetPareto(CachedUnitPPA);
// 3 - Pareto-Optimal Sampling and Binding
for op ∈ nn do
    for u ∈ u_op do
        CacheBindedPPA ← save UnitPPA(u, op)
// 4 - Core-level Exploration
ops = TotalSupportedOps(nn)
for op ∈ ops do
    for n = 1 → min(u_op,pareto, maxn) do
        dataflows += sample(u_op,pareto,n)
    CacheCores ← dataflows
for d ∈ CacheCores do
    for op ∈ nn do
        u = minarg(CacheBindedPPA[d])
        CorePPA += u
    CachedCorePPA ← CorePPA
// 5 - Return Found Designs
FinalDesigns ← GetPareto(CachedCorePPA);
```


minimal runtime and best overall PPA. We now describe the algorithm in depth, detailing the key methods and functionalities,

(1) **Component-Level Exploration:** We independently evaluate all used components (*AllUniqueComponents*) on a small workload (i.e., sample of layers) which uses our early-stage PPA model (*ComponentPPA*), saving the estimated power and performance into *CachedCompPPA*.
(2) **Unit-Level Exploration:** All unique units are iterated, each component's PPA (*UnitPPA*) is looked up from the previously saved *CachedCompPPA*. Then the total unit PPAs are saved into *CachedUnitPPA*, and the Pareto-optimal unit designs based on the small sample are saved into u_{op}.
(3) **Pareto-Optimal Sampling and Binding:** For each operation in the full neural network (*nn*), and for each possible unit binding from u_{op}, the PPA is recalculated using *UnitPPA* with our early-stage models on the selected unit binding and full neural network layer.
(4) **Core-Level Exploration:** This stage explores the possible reconfigurable bindings. The number of dataflows *n* is swept from 1 to *maxn*, the maximum allowed number of dataflows

in a reconfigurable core, i.e., 16. For each type of operation in the neural network, a greedy unit binding selects the best set of hardware for each type based on some PPA metric using *sample*. These different dataflows are saved into *CacheCores*. To get the overall PPA, each dataflow is iterated, and for each operation in the neural network, the best dataflow binding (i.e., pair between dataflow and operation) is identified. The best pairing is selected by selecting the best PPA unit from all possible bindings. The previously saved *CacheBindedPPA* is used to get the unit PPAs. Then, the *CorePPA* is saved into *CachedCorePPA*.

(5) **Return Found Designs:** The Pareto-set of different bound units from the previously saved *CachedCorePPA* is combined to form the Pareto-set of reconfigurable cores, which are returned to the user.

Our heuristic algorithm yields large benefits in runtime by reducing the design space and PPA simulation times. Component-level exploration and Unit-level exploration greatly reduce the simulation time by only running on small layers instead of the whole layer. Pareto-optimal sampling reduces the design space by only evaluating the PPA on Pareto-optimal unit bindings. Core-level exploration greatly reduces the design space of reconfigurable architectures by greedily binding the unit with the best PPA metric (i.e., lowest power, best performance, lowest energy), reducing the space from an exponential to a linear order.

Our heuristic algorithm finds designs close to the real Pareto curve and greatly reduces DSE search time. We found that our approach can find the designs on the Pareto-optimal curve in linear time compared with the brute-force approach, which is exponential.

7 Experiment

We introduce our experimental setup in Section 7.1, our results for power modelling accuracy for several typical AI architecture cores (Section 7.2), perform an ablation study on our power model (Section 7.3), compare several generated designs across multiple convolution and transformer benchmarks (Section 7.4), demonstrate our power model in design space exploration and then show its ability to model an industrial taped-out AI chip (Section 7.6).

7.1 Experimental Setup

As shown in Table 3, we describe the setup for power model training, with the configurations and training data space. For each primitive, we sweep across the primitive type t, hardware features h, and data-toggle features. OSCAR generates Chisel code for each primitive, which is then compiled into RTL, synthesized using the TSMC 40nm library using Synopsys DC Compiler, and simulated using Synopsys VCS. Then, Synopsys Primetime Power Analysis PTPX is used to simulate the average power. Each primitive has around five thousand or more labels. Once training labels are collected, we experimented with a variety of ML models in Scipy, including neural networks, linear, and tree-based models. We found that gradient-boosted tree methods have good training speed and validation accuracy, so XGBoost is adopted for modelling all primitives [32]. We use the default model parameters and logarithm-based normalization for the features. The training time for each primitive is less than 40 seconds.

Next, Table 4 shows the setup for power modelling accuracy comparison. Unlike the training labels that were generated from random synthetic data of various toggling rates, the testing labels are generated from workloads in neural networks. As a result, the toggling behaviour is no longer ideal, but will be affected by loop order, tiling, the type of neural network operation, and any other hardware optimizations enabled. To get the ground truth testing labels, we use a similar flow to power model training and run the logic and power simulation for thousands of cycles to get a stable estimate of the average power instead of running the entire workload. We compare OSCAR against two representative power model baselines, Maestro [3] and Accelergy [6], both of which require

Table 3. The Space of Primitive-Level Training Data

	Primitive Features	Configurations
Primitive	Types and Modes	one-hot, see Table 2 types (t)
	Precisions	4,8,16,32
	Module Inputs/Outputs	2,4,8,16,32,64
	Radix	2,4,16,64,256
	Ratio	1,2,4,8,16,32
	Fanout	1,2,4,8,16,32
	Bank size	(16,256), (32,256), (64,256)
Primitive Input	Data Sparsity	1, 1/2, 1/4 ... 1/64
	Bit-Zero	0, 1, 2, .., prec
	Data Type	INT, Fixed Point
Tech	Clock (GHz)	1, 0.5, 0.1
	Cap Load	1, 0.1, 0.01
	Node	Tsmc40 Typical Corner
	Logic Synthesis	Design Compiler
	Gate Sim.	VCS
	Power Tool	Primetime + Powertime PTPX
	Arch Sim.	Python and C++
	HDL Lang.	Chisel, Verilog

We sweep across the hardware, type and toggle features for each primitive to collect training features and run the golden EDA flow to get the power labels.

Table 4. Key Hierarchical Parameters for the AI Chip Design Space for Power Modelling

Level	Design Space	Configurations
Core	Multi-Precision	4, 8, 16, 4/8, 4/8/16, 8/16
	Num of Dataflows	1, 2, 4, 8, 16
Unit	Op. Units	Dense, Winograd, Matmul, Transformer
	Loop Order	WS, IS, OS, and 8 others
	Single Var. Tiling	2, 4, 8, 16, 32
	Processing Elements	64, 128, 256, 512
	Winograd Kernels	3, 5, 7
Component	Systolic Load	1/32, 1/16, ..., 8, 16, 32
	Systolic Cast	1, 2, 4, 8, 16, 32
	Inter-PE	True or False
	Compression	Input/Weight/Output Zero Map
	Memory Widths	1, 2, 4, ..., 32 Tiles 1/2, 1/4, .., 1/32 Tiles

The design space is exponential in size and cannot be fully exhausted. We generate a variety of accelerators and use Conv Layers, Transformers Prefill + Decode, and ResNets as the workload. The workload and technology parameters are the same as in Table 3.

user-given unit energies, which we provide from our trained primitive models. Because many primitives were not covered by the original works, for fair comparison, we faithfully implemented each baseline for each primitive and used these Accelergy-like and Maestro-like models for comparison. We also compare OSCAR against several ML-based models such as Panda [18] and

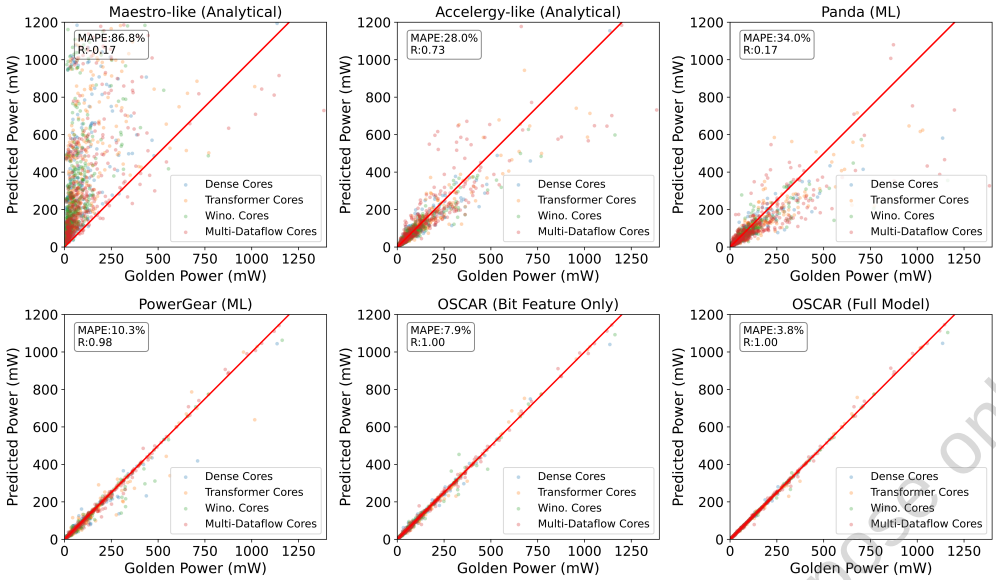


Fig. 10. Modelling of total power of several AI cores running on a wide range of sparsity and bit-zero workloads. Comparison of OSCAR against several baselines.

PowerGear [17] with respect to total power and modify the original implementations to target AI accelerators.

7.2 Power Model Accuracy

Figure 10 and Figure 11 compare OSCAR against several ML models, Panda [18], PowerGear [17] and a version of OSCAR without toggle features, and two representative analytical baselines, Maestro-alike [3] and Accelergy-alike [6], showing per-component power and the total power, respectively. We extend the baselines by incorporating modelling for interconnect, crossbar, the Winograd mapper and other components that were missing in the original implementation with the original work's methodology. We use **mean average percentage error** (**MAPE**) and Pearson Correlation (R) to evaluate predicted power against the ground-truth post-synthesis power from commercial simulators.

Figure 10 shows that OSCAR outperforms baselines with an error of 3.8%, in comparison with 86.8% and 28.0% for Maestro and Accelergy, respectively. The ML models are much more accurate at 34% and 10.3 % compared with analytical models due to their better ability to model non-linear relationships between architectural power and hardware configuration parameters. OSCAR's R correlation is 0.99 and thus rounded to 1.00. For the ablation study, we implement a hardware-only version of OSCAR without data-sensitive features (α and β) and (α), and the MAPE and R are worsened to 0.17%. The accuracy of the bit feature-only model is relatively high, but with both toggling features, the model achieves the best modelling results, suggesting the importance of data-sensitive features. For baseline methods, Maestro-like fixed-cost power models overestimate the power and show poor correlations. Accelergy-like power models are more accurate because they consider the sparsity of the data and account for lower power due to low toggling activity. Yet, they cannot capture the entire toggling spectrum, and thus still have relatively high errors. PowerGear does not have bit-zero features and thus cannot capture nuanced powers of multipliers

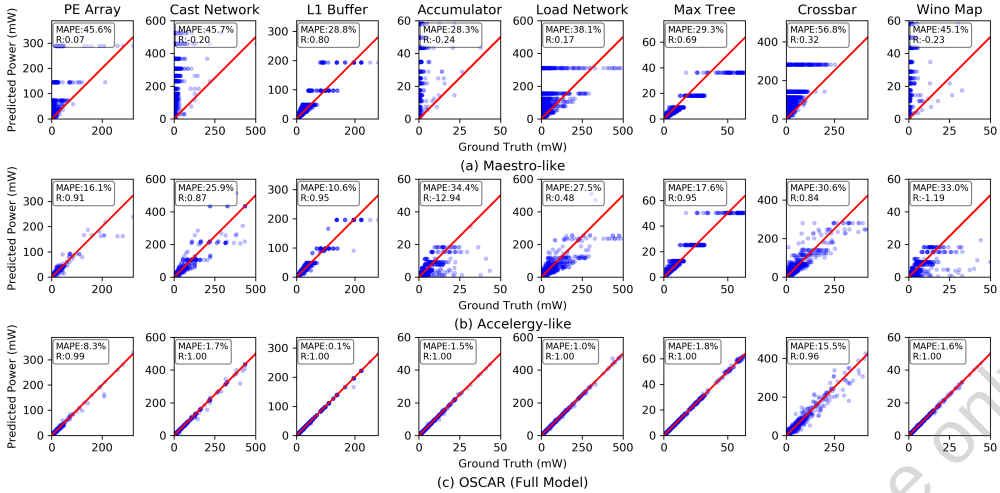


Fig. 11. Component-level power modelling accuracy. (a) The Maestro-like baseline assumes a fixed power cost. (b) The Accelergy-like LUT-based baseline uses zero-toggle and random-toggle power costs. (c) Our method OSCAR uses ML models with data-sensitive features. The power is from different convolution layers, i.e., from ResNet50, with various data sparsity, including bit-level and value-level sparsities.

and networks, resulting in higher error. Panda does not capture toggling at the fine-grained level, therefore has a similar error to Accelergy.

Figure 11 shows the component-level accuracy. OSCAR accurately predicts the power for all components with minimal error and higher correlation compared with prior works. Computation components such as the PE array, max tree, and accumulators have error around 1.6 to 8.3%, and memory and network components from 0.1 to 1.7%. These errors are significantly lower than prior art. Even the crossbar, which has 15% error, still outperforms prior art because input/output toggles are considered at the fine-grained level. Generally, more complex components (i.e., multipliers in PE array) and components with many internal signals (crossbars) are generally harder to model accurately with only knowing input/output information. For baselines, Maestro-like power models generally overestimate the power and have flat plateaus because they assume all primitives consume the same amount of power. Accelergy-like models are more accurate than Maestro-like models because they are sensitive to data-zeros and zero-toggles. Yet, there remain regions of flatness because Accelergy cannot handle bit-zeros and powers between random and zero-toggles. For example, for a workload that is non-sparse but has an average of 2 compared with 6 bit-zeros, Accelergy would predict the same power even though the latter case has lower power. OSCAR can capture these data-sensitive differences.

Finally, although our power model is based on 40nm MOS planar technologies to match our tape-out as described later in Section 7.6, we believe that our modelling methodology will work on FinFet or nano-scale technology nodes as well, for two reasons (1) FinFet leakage power is captured by our ML models because it is largely static and corresponds to a bias term in the ML model (2) dynamic power is correlated to our toggling features. We conducted several similar studies on the Open-source 7nm library [37] and observed $>$ 90% power accuracies and $>$ 0.9 correlations against golden labels.

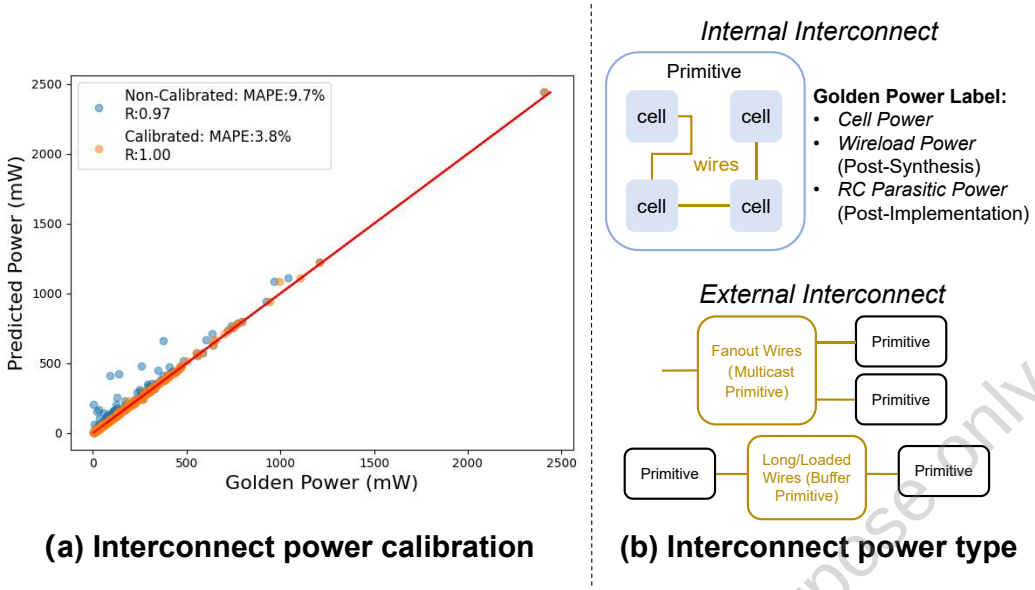


Fig. 12. Ablation study of interconnect power modelling. Internal interconnect power is included in the golden power labels, whereas external interconnect power is captured by OSCAR's buffer/multicasting primitives. (a) shows interconnect power calibration over the set of multi-dataflow cores using the same setup as DSE and power model experiments. (b) shows the two main types of interconnects.

7.3 Ablation Studies of Power Model

A key limitation of naive primitive summing is that it cannot directly capture interconnect parasitics, but this issue can be resolved by adding external interconnects and an MLP model for calibrating the summed primitive power models against the golden power as described in Section 5.5. Generally, interconnection powers are easily captured by the original primitives because the label, which includes interconnect powers, is learned by the ML models. For interconnects between memory and the PE array, we model the power using a network primitive (i.e., multicast) and add it to the original list of primitives, and we find that MLP or XGBoost models are sufficient. For higher accuracy, we use the calibrated OSCAR method in all the other power and DSE experiments.

Figure 12 shows an ablation study of power modelling before and after adding the additional primitive model for interconnects. The results are shown for multi-dataflow and reconfigurable cores. Part (a) of the figure shows that without considering interconnect, the error increases by 10% or more. The error is worse for smaller designs, as interconnect plays a larger role in the total power.

Table 5 shows an ablation study of the MLP model used for calibrating a PE array against the golden ground truth power label after post-synthesis across various sized **PE arrays** (**PEA**) for matrix multiply. It is clear that without MLP calibration, the power modelling error increases drastically. The reason is that power is not linearly scaled with PE size, because logic synthesis tools such as DC Compiler greatly optimize the PPA of RTL designs and can significantly reduce hardware especially for arithmetic units such as the *SimpleMultiplier* used in the table. By using MLP calibrator against total power, the non-linearity between the primitive summation and total power is captured efficiently.

Table 5. Ablation Study of MLP Calibration Between Primitive Sum and Ground Truth Power Label Over the Validation Set

PEA	Ground Truth Power Label (mW)	Primitive Sum w/o MLP (mW)	Err. (%)	Primitive Sum w/ MLP (mW)	Err (%)
2 × 2	1.31	1.32	-0.86	1.23	6.36
8 × 8	11.81	14.21	-20.32	11.71	0.87
16 × 1	4.82	5.28	-9.65	4.91	-1.80
16 × 2	8.27	11.62	-40.53	8.04	2.74
16 × 4	12.83	19.03	-48.36	12.52	2.37
16 × 8	20.47	31.65	-54.66	20.07	1.95
16 × 16	38.42	46.5	-21.07	37.20	3.17

Each PEA uses the SimpleMultiplier primitive and has the same configuration as in the DSE and power model experiments. Post-synthesis average power from running a random workload at 1GHz is used as the ground truth label. The PE array tiling is over Tj and Tk for a matrix multiply unit. MLP calibration is necessary to capture the nuanced optimizations done over PE arrays by the EDA synthesis tools.

7.4 Generator Design Space and Detailed PPA Analysis

As shown in Table 6, we can use OSCAR to generate a variety of accelerators (Diannao-like [7], Eyeriss-like [8], and several reconfigurable cores including Spatial-Spatial based on Diannao and several other tilings, such as with another spatial flow with a tiling of input-channel equal to three, Spatial+Wino based on Diannao and a Winograd dataflow, Spatial+Wino+Fused Act. which has an fused pipeline between the convolution and activation kernels.) and compare with a baseline generator Gemmini[10] and industrial accelerator NVDLA [12]. For fair comparison, we limit the number of PEs, total SRAMs, and inference batch to be the same, set as 256, 128 KB, and single-batch, respectively. All designs were synthesized in 40nm TSMC and simulated at 1 GHz. Table 6 shows the comprehensive design space supported by OSCAR. Generally, across all benchmarks, which include ResNet50 [38], HRNet [39], Yolo [40], UNet [41], and so on, spatial and systolic architectures like Dianao, Eyeriss, and Gemmini have the lowest area and power due to their lower hardware complexity. On the other hand, reconfigurable architectures have the lowest latency but at the cost of higher chip area and power due to the increased number of components.

The lowest chip area is the spatial architecture based on Diannao at $1.79mm^2$, and the highest chip areas are reconfigurable cores such as the spatial+spatial and spatial+Wino+fused core at 3.09 and $2.11mm^2$, respectively. Thanks to resource sharing, the spatial+Wino and spatial+Wino+fused cores only add 17–18% overhead in area. The spatial-spatial cores add the most largely because the SRAM sizes are not completely matched, resulting in duplicated SRAMs, incurring large area overhead.

On ResNet50, the fastest latency is the reconfigurable spatial+wino+fused core at 16.3 ms, and the slowest is Eyeriss at 46.6 ms per image. Eyeriss's runtime is much slower compared with other single-dataflow cores because Eyeriss has tiling across the filter axes (i.e., K_x), which results in low utilization for 1x1 bottleneck layers. The lowest power is Eyeriss at 144 mW, and the highest power is the spatial+wino+fused core at 256 mW. The lower power of Eyeriss is also attributed to low PE utilization, but also because systolic architectures have lower fanout between PE units, unlike spatial architectures like DianNao, which have high fanout to PE units. Spatial+Wino+Fused uses less power than single-dataflow architectures because although Winograd convolution and fused hardware incur higher power costs, the power overhead is amortized as a

Table 6. PPA Results of Baselines and Several OSCAR-Generated Multi-Dataflow AI Cores

Architectural Design	CNN-based Benchmark Power, Performance, and Area												
	Chip Area↓ (mm^2)	ResNet50		SqueezeNet		ResNet18		HRNet-Small		YoloV3		UNet	
		Latency↓ (ms)	Pwr↓ (mW)	Latency↓ (ms)	Pwr↓ (mW)	Latency↓ (ms)	Pwr↓ (mW)	Latency↓ (ms)	Pwr↓ (mW)	Latency↓ (s)	Pwr↓ (mW)	Latency↓ (ms)	Pwr↓ (mW)
NVDLA Generated [12]	3.31	17.2	233	2.6	**38.2**	12.6	163	5.59	252	126	87.18	593	42.16
Gemmini Generated [10]	1.87	14.5	101	2.60	155	10.1	201	5.93	163	271	138	1.38	122
Single: Diannao-like [7]	**1.79**	14.9	74.4	2.57	177	9.32	232	6.61	202	272	130	1455	115
Single:Eyeriss-like [8]	1.81	46.6	144	14.6	91.9	4.04	**129**	11.8	**145**	484	130	2264	115
Spatial+Spatial	3.09	13.7	57.6	2.20	155	9.14	322	5.90	191	216	**67.4**	1304	76.4
Spatial+Wino	2.10	12.6	**36.6**	1.80	122	5.59	160	3.70	161	126	91.6	571	**38.0**
Spatial+Wino+Fused Act.	2.11	**11.6**	36.7	**1.58**	185	**5.34**	167	**3.37**	181	**118**	76.6	**540**	40.2

We evaluate each core across different benchmarks that are characteristic of computer vision AI algorithms. For fair comparison, all cores are generated with 256 PEs and a maximum of 128KB of L1 SRAM. Power is measured at 1GHz with Batch = 4 at the TSMC 40nm node.

Table 7. PPA Results of Baselines and Several OSCAR-Generated Multi-Dataflow AI Cores Over Several Transformer-Based Benchmarks

Architectural Design	Transformer-Based Benchmark Power, Performance, and Area								
	Chip Area↓ (mm^2)	Transformer Prefill		Transformer Decode		ViT		DeiT	
		Latency↓ (s)	Pwr↓ (mW)	Latency↓ (ms)	Pwr↓ (mW)	Latency↓ (s)	Pwr↓ (mW)	Latency↓ (s)	Pwr↓ (mW)
NVDLA Generated [12]	3.31	4.15	98.7	3.23	98.68	0.54	142	5.75	98
Gemmini Generated [10]	1.87	16.5	138	51.2	118	2.14	122	5.78	98.9
Single: Diannao-like [7]	**1.79**	4.15	121	3.23	138	0.55	137	5.75	138
Single:Eyeriss-like [8]	1.81	38.4	113	90.1	131	5.03	113	18.0	131
Spatial+Spatial	3.09	4.11	**78.9**	3.20	**78.8**	0.536	**55.6**	5.66	**69.5**
Spatial+Wino	2.10	4.15	138	3.23	138	0.54	144	5.75	138
Spatial+Wino+Fused Act.	2.11	**3.94**	93.8	**3.07**	113	**0.51**	121	**5.46**	131

The setup is similar to Table 6 but with sequence tokens equal to 1024 or token patches.

result of the more memory-bound and time-consuming but lower power-consuming fully connected layers.

For ResNet18 and HRNet benchmarks, the spatial+Wino+fused core has the lowest latency, and the Eyeriss design has the lowest power. It is interesting to observe that for HRNet-small, the spatial+Wino+Fused core exhibits almost 1.5-2× faster runtime compared with the spatial/wino core. This is attributed to the fact that HRNet is compute-bound and not composed of all bottleneck layers, and has many 3×3 convolutions, especially in the fusion stage, which are bound to use Winograd convolution to speed up inference.

Over complex networks such as Yolo and UNet, which have different bottleneck layers, Winograd and reconfigurable multi-dataflow architectures show the best performance as well as lower power, because less cycles are wasted and better efficiency is achieved with the Winograd layers.

Refering to Table 7, the same architectures were run but for transformer-based models, including Transformer in both encoding/prefill and decode modes [30], ViT [42] and DeiT [43] models. The performance of different architectures are much closer compared with vision, because Winograd convolution cannot be used and generally the tilings fit the large matrix multiply operations. Yet, generally the Spatial+Wino+Fused architecture has the lowest latency because (1) fusion reduces memory accesses of intermediate tensor outputs and speeds-up inference, and (2) multi-dataflows allow suitable tilings and loop orders to adapt to the different operation layers. For example, the reconfigurable multi-dataflows optimize transformers because during prefill, the accelerator's tilings are tuned towards square tilings, whereas during decode, the batch dimensions are lower and more data is read from the **key-value** (**KV**) cache, so flatter tilings are optimal. The power of the spatial+spatial architecture generally is the lowest for two reasons: (1) picking energy efficient

Table 8. Average Data Collection and Training Time for Different Hardware Primitives in OSCAR

Primitive	Data Collection	Training Time
Multiplier	1.1 hour	10s
Adder	30 min	3s
Casting/Serializer	26 min	1s
Deserializer	12 min	2s
SRAM	43 min	1s
Mux	37 min	3s

Table 9. Runtime of OSCAR and Power Simulation for an Accelerator at the Core-Level Running Resnet50

Typical Runtime Per Design	EDA Flow	OSCAR Inference
Architectural	0	1.0 sec
Logic Synthesis	30 min	0
RTL Simulation	10 min	0
Average Power Simulation	1 min	19.1 sec
Total	41 min	20.1 sec

dataflows per operation and (2) optimized loop order matching per operation and data-reuse resulting in lowered power of especially using weight re-use for matrix multiply where reading the weights is one of the bottlenecks.

7.5 Power Model Training Set Runtime

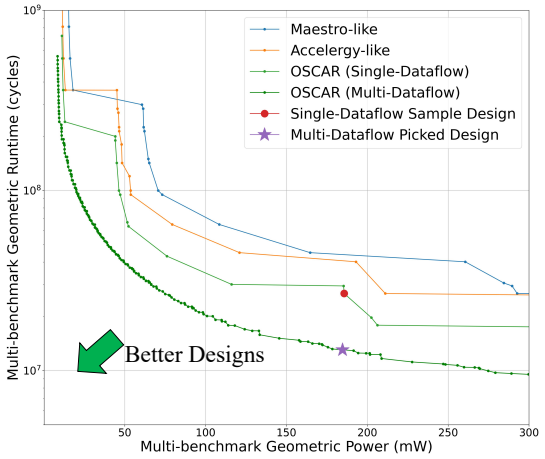
Our generator and power model are efficient in both training and inference, since they are based on a set of basic primitives. Table 8 shows time for the training data collection and training of different primitives. Training data collection uses the parameters from Table 2 and runs synthesis and power simulation to get the ground-truth, with average runtimes from minutes to hours depending on the complexity of the primitive space. Multipliers are more complicated, so data collection and training time are longer. Because primitives are small functional units, power characterization and modeling are very fast and easy. Note the training is only performed once, since it generalizes well for larger designs.

As seen in Table 9, the runtime of estimating the power of an accelerator core using OSCAR is fast. Individual primitive inference runtimes are in the milliseconds; therefore, overall units are longer in runtime but still fast compared with the golden baselines, speeding up power estimation by 120× from around 40 minutes to around 20 seconds.

7.6 Design Space Exploration and Industrial Chip Modelling

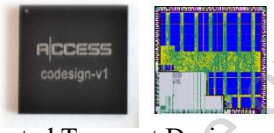
To verify our power model in AI chip design, we apply OSCAR in a design space exploration for a dense convolution chip. Then, we identified optimal designs and validated the generated design through a realistic tape-out process.

We define the search space with the following configurations: (1) at the core level, there can be up to 16 dataflows, (2) at the unit level, the PE is set from 16 to 512 each with 8 different tilings, and (3) at the component level, the multiplier is chosen from 16 different types, the casting network can be systolic or not, and the loading network varies from 1 to 32.



(a) Pareto-Optimal Designs

Setup	Description
Searched Designs	1.5 million+ designs
OSCAR runtime	~8 hours
Tech	1GHz, 40nm, 1.1V
Final Design	Multi-dataflow Reconfigurable Spatial Dataflow with Fusion, 256PEs, 256KB L1
Model	Multiple Benchmarks, 50% weight sparse workload, Batch Size 4



(b) Selected Tape-out Design

Fig. 13. Comparison of DSE results picked by average of models. Ground-truth latency in cycles and power are reported while running several workloads, including ResNet50, VGG16, UNet, SqueezeNet, ResNet18, HRNet, transformer, and YoloV3. We adopt an identified design point on the Pareto curve over the average of all benchmarks and tape out a corresponding AI chip.

Multiple benchmarks were run for DSE. The *multi-benchmark experimental setup* include the following details:

(1) **Multiple benchmarks**, including many common AI workloads, including ResNet50, Yolo, UNet, ResNet18, SqueezeNet, HRNet, Transformers, and so on. These benchmarks are used from the open-source ONNX model zoo on Github.
(2) **Variety of runtime and process corners**, such as the clock frequency ranging from 200MHz to 1GHz, and different PVT technology nodes. We use 40nm as the tape-out is also in 40nm.
(3) **Various weights and inputs**, including synthetic input and real image inputs in order to understand the variations of PPA under different workload setups.

Figure 13(a) compares DSE results between OSCAR and two other baselines. Designs with the lowest power and highest performance (i.e., minimum latency in cycles) are preferable. The X-axis and Y-axis are the geometric summed power and runtime over individual benchmarks. OSCAR our work has two different use cases, one where only single-dataflows are identified, and (2) where multi-dataflow architectures are identified using Algorithm 1 from Section 6. The picked designs are example of designs on the Pareto-front. The actual tape-out is based on a design on the Pareto-front.

The individual benchmark DSEs are shown in Figure 14. The red dot shows the PPA performance of OSCAR's single-dataflow picked design, whereas the purple shows OSCAR's multi-dataflow reconfigurable picked design. Generally, reconfigurable and multi-dataflow designs show better PPA overall multiple benchmarks because single-dataflow architectures are tuned for specific workloads and not optimal over general workloads, such as transformer, HRNets, and so on. due to non-optimality of the loop order or tiling.

Overall, OSCAR identifies better designs (i.e., Pareto-optimal designs in ground-truth) over baseline models, because OSCAR is more accurate and thus better at picking the most promising Pareto-points than prior work. An identified optimal design reduces the latency (that is, cycle

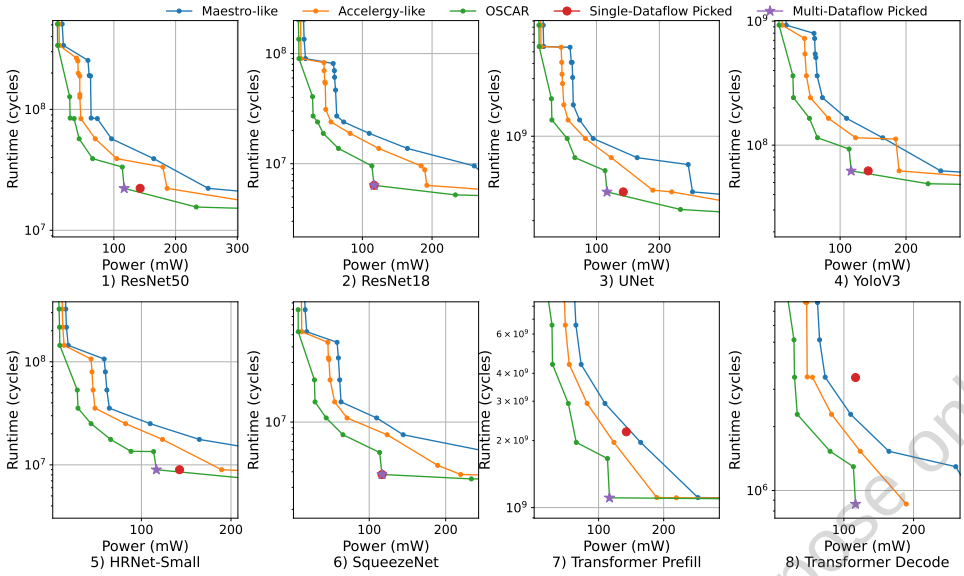


Fig. 14. Comparison of DSE results picked by different models. Ground-truth latency in cycles and power are reported while running several workloads, including ResNet50, VGG16, UNet, SqueezeNet, ResNet18, HRNet, transformer and Yolo. The purple star is a selected multi-dataflow design, whereas the red dot is a single-dataflow design evaluated across multiple benchmarks.

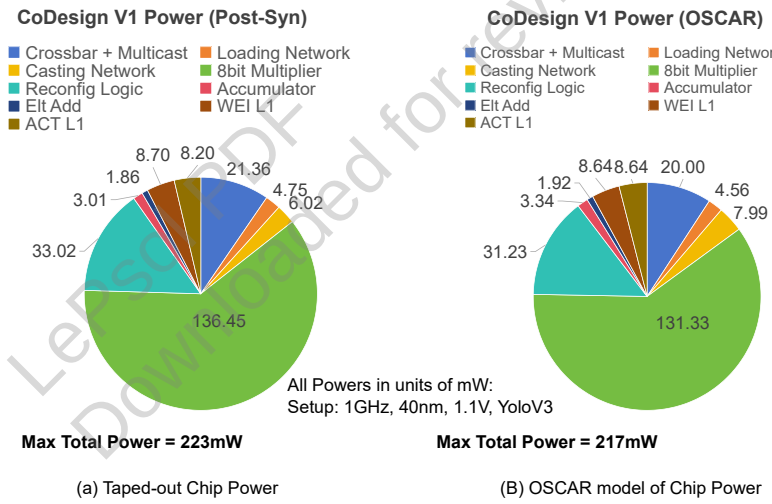


Fig. 15. Comparing a taped-out design's max power against our model. The multi-dataflow accelerator is at the 40nm node, and runs a YOLO workload [40, 44] workload at 1GHz.

counts) by more than 2× and power by 2.5×, compared with designs found by DSE from lower fidelity baseline models.

Based on the DSE results, an identified optimal AI chip design on the Pareto-front was synthesized and taped out. For the tape-out, the DSE was restricted to spatial/systolic reconfigurable units.

Figure 13(b) shows the chip information. The design is a 2-4-8-bit reconfigurable core, has 512 PEs and systolic input loading. The technology is in TSMC 40nm, has 1MB of memory, achieves 250 mW power, has a final chip area of 4mm × 3.75mm including DRAM, PLL and other SOC components. We model the different modules of the taped-out chip using OSCAR. Figure 15 shows the ground-truth power and OSCAR's predictions when running a realistic 8-bit YOLOv3-dense workload at 1 GHz, with OSCAR showing 90% accuracy. Finally, we also measured the real total power in hardware with an oscilloscope, with an average core power of 240 mW, which is close to the simulation and OSCAR's prediction results. We also conducted several experiments with sparse workloads, with power lowered by 30%; OSCAR also predicts a similar lowering in power.

7.7 Future Work

This work focused on the development of an AI accelerator generator and an early-stage model for vision, transformer and convolution neural networks. Future extensions of OSCAR include exploring advanced accelerator techniques, including outlier architectures, group/block sparsity, heterogeneous system integration, and emerging systems, which are important in the latest AI models [45]. We believe that the design principles, early-stage models and methodologies in this work are generalizable and extensible to cover future AI accelerators and workloads.

8 Conclusion

In this article, we propose OSCAR, an open-source AI accelerator generation and estimation framework. The design space covers a wide range of modern AI accelerators and workloads. OSCAR can also generate Chisel code for power and performance validation. Our power modelling achieves around 3.8% error and correlation R>0.99, which is significantly better than other power models. We evaluate our power model against baseline power models for DSE, finding better Pareto-optimal designs. Finally, based on the DSE, a commercial-grade CNN/Matmul accelerator was taped out and its power was modelled using OSCAR with 90% accuracy.

Acknowledgement

The authors acknowledge the use of QWen-V3, a generative AI tool developed by Alibaba, for polishing the article writing and improving the readability of this manuscript. The tool was not used to generate research ideas, technical methods, experimental results, or conclusions. The authors have reviewed and take full responsibility for all content in the manuscript.

References

[1] Shaojun Wei, Xinhan Lin, Fengbin Tu, Yang Wang, Leibo Liu, and Shouyi Yin. 2022. Reconfigurability, why it matters in ai tasks processing: A survey of reconfigurable ai chips. *IEEE Transactions on Circuits and Systems I: Regular Papers*. 70, 3 (2022), 1228–1241.

[2] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. Tileflow: A framework for modeling fusion dataflow via tree-based analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1271–1288.

[3] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *International Symposium on Microarchitecture*.

[4] Jie Wang, Licheng Guo, and Jason S. Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 93–104.

[5] Yakun Sophia Shao, Brandon Reagen, Gu Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE International Symposium on Computer Architecture*.

[6] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014), 269–284.

[8] Yu Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid State Circuits* (2017). 52,1 (2017), 127–138.

[9] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 15–28.

[10] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*.

[11] Jiajun Wu, Mo Song, Jingmin Zhao, Yizhao Gao, Jia Li, and Hayden Kwok-Hay So. 2025. TATAA: Programmable mixed-precision transformer acceleration with a transformable arithmetic architecture. *ACM Trans. Reconfigurable Technol. Syst.* 18, 1, Article 14 (March 2025), 31 pages. DOI: http://doi.org/10.1145/3714416

[12] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim. 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2) (2019). 21–25.

[13] Gang Li, Weixiang Xu, Zhuoran Song, Naifeng Jing, Jian Cheng, and Xiaoyao Liang. 2022. Ristretto: An atomized processing architecture for sparsity-condensed stream flow in cnn. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1434–1450.

[14] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *IEEE/ACM International Symposium on Microarchitecture*.

[15] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *IEEE Trans. Comput.* 70, 8 (2021), 1160–1174.

[16] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, and Mark Horowitz. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*.

[17] Zhe Lin, Zike Yuan, Jieru Zhao, Wei Zhang, Hui Wang, and Yonghong Tian. 2022. PowerGear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. (2022). arXiv:cs.LG/2201.10114

[18] Qijun Zhang, Shiyu Li, Guanglei Zhou, Jingyu Pan, Chen-Chia Chang, Yiran Chen, and Zhiyao Xie. 2023. PANDA: Architecture-level power evaluation by unifying analytical and machine learning solutions. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 01–09.

[19] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[20] Tianqi Tang, Sheng Li, Lifeng Nai, Norm Jouppi, and Yuan Xie. 2021. NeuroMeter: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) . (2021), 841–853.

[21] Sheng Chun Kao, Geonhwa Jeong, and Tushar Krishna. 2020. ConfuciuX: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) . (2020), 841–853.

[22] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling software programmers design efficient fpga accelerators. ACM Transactions of Design Automation and Electronic Systems (Todaes) . 27,4 (2022), 841–853.

[23] Mingjun Li, Pengjia Li, Shuo Yin, Shixin Chen, Beichen Li, Chong Tong, Jianlei Yang, Tinghuan Chen, and Bei Yu. 2024. WinoGen: A highly configurable winograd convolution ip generator for efficient cnn acceleration on FPGA. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[24] Yongan Zhang, Xiaofan Zhang, Pengfei Xu, Yang Zhao, Cong Hao, Deming Chen, and Yingyan Lin. AutoAI2C: An automated hardware generator for dnn acceleration on both fpga and asic. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems . 43, 10 (2024), 3143–3156.

[25] Rangharajan Venkatesan, Priyanka Raina, Yanqing Zhang, Brian Zimmer, and Nathaniel Pinckney. 2019. MAGNet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[26] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2022. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1377–1395.

[27] Steven J.E. Wilton and Norman P. Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (1996), 677–688.
[28] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. (2018). In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). 764–775.
[29] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). (2021). 97–110.
[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762
[31] Yuanchen Wu, Zhiheng Xie, Hongbing Pan, and Yuxuan Wang. 2025. MBS: A high-precision approximation method for softmax and efficient hardware implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2025), 1–10. DOI: http://doi.org/10.1109/TCSI.2025.3559069
[32] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm Sigkdd International Conference on Knowledge Discovery and Data Mining*. 785–794.
[33] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 382–394.
[34] Mircea Vlduiu. 2012. *Computer Arithmetic : Algorithms and Hardware Implementations*. Computer arithmetic. Algorithms and hardware implementations.
[35] L. Benini and G. De Micheli. 2002. Networks on chips: A new SoC paradigm. *Computer* 35, 1 (2002), 70–78. DOI: http://doi.org/10.1109/2.976921
[36] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 1–13.
[37] Xiaoqing Xu, Nishi Shah, Andrew Evans, Saurabh Sinha, Brian Cline, and Greg Yeric. 2018. Standard Cell Library Design and Optimization Methodology for ASAP7 PDK. (2018). arXiv:cs.AR/1807.11396
[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. DOI: http://doi.org/10.1109/CVPR.2016.90
[39] Xianghong Hu, Xuejiao Liu, Yu Liu, Haowei Zhang, Xijie Huang, Xihao Guan, Luhong Liang, Chi Ying Tsui, Xiaoming Xiong, and Kwang-Ting Cheng. 2023. A tiny accelerator for mixed-bit sparse CNN based on efficient fetch method of simo SPad. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 8 (2023), 3079–3083.
[40] Muhammad Hussain. 2024. YOLOv5, YOLOv8 and YOLOv10: The Go-To Detectors for Real-time Vision. (2024). arXiv:cs.CV/2407.02988
[41] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. (2015). arXiv:cs.CV/1505.04597
[42] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. (2021). arXiv:cs.CV/2012.12877
[43] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. (2021). arXiv:cs.CV/2012.12877
[44] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. (2018). arXiv:cs.CV/1804.02767
[45] Lvcheng Chen, Ying Wu, Chenyi Wen, Shizhang Wang, Li Zhang, Bei Yu, Qi Sun, and Cheng Zhuo. 2025. An agile framework for efficient llm accelerator development and model inference. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24)*. Association for Computing Machinery, New York, NY, USA, Article 200, 9 pages. DOI: http://doi.org/10.1145/3676536.3676753

Received 13 February 2026; revised 21 June 2026; accepted 29 July 2026