# FSGen: Agile Fused and Sparse Accelerator Generator with Accurate Power Model for LLM Applications

Jay Zhe-An Mok, Qijun Zhang, Zhiyao Xie*
Hong Kong University of Science and Technology
Hong Kong
{jzmok,qzhangcs}@connect.ust.hk,eezhiyao@ust.hk

## Abstract

With the growing demand of artificial intelligence (AI) applications, large language models (LLMs) have become important workloads in many domains. The question of how to efficiently generate optimal AI chip accelerator designs remains unresolved and challenging. Currently, there is a lack of end-to-end design methodologies for efficient design space exploration (DSE). We propose *FSGen*, an agile framework for attention-based LLM accelerator generation with an early-stage PPA estimator. FSGen supports fused operator dataflows and sparsity with a diverse design space and finds designs with 1.4× better power efficiency or 10× speedup with similar PPA metrics compared to prior work. Pareto-optimal designs have much better performance over a wide range of LLM benchmarks and have 58× better figures of merit (FoM). Design exploration is also faster due to our PPA estimators, which have better accuracy than prior art and reduce DSE runtime drastically.

## 1 Introduction

Recently, artificial intelligence (AI) applications have grown rapidly, large language models (LLMs) have become important workloads in advanced chatbots, speech, visual, audio, and industrial automation domains. LLM inference on edge hardware has become an important application that is becoming more prevalent in embedded hardware in order to address issues of data security, network latency, and inefficiencies for cloud and server solutions [24].

However, optimizing customized AI chip solutions for running LLM workloads remains a challenge. First, it remains challenging to define and characterize a comprehensive design space for hardware solutions. Second, due to the exponential design space and large design size of AI chips, design and optimization efficiency still remains a difficult problem. To alleviate the issue, end-to-end AI chip hardware generators for efficient design have been proposed by prior works. TensorLib/Rubick [9, 14] generates systolic and multicasting spatial arrays for AI chip kernels. They propose a concise hardware design space and an exploration strategy that prunes invalid designs and enables efficient design exploration. BinaryLLM [4] uses low-bit quantization and generates systolic arrays for the LLM workload. They include KV cache support and a limited set of fused operations (i.e. matrix multiply + activation). They also propose an ML-guided design exploration strategy. Stellar [7] is a generator framework for sparse systolic arrays for matrix multiplication kernels, supporting data compression and different fine-grain sparsity dataflows. However, current frameworks have several limitations as summarized below.

**Limited Design Space Configurability**. Important candidate designs are missing from prior hardware generators [4, 7]. 1) *Fused operation dataflows*, which greatly reduce intermediate tensor storage by fusing multiple kernels and are often used for accelerating LLM operations, remain underexplored. Dataflow is very important in the PPA trade-offs of customized AI chips [11, 13]. 2) *Sparsity configurability* at both the algorithm and hardware implementation level is also limited. Sparsity is important for optimizing energy and performance. Prior work [7] only supports one or a limited set of sparsity and sparse networks, and is unable to compare multiple types of sparsity in a unified framework.

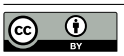
**Insufficient Support to Implementation**. Many design exploration tools for LLM [22, 25] do not support automated RTL generation, preventing users from verifying their designs and quality of results (QoR) on FPGAs or as digital ASICs.

**Limited Early-Stage Estimator Accuracy Due to Simplification.** Existing architecture exploration tools suffer from oversimplified power models. Several works assume that the per-operation power and energy are constant or limited to several states [11, 17, 21]). In reality, the energy is highly sensitive to input activities.

To overcome these challenges, we propose *FSGen*, an open-source agile framework for attention-based LLM accelerator generation with early-stage ML-estimators†. The main contribution of our work is a representation for sparse and fused operator dataflows that is more flexible and expressive of various LLM accelerators and enables efficient exploration of a larger design space compared with prior works. We support multi-level sparsity architectures, full configurability over fused operator tilings and loop orders, which allow more energy-efficient and high-performance designs to be found. Our contributions are summarized below.

- **A Comprehensive Design Space**. FSGen supports full configurability of fused operator dataflows and sparsity for LLM acceleration. Fused operator greatly reduces memory accesses and improves performance. We also propose the *sparse set and tiling*, a unified representation which bridges the gap between high-level design and sparse hardware.
- **Automated Generator from Design Space to Implementation**. A highly configurable end-to-end Chisel-based RTL generator, enabling agile implementation and verification.

*Corresponding author.




*DAC '26, Long Beach, CA, USA*

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804052

†Code released at https://github.com/hkust-zhiyao/FSGen

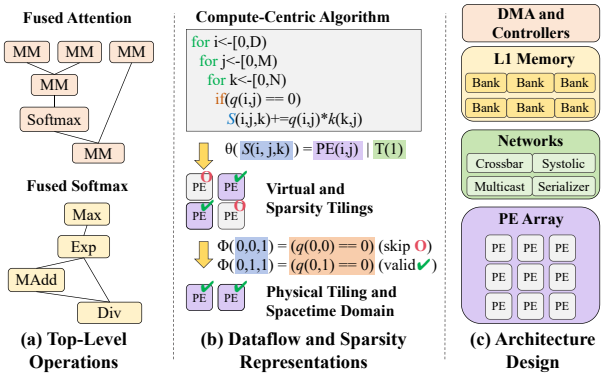



**Figure 1: Our proposed design space and generator covering (a) fused operations, (b) sparsity, and (c) architecture.**

- **Efficient Exploration of AI Accelerators**. FSGen supports fast and accurate early-stage ML-based PPA estimators for power modelling. The model is fast to train, accurate, and generalizes well to full designs. Based on our generator and early-stage models, we enable efficient early-stage design space exploration and rapid implementation and verification of designs, reducing design time and difficulty.

In the remaining sections, we introduce the background (Sec. 2), FSGen's high-level representations (Sec. 3), hardware design space (Sec. 4), hardware generator (Sec. 5), and PPA estimators (Sec. 6).

## 2 Background

*Hardware generators* are an agile design methodology for converting algorithms into hardware. Referring to [13, 14], algorithms are composed of loop instances and a tensor algebra. The *iteration domain* $D_S$ is the set containing all loop instances $S$, which is a tensor algebra indexed by loop iterators $\vec{n}$.

$$D_S = \{S(\vec{n}) | \vec{n} = (i, j, ...)\} \tag{1}$$

A *spatial array* consists of a processing element (PE) array, memory and interconnects. The *space-stamp* is $PE(\vec{p})$ where $\vec{p}$ is a positional index of the PE array. The *time-stamp* $T(\vec{t}), \vec{t} = (t_0, t_1, ..., t_n)$ corresponds to the schedule, where $t_0$ corresponds to the innermost loop and $t_n$ the outermost. Together, the space- and time-stamps form the space-time domain $D_{st}$.

A *spatial dataflow* $\Theta$ is a mapping from a loop instance $S$ to a space- and time-stamp of a spatial architecture.

$$\Theta_{D_S \longrightarrow D_{st}} = \{S(\vec{n}) \longrightarrow (PE(\vec{p}) | T(\vec{t}))\} \tag{2}$$

A *fused operation dataflow*, known as kernel or operation fusion, is a topology on multiple dataflows. Fused operation allows intermediate tensors to be passed directly to the next PE array for processing, thereby reducing costly memory accesses [1, 15]. Fig. 1 (a) shows fused operator dataflows for attention and softmax.

## 3 Accelerator High-Level Representation

FSGen's representation can characterize complex dataflows such as fused operators, i.e., multi-nested imperfect loops, which are more realistic of industrial designs and represent a larger variety of designs, ranging from softmax to fused matrix multiply chains, unlike prior works such as [7, 9, 14] that only consider perfectly nested loops and single dataflows. In addition, unlike prior art that couples sparsity to dataflow [7], FSGen's sparsity design spaces are defined mathematically, general and orthogonal to dataflow, aiding

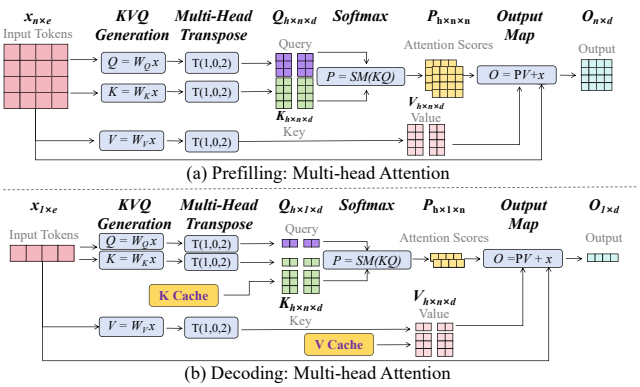



**Figure 2: Typical attention flows. (a) Prefill-Stage Multi-Head Attention (b) Decode-Stage Multi-Head Attention.**

in efficient design exploration. We now discuss the high-level fused operator dataflow (Sec. 3.1) and sparsity representations (Sec. 3.2).

### 3.1 Fused Operator Dataflows for LLM

Fused operator dataflows are equivalent to multiple perfect loop nests sharing a common set of loop variables and iterators. We apply fused operator dataflows to important LLM layers, such as attention, feed-forward networks (FFN), KVQ generation, and normalization layers. Referring to Fig. 2, multi-head attention (MHA) consists of several tensor algebras. The fused operator iteration domain for the prefill stage is represented as several loop instances $S_i$, with iterators $\vec{n} = (b, n, m, h, d, e, o)$, which are the batch, query, key/value sequence, head, hidden size, and input/output embedding, respectively, and SM refers to softmax. The fused operator chains the softmax and output operations to the attention score, targeting the main bottleneck. $S_1, S_2, S_3$ represent score calculation, softmax and the output matrix multiplication, respectively. The iterators and shapes can be inferred from Fig. 2.

$$S_1 : S \mathrel{+}= Q \times K; \quad S_2 : P = \mathrm{SM}(S); \quad S_3 : O = V \times P + x \tag{3}$$

The decoding stage is similar. The key difference is that only 1 token is generated, and prior key-value tokens are loaded from the KV cache. The complete fused operator dataflow is a series of dataflows for each iteration domain $D_{S_i}$ and PE array $PE_i$,

$$\Theta_i = \{S_i(\vec{n}) \longrightarrow (PE_i(\vec{p}) | T_i(\vec{t}))\} \tag{4}$$

Group-query attention (GQA) is very similar to MHA, but with the query partitioned into groups $g$ that share the same KV heads across all groups. Other layers are similarly represented.

### 3.2 Sparse Dataflows

We now discuss our sparsity representation. Our framework can easily represent sparse combinations and group sparsity, which are not easily expressed by sparse data structures from prior works [7, 22]. Our sparse dataflows are also more general and flexibly configured, improving the design space.

**Sparse Set and Mapping**. The *sparse set* $Q$ is a set of *sparse mappings* $\Phi(F(\vec{n}))$, which map from iterators $\vec{n}$ and tensors $\vec{x}, \vec{y}$... to the Boolean domain $\mathbb{B}^{F(\vec{n})} = \{0, 1\}^{F(\vec{n})}$, where $F(\vec{n})$ is the cartesian product of a subset of iterators.

$$Q_{D_S \times (D_x, ...) \longrightarrow \mathbb{B}^{F(\vec{n})}} = \{\Phi[\vec{n}, \vec{x}, ...](F(\vec{n})) \mid \vec{n}, \vec{x} \in D_x, ...\} \tag{5}$$

We highlight several examples of sparse sets $Q$ used in our work. Weight sparse key generation as in [12] is represented as $Q_w =$

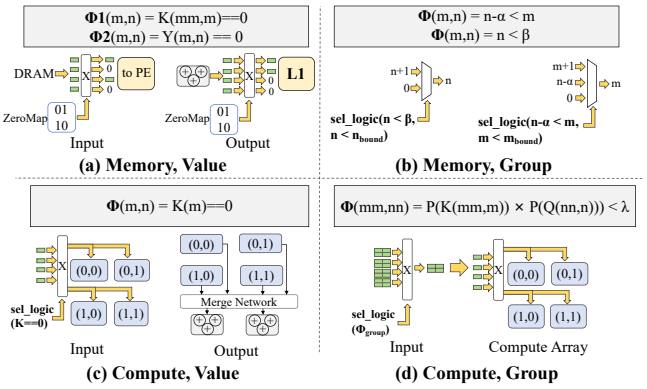



**Figure 3: Different types of sparsity from sparse mappings.**

$\{\Phi_w(\vec{n}) = (W_K(e,d) == 0)\}$, where $W_K$ is the key generation weights. Window attention as in [3] is $Q_{win} = \{\Phi_{win}(\vec{n}) = (n-m \leq W) \& (n-m \geq 0)\}$, where $W$ is the window length, $n$ and $m$ are the attention query and key's sequence length dimensions, respectively. For query-key pruning as in [26], given $mm, nn$ secondary tilings, P is average pooling, omitting the batch, head and hidden size iterators for simplicity,

$$Q_g = \{\Phi_g(nn, mm) = \text{TopK}(\text{P}(q[nn, n]) \times \text{P}(k[mm, m]))\} \quad (6)$$

Note that in the final example, the output domain is a subset of the full domain, namely $F(\vec{n}) = nn \times mm \subset \vec{n} = nn \times mm \times n \times m$. As a result, we observe that a sparse mapping $\Phi$ is *value-based*, if $F(\vec{n}) = \vec{n}$, which means fine-grain values determine skipping. Otherwise, it is *group-based* because a group of tensors are skipped.

**Sparse Dataflow.** *Compute- and memory-based sparse dataflows* are bindings between $Q$ and a dataflow $\Theta$, or the DMA unit, respectively. Fig. 3 shows the main types of sparse dataflows, including the sparse mapping $\Phi$ and its location. Sparse dataflow bridges the gap between the high-level design space and the underlying hardware.

## 4 Hardware Design Space

The hardware design space is the set of mappable designs from the high-level representations. We present several methods that benefit performance and support design spaces that are not well supported by previous generators [4, 7]. We now discuss fused operator (Sec. 4.1) and then sparsity hardware spaces (Sec. 4.2).

### 4.1 Fused Operator Design Space

Fused operator dataflows are composed of multiple dataflows, each is configured by a set of iterator tilings, loop order, and tensor accesses as determined by reuse analysis [9]. Various data-stationary dataflows are set by the loop order ($\vec{t}_{loop} = (t_1, ..., t_n)$ time-stamps). We now discuss other important configurable spaces.

*4.1.1 Hetereogenous Tiling Strategy .* When tilings of different PE arrays are the same, as in [8, 20], downstream arrays are underutilized because they need to wait for valid upstream data. To improve utilization, we propose a heterogeneous tiling strategy. From Fig. 4 (a), downstream tilings can be smaller, as long as the bandwidth is less than the upstream bandwidth. The maximum utilization is when the bandwidths are matched.

*4.1.2 Fused Operator Intermediate Tensor Cache.* Cache buffer sizes for each tensor in the dataflow can be tuned. We propose a multi-tensor caching strategy. As shown in Fig. 4 (b) for an MHA

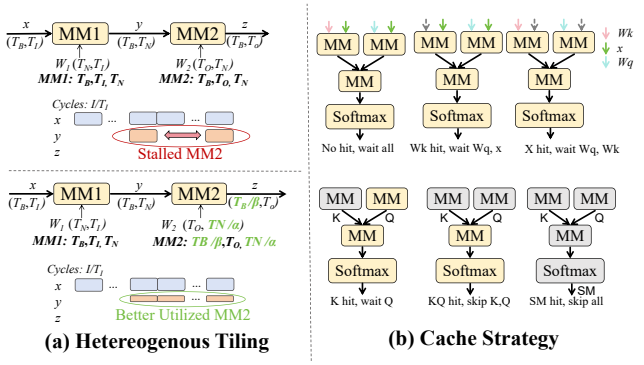



**Figure 4: Fused operator dataflow optimizations.**

dataflow, there are regular cache hit cases, and when both query and keys have cache hits, upstream arrays can be entirely skipped.

*4.1.3 LLM Dataflow Optimizations.* We implement two important optimizations targeting LLMs. First, to support both LLM stages, we modify the fused operator dataflow to be reconfigurable. The prefill mode is the same as the original dataflow. The decode mode allows key and values to be read from the cache, skipping extraneous KVQ generation. Second, we modify the softmax fused operator dataflow to perform online softmax [16, 18], effectively obviating a large softmax input cache and improving utilization.

### 4.2 Sparsity Hardware Design Space

Sparse accelerators contain a PE array and several sparse networks, which determine the data selected into the PE array. In order to support both value and group-based sparsity, the sparse network is multi-level: the group-level network determines the coarse-grained group of tensors to enter the PE, and the value-level network then selects the fine-grained values. The sparse dataflow and sparse tilings define the hardware. A *sparse tiling* encompasses the physical sub-tile $T$, physical tile $TT$, virtual tile $S$, and virtual group tile $SS$. Intuitively, $TT$ represents the PE array size, and $S$ and $SS$ the value- and group-level network, respectively. $T$ allows partitioning of the value-level network to improve timing. The compact notation for the sparse tiling of iterator $i$ is $i = T_i : TT_i : S_i : SS_i$. We now discuss the design space for each type of sparse mapping $\Phi$.

*4.2.1 Compute-Value Sparse Design Space.* Compute-value sparse mappings are mapped into sparse networks. *Sparse networks*, or crossbars, are configured by the parameters copy $c$, group $g$, multicast m, input/output ports $i$ and $o$, and systolic $s$. As illustrated in the top right of Fig. 5, $c$ is the fan-in of each input port, $g$ the

**Algorithm 1:** Compute-based Value Sparsity Mapping

**Input** : High-Level Parameters ($\Phi, TT, T, S, \Theta, \vec{n}$), Tensor $x$.
**Output**: Sparse Network Configuration for $x$ ($c, g, i, o, m$)

$A, E, I$ ▷ Full, external, and inner iterators of a tensor
$B_x = \{\vec{n}\}/A_x$; $B_\Phi = \{\vec{n}\}/A_\Phi$; ▷ Iterator complements
**if** $A_x \in A_\Phi$ **then**
    c = 1; g = $\prod_i^{E_\Phi} \frac{TT_i}{T_i}$; m = $\prod_i^{B_\Phi} TT_i$; i= $\prod_i^{A_\Phi} \frac{S_i}{g}$; n = $\prod_i^{A_\Phi} \frac{TT_i}{g}$;
**else**
    c = $S(B_x)$; g = $\prod_i^{E_x} \frac{TT_i}{T_i}$; m = 1; i= $\frac{1}{g} \prod_i^{E_x} TT_i \prod_j^{I_x} S_j$; n = $\prod_i^{A_x} \frac{TT_i}{g}$;

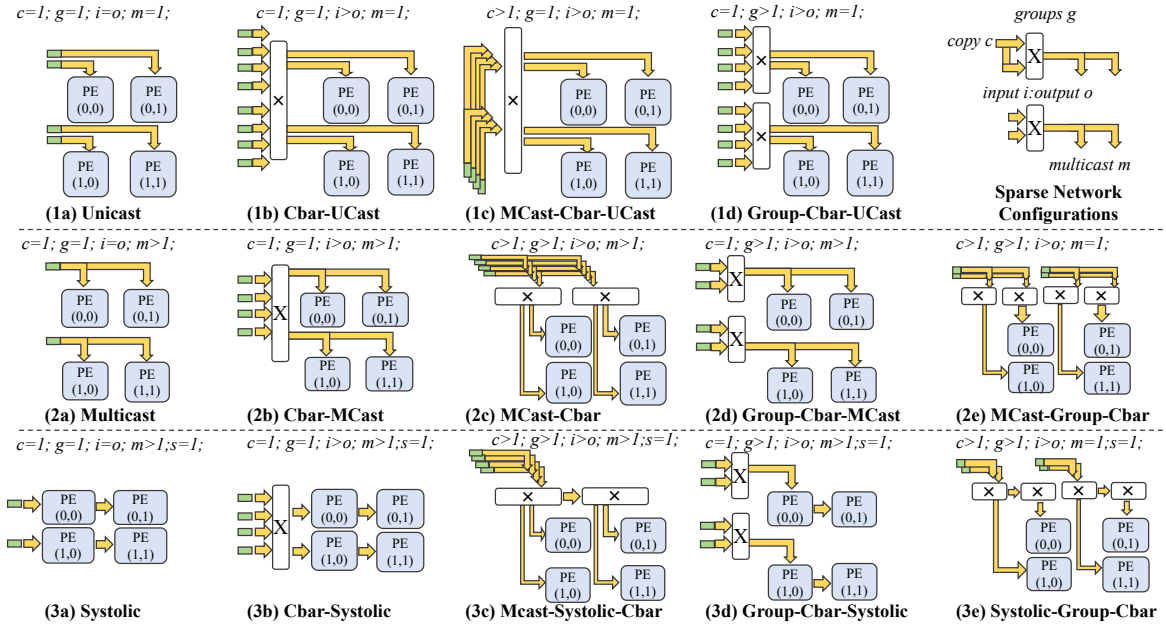



**Figure 5: Design space of networks with a 2×2 PE Array. The space is derived from sparse mappings Φ and sparsity tilings.**

number of crossbars, $m$ the fan-out of each output port, and $s$ the systolic movement as determined by reuse analysis similar to [9]. Fig. 5 tabulates the space of sparse networks with a 2×2 array. There are two cases for each parameter, i.e., $c, g, s, m = 1$ or $> 1$, $i = o$ or $i > o$. Hence, there are $2^5 = 32$ network topologies for each tensor. Removing invalid and inefficient cases (copy and multicast both greater than one), as shown in Fig. 5, there are a total of 14 networks: rows (1) are unicast, (2) multicast, and (3) systolic networks; columns (a) are non-sparse cases, (b) single networks, (c) grouped networks, (d) multicast networks, and (e) mixed combinations.

Algorithm 1 shows how high-level parameters are mapped to sparse network parameters. The key idea behind algorithm 1 is to use $TT$ and $T$ to configure the group, which reduces critical path delays, and $S$ and $TT$ to configure the ratio of the N:M fine-grained sparsity network to improve throughput. Internal iterators, such as the input channel of matrix multiply, do not need to be grouped because they are reduced. $s$ is determined by reuse analysis, as in [9]. For example, for weight sparse matrix multiply with sparse tilings, $\Phi_w = (w == 0)$, tensor $x$ maps to $c = \frac{TT_n}{T_n}, g = \frac{TT_b}{T_b}, m = 1, i = \frac{TT_b}{gS_i}$. Depending on the sparse tilings, the network can become grouped networks (c) or mixed combinations (e) because $m = 1$. Output mergers are also added to collect outputs from different groups.

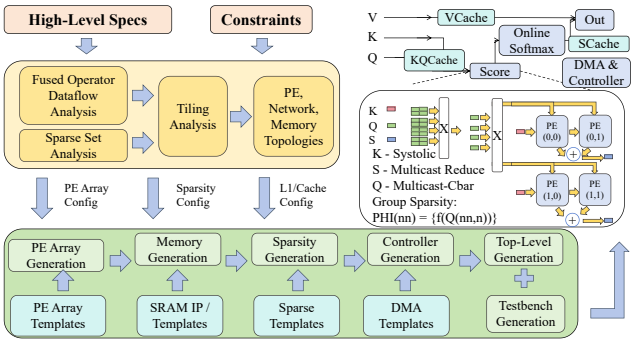



**Figure 6: Our hardware generation flow. High-level design parameters are mapped into RTL hardware.**

*4.2.2 Compute-Group Sparsity Design Space.* Compute-value group sparse mappings are configured by $S$ and $SS$, and use a similar algorithm to Algorithm 1 to map sparse networks. The main change is 1) the network's $i$ and $o$ correspond to $SS/S$ and 1 instead of $S$ and $TT$, and grouping no longer exists because the sparse network is not affected by critical path delay issues, and 2) networks are placed before the value ones (or PE array), as shown in Fig. 3 (d).

*4.2.3 Memory-based Sparsity Design Space.* Memory-based sparse mappings change the state machine (FSM) in the DMA unit. Memory-value sparsity, otherwise called data compression, reduces memory calls by invalidating and skipping unnecessary off-chip requests. Memory-based group sparsity changes the main loop instance counter, which will increment to the specified bounds set by the sparse mapping's condition.

## 5 Hardware Generation

Our work generates attention-based LLM accelerator core RTL in an agile, automated and efficient manner using Chisel [2]. Fig. 6 shows the flow. We highlight the steps used in hardware generation.

**High-Level Design Specification Input.** The input consists of the attention type, dataflow, and sparsity configurations. Fused dataflow parameters include loop order, tilings, and cache sizes. Sparsity parameters include the location, sparsity set and tilings.

**Design analysis and Hardware Output.** PEs are determined from the dataflow, similar to [9], but with queues added in between arrays and pipelines to support fused operation. Memory arrays are based on constraints and sized to store a virtual tile per cycle. Compute-based sparsity networks are based on Algorithm 1. Memory-based sparsity modifies the DMA's FSM controller. The units are then connected to form the top-level module, which exposes an AXI programming interface.

**Software-compiler and Testbench Support**. The compiler reads in LLM models, loads weights, and converts each layer into

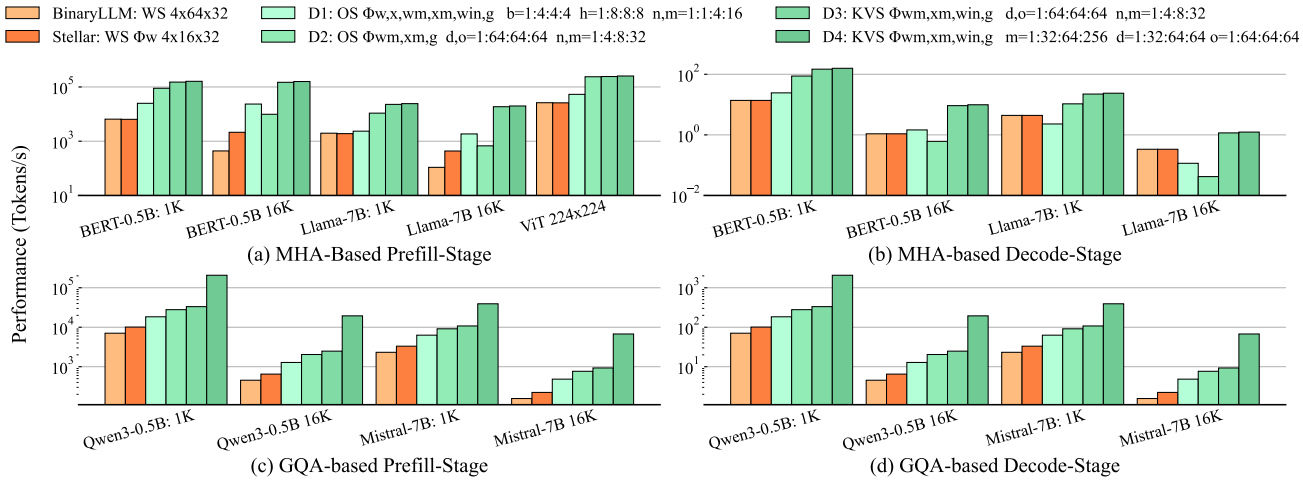



**Figure 7: Performance comparison of Pareto-optimal designs for a variety of LLM benchmarks and token lengths.**

executable kernels. The testbenches support kernel-level simulation, e.g., attention, FFN; full-layer simulations execute multiple kernels. To support LLM operations not limited to attention, the architecture is *reconfigurable*. For FFN layers, the softmax unit is swapped with an activation array; for matrix multiply, the score and output matrix multiply units are run in parallel, supporting a larger tiling.

## 6 Hierarchical Power Modelling

We now discuss our methodology for fast PPA modelling for our designs. Having an accurate early-stage model alongside a generator enables designers to quickly explore the exponential design space of designs rapidly and receive early-stage feedback.

Our early-stage power model is based on primitives. Primitives are the basic modules of accelerators, which include processing elements (i.e., adders, multipliers, exponent units), networks (i.e., crossbars, systolic pipelines, multicasts) and memory (i.e. SRAM banks, registers). The total power or area is therefore the sum of all primitives. The power model $f$ of each primitive is, with $P$ being the output power:

$$P = f(t, h, \alpha, \beta) \tag{7}$$

Inputs include the primitive type $t$, parameters $h$, signal toggle $\alpha$ and zero-bit count $\beta$. $\alpha$ and $\beta$ are estimated from an early-stage cycle-accurate architectural simulator by sampling the executed dataflow and counting the signal bit-level toggles and zero-bit counts per cycle. We explored a wide range of ML models for $f$, including MLP, SVM, and XGBoost [5], etc. We ultimately adopt XGBoost, a gradient tree model for the power model of all primitives, which yields the best accuracy and training time during validation. The benefits of our method include: 1) efficient to generate training data, i.e., 30 minutes for training set creation and minutes to train, and 2) primitive models can combine and generalize well to full designs.

| Design | tps ↑ | $mm^2$↓ | mW ↓ | MHz ↑ | Util. ↑ | FoM ↑ |
|---|---|---|---|---|---|---|
| [4] | 1.9 | 19.3 | 2120 | **1000** | 58% | 0.05 (0.3×) |
| [7] | 1.8 | 10.4 | 1130 | 565 | 50% | 0.15 (1×) |
| D1 | 2.3 | **3.58** | **374** | 812 | **93%** | 1.7 (11×) |
| D2 | 10 | 3.60 | 377 | 712 | 71% | 7.3 (48×) |
| D3 | 22 | 4.79 | 509 | 823 | 39% | **9.0** (58×) |
| D4 | **23** | 7.08 | 763 | 811 | 39% | 4.2 (28×) |

**Table 1: PPA comparison of different designs. The FoM is tokens/s (tps) / power (W) / area ($mm^2$). The tps is an average of the MHA decode-stage benchmarks from Fig. 7.**

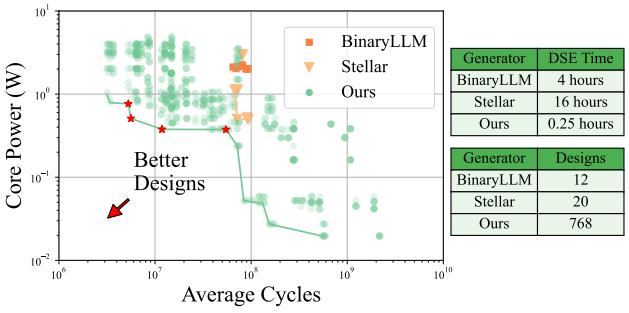
## 7 Experimental Results

### 7.1 Experimental Setup

We evaluate our generator on several common LLM kernels and full networks. For benchmarking, we evaluate Bert-Large [6] and LLama [19], which uses MHA, and Mistral [10] and Qwen3-small [23], which uses GQA. Each model is evaluated in both the prefill and decode stages for different token lengths.

We use Chisel [2] to generate Verilog RTL for LLM accelerators. For ASIC generation and implementation, we use Synopsys Design Compiler for synthesis and area estimation under the TSMC 40nm technology node, Primetime and PTPX for power and timing analysis, and VCS for gate simulation. For power modelling analysis, the results are shown in Fig. 9, and we use the same setup and use Scipy for machine learning algorithms.

As shown in Fig. 7, 8 and Table 1, we conduct a DSE to benchmark and compare the performance and PPA of our FSGen designs against prior hardware generators. We perform a grid-search over the combinations of 3 loop orders, 8 tilings, and 2 compute-value, 2 compute-group, and 4 memory-based sparsity mappings as described in Sec. 3.2, yielding 768 total designs. The baselines BinaryLLM [4] and Stellar [7] are configured based on dataflow, which yield 12 and 20 distinct dataflows, respectively. The DSE's benchmarks are over single MHA layers. Each hardware is constrained to INT8 inference, connected to an 8 GB/s off-chip memory, assigned a maximum of 16KB per tensor and has a maximum of 8 TOPs and a clock speed of 1GHz.




| Generator | DSE Time |
|---|---|
| BinaryLLM | 4 hours |
| Stellar | 16 hours |
| Ours | 0.25 hours |

| Generator | Designs |
|---|---|
| BinaryLLM | 12 |
| Stellar | 20 |
| Ours | 768 |

**Figure 8: Design space exploration comparison of hardware generators. Starred designs are our Pareto-optimal designs, which are further studied in Fig. 7 and Table 1.**

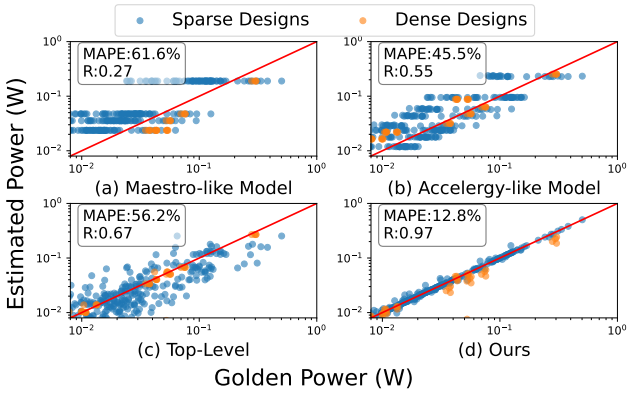



**Figure 9: Comparison of early-stage power modelling against baselines Maestro [11], Accelergy [21], and a top-level ML regressor. Our models are at the primitive-level.**

## 7.2 Generated Design Performance Results

Fig. 7 compares the performance of optimized designs. FSGen's designs are named by their dataflow and sparse mappings. For example, design $D1$ is output-stationary (OS) with balanced tiling over $b, n, h, m$ and supports data sparsity $\Phi_x, w$, memory sparsity $\Phi_{mx,mw}$, blocked attention sparsity $\Phi_g$, and window attention $\Phi_{win}$.

Our designs are faster than designs from prior generators [4, 7], because we support larger design spaces and fused operator dataflows, which greatly reduce memory accesses. For the pre-fill stage, our designs achieve up to 120k token/s (tps). Generally, larger LLMs, longer sequence lengths reduce performance due to quadratically increasing computation. GQA-based networks have even higher performance due to the reduced KV cache.

For the decode stage on MHA benchmarks, our designs achieve up to 110 tps. GQA greatly reduce the kV cache by sharing heads, thereby greatly boosting the decode performance. Some designs are worse in decode because of lowered utilization especially over the input token length iterator $\vec{n}$.

Table 1 shows each design's detailed PPA metrics. Our designs have better overall PPA as highlighted by the figure of merit (FoM), which is 58× higher than prior art. Our design's performance, area, power, and utilization are more optimal. The maximum clock is slightly higher for prior works because of hardware simplicity. Design D1 achieves similar performance as prior works but consumes less power and has better utilization due to the better sparsity support. Design D2 has more balanced tilings, at the cost of slightly larger networks, resulting in lower clock frequency but improved performance. Designs D3 and D4 use larger PE arrays, resulting in improved performance but at the cost of energy and area. We observe that the sparse mappings $\Phi_g, \Phi_{win}$ greatly contribute to improving our model's FoM compared with prior works, which have limited support for different types of sparsity.

## 7.3 Design Space Exploration Results

As shown in Fig. 8, our generator yields a more optimal design space and better Pareto-optimal curve, due to our larger space of sparsity and dataflow support. At a 700mW constraint, our designs have 10× better performance, and at the same cycle count of $8{\times}10^7$ cycles, our designs have 1.4× better power. Because BinaryLLM designs are not sparse, the designs have higher latency. Stellar designs support only compute and memory value sparsity and is not comprehensive. The DSE runtime and space are also better than prior methods due to our early-stage model.

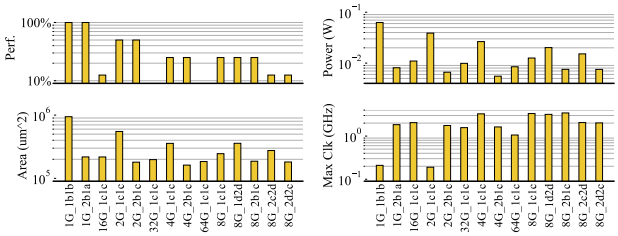



**Figure 10: Comparison of sparse networks for attention. The design space corresponds to the structures in Fig. 5.**

## 7.4 Early-Stage Power Modelling Results

Fig. 9 shows our early-stage power models and comparison against Maestro [11], Accelergy [21], and a top-level version of our power model. Our model is based on primitives, which are trained by varying the primitive type and port data and collecting golden post-synthesis power labels. Each primitive has around 256 training data points, requires around 30 minutes for data collection, and is trained in seconds. For validation, we use FSGen to generate around 20 dense and 200 sparse designs by varying the dataflows, sparsity set and tilings. Our power models achieve 12.8% error and $R = 0.97$, which are better calibrated compared to prior art. Prior art can not capture power accurately because they lack 1) detailed toggling features, and 2) accurate models for high-energy and data-sensitive components, such as sparse networks.

## 7.5 Sparsity Set Design Space Tradeoffs

To better understand the sparse networks supported by FSGen, we generate a variety of 8×8 arrays for attention with $Q = \{(x == 0), (w == 0)\}$. Referring to Fig. 10, a design with name $2G\_2b1a$ means it has group $g = 2$ and $x, w$ networks that are $2b$ and $1a$, respectively. The PPA varies greatly. Unicast networks (1b) have the highest performance but higher delays and power due to large crossbars. Multicast networks (2b) save power and area by broadcasting, but have reduced performance due to under-utilization, especially for low-batch inferences. Grouped networks (1c) improve max clock but trade off performance due to network imbalance. A mixture of networks, such as $2b1a$, yields a good overall PPA.

## 8 Conclusion

In this paper, we propose *FSGen*, an AI chip generator and early-stage estimator for LLM applications. Our framework explores the vast space of sparse and fused-operator dataflows for LLM applications. Using our early-stage PPA ML-based models, we find optimal designs better than prior generators. We identified several designs and synthesized them into ASIC designs. We find that our reconfigurable design, coupled with sparse networks, balanced tilings and sparse sets focusing on compute-group and memory-based sparsity, greatly helps in all stages of LLM acceleration.

## Acknowledgments

This work is supported by Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, National Natural Science Foundation of China (NSFC) 62304192. 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.

## References


[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150

[4] Lvcheng Chen, Ying Wu, et al. 2024. An Agile Framework for Efficient LLM Accelerator Development and Model Inference. In *ICCAD*.

[5] 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.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2019).

[7] Hasan Nazim Genc, Hansung Kim, et al. 2024. Stellar: An Automated Design Framework for Dense and Sparse Spatial Accelerators. In *MICRO*. 409–422. https://doi.org/10.1109/MICRO61859.2024.00038

[8] Tae Jun Ham, Sung Jun Jung, et al. 2020. $A^3$: Accelerating Attention Mechanisms in Neural Networks with Approximation. arXiv:2002.10941 [cs.DC]

[9] Liancheng Jia, Zizhang Luo, et al. 2021. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *DAC*. 865–870.

[10] Armand Joulin et al. 2023. Mistral: Open-weight Mixture of Experts for Efficient Language Modeling. https://mistral.ai. Accessed: 2025-11-16.

[11] Hyoukjun Kwon and et Al. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *MICRO*.

[12] Bingbing Li, Santosh Pandey, et al. 2020. FTRANS: energy-efficient acceleration of transformers using FPGA. In *ISLPED*. ACM.

[13] Liqiang Lu, Naiqing Guan, et al. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *ISCA*.

[14] Liqiang Lu, Zizhang Luo, et al. 2024. Rubick: A Unified Infrastructure for Analyzing, Exploring, and Implementing Spatial Architectures via Dataflow Decomposition. *TCAD* (2024).

[15] Le Qin, Junwei Cui, Weilin Cai, and Jiayi Huang. 2025. Chimera: Communication Fusion for Hybrid Parallelism in Large Language Models. In *ISCA*. Association for Computing Machinery.

[16] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 68658–68685. https://doi.org/10.52202/079017-2193

[17] Yakun Sophia Shao, Brandon Reagen, and et Al. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA*.

[18] Jacob R. Stevens, Rangharajan Venkatesan, et al. 2021. Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 469–474. https://doi.org/10.1109/DAC18074.2021.9586134

[19] Hugo Touvron, Thibaut Lavril, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[20] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *HPCA*.

[21] 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)*.

[22] 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.

[23] An Yang, Anfeng Li, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388

[24] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. 2024. Cambricon-LLM: A Chiplet-Based Hybrid Architecture for On-Device Inference of 70B LLM. arXiv:2409.15654 [cs.AR] https://arxiv.org/abs/2409.15654

[25] Hengrui Zhang, August Ning, Rohan Baskar Prabhakar, and David Wentzlaff. 2025. LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*. 1080–1096.

[26] Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. 2025. Spargeattn: Accurate sparse attention accelerating any model inference. In *International Conference on Machine Learning (ICML)*.