

GRANNITE: Graph Neural Network Inference for Transferable Power Estimation

Yanqing Zhang
NVIDIA
Santa Clara, CA, USA
yanqingz@nvidia.com

Haoxing Ren
NVIDIA
Austin, TX, USA
haoxingr@nvidia.com

Bruce Khailany
NVIDIA
Austin, TX, USA
bkhailany@nvidia.com

Abstract—This paper introduces GRANNITE, a GPU-accelerated novel graph neural network (GNN) model for fast, accurate, and transferable vector-based average power estimation. During training, GRANNITE learns how to propagate average toggle rates through combinational logic: a netlist is represented as a graph, register states and unit inputs from RTL simulation are used as features, and combinational gate toggle rates are used as labels. A trained GNN model can then infer average toggle rates on a new workload of interest or new netlists from RTL simulation results in a few seconds. Compared to traditional power analysis using gate-level simulations, GRANNITE achieves >18.7X speedup with an error of only <5.5% across a diverse set of benchmark circuits. Compared to a GPU-accelerated conventional probabilistic switching activity estimation approach, GRANNITE achieves much better accuracy (on average 25.9% lower error) at similar runtimes.

Index Terms—power estimation, machine learning, graph neural network

I. INTRODUCTION

Today’s computing systems are power-constrained. From datacenters to mobile devices, limits on thermal or electrical power impact achievable performance. As a result, accurate power estimation is a critical part of all aspects of digital VLSI development flows today. Architectural power analysis requires estimating average power over hundreds to millions of cycles. Power integrity signoff requires accurate power analysis on physically-annotated gate-level netlists. Dynamic power optimizations during logic synthesis, clock gate insertion, or place-and-route steps are necessary to meet power targets.

In most power analysis use cases, annotating toggle rates with vectors captured from real workloads is highly preferred to vectorless methods because of their accuracy. However, accurate vector-based power analysis also requires running gate-level simulations (Fig. 1a). These simulations are very slow, typically 10-1000 cycles/s, depending on activity factor and size of design, leading to long turnaround times (hours to days). Slow simulations negatively impact design productivity for architectural power analysis and are impractical for use in dynamic power optimizations, such as automatic clock or data gating or gate resizing during place-and-route. As a result, in such cases, it is more typical to use a switching activity estimator (SAE), shown in Fig. 1b. Average toggle rates for unit inputs and registers are gathered over a window of interest from RTL simulations. Internal toggle rates within synthesized combinational logic are estimated using probabilistic approaches. Although this method is fast, it is inaccurate, due to issues such as signal correlation or reconvergence.

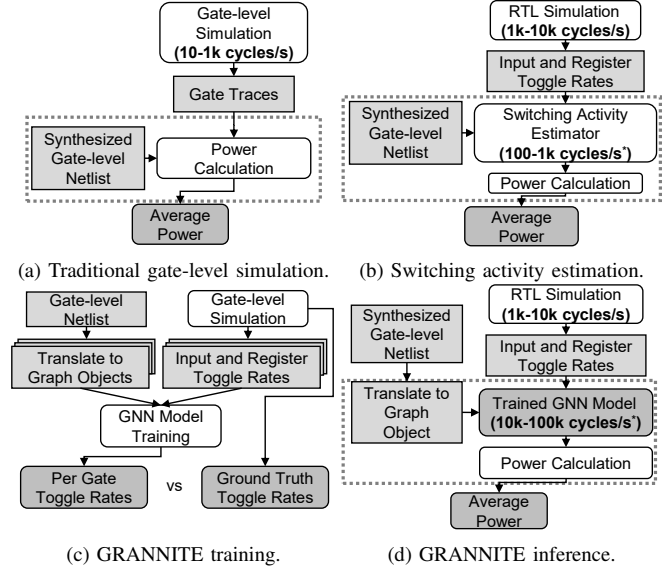


Fig. 1: Average power estimation flows. For (b)(d), throughput is based on a power window of 1000 cycles.

To enable fast and accurate average power estimation, we propose GRANNITE¹, a supervised learning-based SAE for average power inference that foregoes the need for gate-level simulation. During training (Fig. 1c), GRANNITE takes gate-level netlists and corresponding input port and register toggle rates over a power window from simulation as input features. Ground-truth toggle rates per logic gate from gate-level simulation are taken as labels to train against. The trained model can then be used as a learned SAE (Fig. 1d), inferring logic gate toggle rates from new input toggle rate features over a new window of interest for the same designs, or new designs. The inferred toggle rates can then be easily translated into industry-standard formats such as the Switching Activity Interchange Format (SAIF) for average power analysis by commercial tools over the window of interest.

Compared to previous machine learning (ML) based power estimation approaches [1] [2], GRANNITE is *transferable* since it is able to infer power on new gate-level netlists without requiring retraining. This is accomplished by using a novel graph neural network (GNN) model architecture [3] [4] for fast, accurate, and transferable SAE. By achieving an equivalent throughput of >10k cycles/second with a window size of 1000 cycles and skipping gate-level simulation, GRANNITE

¹GRANNITE stands for GRaph Neural Network Inference for Transferable power Estimation.

greatly improves productivity and turnaround time for average power analysis use cases. Our GPU-accelerated implementation of GRANNITE achieves $>18.7\times$ speedup and $<5.5\%$ error on average compared to a traditional gate-level simulation approach. As a comparison to previously proposed SAE methods [5] [6], we also introduce a novel GPU-accelerated implementation of a baseline probabilistic SAE using a similar graph message passing framework to GRANNITE. It achieves $40\times$ - $1125\times$ speedup with similar accuracy (31% average error) compared to a commercial power analysis tool.

The remainder of this paper is organized as follows: Section II introduces background information on the problem of switching activity estimation and related work. Section III presents an overview of GNNs, GRANNITE implementation details, and the baseline probabilistic SAE implementation details. Section IV shows our benchmark and experimental results, and discusses the achieved accuracy and speedups. Section V concludes the paper.

II. BACKGROUND

Fig. 2 shows the problem formulation of switching activity estimation and its application to power analysis. Exact solutions require gate-level simulations to propagate traces from register outputs and unit inputs (typically captured from RTL simulation or FPGA emulation) through a combinational logic netlist. It is common in many cases that only average power is needed. In such cases, it is inefficient to use traditional event-driven simulators when only the final average toggle rates (α) on gate outputs are needed for the dynamic power calculation. Prior research has looked at ways to speed up the calculation of these average gate toggle rates. We highlight four main approaches: accelerated gate-level simulation, statistical sampling approaches, probabilistic switching activity analysis, and machine learning (ML) based approaches.

In recent years, researchers have proposed novel simulation methods using parallel architectures such as GPUs to accelerate gate-level simulation. These methods have focused on either using hybrid event-driven and oblivious methods or parallelizing simulation across cycles and gates to achieve more speedup [7] [8] [9]. However, Amdahl's law effects or GPU device memory can limit speedups, and typically some amount of manual tuning is needed. [8] makes improvements to the memory issue using a novel dynamic memory allocation scheme, and they achieve a throughput of ~ 300 million gate \times cycles per second. This would correspond to a range of 300-3k cycles/s for 100k-1mil gate-sized designs. Considering that windows for average power can range up to millions of cycles, this is still too slow for many use cases.

As an alternative to simulating millions of cycles on a gate-level netlist, statistical sampling approaches take a subset of the cycles captured from FPGA emulation or RTL simulation and replay only that subset during gate-level simulation [10]. This approach is desirable as it can distill long simulations (millions of cycles) down to tractable windows of activity for analysis. However, it has drawbacks in that it does not escape

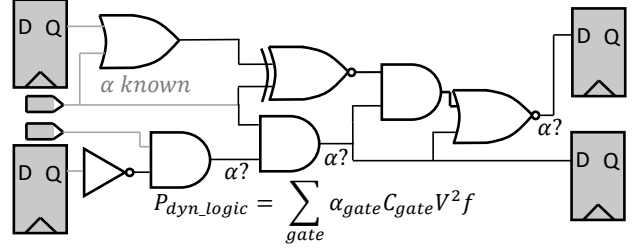


Fig. 2: Depiction of Switching Activity Estimation (SAE) problem.

slow gate-level simulation, and it can miss power analysis corner cases.

Another approach, shown in Fig. 1b, is to move away from simulation altogether, and instead propagate average toggle rates from inputs to outputs based on static probabilities computed from the boolean logic expression of each gate [5] [6]. This approach is often used in commercial power analysis tools. This approach is fast, but can be highly inaccurate, since it does not consider reconvergence correlations. Some work has improved the accuracy of the propagation approach by adding analytical terms to the formulation [5]. In [5], a custom algorithm tags gates in the design that have reconvergent inputs, and simulates those gates by recording their boolean logic expression with regards to primary inputs instead of propagated toggle rates. One drawback with this approach is the massive memory requirement needed for recording the analytical logic expressions, which scales with logic depth and number of primary inputs.

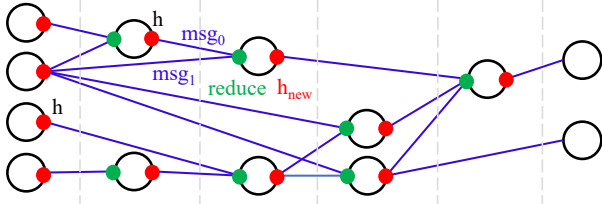
More recently, supervised ML techniques have been proposed for modeling average power of designs, sacrificing a small amount of accuracy for large speedups [1] [2]. Similar to GRANNITE, PRIMAL trains a deep learning (DL) model to infer combinational logic switching activity from RTL simulation traces containing register outputs and unit inputs [2]. PRIMAL demonstrated scalability to large ($\sim 100k$ gate) designs with high accuracy. However, previous ML approaches lack transferability. Although the trained ML models can infer average power for a new workload on the same design, a new ML model must be trained for each new design encountered. This is problematic for fast power analysis of changing netlists or new designs since training can take hours or days.

III. GRANNITE IMPLEMENTATION

GRANNITE is a novel fast, accurate, and transferable ML approach to SAE based on GNNs. We train GRANNITE models on one set of designs (Fig. 1c), and then can run SAE inference (Fig. 1d) in a few seconds on new designs. In this section, we describe the GRANNITE architecture and implementation, written in PyTorch [11] with the Deep Graph Library (DGL) package [12].

A. Overview of GNNs

GNNs are a powerful neural network architecture for machine learning on graphs [3], with many applications in social networking [4] or scene labeling [13]. GNNs operate by assigning node and edge features on a graph, then sharing the features with neighbor nodes through message passing.



Step 1: Message sending Step 2: Message Reduction Step 3: Node Transformation
 $msg = f(\text{edge_features}, h)$ $reduce = f(msg_0, msg_1, msg_2, \dots)$ $h_{new} = f(reduce, h, \text{node_features})$

Fig. 3: Depiction of graph neural network layer mechanisms in GRANNITE. Same level nodes are processed in parallel, while different level nodes are processed in sequence. Step 1 sends a message across each edge to the next node, applying a function of local edge features and predecessor node's propagating features. Step 2 applies a function to conglomerate all incoming messages into one reduced message. Step 3 applies a function of the reduced message, local node features, and previous node features to attain the new propagating node features.

TABLE I: SUMMARY OF LOCAL NODE/EDGE FEATURES

| Type | Description, (Count) | NAND2/A Pin Example Value |
|------|---|---------------------------|
| Node | Intrinsic state probabilities (2) | prob_0=0.25 |
| Node | Intrinsic transition probability (1) | prob_sw=0.1875 |
| Node | Boolean tag if gate is inverting logic (1) | inv=1 |
| Edge | Pin state to output state correlation (1) | state_cor=0.5 |
| Edge | Pin transition to output pin transition correlations (16) | trans_cor_0_to_1=1.0 |

Our model is a variation of a popular type of GNN, the graph convolutional network (GCN). GCNs perform message passing through three steps of message sending, message reduction, and node transformation [4]. While GCNs perform neighbor message passing on all nodes in parallel, our variation does so in a levelized, sequential manner. The final resulting node features then become the output of the graph network layer. Fig. 3 gives a depiction of the GRANNITE GNN layer message-passing mechanisms. GRANNITE learns parameters from input feature data as well as structure of the input graph. Since a logic netlist is represented as a graph, during training, we expect the message passing steps to learn to propagate toggle rates through the netlist from one logic level to the next. In this way, GNNs can learn an approximate solution to SAE based on the netlist features, graph structure, and labeled training data.

B. Toggle Rate Features

Toggle rates of input ports and register outputs are used as inputs to the GRANNITE model, gathered from gate-level simulation during training or RTL simulation during inference. These toggle rate input features are the 'source' of the switching activity of the yet unknown toggles in the combinational logic and are readily available prior to gate-level simulation. We encode the features into an array format with four dimensions representing *[chance to stay low, stay high, switch to low, or switch to high]* over the training power window. During training, the same encoding is used for per-logic-gate toggle rate labels.

C. Graph Object Creation

The other input to GRANNITE is the graph representation of the gate-level netlist, translated via a custom Python script into a DGL graph object, shown in Fig. 4. Gates are mapped

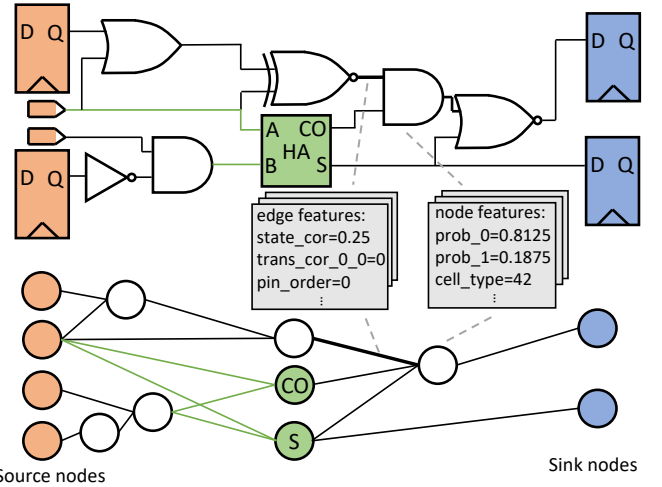


Fig. 4: Translating gate-level netlists to graph objects. Multiple output gates are split into multiple nodes. The process also records node and edge features.

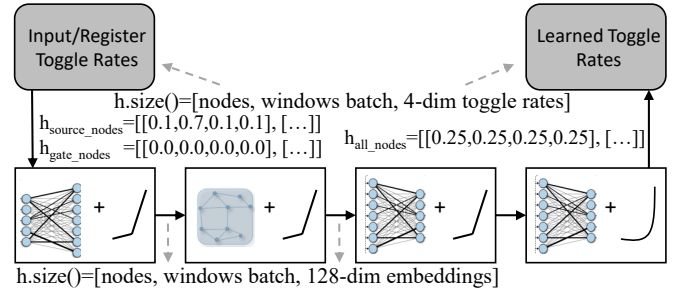


Fig. 5: Diagram of GRANNITE architecture. Arrays have 3 dimensions as we enable multiple power windows to be batched during training/inference.

to graph nodes and output-pin-to-net-to-input-pin connections into graph edges. The translation process automatically splits multiple output gates, such as full-adders, into two separate nodes. The translation preserves both graph connectivity information and local node and edge features that contain characteristics of each gate and net, shown in Table I.

D. GRANNITE Architecture

Fig. 5 describes the overall architecture of GRANNITE. The GNN model consists of 1 fully-connected (FC) layer followed by 1 GNN layer and concludes with 2 FC layers. The first FC layer maps the low (4) dimension input toggle rate features to a higher dimension space (we chose 128 dimensions). We expect the dimensions to represent different learned switching activity embeddings. In essence, the function of the GNN model is to learn the complex, non-linear relationship between input toggle rates, logic, netlist structure, and output toggle rates. Table II defines our GNN message-passing mechanisms. First, message sending concatenates the predecessor nodes' embeddings with local edge transition features before matrix multiplying with local edge state features. Second, message reduction sums all incoming messages. Last, the node transform function concatenates the reduced message with local node features before passing through an FC layer inside the GNN layer. Thus, the calculated embeddings on each node contain both information from predecessor nodes and local node features. Messages are passed from first gate/node stage to last in a levelized manner using the *prop_nodes* function

TABLE II: GNN LAYER MESSAGE PASSING DEFINITIONS

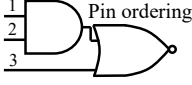

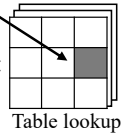
| Implementation | GRANNITE | Baseline |
|--------------------------|--|--|
| Message Sending | $[\text{edge_state}] \times \begin{bmatrix} \text{embeddings}_0, \\ \text{embeddings}_1, \\ \dots \\ \text{embeddings}_{127}, \\ \text{edge_trans}_0, \\ \dots \\ \text{edge_trans}_{15} \end{bmatrix}$ | $\text{msg} = h$ |
| Message Reduction | $\sum_0^n \text{msg}$ |  |
| Node Transform |  $\begin{bmatrix} \text{reduce}_0, \\ \text{reduce}_1, \\ \dots \\ \text{reduce}_{143}, \\ \text{node}_0, \\ \dots \\ \text{node}_{21} \end{bmatrix}$ | $h \times$  |

TABLE III: BASELINE VS. COMMERCIAL TOOL COMPARISON

| Design | Commercial Estimator Error (%) | Baseline Error (%) | Commercial Estimator Throughput (kHz) | Baseline Estimator Throughput (kHz) | Speedup (X) |
|------------|--------------------------------|--------------------|---------------------------------------|-------------------------------------|-------------|
| qadd_pipe | 36.2 | 0.5 | 2.0 | 355.2 | 177.6 |
| qmult_pipe | 52.1 | 0.6 | 1.0 | 40.6 | 40.6 |
| fadd | 48.5 | 51.5 | 2.0 | 2251.0 | 1125.5 |
| fmult | 137.4 | 157.1 | 1.2 | 646.1 | 538.4 |
| NoCRouter | 19.2 | 5.0 | 0.3 | 146.4 | 585.6 |
| RISC-V | 2.8 | 2.8 | 0.2 | 22.4 | 112.0 |

Throughput is calculated with 1000 cycle power windows.

from DGL. The result of the GNN layer sees embeddings on every logic node/gate in the graph, and the last two FC layers map the high dimension embeddings back into low (4) dimension output toggle rate features. Since we know the desired model is highly non-linear, we provide non-linearity in the network by adding LeakyRelu activations on the first three layers and a Softmax activation on the last layer, as the 4-dimension toggle rate features necessarily sum to 1. During training, we expect the learned embeddings will contain both predecessor and local information, and the GNN will learn the correct output toggle rates based on both local logic functions and reconvergence correlation caused by predecessors.

E. Baseline GPU-Accelerated Probabilistic SAE

We note that a conventional probabilistic SAE, much like the zero-delay mode version of [6], can be implemented using a similar PyTorch/DGL framework as the GNN model since *prop_nodes* naturally levelizes the netlist. Recent work has shown the advantages of using DL packages to accelerate EDA workloads on GPUs [14] for access to optimized GPU-accelerated libraries and software productivity. In our case, we demonstrate that we can also leverage PyTorch/DGL on GPUs to speed up probabilistic SAE.

In our baseline probabilistic SAE, we use the same toggle rate features as GRANNITE, since the 4 dimensions contain both state and transition probabilities. We also use the same graph object creation flow (Fig. 4). Local edge features are changed to a *pin_order* numerical key and node features are changed to a *cell_type* key that will be used in steps 2 and

TABLE IV: BENCHMARK CIRCUITS FOR GRANNITE

| Design | Description | Gate Count | Stimulus (40k cycles) |
|----------------|---|------------|-----------------------|
| qadd_pipe | 32-bit fixed point adder | 774 | Random |
| qmult_pipe | 32-bit fixed point multiplier | 1410 | Random |
| fadd | 32-bit floating point adder | 961 | Random |
| fmult | 32-bit floating point multiplier | 2005 | Random |
| NoCRouter | Wormhole router with virtual channels | 10,330 | Operation mode tests |
| RISC-V Core | RISC-V Rocket Core (SmallCore) | 56,243 | dhrystone benchmark |
| datapath units | shifters, encoders, muxes, leading 0/1 detectors, ... | ~1000 each | Random |

Inference is run on each circuit to verify transferability of GRANNITE.

3 of message passing, respectively. In this case, the “virtual neural network” architecture contains only a simple GNN layer. The GNN layer is edited to perform levelization and message passing as described in Table II. Message sending simply sends input pin toggle rates to the node/gate. Message reduction concatenates incoming messages in order of the *pin_order* key so that the pin toggle rates align with the ensuing lookup table (LUT) in node transform step. The LUT is a pre-loaded 3D array that stores the corresponding weights of each pin’s effect on each of the 4-dimension toggle rate features for each logic cell type (dimensions of the LUT are [*logic cell types*, *number of pins*, *4 dimension toggle rate weights*]). Node transforms perform table lookups based on *cell_type* key and matrix multiplies with the pre-aligned input toggle rates, thus completing the probabilistic SAE calculation.

We compare a probabilistic SAE engine in a widely-used commercial power analysis tool to our GPU-accelerated PyTorch/DGL framework implementation and find that we achieve similar or better accuracy at >40X speedup as shown in Table III. For the remainder of this paper, we use our PyTorch/DGL implementation as a baseline comparison for the GRANNITE ML-based SAE, and will refer to it as the *Baseline* implementation in later sections.

IV. RESULTS AND DISCUSSION

To evaluate GRANNITE, we conduct GNN model training and inference experiments on an NVIDIA Tesla V100 GPU with 16GB device memory. GRANNITE and our baseline probabilistic SAE both run on the same logic netlists, synthesized from RTL to a 16nm FinFET standard cell library using a commercial logic synthesis tool.

For training and testing datasets, we use 26 benchmark circuits listed in Table IV, containing small to medium sized units with a wide range of average toggle rates (0.01-0.30). We use open-source RTL for fixed- and floating-point arithmetic units [15]. The open-source Network-on-Chip router is written in SystemC and synthesized to RTL by a high-level synthesis (HLS) tool [16]. The RISC-V core is an RV64IMAC implementation of the open-source RocketChip Generator [17], similar to the SmallCore instance. We also include 20 randomly-chosen datapath IP blocks supplied by a commercial logic synthesis tool (~1k gates each) to increase the diversity of standard cell gate types during training.

GNN models are trained on 25 of the 26 total circuits, leaving 1 circuit outside of the training data set to be the

TABLE V: AVERAGE POWER RESULTS AND ERROR COMPARISON

| Design | Ground Truth Average Power (mW) | Baseline Predicted Power (mW) | GRANNITE Inference Power, Testing (mW) | GRANNITE Inference Power, Validation (mW) |
|----------------|---------------------------------|-------------------------------|--|---|
| qadd_pipe | 0.553 | 0.550 (0.5%) | 0.564 (2.0%) | 0.563 (1.9%) |
| qmult_pipe | 2.018 | 2.006 (0.6%) | 2.156 (6.8%) | 2.084 (3.3%) |
| fadd | 0.068 | 0.103 (51.5%) | 0.071 (4.0%) | 0.068 (0.3%) |
| fmult | 0.219 | 0.563 (157.1%) | 0.215 (1.9%) | 0.221 (1.0%) |
| NoCRouter | 1.036 | 1.088 (5.0%) | 0.897 (13.4%) | 0.918 (11.4%) |
| RISC-V | 0.923 | 0.904 (2.0%) | 0.997 (8.0%) | 0.873 (5.4%) |
| datapath_units | | N/A (3.3%) | N/A (1.0%) | N/A (0.9%) |
| AVERAGE | | (31.4%) | (5.3%) | (3.4%) |

% Error in parentheses vs. ground-truth gate-level simulations. ‘Testing’ refers to when both design and power window are excluded from the training set. ‘Validation’ refers to running inference on a new power window when the design is included in the training set.

test data set. In this way, we may verify the transferability of the proposed GNN architecture on all 26 circuits. In addition to this round-robin ‘n-1’ training/test regime, we also train a GNN model on all 26 circuits, and test on different power windows than the windows in the training data, thereby verifying accuracy on new workloads running on designs in the training set. The input toggle rate features are constructed by calculating toggle rates across a sliding power window of 750 cycles out of a total of 40k simulated cycles for each design. We train for 10 epochs, and, due to our limited data, choose a training batch size of 1 (i. e. 1 power window trained per *backprop* calculation). All training set circuits are trained in parallel by using DGL’s *dgl.batch()* API. Our loss function is Mean Square Error (MSE) loss between predicted and ground truth logic gate toggle rates. After training, inference is performed on new power windows of 1000 cycles on a new test design, or new power windows on the training designs. The output toggle rates are then translated into a SAIF file and average power is computed by a commercial power analysis tool. The same test power windows are used in the baseline probabilistic SAE implementation.

A. Average Power Estimation and Speedup Results

Table V shows the accuracy achieved by GRANNITE compared to the baseline. For brevity, we only list the mean relative error metrics for the 20 unnamed datapath units. Only the combinational average power consumption is reported as sequential power is known (from the register traces).

Compared to the baseline probabilistic SAE, GRANNITE achieves much better overall accuracy (5.3% vs. 31.4%) and achieves <10% error for all the benchmarks but the NoCRouter in the testing (transferable power estimation) results. In all benchmarks, GRANNITE’s error decreases from testing to validation, suggesting that expanding training data will improve accuracy. Since NoCRouter is a larger benchmark, we expect that there is increased likelihood that the training data did not see some feature patterns present in NoCRouter, which would explain the slightly worse error for the NoCRouter design.

Table VI shows the speedup achieved when using GRANNITE over the traditional flow of using gate-level simulation. The quoted throughput is based on running inference at 1000 cycle power windows, which is well within range

TABLE VI: SPEED COMPARISON

| Design | Batch Size=1 Inference Latency (s) | Max Windows Batch Size | Per Cycle Simulation Throughput (kHz) | GNN Inference Throughput (kHz) |
|------------|------------------------------------|------------------------|---------------------------------------|--------------------------------|
| qadd_pipe | 0.293 | 2200 | 2.5 | 5050.9 (2044.9X) |
| qmult_pipe | 0.401 | 1400 | 1.9 | 1679.5 (888.3X) |
| fadd | 0.304 | 1200 | 8.8 | 3024.5 (343.7X) |
| fmult | 0.342 | 950 | 6.2 | 1259.9 (202.5X) |
| NoCRouter | 0.435 | 175 | 1.4 | 194.7 (141.0X) |
| RISC-V | 0.703 | 30 | 2.4 | 44.4 (18.7X) |

Throughput based on 1000 cycle power windows at max batch size. Speedup (X) in parentheses. GRANNITE has an additional benefit of allowing batching of power windows to be calculated in parallel.

for average power estimation purposes. The GPU-accelerated baseline SAE throughput is reported in Table III, and is similar to GRANNITE. Since both are implemented in PyTorch/DGL, we can easily batch many power windows in parallel during one run of inference. The minimum speedup at max batch size is 18.7X, greatly outperforming the traditional approach. Of note is that speedup can be even greater with larger power window size, for example >187X for 10k cycle windows.

B. Discussion on Inference Accuracy

We analyze the embeddings learned by the GNN layer by plotting the tSNE [18] plot of all embeddings across all benchmarks in Fig. 6. We see that clustered points are comprised of similar toggle rates, evidence that GRANNITE is able to learn a wide range of toggle rates. Similar toggle rates form several different clusters, which we think is evidence GRANNITE is able to discern between different switching situations (perhaps correlated vs. uncorrelated signals).

We note that GRANNITE greatly improves the accuracy of average power estimation for the floating-point arithmetic units compared to the baseline. Table VII provides evidence of GRANNITE’s ability to infer signal correlation from the structure of the FADD graph/netlist to improve accuracy. We take the top eight most frequent cell types occurring in the FADD benchmark and record the percentage of those cells that we consider correlated. We define ‘correlated’ as those cells where the difference between the ground truth toggle rate vs. the expected toggle rate if all the inputs were completely random/uncorrelated to each other is more than 20%. We find a high percentage of cells are correlated, which explains the high error for the baseline implementation, which does not consider correlation. We also record the uncorrelated root-mean-square-error (RMSE) for these cells and compare it to the inferred RMSE. We find that for most cell types, GRANNITE improves the error over the uncorrelated case. Admittedly, we note it is hard to analyze for the 2 cell types (NAND2 and NAND3) that have higher RMSE, if their higher RMSE is caused by GRANNITE failing to learn properly, or if those cells are merely propagating higher predecessor errors. To that end, we conclude that GRANNITE shows promise in resolving the signal correlation issue during toggle rate prediction, and propose future work to continue to validate this hypothesis. Adding more circuits and larger circuits to the training dataset could improve data balance in the trained model. Adding dimensions to the register trace input features could help represent input signal correlation.

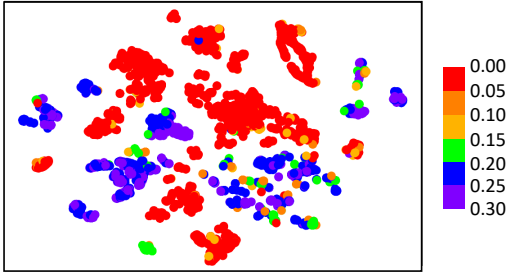


Fig. 6: tSNE plot of 128 dimension embeddings across all gates in all benchmarks. Coloring based on ground truth toggle rates per gate.

TABLE VII: ANALYSIS OF TOP 8 MOST FREQUENT CELLS IN FADD

| Cell Type | Count | Uncorrelated RMSE | GRANNITE Inference RMSE | % of Correlated Cells |
|-----------|-------|-------------------|-------------------------|-----------------------|
| NAND2 | 140 | 0.0121 | 0.0195 | 31 |
| NOR2 | 124 | 0.0152 | 0.0137 | 42 |
| AO22 | 110 | 0.0140 | 0.0128 | 33 |
| AOI22 | 50 | 0.0072 | 0.0060 | 54 |
| XOR2 | 40 | 0.0301 | 0.0119 | 33 |
| OAI21 | 34 | 0.0267 | 0.0181 | 68 |
| OAI211 | 31 | 0.0246 | 0.0117 | 81 |
| NAND3 | 28 | 0.0179 | 0.0225 | 96 |

C. Discussion on Speedup and Memory Requirements

Using the RISC-V circuit as an example, Fig. 7 shows that total inference time grows sub-linearly with regards to batch size. This means that to achieve maximum speedup for a design, we must maximize batch size. Profiling the GPU, we find low GPU utilization (at most hovers around 18%), and high memory capacity usage, meaning there is ample room for improvement. Table VIII gives some insight into memory capacity requirements. For each layer, the neural network parameters are replicated per gate/node per power window. Using this formula, it would seem the FC1 layer would allocate the most memory. However, we are limited by the GNN0 layer, which has many more parameters but a smaller nodes dimension. We expect that optimizations may be possible in the PyTorch/DGL software stack to improve the efficiency of GPU device memory allocation to alleviate this bottleneck on maximum batch size. Another possible way to increase batch size is to do model reduction, but we find our model is dense (0 sparse parameters), so conventional model reduction techniques will not apply. To this end, we leave code optimizations and graph reduction/clustering techniques to increase our maximum batch size to future work. It should be noted that even given this analysis, multiplying our maximum batch size with number of nodes shows GRANNITE can scale up to ~1.6 million gate designs (with batch size 1).

V. CONCLUSIONS

We have presented GRANNITE, a novel GNN model that achieves fast, accurate, and transferable average power estimation by inferring output toggle rates on logic gate cells and foregoing the need for slower gate-level simulation. GRANNITE achieves less than 5.5% average error and at worst 13.4% error for new benchmark circuits and new power windows during inference mode, which is a vast improvement

TABLE VIII: PARAMETER SIZE ANALYSIS

| Layer Name | Layer Size | Params | Max Nodes | Bytes/Window | Estimated Window Batch Size |
|------------|------------|--------|-----------|--------------|-----------------------------|
| FC0 | 4x128 | 640 | 56,243 | 143,982,080 | 117 |
| GNN0 | 154x128 | 19,840 | 5,203 | 412,910,080 | 41 |
| FC1 | 128x32 | 4,128 | 56,243 | 928,684,416 | 18 |
| FC2 | 32x4 | 132 | 56,243 | 29,696,304 | 569 |

RISC-V circuit is used for Max Nodes column. GNN uses double-precision float data format, so each parameter occupies 4 bytes of memory.

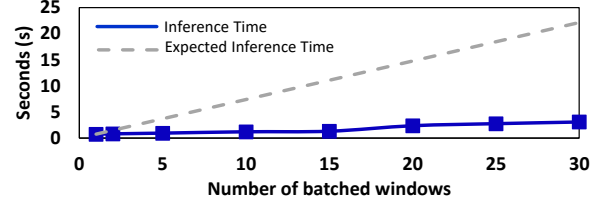


Fig. 7: Relationship between batched windows for RISC-V and total inference runtime. Runtime increases sub-linearly with increased batch size.

over the 31% average error given by a baseline probabilistic estimator implementation. In addition, it achieves >18.7X speedup when compared to traditional per-cycle gate-level simulation. We also provide evidence this approach can help in alleviating some signal correlation inaccuracies, which is an issue commonly plaguing non simulation, switching activity estimation ideas. Future work in increased training dataset sizes and input feature representations can further improve accuracy. Software optimizations or graph/model reductions can lead to increased batch sizes and improved speedups.

REFERENCES

- [1] Jianlei Yang *et al.*, “Early stage real-time SoC power estimation using RTL instrumentation,” in *ASP-DAC*, Jan 2015, pp. 779–784.
- [2] Yuan Zhou *et al.*, “PRIMAL: Power Inference Using Machine Learning,” in *DAC*, 2019, pp. 39:1–39:6.
- [3] Jie Zhou *et al.*, “Graph Neural Networks: A Review of Methods and Applications,” *CoRR*, vol. abs/1812.08434, 2018.
- [4] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” *CoRR*, vol. abs/1609.02907, 2016.
- [5] M. Nourani, S. Nazarian, and A. Afzali-Kusha, “A parallel algorithm for power estimation at gate level,” in *MWSCAS*, Aug 2002, pp. 1–511.
- [6] Huzefa Mehta, “Accurate Estimation of Combinational Circuit Activity,” in *DAC*, June 1995, pp. 618–622.
- [7] D. Chatterjee, A. DeOrio, and V. Bertacco, “Event-driven gate-level simulation with GP-GPUs,” in *DAC*, July 2009, pp. 557–562.
- [8] S. Holst, M. E. Imhof, and H.-J. Wunderlich, “High-Throughput Logic Timing Simulation on GPGPUs,” *TODAES*, pp. 37:1–37:22, Jun. 2015.
- [9] Y. Zhu, B. Wang, and Y. Deng, “Massively Parallel Logic Simulation with GPUs,” *TODAES*, vol. 16, no. 3, pp. 29:1–29:20, Jun. 2011.
- [10] D. Kim *et al.*, “Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL,” in *ISCA*, June 2016, pp. 128–139.
- [11] A. Paszke *et al.*, “Automatic differentiation in pytorch,” in *NIPS*, 2017.
- [12] Minjie Wang *et al.*, “Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs,” *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [13] B. Shuai, Z. Zuo, G. Wang, and B. Wang, “DAG-Recurrent Neural Networks For Scene Labeling,” *CoRR*, vol. abs/1509.00552, 2015.
- [14] Yibo Lin *et al.*, “DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement,” in *DAC*, 2019, pp. 117:1–117:6.
- [15] “OpenCores.org Fixed Point Math Library for Verilog Manual,” https://opencores.org/project/verilog_fixed_point_math_library/manual.
- [16] Bruce Khailany *et al.*, “A modular digital vlsi flow for high-productivity soc design,” in *DAC*, 2018, pp. 72:1–72:6.
- [17] K. Asanović *et al.*, “The Rocket Chip Generator,” EECS, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.
- [18] L. van der Maaten and G. E. Hinton, “Visualizing High-Dimensional Data Using t-SNE,” *Journal of MLR*, vol. 9, pp. 2579–2605, 2008.