Task-parallel Heterogeneous Programming System for Logic Simulation

by

Dian-Lun Lin

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 07/11/2024

The dissertation is approved by the following members of the Final Oral Committee:

Tsung-Wei Huang, Professor, Electrical and Computer Engineering, Chair
Azadeh Davoodi, Professor, Electrical and Computer Engineering
Mikko Lipasti, Professor, Electrical and Computer Engineering
Dan Negrut, Professor, Mechanical Engineering
Haoxing Ren, Director, NVIDIA Design Automation Research

To my family and friends, whose unwavering support and encouragement have been my guiding light throughout this journey.

ACKNOWLEDGMENTS

Completing this PhD thesis has been a challenging yet profoundly rewarding journey, and I am deeply grateful for the support and encouragement I have received from many individuals and institutions along the way.

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Tsung-Wei Huang. Your guidance, insight, and unwavering support have been instrumental in shaping this research. Your patience and constructive feedback have challenged me to grow as a researcher and thinker, and for that, I am truly thankful.

I would also like to extend my heartfelt thanks to my thesis committee members, Dr. Haoxing Ren, Dr. Azadeh Davoodi, Dr. Mikko Lipasti, and Dr. Dan Negrut. Your valuable suggestions and critical evaluations have greatly enhanced the quality of this work.

I would also like to thank my mentors during my internship at NVIDIA Research, Yanqing Zhang, Haoxing Ren, and Brucek Khailany. Your guidance and mentorship have been pivotal to my professional growth.

Special thanks go to my colleagues and fellow PhD candidates in our group. The camaraderie and collaborative spirit within our group have made the long hours of research more enjoyable and fulfilling. I am particularly grateful for your friendship and support.

I am also indebted to the administrative and technical staff at University of Wisconsin-Madison. Your assistance with logistical, technical, and administrative matters has been invaluable throughout my doctoral studies.

On a personal note, I would like to thank my family for their unwavering support and encouragement. To my parents, En-Rong Lin and Mei-Ying Chen, thank you for believing in me and for your endless love and sacrifices.

I would also like to thank my girlfriend, Wan Luan Lee, your love,

patience, and understanding have been my anchor during the most challenging times of this journey.

Finally, I thank my two fluffy pets, Luna and April. Thank you for always being there and for helping me stay focused and motivated.

This thesis is a testament to the collective support of all these wonderful people and institutions. Thank you all for being a part of this journey.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis addresses critical challenges in the realm of parallel and heterogeneous computing, with a specific focus on logic simulation, which is crucial for the design and verification of modern digital systems. As the complexity of neural networks and hardware designs continues to grow, traditional computing methods struggle to meet the demands for higher performance and efficiency. This research aims to develop innovative solutions to these challenges by leveraging advanced parallel computing techniques and heterogeneous computing architectures.

1. *SNIG: A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism.* The increasing size and complexity of deep neural networks (DNNs) necessitate efficient computation methods. SNIG addresses this need by introducing an efficient inference engine for large sparse DNNs, utilizing CUDA Graphs to optimize model and data parallelism. This work not only enhances inference speed but also demonstrates significant scalability, crucial for applications in AI and machine learning where rapid processing of vast amounts of data is essential.

2. *cudaFlow: Efficient GPU Computation using Task Graph Parallelism.* As GPU architectures evolve, the overhead associated with launching individual GPU operations becomes a bottleneck. cudaFlow presents a lightweight task graph programming framework that simplifies the use of CUDA Graphs, enabling efficient GPU computation by automating the management of stream concurrency and event dependencies. This advancement is pivotal for developers aiming to maximize GPU utilization and improve the performance of GPU-accelerated applications.

3. *RTLflow: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus*. The verification of complex hardware designs through RTL simulation is

traditionally time-consuming. RTLflow addresses this by transforming RTL code into CUDA kernels, leveraging GPU acceleration to process batch stimuli efficiently. This approach significantly reduces simulation time, providing a robust solution for the hardware design industry where faster verification cycles can lead to reduced time-to-market and increased innovation.

4. *GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs.* Hardware fuzzing is essential for detecting vulnerabilities in digital designs. GenFuzz introduces a novel GPU-accelerated fuzzing technique using a genetic algorithm to handle multiple inputs simultaneously. This method dramatically improves the speed and effectiveness of hardware fuzzing, crucial for ensuring the security and reliability of hardware components.

5. *TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling*. Building on the success of RTLflow, TaroRTL incorporates coroutine-based scheduling to further enhance the efficiency of CPU-GPU task management. This technique allows for better overlap and synchronization of tasks, reducing overall simulation time and maximizing hardware utilization. Such improvements are vital for the continued advancement of high-performance computing and the development of complex digital systems.

In summary, this thesis contributes to the advancement of parallel and heterogeneous computing by addressing key bottlenecks in logic simulation and neural network inference. Through the development of innovative algorithms and frameworks, this research provides scalable and efficient solutions that are essential for meeting the demands of modern computational workloads.

PREVIOUS WORK

The research presented in this thesis builds upon a rich body of previous work in the fields of neural network inference, GPU computation, RTL simulation, hardware fuzzing, and task scheduling. Below, we summarize previous work relevant to each chapter of the thesis.

1. *SNIG: A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism*. Previous work on large sparse neural network inference has focused on optimizing sparse matrix operations and improving parallelism in neural network computations [5, 6]. Additionally, the MIT/IEEE/Amazon HPEC Graph Challenge [1] has spurred innovations in high-performance inference methods for sparse DNNs, highlighting the need for scalable and efficient algorithms. The BF method [7] is implemented with CUDA+OpenMP. However, their load-balancing method requires communication between CPUs and GPUs at each iteration, resulting in huge overhead. Similar problems also exist in other pipeline-based frameworks. For example, GPipe [8] proposes pipelining computation across GPUs and synchronizing data transfers stage by stage.

2. *cudaFlow: Efficient GPU Computation using Task Graph Parallelism*. In the domain of GPU-accelerated computation, substantial progress has been made in developing frameworks that exploit task graph parallelism. [9] presents a compiler transformation method that translates OpenMP code into CUDA graphs. However, their transformation method only considers explicit graph construction. [10] proposes a compiler-based approach that combines CUDA graph with an image processing DSL and a source-to-source compiler called Hipacc. Their kernel pipelining approach optimizes the schedule specifically for the scattering-pattern applications. [11] presents the Hybrid Task Graph Scheduler (HTGS)

to aid in building hybrid workflows for high performance image processing.

Graph-based model is extensively studied on CPU-parallel architectures. Just name a few: Taskflow [12, 13, 14, 15, 16, 17] develops a simple and powerful task programming model enabling efficient implementations of heterogeneous decomposition strategies. Kokkos [18] uses functional approaches to offer task graph constructions. It enables applications to achieve performance portability on diverse many-core architectures. Legion [19] describes a runtime system that dynamically extracts parallelism from Legion programs, using a distributed, parallel scheduling algorithm that identifies both independent tasks and nested parallelism. These models have their own pros and cons, but they do not target GPU graph parallelism.

3. *RTLflow: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus*. RTL simulation has traditionally been a bottleneck in the hardware design verification process. Verilator [20] has explored RTL partitioning to support multi-threading. ESSENT [21] is a single-threaded simulator which introduces an event-driven algorithm using conditional execution to skip over unnecessary simulation work. ESSENT has shown up to 2–10$\times$ speed-up over single-threaded Verilator but the result does not scale well to large designs due to the lack of multi-threading. CXXRTL is a Yosys [22] simulation back-end which transpiles an internal representation (IR) generated by the Yosys front-end to C++ simulation code. However, CXXRTL suffers from extremely long compilation time on large designs and does not have any multi-threading capability.

   GPU-based RTL simulation has been investigated before, but is limited to a single input stimulus. For instance, [23] offloads the simulation workload to GPU by mapping each RTL process (a group of simulation instructions) to a GPU kernel using one thread per warp. This mapping, however, is not efficient since other threads within a warp are not

utilized. [24, 25, 26, 27] use GPU to accelerate gate-level simulation. Nevertheless, gate-level simulation techniques are not suitable for RTL simulation because of different objectives in design flow.

4. *GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs*. Hardware fuzzing has emerged as a promising automatic verification technique to efficiently discover and verify hardware vulnerabilities. Recent research has explored several approaches to speed up the single-input, single-threaded, per fuzzing iteration time. RFUZZ [28] proposes a mux-coverage metric that treats the select signal of each 2:1 multiplexer as a coverage point. However, RFUZZ cannot scale to large designs since their runtime grows significantly as the number of multiplexers increases. DirectFuzz [29] extends RFUZZ to generate test inputs that maximize the coverage of a specific block. However, the speedup on complex designs is insignificant (e.g., 1.08× on the Sodor1Stage RISC-V processor). DIFUZZRTL [30] introduces a reg-coverage metric to monitor value changes of control registers connected to mux control signals. While DIFUZZRTL's reg coverage shows capability for large designs, their fuzzing technique requires many hours or days to achieve high coverage. TheHuzz [31] explores processor states using multiple coverage metrics. However, it induces significant runtime overhead when collecting coverage data since it must access multiple metrics per fuzzing iteration. Hw-Fuzzing [32] converts a hardware-description-language model into an equivalent software model using Verilator, and performs fuzzing on the software code using software coverage metrics. However, other hardware coverage metrics such as finite-state-machine (FSM) coverage cannot be easily added.

5. *TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling*. Existing RTL simulators have leveraged task graph parallelism to accelerate simulation on a CPU- and/or GPU-

parallel architecture. Verilator [20], an open-source RTL simulator, aggregates adjacent logic elements into macro tasks, which are then scheduled using a static multi-threaded algorithm, achieving optimal performance with 8-10 CPU cores. RepCut [33] improves upon this by partitioning circuits into balanced segments with minimal overlap, reducing synchronization overhead and achieving superlinear speed-ups through task replication. However, it is restricted to strong scaling for single input stimuli. RTLflow [4] further innovates by executing multiple independent input stimuli in parallel on a GPU, utilizing a heterogeneous task set and a work-stealing scheduling algorithm. Despite its advancements, RTLflow encounters significant CPU idle time, as CPU threads must await the completion of GPU tasks within each simulation cycle. Taskflow [34] is a task graph scheduling system that has been adopted by many EDA algorithms [35, 36, 37], including RTLflow. It develops a simple and powerful task programming model enabling efficient implementations of heterogeneous decomposition strategies. However, Taskflow does not support multitasking in a task graph. Libfork [38] is a coroutine-tasking library that is primarily an abstraction for fully-portable, strict, fork-join parallelism. However, it does not support heterogeneous parallelism.

1 SNIG: A NOVEL INFERENCE ALGORITHM FOR LARGE SPARSE NEURAL NETWORK USING TASK GRAPH PARALLELISM

1.1 Abstract

The ever-increasing size of modern deep neural network (DNN) architectures has put increasing strain on the hardware needed to implement them. Sparsified DNNs can greatly reduce memory costs and increase throughput over standard DNNs, if the loss of accuracy can be adequately controlled. However, sparse DNNs present unique computational challenges. Efficient model or data parallelism algorithms are extremely hard to design and implement. The recent effort MIT/IEEE/Amazon HPEC Graph Challenge has drawn attention to high-performance inference methods for large sparse DNNs. In this chapter, we introduce SNIG, an efficient inference engine for large sparse DNNs. SNIG develops highly optimized inference kernels and leverages the power of CUDA Graphs to enable efficient decomposition of model and data parallelisms. Our decomposition strategy is flexible and scalable to different partitions of data volumes, model sizes, and GPU numbers. We have evaluated SNIG on the official benchmarks of HPEC Sparse DNN Challenge and demonstrated its promising performance scalable from a single GPU to multiple GPUs. Compared to the champion of the 2019 HPEC Sparse DNN Challenge, SNIG can finish all inference workloads using only a single GPU. At the largest DNN, which has more than 4 billion parameters across 1920 layers each of 65536 neurons, SNIG is up to 2.3× faster than a state-of-the-art baseline under a machine of 4 GPUs.

1.2 Introduction

Larger deep neural network (DNN) models have brought significant quality improvement to several fields, including natural language processing, speech recognition, and image classification [39, 40, 41]. To relieve the increasing strain on the hardware needed to deploy them, much research over the past decades has focused on the *sparsification* of DNNs in the interest of reduced storage and runtime costs [5, 6, 42]. Computing large sparse DNNs presents unique computational challenges and scaling difficulties. Sparseness can make the application of the DNN on current processors extremely inefficient. This inefficiency limits the size of data to what can be held in GPU memory, or it requires a high-end, expensive cluster of computers to make up for this inefficiency [43]. Also, sparse DNN inference presents unique computational challenges from training, because the kernel efficiency largely depends on non-zero entries that vary from layer to layer. To address these problems for advancing emerging sparse machine learning (ML) systems, the 2019 MIT/IEEE/Amazon HPEC Graph Challenge has developed Sparse DNN Challenge to encourage new solutions for sparse DNN inference [1]. Table 1.1 lists the statistics of each sparse DNN. The largest network contains over 4 billion nonzero parameters across 1920 layers each of 65536 neurons, adding up to 100 GB memory storage.

Neurons/Layers	120	480	1920	Bias	Size	Image Nonzeros
1024	3.9M	15.7M	62.9M	-0.30	1.25 GB	6,374,505
4096	15.7M	62.9M	251.7M	-0.35	5.40 GB	25,019,051
16384	62.9M	251.7M	1.0B	-0.40	22.70 GB	98,858,913
65536	251.7M	1.0B	4.0B	-0.45	94.70 GB	392,191,985

Table 1.1: The statistics of each DNN benchmark in the Challenge [1].

The challenge of computing large sparse DNN inference is twofold, *kernel* and *decomposition algorithms*, both of which require strategic designs

to benefit from parallelism. Existing kernel algorithms focus on optimizing sparse matrix-matrix multiplication kernels or carefully maintaining data sparsity during the weight propagation [44, 45, 46, 47]. However, most of these approaches require models to sit in the GPU memory, and they are difficult to operate on partitioned pieces, due to the cost of maintaining consistent sparse matrix structures between partitions along with iterations. Existing decomposition strategies divide large data or models into partitions and distribute partitions across GPUs [48, 49, 50, 51]. Partitioning data and models can both improve parallelism and alleviate the tension on hardware constraints, including memory limitations and communication bandwidths on GPUs. However, efficient decomposition algorithms are extremely hard to design and implement. We need to address complexity among GPU capacity, scaling flexibility, and inference efficiency. To simplify the design, *pipeline parallelism* has been a popular choice in existing frameworks [7, 8, 52, 53]. The idea of the pipeline is simple and easy to implement, but it suffers from many performance problems, including synchronous execution, imbalanced pipeline stages, and limited pipeline depth.

As a consequence, we introduce SNIG[1], an efficient large sparse DNN inference engine using *task graph parallelism*. SNIG develops highly optimized inference kernels that can effectively avoid unnecessary computation incurred by zero entries during the inference iterations. We leverage the power of modern CUDA Graph [54, 12] to enable efficient decomposition of model and data parallelisms. Our decomposition strategy transforms a partitioned inference workload into a *task dependency graph* that flows CPU-GPU operations naturally with the graph structure, providing improved scheduling efficiency and runtime asynchrony. Compared with pipeline-based frameworks, SNIG is more flexible and cost-efficient in fitting together partitioned data and models into different GPUs under

[1]source code: https://github.com/dian-lun-lin/SNIG

hardware constraints. We demonstrate the flexibility and efficiency of SNIG on the 12 large sparse DNNs provided by the 2019 HPEC Sparse DNN Challenge [1]. SNIG is able to complete all DNNs using only one RTX 2080 Ti GPU of 11 GB memory, and we solve the largest DNN by 2.27× faster than the 2019 champion solution developed by Bisson and Fatica ("BF" method for brevity) [7]. Compared with a pipeline baseline inspired by GPipe [8], SNIG is faster at almost all networks (up to 2.19× speed-up) and scales better on multiple GPUs. We believe SNIG stands out as a unique inference engine for large sparse DNNs, given the ensemble of algorithm tradeoffs and decomposition decisions we have made.

1.3 Problem Formulation of Large Sparse DNN Inference

We target on the 2019 HPEC Sparse DNN Challenge, which is based on a mathematically well-defined DNN inference computation and can be implemented in any programming environment [1]. The input data, Y_0, is derived from the MNIST handwritten letters by resizing each 28×28 pixel image to 32×32 (1024 neurons), 64×64 (4096 neurons), 128×128 (16384 neurons), and 256×256 (65536 neurons). The weight matrices of each sparse DNN, including the bias vectors, are generated by the RadiX-Net synthetic sparse DNN generator with a number of desirable properties such that participants can focus on the difficult, computational part of the problem [55]. The inference problem is to compute $Y_{l+1} = h(Y_l W_l + B_l)$ for each layer where $h(y) = max(y, 0)$ is a nonlinear function of rectified linear unit (ReLU). For the Sparse DNN Challenge, $h(y)$ has an upper limit set to 32. The surrounding I/O and verification provide the context for each sparse DNN inference that allows rigorous definition of both the input and the output. Table 1.1 lists the statistics of each sparse DNN

and its input image set. Loading the smallest DNN can take gigabytes of memory using single-precision floating numbers. Preloading all matrices to GPUs is impractical and discouraged.

1.4 State of the Art: The BF and Pipeline Methods

The BF method [7] is implemented with CUDA+OpenMP. Each GPU owns a part of the input matrix and computes the inference kernel iteratively by one OpenMP thread. At each iteration, each GPU executes two kernels, one for the inference and the other for calculating the non-empty row indices in the resulting matrix. After all GPUs complete execution, the OpenMP threads compute the new global list of non-empty rows and repartition the non-empty rows evenly among the GPUs. However, such a load-balancing method requires communication between CPUs and GPUs at each iteration, resulting in huge overhead. Also, to compute the list of non-empty rows, all GPUs need to be synchronized at each iteration. Synchronization can lead to unnecessary waiting time and waste computing power of GPUs. Besides, BF requires the entire input data to sit in GPUs for implementing load balancing. Similar problems also exist in other pipeline-based frameworks. For example, GPipe [8] proposes pipelining computation across GPUs and synchronizing data transfers stage by stage. The efficiency and scalability are largely limited by the size of partitioned data and available GPU resources that decide the degree of pipeline parallelism.

1.5 SNIG

At a high level, SNIG describes the inference workload in a *task graph* comprising both data- and model-level parallelisms. Our task graph can scale

to arbitrary sizes of DNN and input data under different numbers of GPUs. We develop an *efficient kernel* inside the task graph that computes only necessary entries during the inference iterations. Our in-kernel pruning strategy avoids unwanted computation incurred by sparsified network and data, in no need of additional CPU-GPU or GPU-GPU synchronization to redistribute input data among GPUs.

Task graph parallelism

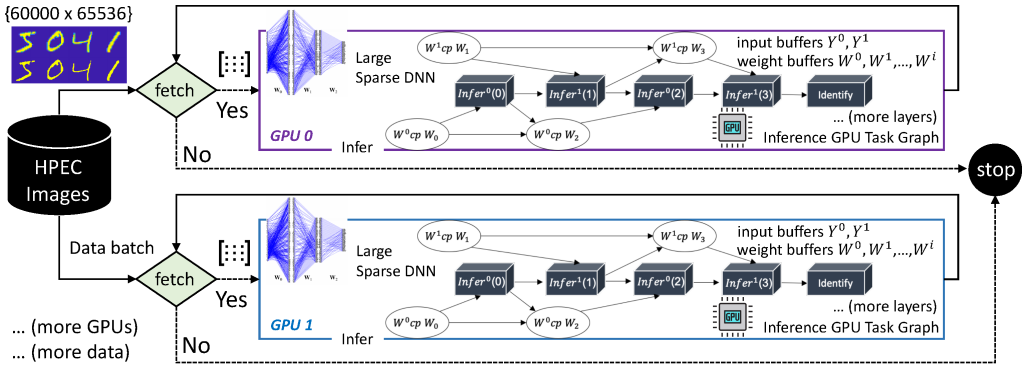


Figure 1.1: Architecture of SNIG.

Figure 1.1 shows the overview of SNIG. SNIG defines the inference workload as a task dependency graph that iterates two stages: *fetch* and *infer*. At the fetch stage, a CPU task grabs a batch of input data of up to size `batch_size`. Users can tune `batch_size` based on available GPU memory. To have multiple threads fetch data at the same time, we use an *atomic* counter to represent the remaining size of data. At the infer stage, a GPU task computes the inference of the batch on a GPU. Each GPU task consists of a *GPU task dependency graph* where each node represents one of the three GPU operations, host-to-device (H2D) copy, device-to-host (D2H) copy, and kernel tasks; each edge represents the dependency of two GPU operations. We leverage the power of modern *cudaGraph* [54] to offload a GPU task dependency graph using a single CPU call, thus

reducing overheads. The architecture of SNIG is decentralized. There is no local or global CPU-GPU synchronization during the inference on a dataset.

We transpose weight matrices and store them using the Compressed Sparse Column (CSC) format. Since preloading all models to the GPU is impossible due to memory limit, we only keep up to $num\_weights$ weight buffers (W^0, W^1, ..., $W^{num\_weights-1}$) on a GPU at a time. All weight buffers have the same size equal to the maximum size of W_l. More weight buffers result in a higher overlap between data communication and kernel computation. Since the inference at one layer only depends on the result from the previous layer, we allocate for each GPU two *result buffers* Y^0 and Y^1 each of size $batch\_size \times num\_neurons$ (number of neurons) to perform rolling swap for space optimization. Each buffer can be accessed via modulo operation on 2; $Infer^{l\%2}(l)$ represents applying the inference kernel to W_l using $Y^{l\%2}$ as input and $Y^{(l+1)\%2}$ as output. After completing the inference at the last layer, the GPU identifies the categories (predicted digits). Users can configure different $batch\_size$ and $num\_weights$ based on available GPU memory to fit arbitrary sizes of models and input data.

Inference kernel

At the *infer* stage, our inference kernel consists of two parts: *forward feeding* and *incremental memory resetting*. Figure 1.2 illustrates one iteration of one row of input data in our kernel. To improve parallelism, we divide each input data into num_secs sections where each of sec_size is $num\_neurons/num\_secs$. Since each GPU keeps Y^0 and Y^1 to perform rolling swap, we allocate for each GPU two $batch\_size \times num\_secs$ boolean buffers, $is\_nonzero\_row^0$ and $is\_nonzero\_row^1$, to record whether a section of data contains nonzero elements. At the beginning of the inference kernel, we inspect each entry in $is\_nonzero\_row^0[r]$.

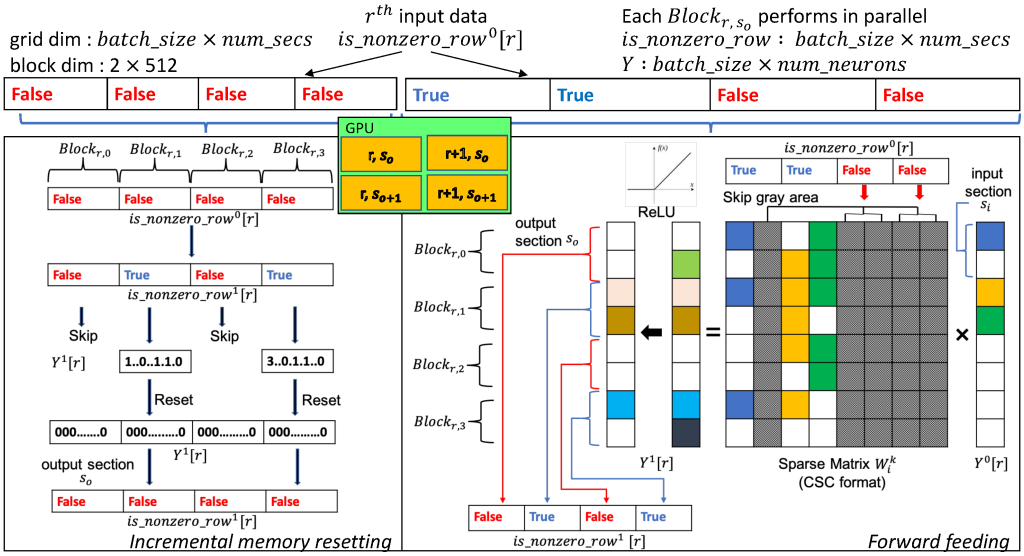


Figure 1.2: Illustration of our inference kernel (Algorithm 1).

If there exists at least one true value, meaning that there is at least one nonzero element in r^{th} input data, we enter *forward feeding*. The *forward feeding* performs matrix multiplication followed by ReLU and passes the result to the next layer via rolling swap. We skip input section s_i ($Y^0[r][k], sec\_size \times s_i \leqslant k < sec\_size \times (s_i + 1)$) that contains only zero entries indicated by $is\_nonzero\_row^0[r][s_i]$ to avoid unnecessary computation. During the matrix multiplication, we can further skip zero input entries. Taking advantage of rolling swap, we perform *incremental memory resetting* to reset buffers. If all entries in $is\_nonzero\_row^0[r]$ are false, we reset the output section s_o ($Y^1[r][k], sec\_size \times s_o \leqslant k < sec\_size \times (s_o + 1)$) including nonzero elements based on $is\_nonzero\_row^1[r][s_o]$. This largely avoids the overhead to reset the entire linear buffer for the next iteration to use. Our implementation computes each output section s_o of each data in parallel and calculates only necessary entries during inference iterations.

Algorithm 1 presents the details of our kernel. The grid dimension is $(batch\_size, num\_secs)$ and the block dimension is $(2, 512)$. We allocate $4 \times sec\_size$ byte of external shared memory. The kernel is launched

by $<<<(batch\_size, num\_secs), (2, 512), 4\times sec\_size>>>$. Each block computes an output section s_o of one row of input data independently.

At the beginning, each block $Block_{r,s_o}$ in the grid determines to execute either *forward feeding* or *incremental memory resetting*. We use is_all_zero to record if all entries in $is\_nonzero\_row^0[r]$ are false (line 5-8). If is_all_zero is true, that is, all elements in $Y^0[r]$ are zero, $Block_{r,s_o}$ enters *incremental memory resetting*. During *incremental memory resetting*, if $is\_nonzero\_row^1[r][s_o]$ is true, $Block_{r,s_o}$ resets all elements of s_o to zero and toggles $is\_nonzeero\_row^1[r][s_o]$ to false (line 10-16). Otherwise, it returns directly (line 17).

$Block_{r,s_o}$ starts *forward feeding* if is_all_zero is false (line 19-52). Each block declares a shared memory array $results$ size of sec_size to store results (line 19) and initializes $results$ to the bias value directly (line 20-22). To avoid synchronization, we use a boolean array $is\_nonzero$ size of 2 to record whether $results$ has nonzero values (line 23-24, line 49). $Block_{r,s_o}$ iterates all input sections to compute results (line 26). If the current entry in $is\_nonzero\_row^0[r]$ is false, meaning that the current input section s_i contains only zero elements, we skip all elements in s_i directly (line 27-29). Otherwise, all threads along y dimension loop through all entries in s_i (line 31). We further skip to the next one if the current input value is zero (line 33-35). All threads along x dimension read col_w (line 36-37) and iterate the weight values and the weight row indices (line 38-40). To compute each s_o independently, we transform the dimension of each CSC weight matrix from $(num\_neurons, num\_neurons)$ to $(num\_neurons, num\_secs \times num\_neurons)$. All column indices are shifted by $j = j + num\_neurons \times (i/sec\_size)$, where (i, j) is the nonzero index of the weight matrix. In line 36-37, we read column indices of the weight matrix via adding the offset. Then, we multiply each nonzero input entry with weight value and add the result to the corresponding location of $results$ (line 41).

After computing results, $\texttt{Block}_{r,s_o}$ loops through the results (line 46). It computes ReLU, writes the result to each element in s_o, and sets is_nonzero[1] to true if there exists a nonzero result in s_o (line 47-49). Finally, we toggle $\texttt{is\_nonzero\_row}^1[r][s]$ to either true or false based on is_nonzero[1] (line 52).

1.6 Experimental Results

We evaluate SNIG's performance on the official MIT/IEEE/Amazon HPEC Sparse DNN Challenge Dataset [1]. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, 4 GeForce RTX 2080 Ti GPUs with 11 GB memory, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 10.1 on a host compiler of GNU GCC-8.3.0 with C++14 standards `-std=c++14` and optimization flags `-O2` enabled. All data is an average of ten runs with `float` type.

Baseline

We consider BF and GPipe* methods for our baseline. The BF method is the champion solution of the 2019 HPEC Sparse DNN Challenge [7]. We implemented the BF method and its kernel using CUDA streams and OpenMP. The original BF method relies on NVLink to transparently exchange data among GPUs using unified addressing. Since we do not have NVLink, such a process can be very time-consuming. We manually partition the input data in the beginning evenly across GPUs and spawn one OpenMP thread to call the inference function per GPU. We implemented the GPipe* method based on GPipe [8]. GPipe is an iterative framework for training large DNNs. We extended its idea to inference by partitioning the DNN into multiple stages across GPUs and pipelining each data batch's execution over these stages using CUDA streams and OpenMP

threads. For fair purposes, the inference kernel inside the pipeline is the same as SNIG. We configure the block dimension of all kernels to 2×512 , the batch size of input data to 5000 for SNIG and GPipe*, , and the number of weight buffers to 2 for SNIG . We will discuss the effect of different parameters in the later section.

Algorithm 1: Inference kernel

Input: col_w: array of column offsets of the weight matrix
Input: row_w: array of row indices
Input: val_w: array of values
r ← **block**.x
s_o ← **block**.y
tid ← **thread**.y * **blockDim**.x + **thread**.x
num_threads ← **blockDim**.x * **blockDim**.y
is_all_zero ← **true**
for s_i ← *0*; s_i < num_secs; ++s_i **do**
  is_all_zero &= !is_nonzero_row^0[r][s_i]
end
if is_all_zero == *true* **then**
  if is_nonzero_row^1[r][s_o] == *true* **then**
    for j ← tid; j < sec_size; j += num_threads **do**
      Y^1[r][sec_size * s_o + j] = 0
    end
    __syncthreads()
    is_nonzero_row^1[r][s_o] ← **false**
  end
  return
end
extern **__shared__** results[]
for k ← tid; k < sec_size; k += num_threads **do**
  results[k] ← bias
end
__shared__ is_nonzero[2]
is_nonzero[1] ← **false**
__syncthreads()
for s_i ← *0*; s_i < num_secs; ++s_i **do**
  if !is_nonzero_row^0[r][s_i] **then**
    continue
  end
  j ← **thread**.y + s_i ∗ sec_size
  for j; j < (s_i + *1*) * sec_size; j += **blockDim**.y **do**
    y^{val} ← Y^0[r][j]
    if y^{val} == *0* **then**
      continue
    end
    w^- ← col_w[s_o * num_nurons + j] + **thread**.x
    w^+ ← col_w[s_o * num_neurons + j + 1]
    for k ← w^-; k < w^+; k += blockDim.x **do**
      w^{row} ← row_w[k]
      w^{val} ← val_w[k]
      atomicAdd(&results[w^{row} - s_o * sec_size], y^{val} * w^{val})
    end
  end
end
__syncthreads()
for i ← tid; i < sec_size; i += num_threads **do**
  v ← min(32, max(results[i], 0))
  Y^1[r][s_o * sec_size + i] ← v
  is_nonzero[$v \neq 0$] ← **true**
end
__syncthreads()
is_nonzero_row^1[r][s_o] = is_nonzero[1]

		Number of GPUs										
		1		2			3			4		
Neurons	Layers	BF	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG	BF	GPipe*	SNIG
1024	120	**345.93** (0.682s)	295.28 (0.799s)	576.84 (0.409s)	**589.82** (0.400s)	455.46 (0.518s)	**761.06** (0.310s)	695.95 (0.339s)	689.85 (0.342s)	867.38 (0.272s)	768.50 (0.307s)	**1248.30** (0.189s)
	480	477.83 (1.975s)	**586.52** (1.609s)	801.11 (1.178s)	**1016.93** (0.928s)	926.12 (1.019s)	1061.55 (0.889s)	1273.57 (0.741s)	**1348.16** (0.700s)	1112.87 (0.848s)	1483.83 (0.636s)	**1982.60** (0.476s)
	1920	524.50 (7.197s)	**718.74** (5.252s)	852.50 (4.428s)	**1187.81** (3.178s)	1184.45 (3.187s)	1133.59 (3.330s)	1575.48 (2.396s)	**1647.69** (2.291s)	1220.45 (3.093s)	1876.17 (2.012s)	**2159.53** (1.748s)
4096	120	409.42 (2.305s)	**586.52** (1.609s)	746.02 (1.265s)	934.37 (1.010s)	**980.99** (0.962s)	1106.35 (0.853s)	1053.25 (0.896s)	**1460.86** (0.646s)	1385.78 (0.681s)	1165.08 (0.810s)	**2241.61** (0.421s)
	480	544.55 (6.932s)	**803.84** (4.696s)	962.73 (3.921s)	1376.68 (2.742s)	**1400.69** (2.695s)	1431.50 (2.637s)	1767.26 (2.136s)	**2062.77** (1.830s)	1743.59 (2.165s)	2069.5 (1.824s)	**2761.42** (1.367s)
	1920	586.38 (25.75s)	**867.28** (17.41s)	1032.09 (14.63s)	1551.53 (9.732s)	**1575.48** (9.584s)	1538.09 (9.817s)	2074.67 (7.278s)	**2284.34** (6.610s)	1879.21 (8.035s)	2506.97 (6.023s)	**2948.54** (5.121s)
16384	120	462.32 (8.165s)	**851.53** (4.433s)	881.36 (4.283s)	1290.55 (2.925s)	**1487.34** (2.538s)	1303.47 (2.896s)	1521.51 (2.481s)	**2183.26** (1.729s)	1621.50 (2.328s)	1684.45 (2.241s)	**2914.96** (1.295s)
	480	616.30 (24.50s)	**1076.99** (14.02s)	1137.01 (13.28s)	1887.67 (7.999s)	**1965.31** (7.683s)	1678.28 (8.997s)	2454.80 (6.151s)	**2824.44** (5.346s)	2072.39 (7.286s)	2894.28 (5.217s)	**3736.57** (4.041s)
	1920	663.34 (91.05s)	**1113.94** (54.22s)	1207.71 (50.01s)	2105.92 (28.68s)	**2127.43** (28.39s)	1808.86 (33.39s)	2817.06 (21.44s)	**3022.92** (19.98s)	2230.35 (27.08s)	3412.31 (17.70s)	**3963.12** (15.24s)
65536	120	28.79 (524.3s)	**1021.61** (14.78s)	57.52 (262.5s)	1323.35 (11.41s)	**1870.36** (8.073s)	1332.70 (11.33s)	1486.17 (10.16s)	**2705.51** (5.581s)	1652.74 (9.136s)	1565.85 (9.643s)	**3436.38** (4.394s)
	480	(>1800s)	**1404.60** (43.00s)	58.81 (1027s)	2083.40 (28.99s)	**2583.31** (23.38s)	1817.57 (33.23s)	2768.00 (21.82s)	**3784.33** (15.96s)	2241.94 (26.94s)	3222.94 (18.74s)	**5071.19** (11.91s)
	1920	(>1800s)	**1489.46** (162.2s)	(>1800s)	1501.50 (160.9s)	**2810.51** (85.96s)	1960.97 (123.2s)	1948.32 (124.0s)	**4149.63** (58.22s)	2450.47 (98.59s)	2784.27 (86.77s)	**5561.50** (43.44s)

Table 1.2: Overall inference rate (gigaedges processed per second) and runtime performance (seconds) of SNIG, BF, and GPipe* across one, two, three, and four GPUs. Bold text represents the best solution in the corresponding benchmark.

Performance Comparison

Table 1.2 compares the overall inference rate and runtime performance between SNIG, BF, and GPipe* using one, two, three, and four GPUs. All results match the golden reference provided by the MIT/IEEE/Amazon Sparse DNN Challenge [1]. Since the GPipe* method is staged on the number of GPUs, we do not report its runtime under one GPU. The result of BF method is different from BF paper due to different GPU platforms. SNIG outperforms BF and GPipe* across nearly all benchmarks. With 4 GPUs, SNIG is 2.3× faster than BF on the largest DNN of 65536 neurons and 1920 layers and is 2.2× faster than GPipe* on the DNN of 65536 neurons and 120 layers. The BF method failed to finish the largest DNN of 65536 neurons and 1920 layers within a reasonable amount of time ($>$ 1800 seconds) under one and two GPUs. This is because BF requires the entire input data to sit in the GPU under unified memory addressing to implement load balancing. CUDA will keep fetching in and out data between CPUs and GPUs if partitioned data does not fit in a GPU's memory. Its kernel design is architecturally constrained by the number of GPUs and available memory. Similar problems exist in the GPipe* method as well since GPipe* requires the entire model to sit in GPUs. We observe long runtime of GPipe* to complete the DNNs of 65536 neurons and 1920 layers.

Figure 1.3 plots the scalability over increasing number of GPUs. Our runtime scales the best among the three methods. In the 16384×1920 scenario, SNIG speeds up BF by 1.7×, 1.8× , 1.7×, and 1.8× at 1, 2, 3, and 4 GPUs, respectively. In the 65536×1920 scenario, SNIG speeds up GPipe* by 1.9×, 2.1× , 2.0× at 2, 3, and 4 GPUs, respectively. We attribute this to the synchronization overhead of both methods (BF at each iteration, GPipe* at each pipeline stage). Figure 1.4 plots the scalability over increasing number of neurons. SNIG outperforms BF and GPipe* in all scenarios. The growth rate of our runtime is much slower than BF and GPipe*, due to our in-

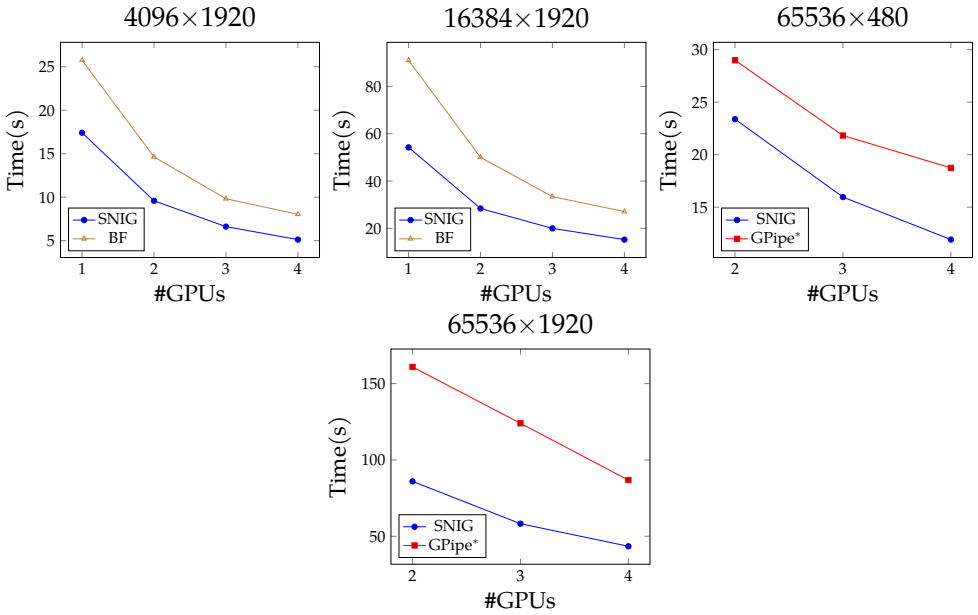


Figure 1.3: Execution time with different numbers of GPUs.

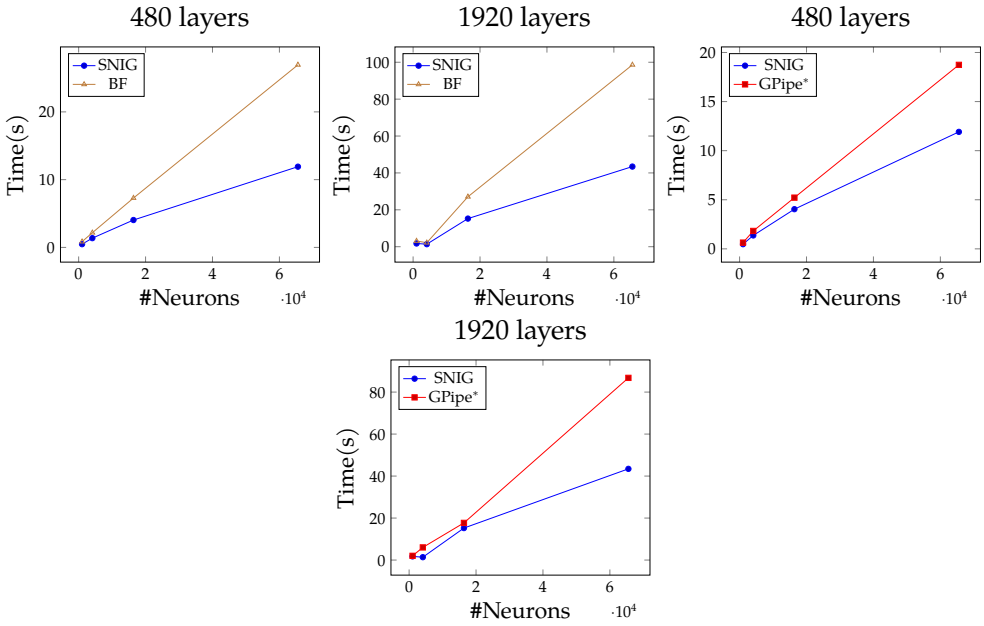


Figure 1.4: Execution time with different neurons under 4 GPUs.

kernel pruning strategy and task parallelism. Figure 1.5 illustrates the peak GPU memory usage of each method. Both SNIG and BF demand less memory than GPipe* because of buffered rolling swap, whereas GPipe* stages the model across GPUs. Our memory is fewer than BF due to batched input data.

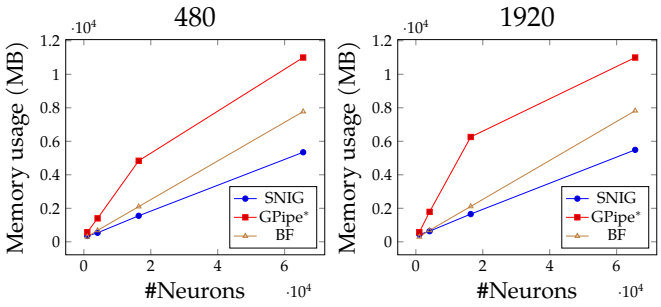


Figure 1.5: Peak GPU memory usage under 4 GPUs.

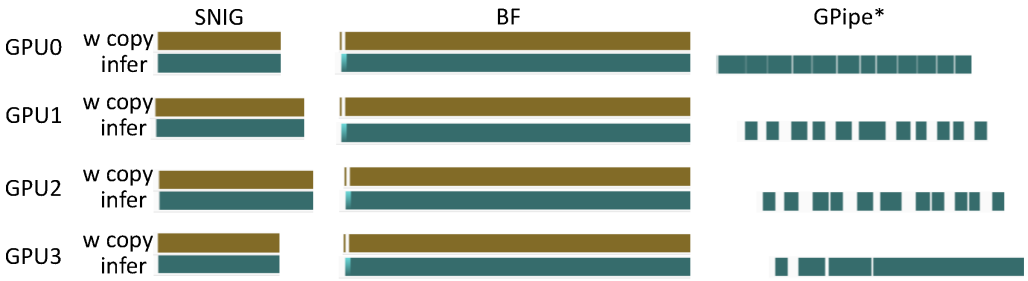


Figure 1.6: Execution timeline of each method on completing 65536 neurons and 1920 layers under 4 GPUs.

Figure 1.6 plots a partial GPU execution timeline of each method using the data extracted from NVIDIA Visual Profiler [56] under the same time scale. Since SNIG and BF do not pipeline the model across GPUs, both methods require weight copy during the inference iterations. However, the time for data transfers is largely overlapped with the kernel computation (i.e., task parallelism in SNIG and stream parallelism in BF). In SNIG, each GPU performs the inference on a data batch independently, and thus the runtime of each GPU is different. The execution timeline of GPipe* at

each GPU is more fragmented and discontinued than SNIG and BF. This is because computation and GPU-to-GPU data transfers at each pipeline level need to synchronize before moving to the next stage. For example, we can clearly see several white spaces between successive GPU operations at GPU 1 and GPU 2.

Parameter Sensitivity

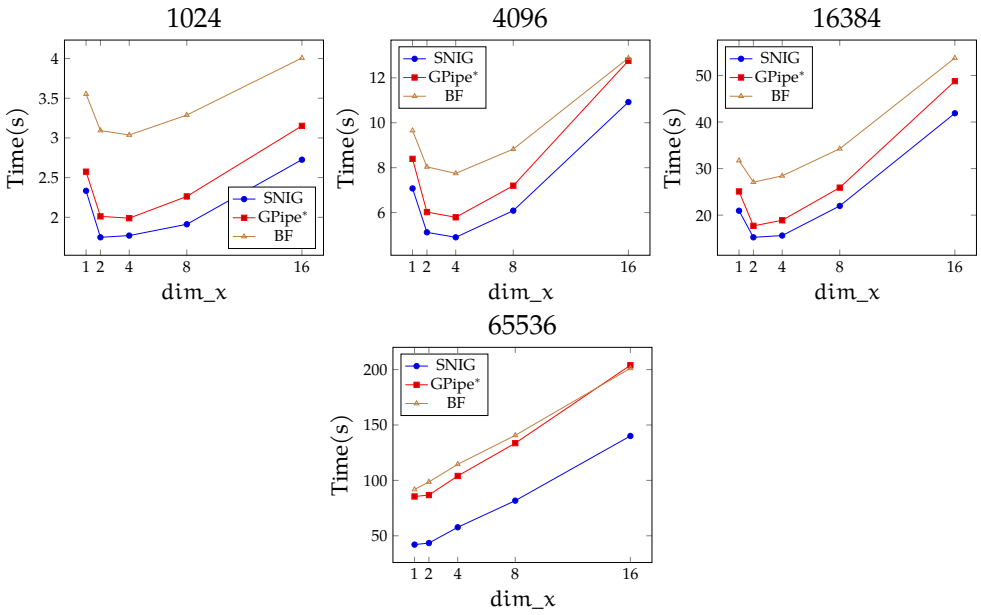


Figure 1.7: Execution time with different block dimensions $(\texttt{dim\_x}, \texttt{dim\_y})$ on 1920 layers under 4 GPUs. The total number of threads $\texttt{dim\_x} \times \texttt{dim\_y}$ remains 1024.

Figure 1.7 shows the impact of different block dimensions. All implementations have the same trend and perform better at lower $\texttt{dim\_x}$, especially under a large number of neurons. All kernels read input data along y dimension and iteratively access weights along x dimension. Since weights are sparse matrices, the overhead is dominated by reading input data. Figure 1.8 shows the impact of different input batch sizes in

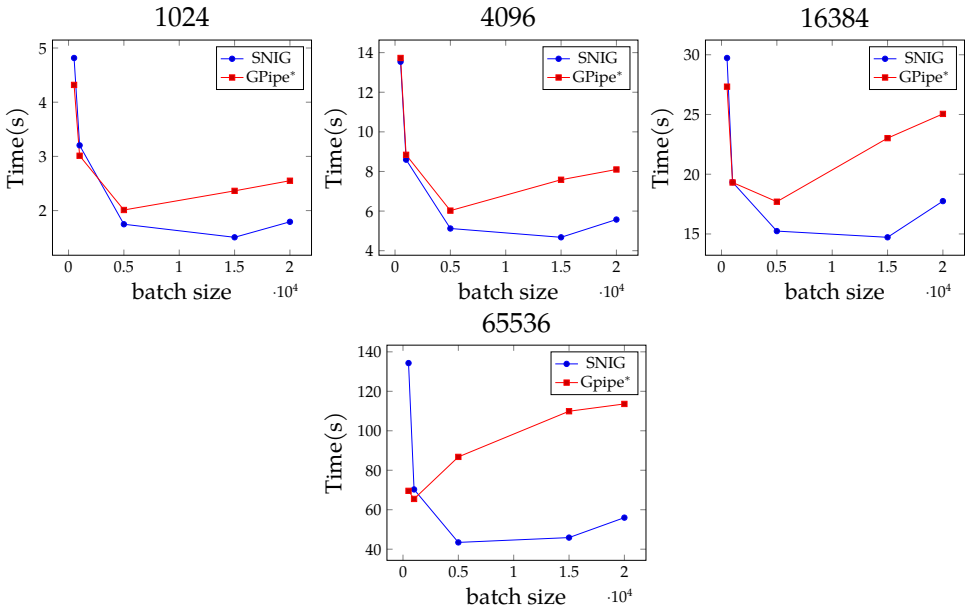


Figure 1.8: Execution time with different batch sizes on 1920 layers under 4 GPUs.

SNIG and GPipe*. Partitioning input data with too small batch size results in a lousy performance, while a bigger batch size doesn't gain speedup. GPipe* has a higher growth rate of runtime than SNIG. We attribute this to the architecture of GPipe* and GPU memory limitation. Since GPipe* pipelines computation across GPUs, large input batch size of large DNNs causes long CPU-GPU and GPU-GPU data communication times. SNIG does not require any GPU-GPU data transfers.

1.7 Conclusion

In this chapter, we have introduced SNIG, an efficient inference engine for large sparse DNNs. We have described the inference workload in a *task graph* comprising both data- and model-level parallelisms. Our decomposition method can scale to arbitrary sizes of DNN and input data

under different numbers of GPUs. Our in-kernel pruning strategy avoids unwanted computation incurred by sparsified network and data, in no need of additional CPU-GPU synchronization to repartition data. With 4 GPUs, SNIG is 2.3× faster than BF and is 2.0× faster than GPipe* on the largest DNN of 65536 neurons and 1920 layers (more than 4 billion nonzero parameters). In this work, Dian-Lun Lin was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the project. All authors contributed to the preparation and review of the final manuscript.

2 CUDAFLOW: EFFICIENT GPU COMPUTATION USING TASK GRAPH PARALLELISM

2.1 Abstract

Recently, CUDA introduces a new task graph programming model, *CUDA graph*, to enable efficient launch and execution of GPU work. Users describe a GPU workload in a task graph rather than aggregated GPU operations, allowing the CUDA runtime to perform whole-graph optimization and significantly reduce the kernel call overheads. However, programming CUDA graphs is extremely challenging. Users need to explicitly construct a graph with verbose parameter settings or implicitly capture a graph that requires complex dependency and concurrency managements using streams and events. To overcome this challenge, we introduce a lightweight task graph programming framework to enable efficient GPU computation using CUDA graph. Users can focus on high-level development of dependent GPU operations, while leaving all the intricate managements of stream concurrency and event dependency to our optimization algorithm. We have evaluated our framework and demonstrated its promising performance on both micro-benchmarks and a large-scale machine learning workload. The result also shows that our optimization algorithm achieves very comparable performance to an optimally-constructed graph and consumes much less GPU resource.

2.2 Introduction

The performance of GPU architectures continues to increase with every new generation. Modern GPUs are fast and, in many scenarios, the time taken by each GPU operation (e.g., kernel or memory copy) is now measured in microseconds. The overheads associated with the submission of

each operation to the GPU, also at the microsecond scale, are becoming significant and can dominate the performance of a GPU algorithm. For instance, inferencing a large neural network launches many dependent kernels on partitioned data and models. If each of these operations is launched to the GPU separately and repetitively, the overheads can combine to form a significant overall degradation to performance. To address this issue, CUDA has recently introduced a new *CUDA graph* programming model to enable efficient launch and execution of GPU work. CUDA graph enables a define-once-run-repeatedly execution flow that reduces the overhead of kernel launching. Users describe dependent GPU operations in a *task graph* rather than aggregated single operations. The CUDA runtime can perform whole-graph optimization and launch the entire graph in a single CPU operation to reduce overheads [57, 58].

However, programming CUDA graphs is extremely challenging. First, users can *explicitly* construct a CUDA graph that maps each vertex to a GPU operation and each edge to a dependency between two GPU operations. Explicit CUDA graph construction is often the most efficient, but it requires all the parameters known upfront, which is impossible for many high-performance third-party libraries, such as cuSparse, cuBLAS, and cuDNN. Also, the CUDA runtime maximally parallelizes the given CUDA graph without limiting the stream usage. In large graphs, the GPU memory can explode. The second option is *implicit* graph construction, which *captures* a CUDA graph using existing stream-based application programming interfaces (APIs). Implicit CUDA graph construction is more flexible and general, allowing users to manually allocate and control streams. However, it requires users to wrangle with concurrency details through events and streams that are known difficult to program correctly.

Consequently, we propose in this chapter a lightweight task graph programming framework to enable efficient GPU computation using CUDA graph. Our framework introduces an *expressive* GPU task graph program-

ming model for users to focus on high-level development of dependent GPU operations with relatively ease of programming. A written task graph is then cast to a native CUDA graph through our transformation algorithm optimized for kernel concurrency and graph size. The process is *transparent*. Users need not to handle any intricate concurrency details and dependency controls using streams and events. More importantly, we identify a research problem of optimizing CUDA graphs through stream capturers. The proposed research can assist CUDA developers in improving the performance of existing GPU applications through new CUDA graph parallelism.

2.3 The Proposed GPU Task Graph Programming Model

Our GPU task graph programming model consists of two parts, *cudaFlow* and *cudaFlowCapturer*, to handle explicit and implicit graph constructions in different use cases.

cudaFlow: Explicit CUDA Graph Construction

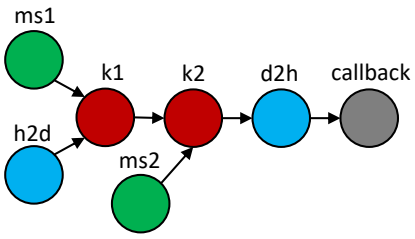


Figure 2.1: An example of GPU task graph.

cudaFlow provides methods for users to explicitly construct a GPU task graph that presents a one-to-one mapping to a native CUDA graph. Each node in the task graph represents a GPU operation (copy, kernel, etc.), and each edge represents a dependency between two operations. Figure 2.1 shows a GPU task graph of seven nodes (two kernels, `k1` and `k2`, two typed copies, `h2d` and `d2h`, two untyped copies, `ms1` and `ms2`, and one host callback,

`callback`) and six dependencies (e.g., `k1`→`k2`). Listing 2.1 gives the implementation of Figure 2.1 using our model. We create a cudaFlow object (`cf`) and use the four methods, `kernel`, `memset`, `copy`, and `host`, to create the seven task graph nodes and use the two methods, `succeed` and `precede`, to relate dependencies between nodes. The code *explains itself* through an expressive graph description language in just 12 lines of code. The same example but written in the plain CUDA graph model is partially shown in Listing 2.2, which requires more than 150 lines of code.

```
cudaFlow cf;
cudaTask h2d = cf.copy(inputVec_d, inputVec_h, inputSize);
cudaTask ms1 = cf.memset(outputVec_d, 0, input_size);
cudaTask ms2 = cf.memset(result_d, 0, 1);
cudaTask k1 = cf.kernel(reduce, inputVec_d, outputVec_d, inputSize);
cudaTask k2 = cf.kernel(reduce_final, outputVec_d, result_d);
cudaTask d2h = cf.copy(result_h, result_d, 1);
cudaTask callback = cf.host(fn, &hostFnData);
k1.succeed(h2d, ms1);
k2.succeed(k1, ms2);
k2.precede(d2h);
d2h.precede(callback);
```

Listing 2.1: Example code of Figure 2.1 using cudaFlow.

```
cudaStream_t streamForGraph;
cudaGraph_t graph;
std::vector<cudaGraphNode_t> nodeDependencies;
cudaGraphNode_t memcpyNode, kernelNode, memsetNode;
checkCudaErrors(cudaStreamCreate(&streamForGraph));
cudaKernelNodeParams kernelNodeParams = {0};
cudaMemcpy3DParms memcpyParams = {0};
cudaMemsetParams memsetParams = {0};
memcpyParams.srcArray = NULL;
memcpyParams.srcPos = make_cudaPos(0, 0, 0);
```

```
memcpyParams.srcPtr =
    make_cudaPitchedPtr(inputVec_h, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.dstArray = NULL;
memcpyParams.dstPos = make_cudaPos(0, 0, 0);
memcpyParams.dstPtr =
    make_cudaPitchedPtr(inputVec_d, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.extent = make_cudaExtent(sizeof(float) * inputSize, 1, 1);
memcpyParams.kind = cudaMemcpyHostToDevice;
checkCudaErrors(cudaGraphCreate(&graph, 0));
checkCudaErrors(
  cudaGraphAddMemcpyNode(&memcpyN, graph, NULL, 0, &memcpyP
));
//... more than 100 lines of code to follow
```

Listing 2.2: Example code of Figure 2.1 using the plain CUDA graph.

cudaFlowCapturer: Implicit CUDA Graph Construction

cudaFlow allows users to explicitly construct a CUDA graph, but it requires all execution parameters known in advance. This property restricts users from using commercial CUDA libraries, such as cuDNN and cuBLAS, that do not provide details for launching kernels but a public stream-based API. To overcome this restriction, we introduce cudaFlowCapturer with a stream-based method to capture GPU kernels and transform the given task graph into a native CUDA graph using our graph transformation algorithm. Listing 2.3 shows the cudaFlowCapturer code of Figure 2.1, assuming the two kernels, `k1` and `k2`, are only invokable through a stream-based API. The cudaFlowCapture provides a method, `on`, that passes a stream created by our optimizer to the callable for users to capturer kernels or other asynchronous GPU operations.

```
cudaFlowCapturer cap;
cudaTask h2d = cap.copy(inputVec_d, inputVec_h, inputSize);
```

```
cudaTask ms1 = cap.memset(outputVec_d, 0, input_size);
cudaTask ms2 = cap.memset(result_d, 0, 1);
cudaTask k1 = cap.on([&](cudaStream_t stream){
  cublas_gemm(stream, my_paremeters...);
});
cudaTask k2 = cap.on([&](cudaStream_t stream){
  cublas_gemv(stream, my_paremeters...);
});
cudaTask d2h = cap.copy(result_h, result_d, 1);
cudaTask callback = cf.host(fn, &hostFnData);
k1.succeed(h2d, ms1);
k2.succeed(k1, ms2);
k2.precede(d2h);
d2h.precede(callback);
```

Listing 2.3: Example code of Figure 2.1 using cudaFlowCapturer.

2.4 Transform a cudaFlowCapturer to a CUDA Graph

By default, we translate a cudaFlow directly into a native CUDA graph and use a single CPU call to offload the graph. To launch a cudaFlowCapturer, we need to transform the task graph defined in the cudaFlowCapturer into a native CUDA graph using stream capturer.

Problem Formulation

We describe the transformation problem as follows: Given a task graph G_t and the number of streams ($num\_streams$), discover an order to construct dependencies between nodes, i.e., assign each node $n \in G_t$ to a stream and decide an event for each node such that the execution

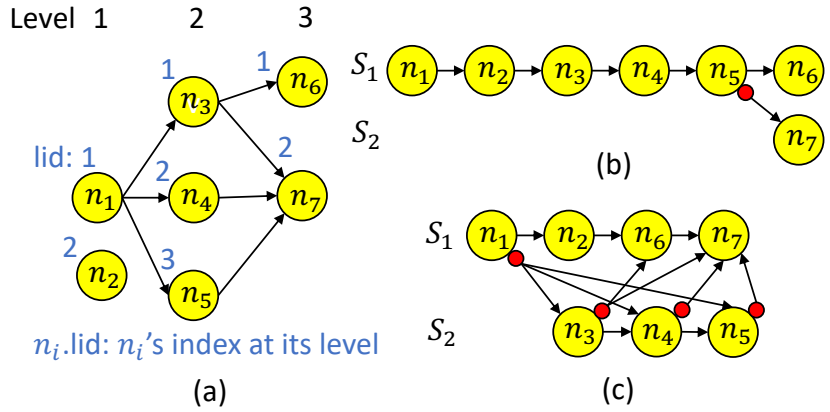


Figure 2.2: Transformation of a task graph to a CUDA graph using two streams.

order of tasks ("transformed CUDA graph") imposed by the streams and events is topologically identical to the original task graph. The objective is to balance the load of each stream and minimize the transformed graph size. For example, using two streams, the task graph in Figure 2.2(a) can be transformed into two different CUDA graphs, (b) and (c), both resulting in different critical paths and graph sizes. CUDA stream is in-order. Placing two dependent nodes at two different streams may require creating an event to build a dependency in the CUDA graph, as shown in the red points. Since the optimal number of streams is highly dependent on application level, we leave `num_streams` to users to tune the number of streams based on their applications.

This transformation problem has two challenges: Firstly, CUDA stream capture is stateful [59]. We can only construct a dependency in *one direction* from an assigned node to the node that is being enqueued to a stream. That is, optimizing the event count and, hence, the graph size, through back-and-forth traversal is not possible. Second, graph size matters. The same task graph can have many feasible transformations (see Figure 2.2). Different transformations result in different execution efficiencies.

Our Algorithm: Round Robin with Dependency Pruning

At a high level, our algorithm assigns each node to a stream in a round-robin fashion and applies a dependency pruning to reduce redundant dependencies. We use Figure 2.3 to illustrate our algorithm transforming the task graph of Figure 2.2(a) to a CUDA graph using two streams. First, we levelize the task graph, G_t, to a 2D level list. Based on the 2D level list, we assign each node n_i to indicate the index of the topological ordering of G_t, and $n_i.lid$ to indicate the index of its level (see Figure 2.2(a)). We assign each node to a stream of id equal to $(n_i.lid+1)\%num\_streams+1$ as a result of the round-robin. For example, n_4 is assigned to stream s_2 (i.e., $(2+1)\%2+1$). Assigning nodes in a round-robin manner at each level facilitates load balancing because nodes are evenly distributed across streams. The motivation of levelization is to implicitly capture dependencies between levels using the same stream. For instance, the dependency between n_1 and n_3 is implicitly captured by s_1.

We iterate each node level by level to perform three steps: *construct dependencies*, *assign stream*, and *decide an event*. At the first level, since n_1 does not have predecessors, we assign it to s_1 (Figure 2.3(a)). We then check if any of n_1's successors (n_3, n_4, n_5) will be assigned to the different stream, s_2. Since n_4 will be assigned to s_2, we need to create an event for n_1 so that the later iteration can wait on it to create a dependency edge (Figure 2.3(b)). At the second level, since n_3 is assigned to the same stream as its predecessor, n_1, we do not create a dependency from n_3 to n_1; but, we create an event for n_3 because its successor n_7 will be assigned to s_2, as shown in Figure 2.3(c). Figure 2.3(d) and (e) show the process of n_4. Since n_4 is assigned to the different stream from it's predecessor, n_1, we need to create a dependency before assigning n_4 to s_2 by waiting on n_1's event. The same procedure continues until we iterate all nodes. Our dependency pruning happens at assigning n_7 to s_2 (Figure 2.3(g) and (h)). n_7's predecessors, n_3 and n_5, are both assigned to s_1. We only construct a

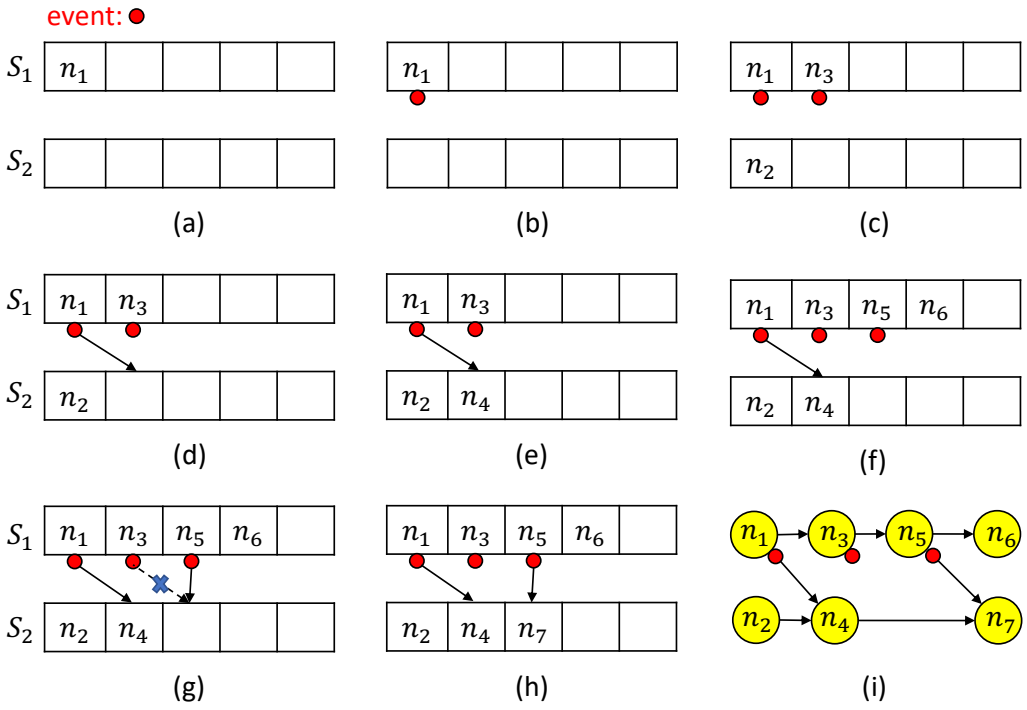


Figure 2.3: Illustration of our algorithm on Figure 2.2 using two streams.

dependency from n_5 to n_7 since n_5 is guaranteed to be executed after n_3 in the same stream, s_1. This pruning reduces redundant dependencies. The transformed CUDA graph from this assignment is shown in Figure 2.3(i).

Algorithm 2 presents the details of our algorithm. We iterate all nodes at each level to perform the three tasks: *construct dependencies*, *assign stream*, and *decide an event*. For simplicity, n.idx represents n_i's index, i.

construct dependencies (lines 6-19): We construct dependencies from n's predecessor, pred, to n. Since n's predecessors may be assigned to the same stream that implicitly capturers sequential order of enqueued nodes, we only need to construct a dependency from the last assigned predecessor, last_assign, to n and prune the other dependencies starting from n's predecessors that is assigned to the same stream.

assign stream (line 20) & decide an event (lines 21-30): We assign

n to s_{sid}, where sid is the id of the stream assigned to n. We decide an event by checking whether n is assigned to a different stream from one of its successors, suc. If true, we create and record an event for n so that suc can construct a dependency from n to suc at the later iteration. We further assign $suc.sm$ to s_{sid} for dependency pruning that happened in the later *construct dependencies* stage.

2.5 Experimental Results

We evaluate the performance of cudaFlow and cudaFlowCapturer on (1) five micro-benchmarks[1] that are representative for many GPU algorithm patterns, and (2) a large-scale machine learning workload directly derived from the 2020 champion of the HPEC Sparse Deep Neural Network (DNN) Inference Challenge [2]. Both cudaFlow and cudaFlowCapturer have different use cases that complement each other. The purpose of our experiment is not to demonstrate which one outperforms another but to highlight that our transformation algorithm can achieve comparable performance (or even better) to the optimally-constructed CUDA graph when explicit graph construction is not possible. By default, we transform a cudaFlow into a CUDA graph of the same topology because all kernel execution parameters are known up-front. In cudaFlowCapturer, we use RR1, RR2, RR4, and RR8 to represent our algorithm using 1, 2, 4, and 8 streams in the round-robin loop, respectively. To demonstrate the effectiveness of our dependency pruning, RR4$^-$ and RR8$^-$ represent our algorithm without dependency pruning under 4 and 8 streams. We do not report RR1$^-$ and RR2$^-$ because redundant dependencies only occur between nodes that are assigned to different streams. Using one or two streams creates few redundant dependencies. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138

[1] source code: https://github.com/dian-lun-lin/cudaFlow-benchmarks

Algorithm 2: Round Robin with Dependency Pruning.

```
   Input: num_streams: number of streams
   Input: graph: task graph defined by users
   /* create streams...                                          */
 1 levelized ← levelize(graph)
 2 for each_level_graph in levelized do
 3     for n in each_level_graph do
 4         sid ← (n.lid + 1)%num_streams + 1
 5         last_assign ← null
 6         for pred in n.predecessors do
 7             psid ← (pred.lid + 1)%num_streams + 1
 8             if s_psid == n.sm then
 9                 if last_assign == null or last_assign.idx < pred.idx
                    then
10                     last_assign = pred
11                 end
12             end
13             else if s_psid != s_sid then
14                 cudaStreamWaitEvent(s_sid, pred.event)
15             end
16         end
17         if last_assign != null then
18             cudaStreamWaitEvent(s_sid, last_assign.event)
19         end
20         n.assign(s_sid)
21         for suc in n.successors do
22             ssid = (suc.lid + 1)%num_streams + 1
23             if s_ssid != s_sid then
24                 if n.event == null then
25                     cudaCreateEvent(n.event)
26                     cudaEventRecord(n.event, s_sid)
27                 end
28                 suc.sm ← s_sid
29             end
30         end
31     end
32 end
```

CPU cores at 2.00 GHz, one GeForce RTX 2080 Ti GPU with 11 GB memory, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 11.1 on a host compiler of GNU GCC-9.2.1 with C++17 standards and optimization flags `-O2` enabled. All data is an average of ten runs.

Micro-benchmarks

We consider five common GPU task graphs as our micro-benchmarks: linear chain (LC), embarrassing parallelism (EP), map-reduce (MR), divide and conquer (DC), and random DAG. LC task graph defines a sequence of sequentially dependent nodes. EP task graph defines only independent nodes. MR task graph defines several iterations each of 16 mappers and one reducer. DC task graph defines a complete binary tree. Random DAG defines a more generalized task graph; we randomly generate up to 50 nodes at each level and create at most five edges per node between successive levels. For all benchmarks, each node contains three sequential GPU operations: host-to-device (H2D) copy, reduction kernel, and device-to-host (D2H) copy. H2D operation first copies 2^{20} integers from CPU to GPU, the reduction kernel performs parallel sum reduction on all elements, and D2H operation copies the reduced sum from GPU to CPU. We focus on large GPU work where the effect of task graph parallelism is significant.

Task graph	cudaFlow	cudaFlowCapturer					
		RR1	RR2	RR4	RR4$^-$	RR8	RR8$^-$
Linear Chain (65536 nodes)	393215	393215	393215	393215	393215	393215	393215
Embarrassing Parallelism (65536 nodes)	327680	393215	393214	393212	393212	393208	393208
Divide and Conquer (16 levels)	393209	393209	425975	442356	442356	450543	450543
Map-Reduce (1024 iterations)	119813	104453	113668	125954	129026	132094	133118
Random DAG (512 levels)	103316	77893	86981	99217	104084	107822	112182
Random DAG (1024 levels)	207552	155437	169574	201875	214088	217454	226009
Random DAG (2048 levels)	410639	311453	347290	403291	423119	437859	447629
Random DAG (4096 levels)	832298	628715	693860	808276	857334	859342	892507

Table 2.1: Comparison of CUDA graph sizes (#nodes+#edges) on linear chain, embarrassing parallelism, divide and conquer, map-reduce, and random DAG task graphs between cudaFlow and cudaFlowCapturer under different stream numbers 1 (RR1), 2 (RR2), 4 (RR4), and 8 (RR8). RR4$^-$ and RR8$^-$ represent our algorithm without the dependency pruning.

Task graph	cudaFlow	cudaFlowCapturer			
		RR1	RR2	RR4	RR8
Linear Chain (65536 nodes)	12	12	13	15	19
Embarrassing Parallelism (65536 nodes)	65547	12	14	18	26
Divide and Conquer (16 levels)	32779	12	14	18	26
Map-Reduce (1024 iterations)	15372	12	14	18	26
Random DAG (512 levels)	5318	12	80	244	559
Random DAG (1024 levels)	10547	12	150	440	1106
Random DAG (2048 levels)	21116	12	263	888	2213
Random DAG (4096 levels)	42545	12	522	1757	4348

Table 2.2: Comparison of the number of streams issued by the CUDA runtime to run each task graph between cudaFlow and cudaFlowCapturer.

Table 2.1 compares the native CUDA graph size (#nodes+#edges) of each benchmark among cudaFlow and cudaFlowCapturer of different stream counts. Apparently, all methods have the same CUDA graph size in the LC task graph. cudaFlowCapturer has a larger CUDA graph size than cudaFlow in the EP task graph, since our algorithm assigns independent nodes to streams that implicitly capture the sequential execution order of enqueued nodes. The same situation happens in the DC task graph, where the number of independent nodes grows exponentially over levels. The CUDA graph size of DC, MR, and random DAG task graphs using cudaFlowCapturer become larger as we increase the number of streams. In our algorithm, more streams can have higher concurrency. However, it may result in more events to implicitly capture the dependencies of the original task graph. Our dependency pruning shows a significant effect on reducing the CUDA graph size in MR and random DAG task graphs. For example, the CUDA graph size on Random DAG with 4096 levels using RR8 is 5.7% smaller than RR8^{-}. This is because MR and random DAG task graphs contain nodes that have more dependencies than others.

Table 2.2 compares the number of streams issued by the CUDA runtime to run each task graph. cudaFlow consumes much larger numbers of streams than cudaFlowCapturer on all benchmarks except the LC graph.

By default, cudaFlow keeps a one-to-one mapping between the task graph and the CUDA graph. The CUDA runtime will issue as many streams as possible to maximize the task concurrency, whereas cudaFlowCapturer transforms the task graph into CUDA graph with a limited number of streams.

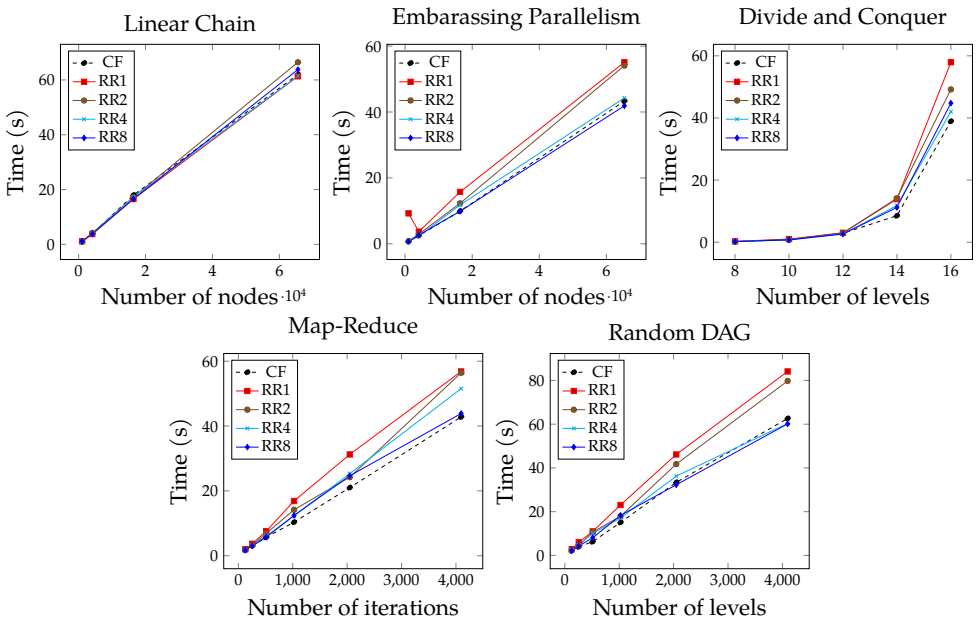


Figure 2.4: Execution time of each task graph at different task graph sizes running on cudaFlow and cudaFlowCapturer of RR1, RR2, RR4, and RR8.

Figure 2.4 shows the execution time (including CUDA graph construction time) of each benchmark. Since LC task graph contains only sequential nodes, all methods have almost the same execution time. RR4, RR8, and cudaFlow are faster than RR1 and RR2 in all other task graphs, because more streams have higher concurrency that leads to faster execution time. Figure 2.5 compares the peak GPU memory usage of each benchmark at different task graph size running on cudaFlow and cudaFlowCapturer. We only compare cudaFlow with RR4 in LC, EP, DC, and MR task graphs since RR1, RR2, RR4, and RR8 have almost the same GPU memory usage

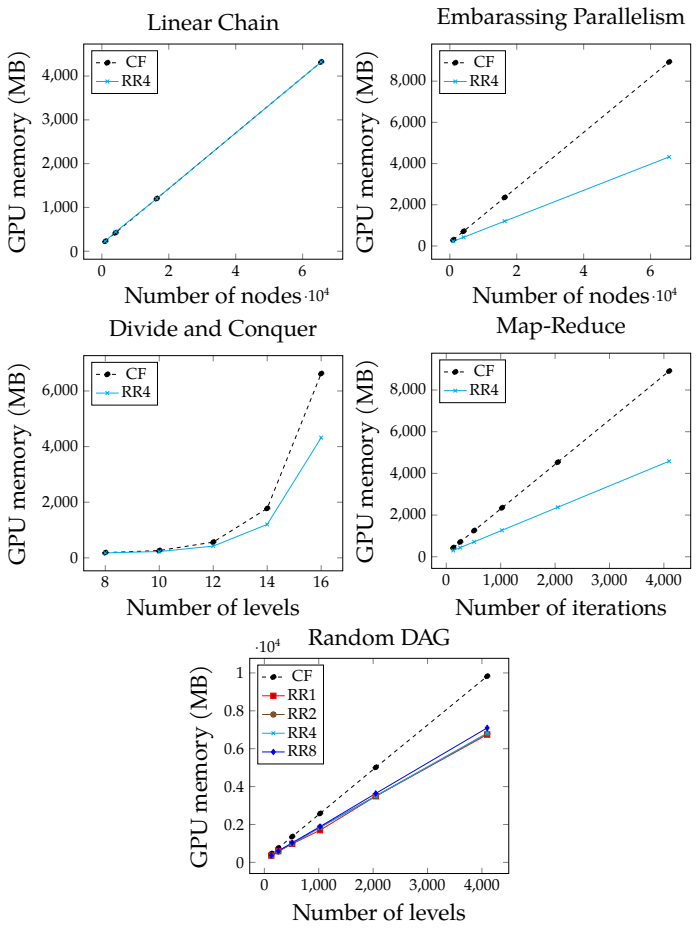
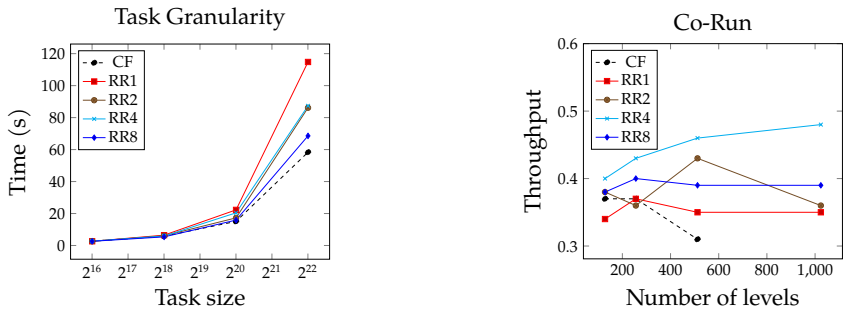


Figure 2.5: Comparison of peak GPU memory usage of each task graph at different task graph sizes between cudaFlow and cudaFlowCapturer.

in these task graphs. The GPU memory usage of cudaFlow is much higher than cudaFlowCapturer on all benchmarks except the LC task graph. In EP task graph, cudaFlow consumes 2.1× more GPU memory than cudaFlowCapturer. This is because the CUDA runtime does not limit the number of streams to run CUDA graphs. Figure 2.6(a) compares the execution time under different task sizes. Task size is the number of elements computed at each node. cudaFlow and RR8 become faster than the others when the task size grows. Compared to lightweight tasks with the same stream



(a) Execution time of random DAG with 1024 levels at different task sizes.

(b) Throughput of corunning random DAG at different task graph sizes.

Figure 2.6: (a) Task granularity and (b) co-run of random DAG running on cudaFlow and cudaFlowCapturer.

count, heavy tasks benefit more from higher kernel concurrency.

Next, we study the throughput of co-running multiple GPU graphs. The motivation is to emulate a server-like environment where multiple client GPU programs run concurrently on the same machine. We consider four co-run processes each executing one random DAG with the same number of levels. The throughput is defined as the execution time of running one process over the execution time of running four processes concurrently [12]. A throughput of 1 implies that the co-run's throughput is the same as if the processes were run consecutively. Figure 2.6(b) compares the throughput of each method. RR4 produces the highest throughput than others, whereas cudaFlow runs out of GPU memory due to unlimited streams.

Machine Learning: Large Sparse Neural Network Inference

The second experiment compares the performance of our transformation algorithm with an optimally-constructed CUDA graph (i.e., cudaFlow) using a large-scale machine learning workload from the IEEE HPEC Graph Challenge 2020. The challenge is to inference extremely large sparse DNN

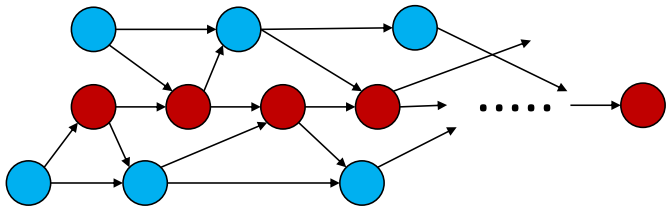


Figure 2.7: [2] describes the inference workload in a task graph. A blue node represents a memory copy, and a red node represents a kernel.

models. We rearchitect the CUDA graph-based champion solution in [2] using cudaFlow and cudaFlowCapturer. We run the experiment on six DNN models composed of different neurons and layers. The statistics of each DNN and its modeled task graph size are summarized in Table 2.3. Figure 2.7 shows a partial task graph of the inference workload.

Neurons/Layers	120	480	1920	Model Size	Image Nonzeros
4096	599	2399	9599	5.40 GB	25,019,051
65536	599	2399	9599	94.70 GB	392,191,985

Table 2.3: The modeled task graph size (#nodes+#edges) and the statistics of each DNN benchmark (model size and image nonzeros).

#Neurons	#Layers	cudaFlow	cudaFlowCapturer			
			RR1	RR2	RR4	RR8
4096	120	1.61	1.34	1.19	1.20	1.19
	480	4.70	4.74	4.19	4.19	4.20
	1920	17.41	19.14	17.08	17.14	17.15
65536	120	14.78	15.99	14.06	14.06	14.05
	480	43.00	50.59	42.92	42.81	42.90
	1920	162.20	193.11	162.12	162.35	162.30

Table 2.4: Comparison of the execution time between cudaFlow and cudaFlowCapturer for completing six DNN models.

#Neurons	#Layers	cudaFlow	cudaFlowCapturer			
			RR1	RR2	RR4	RR8
4096	120	35	23	36	38	42
	480	35	23	36	38	42
	1920	35	23	36	38	42
65536	120	35	23	36	38	42
	480	35	23	36	38	42
	1920	35	23	36	38	42

Table 2.5: Comparison of number of streams issued by the CUDA runtime between cudaFlow and cudaFlowCapturer for completing six DNN models.

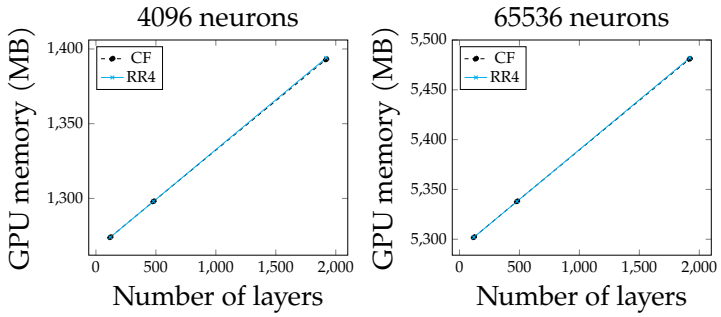


Figure 2.8: Comparison of peak GPU memory usage at different number of layers between cudaFlow and cudaFlowCapturer (RR4).

Table 2.4 compares the execution time (in seconds) of each benchmark using cudaFlow and cudaFlowCapturer at different stream numbers. All methods except RR1 have similar execution time across all DNNs. We observe cudaFlowCapturer of two streams finishes the inference workload with comparable performance of cudaFlow. Using four or eight streams does not decrease the runtime. Table 2.5 compares the number of streams issued by the CUDA runtime. cudaFlow consumes a similar number of streams to cudaFlowCapturer. This is because the maximum degree of concurrency in this particular task graph is around two, and the CUDA

runtime will not consume too many streams to maximize the parallelism. Figure 2.8 compares the peak GPU memory usage at different numbers of layers. Both methods have almost the same peak GPU memory usage due to similar stream usage. This experiment demonstrates the efficiency of our transformation algorithm.

2.6 Conclusion

In this chapter, we have introduced a lightweight task graph programming framework, cudaFlow and cudaFlowCapturer, to enable efficient GPU computation using CUDA graph in different scenarios. In five micro-benchmarks and a real machine learning workload, our transformation algorithm achieves comparable performance to the optimally-constructed CUDA graph and consumes much less GPU resource. The source of our programming model is available in [34]. In this work, Dian-Lun Lin was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the project. All authors participated in discussing the results and contributed to the preparation and review of the final manuscript.

3 RTLFLOW: A GPU ACCELERATION FLOW FOR RTL SIMULATION WITH BATCH STIMULUS

3.1 Abstract

High-throughput RTL simulation is critical for verifying today's highly complex SoCs. Recent research has explored accelerating RTL simulation by leveraging event-driven approaches or partitioning heuristics to speed up simulation on a single stimulus. To further accelerate throughput performance, industry-quality functional verification signoff must explore running multiple stimulus (i.e., batch stimulus) simultaneously, either with directed tests or random inputs. In this chapter, we propose RTLFlow, a GPU-accelerated RTL simulation flow with batch stimulus. RTLflow first transpiles RTL into CUDA kernels that each simulates a partition of the RTL simultaneously across multiple stimulus. It also leverages CUDA Graph and pipeline scheduling for efficient runtime execution. Measuring experimental results on a large industrial design (NVDLA) with 65536 stimulus, we show that RTLflow running on a single A6000 GPU can achieve a 40× runtime speed-up when compared to an 80-thread multi-core CPU baseline.

3.2 Introduction

Register-transfer level (RTL) simulation is a critical part of designing and verifying today's highly complex SoCs, processors, and accelerators [60, 61]. It is widely used in logic design, directed verification, constrained random verification, performance verification, and debugging. For functional verification signoff [62], converging on coverage closure or avoiding bug escape from corner cases typically requires many thousands of nightly regression tests on the same Design-Under-Test (DUT)

with different stimulus, which we refer to as *stimulus-level parallelism*. Different stimulus could be different stimulus outputs from a constrained random testcase generator, or perturbations to directed or random tests with different simulation knobs. As SoC complexity continues to grow, industry-quality functional verification signoff requires a significant and growing amount of compute resource to simulate RTL for dozens of different units within an SoC across many thousands of stimulus daily in the march to tapeout. Speeding up RTL simulation throughput is critical for finding corner case bugs and achieving coverage closure.

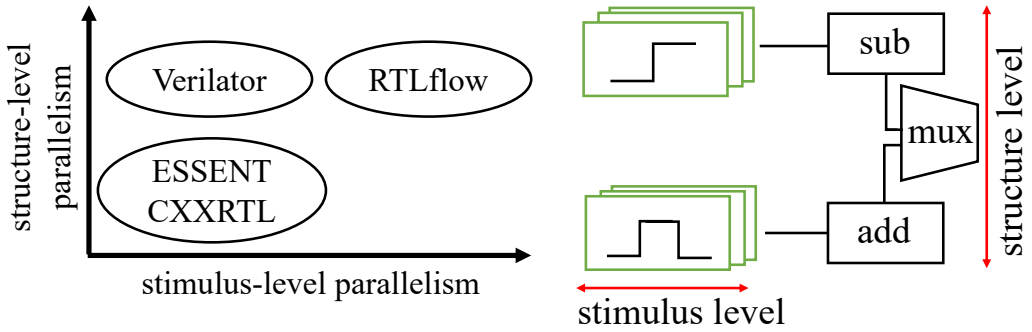


Figure 3.1: RTLflow explores both stimulus- and structure-level parallelisms to achieve high-performance RTL simulation using GPU computing.

In recent years, we have seen increasing interest in accelerating RTL simulation in open-source tools, as shown in Figure 3.1. Verilator [20] is the fastest open-source RTL simulator and has been widely used in both industry and academic design projects. It *transpiles* (source-to-source compile) RTL code into C++ code based on RTL abstract syntax trees (ASTs). Recently, Verilator has explored *structure-level* parallelism via RTL partitioning to support multi-threading. ESSENT [21] is a single-threaded simulator which introduces an event-driven algorithm using conditional execution to skip over unnecessary simulation work. ESSENT has shown up to 2–10× speed-up over single-threaded Verilator but the result does not scale well to large designs due to the lack of multi-threading.

CXXRTL is a Yosys [22] simulation back-end which transpiles an internal representation (IR) generated by the Yosys front-end to C++ simulation code. However, CXXRTL suffers from extremely long compilation time on large designs and does not have any multi-threading capability.

Prior research into accelerating RTL simulation has focused on exploring *strong scaling* of a single stimulus, i.e., reducing time-to-solution for simulating one DUT running one stimulus use case. Multi-stimulus simulation (multiple stimulus running on the same DUT), or *weak scaling*, has been largely ignored in the research because it is commonly done by running multiple instances of single-stimulus simulation on a multi-core CPU system. Modern GPUs support orders of magnitude more parallelism and much higher memory bandwidth than multi-core CPU systems. The large amount of data parallelism exhibited by multiple stimulus provides a unique opportunity to improve the total simulation performance by exploring *stimulus-level* parallelism on modern GPUs.

GPU-based RTL simulation has been investigated before, but also limited to a single stimulus. For instance, [23] offloads the simulation workload to GPU by mapping each RTL process (a group of simulation instructions) to a GPU kernel using one thread per warp. This mapping, however, is not efficient since other threads within a warp are not utilized. [24, 25, 26, 27] use GPU to accelerate gate-level simulation. Nevertheless, gate-level simulation techniques are not suitable for RTL simulation because of different objectives in design flow.

In this chapter, we propose RTLflow, a novel GPU acceleration flow to speed up simultaneous multi-stimulus RTL simulation. As shown in Figure 3.1, RTLflow explores both structure- and stimulus-level parallelisms to achieve high-performance RTL simulation. To the best of our knowledge, this is the first work of GPU-accelerated RTL simulation with multiple stimulus. We summarize three key contributions as follows:

- We introduce an automatic flow to transpile RTL Verilog simulation

code into CUDA equivalents that are optimized for both structure- and stimulus-level parallelisms.

- We introduce a GPU-aware RTL graph partitioning algorithm atop the modern CUDA Graph execution model to explore structure-level parallelism while minimizing kernel call overheads over simulation cycles.
- We introduce a pipeline-based scheduling algorithm that further explores inter-stimulus parallelism to enable efficient resource utilization and computation overlap between CPU and GPU.

We have evaluated RTLflow on industrial designs and demonstrated its promising performance compared to the state-of-the-art Verilator [20] and ESSENT [21]. RTLflow on one A6000 GPU outperforms Verilator and ESSENT on 80 CPU threads with up to 40× speed-up for a Nvidia Deep Learning Accelerator (NVDLA) design [63] with 65536 stimulus. We have conducted detailed experiments to demonstrate performance advantages of our pipeline scheduling, GPU-aware partitioning algorithm, and our CUDA Graph execution strategy that explore various degrees of parallelism compared to the conventional methods. RTLflow is open-source in [4] to benefit the community and inspire software simulation research with heterogeneous parallelism.

3.3 Background and Motivation

RTL simulation represents an input design as a directed graph, namely *RTL graph*. Each node represents a logic element that consumes a set of instructions. Each edge represents a wire to propagate signals between nodes. Simulating a single cycle or a timestamp is an *evaluation* of the graph which consumes inputs and propagates them through logic elements to produce output values. A stimulus provides a sequence of such inputs to

drive simulation. Due to the growing chip sizes, modern RTL simulation requires running many stimulus across different testbenches (e.g., function tests, random tests) to validate the functionality of a design [60].

Conventional RTL Simulation Techniques

RTL simulation typically transpiles the given Verilog to C++ and lets a compiler optimize the simulation code [22, 21, 20]. Listing 3.1 gives an example. The code wraps an input design `dut` with a custom simulator `sim` and simulates the waveforms cycle by cycle. At each cycle iteration, we first set the inputs of `dut` using the given stimulus file. Due to I/O and interaction with external testbench drivers, this step, `set_inputs`, typically runs on CPU and becomes expensive when multiple stimulus exist. We then evaluate the design based on the inputs at rising and falling clocks, `0` and `1`. The iteration continues until the simulator emits a stop signal or completes all simulation cycles.

```
Design dut;
Simulator sim(dut);          // construct a simulator
size_t c = 0;
while(!sim.stop and c <= NUM_CYCLES) {
  dut.set_inputs(c);         // set inputs for the cycle c
  dut.set_clock(0);          // toggle clock to zero
  sim.evaluate();            // evaluate the design
  dut.set_clock(1);          // toggle clock to one
  sim.evaluate();            // evaluate the design
  c = c + 1;                 // move to the next cycle
}
```

Listing 3.1: A transpiled C++ loop for RTL simulation.

Event-Driven and Full-Cycle Simulations

Depending on how values are propagated within a stimulus, simulators can be *event-driven* or *full-cycle*. Event-driven simulators dynamically schedule nodes to perform work only on the active portion of the design. However, managing events requires expensive control-flow costs, making it very difficult to parallelize. Full-cycle simulators instead evaluate the value of every node at every cycle by effectively inlining the entire design and transpiling RTL to straight-line C++ simulation code. The code can be compiled to a highly optimized simulator for the target design. For large designs, simulators can partition the graph and evaluate partitioned subgraphs or tasks in parallel using a static or a dynamic load-balancing scheduler.

Prior Works and their Limitations

Verilator is an open-source full-cycle simulator that has been widely used in both academic and industrial projects due to its absolute speed and robustness [20]. Verilator transpiles input simulation sources (.v) to C++ via AST techniques, applies logical and functional optimizations, and runs simulation for one stimulus on CPU. To further improve the performance, Verilator adopts an iterative partition algorithm [64] to group adjacent nodes into a set of atomic *macro tasks* and models dependent macro tasks in a *task graph* that runs in a multi-threaded environment using a static scheduling algorithm. Verilator defines a *parallelism parameter* (α) to allow fine-tuning the granularity of each macro task.

Despite improved performance, the speed benefit of Verilator has been limited to strong scalability within a stimulus, and the result has largely plateaued at 8–10 CPU cores [20]. To complete the whole simulation workload with batch stimulus, the de facto way is to fork multiple Verilator processes and run independent stimulus in parallel. This organization

is simple but takes no advantage of the large available data parallelism that resides in macro tasks via simulating batch stimulus simultaneously. Specifically, GPU computing provides potential for exploiting this available data parallelism, incorporating high volumes of arithmetic operations. The result can bring significant yet largely untapped performance benefits to various RTL simulation applications, such as functional verification signoff, or design space exploration tasks that count on large numbers of stimulus to validate design choices.

On the other hand, ESSENT adopts an event-driven approach to stop the simulation earlier whenever the activity becomes zero [21]. This approach, however, relies on sophisticated runtime controls and conditionals that are difficult to scale beyond a single thread. The speed-up of ESSENT thus becomes less significant on large designs or simulation workloads with high activities. For example, Verilator of 12 threads can be 5.5× faster than one thread [20], which is far more than the speed-up report of ESSENT in [21]. Moreover, the runtime of ESSENT calls for very frequent dynamic control flow, making it hard to explore massive data parallelism among batch stimulus in a uniform fashion.

Challenges with Batch Stimulus

As the RTL simulation workload continues to increase, in both design size and data size, new simulators must leverage the power of GPU computing to tackle many stimulus simultaneously. To this end, we have identified three major challenges to overcome:

Lack of an Open Infrastructure to Break Language Barrier

RTL speaks a different language from CUDA. It is impractical to ask developers to rewrite every RTL simulation workload to CUDA. While automatic transpilation tools from RTL to C++ are available in the open-source do-

main [22, 20], they cannot be used out of the box for GPUs. The distinct performance characteristics between CPU and GPU require very different settings of memory and data layout transpilation to make the most of GPU computing. An open-source transpilation tool for this purpose will largely fill the gap and inspire broad research efforts in software simulation.

Lack of a GPU-aware RTL Partitioning Algorithm

Existing full-cycle RTL simulators [22, 20] all partition an RTL graph into dependent subgraphs to support multi-threaded CPU parallelism. These partitioning algorithms frequently count on hard-coded parameters to estimate the cost of clustering nodes in terms of CPU instructions. Such an estimate, however, is not reflective of what will happen in a real GPU-based simulation for batch stimulus. For instance, depending on how we schedule batch stimulus to run on a GPU, the generated simulation code (CUDA and C++) and its memory layout can change dramatically after compiler optimization (e.g., nvcc). We need a GPU-aware partitioning algorithm that can perform estimates in real operating conditions. Furthermore, we should notice that partitioned RTL graphs can result in a non-trivial topology of GPU tasks (e.g., kernel, memory copy, operation dependency). As a full-cycle simulator can evaluate many thousands or millions of cycles, launching these dependent GPU tasks can incur significant runtime overheads, such as scheduling streams/events and invoking kernels, that outweigh the performance benefit of GPU computing.

Lack of an Efficient CPU-GPU Task Scheduler

A practical GPU-accelerated RTL simulator with batch stimulus is both CPU- and GPU-intensive. As shown in Listing 3.1, each simulation iteration uses CPU threads to read and set the inputs (`dut.set_inputs`) from an external file or, in our case many stimulus files, before we can offload the evaluation to a GPU (`sim.evaluate`). As we increase the number of stimu-

lus, this sequential computation can incur expensive waiting time between CPU and GPU. Figure 3.2 gives an example of `set_inputs` time and GPU utilization rate at different numbers of stimulus. We can see that the GPU utilization rate drops significantly as the number of stimulus increases, since GPU needs to wait until CPU threads finish setting inputs at each iteration. The CPU-based call to set simulation input, `dut.set_inputs` in Listing 3.1, becomes the primarily bottleneck. To overcome this problem, we need an efficient scheduling method to overlap CPU and GPU tasks across simulation loops.

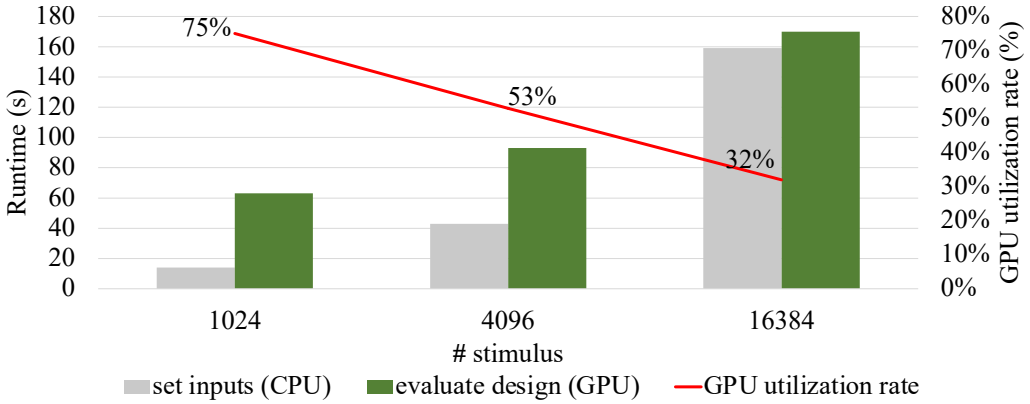


Figure 3.2: Runtime breakdown of a simulation benchmark in terms of setting inputs, evaluating the design, and the corresponding GPU utilization rate under different numbers of stimulus.

3.4 RTLflow

Figure 3.3 shows the overview of RTLflow. At a high level, RTLflow automatically transpiles RTL sources (.v) to C++ and CUDA code to accelerate multi-stimulus simulation on a GPU. Our transpiler is built atop Verilator to inherit its RTL-level optimization facilities, such as inverter pushing, module inlining, and constant propagation that have been rigorously tested for over 25 years in the Verilator community. This decision allows

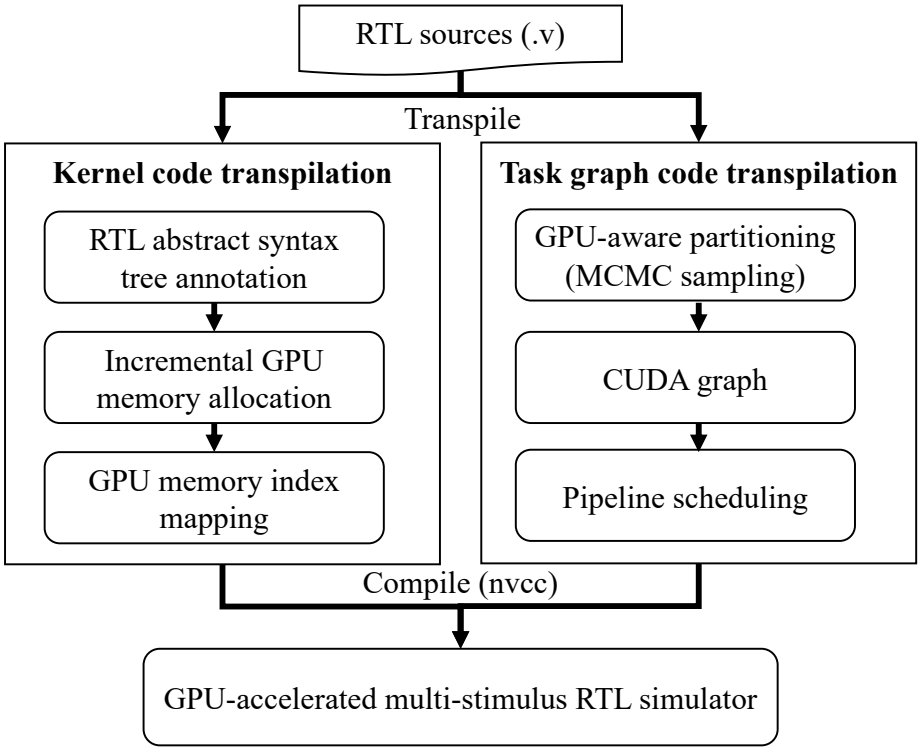


Figure 3.3: Overview of RTLflow.

us to focus on the problem of multi-stimulus simulation itself and to enable future potential integration into Verilator for the benefit of the entire community.

RTLflow consists of two parts, *kernel code transpilation* and *task graph code transpilation*. In kernel code transpilation, we annotate an RTL AST with textual modifications, and transpile the annotated RTL AST into C++ and CUDA using effective GPU memory allocation and mapping algorithms. In task graph code transpilation, we partition the RTL graph into a GPU task graph using a sampling-based algorithm. We execute the GPU task graph using modern CUDA Graph parallelism [54], which is particularly useful for our workload as it largely reduces repetitive kernel call overheads at simulation cycles. To further improve the performance,

we introduce a pipeline-based scheduling algorithm inspired by [65] to explore inter-stimulus parallelism across simulation iterations.

Kernel Code Transpilation

We build our transpilation techniques atop Verilator's RTL AST parser to reuse its I/O infrastructure. However, it is still impossible to transpile Verilog into not only compilable but efficient CUDA code without optimally designed GPU memory management strategies and carefully developed AST-to-CUDA transpiler. For example, one easy way to transpile an RTL AST into CUDA is to traverse the RTL AST and repeatedly allocate GPU memory for a variable (i.e., data signal) as needed. However, this organization induces significant memory allocation overheads and memory fragmentation problems that hamper the performance. To generate optimized CUDA kernels for fast multi-stimulus simulation, the transpiled C++/CUDA code and CUDA kernels must achieve both efficient memory access and minimal memory allocation overheads. To this end, our kernel code transpilation consists of three stages: *AST annotation*, *incremental GPU memory allocation*, and *GPU memory index mapping*. AST annotation annotates each AST node with textual modifications and replaces embedded C++ code with compilable CUDA code. Incremental GPU memory allocation incrementally assigns a GPU memory offset for each variable. GPU memory index mapping transpiles each AST node to CUDA code by mapping each variable to a GPU memory location. In our kernel, each GPU thread is responsible for running the simulation code of one stimulus.

Figure 3.4 shows an RTL AST in Verilator that consists of two modules, `m1` and `m2`. `m1` contains two cells (`c1` and `c2`), two variables (`in` and `sum`), and one function (`func`). A cell is an instance of a module. For example, `c1` is an instance of `m1` that contains two variables, `c1.in` and `c1.sum`. The dotted line between `VARREF` and `VAR` represents that `VARREF` is a variable reference to `VAR`. This RTL AST uses a subtree of seven nodes to describe

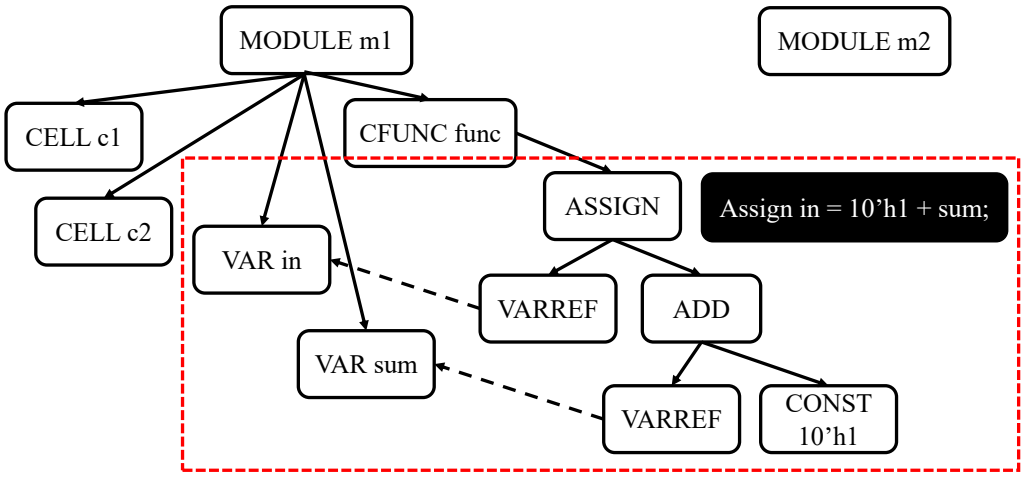


Figure 3.4: An RTL AST that consists of two modules (`m1` and `m2`). `m1` contains two cells (`c1` and `c2`), two variables (`in` and `sum`), and one function (`func`). The RTL AST requires seven AST nodes to describe one line of Verilog code (the assignment statement in black).

one line of assignment code in Verilog. With the AST, Verilator emits C++ simulation code for a single stimulus through a tree traversal algorithm. However, this algorithm cannot directly generate CUDA code because the memory access patterns on GPUs with multiple stimulus are completely different. In the following subsections, we explain three most important strategies that address the transpilation challenge, using Figure 3.4 as an example.

AST Annotation

Manipulating ASTs requires a massive coding effort since we need to carefully take care of each AST node type (more than 300 node types in Verilator) for generating compilable CUDA code. Some AST node types could also include embedded C++ code that does not compile in CUDA. For instance, Figure 3.5 shows the simple and recursive subtrees each rooted at a ARRSEL AST node that is responsible for generating variable

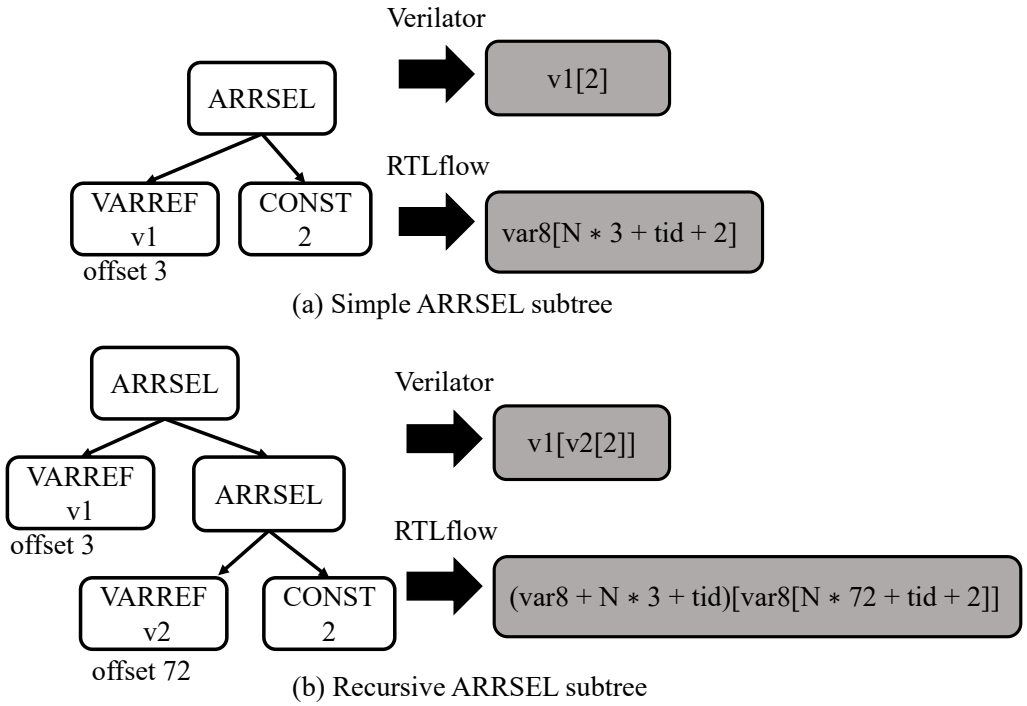


Figure 3.5: (a) Simple ARRSEL subtree and (b) Recursive ARRSEL subtree. Right part shows generated C++/CUDA code using Verilator or RTLflow.

name and index. Verilator transpiles the two subtrees by simply generating `v1[2]` and `v1[v2[2]]`, while RTLflow needs to throughly look into each child node for generating correct syntax (i.e, correct order of "(", ")", "[", and "]"). The correct syntax is annotated at the ARRSEL AST node for later codegen. Another example of AST annotation is adding a keyword (either `__global__` or `__device__`) for functions, as CUDA requires to distinguish whether a function could be called by a host (CPU) or a device (GPU). Since RTLflow partitions an RTL graph into dependent macro tasks that call functions internally, we annotate those macro tasks with `__global__` and others with `__device__`.

Incremental GPU Memory Allocation

Allocating GPU memory for all variables to enable high-performance memory access is challenging, as the width of each variable is largely different. We have researched several strategies to allocate GPU memory for quick memory access, such as dynamic allocated arrays and one fixed-width array. However, none of them can give us a promising performance result during simulation. For example, Figure 3.6 shows an inefficient strategy that uses one fixed-width array of type `uint8_t` to store all variables where `in` is a 6-bit variable and `sum` is a 14-bit variable stored into two memory locations, `sum1` and `sum2`. To load all bits in `sum`, each GPU thread needs to access strided memory twice. This data organization method results in uncoalesced memory access that largely degrades the simulation performance.

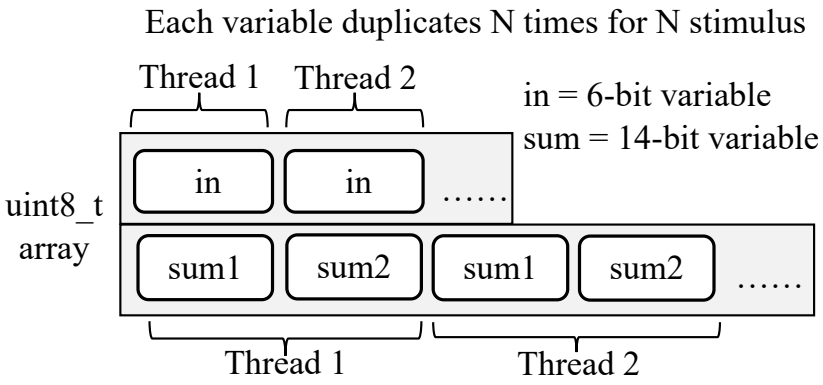


Figure 3.6: GPU memory allocation using one fixed-width memory array of type `uint8_t`. `in` is a 6-bit variable, and `sum` is a 14-bit variable stored into two memory locations, `sum1` and `sum2`.

Our incremental GPU memory allocation overcomes this issue by preallocating four GPU arrays and incrementally assigning each variable a GPU memory offset in reference to the preallocated arrays. The four GPU arrays are `var8`, `var16`, `var32`, and `var64`, each representing a fixed-width variable (`uint8_t` for 8 bits, `uint16_t` for 16 bits, and so on). To minimize

the GPU memory usage, a variable is stored into the smallest of the four types that fits the width of the variable.

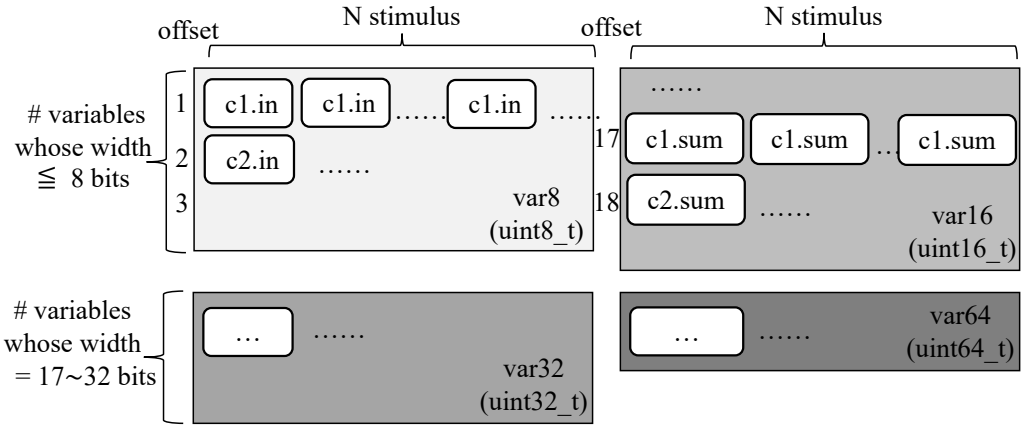


Figure 3.7: GPU memory allocation for Figure 3.4. Each cell (`c1` and `c2`) contains two variables (`in` and `sum`). A variable is stored in the smallest array of types `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` that fits the variable width.

Figure 3.7 shows the memory allocation results of our strategy based on Figure 3.4. Because the width of `sum` is between 9 and 16 bits, we use `uint16_t` to store `c1.sum` and `c2.sum`, similarly for `in` whose width is smaller than eight bits. To handle N stimulus, we duplicate one variable per cell N times in the corresponding array. The size of each array is thus $N \times S_i$, where S_i is the number of variables in array i.

GPU Memory Index Mapping

The goal of GPU memory index mapping is to traverse an RTL AST and use computed GPU memory offsets to emit GPU-efficient CUDA code. Listing 3.2 shows Verilator's transpiled C++ code using the partial RTL AST shown in Figure 3.4. As opposed to Listing 3.2, Listing 3.3 shows the transpiled CUDA code by RTLflow based on the offsets shown in Figure 3.7. To enable efficient GPU memory access, the GPU relies on memory

coalescing to have GPU threads run the same instruction of consecutive memory locations. Since one GPU thread is responsible for a stimulus, we map the GPU memory index of each variable to GPU thread id plus the offset strided by `N`. For instance, the offset of `c1.in` is 1 and thus its index is mapped to `N*1` plus the thread id `tid` that handles the `tid`-th stimulus. The proposed mapping strategy allows all GPU threads to access consecutive GPU memory locations throughout the entire RTL simulation, thus achieving highly coalesced memory access.

```
void m1::c1_func() {
  c1.in = 10h1 + c1.sum;
}
void m1::c2_func() {
  c2.in = 10h1 + c2.sum;
}
```

Listing 3.2: Transpiled C++ simulation pseudocode of Figure 3.4 by Verilator for a single stimulus. A hardware design is allowed to copy a wider value (sum) to a narrower target (in) (truncation will be applied).

```
// RTL simulation code with N stimulus
__device__ void m1::c1_func() {
  tid=blockDim.x*blockIdx.x+threadIdx.x;
  var8[N*1+tid]=          // offset of c1.in is 1
    10h1+var16[N*17+tid]; // offset of c1.sum is 17
}
__device__ void m1::c2_func() {
  tid=blockDim.x*blockIdx.x+threadIdx.x;
  var8[N*2+tid]=          // offset of c2.in is 2
    10h1+var16[N*18+tid]; // offset of c2.sum is 18
}
```

Listing 3.3: Transpiled CUDA kernel pseudocode of Figure 3.4 using the GPU memory offsets in Figure 3.7.

Task Graph Code Transpilation

The goal of task graph code transpilation is to generate efficient execution code using three strategies: 1) GPU-aware partitioning to find a GPU-efficient task graph, 2) CUDA Graph execution to reduce kernel call overheads, and 3) pipeline scheduling to enable efficient CPU-GPU task overlap.

GPU-aware Partitioning

A common RTL graph partitioning algorithm iteratively merges two nodes into a task using static, hard-coded cost estimates [64, 20]. This strategy is simple but is not efficient for RTLflow. Specifically, to maximize the performance of multi-stimulus simulation, we explore several degrees of parallelism (e.g., task graph parallelism and pipeline scheduling) that have dynamic interaction with the CUDA runtime. As a result, we introduce a GPU-aware partitioning algorithm that estimates partition costs in real operating conditions.

Figure 3.8 shows the overview of our algorithm to find a GPU-efficient task graph. We iteratively explore a new weight vector for partitioning (i.e., merging nodes into tasks) using a Markov Chain Monte Carlo (MCMC) sampling algorithm. Our algorithm consists of two components: *estimator* and *optimizer*. The estimator estimates the cost of a proposed GPU task graph by compiling its transpiled code and running it on a GPU. We evaluate the graph with a small number of randomly selected stimulus and cycles, and use the results to predict other combinations. This strategy allows us to discover parameters from a small set of data that is representative for the entire problem. The optimizer iteratively proposes a new graph by randomly and incrementally altering the weight function `weight_sum`

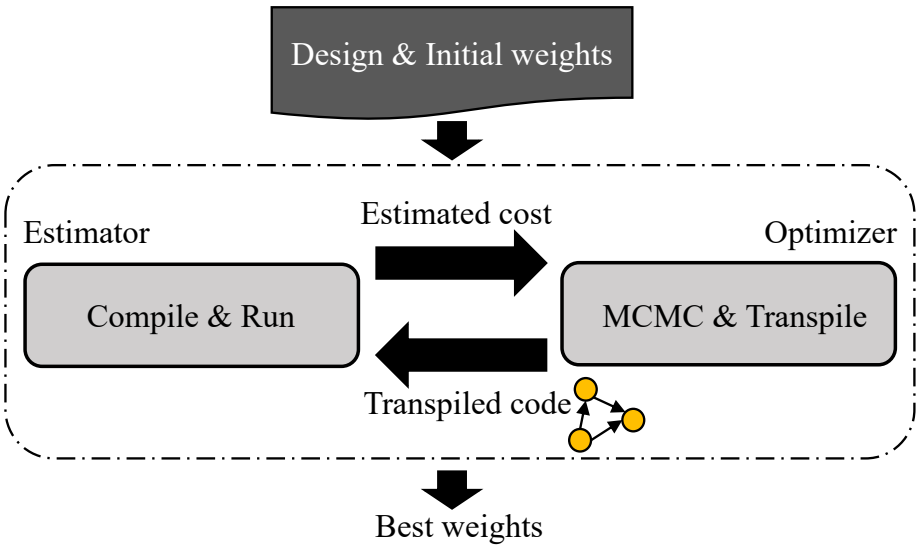


Figure 3.8: GPU-aware partitioning algorithm using MCMC to explore the best combination of weights under real operating conditions (compile + run).

from the previous iteration, defined below:

$$weight\_sum(task) = \sum_{t \in T} w_t * N_t \tag{3.1}$$

where T is the set of top k (e.g., 30) most frequently appeared RTL nodes, w_t is the weight of an RTL node t, and N_t is the number of RTL node t in the given task. Given a merged task, we compute the weighted sum of all RTL nodes in the task and use it to produce a new task graph.

In MCMC sampling, we obtain samples from a probability distribution so that a GPU task graph of faster runtime is visited more often than the slower ones [66, 67]. The probability distribution is the following:

$$p(\mathcal{G}) \propto exp(-\beta * cost(\mathcal{G})) \tag{3.2}$$

where $\mathcal{G}$ is a GPU task graph, $cost(\mathcal{G})$ is the estimated cost of $\mathcal{G}$ from the esti-

mator, and β is a constant that can be chosen. We use Metropolis-Hastings algorithm [67] to generate Markov chains, which keeps the current GPU task graph and proposes a new one $\mathcal{G}^*$. If $\mathcal{G}^*$ is accepted, it replaces the current graph; otherwise we propose another GPU task graph based on $\mathcal{G}$ again. The acceptance rate of a new GPU task graph is the following:

$$\begin{aligned} \alpha(\mathcal{G} \to \mathcal{G}^*) &= \mathrm{min}(1, p(\mathcal{G}^*)/p(\mathcal{G})) \\ &= \mathrm{min}(1, exp(\beta * (cost(\mathcal{G}) - cost(\mathcal{G}^*)))) \end{aligned} \tag{3.3}$$

where $\mathcal{G}^*$ with a lower cost than $\mathcal{G}$ is always accepted, and $\mathcal{G}^*$ with a higher cost than $\mathcal{G}$ may still be accepted with a probability depending on difference between $cost(\mathcal{G})$ and $cost(\mathcal{G}^*)$.

Algorithm 3 shows the pseudocode of our proposed algorithm. At the beginning, we initialize the weight of each RTL node to one (line 5). During the sampling process, the optimizer randomly increases one weight from the current weights (line 7). It then proposes a new GPU task graph in terms of new weights (line 8). The estimator evaluates the proposed graph with the given number of stimulus and cycles and returns an estimated cost (line 9). If the current estimated cost is larger than the new one, the optimizer accepts the new weights and updates the current cost (line 10-14). If not, we generate a random number from 0 to 1 to determine if we accept the proposed graph (line 16-20). The iteration continues until we cannot find a better graph for a maximum number of iterations.

CUDA Graph Execution Model

After obtaining a partitioned GPU task graph, we need to offload it to a GPU. Traditionally, this is done by creating multiple CUDA streams and events to dynamically schedule tasks and manage their dependencies. However, this paradigm will incur significant runtime overheads because

Algorithm 3: GPU-aware partitioning algorithm

```
   Input: dut: a design under test
   Input: MAX_ITER: maximum #iterations
   Input: MAX_UNIMPROVED: maximum #unimproved iterations
 1 cur_cost ← ∞
 2 iter, cnt ← 0
 3 Optimizer opt(dut)
 4 Estimator est(dut)
 5 opt.initialize_weights()
 6 while cnt < MAX_UNIMPROVED and iter++ < MAX_ITER
 7     opt.random_increase()
 8     graph ← opt.propose()
 9     cost ← est.estimate_cost(graph)
10     if cur_cost > cost then
11         opt.update_weights()
12         cur_cost ← cost
13         cnt ← 0
14     end
15     else
16         rand ← uniform_distribution(0, 1)
17         if accept_rate(cost, cur_cost) > rand then
18             opt.update_weights()
19             cur_cost ← cost
20         end
21         cnt++
22     end
23 end
```

it repeats the same stream and event management on the same CUDA task graph over all simulation cycles. The new CUDA Graph execution model [54] is particularly useful for solving this problem via a *define-once-run-repeatedly* CUDA graph. The CUDA runtime can perform whole-graph optimizations that are nearly impossible to achieve by the stream-based approach. Figure 3.9 and 3.10 illustrate the performance advantage of CUDA Graph compared to a stream-based execution diagram that evaluates a GPU task graph. As we can observe, the stream-based execution

incurs multiple CUDA call overheads (e.g., launching kernels through streams, creating event dependencies) within a cycle, and these overheads accumulate across cycles. Such overheads can be eliminated by launching a predefined CUDA graph to improve performance.

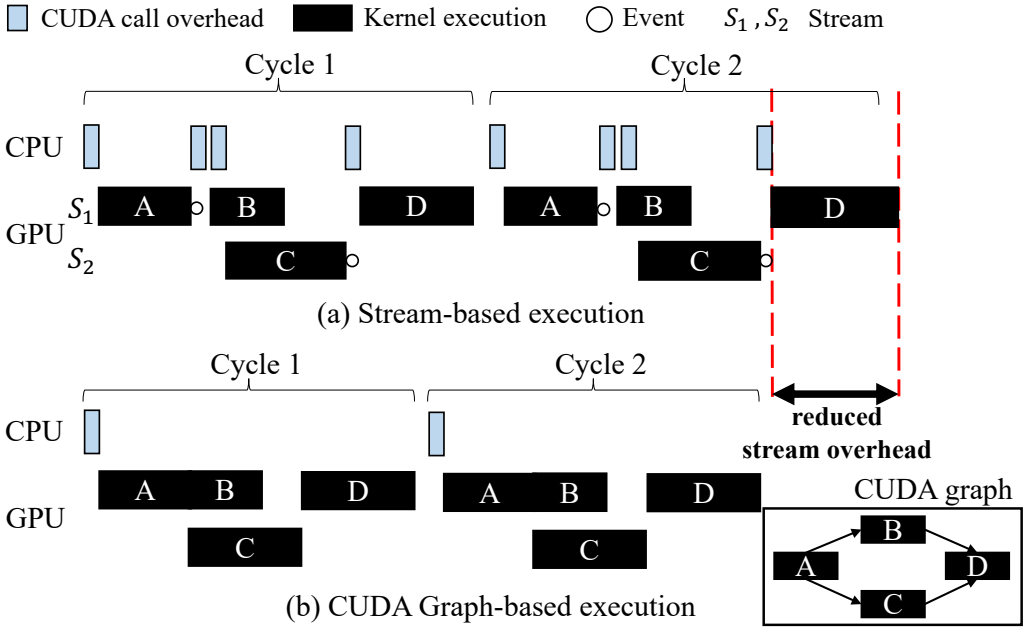


Figure 3.9: Stream-based execution versus CUDA Graph-based execution of the CUDA graph for two cycles. Stream-based execution incurs repetitive CUDA call overheads to schedule dependent kernels at each cycle.

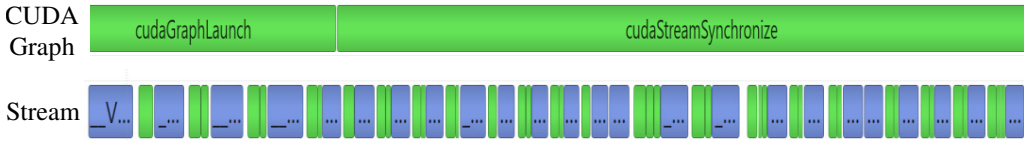


Figure 3.10: Partial simulation timeline of CUDA Graph-based execution and stream-based execution using the data extracted from Nvidia Nsight Systems [3]. Blue bars represent calls to launch CUDA kernels and green bars represent CUDA synchronization calls.

Pipeline Scheduling Algorithm

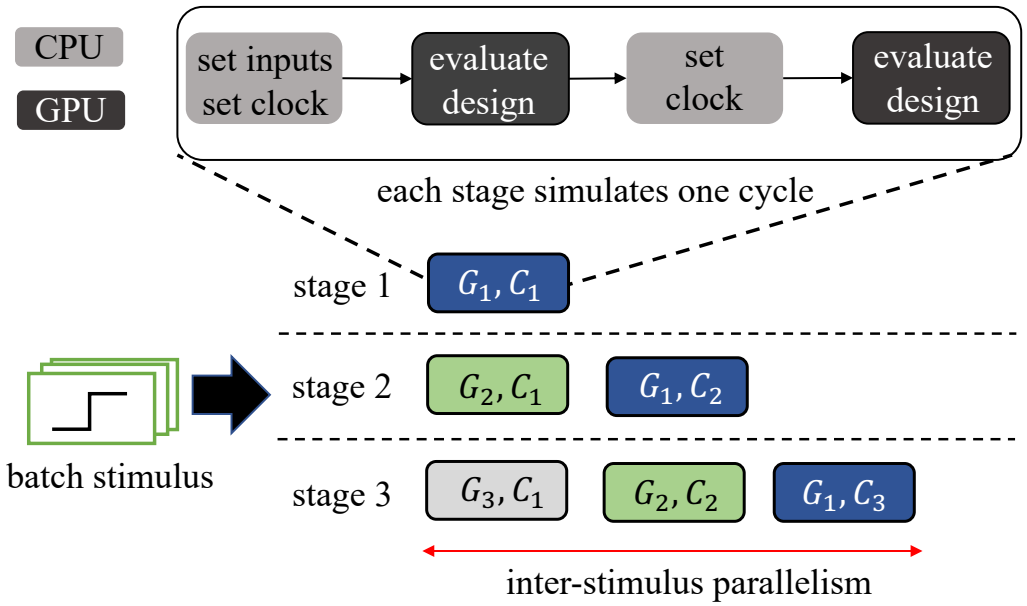


Figure 3.11: The proposed pipeline scheduling algorithm to enable efficient overlap between CPU and GPU tasks.

As shown in Listing 3.1 and Figure 3.2, multi-stimulus RTL simulation incurs significant overheads in setting the inputs, which in turn causes the GPU to wait. To overcome this problem, we further partition batch stimulus into groups and use a pipeline scheduling algorithm to overlap CPU and GPU tasks both inside and outside partitioned stimulus groups.

Figure 3.11 shows the overview of our pipeline scheduling algorithm. We partition batch stimulus into groups that each group, G_i, can be concurrently simulated in a stage. At stage 1, we pass G_1 into our pipeline and simulate it at the first cycle, C_1. Simulating one cycle consists of four dependent CPU/GPU tasks shown in Figure 3.11. At stage 2, we pass G_2 into our pipeline. We then simulate G_1 at C_2 and G_2 at C_1 in parallel. Since our pipeline scheduling does not construct a dependency between groups, tasks in G_1 and tasks in G_2 can be overlapped. For instance, we can execute

Design	Verilog LOC	#AST nodes	Verilator LOC	Verilator T_{trans}	RTLflow LOC	RTLflow T_{trans}
riscv-mini	3306	25224	10640	$<$ 1s	10935	$<$ 1s
Spinal	6858	22888	8429	$<$ 1s	9654	$<$ 1s
NVDLA	511955	1476991	397536	30s	560412	33s

Table 3.1: Statistics of the benchmarks and results of transpiled code for Verilator and RTLflow. The results present lines of code (LOC) and transpilation time (T_{trans}).

`set_inputs` in G_1 and `evaluate_design` in G_2 simultaneously. Specifically, a GPU only needs to wait for CPU threads to finish `set_inputs` for a group, hence overlapping computation between CPU and GPU tasks. Also, since we can offload multiple `evaluation_design` to a GPU at a time, overlaps of `evaluation_design` across different groups can further increase GPU utilization rate.

Note that RTLflow simulates multiple stimulus in parallel, different memory coalescing patterns can occur in terms of the number of stimulus. However, our transpliation can easily handle different coalescing patterns through parameterization. For example, our pipeline scheduling algorithm partitions all stimulus into groups such that all stimulus within a group is simultaneously simulated on a GPU. We can ensure memory access is mostly coalesced by setting the proper group size (e.g., 256 or 1024 stimulus per group).

3.5 Experimental Results

We evaluate RTLflow's performance on three industrial designs, *NVDLA*, *Spinal*, and *riscv-mini*. NVDLA is Nvidia's open-source project of deep learning accelerator [63]. riscv-mini and Spinal are both RISC-V CPU projects [68, 69]. Table 3.1 lists the statistics of each design. All projects have scripts that allow us to generate multiple stimulus with different

configurations. We implement RTLflow using C++17 and CUDA 11.6, and compile RTLflow using nvcc on a host compiler of GCC-8 with optimization `-O2` enabled. We did not observe much performance difference between -O2 and -O3, but -O2 makes the compilation time faster (~3 minutes for -O2 and ~10 minutes for -O3). We use Taskflow [12, 70] and its work-stealing runtime [71] to implement our pipeline scheduling algorithm. We run Verilator and ESSENT on a powerful CPU server and RTLflow on a GPU desktop, described below:

- Machine 1 - a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores (80 CPU threads) at 2.00 GHz and 256 GB RAM
- Machine 2 - a CentOS 8 x86 64-bit machine with 8 Intel i7-11700 CPU cores (16 CPU threads) at 2.5 GHz, one RTX A6000 48 GB GPU, and 128 GB RAM

Note that typical RTL simulation workloads do not involve any floating-point operations, i.e., all computations of RTLflow are integers. We do not leverage any single-precision optimizations to accelerate throughput performance.

Baseline

We consider Verilator and ESSENT as our CPU baselines to measure the performance of RTLflow on simulating batch stimulus. To emulate existing RTL simulation methods for batch stimulus, we fork 80 processes of ESSENT (single-threaded simulator) to run 80 stimulus in parallel, and ten processes of Verilator (multi-threaded simulator) to run ten stimulus in parallel and spawn eight threads per process to run each stimulus. For NVDLA, we observe setting the parallelism parameter (α) to eight in Verilator's RTL graph partitioning algorithm achieves the best performance; for Spinal and riscv-mini, we set α to two for Verilator, and fork

40 processes of Verilator to run 40 stimulus in parallel to achieve the best performance. Compared with NVDLA, Spinal and riscv-mini are smaller designs and do not benefit as much from the partitioning.

Transpilation Results

Table 3.1 shows the benchmark statistics and the complexity of each transpiled code using Verilator and RTLflow. Taking NVDLA for example, RTLflow transpiles 511K lines of RTL to 560K lines of CUDA and C++ simulation code in about 30 seconds. For large designs like NVDLA, it is impractical for developers to rewrite all RTL code to CUDA manually. Our transpiler is fully automatic, and the generated CUDA code can be used out of the box for engineering and research purposes. Without RTLflow, it becomes very difficult for simulation engineers to harness the power of GPU computing using minimal programming effort.

		#cycles								
			10K			100K			500K	
Design	#stimulus	Verilator	RTLflow	Speed-up	Verilator	RTLflow	Speed-up	Verilator	RTLflow	Speed-up
Spinal	256	1s	1s	1×	14s	10s	**1.4**×	1m3s	48s	**1.3**×
	1024	6s	1s	**6**×	52s	10s	**5.2**×	4m2s	50s	**4.8**×
	4096	23s	2s	**11.5**×	3m25s	14s	**14.6**×	15m50s	1m12s	**13.2**×
	16384	1m30s	4s	**22.5**×	13m39s	21s	**39.0**×	1h3m50s	1m37s	**39.5**×
	65536	4m32s	16s	**17.0**×	52m18s	1m12s	**43.6**×	4h10m40s	5m22s	**46.7**×
NVDLA	256	1m2s	1m10s	0.89×	3m48s	8m46s	0.43×	15m16s	41m37s	0.37×
	1024	3m58s	1m29s	**2.7**×	14m39s	10m56s	**1.3**×	1h31m31s	53m1s	**1.7**×
	4096	21m50s	1m46s	**12.4**×	57m52s	13m11s	**4.4**×	4h1m17s	1h2m13s	**3.9**×
	16384	1h22m47s	2m44s	**30.3**×	6h37m50s	18m18s	**21.7**×	22h16m38s	1h24m5s	**15.9**×
	65536	5h31m14s	8m8s	**40.7**×	26h31m52s	49m18s	**32.3**×	89h16m22s	3h45m10s	**23.8**×

Table 3.2: Comparison of elapsed simulation times between Verilator (with 80 CPU threads) and RTLflow (with one A6000 GPU) on Spinal and NVDLA for completing 256, 1024, 4096, 16384, and 65536 stimulus at 10K, 100K, and 500K clock cycles. All signal outputs match the golden reference generated by Verilator.

Overall Performance Comparison

Table 3.2 compares the elapsed simulation times between Verilator (with 80 CPU threads) and RTLflow (with one A6000 GPU) on Spinal and NVDLA. RTLflow outperforms Verilator using 80 CPU threads in almost all scenarios. With 65536 stimulus, RTLflow is 46.7× faster on Spinal at 500K cycles and is 40.7× faster on NVDLA at 10K cycles. We can clearly see the proposed GPU acceleration flow brings significant performance benefits to simulate multiple stimulus simultaneously. Figure 3.12 shows runtime comparisons across different hardware platforms for NVDLA with 16384 stimulus at 10K cycles. Compared to single-threaded Verilator, RTLflow achieves 523× speed-up using one A6000 GPU. The significant performance improvement demonstrates the promise of our multi-stimulus simulation techniques.

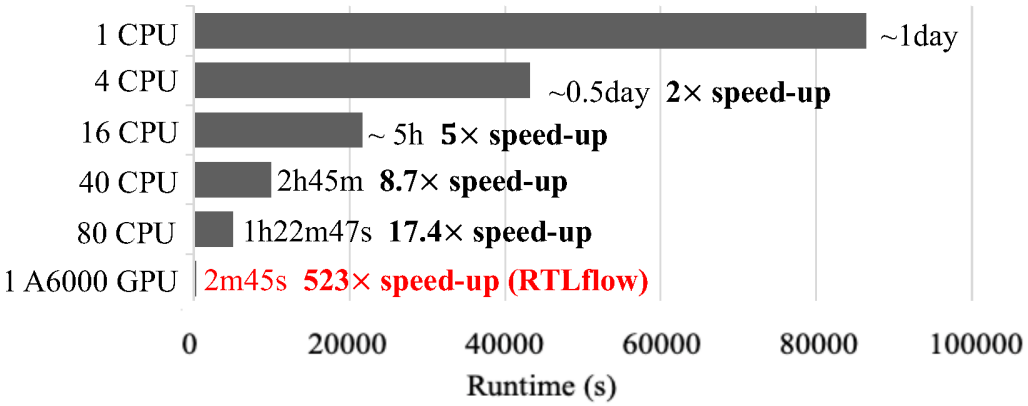


Figure 3.12: Runtime comparisons across different hardware platforms for NVDLA with 16384 stimulus and 10K cycles.

Figure 3.13 shows the runtime growth over increasing numbers of stimulus for Verilator, ESSENT, and RTLflow on riscv-mini with 10K cycles. When the number of stimulus is smaller than 1024, all simulators are able to finish simulation in five seconds, and the advantage of GPU is not pronounced compared to others with 80 threads. When the number of

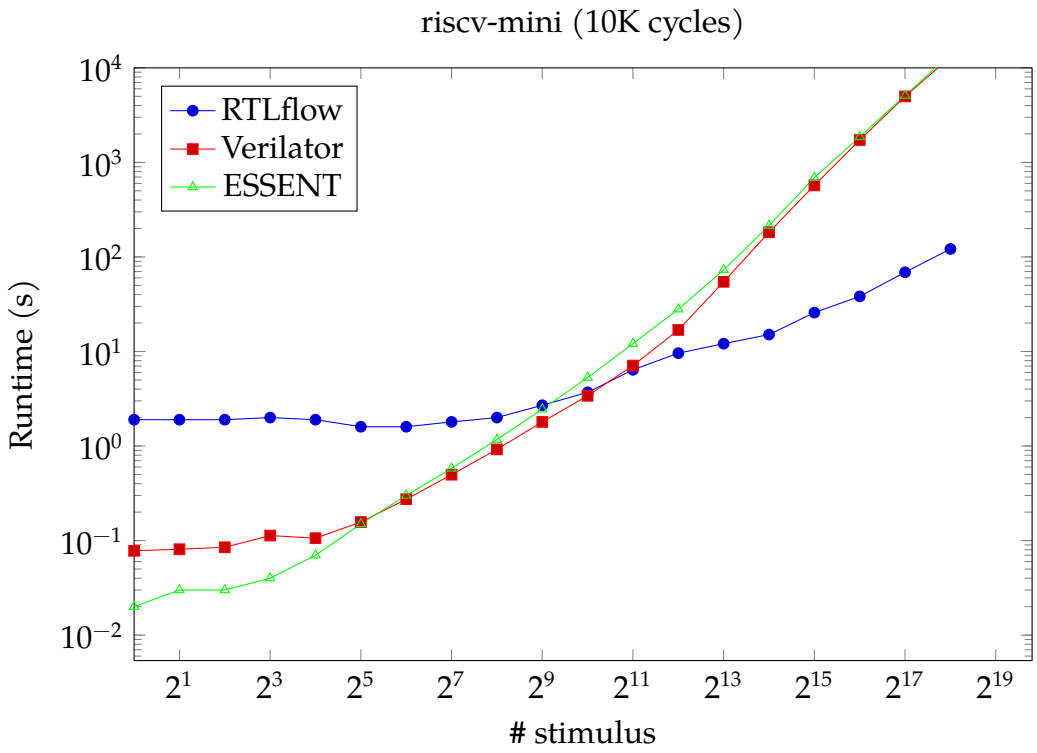


Figure 3.13: Runtime growth over increasing number of stimulus for Verilator, ESSENT, and RTLflow on riscv-mini.

stimulus is larger than 1024, in which data parallelism becomes large, RTLflow starts to scale better than Verilator and ESSENT. For instance, when increasing the number of stimulus from 4096 to 65536, the runtime of RTLflow grows 4× whereas Verilator and ESSENT grow 102× and 66×, respectively.

Absolute efficiency can be measured by the latency needed to complete batch stimulus. Table 3.2 and Figure 3.13 provide some insight: When the number of stimulus is small (e.g., <256), RTLflow does not benefit from much data parallelism and thus CPU-based Verilator is better. However, industrial simulators can easily call many thousands of stimulus where RTLflow (GPU) wins out. The break-even points can be observed in

Table 3.2 for Spinal and NVDLA (256 and 1024 stimulus, respectively). Similar number is also observed in Figure 3.13 for riscv-mini.

Performance Result of GPU Task Graph

Table 3.3 compares the runtime between RTLflow with and without GPU-aware partitioning algorithm (RTLflow^{-g}). For RTLflow, we obtain `weight_sum` by running 150 MCMC sampling iterations where each iteration evaluates the candidate partition (compile + run) using 256 stimulus and 3K cycles. We do not observe much difference beyond this number. Then, we use the weight vector to run different scenarios of cycle and stimulus combinations. For RTLflow^{-g}, we use the default partitioning algorithm in Verilator that hard codes weights [20]. We can clearly see the performance advantage of our GPU-aware partitioning algorithm. RTLflow speeds up RTLflow^{-g} in all scenarios with up to 5.8%. Our algorithm generates a better partitioned GPU task graph by performing estimates in real operating conditions. The result also highlights that our algorithm achieves predictable performance for different cycle and stimulus numbers.

	4096 stimulus		16384 stimulus	
#Cycles	RTLflow^{-g}	RTLflow	RTLflow^{-g}	RTLflow
10K	110.3s	106.8s (↑**3.3%**)	170.1s	163.5s (↑**4%**)
50K	428.9s	405.4s (↑**5.8%**)	611.9s	587.3s (↑**4.2%**)
100K	813.1s	791.0s (↑**2.8%**)	1145.2s	1098.2s (↑**4.3%**)

Table 3.3: Runtime comparison in terms of improvement (↑) between RTLflow with and without GPU-aware partitioning algorithm (RTLflow^{-g}) for NVDLA with 4096 and 16384 stimulus at 10K, 50K, 100K cycles.

Figure 3.14 shows partial RTL task graphs partitioned for Spinal with and without our GPU-aware partitioning algorithm. Based on our observation, our algorithm attempts to find a partition of many parallel tasks,

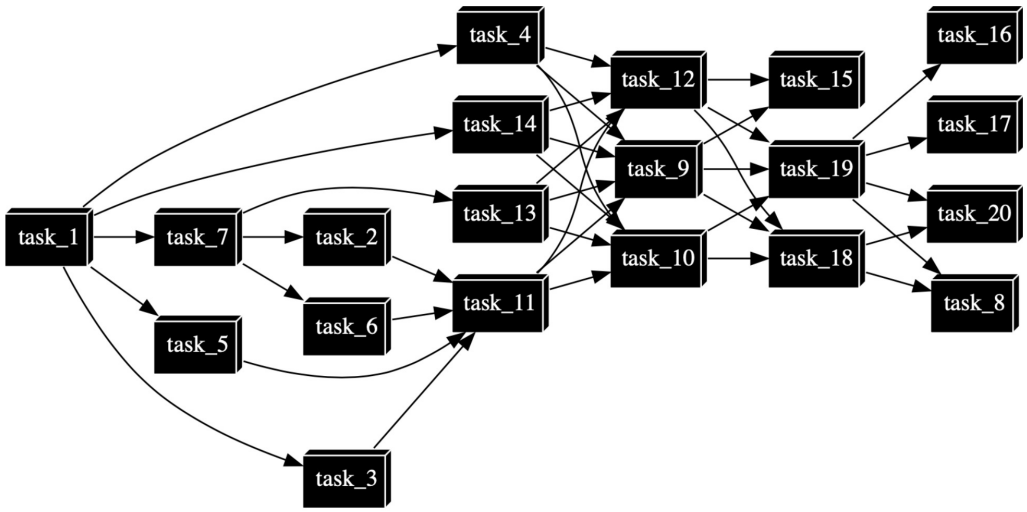
which in turn maximizes the kernel concurrency of the induced CUDA graph. For instance, the task graph in (b) implies many concurrent kernels at a specific level (e.g., `task_B`, `task_D`, `task_G`, `task_C`, `task_E`, `task_F`) that results in a better performance of CUDA Graph execution than (a).

To further demonstrate the effectiveness of CUDA Graph, we implement a stream-based execution algorithm to execute the CUDA graph. Specifically, we implement the state-of-the-art CUDA Graph transformation algorithm [72] to capture a CUDA graph using streams and events while maximizing the kernel concurrency. We use four streams to capture the CUDA graph which achieves the best performance on our A6000 GPU.

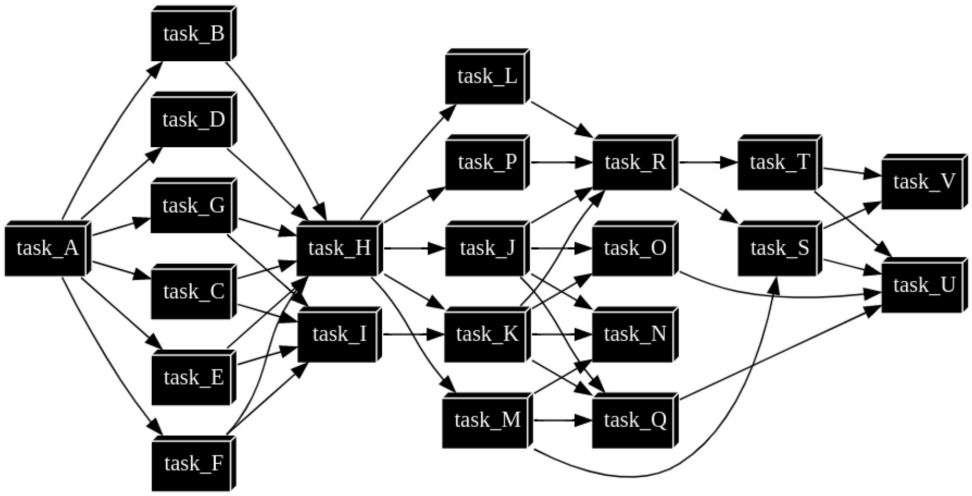
	Spinal		NVDLA	
#Cycles	stream	CUDA Graph	stream	CUDA Graph
10K	11.5s	2.3s (**5**×)	279.8s	106.5s (**2.6**×)
100K	108.0s	14.2s (**7.6**×)	2046.9s	791.2s (**2.6**×)
500K	532.9s	72.3s (**7.4**×)	9718.0s	3733.0s (**2.6**×)

Table 3.4: Performance advantage of CUDA Graph execution in multi-stimulus simulation workloads, measured on Spinal and NVDLA with 4096 stimulus under different numbers of cycles.

Table 3.4 shows performance advantage of CUDA Graph execution in multi-stimulus simulation workloads. We can clearly see the advantage of CUDA Graph. The CUDA Graph-based approach outperforms the stream-based counterpart in all scenarios. For instance, CUDA Graph reaches the goal 7.4× and 2.6× faster than stream with 500K cycles for Spinal and NVDLA, respectively. Compared with the stream-based approach, CUDA Graph launches all dependent GPU tasks in the CUDA graph through a single CPU call per cycle, thus largely reducing the kernel call overheads. Also, the CUDA runtime can perform whole-graph optimizations to schedule a CUDA graph without repetitively launching streams and events to build up the dependency graph that is consistent across all cycles.



(a) GPU-oblivious task graph partition



(b) GPU-aware task graph partition

Figure 3.14: Partial RTL task graphs for Spinal with and without our GPU-aware partitioning algorithm. Each task is a GPU kernel that evaluates the design with batch stimulus.

Performance Result of Pipeline Scheduling

In this section, we study the performance benefit of our pipeline scheduling. Table 3.5 compares the runtime between RTLflow with and without

#Stimulus	Spinal		NVDLA	
	RTLflow^{-p}	RTLflow	RTLflow^{-p}	RTLflow
4096	14.7s	12.4s (↑**19%**)	801.2s	791.2s (↑**1%**)
16384	27.4s	21.4s (↑**28%**)	1399.2s	1098.0s (↑**27%**)
65536	113.8s	72.5s (↑**57%**)	5281.0s	2957.8s (↑**79%**)

Table 3.5: Runtime comparison in terms of improvement (↑) between RTLflow with and without pipeline scheduling (RTLflow^{-p}) for Spinal and NVDLA with 100K cycles at different numbers of stimulus.

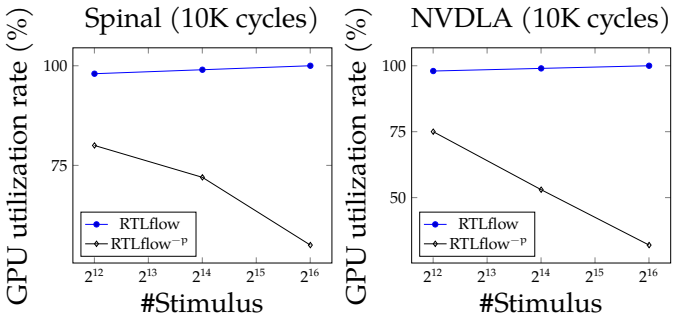


Figure 3.15: Comparison of GPU utilization between RTLflow with and without pipeline scheduling (RTLflow^{-p}) for simulating Spinal and NVDLA with different numbers of stimulus (under 10K cycles).

pipeline (RTLflow^{-p}) at different numbers of stimulus. For fairness purpose, we use OpenMP to parallelize `set_inputs` task in RTLflow^{-p}, and both methods use the same MCMC partitioning algorithm. Compared to RTLflow^{-p}, RTLflow is faster at all numbers of stimulus (up to 79%). The performance gap continues to enlarge as we increase the number of stimulus. Without our pipeline scheduling, RTLflow^{-p} requires a GPU to wait until CPU threads set inputs for all stimulus per cycle. The induced serialization overhead becomes significant as the number of stimulus increases.

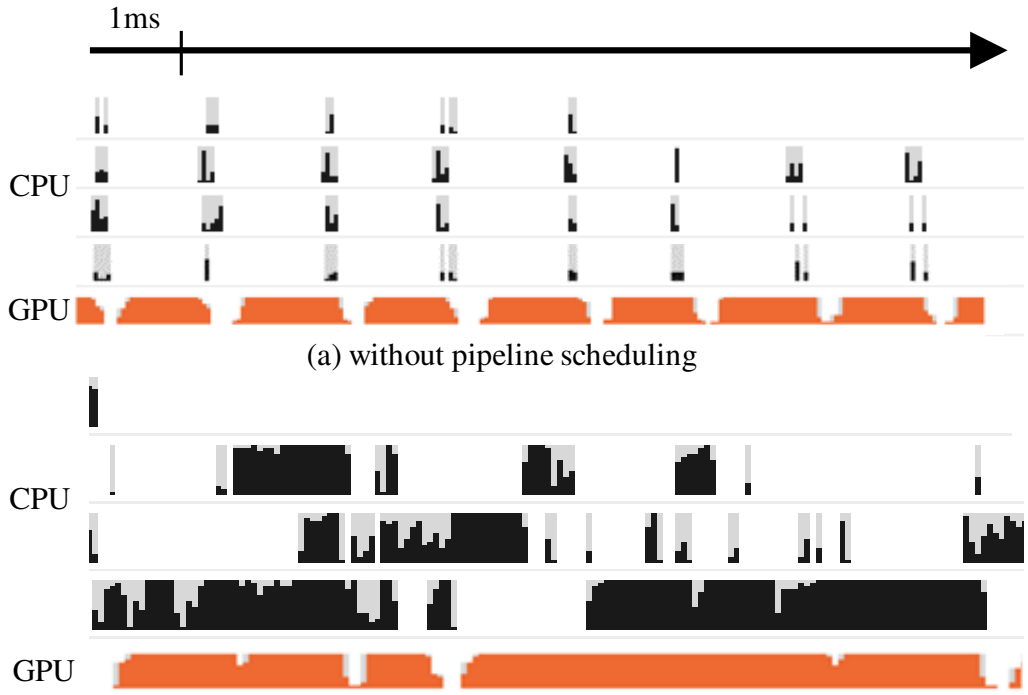
Figure 3.15 plots the average GPU utilization rate profiled by Nvidia System Management Interface [73]. RTLflow achieves nearly 100% GPU utilization rate across all numbers of stimulus on both Spinal and NVDLA,

whereas RTLflow^{-p} suffers from lower utilization rate as the number of stimulus increases. Our pipeline scheduling enables RTLflow to asynchronously dispatch a group of batch stimulus to GPU, thus keeping GPU highly utilized during the entire simulation.

Figure 3.16 plots the utilization timeline of RTLflow^{-p} and RTLflow using the data extracted from Nvidia Nsight Systems [3]. The timeline of CPU threads and GPU in RTLflow is much more overlapped than RTLflow^{-p}. Since our pipeline scheduling processes batch stimulus in groups, GPU does not need to wait for CPU threads to set inputs for all stimulus at each cycle. We also observe high CPU and GPU utilization rates on RTLflow. This is because our pipeline scheduling further explores inter-stimulus parallelism to enable efficient overlap between CPU and GPU.

3.6 Conclusion

In this chapter, we have introduced RTLflow, a GPU acceleration flow to speed up RTL simulation with batch stimulus. RTLflow transpiles the given RTL simulation code to C++ and CUDA, and combines GPU-aware partitioning algorithm with modern CUDA Graph parallelism to efficiently run multiple stimulus on partitioned RTL tasks. Our transpiler prevents designers from manually writing GPU kernels for simulating RTL processes and hence largely improves their productivity. To further enable effective computation overlaps between CPU and GPU, we have introduced a pipeline scheduling algorithm to explore inter-stimulus parallelism. We have evaluated RTLflow on industrial designs and demonstrated its promising performance compared to the industrial-strength RTL simulators, Verilator and ESSENT. For instance, RTLflow on one A6000 GPU outperforms 10 instances of Verilator each running 8 threads (a total of 80 CPU threads) with up to 40$\times$ on the NVDLA of 65536 stimulus. We



(a) without pipeline scheduling

(b) with pipeline scheduling

Figure 3.16: A snapshot of utilization timeline for RTLflow with and without pipeline scheduling, reported by Nvidia Nsight Systems [3].

have made RTLflow open-source to benefit the simulation community. In this work, Dian-Lun Lin was the primary contributor, responsible for the majority of the research and development efforts. Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the project. All authors participated in discussing the results and contributed to the preparation and review of the final manuscript.

4 GENFUZZ: GPU-ACCELERATED HARDWARE FUZZING USING GENETIC ALGORITHM WITH MULTIPLE INPUTS

4.1 Abstract

Hardware fuzzing has emerged as a promising automatic verification technique to efficiently discover and verify hardware vulnerabilities. However, hardware fuzzing can be extremely time-consuming due to compute-intensive iterative simulations. While recent research has explored several approaches to accelerate hardware fuzzing, nearly all of them are limited to single-input fuzzing using one thread of a CPU-based simulator. As a result, we propose Gen-Fuzz, a GPU-accelerated hardware fuzzer using a genetic algorithm with multiple inputs. Measuring experimental results on a real industrial design, we show that GenFuzz running on a single A6000 GPU and eight CPU cores achieves 80× runtime speed-up when compared to state-of-the-art hardware fuzzers.

4.2 Introduction

The ever-increasing complexity of hardware design has put significant strain on System-on-Chip (SoC) designers and system integrators to detect and evaluate hardware vulnerabilities within the design stage [28, 29, 4]. As SoC complexity continues to grow, industry-quality functional verification signoff typically requires a significant and growing amount of engineering effort to generate and simulate many thousands of test cases on the same Design-Under-Test (DUT) for converging on coverage closure and avoiding bug escape from corner cases. Much research over the past decades has focused on automatic *constrained random verification* (CRV) [32] to relieve the increasing strain for hardware engineers. CRV tests a DUT by randomly combining manually-defined inputs (i.e., instruc-

tions plus compiled RTL stimulus) into transaction sequences. However, since CRV relies only on randomly generated inputs, it suffers from zero knowledge of coverage and becomes inefficient when verifying large designs.

Coverage-guided verification, also known as *hardware fuzzing*, has emerged as a promising automatic hardware verification technique to efficiently discover and verify hardware vulnerabilities [28, 30]. Unlike CRV that randomly combines inputs, hardware fuzzing generates inputs by mutating previously interesting inputs (i.e., the inputs that increase coverage) to effectively discover unknown hardware behaviors. However, since this process requires time-consuming feedback analysis, hardware fuzzing often takes hours or days to finish.

To alleviate the long runtime, recent research has explored several approaches to speed up the single-input, single-threaded, per fuzzing iteration time. RFUZZ [28] proposes a mux-coverage metric that treats the select signal of each 2:1 multiplexer as a coverage point. However, RFUZZ cannot scale to large designs since their runtime grows significantly as the number of multiplexers increases. DirectFuzz [29] extends RFUZZ to generate test inputs that maximize the coverage of a specific block. Compared to RFUZZ, DirectFuzz can improve performance on small designs. However, the speedup on complex designs is insignificant (e.g., 1.08$\times$ on the Sodor1Stage RISC-V processor). Also, their work only targets RFUZZ's mux coverage and is not generalizable to other coverage metrics. DIFUZZRTL [30] introduces a reg-coverage metric to monitor value changes of control registers connected to mux control signals. While DIFUZZRTL's reg coverage shows capability for large designs, their fuzzing technique requires many hours or days to achieve high coverage.

TheHuzz [31] explores processor states using multiple coverage metrics. However, it induces over 70% runtime overhead when collecting coverage data since it must access multiple metrics per fuzzing iteration.

Hw-Fuzzing [32] converts a hardware-description-language (HDL) model into an equivalent software model using Verilator, and performs fuzzing on the software code using software coverage metrics. Although it shows software coverage metrics are comparable with HDL line coverage, other hardware coverage metrics such as finite-state-machine (FSM) coverage cannot be easily added. Other research has leveraged FPGAs to accelerate hardware fuzzing [30, 28]. However, there are three drawbacks of FPGA-based hardware fuzzing: 1) It suffers from a complicated compilation setup. 2) It does not provide visibility to internal signals, complicating bug detection. 3) Instrumenting a design on an FPGA is challenging[30, 28].

	RFUZZ	DirectFuzz	DIFUZZRTL	TheHuzz	Hw-Fuzzing	**GenFuzz**
FPGA	Y	N	Y	N	N	N
CPU	single thread	single thread	single thread	single thread	single thread	**multiple threads**
GPU	N	N	N	N	N	**Y**
#Inputs	1	1	1	1	1	**multiple**

Figure 4.1: Comparison between GenFuzz and existing hardware fuzzers.

While all these approaches have shown coverage or runtime improvements, *nearly all of them are limited to single-input fuzzing using one thread of simulation on a CPU architecture*. Recently, RTL simulation research has achieved significant performance improvement by leveraging GPUs to simulate multiple inputs simultaneously [4]. This result inspires us to accelerate hardware fuzzing by exploring data parallelism, which we refer to as *multi-input hardware fuzzing*, using CPU-GPU heterogeneous computing. However, multi-input hardware fuzzing is extremely challenging for three reasons. Firstly, existing works focus on speeding up single-input fuzzing using sequential mutation frameworks. Multi-input hardware fuzzing requires a new CPU-GPU task decomposition strategy to benefit from heterogeneous parallelism. Secondly, multi-input hardware fuzzing

needs an effective mutation algorithm to find the best previous inputs as seeds and generate multiple new inputs of interest. Lastly, fuzzing multiple inputs in parallel can introduce inefficiencies from highly overlapped coverage within a fuzzing iteration. We need to rule out unwanted inputs that cause redundant overlaps.

To overcome these challenges, we propose GenFuzz, a GPU-accelerated hardware fuzzer using a genetic algorithm (GA) with multiple inputs. Figure 4.1 compares the key differences between GenFuzz and existing hardware fuzzers. To the best of our knowledge, this is the first GPU-accelerated hardware fuzzing using GA in the literature. We summarize three key contributions as follows:

- We design an efficient **multi-input hardware fuzzer** to fuzz multiple inputs simultaneously using both CPU and GPU parallelisms.
- We design an effective **GA-based framework** to iteratively produce multiple inputs of interest that are most likely to extend the coverage.
- We design a novel **coverage-maximization algorithm** to avoid overlapped coverage for both inter- and intra- fuzzing iterations.

We have evaluated GenFuzz on real designs and demonstrated its promising performance compared to the state-of-the-art DIFUZZRTL [30] and RFUZZ [28]. GenFuzz using one A6000 GPU and eight CPU cores outperforms DIFUZZRTL on single CPU thread with up to 80× speed-up for BOOMCore design using reg coverage. We have also shown that GenFuzz achieves 2.1× more coverage points when the same number of instructions are fuzzed.

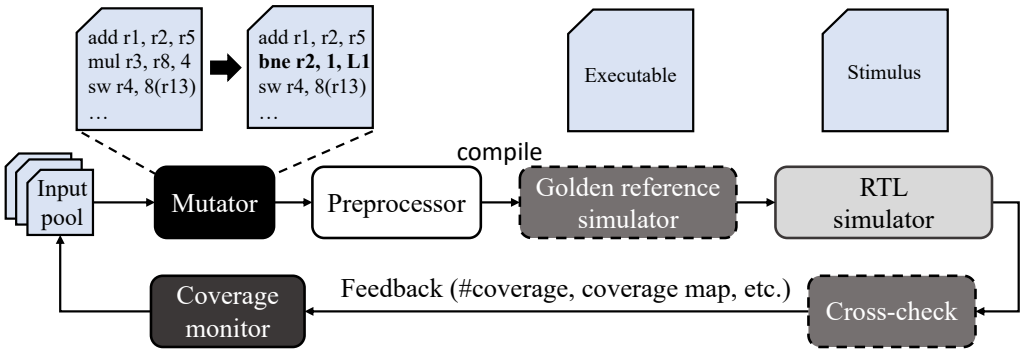


Figure 4.2: Conventional single-input hardware fuzzing flow.

4.3 Background

Conventional Hardware Fuzzing

Figure 4.2 shows the flow of conventional single-input hardware fuzzing [30]. We start by randomly choosing one input from the input pool that maintains a set of interesting inputs. The mutator generates a new input by mutating instructions from the selected input. The preprocessor compiles the input into a stimulus and an executable for simulation. Depending on different hardware fuzzers and coverage metrics, we may need a reference simulator to validate an input. We then use an RTL simulator to simulate the DUT with the mutated stimulus and collect the coverage data. The input that discovers new hardware states (i.e., new coverage points) is considered interesting and is saved back to the input pool for future fuzzing. The fuzzing iteration continues until we cannot explore new states for a maximum number of iterations. Finally, we use assertions or cross-check RTL simulation results against results from the reference simulator to detect bugs.

Genetic Algorithm

GA [74] is a search heuristic that reflects the process of natural selection. The best individuals are selected to produce good offsprings for the next generation. Listing 4.1 gives an example. The code wraps all individuals with a population and applies GA iteratively. A quantitative fitness function evaluates the quality of each individual in each iteration. We then select individuals with better fitness as parents to produce superior offsprings. For each parent pair, we perform crossover to exchange genes between parents to generate offsprings. The new offsprings become a new population for the next GA iteration. The iteration continues until we cannot find a better solution after a maximum number of iterations.

```
Population pop;              // construct a population
Individuals pars;            // construct parents
Individuals offs;            // construct offsprings
pop.initialize();            // initialize a population
while(iter < MAX_ITER) {
  pop.caculate_fitness(). // calculate  fitness
  pars = pop.select();       // select good parents
  offs = pars.crossover();// generate new offsprings
  offs.mutate();             // mutate to add diversity
  pop.replace(offs);         // replace old population
}
```

Listing 4.1: A common C++ genetic algorithm.

4.4 GenFuzz

Multi-input Hardware Fuzzing

At a high level, GenFuzz efficiently discovers new coverage by fuzzing multiple inputs in parallel at each fuzzing iteration. Figure 4.3 shows the overview of GenFuzz. We define the multi-input hardware fuzzing

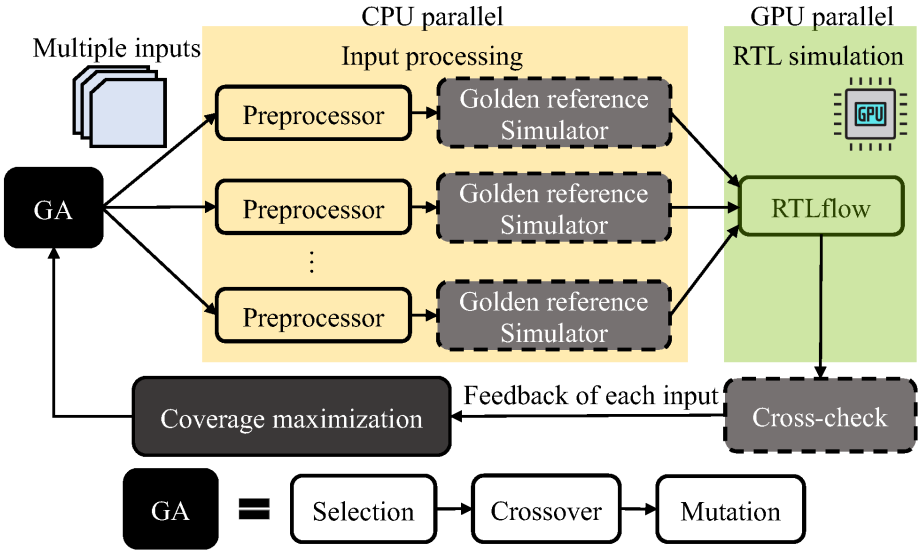


Figure 4.3: Overview of GenFuzz.

workload as a parallel task dependency graph that iterates five stages: *GA*, *input processing*, *RTL simulation*, *cross-check* and *coverage maximization*. At the GA stage, GenFuzz incorporates three GA processes, *selection*, *crossover*, and *mutation*, to generate inputs that are most likely to extend coverage. Based on the results from the previous fuzzing iteration, we select the best inputs as parents to generate new inputs. The input processing stage involves CPU-intensive tasks including the compilation of simulation inputs and file I/O. Without data parallelism, RTL simulation needs to wait until we process all inputs, thus incurring significant overhead. To improve runtime performance, we evenly distribute inputs across different CPUs to process each input in parallel.

We integrate the state-of-the-art RTL simulator, RTLflow [4], into GenFuzz to enable GPU acceleration for multi-stimulus RTL simulation. Unlike existing hardware fuzzers that typically use Verilator or ModelSim to simulate one stimulus at a time, RTLflow achieves high-throughput RTL simulation by running multiple stimuli in parallel using GPU. After the RTL simulation, we cross-check results derived from the reference simu-

lator and RTLflow. We do not parallelize cross-check since this stage is fast (e.g., a few seconds to finish). Our coverage-maximization algorithm scores each input by analyzing the feedback of each input. Finally, we send each input with its fitness to GA for selection. The fuzzing iteration continues until GA cannot find new coverage after a maximum number of iterations.

GA-based Framework

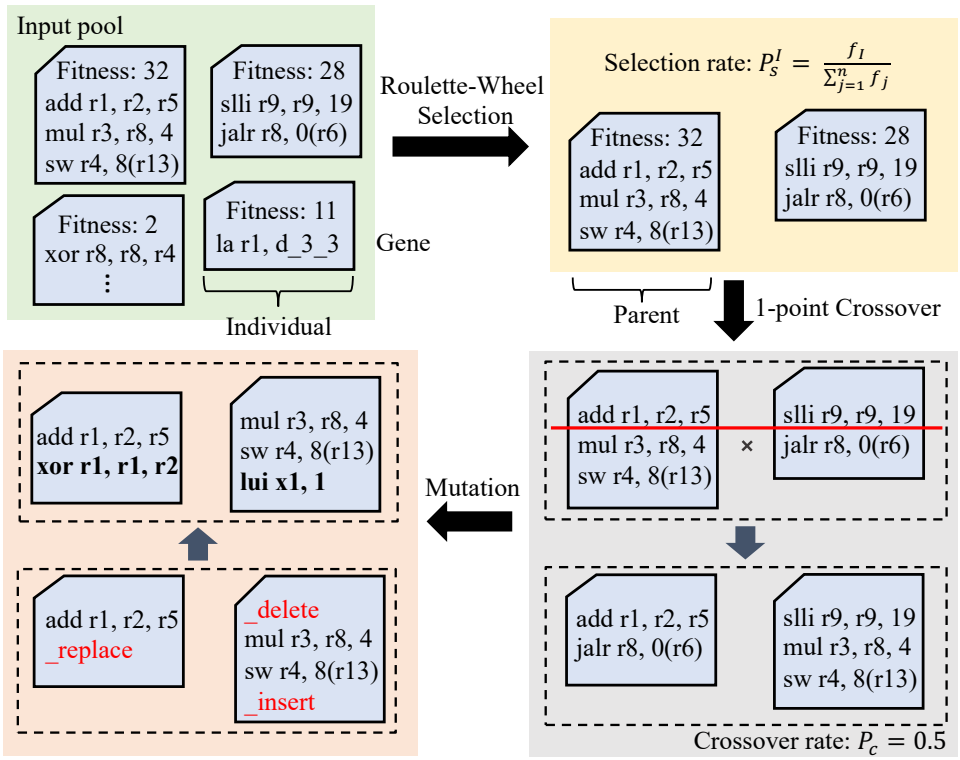


Figure 4.4: Overview of our GA-based fuzzing framework.

The goal of our GA-based framework is to produce new inputs that can maximally extend coverage using results from the previous fuzzing iteration. Our GA consists of *selection*, *crossover*, and *mutation*. Unlike existing mutation approaches that only select one input to mutate, our GA

framework exchanges interesting instructions between two parents and passes instructions to newborn inputs. Furthermore, we adopt variable-length individual representation. Each input can have a different number of instructions, allowing us to efficiently explore coverage on the time dimension.

Figure 4.4 shows the overview of our GA-based framework. Each gene represents an instruction, and each individual consists of multiple genes to form an input. Our GA framework can be applied to arbitrary coverage metrics and inputs. For example, to perform fuzzing on the RTL level using mux coverage [28], we can map the input value of each input pin as a gene. Each individual thus concatenates multiple genes to form a stimulus. Since the mux coverage is collected after RTL simulation, it can be transformed to fitness for the selection process.

Selection

The selection process chooses individuals with higher fitness from the input pool for later reproduction. We apply Roulette-Wheel Selection (RWS) for our framework such that the probability of choosing an individual is proportional to its fitness. Compared to truncation selection which directly eliminates a fixed percentage of the weakest candidates, RWS can still select individuals with lower fitness to create more diversity and to avoid quick convergence. We define the selection rate P_s^I of input I as $P_s^I = \frac{f_I}{\sum_{j=1}^{n} f_j}$ where f_I is the fitness of input I and n is the number of input.

Crossover

The crossover process allows parents to exchange instructions for producing the next generation of individuals. We choose two parents from selected individuals and apply one-point crossover. Since individuals can have different lengths of genes, we randomly choose a crossover point based on the parent with smaller length. As shown in Figure 4.4, genes

of both parents after the crossover point (red line) are interchanged. We choose a crossover rate $P_c = 0.5$ to randomly determine whether two parents occur crossover. If crossover does not happen, the two parents are considered as newborn inputs and passed into the mutation process.

Mutation

The mutation process provides a mechanism for newborn inputs to escape from local regions and to create more diversity. During the mutation phase, we iterate each gene in an individual and decide if the gene is mutated using the mutation rate P_m. Our mutation contains three operators: *insert*, *delete*, and *replace* as shown in Figure 4.4. Once mutation occurs, we randomly choose one operator to mutate the gene.

The mutation rate P_m plays an important role in the mutation process. If the mutation rate is minimal, there will be too many similar individuals. On the other hand, having a large mutation rate can easily direct GA toward random search. To have a better mutation rate for effective GA search, we use a time-dependent mutation rate proposed by [74]:

$$P_m = \begin{cases} 0.6\left[1 - (\frac{t}{T})^2\right], & 0 \leqslant t \leqslant 0.2T \\ 0.2\left[0.1(\frac{t-T}{T})^2\right] + 0.05, & 0.2T < t \leqslant T \end{cases}$$

where T is the total number of fuzzing iterations and t is the t^{th} iteration.

Coverage-Maximization Algorithm

Multi-input hardware fuzzing can introduce inefficiencies due to highly overlapped coverage within a fuzzing iteration. Moreover, selecting inputs with large overlap as parents causes newborn inputs to inherit the same overlap. The overlap grows significantly as we increase the number of fuzzing iterations. To overcome this problem, a greedy solution is to find the top-k inputs based on the maximum coverage problem, defined as follows:

Definition 4.1. *Let* $\{I_j\}_{j=1}^{n}$ *be the sequence of inputs where* I_j *is the* $j-$*th input and* n *is the number of these inputs. Let* $C(I_j)$ *represent the set of coverage discovered by input* I_j*. Then, given a positive number* $k \leqslant n$*, the goal of the maximum coverage problem is to find a subsequence* $\{I_{j_l}\}_{l=1}^{k} = \{I_{j_1}, I_{j_2}, ..., I_{j_k}\}$*, called the top-*$k$ *inputs in* $\{I_j\}_{j=1}^{n}$*, such that their total coverage* $|\bigcup_{l=1}^{k} C(I_{j_l})|$ *is maximized.*

Unfortunately, this problem is NP-hard [75]. Also, existing greedy algorithms that choose one input with the largest uncovered coverage at a time cannot be used out of the box. Specifically, after selecting the best input, greedy algorithms require remaining inputs to re-calculate uncovered coverage by iterating all coverage for each input. The time complexity of greedy algorithms is thus $O((cov\_size * n)^2)$ where cov_size is the coverage size in a design and n is the number of inputs per fuzzing iteration. Since the number of coverage points is in the millions for large designs, such greedy algorithms are extremely time-consuming.

To reduce the time complexity, we introduce two coverage metrics, *delta* and *progressive* coverage, as fitness for GA to select inputs:

Definition 4.2. *The delta coverage* $C_d(I_j)$ *measures how much new coverage is discovered by* I_j *compared to the total coverage explored in previous fuzzing iterations.*

Definition 4.3. *The progressive coverage* $C_p(I_j)$ *measures how much new coverage is discovered by* I_j *compared to the total coverage explored by all inputs before* I_j*.*

Figure 4.5 shows an example of calculating delta and progressive coverage for two inputs I_1, I_2. We represent the total explored coverage at iteration $t-1$ by C^{t-1} and the coverage of input I_j at iteration t by C_j^t.

Since all inputs before the first input are in previous fuzzing iterations, $C_d(I_1) = C_p(I_1)$ are both given by taking the complement of C^{t-1} w.r.t. C_1^t.

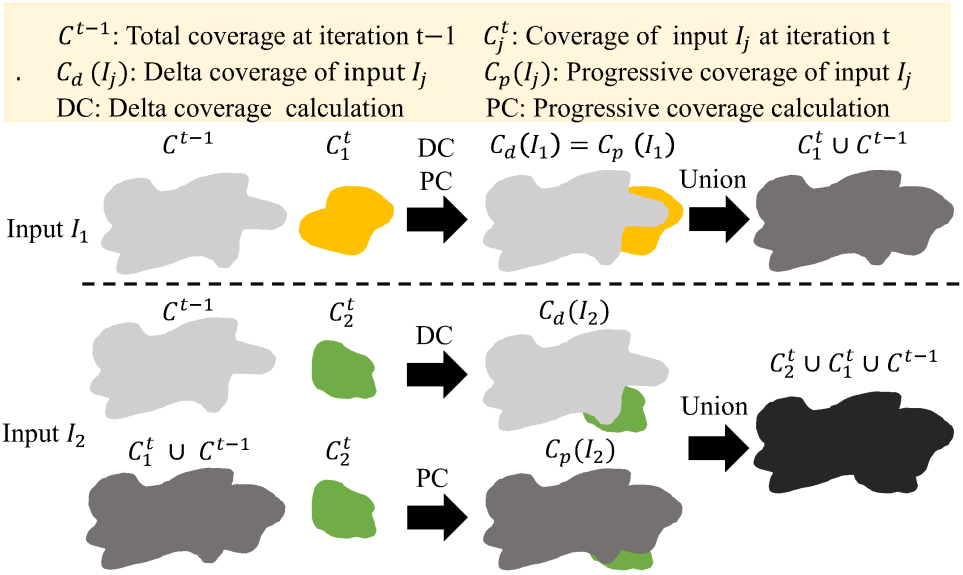


Figure 4.5: The proposed progressive coverage and delta coverage calculation. The progressive coverage and delta coverage of the first input is identical.

Similar to $C_d(I_1)$, we calculate $C_d(I_2)$ by taking the complement of C^{t-1} w.r.t. C_2^t. On the other hand, we take the union of C^{t-1} and C_1^t to get the total coverage explored by all the inputs before I_2. Finally, we calculate $C_p(I_2)$ by taking the complement of $C^{t-1} \cup C_1^t$ w.r.t. C_2^t.

Based on our delta and progressive coverage metrics, we define the fitness function as follows:

$$Fitness(I_j) = Norm(C_d(I_j)) + Norm(C_p(I_j))$$

where Norm represents mix-max normalization that transforms two coverage metrics to the same scale.

Algorithm 4 shows our coverage-maximization algorithm. Each input owns a coverage map of size cov_size to record its coverage. The $total\_cov\_map$ represents total coverage by taking a union of all inputs. In the beginning, we initialize $total\_cov$ and $total\_cov\_map$ by using

Algorithm 4: Coverage-maximization algorithm

```
   Input: NUM_INPUTS: Number of inputs
   Input: COV_SIZE: Coverage size
   Input: prev_total_cov: Total coverage at previous iter
   Input: prev_total_cov_map: Total coverage map at previous iter
   Input: cov_maps: Array of coverage maps
   Output: total_cov: Total coverage
   Output: total_cov_map: Total coverage map
   Output: fitness: Array of fitness
 1 delta_covs.initialize(0)
 2 prog_covs.initialize(0)
 3 total_cov ← prev_total_cov
 4 total_cov_map ← prev_total_cov_map
 5 i, c ← 0
   /* delta coverage caculation                                    */
 6 while i++ < NUM_INPUTS
 7 |   while c++ < COV_SIZE
 8 |   |   result ← cov_maps[i][c]
 9 |   |   prev_result ← prev_total_cov_map[c]
10 |   |   if prev_result == 0 and result == 1 then
11 |   |   |   delta_covs[i]++
12 |   |   end
13 |   end
14 end
15 i, c ← 0
   /* progressive coverage calculation                             */
16 while i++ < NUM_INPUTS
17 |   while c++ < COV_SIZE
18 |   |   result ← cov_maps[i][c]
19 |   |   prev_result ← total_cov_map[c]
20 |   |   if prev_result == 0 and result == 1 then
21 |   |   |   total_cov_map[c] = 1
22 |   |   |   prog_covs[i]++
23 |   |   |   total_cov++
24 |   |   end
25 |   end
26 end
27 min_max_norm(delta_covs, prog_covs)
28 i ← 0
29 while i++ < NUM_INPUTS
30 |   fitness[i] ← delta_covs[i] + prog_covs[i]
31 end
```

coverage explored in previous fuzzing iterations (lines 3-4). During delta coverage calculation, each input iterates its coverage map and compares coverage results with $prev\_total\_cov\_map$ (lines 6-14). If an input discovers a new coverage that was not explored in previous fuzzing iterations, we increment the delta coverage of the input by one (lines 10-12). During progressive coverage calculation, each input iterates its coverage map again and compares coverage results with $total\_cov\_map$ (lines 16-26). If an input discovers a new coverage that was not toggled in $total\_cov\_map$, we increment the progressive coverage of the input and $total\_cov$ by one (lines 22 and 23). Also, we toggle the corresponding coverage in $total\_cov\_map$ to one (line 21). In this case, the next input that discovers the same coverage cannot increment its progressive coverage. After coverage calculation, we normalize both delta and progressive coverage (line 27). Finally, we calculate the fitness for each input by summing up its delta and progressive coverage (lines 29-31). We can clearly see the time complexity of our algorithm is $O(cov\_size * n)$.

4.5 Experimental Results

We conducted our experiments on a 64-bit CentOS Linux machine with one NVIDIA RTX A6000 GPU and eight Intel i7-11700 CPU cores at 2.5 GHz. We compiled our programs with CUDA NVCC 11.6 on a GCC 8.3.0 host compiler and enabled optimization flag `-O3` and C++17 standard `-std=c++17`. Each fuzzing iteration has an input size of 1024. For each run, we used 1024 GPU threads for RTL simulation and 8 CPU cores for all host operations. We used Taskflow [34, 72] to parallelize our task dependency graph. All data is an average of five runs.

We consider two state-of-the-art hardware fuzzers, RFUZZ [28] and DIFUZZRTL [30], as our baselines. RFUZZ and DIFUZZRTL each proposed coverage metrics (i.e., mux coverage and reg coverage) to measure

the number of discovered design states. We compare GenFuzz with each fuzzer using their coverage metrics. DIFUZZRTL performs fuzzing on its CPU input format at the instruction level. It combines several instructions into a word and performs per-word mutations. On the other hand, RFUZZ directly fuzzes at the RTL level. It concatenates all input pins as an input vector to represent input values in one test cycle. To explore coverage on the time dimension, it further concatenates several single-cycle stimuli to form a multi-cycle stimulus. For a fair comparison, we perform fuzzing at the same level as DIFUZZRTL and RFUZZ. We use Verilator as the RTL simulator for RFUZZ and DIFUZZRTL. For single-input hardware fuzzing, RTLflow is slower than Verilator due to little data parallelism [4]. To demonstrate our efficiency, we evaluate GenFuzz on five real designs, BoomCore1, BoomCore2, RocketCore, Sodor3Stage, and Sodor5Stage, provided by RFUZZ and DIFUZZRTL.

			Runtime (s) for achieving K% #Coverage					
			K=50		K=70		K=100	
Benchmark	Verilog LOC	#Coverage	DIFUZZRTL	GenFuzz	DIFUZZRTL	GenFuzz	DIFUZZRTL	GenFuzz
RocketCore	81883	110509	6218s	578s (**10.8**×)	18972s	1216s (**15.6**×)	172800s	2835s (**61.0**×)
BoomCore1	193689	518506	7503s	484s (**15.5**×)	22278s	1180s (**18.9**×)	172800s	2160s (**80.0**×)
BoomCore2	239282	684202	9208s	835s (**11.0**×)	24555s	1428s (**17.2**×)	172800s	2690s (**64.2**×)

Table 4.1: Overall performance comparison between DIFUZZRTL and GenFuzz on different benchmarks for achieving 50%, 70%, and 100% coverage using reg coverage. The benchmark statistics show Verilog lines of code (Verilog LOC) and number of achieved coverage (#Coverage) by running DIFUZZRTL for 48 hours. Bold text represents speed-up.

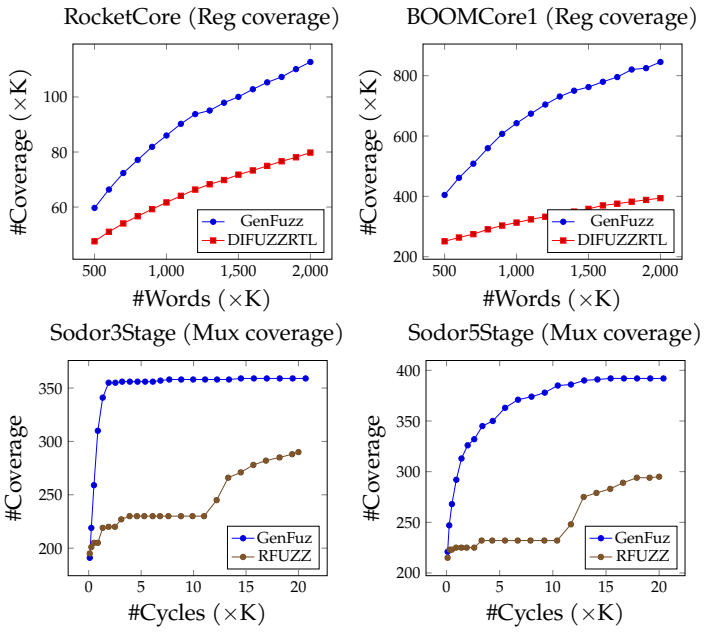


Figure 4.6: Comparison of coverage throughput among GenFuzz, DIFUZZRTL, and RFUZZ on RocketCore, BOOMCore, Sodor3Stage, and Sodor5Stage. The x-axis uses the number of words and the number of cycles.

Overall Performance Comparison

Table 4.1 compares the overall runtime between GenFuzz and DIFUZZRTL on RocketCore, BoomCore1, and BoomCore2. We run DIFUZZRTL for 48 hours to derive the total number of coverage (#Coverage), and compare runtime of GenFuzz and DIFUZZRTL for achieving 50%, 70%, and 100% #coverage. GenFuzz outperforms DIFUZZRTL in all scenarios. For achieving 100% #coverage, GenFuzz is 61× faster on RocketCore and is 80× faster on BOOMCore1. The significant improvement on runtime demonstrates the promise of our multi-input hardware fuzzing techniques. The speedup of achieving 100% #coverage is larger than 50% and 70% #coverage. When discovered coverage becomes large, the coverage of DIFUZZRTL starts to saturate. On the other hand, our genetic algorithm keeps finding new inputs of interest and thus efficiently increases the coverage. Figure 4.6

compares the coverage throughput among GenFuzz, DIFUZZRTL, and RFUZZ on different designs. To demonstrate the efficiency of coverage throughput, we use the number of words and cycles as the x-axis for different coverage metrics. Compared to DIFUZZRTL using reg coverage, GenFuzz achieves 2.1× speed-up using the same amount of words. The throughput gap continues to grow as we increase the number of inputs. Compared to RFUZZ using mux coverage, GenFuzz achieves 1.6× speed-up using the same amount of cycles. The number of coverage quickly saturates since both Sodor3Stage and Sodor5Stage are small designs that do not have many coverage points to discover.

Performance Result of Coverage-maximization Algorithm

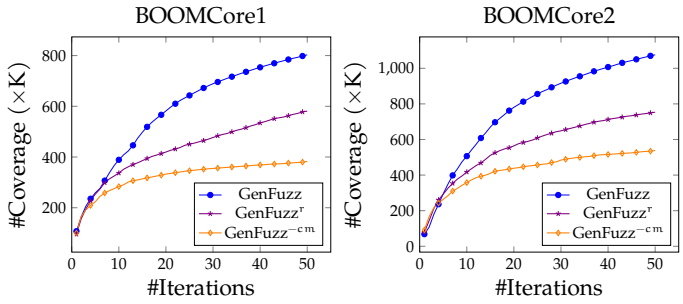


Figure 4.7: Coverage growth over increasing numbers of iterations for GenFuzz, GenFuzz with mutation rate $P_r = 1$ (GenFuzz^r), and GenFuzz without coverage-maximization algorithm (GenFuzz^{-cm}) on BOOMCore using reg coverage with 256 inputs.

In this section, we study the performance benefit of our coverage-maximization algorithm. Figure 4.7 shows total coverage over increasing numbers of fuzzing iterations for GenFuzz, random (GenFuzz^r), and GenFuzz without coverage-maximization algorithm (GenFuzz^{-cm}). In GenFuzz^r, we set the mutation rate to 1 to achieve random testing. In GenFuzz^{-cm}, each input uses the coverage number achieved by itself as

fitness. We can clearly see the performance advantage of our coverage-maximization algorithm. GenFuzz achieves a higher coverage number than other implementations in almost all scenarios. The performance gap continues to enlarge as we increase the number of fuzzing iterations. Without our coverage-maximization algorithm, both GenFuzz^{r} and GenFuzz^{-cm} fail to distinguish overlapped coverage within each fuzzing iteration, thus misguiding GA to select inputs that are less likely to extend coverage. The coverage growth of GenFuzz^{-cm} is even slower than GenFuzz^{r} because GenFuzz^{-cm} does not consider coverage improvement at each fuzzing iteration. GA converges within 20 iterations and hardly discovers new coverage points.

Runtime Breakdown of GenFuzz

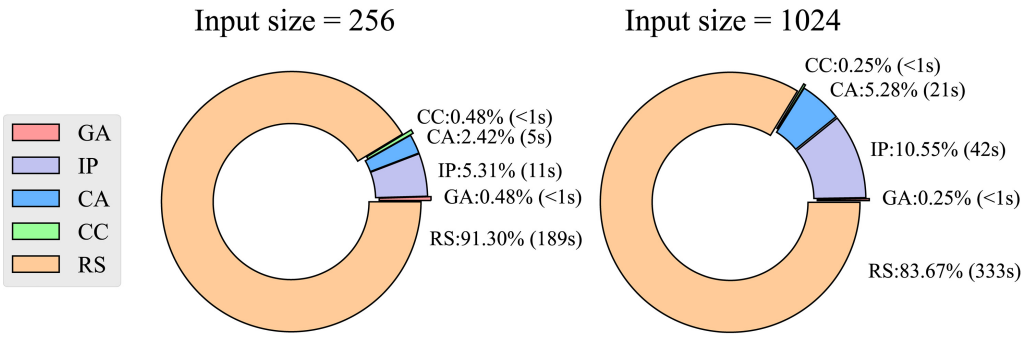


Figure 4.8: Runtime breakdown of GenFuzz on BoomCore1.

Figure 4.8 shows the runtime breakdown of GenFuzz on BoomCore1 with 256 and 1024 inputs where GA, IP, CA, CC, and RS represent the genetic algorithm, input processing, coverage-maximization algorithm, cross-check, and RTL simulation components, respectively. The majority of the runtime (>80%) is taken by RS. Both GA and CA only take <6% runtime of the entire flow. Taking 1024 inputs as an example, CA only takes 21 seconds to finish. Without our coverage-maximization algorithm,

Table 4.2: Runtime Comparison for finding bugs in BOOMCore between DIFUZZRTL and GenFuzz.

Bug ID	DIFUZZRTL	GenFuzz
Issue #492	20.3h	1027s (**71.1**×)
Issue #567	✗	854s

existing greedy algorithms require $cov\_size * n * 21$ seconds to finish, where cov_size is typically millions in large designs.

Bug Detection

Table 4.2 compares the runtime of GenFuzz and DIFUZZRTL for finding bugs in the BoomCore1 [76]. We derive DIFUZZRTL runtime results from the paper. Compared to DIFUZZRTL, GenFuzz achieves 71.1× speed-up to find Issue #492. Furthermore, GenFuzz can find Issue #567 not discovered by DIFUZZRTL.

4.6 Conclusion

In this chapter , we have introduced GenFuzz, a GPU-accelerated hardware fuzzer using a novel genetic algorithm to speed up hardware fuzzing with multiple inputs. GenFuzz introduces a multi-input hardware fuzzer, a genetic algorithm-based framework, and a coverage-maximization algorithm to accelerate hardware fuzzing to a new performance milestone. Measuring experimental results on real industrial designs, GenFuzz achieves up to 80× runtime speed-up when compared to DIFUZZRTL and RFUZZ. In this work, Dian-Lun Lin was the primary contributor, responsible for the majority of the research and development efforts. Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the project. Shih-Hsin Wang

assisted in developing the theoretical proof for the coverage-maximization algorithm. All authors participated in discussing the results and contributed to the preparation and review of the final manuscript.

5 TARORTL: ACCELERATING RTL SIMULATION USING COROUTINE-BASED HETEROGENEOUS TASK GRAPH SCHEDULING

5.1 abstract

RTL simulation is critical for validating hardware designs. However, RTL simulation can be time-consuming for large designs. Existing RTL simulators have leveraged task graph parallelism to accelerate simulation on a CPU- and/or GPU-parallel architecture. Despite the improved performance, they all assume atomic execution per task and do not anticipate multitasking that can bring significant performance advantages. As a result, we introduce TaroRTL, a coroutine-based task graph scheduler for efficient RTL simulation. TaroRTL enables non-blocking GPU and I/O tasks within a task graph, ensuring that threads are not blocked waiting for GPU or I/O tasks to finish. It also designs a coroutine-aware work-stealing algorithm to avoid unnecessary context switches. Compared to a state-of-the-art GPU-accelerated RTL simulator, TaroRTL can further achieve 40–80% speed-up while using fewer CPU resources to simulate large industrial designs.

5.2 Introduction

The time-consuming nature of Register-transfer level (RTL) simulation poses a significant challenge for verifying today's highly complex SoCs, processors, and accelerators [4, 60, 33]. As SoC complexity continues to grow, achieving industry-quality functional verification signoff typically demands a significant and growing amount of simulation tests on the same Design-Under-Test (DUT) with different input stimuli, all in preparation

for tapeout. For a comprehensive analysis of the design's behavior, SoC designers even require a Value-Change-Dump (VCD) file, resulting in substantial long runtime associated with input and output (I/O) operations to capture and process traces [77]. Speeding up RTL simulation is crucial for coping with the rapidly increasing design complexity and the shorter time-to-market demands.

State-of-the-art RTL simulators have leveraged *task graph parallelism* to accelerate simulation on a CPU- and/or GPU-parallel architecture [20, 33, 4]. This task graph consists of various tasks performed per simulation cycle, such as evaluating logic elements, setting inputs, or I/O VCD dump. Through task graph scheduling, multiple tasks can be scheduled and executed concurrently once the dependency constraints are met. To name a few state of the art: Verilator [20] is an open-source RTL simulator that has been widely used in both academic and industrial projects. They group adjacent logic elements into a set of macro tasks and dependencies. This macro task graph is scheduled using a static multi-threaded scheduling algorithm. Despite improved performance, the result has largely plateaued at 8-10 CPU cores. RepCut [33] cuts the circuit into balanced partitions with small overlaps. Their task graph scheduling reduces synchronization and achieves superlinear speed-up by removing inter-task dependencies with replication. However, RepCut has been limited to strong scaling of a single input stimulus and does not consider weak scaling of multi-stimulus simulation. To further improve the throughput performance, RTLflow [4] runs many independent input stimuli in parallel on a GPU. They incorporate a heterogeneous set of tasks (i.e., CPU and GPU) within a task graph and schedule them using a work-stealing algorithm [12]. However, RTLflow requires a CPU thread to wait for GPU tasks to finish per simulation cycle, resulting in significant CPU waiting time.

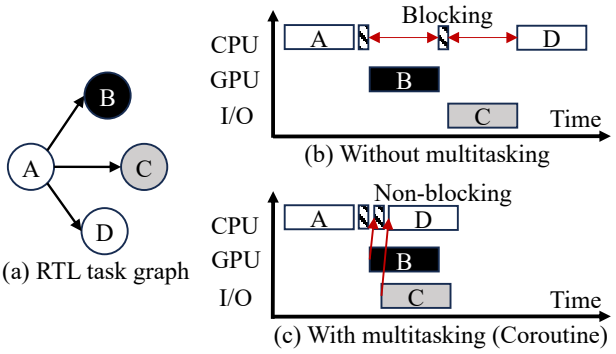


Figure 5.1: Performance comparison with and without multitasking using one CPU and one GPU. TaroRTL enables non-blocking GPU and I/O tasks to improve total runtime. The patterned rectangle represents the kernel call overhead (GPU and I/O).

While all these approaches have shown runtime or throughput improvements, the performance is far from optimal. Specifically, existing task graph scheduling solutions for RTL simulation all assume *atomic* execution per task (i.e., a thread runs or blocks until its assigned task is complete) and do not anticipate multitasking that can reduce CPU waiting time on awaiting GPU and I/O tasks to finish. For instance, Figure 5.1 shows an RTL task graph with different task types [4]. Without multitasking as shown in (b), when a CPU thread invokes a GPU task (task B) or an I/O task (task C), it will block until the task finishes.

Recently, the new C++20/23 standard has introduced Coroutine [78]. Coroutine offers a new mechanism for programming multitasking by allowing suspension and resumption of a function from its running thread. This mechanism has inspired us to design a new coroutine-based task graph scheduling solution with significantly improved performance. As shown in Figure 1(c), after invoking task B and task C, the CPU thread suspends those tasks and multitask to task D without being blocked. Compared to (b), using Coroutine enables better utilization of computing resources and reduces the total runtime.

However, designing a coroutine-based task graph scheduler is very challenging for three reasons. First, coroutines present a different execution mechanism compared to traditional function calls, primarily due to

their ability to suspend and resume execution at certain points rather than executing to completion like traditional functions. This difference poses challenges for existing task graph schedulers [20, 4, 33, 34, 72, 79], as they are typically designed with the assumption of traditional function calls and lack support for coroutine-specific features. Second, Coroutine's suspension and resumption ability requires a specially designed scheduling algorithm to minimize the cost of context switches. Furthermore, a coroutine does not automatically resume after suspension; instead, it requires a scheduler to track and control its execution. Managing and tracking the execution status of each coroutine becomes complicated when dealing with complex workloads.

To overcome these challenges, we introduce TaroRTL, an efficient coroutine-based task graph scheduler for RTL simulation. We summarize three key contributions as follows:

- We design a coroutine-based task graph scheduling model to enable non-blocking GPU and I/O tasks within a task graph.
- We design a coroutine-aware work-stealing algorithm to avoid unnecessary context switches and cache misses.
- We design an execution control strategy to track and control the execution of each invoked GPU and I/O task.

We have evaluated TaroRTL on industrial designs and demonstrated its promising performance compared to the state-of-the-art RTLflow (CPU- and GPU-based) [4] and Verilator (CPU-based) [20]. As an example, TaroRTL can speed up RTLflow by 40–80% while using fewer CPU resources to simulate large industrial designs.

5.3 The Motivation of Using Coroutine in RTL Simulation

Existing task graph scheduling solutions for RTL simulation all assume atomic execution, resulting in significant CPU waiting time on awaiting GPU and I/O tasks to finish. This problem prevents us from fully unleashing the power of heterogeneous simulation. Figure 5.2 gives an example of CPU waiting and active time growth over increasing input stimuli in RTLflow [4] (CPU- and GPU-based). The CPU waiting time inevitably reduces the overall efficiency of an RTL simulator, especially when simulating a design with numerous input stimuli. The increasing CPU waiting time over increasing number of input stimuli indicates untapped performance potential within the RTL simulation. Furthermore, CPUs need to keep spinning until GPU completes its task, wasting a lot of unnecessary CPU resources.

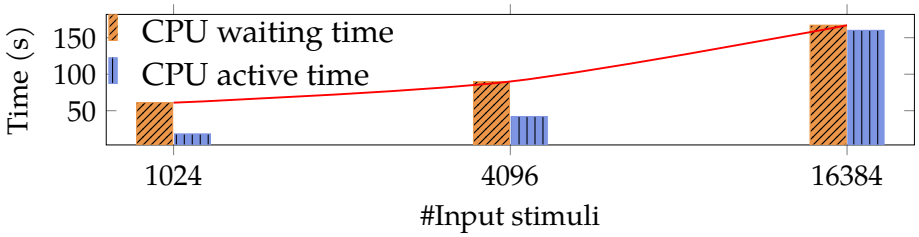


Figure 5.2: CPU waiting and active time growth over increasing numbers of input stimuli in RTLflow [4]. The relative ratio of waiting time gets smaller as the number of input stimuli increases because a large number of input stimuli induce a significant amount of CPU computation for setting inputs.

Unlike a traditional function that runs to completion and returns a value, a coroutine can be suspended and resumed at specific points without losing its state. Specifically, modern C++ Coroutine allows a CPU thread to suspend its current task and resume other tasks (i.e., multitasking) while awaiting GPU or I/O operations to finish. This property makes

coroutines particularly useful for parallel RTL simulation. Listing 5.1 gives a CPU-GPU simulation example for simulating a design with two input stimuli using Coroutine. The code simulate an input design *dut* cycle by cycle with a scheduler *s*. At each cycle iteration, we first set the inputs of *dut* using the given input stimulus (Line 5). When a CPU thread offloads `eval` to GPU (Line 7 and 9), it can multitask to another input stimulus for `set_inputs`. Without Coroutine, existing RTL simulators such as RTLflow [4] require a CPU thread to wait for `eval` on GPU to finish.

```
1  void sim(Stimulus& stim) {
2    Design dut;
3    size_t c{0};
4    while(!dut.stop and c <= NUM_CYCLES) {
5      dut.set_inputs(stim, c);
6      dut.set_clock(0);
7      co_await dut.eval();    // offload to GPU and multitask
8      dut.set_clock(1);
9      co_await dut.eval();    // offload to GPU and multitask
10     c += 1;
11   }
12 }
13 int main() {
14   Scheduler s;
15   Stimuli stim = get_stimuli(); // get input stimuli
16   s.emplace(sim, stim[0]); // emplace a sim task for stim 0
17   s.emplace(sim, stim[1]); // emplace a sim task for stim 1
18   s.schedule();            // schedule the two sim tasks
19   return 0;
20 }
```

Listing 5.1: An example of RTL simulation using Coroutine. When co_await, a CPU thread multitasks to another input stimulus. The scheduler needs to track and control the execution of each invoked task.

5.4 TaroRTL

At a high level, TaroRTL enables multitasking within a task graph through Coroutine. We allow CPU threads to multitask without being blocked by GPU or I/O tasks. We introduce a coroutine-aware work stealing to minimize context switches. Our execution control strategy effectively tracks and controls the execution of each GPU and I/O task.

Overview

Figure 5.3 shows an example of TaroRTL scheduling a task graph using two CPU threads (workers), one GPU stream, and one I/O buffer. Each worker maintains a high-priority queue (HPQ) and a low-priority queue (LPQ). HPQ stores suspended tasks that have been lately executed by the worker, while LPQ stores new tasks that have met task dependency constraints. During scheduling, each worker extracts tasks from its HPQ to LPQ, ensuring that a suspended task is prioritized over a new task. Such prioritization allows efficient caching by ensuring that suspended tasks, which have been recently executed, take precedence over new tasks in the scheduling process. We leverage work-stealing queues [80] to support our scheduling architecture. Only the queue owner [80] can pop/push a task from/into one end of the queue, while multiple workers can steal a task from the other end at the same time. When both of a worker's queues are empty, that worker tries to steal a task from another worker's LPQ to HPQ. This strategy not only balances the workload among the workers, but also reduces the chances of different threads stealing (i.e., resuming) the suspended task. As shown in Figure 5.3, the algorithm follows these steps:

(b) **Enqueue A**: TaroRTL enqueues A into Worker 1's LPQ.

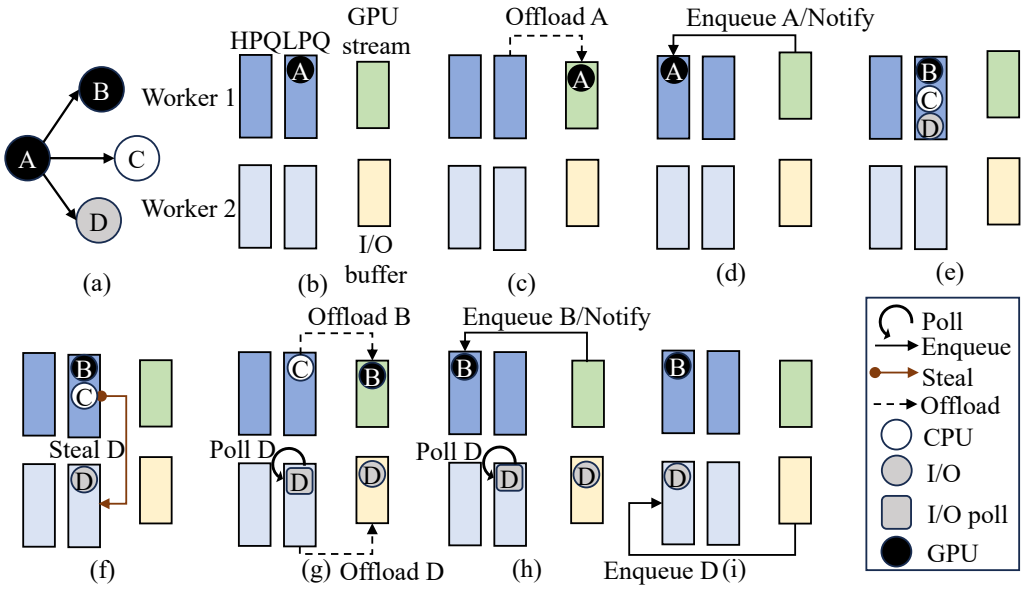


Figure 5.3: TaroRTL schedules a task graph using two CPU workers, one GPU stream, and one I/O buffer. Each worker owns a high-priority task queue (HPQ) and a low-priority task queue (LPQ) to prioritize resuming a suspended task over a new task.

(c) **Offload A and register a CUDA callback**: Worker 1 executes and offloads A into the CUDA stream. Worker 1 then registers a CUDA callback for A.

(d) **Invoke the CUDA callback for A**: After A finishes, CUDA runtime invokes the callback that enqueues A back into Worker 1's HPQ and notifies Worker 1. This strategy ensures Worker 1 resumes A rather than Worker 2, avoiding unnecessary context switches.

(e) **Enqueue B, C, and D**: Worker 1 resumes A for cleanup and resolving task dependencies, and enqueues B, C, and D into its LPQ.

(f) **Steal task from Worker 1**: Worker 2's queues are empty. Worker 2 steals D from Worker 1's LPQ. Since D has not yet started, this steal does not induce context switches.

(g) **Offload B and D**: Worker 1 and 2 offload B and D into CUDA stream and I/O buffer, respectively. Worker 1 registers a CUDA callback for B.

(g) **Poll the status of D**: Worker 2 creates a query task and enqueues the task into its LPQ to poll the execution status of D.

(g) **Multitask to C**: Worker 1 multitasks to C.

(h) **Invoke the CUDA callback for B**: After B finishes, CUDA runtime invokes the callback that enqueues B back into Worker 1's HPQ and notifies Worker 1.

(i) **Resume D**: Worker 2 resumes D for cleanup after verifying that D has finished.

(i) **Continue until complete**: The scheduling process continues until each worker completes its assigned task.

Coroutine-Aware Work Stealing

Conventional work-stealing algorithms cannot be used out of the box due to the distinct performance characteristics between atomic and suspendable executions. For instance, when a suspended task is ready, conventional work-stealing algorithms notify any available CPU threads to resume that task [81, 82, 34, 4]. This strategy may resume a task using different CPU threads, resulting in frequent context switches and cache misses.

Recently, C++20 released a new synchronization primitive of *atomic wait and notify*, which allows a thread to wait on an atomic variable until other threads change its value and notify that thread. This new feature has inspired us to tackle this challenge by assigning each worker an atomic variable to communicate with a specific worker while tracking each worker's state. This approach aligns with the goal of reducing context switches and

cache misses by ensuring that a task is mostly resumed by the same worker. Furthermore, the new atomic features have shown improved performance compared to condition variables, which are commonly used by existing schedulers for synchronization purposes.

Algorithm 5 presents the pseudocode of our work-stealing algorithm for each worker. Each worker has an atomic variable, `state`, with three possible states: BUSY, SIGNALED, and SLEEP. BUSY indicates a worker is actively processing tasks. SIGNALED signifies a worker has been notified by other workers. SLEEP represents a worker who is inactive and waiting for other workers to notify. Initially, each worker waits on the SLEEP (line 2), ensuring its inactivity until scheduled by the scheduler. When the schedule function is called, the scheduler evenly distributes the source tasks to each worker's LPQ. It then changes each worker's state to SIGNALED and notifies them, indicating they are ready to execute tasks.

Once a worker wakes up, it changes its state to BUSY and starts popping tasks from its own HPQ to LPQ (lines 4-9). The worker first attempts to pop a task from its HPQ. Since CUDA runtime or other workers can simultaneously enqueue suspended tasks into HPQ (e.g., Figure 5.3 (d) and (h)), we use `steal` to extract a task from the HPQ that avoids data races. If the HPQ is empty, the worker proceeds to pop a task from its LPQ. The LPQ is managed by the worker, and enqueuing/popping a LPQ by other workers requires a lock [80]. In the event that both of the worker's queues are empty, the worker randomly selects another worker and steals a task from its LPQ to HPQ (lines 10-22). The iteration continues until we successfully steal a task or fail to steal a task after MAX_STEAL times.

To keep track of the overall progress, we maintain an atomic variable, `pending_tasks`, that represents the total number of tasks ready to be invoked. If no tasks are available at a given point, resulting in `pending_tasks` becoming zero, the worker changes its state to SLEEP and checks if its original state has been changed by another worker (lines

35-37). If the worker's state has changed, indicating at least one other worker has changed the state to SIGNALED, the worker continues to work. Otherwise, the worker waits until other workers change its state and notify it.

After a worker invokes a task, there are two situations: 1) The task is complete (Figure 5.3 (e)). 2) The task is suspended (Figure 5.3 (g)). If the task is complete, the worker checks the task's successors and enqueues them into its LPQ if the dependency constraints are met (lines 26-32). On the other hand, if the task is suspended, indicating the worker has invoked GPU or I/O tasks, the worker continues without blocking. Once invoked GPU or I/O tasks are complete, CUDA runtime or a worker will enqueue the suspended task back into the worker's HPQ and notify the worker.

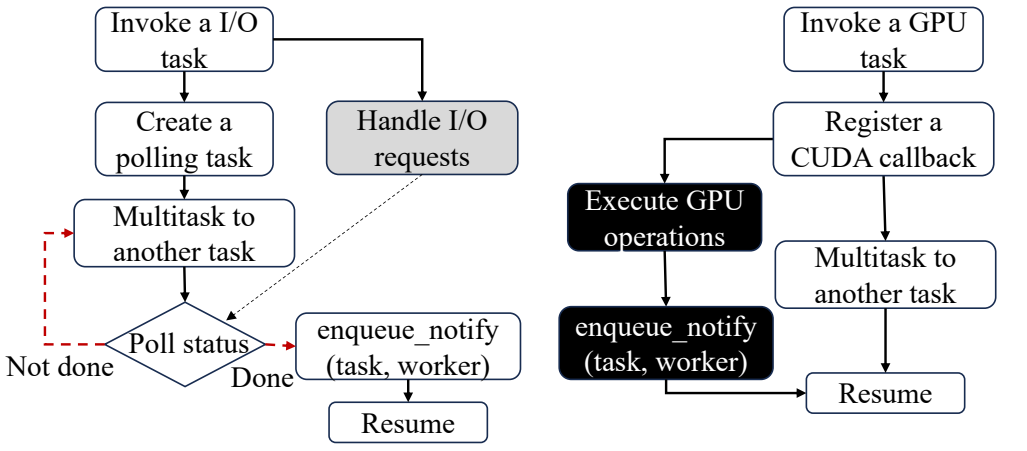
Algorithm 6 outlines how CUDA runtime or a worker enqueues a task and notifies the worker. If *worker* is NULL, it means we want to enqueue a new successor task. The current worker enqueues the task into its LPQ and notifies one available worker (lines 1-14). This strategy allows an available worker to steal a new task from the worker to avoid under-utilization. We iterate through each worker and check if its state is SLEEP using the Compare and Swap (CAS) operation. If a worker's state is SLEEP and its state is successfully changed to SIGNALED (i.e., the CAS operation returns true), it is notified. Otherwise, we iterate to the next worker. The process continues until either a worker is successfully notified or all workers have been checked. If no worker is in SLEEP, we exit the loop without notifying any worker.

If *worker* is not NULL, it means we want to enqueue the task into a specific worker's HPQ and notify that worker (lines 15-21). We require a lock since HPQ may be simultaneously enqueued by other workers. After the task is enqueued, we update the worker's state to SIGNALED, indicating this task is ready to resume. If the original state of the worker was SLEEP, it implies that the worker is inactive and waiting to be notified.

We notify the worker using its state. However, if the worker's state was not SLEEP, implying that the worker is already active, we skip the notification. This organization minimizes unnecessary notification overhead and helps improve overall performance.

Execution Control Strategy

After a GPU or an I/O task finishes, we need to resume the task for cleanup, such as freeing up GPU or I/O resources, releasing memory, and resolving task dependencies. While Coroutine allows for the suspension of a task, it does not automatically resume a suspended task. We need an execution control strategy to track the execution status of a suspended task and trigger its resumption.



(a) I/O task (io_uring) (b) GPU task (cudaLaunchHostFunc)

Figure 5.4: A flowchart of our execution control strategy for (a) I/O and (b) GPU tasks. Gray (Black) blocks represent actions performed by io_uring (CUDA runtime).

Non-blocking I/O tasks

We leverage *io_uring* [83], a new asynchronous I/O framework in Linux that provides efficient and scalable support for asynchronous I/O operations. io_uring implements a ring buffer structure to manage I/O requests. This ring buffer allows for the efficient submission and retrieval of I/O requests without the need for blocking system calls or copies. By incorporating C++ coroutine with io_uring, we are able to submit non-blocking I/O tasks to the ring buffer and seamlessly multitask to a different task.

Figure 5.4 (a) illustrates our strategy for an I/O task. After invoking and suspending an I/O task, the worker creates a polling task and multitasks to another task. The polling task is also a coroutine and can be stolen by other workers to repeatedly check the I/O status. It is a lightweight task that incurs little CPU migration overhead when stolen. Once the status becomes done, the worker that executes this polling task enqueues the suspended task back into the invoked worker's HPQ and notifies that worker.

Non-blocking GPU tasks

Figure 5.4 (b) illustrates our strategy for a GPU task. To probe the execution status of an offloaded GPU task, we utilize CUDA's API, `cudaLaunchHostFunc`, which allows us to register a callback for the offloaded GPU task. The worker then multitasks to other tasks without being blocked. Once the offloaded GPU task is complete, CUDA runtime invokes the callback to enqueue the suspended task back into the worker's HPQ and notify the worker. CUDA callback has a certain cost. In cases where the cost of a GPU task is negligible (e.g., simulate a small design) or CUDA callback is not applicable, we utilize a CUDA event to record the execution status of an offloaded GPU task and poll the status, similar to non-blocking I/O tasks.

Performance Improvement Analysis

In this section, we analyze the time complexity of TaroRTL. As different designs have different task graph structures and task runtimes, it is not practical to analyze the time complexity in a universal manner. Instead, we focus on a more constrained scenario where TaroRTL can achieve the best efficiency over non-coroutine-based approaches. Assuming on a CPU-GPU simulation workload at timeframe P:

- There are n_c CPU threads and n_g GPU streams available.
- There are N identical tasks ready to be executed, where N is larger than n_c and n_g.
- Each task consists of a CPU subtask s_c with a cost of t_c, followed by a GPU subtask s_g with a cost of t_g.

By these assumptions, we can compute a lower bound on the time difference between TaroRTL ($T_{TaroRTL}$) and a scheduler without multitasking (T) at a specific timeframe. In the beginning, all n_c CPU threads simultaneously execute the CPU subtask s_c within n_c tasks, incurring a cost of t_c. Subsequently, all n_g GPU streams execute the GPU subtask s_g within these n_c tasks, resulting in $t_g \cdot \lceil n_c/n_g \rceil$. In TaroRTL, CPUs can multitask to the next n_c CPU subtasks while simultaneously waiting for the GPU to complete the current n_c GPU subtasks. As shown in Figure 5.5, the overlapped time of s_c and s_g is $\min\{t_c, t_g \cdot \lceil n_c/n_g \rceil\}$ per n_c tasks. Specifically, TaroRTL saves at least $\min\{t_c, t_g \cdot \lceil n_c/n_g \rceil\}$ per n_c tasks. With $\lceil N/n_c \rceil - 1$ times,

$$T - T_{TaroRTL} \geqslant (\left\lceil \frac{N}{n_c} \right\rceil - 1) \cdot \min\{t_c, t_g \cdot \left\lceil \frac{n_c}{n_g} \right\rceil\}.$$

The time difference expands as the number of tasks increases or the CPU and GPU subtasks' cost becomes larger.

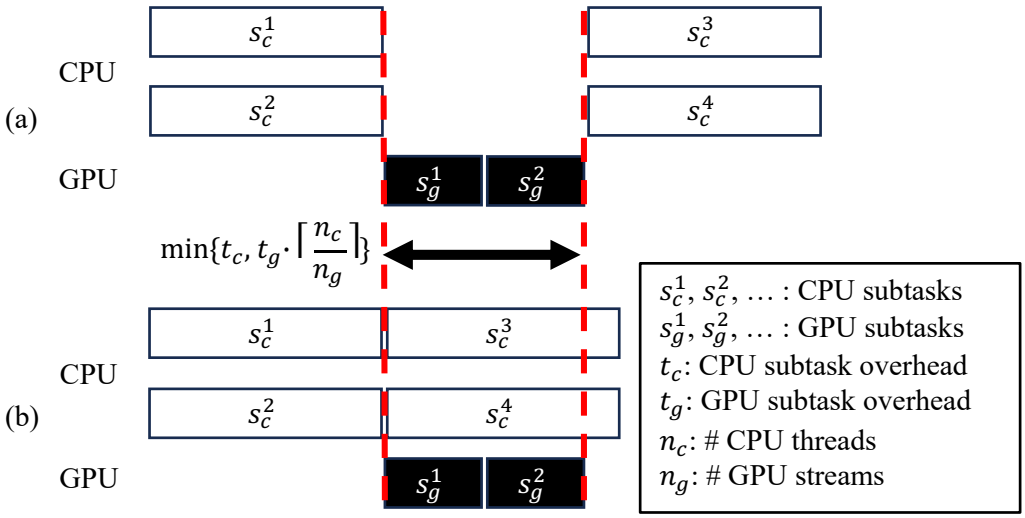


Figure 5.5: The time difference between (a) a scheduler without coroutine and (b) TaroRTL at a specific timeframe. In this example, n_c is 2, n_g is 1, and $t_c > t_g \cdot \left\lceil \frac{n_c}{n_g} \right\rceil$.

5.5 Experimental Results

We evaluate the performance of TaroRTL on 1) three industrial designs: Spinal, riscv-mini, and NVDLA, and 2) two micro-benchmarks: Divide and Conquer (DC) and Wavefront (WF) [72]. We conducted our experiments on a 3.2 GHz 64-bit Linux machine with one NVIDIA RTX 3080 ti GPU and ten Intel i9-12900KF CPU cores. We compiled our programs with CUDA NVCC 12.1 on a GCC 12.1 host compiler and enabled optimization flag `-O2` and `-std=c++20`. We use an equal number of CUDA streams and CPU threads. All data is an average of ten runs. TaroRTL adopts heap allocation by default to store the stack of a C++ coroutine.

CPU-GPU RTL Simulation

In this section, we analyze the performance benefits of TaroRTL with non-blocking GPU tasks. We consider RTLflow as our baseline. RTLflow [4]

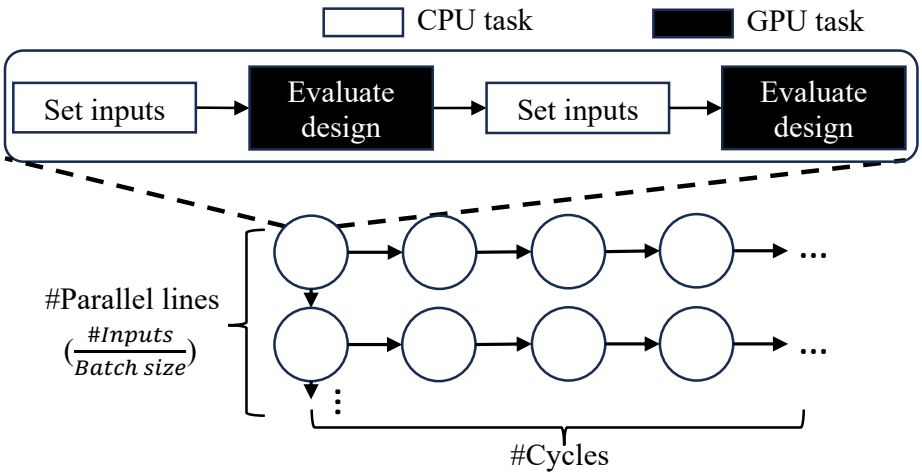


Figure 5.6: The heterogeneous RTL task graph in RTLflow. Each task contains two CPU and GPU subtasks.

improves the throughput performance by running multiple input stimuli simultaneously using both CPU and GPU parallelisms. As shown in Figure 5.6, RTLflow describes the RTL simulation workload as a heterogeneous task graph, where each task consists of four dependent CPU/GPU subtasks. The size of the task graph is determined by the number of inputs, the number of simulation cycles, and the chosen batch size. RTLflow splits inputs into batches to allow more parallelism (i.e., more parallel lines). For our experiments, we fixed the default batch size of 1024. However, RTLflow lacks support for multitasking, resulting in a CPU thread waiting for GPU tasks to finish. By scheduling RTLflow's heterogeneous task graph using TaroRTL, we are able to improve total runtime while using fewer CPU resources.

Table 5.1 compares overall runtime between RTLflow and TaroRTL on riscv-mini and NVDLA designs using different numbers of threads. TaroRTL outperforms RTLflow in almost all scenarios. TaroRTL achieves at least 1.4× speed-up and 1.6× speed-up on riscv-mini and NVDLA designs. We can clearly see the proposed coroutine-based task graph scheduling

brings significant performance benefits to the CPU-GPU RTL simulation workload. Compared to RTLflow, TaroRTL achieves 1.8× speed-up on NVDLA. In the case of the Spinal design, RTLflow and TaroRTL exhibit similar runtimes, and the advantage of coroutines is less pronounced compared to other designs. When SLOC (lines of setting inputs) is small, where the CPU overhead is extremely small, TaroRTL does not benefit much from multitasking. However, modern designs typically have many thousands of ELOC and hundreds of SLOC [60] where TaroRTL can stand out.

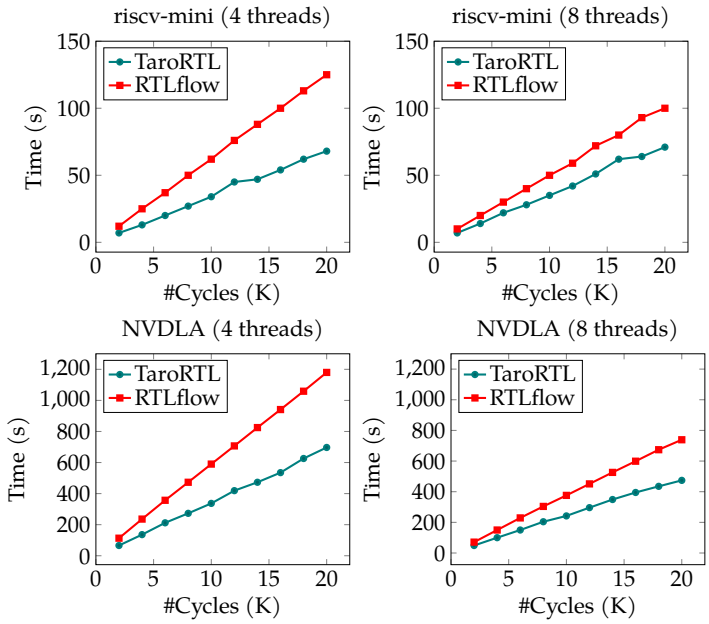


Figure 5.7: Runtime growth over increasing numbers of cycles for TaroRTL and RTLflow using four and eight threads.

Figure 5.7 shows the runtime growth over increasing numbers of cycles for TaroRTL and RTLflow on different designs using four and eight threads. TaroRTL outperforms RTLflow in all scenarios. Compared to RTLflow, TaroRTL using four threads achieves 1.8× speed-up for riscv-mini design at 20K cycles. The significant improvement on runtime demonstrates the

promise of our multitasking techniques. We can also clearly see the results are aligned with the speedup analysis. For example, the performance gap between TaroRTL and RTLflow continues to enlarge as we increase the number of cycles (i.e., the number of tasks N).

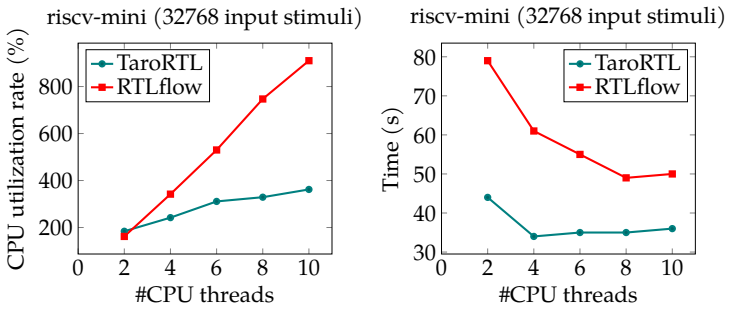


Figure 5.8: Average CPU utilization rate reported by `/usr/bin/time` and runtime decrease over increasing numbers of CPU threads for TaroRTL and RTLflow on the riscv-mini design.

Figure 5.8 shows the CPU utilization rate and runtime decrease over increasing numbers of threads on the riscv-mini design. TaroRTL using 362% CPU achieves 1.4× faster than RTLflow using 910% CPU. riscv-mini is a midsize design that does not induce large CPU computation. However, RTLflow requires CPUs to keep spinning until GPU finishes its operations, resulting in an unnecessarily high CPU utilization rate.

RTL Simulation with I/O

In this section, we study the performance benefits of RTL simulation with non-blocking I/O tasks. We consider Verilator, which supports VCD file dumping, as our baseline. Verilator is a single-stimulus simulator. For multi-stimulus simulation, the de facto way is to create multiple instances of Verilator and run independent input stimulus in parallel [4]. After an evaluation of the design, each Verilator stores traces in a buffer and dumps the buffer to a file once it is full. Since Verilator does not support multitasking, it requires a CPU thread to wait until I/O dumping finishes.

By enabling our non-blocking I/O using TaroRTL, we are able to improve the simulation efficiency.

Figure 5.9 illustrates the achieved speed-up by TaroRTL over Verilator at different numbers of input stimuli using eight threads on riscv-mini. When the number of input stimuli equals 8, Verilator is faster due to the limited parallelism available for eight threads. However, as the number of input stimuli exceeds 8, where parallelism becomes more abundant (i.e., more independent tasks), TaroRTL starts to outperform Verilator. Since RTL simulation typically involves many input stimuli on the same design [60, 4], TaroRTL's ability to handle larger parallelism provides a significant advantage.

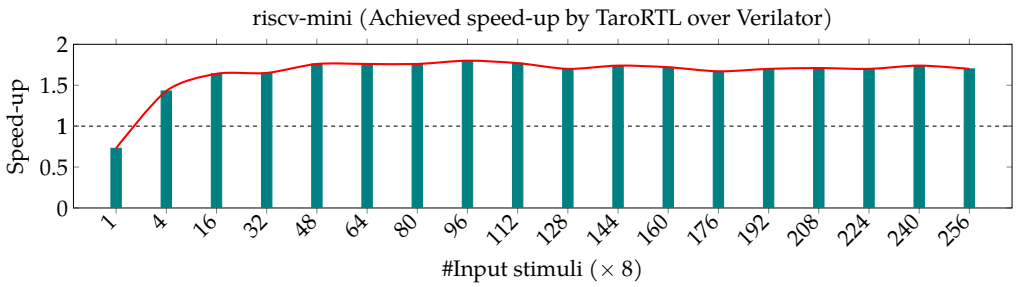


Figure 5.9: Achieved speed-up by TaroRTL over Verilator at different numbers of input stimuli using eight threads for 3K cycles.

Efficiency of Coroutine-Aware Work Stealing

We finally study the efficiency of our coroutine-aware work stealing to demonstrate its potential capability beyond RTL simulation. We consider two state-of-the-art task schedulers, Boost Fiber [84] and Taskflow [34], that support CPU-GPU scheduling as our baselines. Boost Fiber is a C++ library that utilizes a cooperative scheduling mechanism to achieve multi-tasking. They provide fiber, a lightweight thread that can be suspended and resumed independently. However, when fibers are migrated between

different CPUs, there is an inherent CPU migration overhead. As a result, fibers undergo frequent migration, and the aggregated overhead becomes significant. Taskflow is a task graph scheduling system that has been adopted by many EDA algorithms, including RTLflow [4]. However, it does not support multitasking in a task graph. We consider two common task graphs as our micro-benchmarks. Divide and Conquer (DC) defines a complete binary tree, and Wavefront (WF) defines a wavefront that propagates task dependencies from the top-left task all the way to the bottom-right task. Both micro-benchmarks are representative of modern data-driven task graphs such as iterative algorithms, data processing pipelines, and simulation [34]. In our task graph, each task consists of 300 repeated CPU and GPU subtasks, where each GPU subtask generates a random number and adds it to each element in a 10K-element vector, and each CPU subtask sorts the vector.

Table 5.2 compares various statistics among Taskflow, Boost Fiber, and TaroRTL on WF with 10K tasks using eight threads. The results clearly demonstrate the performance advantage of our algorithm. TaroRTL exhibits the lowest number of context switches, CPU migrations, and cache misses. Without our coroutine-aware work stealing, Boost fiber struggles to maintain task continuity on the same thread, leading to significant CPU migration overhead. Such overhead outweighs the performance benefit derived from multitasking. Figure 5.10 compares runtime growth over increasing graph size among Taskflow, Fiber, and TaroRTL using eight threads. TaroRTL outperforms all other schedulers in all scenarios. Compared to Fiber and Taskflow, TaroRTL achieves 4.6× and 3.9× speed-up on DC with 10K tasks. The gap continues to enlarge as we increase the graph size.

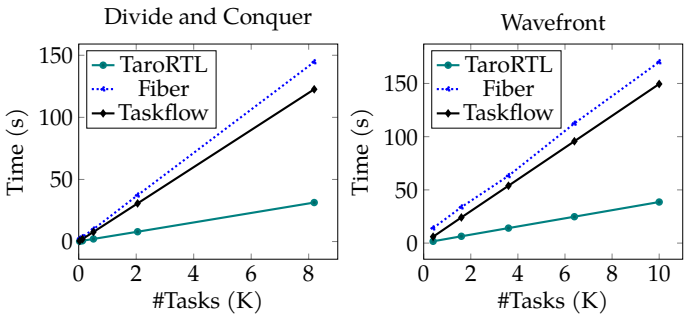


Figure 5.10: Runtime comparison among Taskflow, Fiber, and TaroRTL for DC and WF task graphs using eight threads.

5.6 Conclusion

In this chapter, we have introduced TaroRTL, a coroutine-based task graph scheduler for RTL simulation. TaroRTL has introduced a coroutine-based task graph scheduling model to enable multitasking in a task graph. TaroRTL has also introduced a coroutine-aware work-stealing algorithm to reduce unnecessary context switches. Compared to RTLflow, TaroRTL can further achieve 40–80% speed-up while using fewer CPU resources to simulate large industrial designs. In this work, Dian-Lun Lin was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the project. All authors participated in discussing the results and contributed to the preparation and review of the final manuscript.

Algorithm 5: Coroutine-aware work-stealing algorithm

```
1  worker ← this_worker();/* current worker                    */
2  worker.state.wait(SLEEP);
3  do
4      worker.state.store(BUSY);
5      do
           /* get from worker's own HPQ to LPQ                 */
6          task ← worker.HPQ.steal();
7          if task == NULL then
8              task ← worker.LPQ.pop();
9          end
           /* steal from another worker's LPQ to HPQ           */
10         if task == NULL then
11             cnt ← 0;
12             while cnt++ < MAX_STEAL do
13                 aworker ← random_select();
14                 task ← aworker.LPQ.steal();
15                 if task == NULL then
16                     task ← aworker.HPQ.steal();
17                 end
18                 if task ! = NULL then
19                     break;
20                 end
21             end
22         end
           /* invoke the task and enqueue successors           */
23         if task ! = NULL then
24             pending_tasks.fetch_sub();
25             invoke(task);
26             if task.is_done() then
27                 for succ : task.successors do
28                     if succ.dependency.is_met() then
29                         enqueue_notify(succ, NULL);
30                     end
31                 end
32             end
33         end
34     while pending_tasks.load() > 0;
35     if worker.state.exchange(SLEEP) == BUSY then
           /* wait to be notified                              */
36         worker.state.wait(SLEEP);
37     end
38 while !stop;
```

Algorithm 6: enqueue_notify(task, worker)

```
   Input: task: a suspended task to be enqueued
   Input: worker: a worker to be notified
 1 if worker == NULL then
       /* enqueue to current worker's own LPQ                    */
 2     worker ← this_worker();
 3     worker.LPQ.push(task);
 4     pending_tasks.fetch_add();
       /* notify one SLEEP worker                                */
 5     cnt ← 1;
 6     do
 7         idx ← (worker.idx + cnt)%NUM_WORKERS;
 8         tmp ← SLEEP;
 9         if workers[idx].state.CAS(tmp, SIGNALED) then
10             workers[idx].state.notify_one();
11             return;
12         end
13     while ++cnt < NUM_WORKERS;
14 end
15 else
       /* enqueue and notify a specific worker                   */
16     lock{worker.HPQ.push(task)};
17     pending_tasks.fetch_add();
18     if worker.state.exchange(SIGNALED) == SLEEP then
19         worker.state.notify_one();
20     end
21 end
```

Design	ELOC	SLOC	#Threads	RTLflow	TaroRTL	Speed-up
Spinal	9654	6	2	36s	**35s**	2.9%
			4	**20s**	21s	-
			6	**16s**	17s	-
			8	**11s**	14s	-
			10	**11s**	14s	-
riscv-mini	10935	340	2	79s	**44s**	79.5%
			4	61s	**34s**	79.4%
			6	55s	**35s**	57.1%
			8	49s	**35s**	40.0%
			10	50s	**36s**	38.9%
NVDLA	560412	860	2	1082s	**598s**	80.9%
			4	600s	**337s**	78.0%
			6	482s	**284s**	69.7%
			8	376s	**242s**	55.4%
			10	379s	**236s**	60.6%

Table 5.1: Comparison between RTLflow and TaroRTL on Spinal, riscv-mini, and NVDLA designs using different numbers of threads for completing 32768 input stimuli. ELOC and SLOC represent lines of code for evaluation and lines of code for setting inputs, respectively. Bold texts represent the best results. All simulation results match the golden reference provided by RTLflow.

Statistics	Taskflow	Fiber	TaroRTL
context switches	38.7M	14.2M	**9.3M**
CPU migrations	63.4K	21.6K	**13.7K**
L1-dcache-load-misses	19.0B	9.0B	**7.5B**
L1-icache-load-misses	60.1B	24.9B	**16.9B**
LLC-load-misses	36.0M	11.0M	**8.2M**
LLC-store-misses	29.1M	6.3M	**5.7M**
Runtime	149.5s	170.2s	**38.6s**

Table 5.2: Comparison among Taskflow, Boost Fiber, and TaroRTL on a WF task graph with 10K nodes using eight threads.

6 CONCLUSION

In conclusion, this thesis presents several significant advancements in the field of heterogeneous computing and logic simulation through novel algorithms and frameworks designed to enhance performance and efficiency. The key contributions and findings of this research can be summarized as follows:

1. *SNIG: A Novel Inference Algorithm for Large Sparse DNNs.* We have introduced SNIG, an efficient inference engine for large sparse deep neural networks. We have demonstrated that SNIG's task graph decomposition can scale to various sizes of sparse DNNs and input data, achieving a 2.3× runtime speed-up over existing methods by avoiding unnecessary computations and synchronization overheads.

2. *cudaFlow: Efficient GPU Computation using Task Graph Parallelism*. We have proposed a lightweight task graph programming framework to simplify the development of GPU applications using CUDA graphs. We have showed that cudaFlow and cudaFlowCapturer can achieve performance comparable to optimally constructed CUDA graphs while reducing the complexity of graph management for users.

3. *RTLflow: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus*. We have developed RTLflow, a framework for accelerating RTL simulation using GPUs. We have demonstrated a 40× runtime speed-up in RTL simulation by leveraging GPU parallelism and efficient task scheduling techniques.

4. *GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs*. We have introduced GenFuzz, a hardware fuzzer that employs a genetic algorithm to enhance the efficiency of hardware fuzzing. We have achieved up to an 80× speed-up in runtime

compared to existing fuzzers, showcasing the effectiveness of the genetic algorithm and multi-input approach in identifying hardware bugs more efficiently.

5. *TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling*. We have proposed TaroRTL, a coroutine-based task graph scheduler designed to optimize RTL simulation. We have showed that TaroRTL reduces context switches, CPU migrations, and cache misses, resulting in a 70% speed-up over traditional scheduling methods while using fewer CPU resources.

These contributions collectively advance the state of the art in heterogeneous computing and hardware simulation, providing practical frameworks and algorithms that can be adopted in various high-performance computing applications. The research emphasizes the importance of efficient task graph management, parallelism, and innovative algorithm design in achieving superior computational performance.

Future work should investigate the applicability of SNIG to other machine learning models, support more complex workflows in cudaFlow with dynamic task dependencies and real-time data processing, explore the scalability of RTLflow for larger and more complex hardware designs, incorporate advanced genetic algorithms and machine learning techniques into GenFuzz to further enhance its efficiency and coverage capabilities, and generalize TaroRTL into a generic library with a new programming model that can be easily integrated with various applications. By pursuing these directions, the frameworks and algorithms developed in this thesis can be further expanded and refined, contributing to ongoing advancements in heterogeneous computing and hardware simulation technologies.

BIBLIOGRAPHY

[1] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," *IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.

[2] D.-L. Lin and T.-W. Huang, "A novel inference algorithm for large sparse neural network using task graph parallelism," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[3] "NVIDIA Nsight Systems," "https://developer.nvidia.com/nsight-systems.

[4] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus," in *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*, 2022, pp. 1–12.

[5] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149*, 2015.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5MB model size," *arXiv:1602.07360*, 2016.

[7] M. Bisson and M. Fatica, "A GPU implementation of the sparse deep neural network graph challenge," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–8.

[8] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Advances in neural information processing systems (NIPS)*, 2019, pp. 103–112.

[9] C. Yu, S. Royuela, and E. Quiñones, "OpenMP to CUDA Graphs: A compiler-based transformation to enhance the programmability of nvidia devices," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, 2020, p. 42–47.

[10] B. Qiao, M. Akif Özkan, J. Teich, and F. Hannig, "The Best of Both Worlds: Combining CUDA Graph with an image processing DSL," in *57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[11] T. Blattner, W. Keyrouz, S. S. Bhattacharyya, M. Halem, and M. Brady, "A hybrid task graph scheduler for high performance image processing workflows," *J. Signal Process. Syst.*, p. 457–467, 2017.

[12] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 974–983.

[13] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A modern c++ parallel task programming library," in *ACM Multimedia Conference*, 2019, p. 2284–2287.

[14] T.-W. Huang, "A general-purpose parallel and heterogeneous task programming system for VLSI CAD," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020, pp. 1–2.

[15] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-taskflow: A general-purpose parallel task programming system at scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 40, no. 8, pp. 1687–1700, 2020.

[16] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A general-purpose parallel and heterogeneous task programming system,"

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (*TCAD*), vol. 41, no. 5, pp. 1448–1452, 2021.

[17] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "An efficient and composable parallel task programming library," in *IEEE High Performance Extreme Computing* (*HPEC*), 2019, pp. 1–7.

[18] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," in *Journal of parallel and distributed computing*, vol. 74, no. 12, 2014, pp. 3202–3216.

[19] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[20] W. Snyder, "Verilator 4.0: open simulation goes multithreaded," https://veripool.org/papers/Verilator_v4_Multithreaded_OrConf2018.pdf, 2018.

[21] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate RTL simulation," in *57th ACM/IEEE Design Automation Conference* (*DAC*), 2020, pp. 1–6.

[22] C. X. Wolf, "Yosys," https://yosyshq.net/yosys/, 2012.

[23] H. Qian and Y. Deng, "Accelerating RTL simulation with GPUs," in *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2011, pp. 687–693.

[24] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with GPUs," in *ACM Transactions on Design Automation of Electronic Systems* (*TODAES*), vol. 16, no. 3, 2011, pp. 1–20.

[25] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with GPU computing," in *ACM Transactions on Design Automation of Electronic Systems* (*TODAES*), vol. 16, no. 3, 2011, pp. 1–26.

[26] Uri Tal, "RocketSim: A GPU-based simulation accelerator for chip verification," https://on-demand-gtc.gputechconf.com/gtcnew/speakerName.php?speaker=Uri+Tal, 2013.

[27] Y. Zhang, H. Ren, A. Sridharan, and B. Khailany, "GATSPI: GPU accelerated gate-level simulation for power improvement," in *Proceedings of the 59th ACM/IEEE Design Automation Conference* (*DAC*), 2022, pp. 1231–1236.

[28] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2018, pp. 1–8.

[29] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *ACM/IEEE Design Automation Conference* (*DAC*), 2021.

[30] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DIFUZZRTL: Differential fuzz testing to find CPU bugs," in *IEEE Symposium on Security and Privacy* (*SP*), 2021, pp. 1286–1303.

[31] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *31st USENIX Security Symposium* (*USENIX Security*), 2022, pp. 3219–3236.

[32] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security)*, 2022, pp. 3237–3254.

[33] H. Wang and S. Beamer, "RepCut: Superlinear parallel RTL simulation with replication-aided partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 3*, 2023, pp. 572–585.

[34] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 6, 2021, pp. 1303–1320.

[35] W. L. Lee, D.-L. Lin, T.-W. Huang, S. Jiang, T.-Y. Ho, Y. Lin, and B. Yu, "G-kway: Multilevel GPU-accelerated k-way graph partitioner," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[36] B. Zhang, L. Dian-Lun, C. Chang, C.-H. Chiu, B. Wang, L. W. Luan, C. Chih-Chun, F. Donghao, and H. Tsung-Wei, "G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[37] C. Chang, T.-W. Huang, D.-L. Lin, S. Lin, and G. Guo, "Ink: Efficient k-critical path generation," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[38] C. J. Williams and J. Elliott, "Libfork: portable continuation-stealing with stackless coroutines," *arXiv preprint arXiv:2402.18480*, 2024.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American*

Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019, pp. 4171–4186.

[40] B. McCann, J. Bradbury, C. Xiong, and R. Socher, "Learned in Translation: Contextualized word vectors," in *Advances in neural information processing systems (NIPS)*, vol. 30, 2017.

[41] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *OpenAI blog*, vol. 1, no. 8, 2018, p. 9.

[42] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi, "Sparse deep neural network exact solutions," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2018, pp. 1–8.

[43] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep learning for supercomputers," in *Advances in neural information processing systems (NIPS)*, 2018, pp. 10 414–10 423.

[44] M. Grossman, C. Thiele, M. Araya-Polo, F. Frank, F. O. Alpak, and V. Sarkar, "A survey of sparse matrix-vector multiplication performance on large matrices," *arXiv preprint arXiv:1608.00636*, 2016.

[45] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[46] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ACM International Symposium on Computer Architecture (ISCA)*, 2017, p. 27–40.

[47] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient DNN inference on GPU," in *Proceedings of the AAAI conference on artificial intelligence* (*AAAI*), vol. 33, no. 01, 2019, pp. 5676–5683.

[48] J. A. Ellis and S. Rajamanickam, "Scalable inference for sparse deep neural networks using kokkos kernels," in *IEEE High Performance Extreme Computing Conference* (*HPEC*), 2019, pp. 1–7.

[49] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks." vol. 2279, p. 2288, 2018.

[50] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[51] M. Wang, C.-c. Huang, and J. Li, "Unifying data, model and hybrid parallelism in deep learning via tensor tiling," *arXiv preprint arXiv:1805.04170*, 2018.

[52] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM symposium on operating systems principles* (*SOSP*), 2019, pp. 1–15.

[53] A. Petrowski, G. Dreyfus, and C. Girault, "Performance analysis of a pipelined backpropagation parallel algorithm," *IEEE Transactions on Neural Networks*, vol. 4, no. 6, pp. 970–981, 1993.

[54] "CUDA Graph," https://devblogs.nvidia.com/cuda-graphs/.

[55] J. Kepner and R. Robinett, "Radix-net: Structured sparse matrices for deep neural networks," in *IEEE international parallel and distributed processing symposium workshops* (*IPDPSW*), 2019, pp. 268–274.

[56] "NVIDIA Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler.

[57] "Effortless CUDA Graphs," https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32082/.

[58] "CUDA Graph in TensorFlow," "https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31312/".

[59] "NVIDIA CUDA Graph example," "https://github.com/NVIDIA/cuda-samples/blob/master/Samples/simpleCudaGraphs/simpleCudaGraphs.cu.

[60] Y. Zhang, H. Ren, and B. Khailany, "Opportunities for RTL and gate level simulation using GPUs," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–5.

[61] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers Inc., 2009.

[62] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *2011 Design, Automation Test in Europe*, 2011, pp. 1–6.

[63] "NVIDIA Deep Learning Accelerator Design (NVDLA)," "http://nvdla.org/.

[64] V. Sarkar, "Partitioning and scheduling parallel programs for execution on multiprocessors," Ph.D. dissertation, Stanford University, 1987.

[65] C.-H. Chiu and T.-W. Huang, "Efficient timing propagation with simultaneous structural and pipeline parallelisms," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1388–1389.

[66] W. R. Gilks, S. Richardson, and D. Spiegelhalter, in *Markov chain Monte Carlo in practice*. CRC press, 1995.

[67] W. K. Hastings, in *Monte Carlo sampling methods using Markov chains and their applications*. Oxford University Press, 1970.

[68] "Spinal," "https://github.com/SpinalHDL/VexRiscv.

[69] "riscv-mini," "https://github.com/ucb-bar/riscv-mini.

[70] C.-H. Chiu and T.-W. Huang, "Composing pipeline parallelism using control Taskflow graph," in *ACM HPDC*, 2022, pp. 283—284.

[71] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *IEEE 26th international conference on parallel and distributed systems* (*ICPADS*), 2020, pp. 64–71.

[72] D.-L. Lin and T.-W. Huang, "Efficient GPU computation using task graph parallelism," in *27th International Conference on Parallel and Distributed Computing* (*Euro-Par*), 2021, pp. 435–450.

[73] "NVIDIA System Management Interface," "https://developer.nvidia.com/nvidia-system-management-interface.

[74] K. Tan, C. Goh, Y. Yang, and T. Lee, "Evolving better population distribution and exploration in evolutionary multi-objective optimization," *European Journal of Operational Research*, 2006.

[75] D. S. Hochba, "Approximation algorithms for NP-Hard problems," *SIGACT News*, 1997.

[76] "Boom issue page," https://github.com/riscv-boom/riscv-boom/issues.

[77] D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, "Accelerated analysis of simulation dumps through parallelization on multicore architectures," in *24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2021, pp. 69–74.

[78] "C++ Coroutine." https://en.cppreference.com/w/cpp/language/coroutines.

[79] D.-L. Lin and T.-W. Huang, "Accelerating large sparse neural network inference using GPU task graph parallelism," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 11, 2022, pp. 3041–3052.

[80] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models," in *ACM SIGPLAN Notices*, vol. 48, no. 8, 2013, pp. 69–80.

[81] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 1998, pp. 119–129.

[82] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 1299–1308.

[83] "Efficient io with io_uring," https://kernel.dk/io_uring.pdf.

[84] "Boost fiber," https://www.boost.org/doc/libs/1_80_0/libs/fiber/doc/html/index.html.