From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus

Dian-Lun Lin
dian-lun.lin@utah.edu
University of Utah, USA
Salt Lake City, UT, USA

Haoxing Ren
haoxingr@nvidia.com
Nvidia Corporation
Austin, TX, USA

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

Brucek Khailany
bkhailany@nvidia.com
Nvidia Corporation
Austin, TX, USA

Tsung-Wei Huang
tsung-wei.huang@utah.edu
University of Utah
Salt Lake City, UT, USA

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 paper, 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.

ACM Reference Format:
Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545091

1 INTRODUCTION

Register-transfer level (RTL) simulation is a critical part of designing and verifying today's highly complex SoCs, processors, and accelerators [30, 31]. It is widely used in logic design, directed verification, constrained random verification, performance verification, and debugging. For functional verification signoff [25], 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.

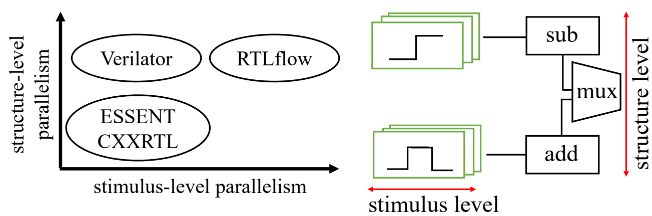


Figure 1: RTLflow explores both stimulus- and structure-level parallelisms to achieve high-performance RTL simulation using GPU computing.

In recent years, we have seen increasing interest in accelerating RTL simulation in open-source tools, as shown in Figure 1. Verilator [28] 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 [9] 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 [2] 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.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545091

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, [26] 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. [10, 29, 32, 33] 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 paper, we propose RTLflow, a novel GPU acceleration flow to speed up simultaneous multi-stimulus RTL simulation. As shown in Figure 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 [28] and ESSENT [9]. 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 [4] 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 [8] to benefit the community and inspire software simulation research with heterogeneous parallelism.

2 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 [31].

2.1 Conventional RTL Simulation Techniques

RTL simulation typically transpiles the given Verilog to C++ and lets a compiler optimize the simulation code [2, 9, 28]. Listing 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 1: A transpiled C++ loop for RTL simulation.

2.2 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.

2.3 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 [28]. 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 [27] 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 [28]. 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 [9]. 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 [28], which is far more than the speed-up report of ESSENT in [9]. 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.

2.4 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:

2.4.1 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 domain [2, 28], 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.

2.4.2 Lack of a GPU-aware RTL Partitioning Algorithm. Existing full-cycle RTL simulators [2, 28] 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.

2.4.3 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 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 stimulus, this sequential computation can incur expensive waiting time between CPU and GPU. Figure 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 1, becomes the primarily bottleneck. To overcome this problem, we need an efficient scheduling method to overlap CPU and GPU tasks across simulation loops.

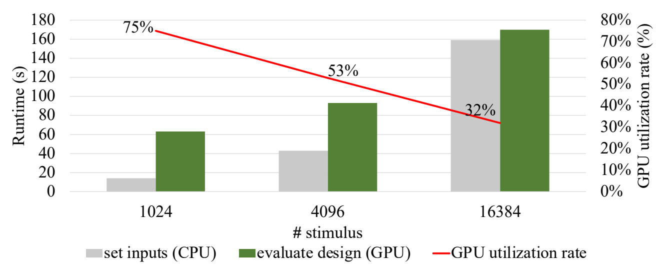


Figure 2: Runtime breakdown of a simulation benchmark in terms of setting inputs, evaluating the design, and the corresponding GPU utilization rate under different numbers of stimulus.

3 RTLFLOW

Figure 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 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 [7], 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 [12] to explore inter-stimulus parallelism across simulation iterations.

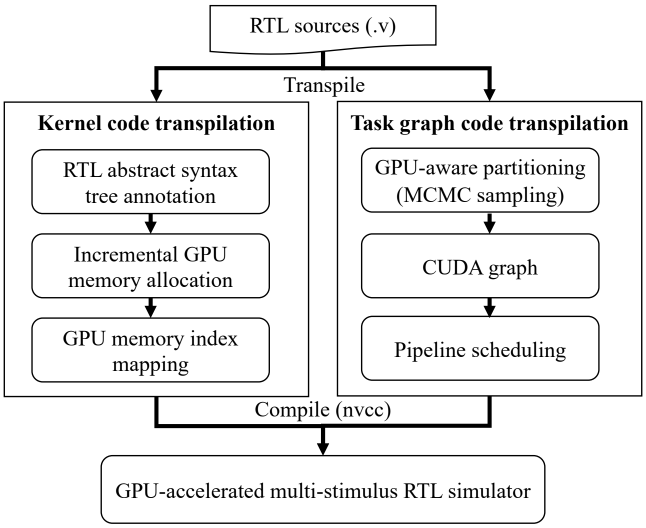


Figure 3: Overview of RTLflow.

3.1 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 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 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

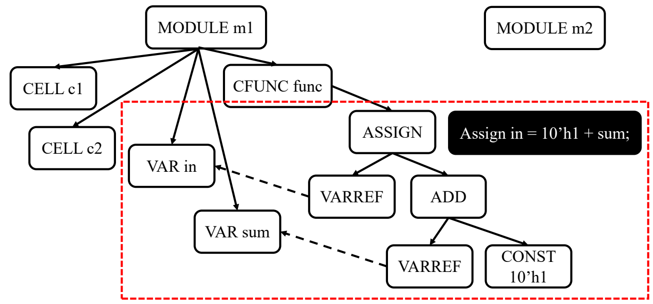


Figure 4: An RTL AST that consists of two modules (`m1` and `m2`). `m1` contains two cells (`c1` and `c2`), two variables (`in` and `sum`), and one function (`func`). The RTL AST requires seven AST nodes to describe one line of Verilog code (the assignment statement in black).

most important strategies that address the transpilation challenge, using Figure 4 as an example.

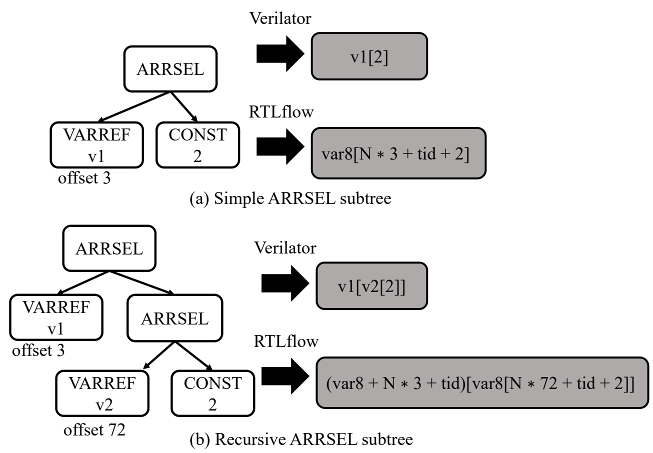


Figure 5: (a) Simple ARRSEL subtree and (b) Recursive ARRSEL subtree. Right part shows generated C++/CUDA code using Verilator or RTLflow.

3.1.1 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 5 shows the simple and recursive subtrees each rooted at a ARRSEL AST node that is responsible for generating variable 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__`.

3.1.2 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 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.

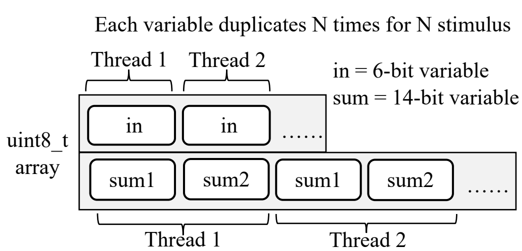


Figure 6: GPU memory allocation using one fixed-width memory array of type `uint8_t`. `in` is a 6-bit variable, and `sum` is a 14-bit variable stored into two memory locations, `sum1` and `sum2`.

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.

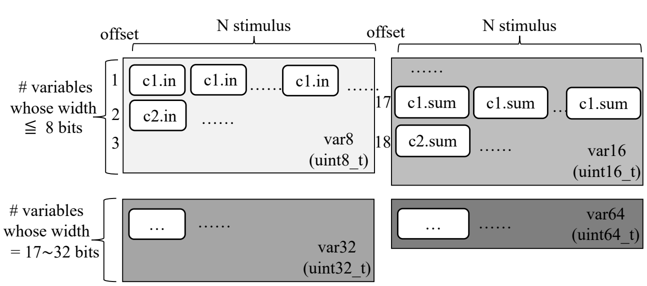


Figure 7: GPU memory allocation for Figure 4. Each cell (`c1` and `c2`) contains two variables (`in` and `sum`). A variable is stored in the smallest array of types `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` that fits the variable width.

Figure 7 shows the memory allocation results of our strategy based on Figure 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.

3.1.3 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 2 shows Verilator's transpiled C++ code using the partial RTL AST shown in Figure 4. As opposed to Listing 2, Listing 3 shows the transpiled CUDA code by RTLflow based on the offsets shown in Figure 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 2: Transpiled C++ simulation pseudocode of Figure 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: Transpiled CUDA kernel pseudocode of Figure 4 using the GPU memory offsets in Figure 7.

3.2 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.

3.2.1 GPU-aware Partitioning. A common RTL graph partitioning algorithm iteratively merges two nodes into a task using static, hard-coded cost estimates [27, 28]. 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.

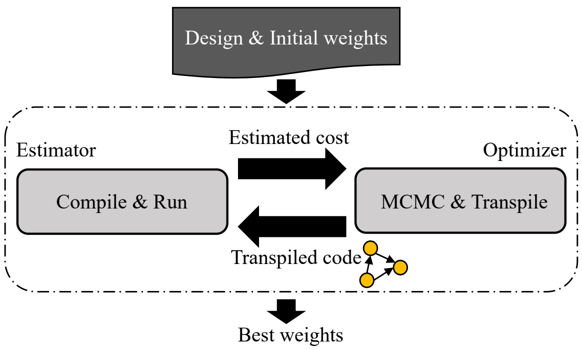


Figure 8: GPU-aware partitioning algorithm using MCMC to explore the best combination of weights under real operating conditions (compile + run).

Figure 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` from the previous iteration, defined below:

$$weight\_sum(task) = \sum_{t \in T} w_t * N_t \tag{1}$$

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 [13, 16]. The probability distribution is the following:

$$p(\mathcal{G}) \propto exp(-\beta * cost(\mathcal{G})) \tag{2}$$

where $\mathcal{G}$ is a GPU task graph, $cost(\mathcal{G})$ is the estimated cost of $\mathcal{G}$ from the estimator, and β is a constant that can be chosen. We use Metropolis-Hastings algorithm [16] 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:

$$\begin{aligned} \alpha(\mathcal{G} \rightarrow \mathcal{G}^*) &= min(1, p(\mathcal{G}^*)/p(\mathcal{G})) \\ &= min(1, exp(\beta * (cost(\mathcal{G}) - cost(\mathcal{G}^*)))) \end{aligned} \tag{3}$$

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 1 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.

Algorithm 1: 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
```

3.2.2 *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 it repeats the same stream and event management on the same CUDA task graph over all simulation cycles. The new CUDA Graph execution model [7] 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 9 and 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.

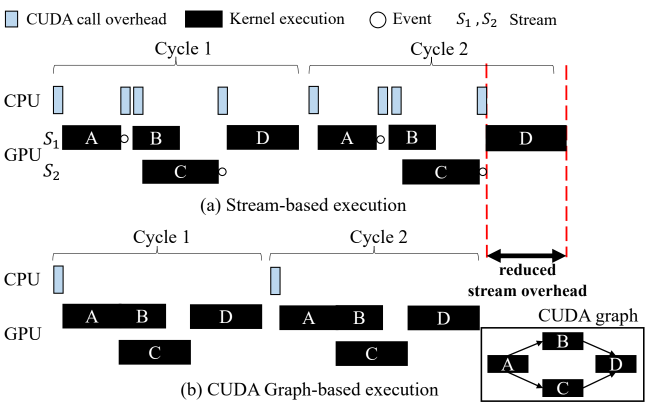


Figure 9: Stream-based execution versus CUDA Graph-based execution of the CUDA graph for two cycles. Stream-based execution incurs repetitive CUDA call overheads to schedule dependent kernels at each cycle.

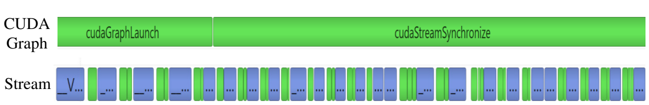


Figure 10: Partial simulation timeline of CUDA Graph-based execution and stream-based execution using the data extracted from Nvidia Nsight Systems [5]. Blue bars represent calls to launch CUDA kernels and green bars represent CUDA synchronization calls.

3.2.3 Pipeline Scheduling Algorithm. As shown in Listing 1 and Figure 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 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 11. At stage 2, we pass G_2 into our pipeline. We then simulate

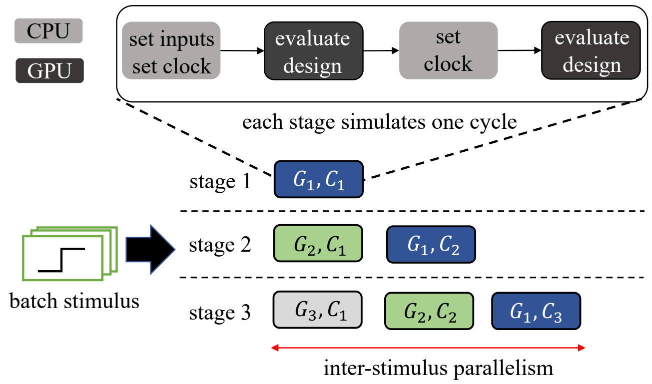


Figure 11: The proposed pipeline scheduling algorithm to enable efficient overlap between CPU and GPU tasks.

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 `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).

4 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 [4]. riscv-mini and Spinal are both RISC-V CPU projects [3, 6]. Table 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 [11, 18, 19] and its work-stealing runtime [22] 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

Design	Verilog LOC	#AST nodes	Verilator LOC	Verilator CC_{avg}	Verilator #Tokens	Verilator T_{trans}	RTLflow LOC	RTLflow CC_{avg}	RTLflow #Tokens	RTLflow T_{trans}
riscv-mini	3306	25224	10640	21.7	66343	< 1s	10935	15.7	171454	< 1s
Spinal	6858	22888	8429	17.7	52646	< 1s	9654	21.7	152459	< 1s
NVDLA	511955	1476991	397536	16.4	3190699	30s	560412	4.8	10424172	33s

Table 1: Statistics of the benchmarks and results of transpiled code for Verilator and RTLflow. The results present lines of code (LOC), average cyclomatic complexity per function (CC_{avg}), total number of tokens (#Tokens), and transpilation time (T_{trans}).

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.

4.1 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.

4.2 Transpilation Results

Table 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.

4.3 Overall Performance Comparison

Table 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 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.

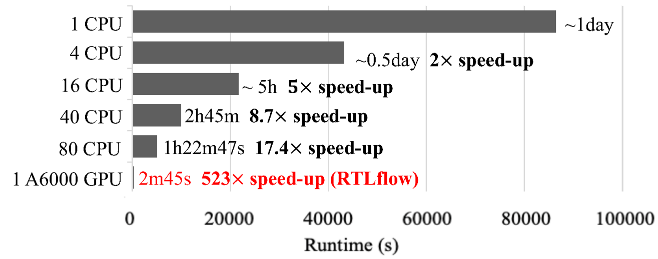


Figure 12: Runtime comparisons across different hardware platforms for NVDLA with 16384 stimulus and 10K cycles.

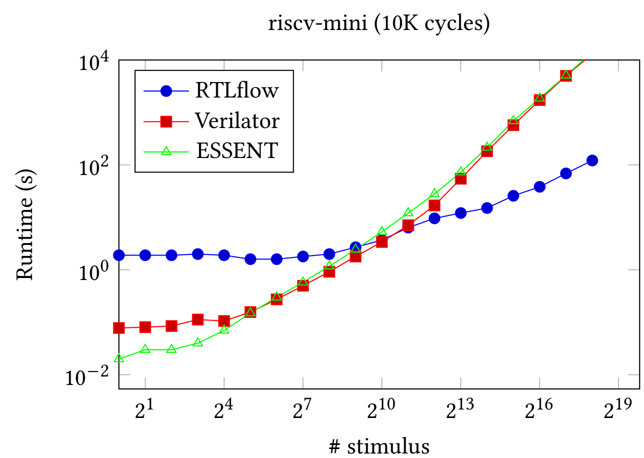


Figure 13: Runtime growth over increasing number of stimulus for Verilator, ESSENT, and RTLflow on riscv-mini.

Figure 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 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 2 and Figure 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

		#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 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.

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 2 for Spinal and NVDLA (256 and 1024 stimulus, respectively). Similar number is also observed in Figure 13 for riscv-mini.

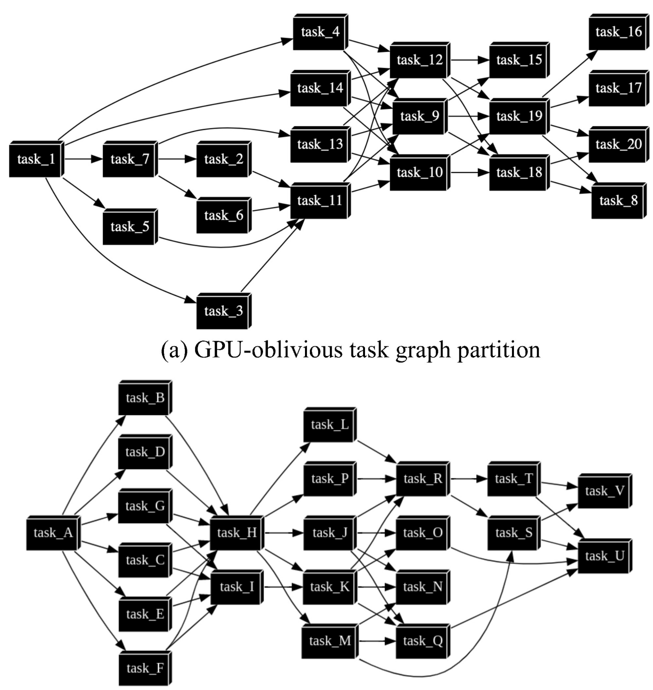
4.4 Performance Result of GPU Task Graph

Table 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 [28]. 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: 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 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).



(a) GPU-oblivious task graph partition

(b) GPU-aware task graph partition

Figure 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.

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 [23, 24] 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 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 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.

4.5 Performance Result of Pipeline Scheduling

	Spinal		NVDLA	
#stimulus	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 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.

In this section, we study the performance benefit of our pipeline scheduling. Table 5 compares the runtime between RTLflow with and without 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.

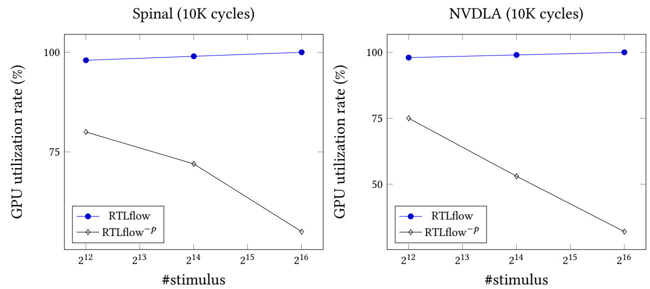


Figure 15: Comparison of GPU utilization between RTLflow with and without pipeline scheduling (RTLflow^{-p}) for simulating Spinal and NVDLA with different numbers of stimulus (under 10K cycles).

Figure 15 plots the average GPU utilization rate profiled by Nvidia System Management Interface [1]. 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 16 plots the utilization timeline of RTLflow^{-p} and RTLflow using the data extracted from Nvidia Nsight Systems [5]. 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.

5 CONCLUSION

In this paper, 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× on the NVDLA of 65536 stimulus. We have made RTLflow open-source to benefit the entire software simulation community [8].

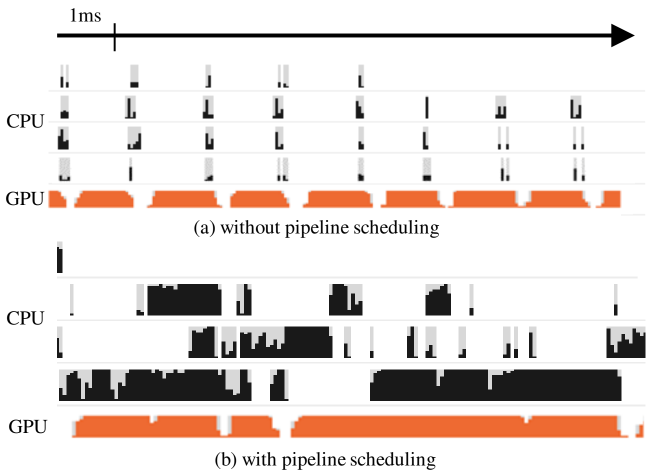


Figure 16: A snapshot of utilization timeline for RTLflow with and without pipeline scheduling, reported by Nvidia Nsight Systems [5].

In the future, we plan to evaluate RTLflow on a much larger range of designs and stimulus, covering a wide range of design sizes and activity factors. Also, we plan to enhance the performance of our pipeline scheduling algorithm using the newest C++20 coroutine. Coroutine will allow us to perform efficient multitasking between CPU and GPU when a stage task at the pipeline submits its CUDA Graph to a GPU. Furthermore, we plan to leverage the proposed algorithms to accelerate other important research problems in circuit designs [14, 15, 17, 20, 21] that share similar performance characteristics with RTL simulation.

ACKNOWLEDGMENTS

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523, and OAC-2209957. We appreciate Nvidia Research's support for running our experiments and all reviewers' comments for improving this paper.

REFERENCES

[1] 2012. Nvidia System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface. (2012).
[2] 2012. Yosys. https://yosyshq.net/yosys/. (2012).
[3] 2016. Spinal. https://github.com/SpinalHDL/VexRiscv. (2016).
[4] 2017. Nvidia Deep Learning Accelerator Design (NVDLA). http://nvdla.org/. (2017).
[5] 2017. Nvidia Nsight Systems. https://developer.nvidia.com/nsight-systems. (2017).
[6] 2018. riscv-mini. https://github.com/ucb-bar/riscv-mini. (2018).
[7] 2019. CUDA Graph. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html. (2019).
[8] 2022. RTLflow. https://github.com/dian-lun-lin/RTLflow. (2022).
[9] Scott Beamer and David Donofrio. 2020. Efficiently exploiting low activity factors to accelerate RTL simulation. In *ACM/IEEE DAC*. 1–6.
[10] Debapriya Chatterjee, Andrew Deorio, and Valeria Bertacco. 2011. Gate-Level Simulation with GPU Computing. *ACM TODAES* 16, 3.
[11] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism Using Control Taskflow Graph. In *ACM HPDC*. 283––284.
[12] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE DAC*.
[13] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. 1995. Markov chain Monte Carlo in practice. CRC press.
[14] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Pash-based Timing Analysis. In *ACM/IEEE DAC*.
[15] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM ICCAD*. 1–8.
[16] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. Oxford University Press.
[17] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* 40, 4 (2021), 776–789.
[18] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*. 974–983.
[19] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems* 33, 6, 1303–1320.
[20] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM ICCAD*. 895–902.
[21] Tsung-Wei Huang, Martin D. F. Wong, Debjit Sinha, Kerim Kalafala, and Natesan Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *ACM/IEEE DAC*. 116:1–116:6.
[22] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE ICPADS*. 64–71.
[23] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *Euro-Par*. 435–450.
[24] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE TPDS* 33, 11 (2022), 3041–3052.
[25] Lingyi Liu and Shobha Vasudevan. 2011. Efficient validation input generation in RTL by hybridized source code analysis. In *2011 Design, Automation Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763253
[26] Hao Qian and Yangdong Deng. 2011. Accelerating RTL simulation with GPUs. In *IEEE/ACM ICCAD*. 687–693.
[27] Vivek Sarkar. 1987. *Partitioning and scheduling parallel programs for execution on multiprocessors*. Ph.D. Dissertation. Stanford University.
[28] Wilson Snyder. 2018. Verilator 4.0: open simulation goes multithreaded. https://veripool.org/papers/Verilator_v4_Multithreaded_OrConf2018.pdf. (2018).
[29] Uri Tal. 2013. RocketSim: A GPU-based Simulation Accelerator for Chip Verification. https://on-demand-gtc.gputechconf.com/gtcnew/speakerName.php?speaker=Uri+Tal. (2013).
[30] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng. 2009. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers Inc.
[31] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2020. Opportunities for RTL and Gate Level Simulation using GPUs. In *IEEE/ACM ICCAD*. 1–5.
[32] Yanqing Zhang, Haoxing Ren, Akshay Sridharan, and Brucek Khailany. 2022. GATSPI: GPU Accelerated Gate-Level Simulation for Power Improvement. In *IEEE/ACM DAC*.
[33] Yuhao Zhu, Bo Wang, and Yangdong Deng. 2011. Massively Parallel Logic Simulation with GPUs. *ACM TODAES* 16, 3.

APPENDIX

We have made RTLflow [8] open-source to beneift the entire community and inspire software simulation research.

A.1 Computational Artifacts: Yes

A.2 Experimental Environment

Verilator and ESSENT have their own optimization flags to enable their unique optimization techniques. We turn on the highest level of optimizations for both baselines on each benchmark to achieve their best performance. For RTLflow, we use Taskflow v3 to implement our pipeline scheduling algorithm.

A.2.1 Hardware.

- Verilator & ESSENT - 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
- RTLflow - 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

A.2.2 Software.

- Verilator & ESSENT - GCC-8 with optimization `-O2` enabled
- RTLflow - nvcc of CUDA 11.6 on a host compiler of GCC-8 with optimization `-O2` enabled

A.3 Software Artifact Availability

To compile RTLflow, simply run,

```
~/RTLflow$ autoconf
~/RTLflow$ ./configure
~/RTLflow$ make -j8
```

We set nvcc flag `-arch=sm_86` to achieve the best performance under our environment.

You can find baselines, RTLflow, and Taskflow here:

- Verilator - https://github.com/verilator/verilator.git
- ESSENT - https://github.com/ucsc-vama/essent
- RTLfflow - https://github.com/dian-lun-lin/RTLflow
- Taskflow - https://github.com/taskflow/taskflow

A.4 Benchmarks

Before simulating a design, you need to build it for generating Verilog code. You can build each design (riscv-mini, Spinal, and NVDLA) by simply typing:

```
~RTLflow_benchmarks$ cd design
~RTLflow_benchmarks/design$ make
```

After make, NVDLA wll ask you to setup your environment (i.e., deisgn configuration, g++, Verilator, and RTLflow paths) to build NVDLA. The configuration of NVDLA in this paper is `hw_small`.

For each design, you can generate multiple sitmulus by using scripts. The scripts generate multiple stimulus by randomly concatenating stimulus offered by each design.

The link of each benchmark is the following:

- NVDLA - https://github.com/nvdla/hw
- Spinal - https://github.com/tomverbeure/cxxrtl_eval
- riscv-mini - https://github.com/ucb-bar/riscv-mini
- RTLflow-benchmarks - https://github.com/dian-lun-lin/RTLflow-benchmarks