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

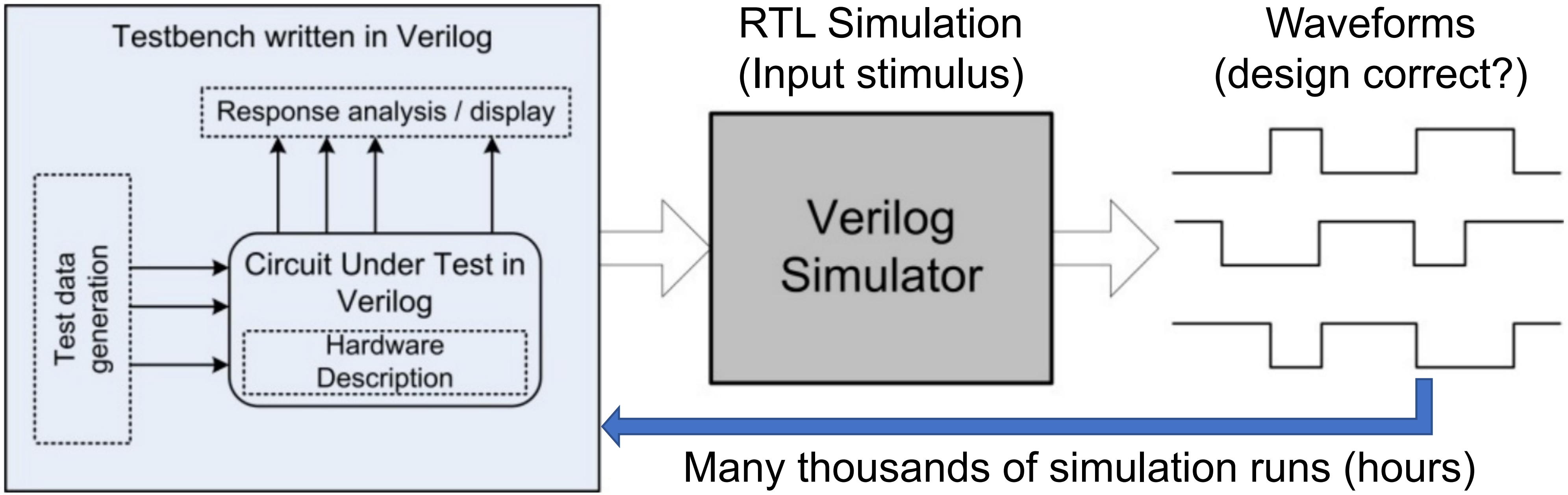
Dr. Tsung-Wei (TW) Huang, Assistant Professor
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Takeaway

- Understand importance of *faster* RTL simulation with GPU
- Discuss limitations of existing RTL simulators
- Identify challenges of GPU-accelerated RTL simulation
- Introduce RTLflow “source-to-source RTL to CUDA transpiler”
- Present experimental results

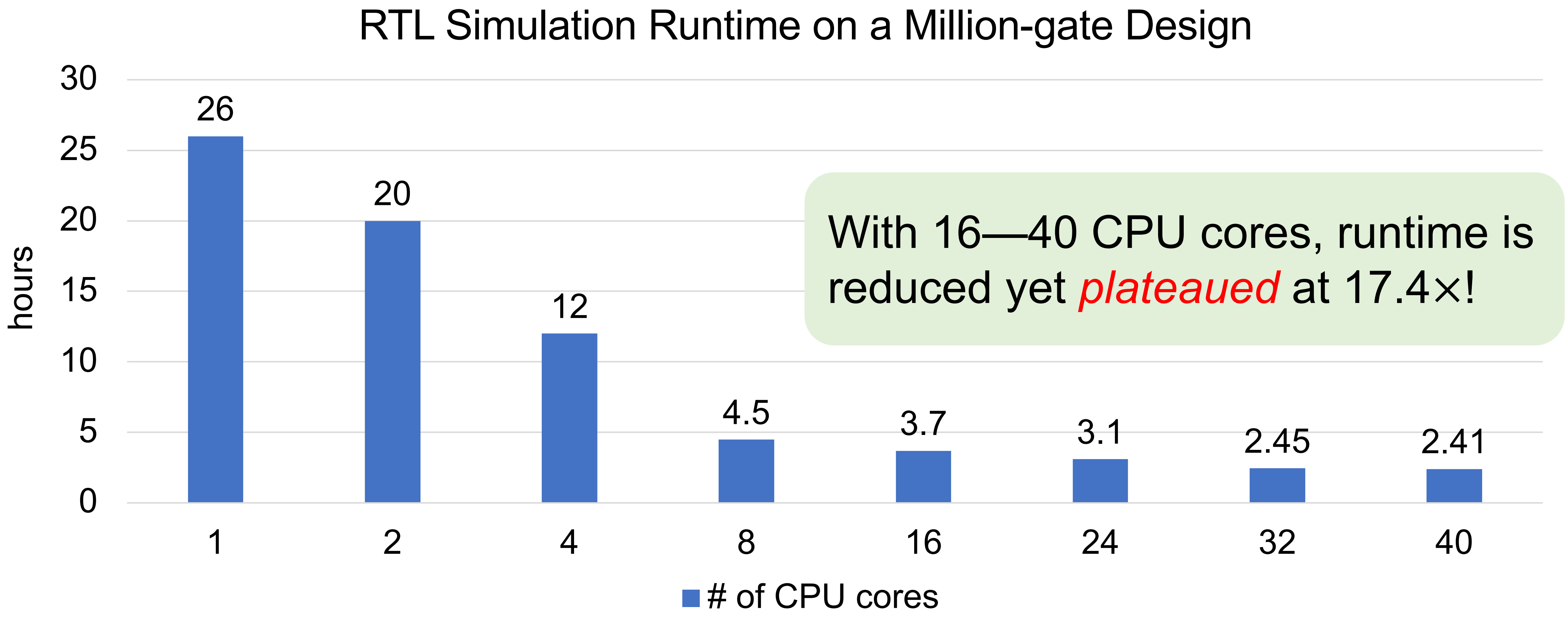
Register-Transfer Level (RTL) Simulation

- **RTL simulation is a critical step in the circuit design flow**
  - Verify functionality of processor and system-on-chips (SoCs) designs
- **However, RTL simulation is a time-consuming process**
  - Run many thousands of nightly tests on a Design-Under-Test (DUT)



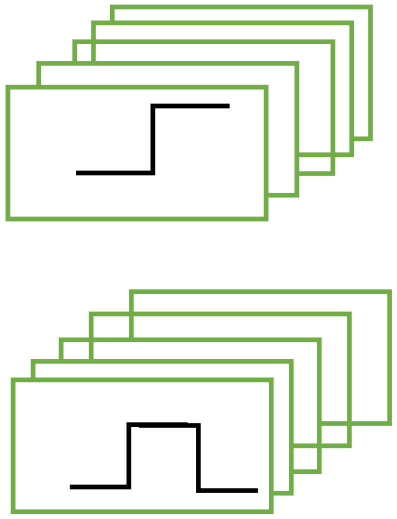
CPU-parallel RTL Simulation

- **Leverage many-core CPU parallelism to reduce the runtime**

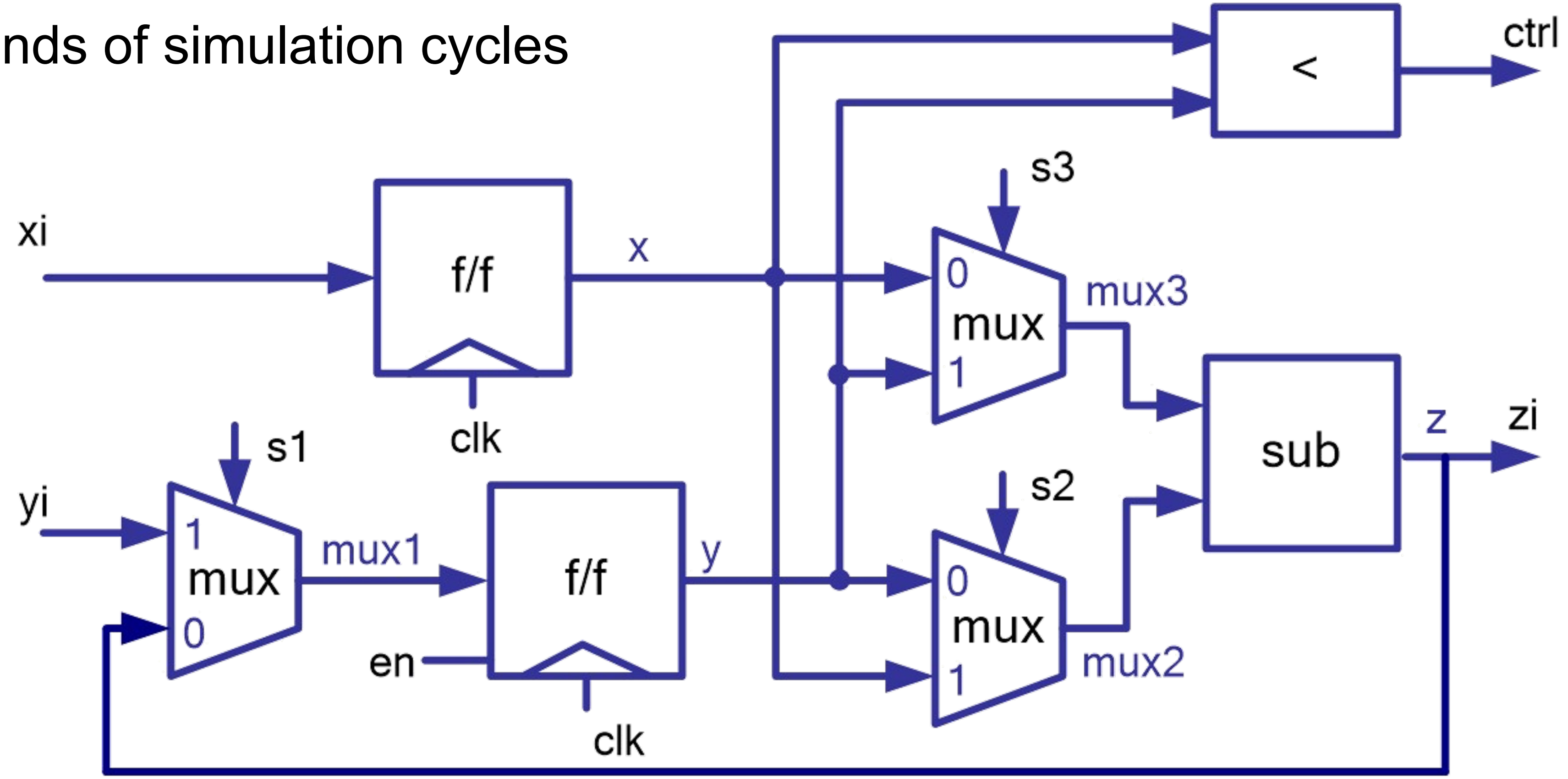


Data-parallelism in RTL Simulation

- **Input many different stimulus batches on the same design**
  - Many thousands of stimulus batches
  - Many thousands of simulation cycles

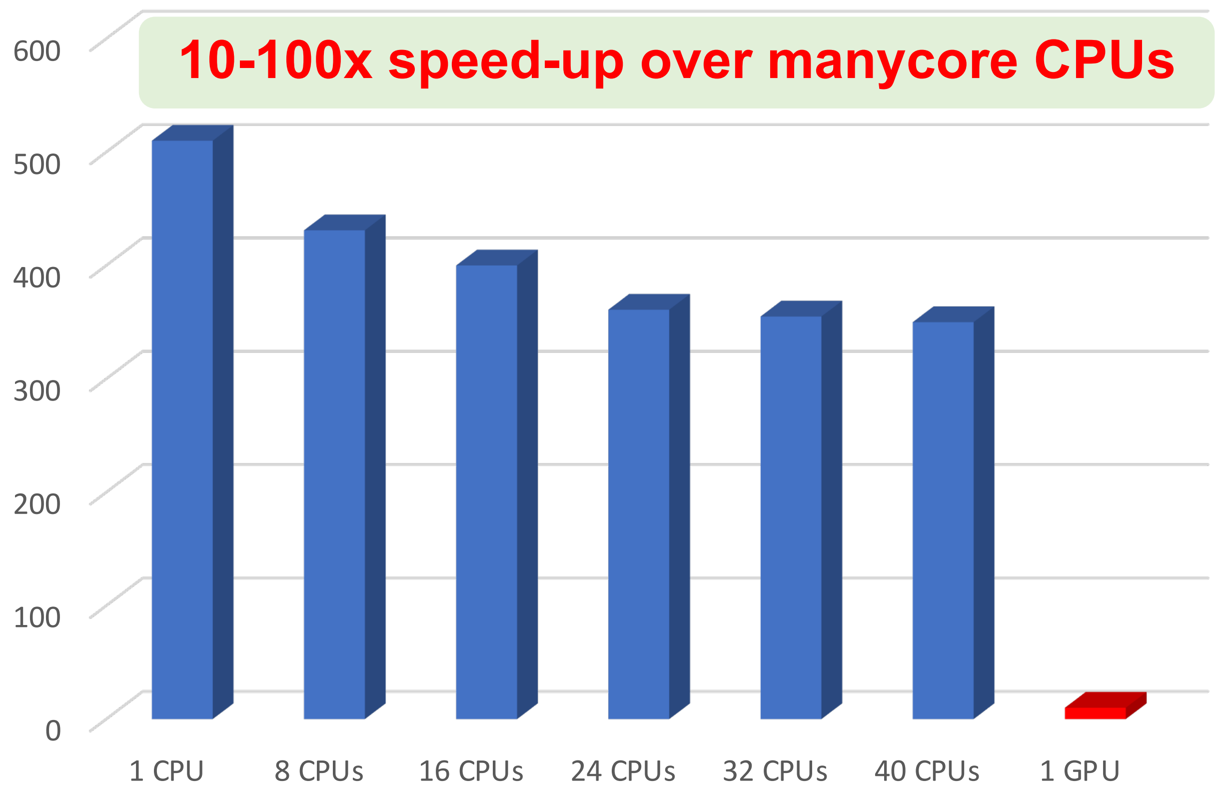


Input stimulus batches

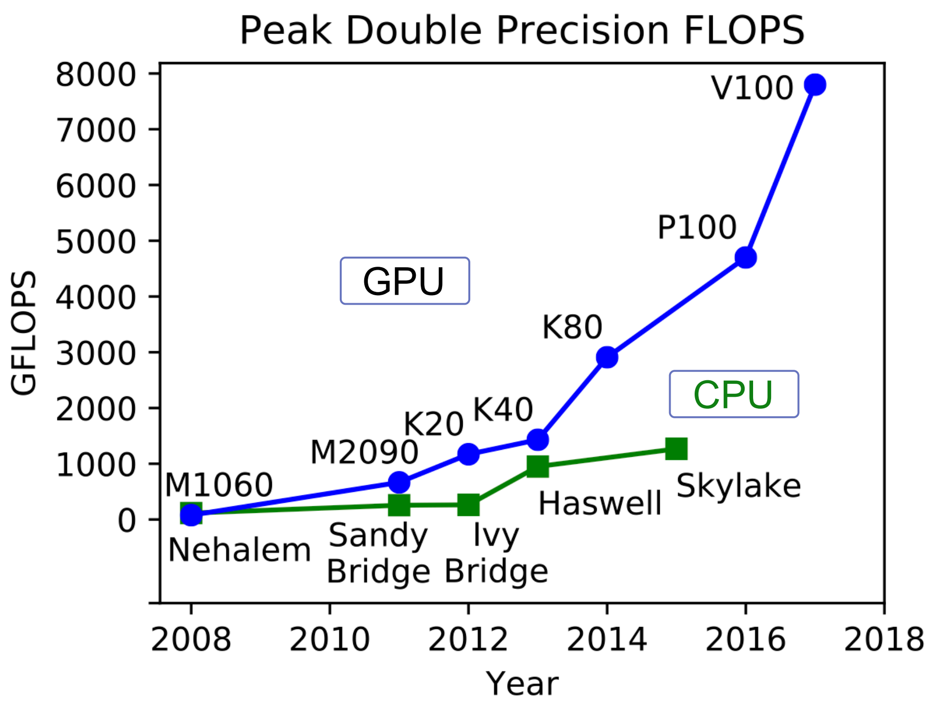


Graphics Processing Unit (GPU) can Help

- **GPU has advanced our computing applications to new levels**



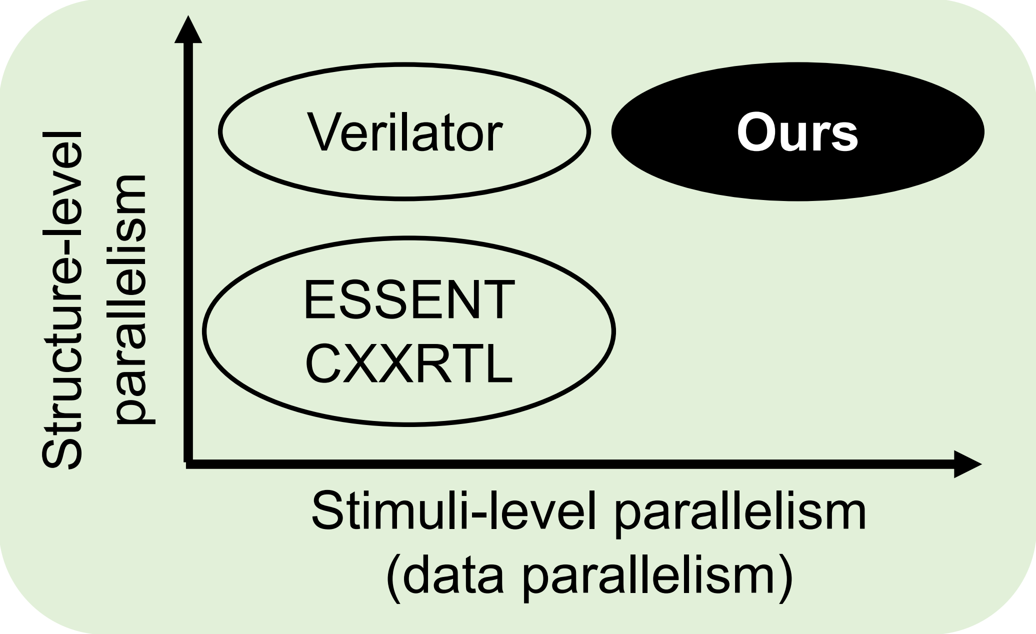
Time (minutes) to solve a machine learning workload



Over **60x** speedup in neural network training since 2013

Limitations of Existing RTL Simulators

- **Existing RTL simulators focus on "structure-level" parallelism**
  - ☺ Partition the design into several RTL processes
  - ☺ Explore parallelism across independent partitions
  - ☺ Counts on compiler to perform data/memory layout optimization
  - ☹ Speed-up is limited by the circuit structure itself
- **Event-driven simulators**
  - ☺ Skip evaluation of zero-activity blocks
  - ☹ Count on sophisticated control flow
  - ☹ Hard to scale to many threads



Verilator: https://www.veripool.org/verilato

CXXRTL: https://github.com/YosysHQ/yosys

ESSENT: https://github.com/ucsc-vama/essent

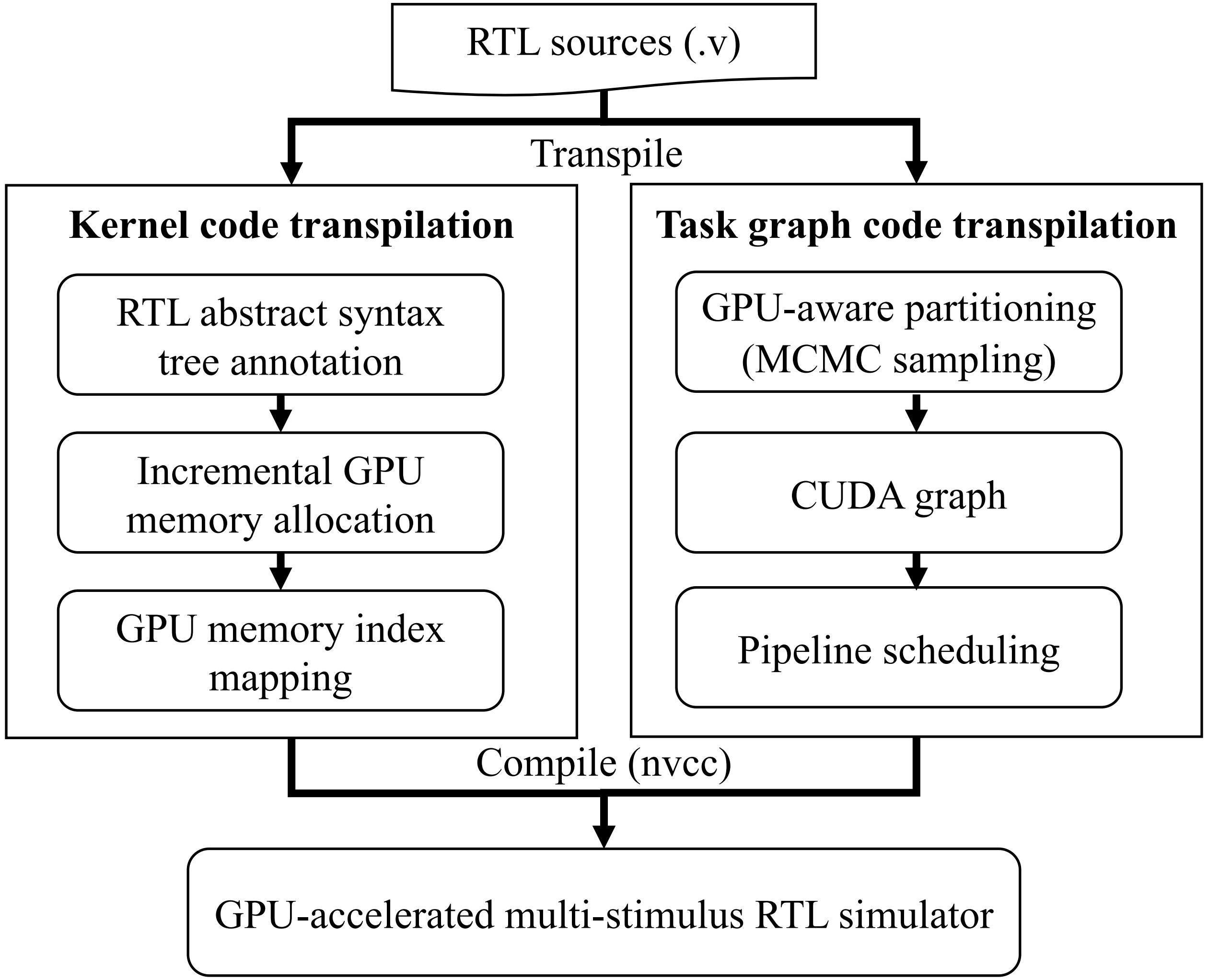
Heterogeneous RTL Simulation Challenges

- **Lack of an open infrastructure to break language barrier**
  - We cannot rewrite RTL simulation code to GPU (e.g., Nvidia CUDA)
  - We need a source-to-source transpiler to automatically go from RTL to CUDA
- **Lack of a GPU-aware partitioning algorithm**
  - We cannot reuse CPU-based partitioner to GPU due to distinct perf models
  - We cannot use static partitioners that count on hard-coded CPU instructions
  - We need a new partitioning algorithm that understands how GPU runs
- **Lack of an efficient CPU-GPU task scheduling algorithm**
  - We cannot stand too much data movement cost between CPU and GPU
  - We need an efficient scheduler to overlap CPU and GPU tasks or, in other words, hide data movement and synchronization overheads

GPU-accelerated RTL Simulator: RTLflow

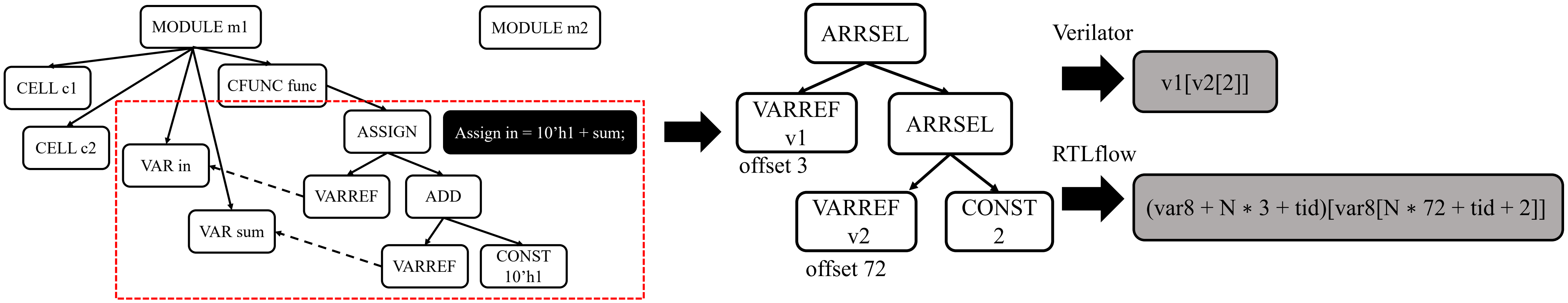
```
Design dut;
Simulator sim(dut);
size_t c = 0;
while(!sim.stop and c <= NUM_CYCLES) {
  dut.set_inputs(c);             CPU-intensive
  dut.set_clock(0);
  sim.evaluate();                GPU-intensive
  dut.set_clock(1);
  sim.evaluate();
  c = c + 1;
}
```

A typical transpiled C++ code example for a targeted RTL simulation workload



Kernel Code Transpilation

1. **Annotate an RTL abstract syntax tree (AST) with textual info**
   - Flatten the hierarchies (i.e., module) to have a single view point of the design
   - Understand the data layout, numbers of variables, simulation instructions
2. **Transpile the annotated RTL AST into C++ and CUDA**
   - Optimize data layout and memory coalescing for efficient GPU computing



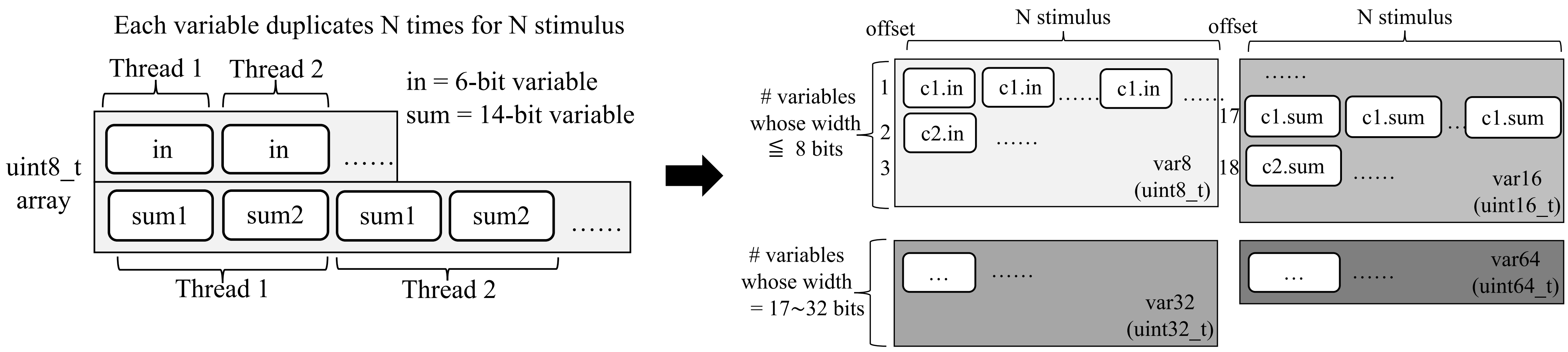
Kernel Code Transpilation (cont'd)

3. Incremental GPU memory allocation

- Separate data types of different widths into different areas
- Allow thread to access data in a coalesced fashion

4. GPU memory index mapping

- Traverse the AST with computed memory offsets to emit efficient kernel code



Kernel Code Transpilation Example

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

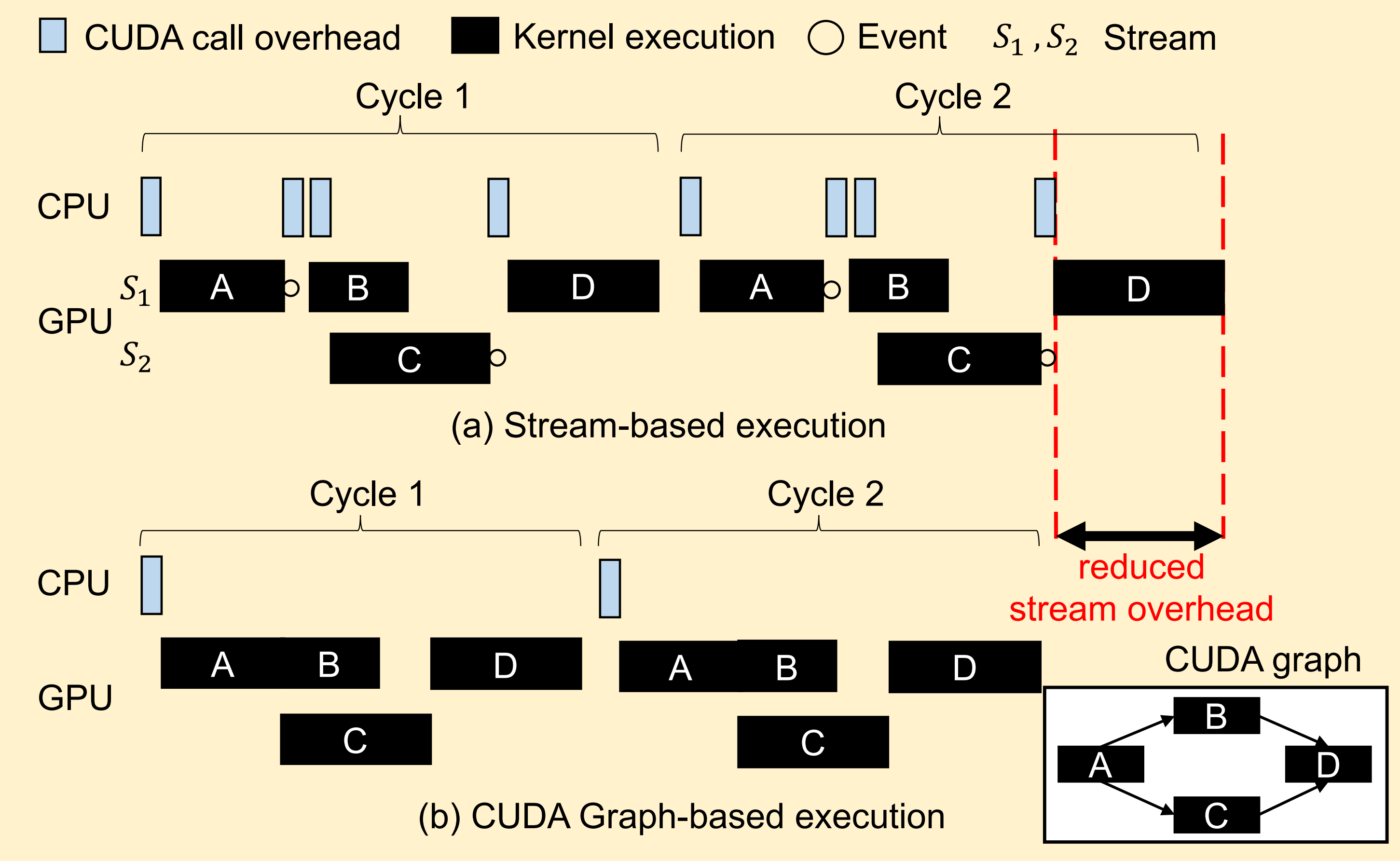
Transpiled CUDA kernel code with optimized data layout for coalesced memory access

```
// 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
}
```

Task Graph Code Transpilation

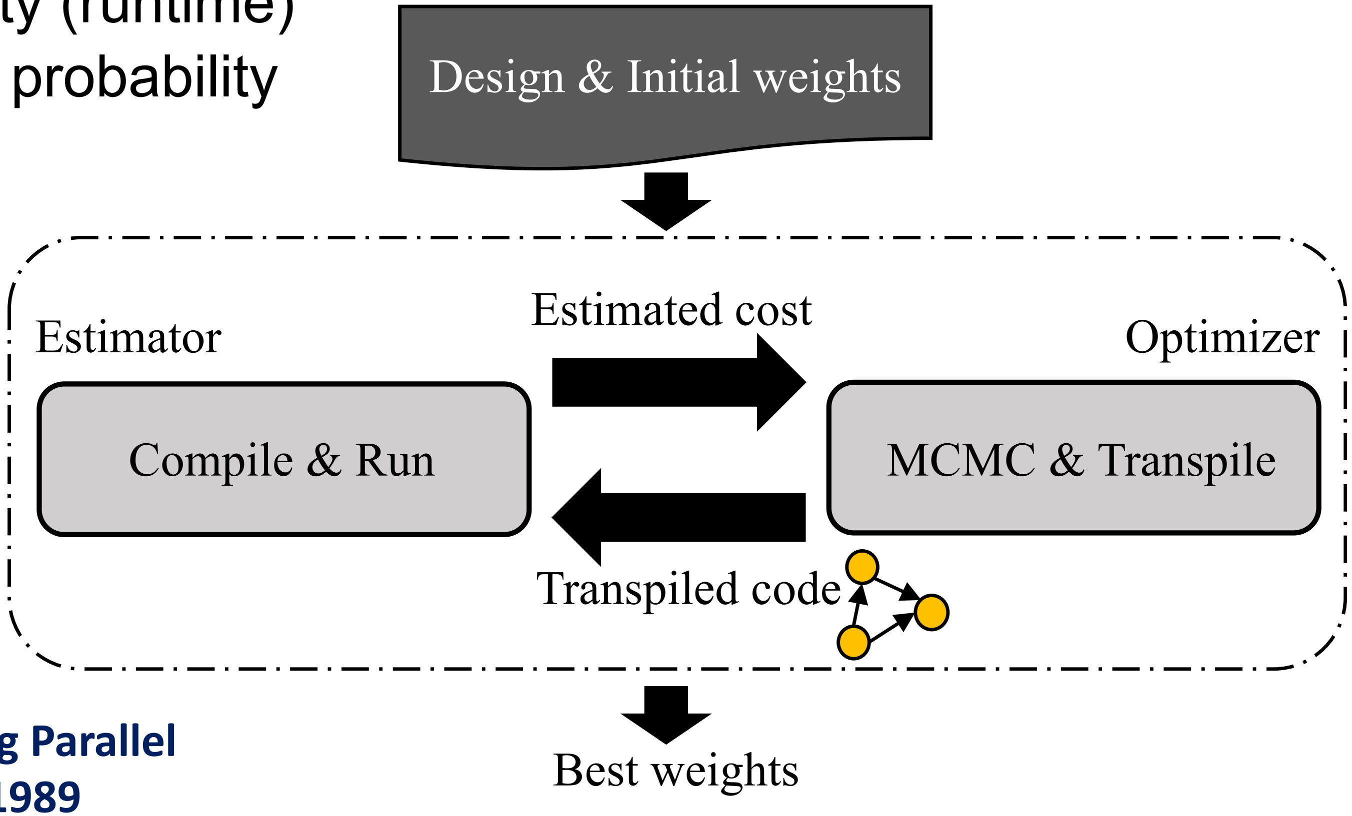
- **Generate fast task-level execution code with three strategies**

1. CUDA Graph execution to reduce kernel call overheads
2. GPU-aware partitioning to find a GPU-efficient task graph
3. Pipeline scheduling to enable efficient CPU-GPU task overlap



Task Graph Optimization

- **Markov Chain Monte Carlo (MCMC)-based graph optimization**
  - Propose a graph partition based on Verilator's partitioning algorithm*
  - Estimate the partition quality (runtime)
  - Accept the proposal with a probability
- **Advantages of MCMC**
  - Run on a *real* condition
  - Learn env parameters
    - CUDA runtime
    - Machine properties
    - Scheduling behaviors
    - …



***Vivek Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessor," *MIT Press*, 1989**

Task Graph Generation (cont'd)

Algorithm 1: GPU-aware partitioning algorithm

```
Input: dut: a design under test
Input: MAX_ITER: maximum #iterations
Input: MAX_UNIMPROVED: maximum #unimproved
       iterations
cur_cost ← ∞
iter, cnt ← 0
Optimizer opt(dut)
Estimator est(dut)
opt.initialize_weights()
while cnt < MAX_UNIMPROVED and iter++ < MAX_ITER
    opt.random_increase()
    graph ← opt.propose()
    cost ← est.estimate_cost(graph)
    if cur_cost > cost then
        opt.update_weights()
        cur_cost ← cost
        cnt ← 0
    end
    else
        rand ← uniform_distribution(0, 1)
        if accept_rate(cost, cur_cost) > rand then
            opt.update_weights()
            cur_cost ← cost
        end
        cnt++
    end
end
```

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

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

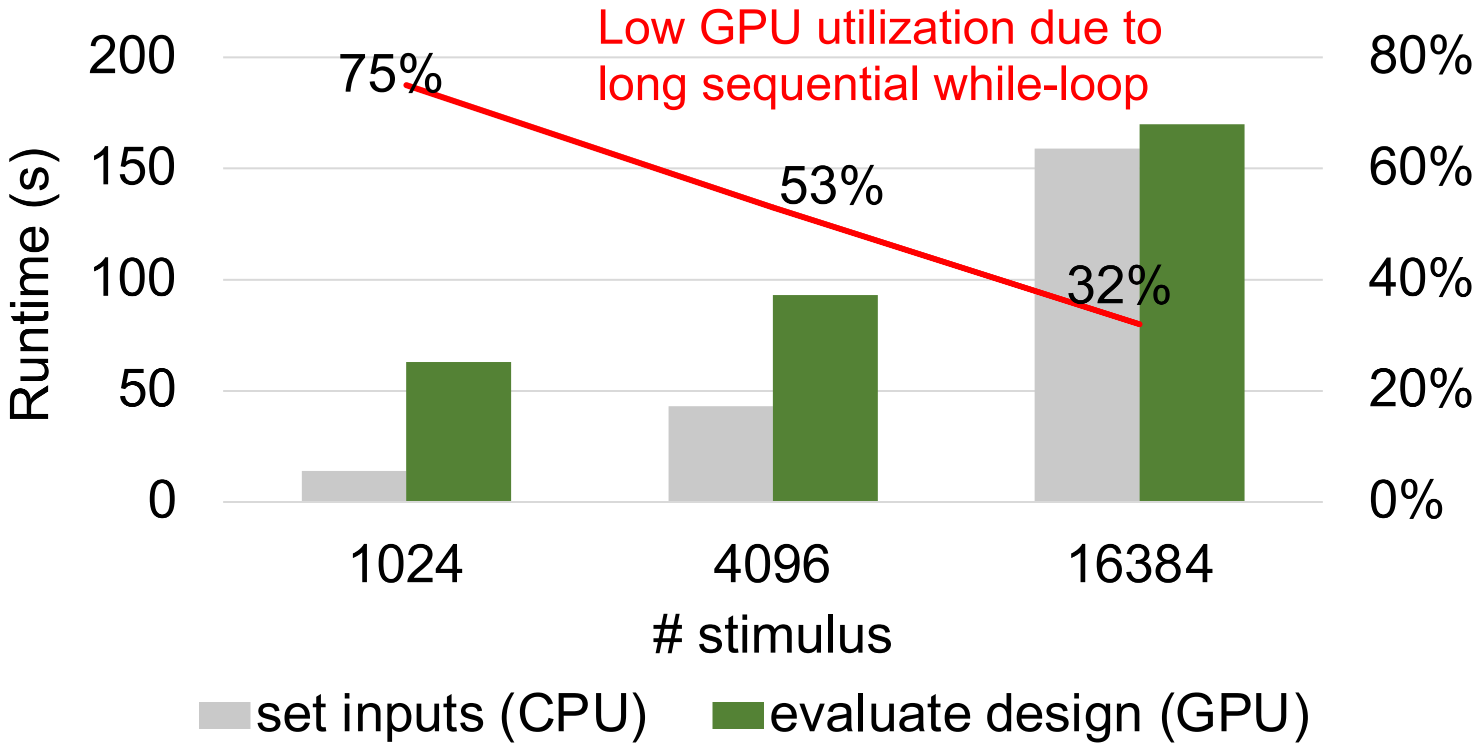
$$\begin{aligned} \alpha(\mathcal{G} \rightarrow \mathcal{G}^*) &= min(1, p(\mathcal{G}^*)/p(\mathcal{G})) \\ &= min(1, exp(\beta * (cost(\mathcal{G}) - cost(\mathcal{G}^*)))) \end{aligned}$$

Pipeline-based Task Scheduling

- **Enable efficient computation overlaps between CPU and GPU**
  - Large simulation workload running in sequential results in long GPU idle time

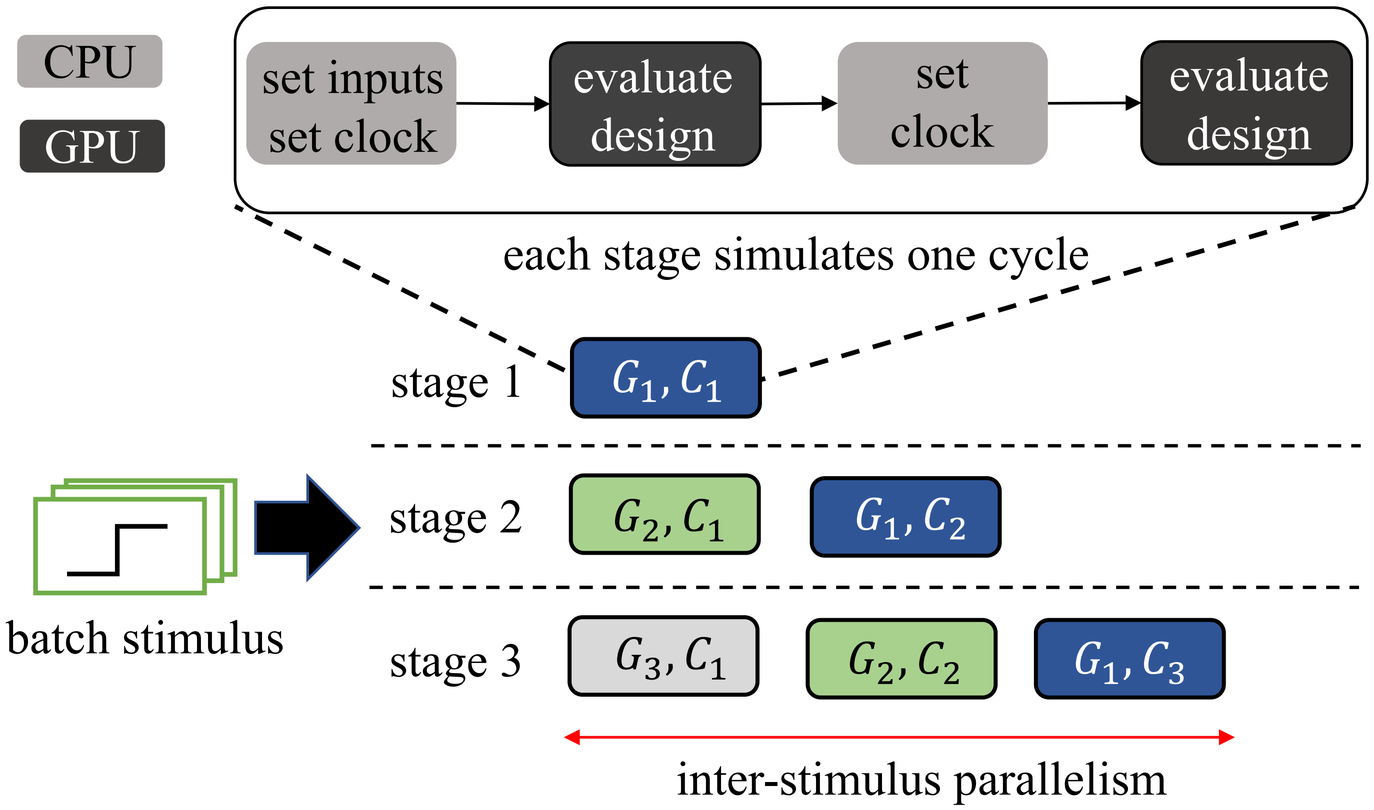
```
Design dut;
Simulator sim(dut);
size_t c = 0;
while(!sim.stop and c <= NUM_CYCLES) {
  dut.set_inputs(c);        CPU-intensive
  dut.set_clock(0);
  sim.evaluate();           GPU-intensive
  dut.set_clock(1);
  sim.evaluate();
  c = c + 1;
}
```

Transpiled C++ code for a targeted RTL simulation workload



Pipeline-based Task Scheduling (cont'd)

- **Partition stimulus batches into groups and pipeline them**



Experimental Results

- **Implemented RTLflow with C++17 and CUDA 11.6**
  - Compiled using GCC-8 with optimization –O2
  - Leveraged Taskflow (https://taskflow.github.io/) for pipeline programming
- **Evaluate RTLflow's performance on three industrial designs**
  - NVDLA (Nvidia's open-source accelerator design: http://nvdla.org/)
  - Spinal (riscv CPU project: https://spinalhdl.github.io/)
  - riscv-mini (riscv CPU project: https://github.com/ucb-bar/riscv-mini)
- **Compared with two baselines, Verilator and ESSENT, on**
  - An Ubuntu server with 40 Intel Xeon Gold 6138 CPU cores
  - A CentOS desktop with 8 Intel i7-11700 CPU cores and an RTX A6000 GPU

Transpilation Results

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}).

			Verilator				RTLflow			
Design	Verilog LOC	#AST nodes	LOC	CC_{avg}	#Tokens	T_{trans}	LOC	CC_{avg}	#Tokens	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

- LOC: lines of transpiled code
- #Tokens: total number of tokens
- T_{tran}: transpilation time
- CC_{avg}: average cyclomatic complexity per function

Significantly improved designers' productivity!

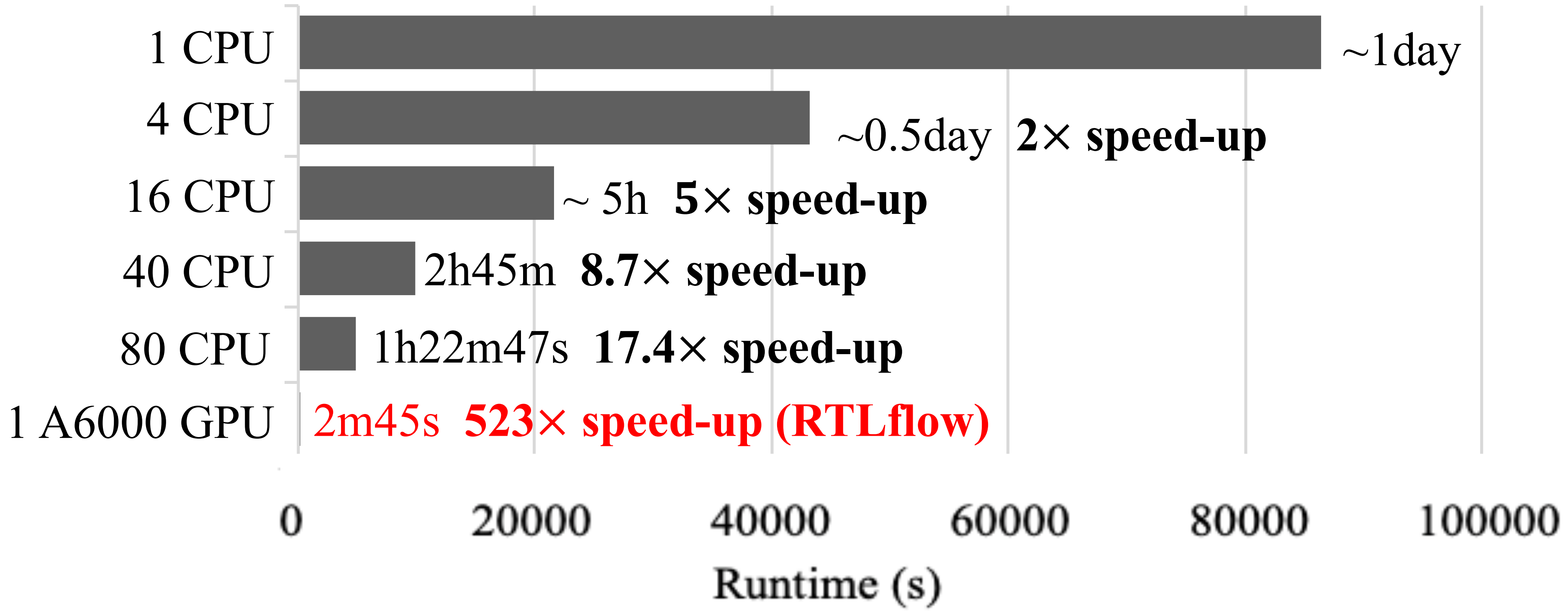
Overall Performance Comparison

		#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.

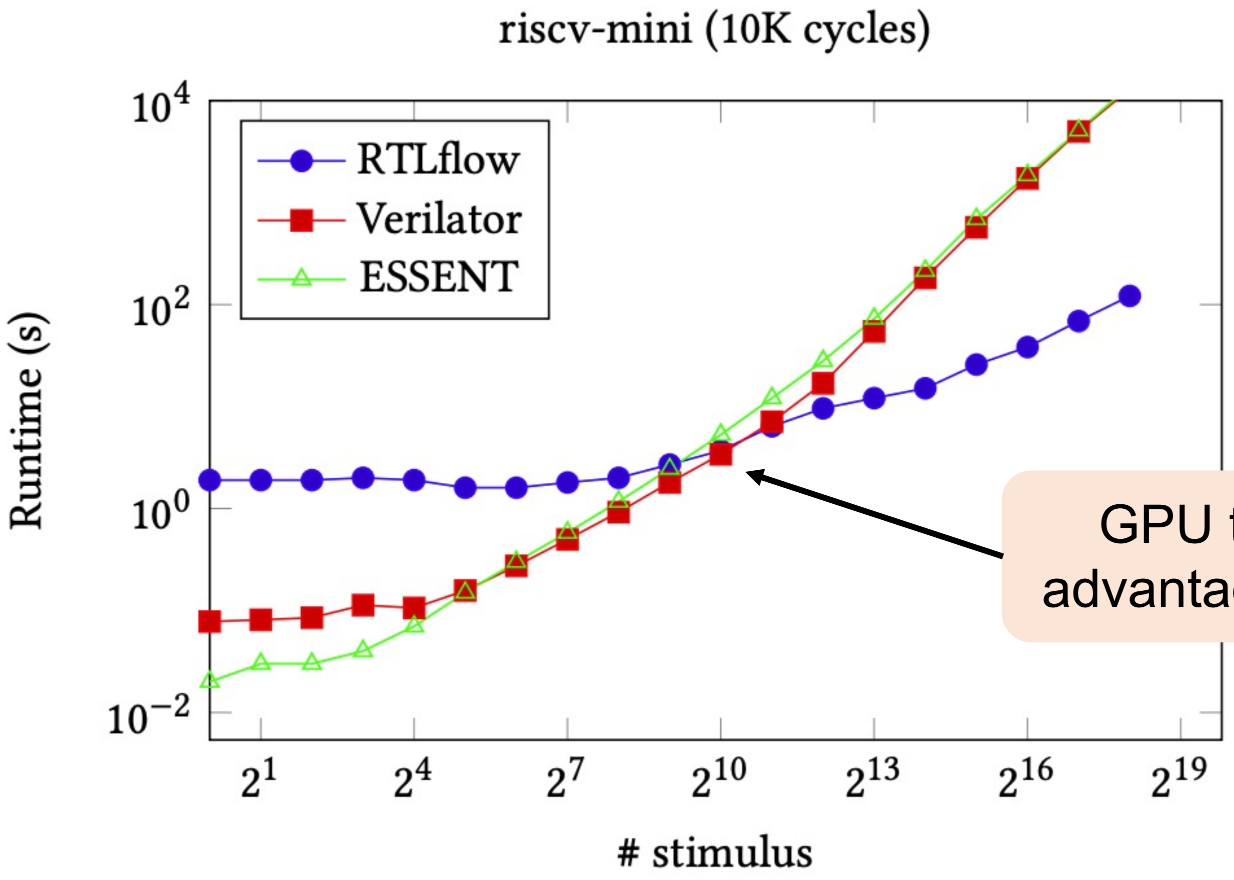
Overall Performance Comparison (cont'd)

- **Simulation time for NVDLA with 16384 batches and 10K cycles**



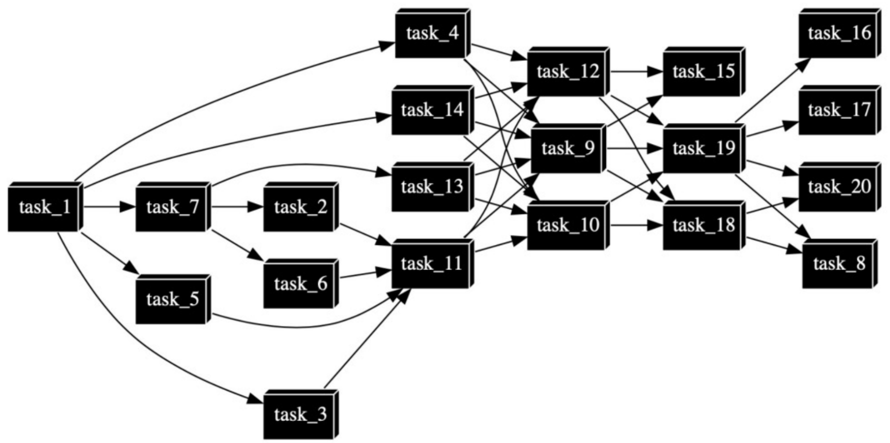
Absolute Efficiency

- **Beyond 1024 stimulus batches RTL is always faster**

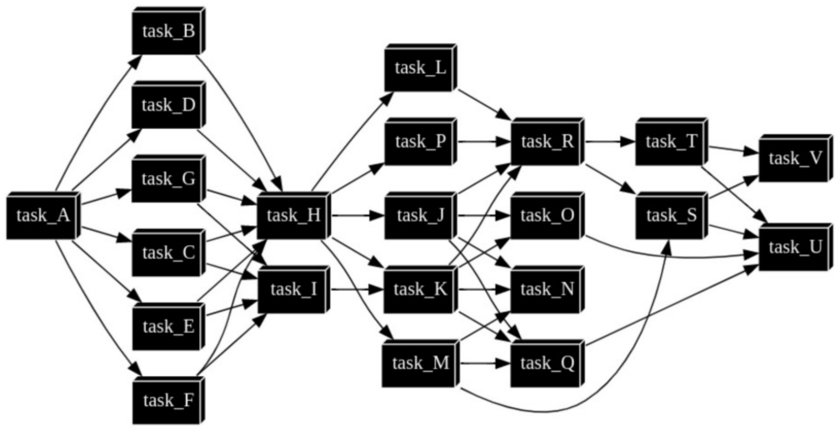


GPU throughput performance advantage begins in 1024 batches

Performance of GPU Task Graphs



(a) GPU-oblivious task graph partition



(b) GPU-aware task graph partition

#cycles	4096 stimulus RTLflow^{-g}	4096 stimulus RTLflow	16384 stimulus RTLflow^{-g}	16384 stimulus 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.

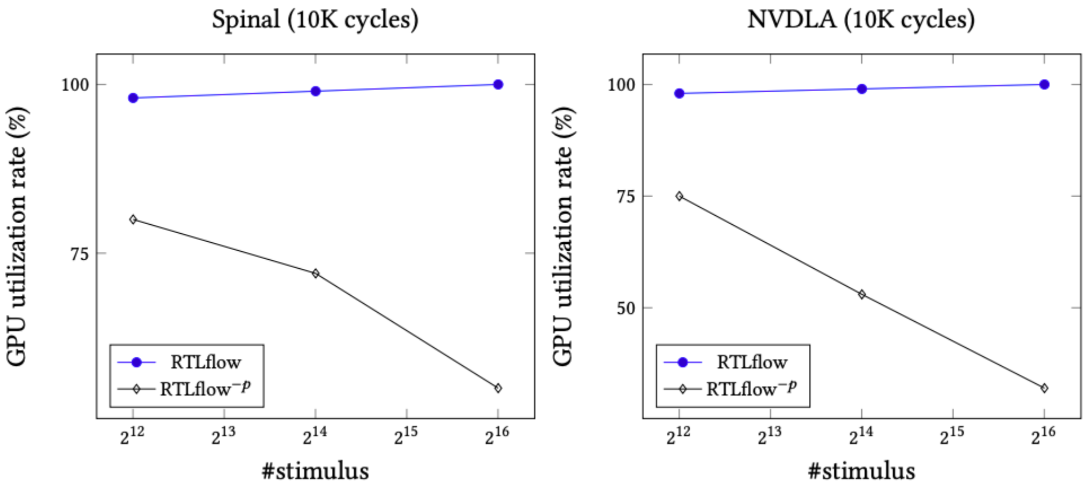
#cycles	Spinal stream	Spinal CUDA Graph	NVDLA stream	NVDLA CUDA Graph
10K	11.5s	2.3s (**5×**)	279.8s	106.5s (**2.6×**)
100K	108.0s	14.2s (**7.6×**)	2046.9s	791.2s (**2.6×**)
500K	532.9s	72.3s (**7.4×**)	9718.0s	3733.0s (**2.6×**)

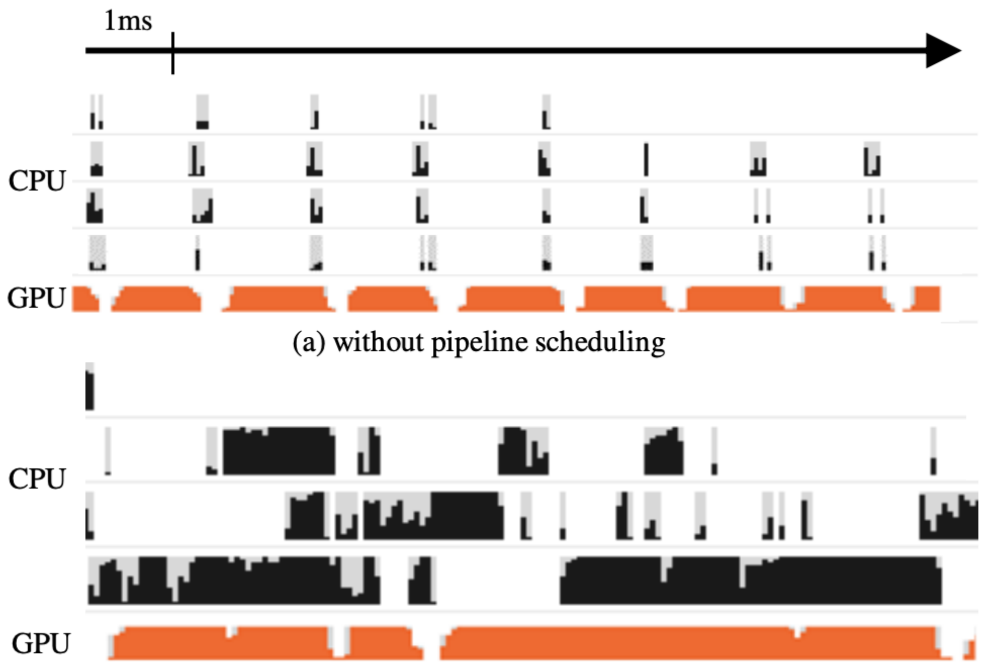
Table 4: Performance advantage of CUDA Graph execution in multi-stimulus simulation workloads, measured on Spinal and NVDLA with 4096 stimulus under different numbers of cycles.

Performance of Pipeline Scheduling

#stimulus	Spinal RTLflow^{-p}	Spinal RTLflow	NVDLA RTLflow^{-p}	NVDLA 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.





(a) without pipeline scheduling

(b) with pipeline scheduling

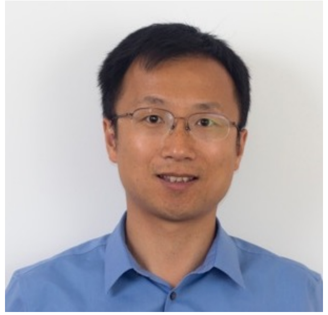
Pipeline enable nearly full GPU utilization all the time

Conclusion

- **Understood importance of *faster* RTL simulation with GPU**
- **Discussed limitations of existing RTL simulators**
- **Identified challenges of GPU-accelerated RTL simulation**
- **Introduced RTLflow "source-to-source RTL to CUDA transpiler"**
  - Transpiled kernel code with optimized memory/data layout on GPU
  - Transpiled task graph code with optimized execution efficiency
- **Presented experimental results**
  - Showed significantly improved programming productivity
  - Showed significantly improved runtime performance via data parallelism
  - Showed the efficiency and effectiveness of the proposed algorithms
- **Future work plans to apply RTLflow to accelerate fuzzing**

Acknowledgement

D-L Lin



Dr. Mark Ren

Dr. B Khailany

Dr. Y Zhang

Use the right tool for the right job

RTLflow: https://github.com/dian-lun-lin/RTLflow

Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, and Tsung-Wei Huang, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," *ACM International Conference on Parallel Processing (ICPP)*, Bordeaux, France, 2022

Thank You