Rapid Synthesis and Simulation of Computational Circuits in an MPPA

David Grant, Graeme Smecher, Guy G. F. Lemieux, Rosemary Francis*

University of British Columbia
2332 Main Mall, Vancouver, BC Canada V6T 1Z4
davidg,gsmecher,lemieux@ece.ubc.ca

* *University of Cambridge*
Rosemary.Francis@cl.cam.ac.uk

***Abstract*—A *computational circuit* is custom-designed hardware which promises to offer maximum speedup of computationally intensive software algorithms. However, the practical needs to manage development cost and many low-level physical design details erodes much of the potential speedup by distracting attention away from high-level architectural design. Instead, designers need an inexpensive, processor-like platform where computational circuits can be rapidly synthesized and simulated. This enables rapid architectural evolution and mitigates the risk of producing custom hardware. In this paper we present a tool flow (RVETool) for compiling computational circuits into a massively parallel processor array (MPPA). We demonstrate the CAD runtime is on average 70x faster than FPGA tools, with a circuit speed 6.4x slower than FPGA devices. Unlike the fixed logic capacity of FPGAs, RVETool can trade area for simulation performance by targeting a wide range of processor cores.**

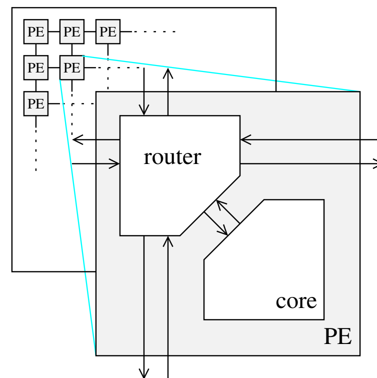

Fig. 1: Architecture overview

I. Introduction

To realize performance gains in many computationally intensive software algorithms, designers are implementing them in hardware as *computational circuits*. The end goal is often to fabricate an ASIC to achieve the highest possible performance. This is being done for a wide range of algorithms, including molecular dynamics, weather simulation, video processing, financial modeling, rendering, and nuclear simulation. Computational circuits are word-oriented and are often very large, requiring millions of gates.

Creating a computational circuit can be challenging and slow. A designer must repeatedly *synthesize* and *simulate* the circuit while debugging, improving, and verifying the design. A correct design is important for avoiding costly ASIC re-spins. Many such circuits are also modelled in C or Matlab to ensure algorithmic correctness before the HDL is even attempted, further increasing design time. As circuit size increases, it takes longer to synthesize and simulate, reducing designer productivity, increasing time-to-market, and worsening the risk of missing a costly bug.

There are currently no solutions on the market which simultaneously address both synthesis speed and simulation speed. Very high synthesis speeds (on the order of seconds) are achieved with compiled-code tools like Synopsis VCS by translating HDL into compiled C. Although these tools offer best-in-class simulation speed, simulating a hardware design on a high-performance processor still yields emulation rates of 1MHz or lower. Parallel simulation may help, but simulation is still dominated by communication costs: one of the highest speedups reported is 13x for 32 processors [1]. Slow simulation speeds are a major obstacle for using these tools to design computational circuits.

In contrast, fast simulation speeds of 100 MHz+ can obtained with FPGA devices. However, the synthesis time to map HDL to an FPGA can take hours or even days. Furthermore, if the HDL does not fit in the target FPGA, designers must resort back to simulation, buy a bigger device (if one exists), or partition the circuit into multiple FPGAs for testing. Slow synthesis and a strict capacity limit are major obstacles for using FPGAs to design computational circuits.

To address both the synthesis and simulation speed problems, we propose to compile Verilog to run on a massively parallel processor array (MPPA). Our tool flow, RVETool (Rapid Verilog Execution Tool), quickly synthesizes word-oriented computational circuits for RVEArch, our MPPA optimized for Verilog execution. RVETool can also target other MPPAs, such as Ambric Am2045 by Nethra Imaging Inc., at the expense of simulation speed.

The RVEArch architecture, first presented in [2], is an array of processors that uses high-speed pipelined interconnect and time-multiplexing to achieve a softer capacity limit where capacity can be traded for performance. The key to accelerating simulation is more than just using additional processors; the low-latency, high-bandwidth NoC in many MPPAs is necessary to overcome the communication bot-

tlenecks in traditional parallel simulators. The pipelined and deterministically scheduled interconnect in RVEArch further enables it to efficiently emulate a circuit.

The objective of the toolflow, which is the topic of this paper, is to map a circuit onto RVEArch quickly and efficiently, resulting in high-speed emulation on the architecture. The tool keeps the circuit at a behavioural level to leverage the coarseness of the underlying architecture and improve synthesis speed. In this paper, we demonstrate that our tools can trade implementation area for speed in a coarse-grained MPPA, a tradeoff first shown for a fine-grained architecture by VEGA [3]. For our benchmarks, we automatically scale circuits from 1 to 256 processors to cover a range of speed versus area solutions that is unmatched by commercial FPGAs. This soft capacity limit is essential for enabling the design of large complex computational circuits. We also show that larger circuits can better utilize more processor cores than small circuits. Hence, we anticipate that much larger circuits can scale to efficiently utilize thousands of processors.

Having a fast Verilog synthesis and simulation flow removes the need for a C or Matlab implementation in the design flow, further saving time. Algorithmic correctness can be demonstrated with behavioural Verilog, laying the groundwork for the final RTL implementation in Verilog.

The overall research goals of this work are to provide a new platform for synthesizing and simulating computational circuits. This platform should improve upon FPGAs by offering 10x faster synthesis, 10x density, and at least $\frac{1}{10}$th of the simulation performance. Although the architecture and tools are still in their infancy, we have already exceeded these goals.

This paper makes the following contributions:

1) It introduces a tool flow to quickly synthesize Verilog for an MPPA architecture.
2) It shows the platform (tools+architecture) can achieve fast synthesis and fast simulation.
3) It demonstrates the tools can automatically scale a circuit to trade area for performance.

Section II presents related work on accelerating circuit simulation. Section III provides a brief overview of our execution model and architecture, and Section IV details the tool flow for the architecture. We present the results of several experiments in Section V, and conclude in Section VI.

II. Related Work

There are several general techniques for accelerating circuit simulation: compiled-code simulators, parallel simulators, and hardware-based accelerators.

Compiled-code simulators translate a circuit into a fixed program for native execution on a fast CPU. Compilation is very fast compared to traditional FPGA CAD flows because no placement or routing is required. Compilation can remain at the behavioural level (rather than gate level) for additional speed, and can also be combined with event-driven simulation [4] to avoid updating parts of a circuit that do not change. However, the resulting program is single-threaded so the final simulation speed is still slow (on the order of 1MHz). Verilator [5], VTOC [6], VBS [7], Symphony's VHDL Simili [8], Synposys VCS, and Cadence NC-Verilog are all examples of compiled-code simulators.

Parallel simulators are concerned with the same fundamental task: simulating a circuit concurrently on as many processors as possible. Parallel gate-level simulation is well researched [9], [10], [11], [12], [13]; speedups are usually less than 10 due to high inter-processor communication costs. PVSim [14] is a combined compiled-code and parallel simulator for up to 8 CPUs implemented with MPI. Compared to FPGAs, parallel simulators are up to six orders of magnitude slower [15], [16], making them prohibitive for verification. For parallel simulation with thousands of processors, a low-latency, high-bandwidth communication network is required.

Hardware based accelerators, like Cadence's Palladium [16] and Mentor Graphics VStation Pro [17], use processors or FPGAs for acceleration. These systems require a complete gate-level synthesis and emulate a design at roughly 2MHz. Although this is slow compared to an FPGA, they can handle very large circuits. These simulators are also large and expensive: ranging in size from a mini-tower computer to a rack, they cost 0.4 to 10 million dollars [18]. Slow synthesis and prohibitive cost are significant barriers to using these devices.

Coarse-grain reconfigurable arrays (CGRAs) and massively parallel processor arrays (MPPAs) are potentially well-suited for speeding up simulation of computational circuits. As a result, our architecture bears great resemblance to these architectures. A wide range of CGRAs exist (e.g. ADRES [19], PipeRench [20], Tartan [21], RaPiD [22], SCORE [23]) and MPPAs (e.g., Ambric [24], Tilera (based on RAW) [25]).

However, in contrast to these existing architectures, we offer several improvements for increased efficiency of simulating computational circuits: no resources are used to implement a C or C-like programming model (e.g., no branch instructions or global memory), efficient implementation of logic gates like multiplexers (in the ALU) and bit-level signals (in a PLA), and concurrent execution of routing and processing tasks.

GPUs are often considered for high-performance computing, but current GPUs are designed for SIMD operations and have a large global memory optimized for coherent (structured) memory accesses. For circuit simulation, each core must execute a different program, so a true MIMD architecture is required. Poor core-to-core communications and high latencies from using unstructured data accesses throttle any potential speedups from a GPU [26].

III. Our Execution Model and Architecture

Computational circuits will be implemented in Verilog, or translated into Verilog from a high-level language like C. Rapid simulation requires a highly parallel, word-oriented platform with very low latency network-on-chip interconnect. The fast interconnect is critical because it must compete with the communication speed of the bare wires in the circuit it is emulating.

Our approach to implementing a computational circuit can be applied to almost any MPPA which supports processor-

to-processor messaging, such as Ambric's. We view each processing element (PE) as containing a router and a core. In RVEArch they are separate hardware entities; in our Ambric implementation a single SRD implements both. Both are time-multiplexed and both follow a pre-programmed static schedule. The router is a 5x5 crossbar with registered outputs. Long-distance (pipelined) communication pathways are created by routing data through several PEs.

Using this time-multiplexed approach, the user clock is different than the system clock. Each PE router and core contain a schedule with exactly n instructions (the schedule length) to be executed in an infinite loop to implement the overall circuit. One pass of this schedule is equivalent to one user clock cycle, so the user clock has a frequency of $\frac{1}{n} \cdot 1$ GHz on RVEArch.

These pre-determined schedules mean the entire architecture is deterministic. This is similar to the GraphStep execution model [27]. It is the responsibility of the tools to schedule the code for each processor and for each router so that data is always in the correct place at the correct time. Non-deterministic delays, such as waiting for input data from an external device, must be handled at the user-circuit level. We are currently assuming the circuit uses a single clock domain. We believe this greatly simplifies the design of computational circuits, which is also the objective of this platform. Multiple clock domains remains an issue for future work.

A. Ambric Implementation

We have implemented this execution model on a slightly modified "idealized" Ambric architecture. We assume a 1 GHz clock instead of the 300 MHz clock used in the Am2045, and assume a single-cycle communication delay between neighbouring Ambric SRD processors. We also assume that every instruction executes in a single cycle, and that sufficient memory exists to hold the entire schedule.

A single SRD implements both the router and core of our execution model. It uses lookup tables to determine which neighbouring SRDs to write to, then read from, then which core instruction to execute to complete a system clock cycle. Ambric uses blocking registers, requiring that all reads and writes are matched; our deterministic schedule gives us exactly this, so there is no risk of deadlocking the architecture.

In Section V we have used the output of RVETool to estimate the native-compiled code schedule length of each test circuit on this idealized architecture. The schedule length is calculated by summing the maximum number of reads and writes required across all PEs in each timeslot, and adding one to each total for the actual instruction execution.

B. RVEArch

RVEArch, shown in Fig. 1, distinguishes itself from other MPPAs in several ways:

- It uses very low-latency, very high-bandwidth interconnect to connect only neighbouring PEs.
- It is intended to scale up to 100×100 PEs on a single chip by using a skew-tolerant clock distribution network.

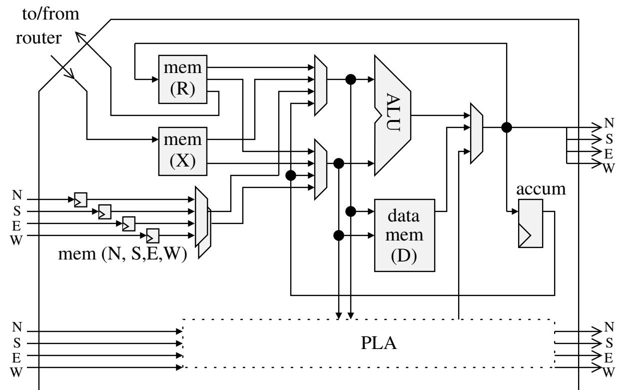

Fig. 2: RVEArch PE core

- It contains a dedicated router.
- It contains a PLA to implement bit-level operations.

RVEArch uses a global clock, but PEs communicate only over short distances. Thus, while local skew between all neighbouring PEs needs to be small and bounded, global skew requirements are relaxed and permit the use of a low-energy clock distribution network. Since all PEs use the same clock source, the neighbours will still operate in synchrony. With this design, the architecture is readily scalable to high clock frequencies. Results presented in this paper assume a 1 GHz clock is realizable in 65nm.

The RVEArch processor core is shown in Fig. 2. It is a simple processor with time-multiplexed ALU and data memory D to implement user-level circuit behaviour. There are no branch instructions, but conditional moves and multiplex (select) functions are supported. Node memory R is used to temporarily store ALU results—emulating a wire for data used in the same user clock cycle and a flip-flop for data needed in the next user cycle. Likewise, node memory X is used by the router to store data destined for this core. There are also four buffered, direct links to adjacent PE cores (node memory N, S, E, and W) which are used as a lower-latency alternative to the router for neighbour-only connections. These links are enabled in RVEArch for all results presented in this paper.

The dedicated router in each PE allows all five router outputs to be assigned in a single cycle. This is a significant advantage over the Ambric implemetnation where communication is done serially and inline with the PE core computation. It is a key feature to keep communication costs low.

All buses are 32 bits wide, the node memory X and R are each 16x32-bit, the data memory is 8 kB (2048x32-bit), the remaining node memories are 4x32-bit. We assume all ALU operations including multiply are single-cycle.

Computational circuits may also contain single-bit (e.g. control) signals that map poorly to word-oriented processors. The PE core in Fig. 2 shows a PLA that is not time-multiplexed, which generates these signals. We are investigating the implementation of the PLA, so it has been omitted from all results in this paper. Instead, the tools currently implement single-bit logic using ALU instructions. (Few of our benchmark circuits use bit-wide signals.)

IV. Our Toolflow - RVETool

This section introduces RVETool, a tool flow which maps circuits onto RVEArch and other MPPA architectures. The input is Verilog circuit, and the output is a bitstream for the architecture or a simulator. The input circuit is also allowed be many disjoint circuits, provided a common clock is used.

The tools use a graph representation of the circuit where graph nodes are circuit operations and graph edges are communication. The tool flow objective is to partition the circuit into clusters of executable code, one for each PE, and then schedule data movement and order code execution to reproduce the behaviour of the original circuit. At all steps, decisions favour minimizing the length of the overall schedule for the fastest possible simulation.

To test different parameters, an architecture file is used to specify the number of PEs, the width of buses, the size of each memory, and the resources in each PE (e.g., if the PE can perform I/O). These parameters act as constraints in the tool flow. In this paper we only vary the number of PEs.

RVETool is separated into five sub-tools: **Parallelize**, **Combine**, **Place**, **Route**, and **Schedule**. Additionally, a **Simulate** step is used to mimic our target architecture. Each step is presented in the following sections.

A. Parallelize

The **Parallelize** tool parses the Verilog source and constructs a graph representation of the circuit (including the control and data flow) at a behavioural level. It then performs graph legalization for execution on the target architecture.

The tool uses a modified version of Verilator [5] for parsing and graph construction. We allow Verilator to perform several processing steps and simple optimizations like module elaboration, dead code elimination, and constant folding. It also converts "free" hardware operations like bit-shifts or word-length truncations into shift and mask instructions. Verilator would normally generate a serialized C++ program for compilation and execution on the host processor, but we terminate it before it begins to serialize the graph.

The graph legalization is similar to technology mapping. Many operations in the Verilator output (e.g., the arithmetic and logic operations) map directly into PE instructions and are trivially converted. There are, however, other required graph transformations:

- Multiple writes to a single variable (a register, wire, or variable in the source Verilog) are mapped to a chain of multiplexers that feed a single write operation. This allows the computation of the written value to be (potentially) distributed among processors, while ensuring only one instance of the final value exists.
- Circuit inputs and outputs are mapped into I/O load and store operations. These operations are later placed on PEs with the I/O resource by the **Place** tool.
- User-instantiated memories, represented as array operations in the behavioural graph, are mapped to PE memory operations (load and store). The **Combine** tool ensures all operations on a user memory reside in the same PE.
- Any node fanning out to a register is flagged as "end of cycle". If the node also fans out to a non-register it is duplicated first. All registers are then replaced with wires. This flag causes special treatment in the scheduler to recreate the expected clock-edge register behaviour.

B. Combine

The **Combine** tool groups operations in the graph into code clusters for each PE. **Combine** is similar to clustering in CAD tools: wide nets and heavy communication nets are absorbed into a single cluster and kept off the communication network. It is this ability to combine code to varying degrees that trades off area (PEs) for performance and achieves a soft capacity limit. This is demonstrated in Section V-B.

The tool uses hMETIS [28] to partition the graph using recursive bisection. To guide hMETIS, all nodes are assigned a weight of 1, except constant inputs which are assigned a weight of 0 so they can be placed in any cluster for free. All edges are assigned a weight equal to the bit-width of the variable on that edge. Load and store operations from data memory are artificially connected with high edge weights to ensure they will not be separated.

C. Placement

The **Place** tool assigns code clusters to physical PEs while trying to keep the critical path as small as possible.

The problem is similar to the FPGA CAD placement problem, so we use VPR's [29] timing-driven annealing algorithm with a different cost function. The pipelined routing network in RVEArch means the delay between two nodes is equal to the Manhattan distance, not a propagation delay along a wire as in conventional CAD. The time-multiplexed PE cores introduce an additional level of complexity not found in FPGAs: two nodes within the same PE may be scheduled in timeslots far apart, causing additional critical-path delay. Unfortunately this delay is not known until scheduling is complete, so at this stage we assume it is zero.

The delay in clock cycles between two nodes i and j is:

$$delay(i,j) = \begin{cases} 1 & i,j \text{ placed in same PE} \\ 2 & i,j \text{ placed in adjacent PEs} \\ 3 + mh(i,j) & \text{otherwise} \end{cases}$$

Where $mh(i,j)$ is the Manhattan distance between the PEs of i and j. For nodes in the same PE, the PE core must execute an instruction to produce the next value. For adjacent PEs, the value must be produced and communicated over a neighbour link. For distant PEs, there is a two cycle penalty to access the pipelined routing network, plus a number of cycles equal to the Manhattan distance to traverse it. The $delay(i,j)$ is used with a slack and criticality computation to calculate the $timing_cost$ of the circuit, which is part of the placement cost function. The slack, criticality, and $timing_cost$ computations are the same as in VPR [29].

In addition to timing cost, the VPR placement cost function uses a wiring cost, which we compute differently than VPR:

$$wiring_cost = \sum_{\forall i,j \in circuit} mh(i,j)$$

i.e., the Manhattan distance from every source to every sink. Since all communication links are time-multiplexed, the Manhattan distance prioritizes latency (performance) over interconnect utilization. This is different from traditional bounding-box minimization [30], where limited physical wiring forces wirelength to be more important than delay.

The final placement cost function is similar to VPR's:

$$\Delta C = \lambda \cdot \frac{\Delta timing_cost}{previous_timing_cost} + (1-\lambda) \cdot \frac{\Delta wiring_cost}{previous_wiring_cost} + penalty$$

The variable $penalty$ is used to discourage illegal placements by adding 1000 to the cost each time memory size is exceeded, unavailable PE resources are used, or too few/many PEs are used. λ is used to place more (or less) emphasis on the timing-driven aspects of placement, and for this work $\lambda = 0.9$.

At the end of placement, the **Place** tool packs any user-instantiated memories into single PEs, ensuring they do not overlap. The problem of splitting a large user memory across multiple PEs is left for future work.

D. Route

After placement, the **Route** tool routes data between all communicating PEs (ignoring internal PE communications). The routing problem is different from conventional CAD flows because the routing network is time-multiplexed, so the router must make temporal as well as the conventional spatial routing decisions. RVEArch also contains two routing resources: PE routers for routing data over any distance (requiring $2 + h$ cycles, where h is the number of hops), and single-cycle lower-latency links for communicating between adjacent PEs.

For long-distance routes, the tool uses a simple horizontal-then-vertical routing strategy, routing each path without considering congestion or collisions. In practice, we find there is little routing contention over space and time, and that the router can operate adequately without considering congestion at all. The **Schedule** tool resolves any routing conflicts by introducing a "hold slot" and pushing back the conflicting route to a later timeslot.

For neighbour communication, the router marks the required neighbour-neighbour link, again ignoring congestion and conflicts. The **Schedule** tool assigns the link usage to a timeslot and directs the value to a specific offset of the N, S, E, or W node memory, or may delay the producing node if resources are unavailable.

E. Schedule

The **Schedule** tool orchestrates the overall execution of code and movement of data to reproduce the behaviour of the original circuit. It assigns each instruction to a timeslot in a PE core, and assigns each route-hop to a timeslot in the PE routers, resolving all routing collisions along the way.

The main loop of the **Schedule** tool is shown in Algorithm 1. The scheduler begins at $timeslot = 1$ and assigns as many nodes as it can across all PEs in that timeslot. It then moves on to the second timeslot, and so on. This timeslot-oriented approach ensures the scheduler is fast and is always making forward progress. NOP instructions are inserted in all timeslots that do not contain a circuit node after scheduling.

Algorithm 1 Schedule Tool Main Loop

```
1:  ready_queue ← all nodes flagged "end of cycle" or nodes
    with no parent
2:  timeslot ← 1
3:  loop
4:    if is_empty(ready_queue) then
5:      if is_empty(next_queue) then
6:        return /* Scheduling complete */
7:      end if
8:      ready_queue ← next_queue
9:      timeslot ← timeslot + 1
10:   end if
11:   node ← dequeue(ready_queue)
12:   if not is_schedulable(node) then
13:     enqueue(next_queue, node)
14:     continue /* Restart loop */
15:   end if
16:   schedule_routes(node)
17:   node.timeslot ← timeslot
18:   for all node.children do
19:     child.sched_parent ← child.sched_parent + 1
20:     if child.sched_parent = |child.parents| then
21:       enqueue(next_queue, child)
22:     end if
23:   end for
24: end loop
```

The is_schedulable($node$) function checks whether $node$ is schedulable in the current $timeslot$. It verifies that all of the following are true:

- The code position at $timeslot$ is empty in the PE
- $node$ may be scheduled as early as $timeslot$
- All PE core resources required by $node$ are available
- All PE router resources required by the output of $node$ are available for the first-hop of the route

The schedule_routes($node$) function assigns route-hops to timeslots in the PE routers. When a routing conflict arises, a router holds the value for as many timeslots as necessary for a free timeslot to be found in the next hop. When each route arrives at the destination PE, the destination node(s) are marked with the arrival timeslot; those nodes cannot be scheduled before the arrival timeslot.

At the end of scheduling, memory accesses (to node memory and data memory) are assigned specific offsets using a greedy approach. The code for each router and core is finally packed into a single output bitstream.

F. Simulation

The bitstream is given to a compiled-code architectural simulator to ensure functional correctness, or given to a

statistics tool to extract useful performance data. All the results for our architecture presented in this paper came from the statistics tool after a complete pass through the toolflow. The simulation tool runs sequentially on a PC, but uses its own scheduler to emulate parallel execution of the bitstream. The simulation accepts input stimulus from vector waveform files and writes output waveform files. We verified the correctness of our entire toolflow by simulating the same Verilog circuit in Quartus II and comparing output waveforms.

V. Experimental Results

In the following sections, we evaluate RVETool and RVEArch. We begin by examining their overall ability to quickly synthesize and simulate computational circuits compared to traditional FPGA tools and software simulators. Next, we evaluate their ability to scale a circuit across a number of PEs, to better exploit parallelism than software simulators. Finally, we compare computer-generated bitstreams with hand-tuned ones, to explore the performance achievable by further refining our CAD tools.

To evaluate each of these properties, we introduce several benchmarks written in Verilog:

- The **chem**, **dir**, **honda**, **mcm**, **pr**, and **wang** benchmarks [31] are dataflow– and DSP-style non-pipelined computational circuits described in behavioural Verilog.
- **AES** is a byte-oriented implementation taken from Altera's QUIP suite [32]. It was modified in several places to use lookup tables instead of bit-level logic operations.
- **Motion Estimation** (ME) is an image-processing algorithm described in [2]; a block R of 16x16 pixels is swept against a 32x32-pixel reference image M, searching for the displacement which produces the lowest sum of absolute differences (SAD).
- **FFT8** and **FFT16** are 8- and 16-point complex FFTs respectively, implemented using a radix-2, decimation-in-time decomposition.

Of these benchmarks, ME and FFT16 are the largest, involving 3609 and 1192 circuit nodes, respectively. The other benchmarks' complexity varies from 90 nodes (wang) to 553 nodes (chem). A node is roughly equivalent to one arithmetic, logical, or routing operation up to 32 bits in size.

A. Synthesis and Simulation Speed

We first consider the performance of the toolflow and architecture. Table I shows both the time required to synthesize Verilog into a format suitable for simulation, as well as the corresponding simulation speeds for several platforms:

- Verilator (abbreviated Vltr), translates Verilog into a C++ program run on a 2.8 GHz Intel P4 PC,
- ModelSim 6.3d (abbreviated MS) run on the same PC,
- Idealized Ambric (abbreviated Amb, see Section III-A) at 1 GHz, code estimated from output of RVETool,
- Altera Stratix-III FPGA (EP3SE80F, speed -2; abbreviated Stx-III), using Quartus II 9.0, and
- RVEArch (abbreviated RVE) at 1 GHz, code generated by RVETool.

Fig. 3 graphs the simulation speed results. In all cases, the FPGA implementation runs faster than the alternatives, but takes a much longer time to synthesize. The RVEArch provides the fastest simulation results of the software-based alternatives, with a synthesis time comparable to Verilator. The Idealized Ambric results show that RVETool can generate fast, scalable code for other MPPA architectures as well. Not only do the software platforms exhibit the poorest performance, they demonstrate poor scalability as the larger circuits tend to run more slowly. In comparison, RVEArch and Ambric maintain a relatively constant simulation performance as the circuit size increases.

The fast synthesis speed of RVETool is a combination of: targeting a coarse-grained MPPA (less work for the tools), not synthesizing to gates (less work for the tools), and using algorithms that allow quality of results to be balanced against runtime.

Our tools presently exhibit two anomalous results. First, the AES circuit optimizes well to memory-based lookup tables, resulting in high performance on an FPGA and in ModelSim. However, Verilator does not optimize AES as aggressively, resulting in poorer performance in our architecture. Second, the long build time of the ME circuit is caused by our conversion script taking 18 seconds to convert Verilator's nested tree output into a graph. The converter is written in an interpreted language (PHP) and does many complex array operations. This time could be significantly reduced by using a non-interpreted language with static memory allocation.

From Table I, at peak performance the RVE average number of PEs used is 54 and the average density is 18 ALMs/PE. We roughly estimate one PE is the silicon-area equivalent of 80 Stratix-III ALMs. The tools can scale the implementation to any number of PEs, and Table II shows the simulation speed results of our architecture at a target density of 80 ALMs/PE (meaning that the circuit is implemented in the same silicon area as an FPGA). The average speed is slightly less than 1/10th that of the FPGA. Continuing the scaling to our target of 10x FPGA density (800 ALMs/PE), the average speed is 3.9 MHz. At this density, all circuits smaller than 'FFT8' are implemented on a single PE. While this speed is not ideal, it

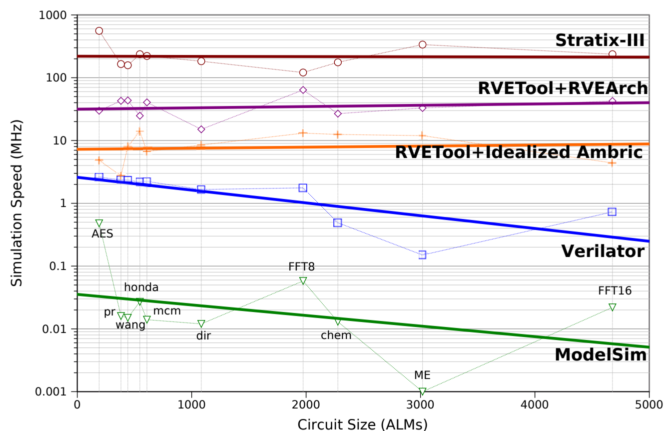

Fig. 3: Simulation speed comparison

TABLE I: Synthesis/compile time, simulation speed, and density comparisons

Circuit	Synthesis/compile time (s) Vltr	MS	Stx-III	Amb, RVE	Best simulation speed (MHz) Vltr	MS	Amb	Stx-III	RVE	Density at best speed Stx-III (ALMs)	RVE (PEs)	Ratio (ALMs/PE)
AES	3	1	148	4	2.61	0.48	4.8	559	29.9	191	56	3.4
pr	3	1	228	2	2.36	0.016	2.7	165	42.9	382	16	23.9
wang	3	1	182	2	2.33	0.015	8.0	158	43.5	442	16	27.6
honda	3	1	202	2	2.19	0.027	14.0	237	24.8	547	28	19.5
mcm	3	1	219	2	2.23	0.014	6.7	222	40.5	609	36	16.9
dir	3	1	372	5	1.66	0.012	8.5	183	15.1	1084	56	19.4
FFT8	3	1	207	4	1.76	0.058	13.1	121	63.8	1974	160	12.3
chem	4	1	477	3	0.49	0.013	12.5	176	26.8	2278	48	47.5
ME	3	1	277	27	0.15	0.001	12.0	337	33.0	3018	236	12.8
FFT16	3	1	790	8	0.73	0.022	4.4	237	42.6	4678	140	33.4
Geo. Mean	3	1	272	3.9	1.27	0.019	7.7	218	33.9	954.4	54	18.0
FFT8-human									167		64	
ME-human									250		64	

demonstrates that our tools can fold a design in space.

B. Platform Scalability

In this section, we demonstrate the ability of the toolflow to trade the number of available processing elements (PEs) with the effective user clock rate (simulation speed), avoiding the hard capacity limits imposed by commercial FPGAs.

Fig. 4 shows the simulation speed as the size of the array ranges from 1 PE (1×1) to 256 PEs (16×16). When a small number of PEs are used, a nearly linear relationship exists between the benchmark performance and number of PEs. Outside the linear region, each benchmark exhibits an optimum PE allocation where its performance is maximal. Beyond this maximum, performance is limited by inter-PE communications and the use of additional PEs is detrimental.

As the number of PEs increase, PE utilization decreases from 100% to below 5% in the 256-PE case. At 256 PEs, on average, 81% of the idle (NOP) instructions are due to the abundance of compute resources in comparison to the circuit size; the schedule cannot be smaller than the critical path. Waiting for data from inter-PE communications is the remaining 19%. At peak performance the utilization ranges from 15% to 70%, with no correlation to circuit size.

Of the benchmarks shown in Fig. 4, Motion Estimation (ME) scales to large processor counts because it is a large circuit and it exhibits chiefly nearest-neighbour communications. FFT16 also scales well, although it exhibits more complex communication patterns. In all cases, a benchmark's peak speedup is strongly correlated with its size: the smallest (wang and pr) show a peak performance at the fewest PEs. Real computational circuits would likely be much larger than any of these small benchmarks, so we anticipate the architecture is scalable far beyond 256 cores.

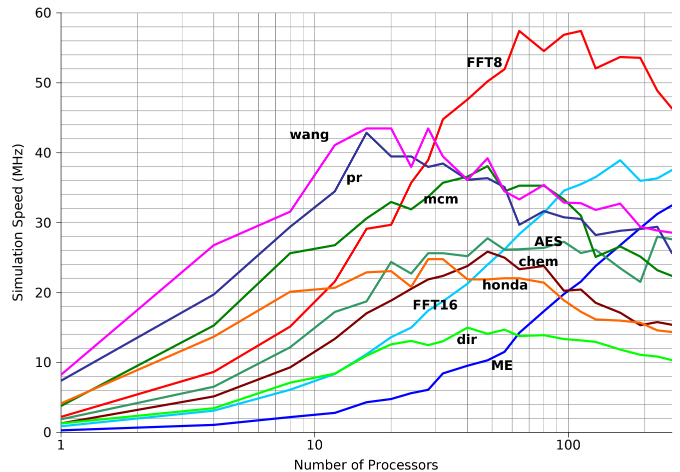

Fig. 4: Simulation speed for all circuits for 1-256 processors

TABLE II: Simulation speed at 80 and 800 ALMs/PE density

Circuit	ALMs	80 ALMs/PE Req'd PEs	80 ALMs/PE Speed (MHz)	800 ALMs/PE Req'd PEs	800 ALMs/PE Speed (MHz)
AES	191	4	6.6	1	1.9
pr	382	4	19.7	1	7.4
wang	442	4	26.8	1	8.3
honda	547	8	20.1	1	4.2
mcm	609	8	25.6	1	3.8
dir	1084	12	8.4	1	1.3
FFT8	1974	28	39.0	4	8.7
chem	2278	28	21.9	4	5.2
ME	3018	36	8.4	4	1.1
FFT16	4678	60	27.8	8	6.1
Geo. Mean	954.4	12.4	17.7	1.9	3.9

C. CAD Tool Efficiency

The simulation speed of each benchmark is determined by the benchmark itself, the CAD tools, and the architecture. To investigate how well our toolflow maps code onto each PE, we compare the scalability of automatically-generated and hand-written code for two of the larger benchmarks (FFT8 and ME).

Fig. 5 shows the scalability of each implementation, normalized to their single-core performance. The hand-written ME benchmark scales superlinearly on this architecture because the distributed implementation is able to avoid memory loads and stores by instead sending results to neighbouring PEs for processing. This figure demonstrates that RVEArch is able

to scale aggressively, and that RVETool is able to track this scalability curve up to a certain point.

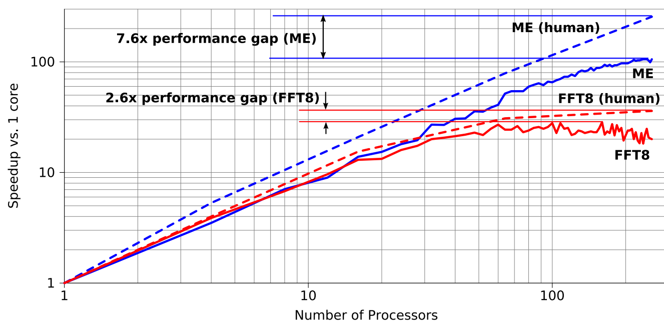

Fig. 5: Performance of human- and machine-generated bitstreams

At peak performances shown, FFT8 and ME show performance gaps of roughly 2.6x and 7.6x, respectively, between handcrafted and tool-produced results. At this stage, the tools are all first-generation algorithms; the focus has been on infrastructure development and integration, not performance or quality of results. We hope to reduce this performance gap as we refine the algorithms.

VI. Conclusions

In this paper, we presented a CAD toolflow (RVETool) that maps computational circuits expressed in Verilog onto an MPPA architecture. We evaluated the tools using a number of dataflow and DSP-type benchmarks, and demonstrated their performance relative to FPGAs and software simulators. The RVEArch architecture simulates computational circuits with better performance than software simulators, without incurring the long synthesis time of FPGA tools.

We explored the trade-off between density (number of PEs used) and simulation speed of RVEArch, demonstrating that it is effectively able to avoid the hard capacity limit of FPGAs using time multiplexing. We also examined RVETool's ability to effectively distribute a circuit simulation across a number of PEs. Although there is still significant room to improve our algorithms, neither the architecture nor the tools have been fully tuned and we have already exceeded our goals of 10x faster synthesis runtime and $\frac{1}{10}$th simulation performance of an FPGA. Additionally we have shown that our density goal of 10x an FPGA can be met through time multiplexing.

As more algorithms are converted to computational circuits, a means for fast synthesis and fast simulation becomes even more important. Besides reducing design time and risk, designing a circuit at a behavioural level does not require as much hardware design experience. This allows software designers to participate in the hardware design (and testing) process, further encouraging the development of computational circuits.

VII. Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Equipment donations by CMC Microsystems and Ambric are gratefully acknowledged. The authors would also like to thank Deming Chen for providing several benchmark circuits as well as Wilson Snyder, Duane Galbi, Paul Wasson, and the many additional contributors to the Verilator open source tool.

References

[1] A. Boukerche, "Conservative circuit simulation on multiprocessor machines," in *Proc. High Performance Computing*, 2000, pp. 415–424.
[2] D. Grant and G. Lemieux, "A spatial computing architecture for implementing computational circuits," in *Proc. MNRC*, Oct. 2008, pp. 41–44.
[3] D. Jones and D. Lewis, "A time-multiplexed FPGA architecture for logic emulation," in *Proc. Custom Integrated Circuits*, 1995, pp. 495–498.
[4] D. Lewis, "A hierarchical compiled-code event-driven logic simulator," *IEEE Trans. CAD*, vol. 10, no. 6, 1991.
[5] W. Snyder. (2007, June) Verilator-3.652. [Online]. Available: www.veripool.com/verilator_doc.pdf
[6] D. Greaves, "A Verilog to C compiler," in *Proc. Rapid System Prototyping (RSP)*, 2000, pp. 122–127.
[7] J. Ching. (2007, Mar.) VBS project homepage. [Online]. Available: www.flex.com/~jching/
[8] Symphony EDA. (2008, June) VHDL Simili v3.1 whitepaper. [Online]. Available: www.symphonyeda.com/white_paper.htm
[9] L. Soulé and T. Blank, "Parallel logic simulation on general purpose machines," in *DAC*, 1988, pp. 166–171.
[10] M. L. Bailey *et al.*, "Parallel logic simulation of VLSI systems," *ACM Computing Surveys*, vol. 26, no. 3, pp. 255–294, 1994.
[11] D. Webber and A. Sangiovanni-Vincentelli, "Circuit simulation on the connection machine," in *DAC*, June 1987, pp. 108–113.
[12] L. Li *et al.*, "DVS: An object-oriented framework for distributed Verilog simulation," in *Proc. Parallel and Distributed Simulation*, 2003, p. 173.
[13] W. Dong *et al.*, "WavePipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proc. Design Automation Conference*, 2008, pp. 238–243.
[14] T. Li *et al.*, "Design and implementation of a parallel Verilog simulator: PVSim," in *Proc. VLSI Design*, 2004, pp. 329–334.
[15] J. Jaeger. (2007, Dec.) Virtually every ASIC ends up an FPGA. [Online]. Available: www.eetimes.com/showArticle.jhtml?articleID=204702700
[16] Cadence. Incisive Enterprise Palladium series with Incisive XE software (datasheet).
[17] Mentor Graphics, "VStationPRO high-performance system verification (datasheet)," 2008.
[18] Richard Goering. (2004, Oct.) Cadence touts emulation/acceleration performance. [Online]. Available: www.eetimes.com/showArticle.jhtml?articleID=51200173
[19] B. Mei *et al.*, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. Field-Programmable Logic and Applications*, 2003, pp. 61–70.
[20] S. C. Goldstein *et al.*, "PipeRench: A coprocessor for streaming multimedia acceleration," in *ISCA*, 1999, pp. 28–39.
[21] S. C. Mishra, Mahim; Goldstein, "Virtualization on the Tartan reconfigurable architecture," in *FPL*, 2007, pp. 323–330.
[22] C. Ebeling *et al.*, "RaPiD - reconfigurable pipelined datapath," in *Proc. Field-Programmable Logic and Applications*, 1996, pp. 126–135.
[23] E. Caspi *et al.*, "Stream Computations Organized for Reconfigurable Execution (SCORE)," in *FPL*, 2000, pp. 605–614.
[24] T. R. Halfhill, "Ambric's new parallel processor," *Microprocessor Report*, vol. 20, no. 10, pp. 19–26, Oct. 2006.
[25] Tilera. (2007) Tile64 processor product brief. [Online]. Available: www.tilera.com/pdf/ProBrief_Tile64_Web.pdf
[26] A. S. Perinkulam, "Logic simulation using graphics processors," Master's thesis, University of Massachusetts Amherst, 2007.
[27] M. deLorimier *et al.*, "GraphStep: A System Architecture for Sparse-Graph Algorithms," in *FCCM*, 2006, pp. 143–151.
[28] G. Karypis *et al.*, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Trans. VLSI*, vol. 7, no. 1, pp. 69–79, Mar 1999.
[29] A. Marquardt *et al.*, "Timing-driven placement for fpgas," in *Proc. Field Programmable Gate Arrays*, 2000, pp. 203–213.
[30] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. FPL*, 1997, pp. 213–222.
[31] M. B. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," *IEEE Trans. VLSI*, vol. 3, no. 1, pp. 2–19, 1995.
[32] Altera Corporation, *Benchmark Designs for the Quartus University Interface Program (QUIP)*, 2006.