SPICE^2: Spatial Processors Interconnected for Concurrent Execution for accelerating the SPICE Circuit Simulator using an FPGA

Nachiket Kapre
Imperial College London
London, SW7 2AZ
nachiket@imperial.ac.uk

André DeHon
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

Abstract—**Spatial processing of sparse, irregular floating-point computation using a single FPGA enables up to an order of magnitude speedup (mean 2.8× speedup) over a conventional microprocessor for the SPICE circuit simulator. We decompose SPICE into its three constituent phases: Model-Evaluation, Sparse Matrix-Solve, and Iteration Control and parallelize each phase independently. We exploit data-parallel device evaluations in the Model-Evaluation phase, sparse dataflow parallelism in the Sparse Matrix-Solve phase and compose the complete design including the Iteration Control phase in a streaming fashion. We program the parallel architecture with a high-level, domain-specific framework that identifies, exposes and exploits parallelism available in the SPICE circuit simulator. Our design is optimized with an auto-tuner that can scale the design to use larger FPGA capacities without expert intervention and can even target other parallel architectures with the assistance of automated code-generation. This FPGA architecture is able to outperform conventional processors due to a combination of factors including high utilization of statically-scheduled resources, low-overhead dataflow scheduling of fine-grained tasks, and overlapped processing of the control algorithms. We demonstrate that we can independently accelerate Model-Evaluation by a mean factor of 6.5×(1.4–23×) across a range of non-linear device models and Matrix-Solve by 2.4×(0.6–13×) across various benchmark matrices while delivering a mean combined speedup of 2.8×(0.2–11×) for the composite design when comparing a Xilinx Virtex-6 LX760 (40nm) with an Intel Core i7 965 (45nm). With our high-level framework, we can also accelerate Single-Precision Model-Evaluation on NVIDIA GPUs, ATI GPUs, IBM Cell, and Sun Niagara 2 architectures. This paper summarizes work from our previous publications in [3]–[6].**

I. Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) is an analog circuit simulator that can take days or weeks of runtime on real-world problems. It models the analog behavior of semiconductor circuits using a compute-intensive non-linear differential equation solver. SPICE is notoriously difficult to parallelize due to its irregular, unpredictable compute structure, and a sloppy sequential description. It has been observed that less than 7% of the floating-point operations in SPICE are automatically vectorizable [1]. SPICE is part of the SPEC92-FP [2] benchmark collection which is a set of challenge problems for microprocessors. Modern FPGAs can efficiently support SPICE by exploiting spatial parallelism effectively using multiple floating-point operators coupled to hundreds of distributed, on-chip memories and interconnected by a flexible routing network. Our parallel SPICE solver exploits the distinct forms of parallelism when considering both phases of its operation as well as their integration.

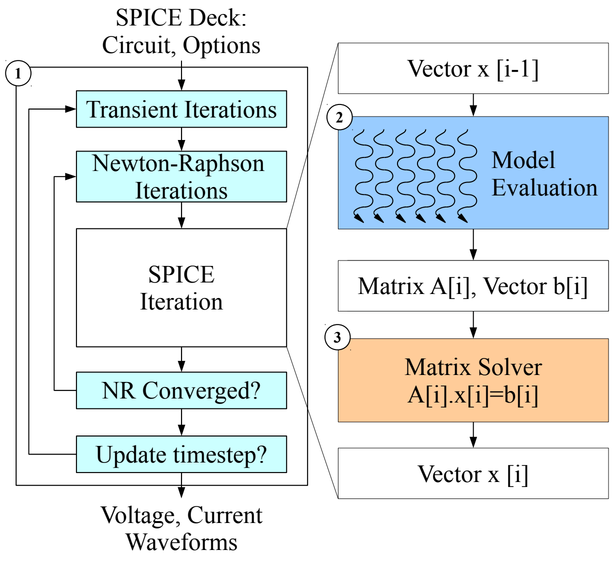


Fig. 1: Flowchart of a SPICE Simulator

As shown in Figure 1, a SPICE simulation is an iterative computation that consists of two key computationally-intensive phases per iteration: **Model Evaluation** (② in Figure 1) followed by **Matrix Solve** (③ in Figure 1). The iterations themselves are managed in the third phase of SPICE which is the **Iteration Controller** (① in Figure 1). In Figure 2, we show performance scaling trends for the sequential implementation of the open-source `spice3f5` package on an Intel Core i7 965 across a range of benchmark circuits. We observe that runtime scales as $O(N^{1.2})$ as we increase circuit size N. What contributes to this runtime? To understand this, we break down the contribution to total runtime from the different phases of SPICE in Figure 3. We observe that Model-Evaluation and Sparse Matrix-Solve phases account for over 90% of total SPICE runtime across the entire benchmark

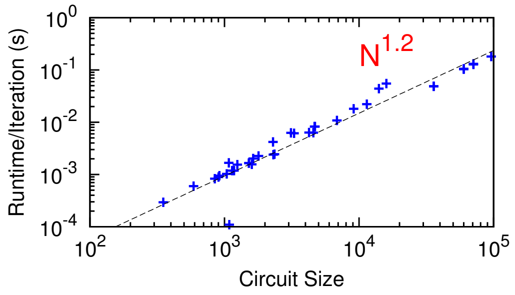


Fig. 2: Sequential Runtime Scaling of SPICE Simulator

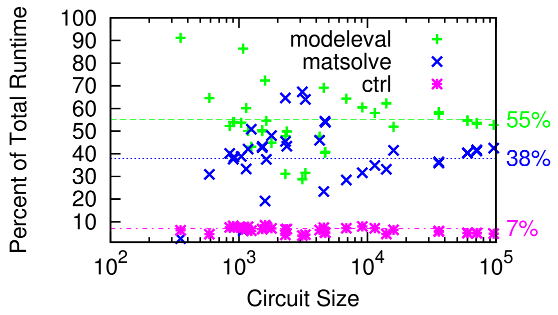


Fig. 3: Sequential Runtime Distribution of SPICE Simulator

set. For circuits dominated by non-linear devices, Model-Evaluation phase accounts for as much as 90% (55% average) of total runtime as Model Evaluation runtime scales linearly with the number of non-linear devices in the circuit. Simulations of circuits with a large number of resistors and capacitors (*i.e.* linear elements) generate large matrices and consequently the Sparse Matrix-Solve phase accounts for as much as 70% of runtime (38% average). This phase empirically scales as $O(N^{1.2})$ which explains the super-linear scaling of overall SPICE runtime. Finally, the Iteration Controller phase of SPICE take a small fraction (≈7%) of total runtime. To avoid Amdahl's law limits, we must parallelize all three phases.

This paper reviews the key results from our previous research [3]–[6]:

- Design and demonstration of a statically-scheduled VLIW (Very Large Instruction Word) architecture for accelerating Model-Evaluation of device models on FPGAs. [4]
- Design and demonstration of a dynamically-scheduled token dataflow architecture for accelerating Matrix-Solve on FPGAs when using the KLU Solver. [5]
- Design and demonstration of a hybrid VLIW FPGA architecture that combines static and dynamic scheduling for implementing the Iteration Controller phase of SPICE.
- Code-Generation and Auto-Tuning tools to map static, data-parallel, feed-forward, compute graphs to FPGAs, Multi-Core, Cell, GPUs, etc. [6]
- Composition of the SPICE simulator using a high-level, domain-specific framework that combines parallel descriptions in Verilog-AMS and SCORE [7] while extracting static dataflow graph from the KLU Matrix-Solve package.
- Quantitative empirical comparison of Model-Evaluation on a Xilinx V5LX330T, NVIDIA GT9600 and GT285 GPUs, ATI FireGL 5700 and Firestream 9270 GPUs, IBM PS3 Cell, Sun Niagara 2, and Intel Xeon 5160 across a variety of open-source Verilog-AMS models (single precision, 90nm technology). [6]
- Quantitative empirical comparison of Matrix-Solve on a Xilinx V6LX760 and an Intel Core i7 965 on a variety of benchmark matrices (45nm and 40nm process). [5]

II. Background

A. Summary of SPICE Algorithms

SPICE simulates the dynamic analog behavior of a circuit described by non-linear differential equations. SPICE circuit equations model the linear (*e.g.* resistors, capacitors, inductors) and non-linear (*e.g.* diodes, transistors) behavior of devices and the Kirchoff's Current Law at the different nodes and branches of the circuit. SPICE solves the non-linear differential circuit equations by computing small-signal linear operating-point approximations for the non-linear elements and discretizing continuous time behavior of time-varying elements until termination (① in Figure 1). The linearized system of equations is represented as a solution of $A\vec{x} = \vec{b}$, where A is the matrix of circuit conductances, $\vec{b}$ is the vector of known currents and voltage quantities and $\vec{x}$ is the vector of unknown voltages and branch currents. The simulator calculates entries in A and $\vec{b}$ from the device model equations that describe device transconductance (*e.g.*, Ohm's law for resistors, transistor I-V characteristics) in the **Model-Evaluation** phase (② in Figure 1). It then solves for $\vec{x}$ using a sparse linear matrix solver in the **Matrix-Solve** phase (③ in Figure 1).

B. SPICE Model-Evaluation

In the Model-Evaluation phase, the simulator computes conductances and currents through different elements of the circuit and updates corresponding entries in the matrix with those values. For resistors this needs to be done only once at the start of the simulation. For non-linear elements, the simulator must search for an operating-point using Newton-Raphson iterations that requires repeated evaluation of the model equations and matrix-solve multiple times per timestep as shown by the innermost loop in step ① of Figure 1. For time-varying components, the simulator must recalculate their contributions at each timestep based on voltages at several previous timesteps in the outer loop in step ① of Figure 1. We compile the device equations from a high-level domain-specific language called Verilog-AMS [8] which is more amenable to parallelization and optimization than existing C description in `spice3f5`. The compilation allows us to capture the device equations in an intermediate form suitable for performance optimizations and parallel mapping to potentially many target architectures.

C. SPICE Matrix Solve ($A\vec{x} = \vec{b}$)

Modern SPICE simulators use Modified Nodal Analysis (MNA) [9] to assemble circuit equations into the matrix A. We use the state-of-the-art sparse, direct KLU matrix solver [10] optimized for SPICE circuit simulation. The solver reorders the matrix A to minimize fillin using Block Triangular

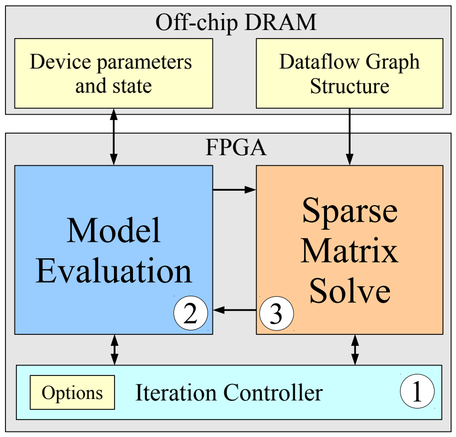


Fig. 4: FPGA Organization

Factorization (BTF) and Column Approximate Minimum Degree (COLAMD) techniques. The solver uses the left-looking Gilbert-Peierls [11] algorithm to compute the LU factors of the matrix such that $A = LU$. It calculates the unknown $\vec{x}$ using Front-Solve $L\vec{y} = \vec{b}$ and Back-Solve $U\vec{x} = \vec{y}$ operations. We use the refactorization operation in the KLU solver to further reduce the factorization runtimes for subsequent iterations. The approach uses the partial pivoting technique to generate a fixed non-zero structure in the LU factors at the start of the simulation (during first factorization). The preordering and symbolic analysis step computes the non-zero positions of the factors at the start while the refactorization and solve steps to solve the system of equations in each iteration.

D. SPICE Iteration Controller

The SPICE iteration controller is responsible for two kinds of iterative loops shown in Figure 1: (1) *inner loop*: linearization iterations for non-linear devices and (2) *outer loop*: adaptive time-stepping for time-varying devices. The Newton-Raphson algorithm is responsible for computing the linear operating-point for the non-linear devices like diodes and transistors. Additionally, an adaptive time-stepping algorithm based on truncation error calculation (Trapezoidal approximation, Gear approximation) is used for handling the time-varying devices like capacitors and inductors. The controller also implements the loops in a data-dependent manner using customized convergence conditions and local truncation error estimations.

E. Parallel Potential

Each phase of SPICE is characterized by a general parallel *pattern* that captures the structural characteristics of the underlying computation. Our parallel implementation exploits the characteristics using a compute organization tailored for each *pattern*.

The SPICE Model-Evaluation phase has high **data parallelism** consisting of thousands of independent device evaluations each requiring hundreds of floating-point operations. The simulator evaluates all devices in each iteration thereby generating a fixed-sized workload. There is a limited diversity in the number of non-linear device types in a simulation (*e.g.* typically only `diode` and `transistors` models). There is high pipeline parallelism within each device evaluation as operations can be represented as an acyclic feed-forward dataflow graph (DAG) with nodes representing operations and edges representing dependencies between the operations. These DAGs are static graphs that are known entirely in advance and do not change during the simulation enabling efficient offline scheduling of instructions. Individual device instances are predominantly characterized by constant parameters (e.g. V_{th}, Temperature, T_{ox}) that are determined by the CMOS process leaving only a handful of parameters that vary from device to device (e.g. W, L of device).

The Matrix-Solve phase of the KLU Gilbert-Peierls algorithm has irregular, **fine-grained task parallelism** during LU factorization. Since circuit elements tend to be connected to only a few other elements, the MNA circuit matrix is highly sparse (except high-fanout nets like power lines, etc). The underlying non-zero structure of the matrix is defined by the topology of the circuit and consequently remains unchanged throughout the duration of the simulation. We extract the static dataflow graph at the beginning of the simulation and exploit parallelism within the branches of the dataflow graph. We observe there are two forms of parallel structure in the Matrix-Solve dataflow graph that we can exploit in our parallel design: (1) factorization of independent columns organized into parallel subtrees and (2) fine-grained dataflow parallelism within the column. We empirically observe for our benchmark set of matrices that the number of floating-point operations in the Matrix-Solve computation scale as $O(N^{1.4})$ while the latency of the critical path through the compute graph scales as $O(N^{0.7})$. This suggests a parallel potential of $O(N^{0.7})$.

The Iteration Control phase of SPICE is dominated by **data-parallel** operations in convergence detection and truncation error-estimation which can be described effectively in a **streaming** fashion. The loop management logic for the Newton-Raphson and Timestepping iterations is control-intensive and highly irregular. While the Iteration Control phase only accounts for $\approx$7% of total sequential runtime, our parallel implementation takes care to efficiently implement this portion to avoid an Amdahl's Law bottleneck.

III. FPGA Architecture

As discussed earlier, we must parallelize all three phases of SPICE to get balanced total speedup. At a high-level we organize our parallel FPGA architecture into three blocks as shown in Figure 4. We develop a custom processing architecture for each phase of SPICE tailored to match the nature of parallelism in that phase. We integrate this heterogeneous architecture that combines VLIW, Dataflow and Streaming organizations to implement the complete design.

A. VLIW Architecture for Model-Evaluation [4]

The device equations can be represented as static, feed-forward dataflow graphs (see Section II-E). Fully-spatial implementations (circuit-style implementation of dataflow graphs) are too large to fit on current FPGAs and computation must be *virtualized* over limited resources. These graphs contain a diverse set of floating-point operators such as adds, multiplies, divides, square-roots, exponentials and logarithms. We map these graphs to custom VLIW "*processing tiles*". We statically schedule these resources offline in VLIW [13] fashion and perform loop-unrolling, tiling and software pipelining optimizations to improve performance. Each *tile* in the virtualized architecture consists of a heterogeneous set of floating-point operators coupled to local, high-bandwidth memories and interconnected to other operators through a communication network. We also include spatial implementations of elementary functions like `log`, `exp` that may require multiple cycles on a processor. We build pipelined datapaths customized for Model Evaluation which is not possible with CPUs or GPUs. In each *tile*, we choose an operator mix per *tile* proportional to the frequency of occurrence of those floating-point operations in the graph. Since we use a statically-scheduled fat tree [12] to connect these operators, we also tune the interconnect bandwidth to reflect communication requirements between the operators.

B. Token-Dataflow Architecture for Matrix-Solve [5]

The Sparse Matrix-Solve computation can be represented as a sparse, irregular dataflow graph that is fixed at the beginning of the simulation. We recognize that static online scheduling of this parallel structure may be infeasible due to the prohibitively large size of these sparse matrix factorization graphs (millions of nodes and edges where nodes are floating-point operations and edges are dependencies). Hence, we organize our architecture as a dynamically-scheduled Token Dataflow [14] machine. The architecture consists of multiple interconnected "*Processing Elements*" (PEs) each holding hundreds to thousands of graph nodes. Each PE can fire a node dynamically based on a fine-grained dataflow triggering rule. The *Dataflow Trigger* in the PE keeps track of ready nodes and issues operations when the nodes have received all inputs. Tokens of data representing dataflow dependencies are routed between the PEs over a packet-switched network. For very large graphs, we partition the graph and perform static prefetch of the subgraphs from external DRAM. This is possible since the graph is completely feed forward.

C. Hybrid VLIW Architecture for Iteration Control

[3] The Iteration Control implementation is a hybrid VLIW architecture that is mostly similar to the Model-Evaluation design. The data-parallel convergence detection and truncation error estimation operations are statically scheduled in VLIW fashion. In contrast, the loop control state machine transition operations are evaluated dynamically making this a hybrid VLIW design that combines static and dynamic scheduling.

IV. Methodology

We now explain the methodology and framework setup for mapping SPICE simulations to FPGAs. We show the complete FPGA mapping flow in Figure 5. At a high level, our FPGA flow is organized into different paths that are customized for the specific SPICE phase. Our mapping flow is further broken down into three key stages: Input, Compilation/CAD and Hardware. Additionally, we separate the steps into offline and runtime operations depending on the binding time of input.

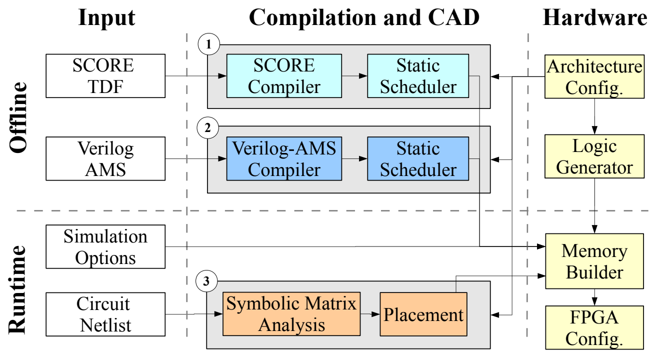


Fig. 5: FPGA SPICE Toolflow

We map this parallelism to the FPGA fabric using customized compute organizations described in Section III. Instead of recompiling the entire FPGA configuration for each circuit, we statically generate a hardware logic configuration for the parallel SPICE simulator that is shared across all circuits. We must only build customized memory images for each circuit dynamically at runtime.

Offline Logic Configuration: We generate the logic for implementing the VLIW, Dataflow and Streaming architectures by choosing an appropriate balance of area and memory resources through an area-time tradeoff analysis. We use an auto-tuner to select this balance and tune implementation parameters through an exhaustive exploration of the design space. The FPGA configuration includes the VLIW programming information for the PEs and switches of the Model-Evaluation and Iteration Control processing elements (output of the "Static Scheduler" blocks shown in Figure 5).

Runtime Memory Configuration: For each circuit, we must program memory resources to store the circuit-specific variables and data-structures relevant for the simulation. For the non-linear devices and independent sources, we store the device-specific constant parameters from the circuit netlist in FPGA onchip memory or offchip DRAM memory if necessary. We generate and distribute the sparse dataflow graph across the Matrix-Solve processing elements (shown by the "Placement" block in Figure 5) and store the graph in offchip DRAM memory when it does not fit onchip capacity. Finally, we load a few simulation control parameters (*e.g.* `abstol`, `reltol`, `final_time`) to help the Iteration Control phase declare convergence and termination of the simulation.

We use spatial implementations of individual floating-point *add*, *multiply*, *divide* and *square-root* operators from the Xilinx Floating-Point library in CoreGen [15]. For the *exp* and *log* operators we use FPLibrary from Arénaire [16] group. For the Model-Evaluation and Iteration Control architectures, we interconnect the operators using a time-multiplexed butterfly-fat-tree (BFT) network that routes 64-bit doubles or 32-bit floats using time-multiplexed switches. For the Matrix-Solve architecture, we interconnect the floating-point operators using a bidirectional mesh packet-switched network that routes 84-bit 1-flit packets (64-bit double and 20-bit node address). We synthesize and implement a sample double-precision 8-operator design for the `bsim3` model (250 MHz) and a double-precision 4-PE Matrix-Solve design (250 MHz) on a Xilinx Virtex-5 device using Synplify Pro 9.6.1 and Xilinx ISE 10.1. We report cycle counts from time-multiplexed schedule (Model-Evaluation and Iteration Controller) and a cycle-accurate simulation (Matrix-Solve). We use CPU runtimes for the open-source `spice3f5` package coupled with the KLU Solver running on an Intel Core i7 965.

V. Evaluation

We now report the achieved performance and energy requirements of our parallel SPICE implementation. We show total speedups for the SPICE solver when comparing an Intel Core i7 965 with a Virtex-6 LX760 FPGA in Figure 6a. We observe a mean speedup of 2.8× across our benchmark set with a peak speedup of 11× for the largest benchmark. We also show the ratio of energy consumption between the two architectures in Figure 6b. We estimate power consumption of the FPGA using the Xilinx XPower tool assuming 20% activity on the Flip-Flops, Onchip-Memory ports and external IO ports. When comparing energy consumption, the FPGA is able to deliver these speedups while consuming much less energy. We observe that the FPGA consumes up to 40.9×(geomean 8.9×) lower energy than the microprocessor implementation.

A. Model-Evaluation

In Figure 7a, we compare the performance achieved for a double-precision implementation of Model-Evaluation on a quad-core Intel Core i7 965 (loop-unrolled and multi-threaded) with that achieved on a single Virtex-6 LX760T (loop-unrolled, tiled and statically scheduled). We observe speedups between 1.4×–23× (mean 6.5×) across our non-linear device model benchmarks. We are able to deliver these speedups due to higher utilization of statically-scheduled floating-point resources, explicit routing of graph dependencies over physical interconnect and spatial implementation of elementary floating-point functions (*e.g.* `exp`, `log`). The FPGA is able to achieve higher speedups for smaller, simpler devices than larger, complex ones. Smaller compute graphs have fewer edges requiring smaller interconnect context and a lower memory footprint per unroll. We compare single-precision implementations on 65nm generation devices in Figure 7b and observe much higher speedups of 4.5–123× for a Virtex-5 LX330, 10–64× for an NVIDIA 9600GT GPU, 0.4–6× for an ATI FireGL 5700 GPU, 3.8–16× for an IBM Cell and 0.4–1.4× for a Sun Niagara 2. The increased FPGA speedups are due to higher floating-point processing capacity made possible by smaller single-precision FPGA operators, smaller network and lower storage requirements. This additional speedup is only possible if we relax the SPICE convergence conditions by reducing tolerances (acceptable for many scenarios).

B. Matrix-Solve

In Figure 8, we compare double-precision performance of our FPGA architecture implemented on a Virtex-6 LX760 with an Intel Core i7 965. We observe speedups of 0.6–13.4× (geomean 2.4×) for the 25-PE Virtex-6 LX760 mapping over a range of benchmark matrices. Our FPGA implementation allow efficient processing of the fine-grained factorization operations which can be synchronized at the granularity of individual floating-point operations. Additional placement for locality is vital for delivering high speedups for irregular sparse matrix graphs.

C. Iteration Control

Our spatial FPGA implementation of the Iteration-Control phase delivers modest speedups of 1.07–3.3× (mean 2.1×). This allows us to improve overall mean SPICE speedups from 2.4× (sequential Iteration Control on CPU) to 2.6× (parallel Iteration Control on FPGA).

VI. Future Work

We now identify additional opportunities for parallelizing SPICE further and improving the FPGA design. The key performance bottleneck of the current design is the Dataflow implementation of the Sparse Matrix-Solve phase of SPICE. We will explore newer domain-decomposition approaches for exposing more coarse-grained parallelism and associative reformulation for improved scalability. We can redesign the Model-Evaluation datapaths with lower precision while satisfying accuracy requirements to obtain additional acceleration at lower cost. Finally, with larger FPGAs, we can achieve scalable speedups by converting additional resources into performance far more effectively than competing architectures.

VII. Conclusions

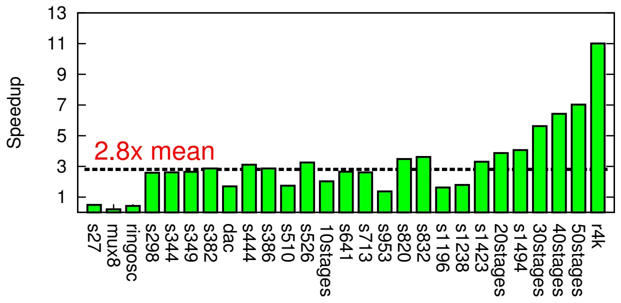
We show how to use FPGAs to accelerate the SPICE circuit simulator up to an order of magnitude (mean 2.8×) when comparing a Xilinx Virtex-6 LX760 with an Intel Core i7 965. We were able to deliver these speedups by exposing available parallelism in all phases of SPICE using a high-level, domain-specific framework and customizing FPGA hardware to match the nature of parallelism in each phase. The tools and techniques we develop for mapping SPICE to FPGAs are general and applicable to a broader range of designs. We believe these ideas are applicable where computation is characterized by static, data-parallel processing and in cases where the algorithm operates on sparse, irregular data structures. We expect such high-level approaches based on exploiting spatial parallelism to become important for improving performance and energy-efficiency for a variety of important, computationally-intensive problems.

References

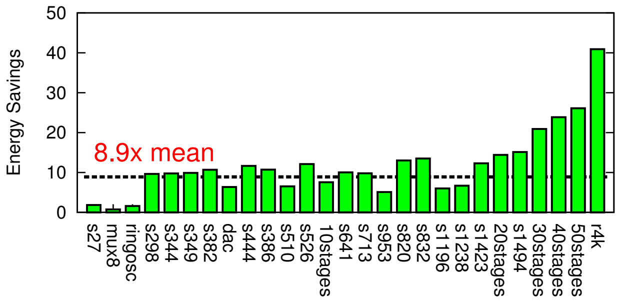
[1] J. Hennesey and D. Patterson, *Computer Architecture A Quantitative Approach*, 2nd ed. Morgan Kauffman, 1996. I

[2] S. P. E. Corporation, "SPEC CFP92 Benchmarks," 1992. I

[3] N. Kapre, "SPICE2 - A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator," PhD Thesis, California Institute of Technology, 2010. (document), I, III-C

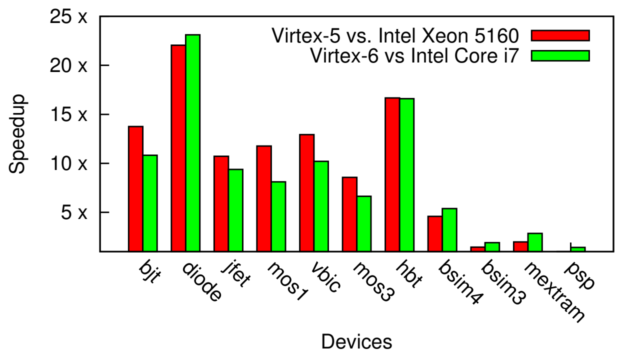


(a) Total Speedup

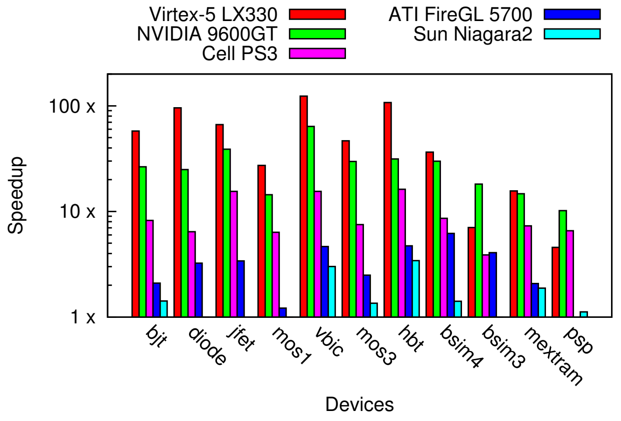


(b) Energy Ratio

Fig. 6: Comparing Xilinx Virtex-6 LX760 FPGA (40nm) and Intel Core i7 965 (45nm) Implementations



(a) Double-Precision (CPU vs. FPGA)



(b) Single-Precision (vs. Xeon 5160)

Fig. 7: Speedups for Model-Evaluation

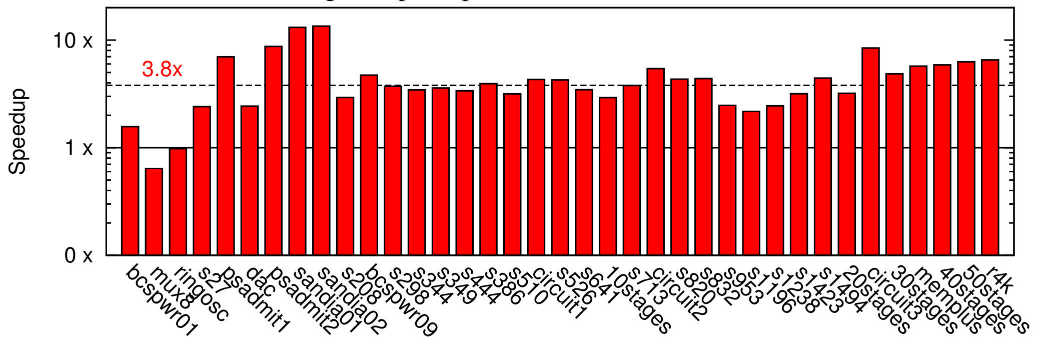


Fig. 8: Speedups for Double-Precision Matrix-Solve (vs. Core i7 965)

[4] N. Kapre and A. DeHon, "Accelerating SPICE Model-Evaluation using FPGAs," in *IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 37–44. I, III-A

[5] ——, "Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs," in *International Conference on Field-Programmable Technology*, 2009, pp. 190–198. (document), I, III-B

[6] ——, "Performance comparison of single-precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core processors," in *International Conference on Field Programmable Logic and Applications*, 2009, pp. 65–72. (document), I

[7] E. Caspi, "Design Automation for Streaming Systems," PhD Thesis, University of California, Berkeley, 2005. I

[8] L. Lemaitre, G. Coram, C. McAndrew, K. Kundert, M. Inc, and S. Geneva, "Extensions to Verilog-A to support compact device modeling," in *Proceedings of the Behavioral Modeling and Simulation Conference*, 2003, pp. 7–8. II-B

[9] Chung-Wen Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, 1975. II-C

[10] E. Natarajan, "KLU A high performance sparse linear solver for circuit simulation problems," Master's Thesis, University of Florida Gainesville, 2005. II-C

[11] J. Gilbert and T. Peierls, "Sparse Partial Pivoting in Time Proportional to Arithmetic Operations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 5, pp. 862–874, 1988. II-C

[12] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216. III-A

[13] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer*, vol. 17, no. 7, pp. 45–53, 1984. III-A

[14] G. Papadopoulos and D. Culler, "Monsoon: an explicit token-store architecture," *Proceedings of the Annual International Symposium on Computer Architecture*, vol. 18, no. 3a, pp. 82–91, 1990. III-B

[15] Xilinx, "Floating-Point Operator v5.0," pp. 1–31, 2009. IV

[16] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," *Proceedings of the International ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, p. 260, 2008. IV