SPICE2: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA

Kapre, Nachiket; DeHon, André

2012

Kapre, N., & DeHon, A. (2012). SPICE2: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 31(1), 9-22.

https://hdl.handle.net/10356/81197

https://doi.org/10.1109/TCAD.2011.2173199

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

PAR-CAD 2010: SPICE^2: Spatial Processors Interconnected for Concurrent Execution for accelerating the SPICE Circuit Simulator using an FPGA

Journal:	*Transactions on Computer-Aided Design of Integrated Circuits and Systems*
Manuscript ID:	TCAD-2011-0032
Manuscript Type:	Special Section Full Paper
Date Submitted by the Author:	31-Jan-2011
Complete List of Authors:	Kapre, Nachiket; Imperial College London, Electrical and Electronic Engineering DeHon, Andre
Keywords:	architecture, FPGA, reconfigurable logic, simulation

Special Section on PAR-CAD 2011

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

Nachiket Kapre, *Member, IEEE,* André DeHon, *Member, IEEE*

Abstract*—Spatial processing of sparse, irregular, double-precision 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 develop a parallel, FPGA-based, heterogeneous architecture customized for accelerating the SPICE simulator to deliver this speedup. To properly parallelize the complete simulator, we decompose SPICE into its three constituent phases – Model-Evaluation, Sparse Matrix-Solve, and Iteration Control – and customize a spatial architecture for each phase independently. Our heterogeneous FPGA organization mixes VLIW, Dataflow and Streaming architectures into a cohesive, unified design to match the parallel patterns exposed by our programming framework. 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 streaming, 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). We also estimate mean energy savings of 8.9×(up to 40.9×) when comparing a Xilinx Virtex-6 LX760 with an Intel Core i7 965. 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.

I. Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) is an analog circuit simulator used extensively in industry to simulate and verify operation of silicon circuits. It models the analog behavior of semiconductor circuits using a compute-intensive, non-linear, differential equation solver. This can take days or weeks of runtime on real-world circuits. SPICE is notoriously difficult to parallelize due to its irregular compute structure, and a sloppy sequential description [36]. It has been observed that less than 7% of the floating-point operations in SPICE are automatically vectorizable [18].

Spatial parallelism provides a suitable framework for constructing accelerators for challenging problems like SPICE. It offers a natural way to express the heterogeneous computational structure in SPICE and exposes the inherent parallelism available in the problem. Furthermore, modern FPGAs can configured to efficiently support spatial parallelism with multiple floating-point operators coupled to hundreds of distributed, on-chip memories and interconnected by a flexible routing network. In Table I, we observe that modern FPGAs can match and even surpass the peak floating-point capacity of modern multi-core processors while dissipating far less power. Spatial parallelism allows us to configure the FPGA to deliver a higher fraction of this floating-point peak through a combination of careful static scheduling and low-overhead distributed processing.

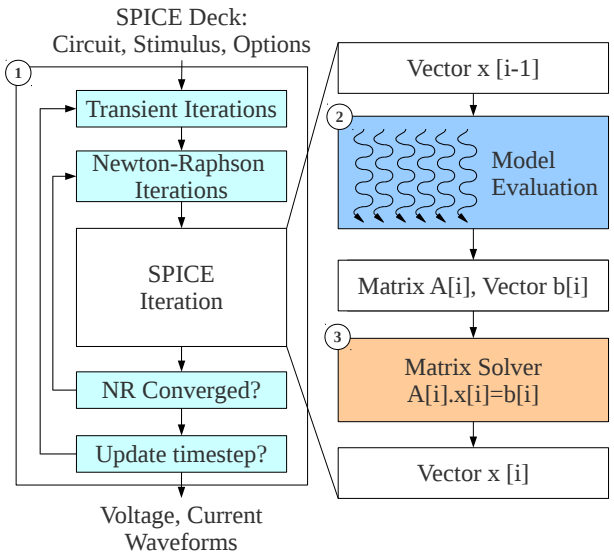


Fig. 1: Flowchart of a SPICE Simulator

As shown in Figure 1, a SPICE simulation accepts a netlist description of the circuit to be simulated along with input stimulus and returns the response of the circuit in the form of output analog waveforms. The simulation algorithm discretizes circuit response and repeatedly solves circuit equations at each discrete step to generate output waveforms. We also show an abstract internal representation of the simulation algorithm in Figure 1. This iterative simulation consists of two key computationally-intensive phases per iteration: **Model Evaluation** (② in Figure 1) followed by **Matrix Solve** (③ in Figure 1). This organization allows the non-linear, differential equation solver to be simplified to a system of linear equations $A\vec{x} = \vec{b}$ which is handled in the **Matrix Solve** phase. The non-linear, time-varying circuit elements are linearized using a Newton-Raphson loop and discretized using Trapezoidal integration in the **Model-Evaluation phase**. These two loops are managed in the third phase of SPICE which is the **Iteration**

N. Kapre is with Imperial College London. This work was performed when he was with Computer Science Department, California Institute of Technology.
A. DeHon is with the University of Pennsylvania.

Chip	Tech. (nm)	Clock (GHz)	Peak GFLOPS (Double)	Power (Watts)
Intel Core i7 965	45	3.2	25	130
Xilinx Virtex-6 LX760	40	0.2	26	20–30

TABLE I: Peak Floating-Point Throughputs (Double-Precision)

Controller (① in Figure 1). A well-balanced, scalable, parallel architecture must accelerate all three phases of SPICE.

This paper reviews and expands on our previous research [23]–[25]. We expand on the previous publications by delivering a parallel solution for the Iteration Control phase and integrating the complete solver. Our integrated SPICE simulator is mapped onto a heterogeneous parallel architecture using a high-level, domain-specific framework that combines parallel descriptions in Verilog-AMS and SCORE [7], [13] while extracting static dataflow graph from the KLU [37] Matrix-Solve package.

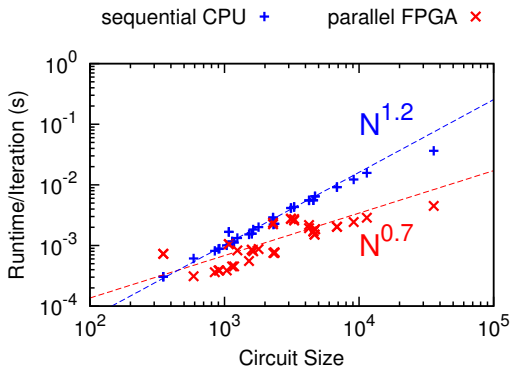
- We show how to accelerate the Model-Evaluation phase of SPICE using an FPGA [23]. We also perform a quantitative empirical comparison of Model-Evaluation on a Xilinx V5LX330T and V6LX760T, NVIDIA GT9600 and GT285 GPUs, ATI FireGL 5700 and Firestream 9270 GPUs, IBM PS3 Cell, Sun Niagara 2, and Intel Xeon 5160 and Core i7 965 across different Verilog-AMS models [25].
- We show how to implement the Sparse Matrix-Solve phase of SPICE on an FPGA [24] when using the KLU solver. Additionally, we perform a 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).
- We show how to design a parallel architecture for implementing the Iteration Controller phase of SPICE. We quantify the performance of the Iteration Control phase implemented on a Microblaze with a spatial hybrid VLIW implementation on a Xilinx V6LX760 FPGA for a variety of SPICE circuits.
- We compose and integrate the complete SPICE solver using a Xilinx Virtex 6 LX760 FPGA and compare it to an Intel Core i7 965 processor for performance and energy.

The rest of this paper is organized as follows. We explain the underlying computational characteristics of SPICE in Section II. Next, we discuss suitable FPGA compute organizations for implementing the SPICE computation in Section III. In Section IV we provide details about our FPGA compilation framework and cost model. We then outline the experimental framework used to perform a fair and robust comparison of multiple implementations in Section V. We present results of our experimental evaluation in Section VI. Finally we identify opportunities for future work and wrap up with some key insights and lessons in Section VII and Section VIII respectively.

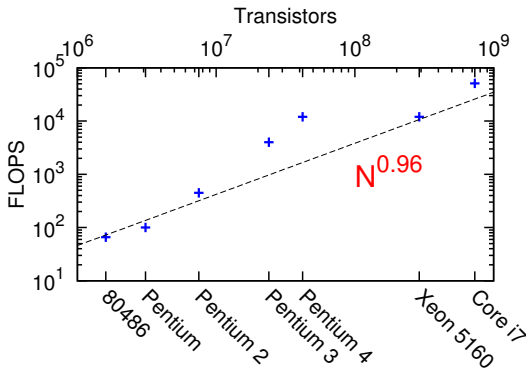
II. Background

A. Summary of SPICE Algorithms

SPICE simulates the dynamic analog behavior of a circuit described by non-linear differential equations. SPICE solves the non-linear differential circuit equations by computing



(a) Sequential Runtime Scaling of SPICE Simulator



(b) Peak FLOPS scaling of Intel CPUs

Fig. 2: Scaling Trends for CPU FLOPS and `spice3f5` runtime

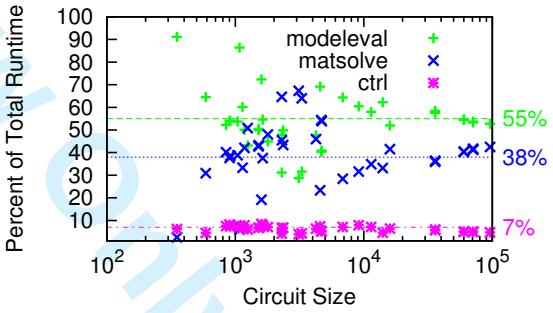


Fig. 3: Sequential Runtime Distribution of SPICE Simulator

small-signal linear operating-point approximations for the non-linear and time-varying elements until termination (① in Figure 1). The linearized system of equations is represented as a solution of $A\vec{x} = \vec{b}$ handled in the **Matrix-Solve** phase (③ in Figure 1), 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).

B. SPICE Performance Analysis

Since the SPICE simulation is an iterative algorithm, we can understand key characteristics of the complete simulation by analyzing a single iteration. In Figure 2a, we show performance scaling trends for a single iteration of the SPICE

solver for two scenarios. First we show data for sequential implementation of the open-source `spice3f5` package on an Intel Core i7 965 across a range of benchmark circuits shown later in Appendix A. We also show data for our parallel FPGA implementation across the same benchmarks. We observe that sequential runtime for one iteration scales as $O(N^{1.2})$ as we increase circuit size, N, while parallel runtime scales faster as $O(N^{0.7})$. In Figure 2b, we show the peak floating-point scaling trends of Intel CPUs obtained from Intel datasheets to contrast against SPICE runtime trends. We observe that the sequential CPU FLOPS (peak) have barely scaled as $O(N)$ while `spice3f5` runtimes have scaled faster as $O(N^{1.2})$. While Moore's Law continues to deliver increasing circuit sizes (for both circuit simulation and CPU processing), the CPU floating-point peaks have been unable to keep up with the super-linear scaling rate of simulation times. This means there is a widening performance gap between CPU capacity and SPICE runtime. In contrast, the FPGA processing capabilities shown in Table I can be organized entirely in parallel thereby allowing performance to scale as the critical latency of the computation $O(N^{0.7})$ as shown in Figure 2a.

To further understand SPICE performance trends, 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 set. For circuits dominated by non-linear devices, Model-Evaluation phase accounts for as much as 90% (55% average) of total runtime since the runtime of this phase 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 comprises a small but non-trivial fraction (≈7%) of total runtime. Thus, our parallel FPGA architecture must parallelize all three phases of SPICE.

C. 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 a linear solve multiple times per time-step 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 [29] which is more amenable to parallelization and optimization than existing C description in `spice3f5`. Verilog-AMS descriptions clearly identify the inputs and outputs for the device equations and also provide a mechanism to specify constant parameters easily. In contrast, the `spice3f5` descriptions make extensive use of pointers into shared data-structures that are harder to analyze and do no provide a clean way to separate variables from constants. The Verilog-AMS compilation also allows us to capture the device equations in an intermediate form suitable for performance optimizations and parallel mapping to many potent target architectures.

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. In Figure 4, we plot the number of floating-point operations and the latency of evaluation (floating-point operations along critical path from input to output) as a function of the number of non-linear elements in the circuit. Since each device contributes a fixed number of floating-point operations per instance, we see a linear growth in the number of operations. However, the latency of evaluation stays constant since each evaluation is completely independent and can be evaluated simultaneously. This highly data-parallel computations is suitable for implementation on FPGAs, GPUs, as well as multi-cores. We compare these implementations later in Section VI. Additionally, we make other structural observations that will help simplify and enhance our FPGA mapping. We note that 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). This suggests specialization potential through constant-folding, identity simplification and other compiler optimizations that can eliminate repeated, unnecessary work. We later show the optimized instruction counts for different non-linear device models in Table IV.

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

Modern SPICE simulators use Modified Nodal Analysis (MNA) [8] to assemble circuit equations into the matrix A. This generates highly-sparse, asymmetric matrices which are processed using sparse, direct LU factorization techniques to deliver robust simulation results. A parallel implementation of Matrix Solve should avoid dynamic changes to the matrix data-structures to enable an efficient mapping. Large, dynamically-changing compute structures are difficult to distribute for parallel evaluation. Unfortunately, the default matrix package in `spice3f5`, Sparse 1.3, has a highly-dynamic nature which

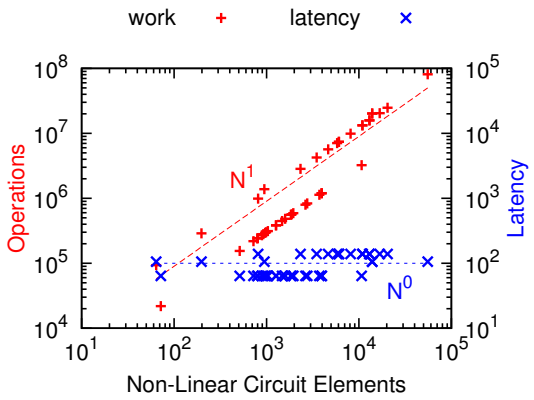


Fig. 4: Work vs. Latency for Model-Evaluation

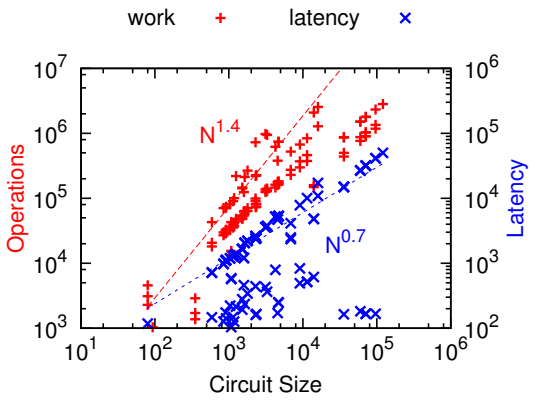


Fig. 5: Work vs. Latency for Matrix-Solve

changes the factorization compute structure at each SPICE iteration. We are forced to sequentially discover suitable pivot positions that may change in each iteration. Our approach uses the state-of-the-art KLU matrix solver [37] optimized for SPICE circuit simulation and avoids per-iteration changes to the matrix structures. The static non-zero pattern enables reuse of the matrix factorization graph across all SPICE iterations and allows us to perform a one-time distribution of computation across a parallel architecture. The solver reorders the matrix A to minimize fillin using Block Triangular Factorization (BTF) and Column Approximate Minimum Degree (COLAMD) techniques. It then uses the left-looking Gilbert-Peierls [16] algorithm to compute the LU factors of the matrix column-by-column such that $A = LU$. Finally, it calculates the unknown $\vec{x}$ using Front-Solve $L\vec{y} = \vec{b}$ and Back-Solve $U\vec{x} = \vec{y}$ operations. The KLU 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). This is followed by preordering and symbolic analysis phase to compute non-zero positions of the LU factors. For subsequent iterations we perform refactorization which reuses the non-zero position information to perform a numerical factorization.

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. Upon analysis, we observe that 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. In Figure 5, we plot the number of floating-point operations in the factorization and latency of evaluation as a function of the size of the circuit. We observe 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 $O(N^{0.7})$ which can be realized by distributing the dataflow graph across ideal parallel hardware (*e.g.* no communication delays, perfect distribution, unlimited internal processing bandwidth).

E. SPICE Iteration Controller

The SPICE iteration controller shown in Figure 1 is responsible for two kinds of iterative loops: (1) *inner loop*: Newton-Raphson 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 implements customized convergence conditions and local truncation error estimations that determine how the transient analysis state machines are advanced at runtime in a data-dependent manner. The state-machine and breakpoint-processing logic are highly data-dependent and determine the total number of SPICE iterations required for the complete simulation.

As we saw earlier in Figure 3, the Iteration Control phase only accounts for ≈7% of total sequential runtime. However, our parallel SPICE implementation takes care to efficiently implement this portion to avoid an Amdahl's Law bottleneck. We show the danger of ignoring this phase for parallelization in Figure 6 which shows the runtime breakdown for the `r4k` netlist in different implementation scenarios. We observe that we can get a speedup of ≈6× when parallelizing the Model-Evaluation and Sparse Matrix-Solve phase of SPICE (parallel FPGA runtimes obtained from Section VI). If we parallelize the Iteration Control phase, we can improve overall speedup to ≈9×. 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. We can capture this structure effectively using a streaming framework that can represent the data-parallel as well as control-intensive computation simultaneously.

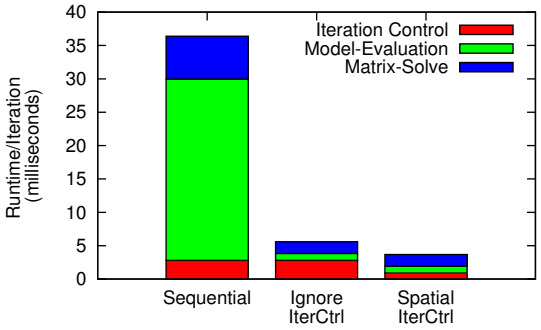


Fig. 6: Parallel Potential for Iteration Control (`r4k` netlist)

F. Historical Review

We now review the various studies and research projects in the past three and a half decades that have attempted to build parallel SPICE systems. Some of these studies accelerate SPICE by devoting expensive hardware resources to squeeze additional performance while others reorganize the computation to use lower-precision evaluation that is easier to parallelize. Our approach expands on certain ideas from the past while delivering a cheaper, SPICE-accurate accelerator.

One of the earliest SPICE parallelization studies [19] extracts the static triangulation graph of the tiny circuit matrices from that era and does not consider communication costs when exploiting parallelism. Awsim-3 [30], [31] uses a compiled code approach and a special-purpose system with lower-precision, table-lookup Model-Evaluation to provide a speedup of 560× over a Sun 3/60. However, a bulk of these speedups are due to dedicated hardware floating-point unit since the Sun 3/60 implements floating-point in software (tens of cycles/-operation). Additionally, table-lookup approximations avoid a large fraction of floating-point work resulting in a simulation with accuracy tradeoffs. A message-passing, parallel SPICE implementation [20] on an expensive, 40-node SGI Origin 2000 supercomputer (MIPS R10K processors) was able to speedup SPICE for certain specialized benchmarks by 24×. More recently, in [28], a multi-threaded version of SPICE is developed using PThreads. It achieves a speedup of 5× using 8 SMP (Symmetric Multi-Processors) on a small benchmark set which is amenable to parallel matrix factorization. GPUs have been used to speedup the data-parallel Model-Evaluation phase of SPICE by 50× [2] (double-precision on ATI GPU) or 32× [17] (lower-accuracy, single-precision on NVIDIA GPU) but can accelerate the complete SPICE simulator in tandem with the CPU by 3× for the 2-chip GPU-CPU processing system. FPGAs have enjoyed limited use for accelerating SPICE due to scarce FPGA resources and lack of tools and methodology for attacking a problem of this magnitude. A compiled code, partial evaluation approach for timing simulation (lower precision than SPICE) using FPGAs was demonstrated in [46] where the processing architecture was customized for each SPICE circuit using fixed-point computation. Recent approaches [40] have use coarse-grained domain-decomposition techniques shown how to parallelize SPICE by 31×–870×(mean 119×) across a 32 processor grid at SPICE-level accuracy.

Our FPGA-based approach accelerates the SPICE computation while retaining the accuracy of `spice3f5` and developing an economical single-FPGA system for accelerating SPICE. We reuse the idea of compiled-code methodology popularized by many previous approaches. We can compose our technique with KLU-based domain-decomposition approaches [40] to scale to even large problems and system sizes *e.g.* multi-FPGA systems. Additionally, we can integrate lower-precision techniques *e.g.* table-lookup into our mapping flow to get cumulative benefits.

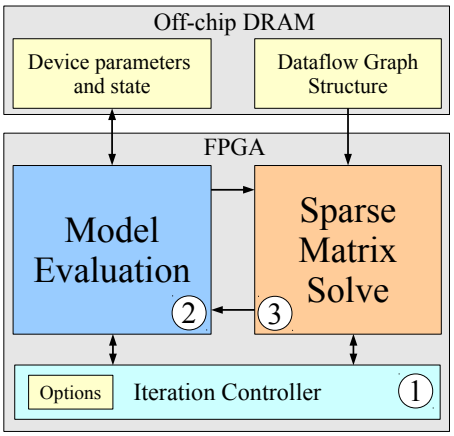


Fig. 7: FPGA Organization

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 7. We develop a custom processing architecture for each phase of SPICE tailored to match the nature of parallelism in that phase. In Figure 8 we show a cartoon internal representation of the different compute organizations in each phase. We note that the Model-Evaluation and Iteration-Control organizations are statically scheduled and store the statically generated program context. In contrast, the Sparse Matrix-Solve organization is dynamically scheduled and routes data between the floating-point operators using a dynamic packet-switched network. Furthermore, our Iteration Control architecture support streams for interconnecting the complete design. We now look at each design style and show how we selected and configured this compute organization.

A. VLIW Architecture for Model-Evaluation

The device equations can be represented as static, feed-forward dataflow graphs. Fully-spatial implementations (circuit-style implementation of dataflow graphs) are too large to fit on current FPGAs and computation must be *time-shared* 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*" with spatial implementation of the floating-point operators. Pipelined, spatial FPGA implementations of elementary functions like `log`, `exp` operate at a high throughput of one evaluation/cycle (250 MHz) while the processor implementations require 100s

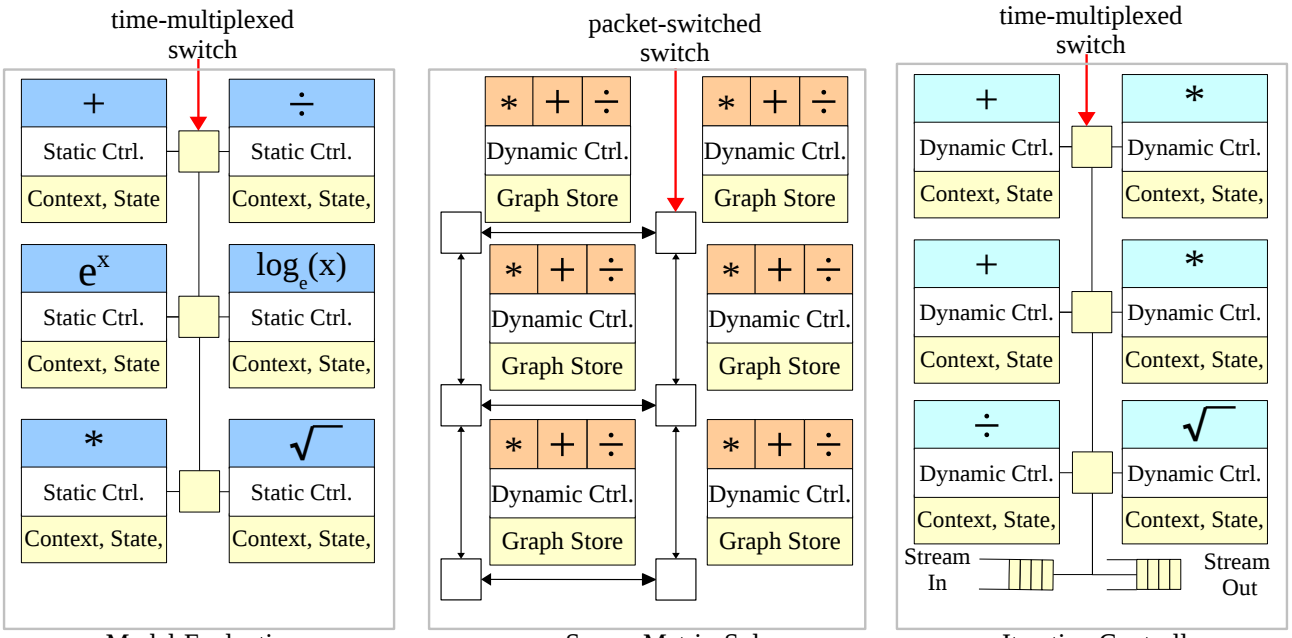


Fig. 8: Internal Organization

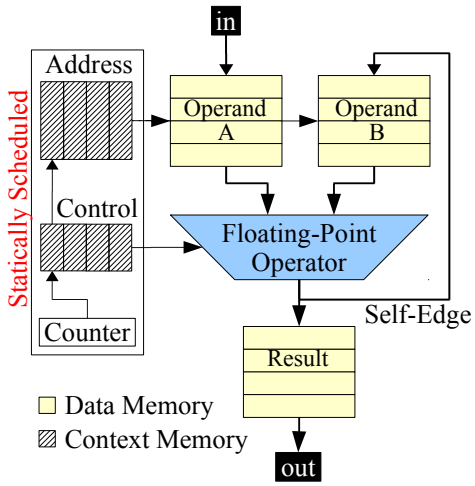


Fig. 9: VLIW Model-Evaluation Architecture

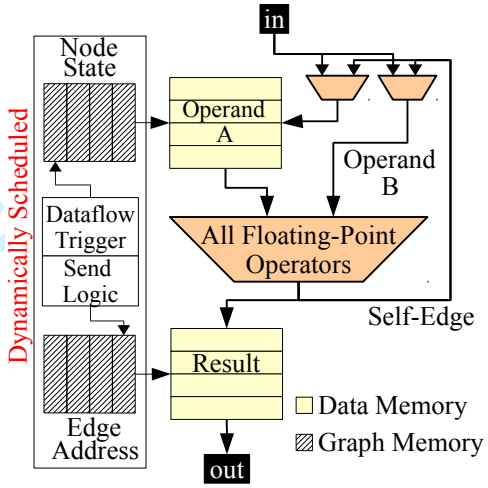


Fig. 10: Dataflow Matrix-Solve Architecture

of cycles (3 GHz). Additionally, we support these spatial operators by coupling them to local, distributed, high-bandwidth memories, as shown in Figure 9, which is not possible with fixed-function CPUs or GPUs. An FPGA can deliver ≈10× higher onchip bandwidth compared to a processor [14]. We statically schedule these resources offline in VLIW [15] fashion and perform loop-unrolling, tiling and software pipelining optimizations to improve performance. Each *tile* in the time-shared 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 as shown in Figure 8. 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 [26] to connect these operators, we also tune the interconnect bandwidth to reflect communication requirements between the operators. Later in Section VI, we observe floating-point utilization as high as 70% for this customized VLIW architecture.

B. *Token-Dataflow Architecture for Matrix-Solve*

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 [39] machine. This organization is capable of ex-

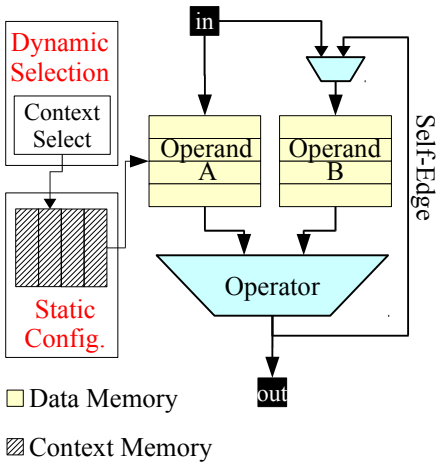


Fig. 11: Streaming VLIW Iteration-Control Architecture

ploiting parallelism across a sparse, irregular graph with fully decentralized, distributed control. The architecture consists of multiple interconnected "*Processing Elements*" (PEs) each holding hundreds to thousands of graph nodes as shown in Figure 8. Each PE, as shown in Figure 10, can fire a node dynamically based on a fine-grained dataflow triggering rule. This allows parallel evaluation of multiple graph nodes which have received their inputs as computation proceeds down the graph. 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. The *Send Logic* in the PE injects messages into the network for nodes that have already been processed. 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. We show the performance possible with this architecture in Section VI.

C. Hybrid VLIW Architecture for Iteration Control

Traditionally, FPGA designs offload the sequential control portion of a spatial design either to host CPUs or embedded Microblaze [49] controllers. Such techniques are unsuitable for stand-alone accelerator systems (no host CPU) or double-precision floating-point computation (poor support on Microblaze). Hence, we consider spatial designs that can implement this computation in the FPGA fabric directly. We observe that the computation is a combination of (1) data-parallel convergence detection and truncation error calculation and (2) sparsely activated, control-intensive SPICE analysis state-machine logic. The underlying FPGA architecture is organized as "Hybrid VLIW tiles" shown in Figure 8 interconnected through streams. Each tile is a collection of floating-point operators (limited to add, multiply, divide and square-root) that are internally connected with a time-multiplexed network. Each operator is managed by a hybrid controller that dynamically selects between statically-scheduled configurations as shown in Figure 11. We allocate the number and type of floating-point units to each SCORE operator as well as pick a suitable unroll factor for best performance. The spatial mapping flow

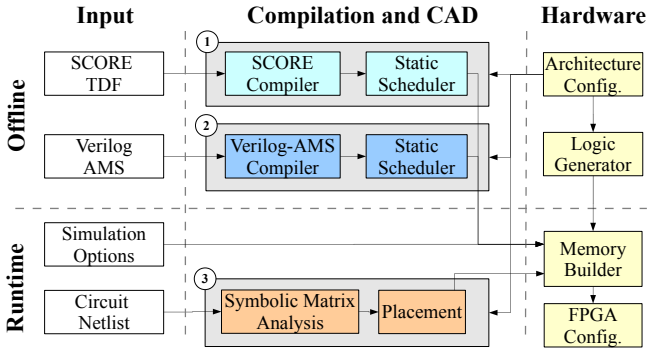


Fig. 12: FPGA SPICE Mapping Toolflow

combines loop-unrolled, software-pipelined scheduling for data-parallel components like truncation error calculation and convergence detection logic along with dataflow scheduling for sparsely activated state-machine logic. The hybrid VLIW architecture is mostly similar to the Model-Evaluation design and we reuse its backend scheduling framework.

IV. FPGA Implementation Methodology

We now explain the methodology and framework for mapping SPICE simulations to FPGAs. In our compilation flow, we first generate a single FPGA bitstream for the SPICE architecture in order to simplify the configuration flow for each circuit to a memory generation step. Thus, we do not need to invoke the FPGA CAD flow for each circuit instance. We show the complete FPGA mapping flow in Figure 12. 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 decomposed into three key stages: Input, Compilation/CAD and Hardware. Additionally, we separate the steps into offline and runtime operations depending on the data binding time. We map this parallelism to the FPGA using customized compute organizations described in Section III.

A. 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. In Table III, we show a distribution of resources among the three SPICE phases for the Xilinx Virtex-6 LX760 device. The FPGA logic configuration includes the VLIW programming for the PEs and switches of the Model-Evaluation and Iteration Control processing elements (output of the "Static Scheduler" block shown in Figure 12).

B. Runtime Memory Configuration

For each circuit, we must program memory resources to store the circuit-specific variables and data-structures relevant for the simulation. This is primarily necessary to support the circuit-specific matrix factorization graph required for the Sparse Matrix Solve phase. 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 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 also need to generate a static dataflow graph for the Matrix-Solve phase at the start of the simulation through symbolic analysis. We distribute the sparse dataflow graph across the Matrix-Solve processing elements (shown by the "Placement" block in Figure 12) and store the graph in offchip DRAM memory when it does not fit onchip capacity. We compute a static ordering of loads from the offchip memory to appropriately stream the graph structure onchip. Once we have the dataflow graphs, we assign nodes to PEs of our parallel architecture using placement for locality with MLPart [6] with fanout decomposition.

C. Hardware Library and Cost Model

We tabulate the resource requirements and performance characteristics of the hardware elements used to compose the system in Table II. We use spatial implementations of individual floating-point *add*, *multiply*, *divide* and *square-root* operators from the Xilinx Floating-Point library in CoreGen [48]. For the *exponential* and *logarithm* operators we use FPLibrary from Arénaire [12] 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 when considering Single-Precision implementation) 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) using Dimension-Ordered Routing. We use a hardware generation framework to automatically generate structural VHDL code for the system based on selected implementation parameters such as system size, network topology, and network bandwidth. The software infrastructure to support time-multiplexed scheduling and packet-switched simulation is extended to provide this hardware generation functionality. We store the static schedules as read-only constants in local onchip distributed memories. We 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.

D. FPGA Cycle Measurement

We express the total number of cycles required by our FPGA implementation as shown in Figure 13. This model assumes we must fit all three phases of the SPICE solver on the FPGA simultaneously while overlapping of a part of the Iteration Control phase with the other two phases of SPICE. Unfortunately, for this implementation we must execute the Model-Evaluation and Matrix-Solve phases one after another (See Section VII for ideas to eliminate this limitation). The state-machine control logic for advancing the simulation cannot be overlapped and must be run in sequence.

We report cycle counts from time-multiplexed schedule (Model-Evaluation and Iteration Controller) and a cycle-accurate simulation (Matrix-Solve). In some cases, when the

Block	Area (Slices)	Latency (clocks)	Speed (MHz)	Ref.
Double-Precision Floating-Point Operators				
Add	334	8	344	[47]
Multiply	131	10	294	[47]
Divide	1606	57	277	[47]
Square Root	822	57	282	[47], [48]
Exponential	1022	30	200	[12]
Logarithm	1561	30	200	[12]
Network Elements				
TM BFT T-Switch	48	2	300	[26], [33]
TM BFT Pi-Switch	64	2	300	[26], [33]
PS Mesh Switch	642	4	312	-
Switch-Switch	32	2	300	-
Processing Elements and Miscellaneous				
VLIW Tile Ctrl.	82	-	300	-
Dataflow PE Ctrl.	297	-	270	-
Microblaze Ctrl.	-	-	-	-
DDR2 Ctrl.	1892	-	250	[34]

TABLE II: Area and Latency model for SPICE Hardware (Virtex-6 LX760), Multiply block also uses 11 DSP48 units

SPICE Phase	Area		Memory	
	Slices	%	BRAMs	%
Model-Evaluation (`bsim4`)	62512	53	448	62
Sparse Matrix-Solve	27090	23	180	25
Iteration Control	17848	15	32	5
Total	107450	91	660	92

TABLE III: FPGA Resource Distribution for complete SPICE Solver (Virtex-6 LX760)

Sparse Matrix-Solve factorization graph will not fit entirely in the FPGA onchip memories, we statically stream portions of the dataflow graph from an offchip DRAM memory. We estimate memory load time for large matrices using streaming loads over the external DDR2-500 MHz memory interface using lowerbound bandwidth calculations. To help compute circuit-specific FPGA cycles required for the Iteration-Control phase of the FPGA SPICE solver, we measure the state activations corresponding to the high-level SCORE operator graph for each benchmark. When we multiply these frequencies with the statically-scheduled cycle count per state, we can compute the total cycles required for the Iteration Control phase.

V. Experimental Framework

We now explain our experimental framework that allows us to compare the performance and energy requirements of the parallel FPGA SPICE mapping with a sequential CPU implementation along with some comparisons with other parallel organizations. For overall speedup calculations, we compare

$$Cycles = max(T_{modeleval} + T_{matsolve}, T_{iterctrl}(dp)) + T_{iterctrl}(stmc)$$

$T_{modeleval}$ = VLIW Model-Evaluation cycles

$T_{matsolve}$ = Dataflow Matrix-Solve cycles

$T_{iterctrl}(dp)$ = Data-Parallel VLIW Iteration-Control cycles

$T_{iterctrl}(stmc)$ = State-Machine Iteration-Control cycles

Fig. 13: Measuring FPGA cycle count

Model	Instruction Distribution (Optimized)						
	Add	Mult.	Divide	Sqrt.	Exp.	Log.	Rest
bjt	22	30	17	0	2	0	8
diode	7	5	4	0	1	2	9
jfet	13	31	2	0	2	0	8
mos1	24	36	7	1	0	0	21
vbic	36	43	18	1	10	4	9
mos3	46	82	20	4	3	0	38
hbt	112	57	51	0	23	18	60
bsim4	222	286	85	16	24	9	137
bsim3	281	629	120	9	8	1	117
mextram	675	1626	397	22	52	37	238
psp	1345	2319	247	30	19	10	263

TABLE IV: Device model instruction counts
Column Rest includes MUX, BOOL and INT operations

the FPGA implementation with Intel Core i7 965 CPU runtimes for the open-source `spice3f5` package coupled with the KLU Solver. When comparing performance for the **Model-Evaluation** phase, we also consider several parallel software implementations running on Intel multi-core, NVIDIA and ATI GPUs, IBM Cell and Sun Niagara2 processors. For the **Sparse Matrix-Solve** phase, we only consider a single-core sequential implementation running on an Intel Core i7 965 as the multi-core implementation of the fine-grained, irregular computation extracted from this direct LU solver does not deliver meaningful performance benefits. Finally, we explore a few implementation alternatives for mapping the sequential **Iteration Control** phase of SPICE.

A. Model-Evaluation Phase

As mentioned earlier, we compile Verilog-AMS descriptions of non-linear device models using our own compilation framework. We generate optimized dataflow graphs that are 3–7× smaller than the raw, unoptimized equations. We use open-source Verilog-AMS non-linear models from Simucad ranging from the small, simple `diode` model to the large, complex `bsim3, psp` models. We tabulate the optimized instruction counts for the different device models in Table IV.

We map the data-parallel model equations to a variety of parallel architectures. To target this diversity of architectures we use a combination of *automated code-generation* and *auto-tuning* to generate optimized implementations across these different systems. Our code-generator writes out multiple configurations of data-parallel code based on architecture-specific templates. Our auto tuner exhaustively explores several implementation parameters for the different architectures as shown in Table VII. Such an exhaustive approach is possible in our case since the Model Evaluation graphs are completely known in advance and the design space is small.

B. Sparse Matrix-Solve Phase

In our Matrix-Solve experimental flow, we use `spice3f5` simulator with its Sparse 1.3 [27] solver to obtain a reference functional baseline for comparison. We the replace Sparse 1.3 with the new KLU solver to measure optimized sequential performance. We use a rich and diverse set of benchmark circuit-simulation matrices detailed later in Appendix A (Table VIII).

Arch.	Compiler	Libraries	Timing
Intel CPUs	`gcc-4.4.3` (-O3)	OpenMP 3.0 [10], GNU libm, Intel MKL 10.1	PAPI 4.0.0 [35], PAPI_flops()
Nvidia GPUs	`nvcc`, CUDA SDK 2.3 [38]	CUDA libraries	cudaEventRecord()
ATI GPUs	`brcc g++-4.1.2`, ATI Stream CAL 1.4beta [1]	ATI Brook libraries	gettimeofday()
IBM Cell	`spu-gcc`, `ppu-gcc`, Cell SDK 3.1 [21]	Simdmath, MASS	gettimeofday()
Sun Niagara2	`cc`, Sun Studio 12.1 [43]	OpenMP [10], libm	PAPI 3.7.0 [35], PAP_flops()
Xilinx FPGA	Synplify Pro 9.6.1, Xilinx ISE 10.1	Xilinx Core-gen [47], Arénaire [12]	-

TABLE V: Parallel Software Environments

Family	Chip	Peak GFLOPS		GFLOPs per Watt
		Double	Single	
65nm Architectures				
Intel Xeon	5160	12	24	0.3
Xilinx Virtex-5	LX330T	11	33	1.1
IBM Cell	PS3	10	204	1.5
Sun Niagara	Ultrasparc T2	8	9.6	0.1
NVIDIA GPU	9600GT	-	312	3.2
AMD FireGL GPU	5700	120	144	-
45nm Architectures				
Intel Core i7	965	25	51	0.4
Xilinx Virtex-6	LX760	26	75	2.5
NVIDIA GPU	GTX285	132	1062	5.2
AMD Firestream GPU	9270	240	1200	5.4

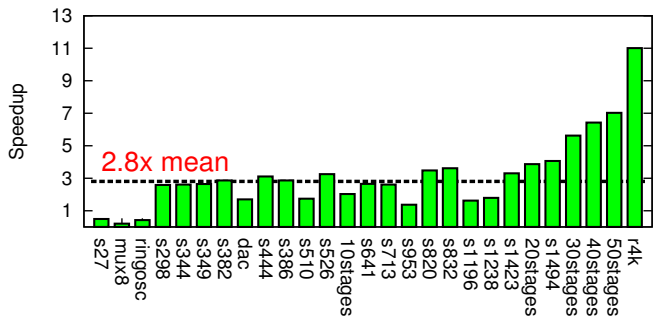
TABLE VI: Peak Floating-Point Throughput
(GFLOPs per Watt is for Single-Precision)

C. Iteration Control

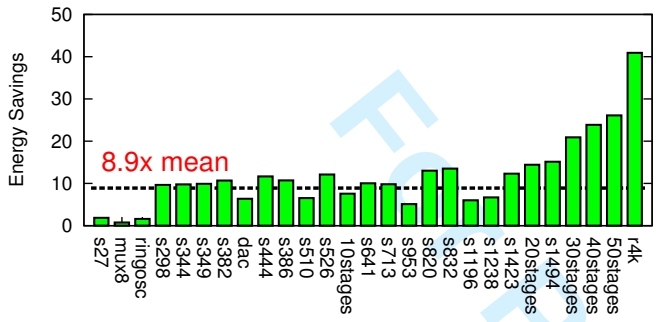
We generate multi-threaded C++ code from the SCORE compiler [7], [13] to obtain a software implementation for functional verification with `spice3f5`. We use PAPI to measure the CPU runtime of the Iteration Control phase in `spice3f5`. We develop a SCORE runtime customized for the Microblaze soft processor to support Iteration Control computation on the Microblaze. This is done through automated code-generation in a flavor of C suitable for use with a light-weight embedded operating system running on the Microblaze

Architecture	Parameter	Range	Increment
Intel	Loop-Unroll Factor	1–5	+1
	MKL Vector	true/false	
NVIDIA GPU	Loop-Unroll Factor	1–2	+1
	Threads per block	8–512	×2
	Registers/Thread	16–128	×2
ATI GPU	Loop-Unroll Factor	1–2	+1
IBM Cell	Loop-Unroll Factor	1–3	+1
	MASS Vector	true/false	
Sun Niagara2	Loop-Unroll Factor	1–3	+1
	Number of Threads	1–64	×2
FPGA	Loop-Unroll Factor	1–15	+1
	Operators per PE	8–64	×2
	BFT Rent Parameter	0.0–1.0	+0.1

TABLE VII: Auto-Tuning Parameters



(a) Total Per-Chip Speedup



(b) Energy Ratio

Fig. 14: Comparing Xilinx Virtex-6 LX760 FPGA (40nm) and Intel Core i7 965 (45nm) Implementations

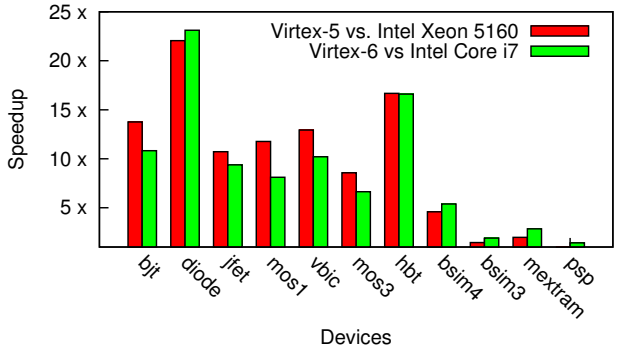
(Xilkernel [50]). We measure the number of Microblaze clock cycles to implement each state of every SCORE operator using a hardware counter. The Xilinx Microblaze controller along with supporting logic is designed to operate at 100 MHz by Xilinx Core Generator [47].

VI. Evaluation

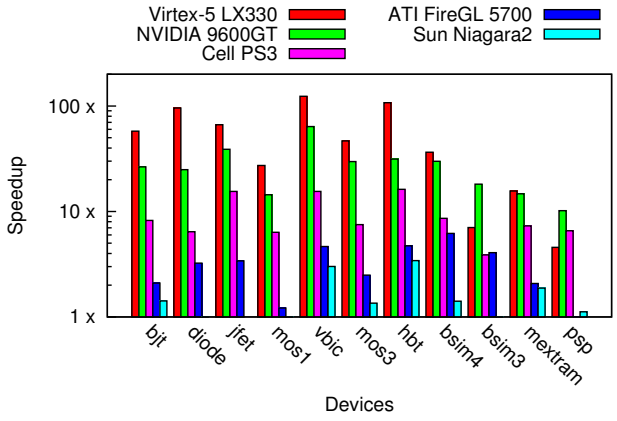
We now report the achieved performance and energy requirements of our parallel SPICE implementation. We show total speedups for the SPICE solver when comparing an Intel Core i7 965 with a Virtex-6 LX760 FPGA in Figure 14a. We observe a mean speedup of 2.8× across our benchmark set with a peak speedup of 11× for the largest benchmark. We also show the ratio of energy consumption between the two architectures in Figure 14b. 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

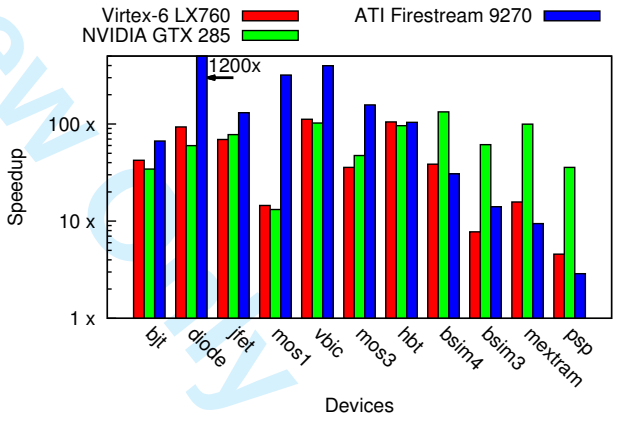
In Figure 15a, we compare the performance achieved for a double-precision implementation of Model-Evaluation on 45nm parallel architectures which include a quad-core Intel Core i7 965 (loop-unrolled and multi-threaded) and a Xilinx Virtex-6 LX760T FPGA (loop-unrolled, tiled and statically



(a) Double-Precision (CPU vs. FPGA) at 65nm



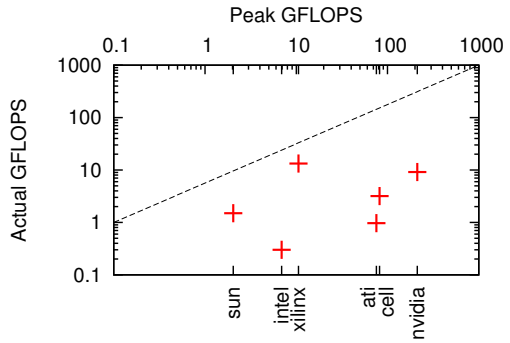
(b) Single-Precision (vs. Xeon 5160) at 65nm



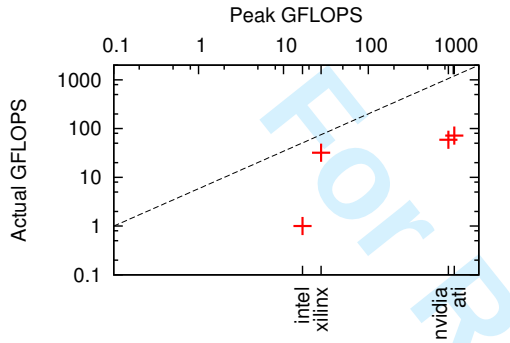
(c) Single-Precision (CPU vs. FPGA) at 45nm

Fig. 15: Speedups for Model-Evaluation

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 15b and observe much higher speedups of 4.5–123× for a Virtex-5 LX330, 10–64× for an NVIDIA



(a) 65nm Architectures



(b) 45nm Architectures

Fig. 16: Actual vs. Peak Single-Precision Throughputs

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). In Figure 16a we plot the mean floating-point utilization across all non-linear devices when considering parallel architectures at 65nm. At 65nm architectures, we observe that the FPGA is able to achieve the highest actual floating-point throughput (≈40% utilization of peak) compared to all other architectures despite not having the highest peak floating-point throughput (NVIDIA 9600 GT GPU with ≈3% utilization of peak). Similarly for 45nm architectures, compared in Figure 16b, we observe that the FPGA delivers a large fraction of its peak throughput (mean ≈50% utilization of peak) but delivers fewer total FLOPS as compared to the GPUs (mean 10% utilization of peak) which require ≈5× higher peak throughput to outperform the FPGA.

B. Matrix-Solve

When we integrate the KLU matrix-solver in `spice3f5` instead of the default Sparse 1.3 solver, we are able to speedup the software implementation by ≈35% across our benchmark circuits as shown in Figure 17. We achieve higher improvements for larger benchmarks since the symbolic analysis overheads can be amortized easily for large matrices. We use this as our software baseline for comparing with the

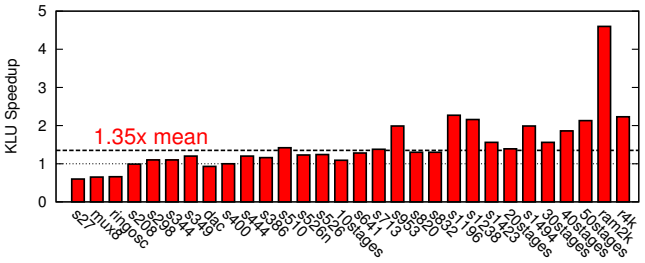


Fig. 17: Software Speedup for `spice3f5`: Sparse 1.3 vs. KLU Matrix Solver

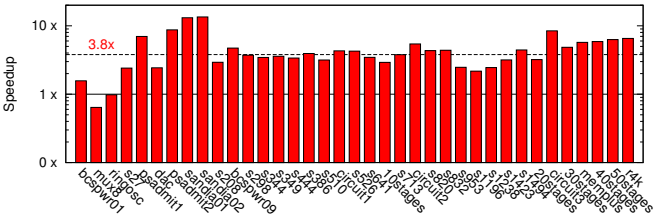


Fig. 18: Speedups for Double-Precision Matrix-Solve (vs. Core i7 965)

FPGA implementation. In Figure 18, 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 allows efficient processing of the fine-grained factorization operations which can be synchronized at the granularity of individual floating-point operations. To better understand the speedups we plot the distribution of parallel runtime across the different steps of the matrix-solve implementation in Figure 19. We observe that performance is dominated by the cost of loading the large dataflow graph from offchip memory. We may be able to reduce this overhead with better DRAM memory interfaces and higher onchip capacity. In Figure 20, we look at the scaling trends of the dataflow architecture as we increase the number of PEs in the system. We see varied scaling trends for our benchmark set with some matrices scaling well (medium sized matrices or those with low circuit fanout *i.e.* small critical paths) whereas others scale very poorly (small sized matrices or those with high circuit fanout *i.e.* long critical paths). To improve scaling, we will need to consider decomposing the performance-limiting long critical paths through coarse-grained matrix decomposition approaches [40] or through associative reformulation [22].

C. Iteration Control

We now consider the impact of parallelizing the Iteration Control phase on the overall speedups of the FPGA system. In Figure 21, we now show the overall SPICE speedups under three implementation scenarios (1) offload to sequential host CPU over PCI (2) offload to Microblaze soft-processor (3) spatial implementation over hybrid VLIW design. We observe that the spatial implementation can deliver modest improvements of 2.6×(mean) over the sequential CPU implementation. We can show this benefit by localizing all communication within the FPGA system and exploiting data parallelism in the

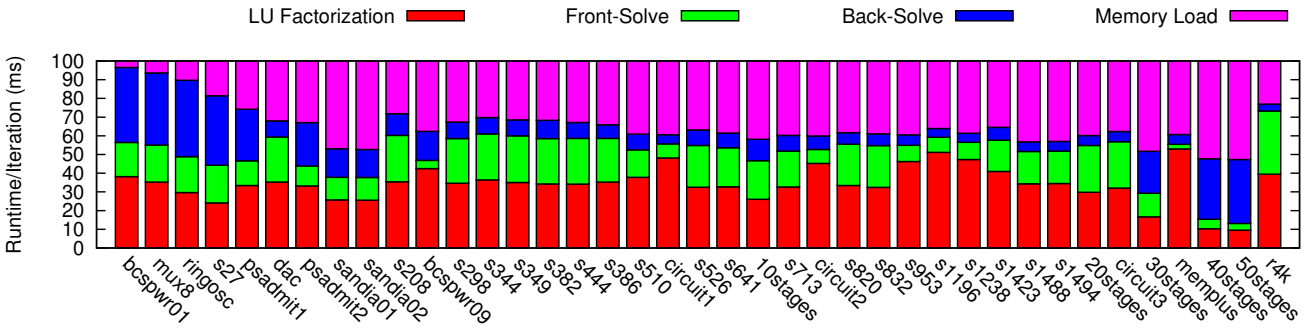


Fig. 19: Parallel Runtime Distribution for Matrix-Solve

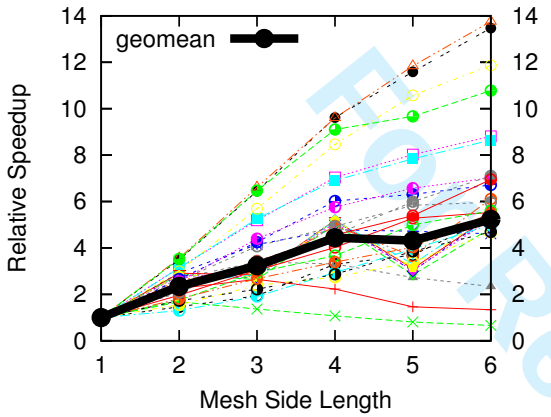


Fig. 20: Performance Scaling Trends for Matrix-Solve (each colored line represents a sparse matrix)

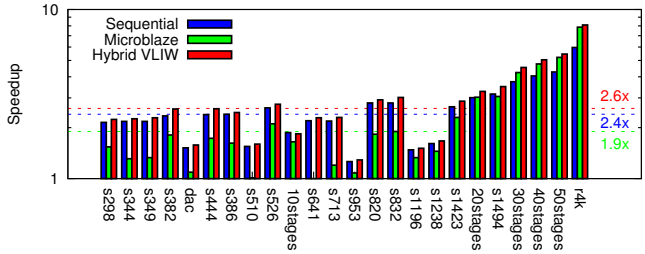


Fig. 21: Speedup for the Overall SPICE Simulator for different Iteration Control Implementations

convergence detection and truncation error calculation steps. However the amount of overall improvement is not very high since the Iteration Control phase accounts for merely ≈7% of sequential SPICE runtime. Other FPGA studies [41] prefer to implement such sequential fraction of the application on embedded soft-processors like the Xilinx Microblaze. We see the limits of using the Microblaze (1.9× mean speedup) to implement this sequential computation as it can be worse than even offloading the processing to the host CPU over PCI (2.4× mean speedup). The Microblaze soft-processor offers poor double-precision floating-point support and schedules computation sequentially over the ALU thereby limiting potential performance. In contrast, the spatial VLIW design exploits the available data parallelism and implements the state-machine processing with lightweight decision-making hardware thus delivering better performance at modest cost.

VII. Future Work

We now identify additional opportunities for improving the performance of the parallel FPGA design.

1) 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 [40] approaches for exposing more coarse-grained parallelism and associative reformulation [22] for improved scalability. With domain-decomposition, we can break up the large matrix into multiple submatrices that can be solved independently and possibly even distributed across multiple FPGAs.
2) Double-precision floating-point operators consume a large amount of area on FPGAs. Custom floating-point or fixed-point operators that operate at just enough precision might provide an opportunity for improving the compute density on FPGAs. We can redesign the Model-Evaluation datapaths with lower precision while satisfying accuracy requirements by adapting existing techniques [4], [32] to obtain additional acceleration at lower cost.
3) Sparse matrix solve operations on large matrices can generate large dataflow graphs with millions of nodes and edges. These large graphs can take be challenging to place and distribute across parallel compute elements if we want to maximize locality. We can accelerate the placement algorithm itself using parallelism to minimize the one-time setup cost of the parallel simulation.

Our current design exposes most, but not all, of the parallelism available in the SPICE simulator. We must investigate the following key opportunities for additional improvement in parallel SPICE performance:

1) We can overlap the Model-Evaluation phase with the Sparse Matrix-Solve phase of SPICE. Our streaming high-level capture in SCORE offers the ability to integrate a scheduler that can facilitate this overlap. The scheduler needs to statically compute a suitable ordering of the device evaluation in Model-Evaluation to match the dataflow ordering in the Sparse Matrix-Solve computation.
2) Additionally, we can improve the performance of the Model-Evaluation phase with extra loop-unrolling and the use of offchip memory capacity. We need to develop

an extension to our VLIW architecture to migrate data offchip when necessary.

3) Apart from these approaches, it may be useful to consider completely different algorithms (iterative matrix-free fixed-point simulation [9] or constant-Jacobian [51]; for SPICE simulations that completely eliminate the need for performing per-iteration matrix factorization.

VIII. 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 note that GPU implementation of Model-Evaluation manages to outperform the FPGA mapping by 2–3× in a few cases but is incapable of accelerating the irregular Matrix-Solve phase thereby limiting total system speedup. We observe that fine-grained, parallel dataflow evaluation of large sparse matrix factorization graphs does not deliver a large speedup suggesting further investigation into coarse-grained matrix factorization techniques. We were able to compose the overall heterogeneous design that mixes VLIW, Dataflow and Streaming organizations into a unified implementation with the assistance of suitable SCORE composition framework. We believe the ideas explored in this research are relevant across an important class of problems where computation is characterized by static, data-parallel processing and 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 of general-purpose computation.

References

[1] AMD. Programming Guide ATI Stream Computing, 2010.

[2] A. M. Bayoumi and Y. Y. Hanafy. Massive parallelization of SPICE device model evaluation on GPU-based SIMD architectures. In *Proceedings of the 1st international forum on Next-generation multicore/-manycore technologies*, pages 1–5, Cairo, Egypt, 2008. ACM.

[3] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and JJ. The Matrix Market: A web resource for test matrix collections. *Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[4] D. Boland and G. A. Constantinides. Automated Precision Analysis: A Polynomial Algebraic Approach. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 157–164, May 2010.

[5] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. *IEEE International Symposium on Circuits and Systems*, 3(May 1989):1929–1934, 1989.

[6] A. Caldwell, A. Kahng, and I. Markov. Improved algorithms for hypergraph bipartitioning. *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 661–666, 2000.

[7] E. Caspi. *Design Automation for Streaming Systems*. Phd, University of California, Berkeley, 2005.

[8] Chung-Wen Ho, A. Ruehli, and P. Brennan. The modified nodal approach to network analysis. *IEEE Transactions on Circuits and Systems*, 22(6):504–509, 1975.

[9] B. Conn. XPICE Circuit Simulation Software. *(unpublished)*, 2008.

[10] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[11] T. Davis. The University of Florida Sparse Matrix Collection. *(unpublished) ACM Transactions on Mathematical Software*, 2007.

[12] 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*, page 260, 2008.

[13] A. Dehon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, 30(6):334–354, Sept. 2006.

[14] M. DeLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, 2006.

[15] J. A. Fisher. The VLIW Machine: A Multiprocessor for Compiling Scientific Code. *IEEE Computer*, 17(7):45–53, 1984.

[16] J. Gilbert and T. Peierls. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 1988.

[17] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry. Fast circuit simulation on graphics processing units. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 403–408, 2009.

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

[19] J. Huang and O. Wing. Optimal parallel triangulation of a sparse matrix. *IEEE Transactions on Circuits and Systems*, 26(9):726–732, 1979.

[20] S. Hutchinson, E. Keiter, R. Hoekstra, H. Watts, A. Waters, R. SCHELLS, and S. WIX. The Xyce Parallel Electronic Simulator - An Overview. *IEEE International Symposium on Circuits and Systems*, 2000.

[21] IBM. Software Development Kit for Multicore Acceleration Version 3.1: Programmer's Guide, 2008.

[22] N. Kapre and A. DeHon. Optimistic parallelization of floating-point accumulation. *IEEE Symposium on Computer Arithmetic*, pages 205–216, 2007.

[23] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. In *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 37–44. IEEE, 2009.

[24] N. Kapre and A. DeHon. Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs. In *International Conference on Field-Programmable Technology*, pages 190–198, 2009.

[25] N. Kapre and A. DeHon. 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*, pages 65–72, 2009.

[26] 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*, pages 205–216, 2006.

[27] K. S. Kundert and A. Sangiovanni-Vincentelli. Sparse User's Guide: A Sparse Linear Equation Solver, 1988.

[28] P. Lee, S. Ito, T. Hashimoto, J. Sato, T. Touma, and G. Yokomizo. A parallel and accelerated circuit simulator with precise accuracy. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, pages 213–218, 2002.

[29] 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*, pages 7–8, 2003.

[30] D. Lewis. A programmable hardware accelerator for compiled electrical simulation. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 172–177, 1988.

[31] D. Lewis. A compiled-code hardware accelerator for circuit simulation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 555 – 565, 1992.

[32] M. Linderman, M. Ho, D. Dill, T. Meng, and G. Nolan. Towards program optimization through automated analysis of numerical precision. In *Proceedings of the IEEE/ ACM international symposium on Code Generation and Optimization*, pages 230–237, New York, New York, USA, 2010. ACM.

[33] N. Mehta. *Time-Multiplexed FPGA Overlay Networks On Chip*. Master's thesis, California Institute of Technology, 2006.

[34] Microsoft Research. DDR2 DRAM Controller for BEE3, 2008.

Bmarks.	Matrix Size	Sparsity (%)	Mult. Sub.	Divide	Total Ops.	Fanout (DFG)	Fanin (NZ)	Latency (cycles)
spice3f5, Simucad [42]								
mux8	42	15.0793	488	138	626	8	20	1.9K
ringosc	104	6.4903	1.3K	351	1.6K	4	92	3.7K
dac	654	1.5849	20.2K	3.3K	23.6K	10	1136	7.7K
ram2k	4875	0.3107	1.0M	38.5K	1.0M	137	9618	62.2K
spice3f5, Clocktrees [44]								
r4k1	39948	0.0131	390.3K	125.1K	515.5K	6	29910	127.8K
spice3f5, Wave-pipelined Interconnect [45]								
10stages	3920	0.1753	57.8K	14.8K	72.7K	8	2384	18.6K
20stages	11225	0.0618	174.8K	44.4K	219.2K	9	9442	46.2K
30stages	16815	0.0410	244.3K	61.7K	306.0K	11	4688	88.6K
40stages	22405	0.0307	316.1K	79.5K	395.7K	9	600	134.2K
50stages	27995	0.0245	394.7K	99.2K	493.9K	10	484	169.7K
spice3f5, ISCAS89 Netlists [5]								
s27	189	3.4405	2.1K	573	2.7K	6	50	3.6K
s208	1296	0.5277	19.7K	4.9K	24.6K	11	1414	11.3K
s298	1801	0.4026	32.6K	7.3K	40.0K	13	1938	13.1K
s344	1992	0.3522	32.3K	7.8K	40.1K	12	2178	14.7K
s349	2017	0.3512	33.9K	8.0K	41.9K	14	2218	14.7K
s382	2219	0.3184	37.2K	8.7K	45.9K	16	2358	16.1K
s444	2409	0.2952	41.4K	9.6K	51.1K	16	2526	16.6K
s386	2487	0.2927	46.4K	10.0K	56.5K	20	2626	15.7K
s510	2621	0.3124	105.3K	11.9K	117.2K	54	2722	21.4K
s526n	3154	0.2362	66.1K	13.0K	79.2K	25	3280	21.9K
s526	3159	0.2376	68.1K	13.3K	81.4K	26	3294	20.7K
s641	3740	0.2000	100.2K	15.6K	115.9K	39	4066	26.5K
s713	4040	0.1890	126.4K	17.1K	143.5K	47	4380	30.3K
s820	4625	0.1655	103.2K	19.6K	122.8K	29	4766	26.1K
s832	4715	0.1629	105.7K	20.0K	125.8K	29	4846	26.6K
s953	4872	0.1876	353.9K	24.3K	378.2K	85	5212	37.9K
s1196	6604	0.1399	475.3K	33.0K	508.3K	83	7146	46.4K
s1238	6899	0.1325	457.9K	34.2K	492.2K	78	7454	46.6K
s1423	9304	0.0820	296.0K	39.4K	335.4K	64	10384	64.5K
s1488	9849	0.0827	354.7K	44.7K	399.4K	49	10606	54.8K
s1494	9919	0.0817	352.4K	44.8K	397.3K	50	10646	54.6K

TABLE VIII: Circuit Simulation Benchmark Matrices

[35] P. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. *Proceedings of the Department of Defense High Performance Computing Modernization Program Users Group Conference*, pages 7–10, 1999.
[36] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California Berkeley, 1975.
[37] E. Natarajan. *KLU A high performance sparse linear solver for circuit simulation problems*. Master's thesis, University of Florida Gainesville, 2005.
[38] NVIDIA. NVIDIA CUDA Programming Guide, 2009.
[39] G. Papadopoulos and D. Culler. Monsoon: an explicit token-store architecture. *Proceedings of the Annual International Symposium on Computer Architecture*, 18(3a):82–91, 1990.
[40] H. Peng and C. K. Cheng. Parallel transistor level circuit simulation using domain decomposition methods. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 397–402. IEEE Press Piscataway, NJ, USA, 2009.
[41] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. In *Proceedings of the International Symposium on Computer Architecture*, volume 37, page 395. ACM, June 2009.
[42] Simucad/Silvaco. BSIM3, BSIM4 and PSP benchmarks from Simucad, 2007.
[43] Sun/Oracle. Sun Studio Compiler 12.1 (Renamed Oracle Solaris Studio), 2009.
[44] C. Sze, P. Restle, G. Nam, and C. Alpert. ISPD2009 clock network synthesis contest. In *Proceedings of the 2009 International Symposium on Physical design*, page 149, New York, New York, USA, 2009. ACM Press.
[45] P. Teehan, G. Lemieux, and M. Greenstreet. Towards reliable 5Gbps wave-pipelined and 3Gbps surfing interconnect in 65nm FPGAs. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 43–52. ACM, 2009.
[46] Q. Wang and D. M. Lewis. Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation. In *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 145 – 154, Napa Valley, 1997.
[47] Xilinx. Xilinx CoreGen Reference Guide, 2000.
[48] Xilinx. Floating-Point Operator v5.0, 2009.
[49] Xilinx. MicroBlaze Processor Reference Guide, 2010.
[50] Xilinx. OS and Libraries Document Collection, 2010.
[51] X. Ye, W. Dong, P. Li, and S. Nassif. MAPS: Multi-Algorithm Parallel circuit Simulation. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 73–78, Nov. 2008.

APPENDIX A
CIRCUIT BENCHMARKS

We show the matrix characteristics of the circuit benchmarks used in our experiments in Table VIII. We use circuit-simulation matrices from the University of Florida Sparse-Matrix Collection [11] as well as Power-system matrices from the Harwell-Boeing Matrix-Market Suite [3]. For matrices generated from `spicef5`, we use RAM netlist benchmarks provided by Simucad [42], clocktrees from University of Michigan [44], wave-pipelined circuits obtained from UBC [45] and the ISCAS 1989 benchmark set from IBM [5].