Performance Comparison of Single-Precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core Processors

Nachiket Kapre

Computer Science
California Institute of Technology
Pasadena CA 91125
email: nachiket@caltech.edu

André DeHon

Electrical and Systems Engineering
University of Pennsylvania
Philadelphia PA 19104
email: andre@seas.upenn.edu

ABSTRACT

Automated code generation and performance tuning techniques for concurrent architectures such as GPUs, Cell and FPGAs can provide integer factor speedups over multi-core processor organizations for data-parallel, floating-point computation in SPICE Model-Evaluation. Our Verilog AMS compiler produces code for parallel evaluation of non-linear circuit models suitable for use in SPICE simulations where the same model is evaluated several times for all the devices in the circuit. Our compiler uses architecture specific parallelization strategies (OpenMP for multi-core, PThreads for Cell, CUDA for GPU, statically scheduled VLIW for FPGA) when producing code for these different architectures. We automatically explore different implementation configurations (e.g. unroll factor, vector length) using our performance-tuner to identify the best possible configuration for each architecture. We demonstrate speedups of 3–182× for a Xilinx Virtex5 LX 330T, 1.3–33× for an IBM Cell, and 3–131× for an NVIDIA 9600 GT GPU over a 3 GHz Intel Xeon 5160 implementation for a variety of single-precision device models.

1. INTRODUCTION

SPICE (Simulation Program with Integrated Circuit Emphasis) [1] is an analog circuit-simulator that is used to model the behavior of electronic circuits. Accurate SPICE simulations of large sub-micron circuits can often take days or weeks (see Table 1) of runtime on modern processors. Various attempts at reducing SPICE runtimes by parallelizing SPICE have met with mixed success (See Section 2.5 of [9]). SPICE does not parallelize easily on conventional processors due to the irregular structure of the computation, limited peak floating-point capacities and scarce memory bandwidth.

Newer parallel architectures such as GPUs, Cell and FPGAs provide opportunities for greater parallelism in accelerating SPICE. Modern FPGAs can now support large floating-point computations on a single-chip and can be customized to implement irregular floating-point datapaths. GPUs support massively-parallel processing of concurrent threads over hundreds of single-precision floating-point graphics pipelines (newer GPUs support double-precision). The IBM Cell processor is also capable of running several threads in parallel over eight vector floating-point processing elements (SPUs).

However, programming these architectures continues to be a challenge. In order to properly exploit available parallelism, developers are forced to use a laborious, low-level programming approach to manually tune the implementation for best performance. This makes it hard to port the design to a different parallel organization or scale the application to use increasing parallel capacity provided by Moore's law. In this paper, we use automated code-generation and performance-tuning of SPICE Model-Evaluation computation to demonstrate productive application development on diverse parallel architectures that can take advantage of increasing on-chip parallelism. This avoids an architecture-specific manual parallelization effort and eliminates the need for programmer intervention in tuning an implementation for best performance.

We previously reported double-precision floating-point FPGA implementations for SPICE Model-Evaluation and compared them to a 65nm processor mapping [9]. Here, we replace the double-precision floating-point operators with single-precision operators and compare such an FPGA mapping with the 65nm processor as well as GPUs and Cell implementations. We also consider a 45nm comparison between the processor and FPGA. In this paper, we focus only on parallelizing the Model-Evaluation phase of SPICE; in future work we intend to parallelize the Matrix-Solve phase and integrate a complete SPICE simulator.

The key contributions of this paper include:

- Development of a code-generation and performance-tuning framework for SPICE Model-Evaluation to produce optimized parallel code for the GPU, Cell, FPGA and multi-core architectures.

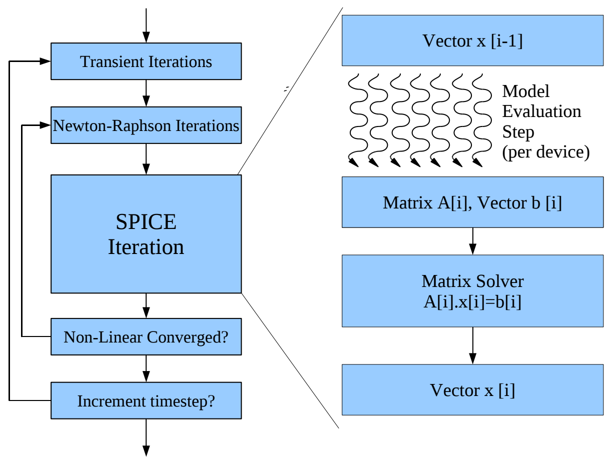

Transient Iterations
Newton-Raphson Iterations
SPICE Iteration
Non-Linear Converged?
Increment timestep?
Vector x [i-1]
Model Evaluation Step (per device)
Matrix A[i], Vector b [i]
Matrix Solver A[i].x[i]=b[i]
Vector x [i]

Fig. 1. Flowchart of a SPICE Simulator

- Quantification of the impact on accuracy and iterations of using single-precision Model-Evaluation with double-precision Matrix-Solve in `spice3f5` over a set of benchmark circuits using the bsim3 device model.
- Quantitative empirical comparison of SPICE model evaluation on the Intel Xeon 5160 processor, NVIDIA 9600 GT GPU, IBM Cell (1st generation) and Virtex-5 LX330T FPGA (65nm) as well as on the Intel Core i7 965 processor and Virtex-6 LX 760 FPGA (45nm).

2. BACKGROUND

2.1. Review of SPICE

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 while obeying the conservation constraints (*i.e.* Kirchoff's conservation laws—KCL and KVL) at the different nodes and branches of the circuit. SPICE solves the non-linear circuit equations by alternately computing small-signal linear operating-point approximations for the non-linear elements and solving the resulting system of linear equations until it reaches a fixed point. 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. It then solves for $\vec{x}$ using a sparse-direct linear matrix solver in the **Matrix-Solve** phase. We illustrate the steps in the SPICE algorithm in Figure 1. The inner loop iteration supports the operating-point calculation for the non-linear circuit elements, while the outer loop models the dynamics of time-varying devices such as capacitors.

Table 1. `spice3f5` Runtimes (Intel Core i7 965)

bsim3 Circuits	Model Eval.	Matrix Solve	Percent Model-Eval.
runtime (seconds)			
`ram2k`	30	12	71%
`ram8k`	130	73	64%
`ram64k`	1105	908	54%
floating-point ops. (millions)			
`ram2k`	25.29	1.89	93%
`ram8k`	101.11	7.56	93%
`ram64k`	809.57	60.50	93%

2.2. 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 the linear elements (e.g. 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 which requires repeated evaluation of the model equations multiple times per time-step as shown by the innermost loop (labeled Newton-Raphson iterations) in Figure 1. For time-varying components, the simulator must recalculate their contributions at each timestep based on voltages at several previous timesteps. This also requires repeated re-evaluations of the device-model as shown by the outer loop (labeled Transient iterations).

For circuits dominated by non-linear transistor devices, the simulator can spend more than half its time evaluating the device models (see "runtime" in Table 1). Moreover, the Model-Evaluation phase can be responsible for almost 90% of total floating-point operations in the simulation (see "floating-point ops." in Table 1). Additionally, as transistor devices shrink in feature-size, the complexity of the device models required to simulate them correctly grows over time. Newer device models often have complexity 3–5× that of the classic `bsim3` model [2] (as shown in Table 2, the `psp` model [3] is 5× more complex than the `bsim3` model).

2.3. Parallelism Potential

The SPICE Model-Evaluation phase has high **data parallelism** consisting of thousands of independent device evaluations each requiring hundreds of floating-point operations. There is high **pipeline parallelism** within each device eval-

Table 2. Verilog-AMS Compiler Output

Models	Instruction Distribution					
	Add	Mult.	Div.	Sqrt.	Exp.	Log
`bjt`	22	30	17	0	2	0
`diode`	7	5	4	0	1	2
`hbt`	112	57	51	0	23	18
`jfet`	13	31	2	0	2	0
`mos1`	24	36	7	1	0	0
`vbic`	36	43	18	1	10	4
`mos3`	46	82	20	4	3	0
`mextram`	675	1626	397	22	52	37
`bsim3 v3.2`	283	634	122	9	8	1
`bsim4 v3.0`	222	286	85	16	24	9
`psp`	1345	2319	247	30	19	10

uation 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. Within a simulation, there may be very few unique device models active (e.g. typically all transistors in a circuit will use the same `bsim3` model). 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 which vary from device to device (e.g. W, L of device).

2.4. Verilog-AMS

Modern SPICE simulators accept a wide variety of device models that cater to different designer requirements. These device models are released as simulator independent Verilog-AMS descriptions [4, 5]. We use open-source Verilog-AMS descriptions of a variety of devices available from Silvaco [6]. We developed a Verilog-AMS compiler that supports a subset of the Verilog-AMS language for device models [4]. We compile the device model equations into a flexible intermediate representation that allows us to perform analysis, optimization and code-generation for different architectures easily. Our compiler generates a generic feed-forward dataflow graph of the computation that is processed by architecture-specific backend tools. The instruction counts and distribution for different device models is shown in Table 2.

2.5. Related Work

We compare several parallel SPICE efforts in [9]. More recent work has focussed on parallelizing Model-Evaluation on GPUs. The use of GPUs for accelerating SPICE Model-Evaluation of the `bsim3` model was first explored in [7] (double-precision) and subsequently in [8] (single-precision). [7] showed speedups of 10×–50× over a quad-core AMD CPU when using an AMD Firestream 9170 GPU (512 processors). [8] showed speedups of 32×–40× over a quad-core Intel CPU when using an NVIDIA 8800 GTX GPU (128 processors). Our approach shows a speedup of 30× for `bsim3` model over a dual-core Intel Xeon processor when using an NVIDIA 9600 GT GPU with only 64 processors. In this paper, We benchmark a wide variety of device models in addition to the `bsim3` model (shown in Table 2) and also evaluate other parallel architectures.

3. IMPLEMENTATION OF COMPUTATION

There are several competitive architectural choices for accelerating floating-point applications (See Table 5). These architectures exploit different forms of parallelism, support various programming models and require differing amount of programming effort. Their raw floating-point peak performance varies across two orders of magnitude and they have different underlying compute organizations. This suggests these architectures may deliver different performance across the varying needs of diverse device models. This paper focuses on comparing the performance of the Intel Xeon 5160, NVIDIA GPU 9600 GT, IBM Cell (1st generation) with the Xilinx Virtex 5 FPGAs (65nm technology) and the Intel Core i7 965, with the Xilinx Virtex 6 FPGAs (45nm technology or smaller).

3.1. Code Generation

As noted in Section 2.3, SPICE Model-Evaluation is a data-parallel computation. We exploit this data-parallelism when generating parallel code for the different architectures. Each architecture provides a different parallel construct to expose this parallelism to the compiler as shown in Table 3. Our code-generator produces custom code using these constructs for respective architectures without any programmer assistance. We generate code with simple OpenMP pragma `omp parallel for` shown in Table 3(a) to distribute Model-Evaluation across 8 threads on the Intel Core i7 processor. We express each individual device evaluation as one scalar thread of work and let the GPU thread scheduler distribute these threads across the GPU using the CUDA API constructs shown in Table 3(b). We distribute processing across the six user-programmable PS3 Cell SPUs by using PThreads [10] shown in Table 3(c) to create and manage parallel threads. We generate custom VLIW instructions for our FPGA architecture described in [9].

```
# pragma omp parallel for
for(i=0;i<DEVICES;i++)
    kernel();

void kernel() {
    // device evaluation code
}
```

(a) OpenMP

```
dim3 grid(32,1,1);
dim3 threads(DEVICES/32,1,1);

kernel<<<grid, threads>>>();

__global__ void kernel() {
    // device evaluation code
}
```

(b) CUDA SDK

```
for(i=0;i<THREADS;i++)
    pthread_create(thread[i]);
for(i=0;i<THREADS;i++)
    pthread_join(thread[i]);

int main(arguments_to_thread) {
    for(i=0;i<DEVICES_PER_THREAD;i++) {
        // device evaluation code
    }
}
```

(c) Cell SDK

Table 3. A comparison of data-parallel constructs across three architectures

3.2. Optimizations

In addition to data-parallelism, we can exploit other characteristics of SPICE Model-Evaluation graphs to get better performance. For example, we notice that SPICE model-evaluation graphs are characterized by long critical paths with little work off this path which may significantly under-utilize processor capacity. To increase utilization and improve performance, we can perform loop unrolling or vectorize the device loop so that multiple devices are scheduled together. For example, the `bsim3` device model requires 365 cycles per evaluation with no unrolling, which reduces to 202 cycles per evaluation when unrolled twice (see Figure 3b). Our FPGA implementation described in [9] exploits the static nature of the graphs to perform efficient offline scheduling of computation. We can even exploit pipeline parallelism within the graphs to perform software-pipelining across independent device evaluations. This allows us to retime the graphs to overlap computation and communication and schedule them independently to achieve better performance.

3.3. Auto Tuning

The process of manually customizing and tuning an application mapping to a given architecture is a time-consuming process that produces non-portable implementations. Since our experiment targets multiple architectures with different organizations and optimization parameters, we choose an automated approach that empirically tunes the mapping for each architecture. The approach is similar to the auto tuner used in the ATLAS framework [11] for optimizing dense linear algebra kernels. Our auto tuner can explore several implementation parameters for the different architectures as shown in Table 4. For example, our GPU implementation organizes device-evaluations into threads (mapped to an ALU) which must be grouped into blocks (mapped to multiprocessor: collection of ALUs) and grids (mapped to GPU: collection of multi-processors) for a CUDA implementation. Our auto-tuner picks the number of threads in each block (grid configuration) to maximize GPU usage and deliver best performance. Similarly, our FPGA architecture includes several different parameters that can be chosen to make best use of available resources for a given problem. Currently, the entire space of implementation parameters we evaluate during the tuning phase across all architectures is small. Hence, a simple exhaustive sweep of this space is possible and runs in a reasonable amount of time.

Table 4. Auto-Tuning Parameters

Architecture	Parameter	Range (Step)
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^x)
IBM Cell	Loop-Unroll Factor	1–3 (1)
FPGA	Loop-Unroll Factor	1–15 (5)
	Operators per PE	8–64 (2^x)
	BFT Rent Parameter	0.0–1.0 (0.1)

4. EXPERIMENTAL METHODOLOGY

We now explain the experimental methodology we use when evaluating different architectures.

4.1. Development Environments

We tabulate the different compilers, tools, libraries and timing-functions used in our experiments in Table 6. We report runtime averaged across a large number of device evaluations to minimize the effect of startup costs, OS overheads and measurement noise.

4.2. FPGA Hardware Implementation

Our FPGA processing-element shown in Figure 2 consists of spatial floating-point operators coupled to on-chip memories and configured in VLIW fashion. These operators are interconnected with a time-shared network that is fully pipelined for high-performance. We limit our implementations to fit

Table 5. Peak Floating-Point Throughput

Family	Intel Xeon	Intel Core i7	Xilinx V5	Xilinx V6	IBM Cell	NVIDIA GPU	AMD GPU
Chip	5160	965	LX330T	LX760	PS3	9600 GT	AMD 9270
Technology	65 nm	45 nm	65 nm	40 nm	65 nm	65 nm	55 nm
Clock	3 GHz	3.2 GHz	200 MHz	200 MHz	3.2 GHz	1.625 GHz	750 MHz
Double-Precision (GFLOPS)	12	25.6	11.4	26	10.5	-	240
Single-Precision (GFLOPS)	24	51.2	33	75.6	204.8	312	1200
Power	80 Watts	130 Watts	20–30 Watts	20–30 Watts	135 Watts	59–96 Watts	160–220 Watts

Table 6. Software Environments

Arch.	**Compiler**	**Libraries**	**Timing**
Intel	`gcc-4.3` (-O3)	libm, Intel MKL 10.1	PAPI 3.6.2 [12], PAPI_flops()
Nvidia GPU	`nvcc`, CUDA SDK 2.1 [13]	CUDA libraries	cudaEventRecord()
IBM Cell	`spu-gcc`, `ppu-gcc`, Cell SDK 3.1 [14]	Simdmath, MASS	gettimeofday()
Xilinx FPGA	Synplify Pro 9.6.1, Xilinx ISE 10.1	CoreGen, Arénaire [15], [16]	-

Table 7. FPGA Cost Model

	Area (Slices)	**Latency** (clocks)	**Speed** (MHz)	**Ref.**
Add	296	8	280	[15]
Multiply	611	9	237	[15]
Divide	1499	57	258	[15]
Square Root	822	57	282	[15]
Exponential	1022	30	200	[16]
Logarithm	1561	30	200	[16]
PE support logic	82	-	300	-
BFT T-Switchbox	48	2	300	-
BFT Pi-Switchbox	64	2	300	-
Switch-Switch Wire	32	2	300	-

on a **single chip** and use only on-chip memory resources for storing intermediate results. You can find additional details about the VLIW architecture and the mapping tools used to implement computation on that architecture in [9].

We synthesize and implement a sample single-precision 8-operator design for the `bsim3` model on a Xilinx Virtex-5 device [17] using Synplify Pro 9.6.1 and Xilinx ISE 10.1. We provide placement and timing constraints to the back-end tools and attain a frequency of 200 MHz (See Table 7, aggressive pipelining of *exp* and *log* operators should enable higher rates).

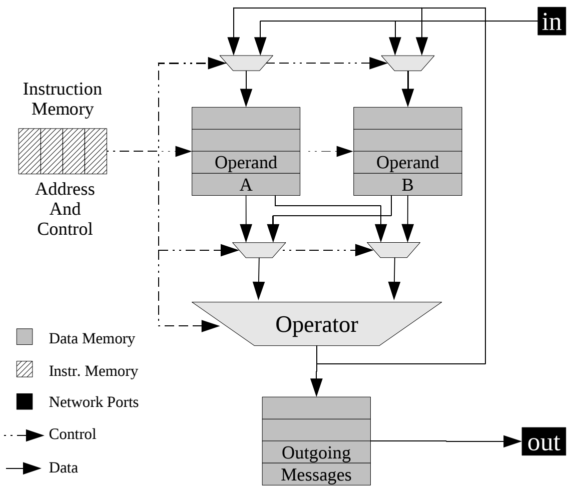


Fig. 2. Virtualized (VLIW) Operator Architecture

5. EVALUATION

In this section, we discuss the impact of single-precision model evaluation on `spice3f5` and performance across different architectures when evaulating a range of SPICE device models.

5.1. Impact of Single-Precision on `spice3f5`

In Table 8, we show the impact of performing single-precision evaluation of device models (while retaining double-precision processing of Matrix-Solve) on accuracy and iterations of `spice3f5`. With single-precision calculations we expect that there will be a small loss in simulation quality over double-precision. To allow the simulation to converge at this lower quality, we adjust the simulator's tolerance parameters *reltol* (relative tolerance), *abstol* (absolute current tolerance) and *vntol* (absolute voltage tolerance). `spice3f5` Newton-Rhapson iterations converge when, for all voltages,

Table 8. Impact of Precision

bsim3 circuits	**Single-Precision**				**Double-Precision**
	reltol	abstol	vntol	Iter.	Iter.
`ram2k`	$1e^{-3}$	$1e^{-11}$	$1e^{-4}$	656	611
`ram8k`	$1e^{-3}$	$1e^{-11}$	$1e^{-3}$	652	607
`ram64k`	$1e^{-2}$	$1e^{-11}$	$1e^{-3}$	423	420

V, and currents, I:

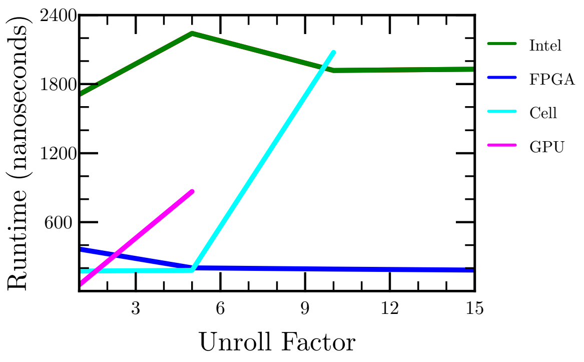
$$|V_i - V_{i-1}| \leq reltol \cdot \max(|V_i|, |V_{i-1}|) + vntol \quad (1)$$
$$|I_i - I_{i-1}| \leq reltol \cdot \max(|I_i|, |I_{i-1}|) + abstol \quad (2)$$

Here V_i or I_i represent the value of the respective voltage or current on the i-th iteration of the Newton-Rhapson loop.

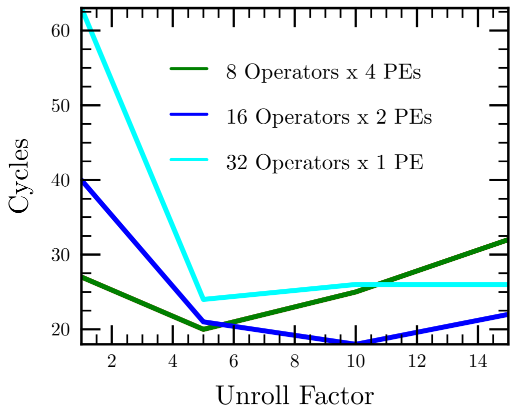
For double-precision evaluation of the models, all circuits in Table 8 converge with the default tolerance values in the simulator; $reltol = 1e^{-3}$, $abstol = 1e^{-12}$, $vntol = 1e^{-6}$. For single-precision evaluation, we observe that $abstol$ must be relaxed to $1e^{-11}$ (accuracy of 10 picoAmperes), $vntol$ must be relaxed to $1e^{-3}$ (accuracy of 1 milliVolt) and $reltol$ must be relaxed to $1e^{-2}$ (accuracy 1 part in 100). This relaxation may be acceptable for most circuit simulation scenarios. We also observe a modest 10% increase in SPICE iterations, but each iteration will be faster. Overall, once we integrate the entire simulator (part of our future work), we expect single-precision evaluation to run faster with a slight loss in result quality.

5.2. Auto-Tuning

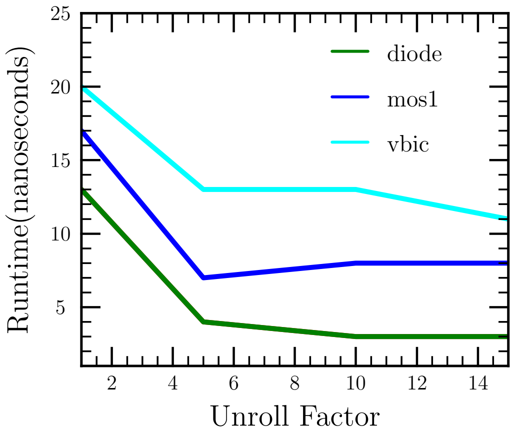
In Figure 3a, Figure 3b and Figure 3c we illustrate the auto-tuning process for different device models and different architectures using loop-unrolling as an example. Our auto-tuner will try several unroll factors over a pre-determined range and pick the one that provides best performance for each device model and architecture separately. In Figure 3a, we observe that for the large `bsim3` device model, loop-unrolling is not very useful on the Intel processor, NVIDIA GPU or the IBM Cell. In some cases, a modest amount of loop-unrolling even exhausts the finite memory resources available on the GPU and Cell. The FPGA design (8 operators/PE) however is able to improve performance by 2× for an unroll factor of 15×. Figure 3b illustrates how the choice of unroll factor interacts with the choice of another parameter in the auto-tuning process; *i.e.* number of floating-point operators in an FPGA for the `mos3` device model. We observe that best unroll factor may by 5 if using 8 or 32 operator PEs and 10 when using 16 operator PEs (Architecture details in [9]). Figure 3c shows that the choice of unroll factor varies with the device model (unroll=5 is best for `mos1`, unroll=10 is best for `diode` and unroll=15 is best for `vbic`).



(a) Impact of Unroll Factor across different architectures (`bsim3`)



(b) Interaction between Unroll Factor and PEs (`mos3`)



(c) Unroll Factor for different devices

Fig. 3. Impact of Auto-Tuning

5.3. 65nm Single-Precision Evaluation

In Figure 4a, we show the Model-Evaluation runtime for different devices using single-precision arithmetic on 65nm architectures. We observe that Intel multi-core processor is outperformed by other architectures for all device models. In Figure 4c, we see that the FPGA implementation provides the best speedup compared to all other architectures for small devices (`bjt`, `diode`, `jfet`, `mos1`, `vbic`, `hbt`) as they can effectively exploit limited FPGA resources. For the larger devices (`mos3`, `mextram`, `bsim3`, `bsim4`, `psp`) the GPU implementation is able to provide better performance than the FPGA due to lower thread scheduling overheads. FPGA performance gets worse at larger devices sizes due to several factors including larger interconnect requirement, greedy placement and long operator latencies which we will address in the near future. In Figure 4e, we observe that FPGAs are able to exploit up to 70% of the peak floating-point processing capacity of the chip while the rest of the architectures are unable to reach more than 20% of their respective peaks.

5.4. 45nm Single-Precision Evaluation

In Figure 4b, we show the performance comparison between the latest Xilinx Virtex-6 FPGA and the latest Intel Core i7 965 when using single-precision arithmetic on 45nm architectures. We observe that the FPGA is able to provide speedups between 4×–63× over the multi-core processor (Figure 4d). These speedups suggest that the performance gap between the FPGA and the multi-core processor for data-parallel applications will remain even as we scale to more advanced processes. In Figure 4f, we see that FPGAs can again achieve up to 70% of their peak capacity while the processor is still only able to use 20% of its peak.

6. FUTURE WORK

We identify the following broad areas for additional research that can improve upon our current parallel design.

- A parallel solution to the sparse **Matrix-Solve** phase is essential for achieving balanced total speedup for the SPICE application. [23] demonstrates a potential for at least 10× speedup for sparse-direct LU factorization on FPGAs.
- Reducing precision floating-point FPGA datapaths (even below single-precision) provide the potential to deliver greater acceleration per FPGA. Additional work is needed to determine the precision sufficient to achieve a given accuracy requirement.
- Additional work is necessary to address FPGA bottlenecks at large model sizes and improve FPGA performance further. It may even be possible to exploit **pipeline parallelism** to further improve GPU and Cell performance.

7. CONCLUSIONS

We are able to show that we can accelerate Single-Precision SPICE Model-Evaluation by 3×–182× when using a Xilinx Virtex5 FPGA, 1.3×-33× when using an IBM Cell, and 3×-131× when using the NVIDIA 9600 GT GPU compared to a 3 GHz Intel Xeon 5160. FPGAs are able to outperform all other architectures for small devices as the highly-optimized custom VLIW architecture can make efficient use of limited FPGA resources. For larger devices, GPUs are able to provide superior performance due to lower scheduling overheads and the greater raw operator parallelism. Our code-generation and auto-tuning framework allows us to produce high-quality parallel code across vastly different organizations. We expect auto-tuning will become increasingly important in achieving performance portability across diverse parallel architectures.

8. REFERENCES

[1] L. W. Nagel, "SPICE2: a computer program to simulate semiconductor circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975.

[2] P. Ko, J. Huang, Z. Liu, and C. Hu, "BSIM3 for analog and digital circuit simulation," in *Proceedings of the IEEE Symposium on VLSI Technology CAD*, 1993, pp. 400–429.

[3] G. Gildenblat, et al, "PSP: an advanced Surface-Potential-Based MOSFET model for circuit simulation," *IEEE Transactions on Electron Devices*, vol. 53, no. 9, pp. 1979–1993, 2006.

[4] L. Lemaitre, G. Coram, et al, "Extensions to Verilog-A to support compact device modeling," in *Proceedings of the Intl. Workshop on Behavioral Modeling and Simulation*, 2003, pp. 134–138.

[5] B. Wan, B. Hu, L. Zhou, and C. Shi, "MCAST: an abstract-syntax-tree based model compiler for circuit simulation," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2003, pp. 249–252.

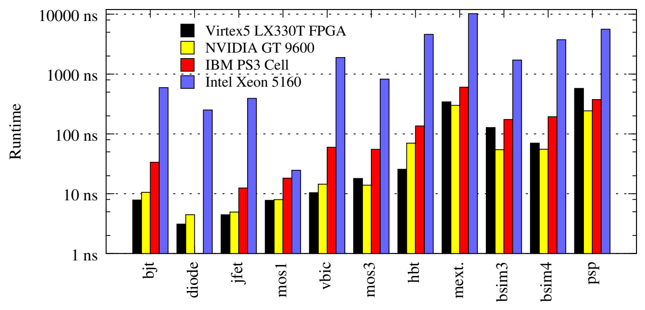
[6] "Open-Source Simucad Verilog-A models,"

[7] A. M. Bayoumi and Y. Y. Hanafy, "Massive parallelization of SPICE device model evaluation on GPU-based SIMD architectures," in *1st Intl. ACM Forum on Next-generation multicore/manycore technologies*. 2008, pp. 1–5.

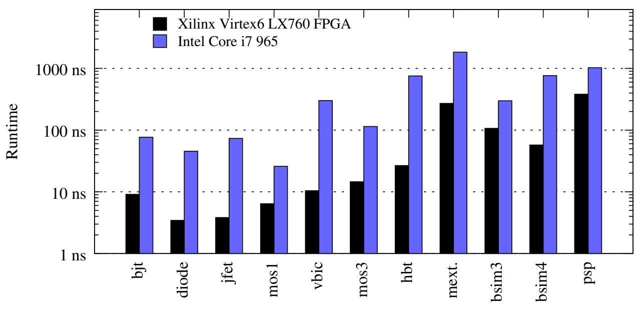
[8] K. Gulati, J. F. Croix, S. P. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Design Automation Conference, Asia and South Pacific*, 2009, pp. 403–408.

[9] N. Kapre and A. DeHon, "Accelerating SPICE Model-Evaluation using FPGAs," in *Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.

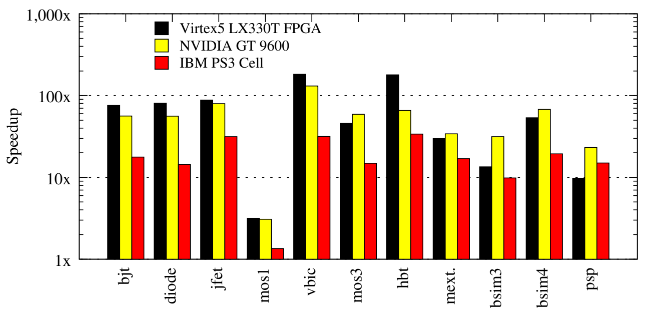
[10] B. Nichols and D. Buttlar, *Pthreads programming*. O'Reilly Media, Inc., 1996.



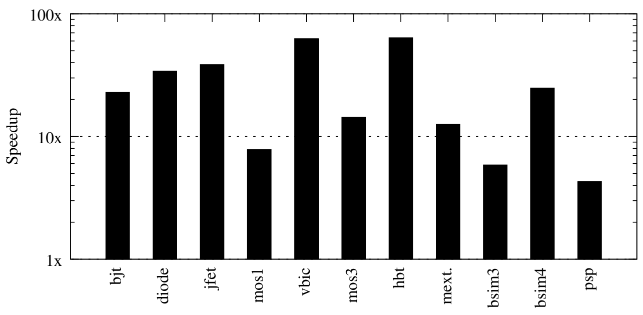
(a) Runtime Comparison (65nm)



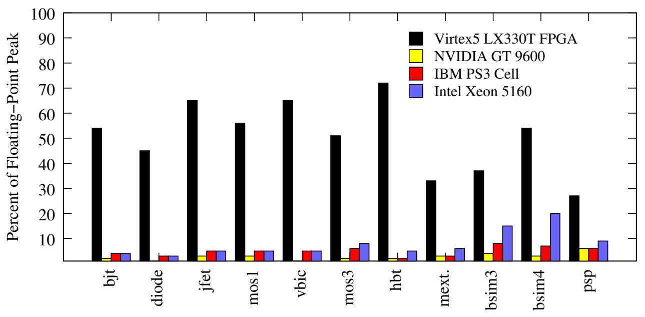
(b) Runtime Comparison (45nm)



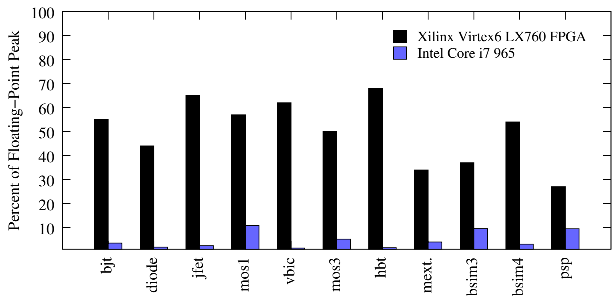
(c) Speedup over Intel Xeon 5160 (65nm)



(d) Speedup over Intel Core i7 965 (45nm)



(e) Percentage of Floating-Point Peak (65nm)



(f) Percentage of Floating-Point Peak (45nm)

Fig. 4. Performance Analysis on 65nm and 45nm Architectures

[11] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27.

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: a portable interface to hardware performance counters," in *Proc. Dept. of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[13] NVIDIA, *Compute Unified Device Architecture (CUDA) SDK 2.1*, 2008.

[14] IBM, "Cell SDK 3.1," 2008.

[15] "Floating-Point operator v4.0 datasheet" April 2008

[16] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems*, vol. 31, no. 8, pp. 537–545, Dec. 2007.

[17] Xilinx, *Virtex-5 Datasheet*, 2006.

[18] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

[19] Y. Lin, "Recent developments in high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, no. 1, pp. 2–21, 1997.

[20] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 768–781, 1989.

[21] N. Kapre, N. Mehta, et al, "Packet switched vs. time multiplexed FPGA overlay networks," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216.

[22] N. Mehta, "Time-multiplexed FPGA overlay networks on chip," Master's Thesis, California Institute of Technology, Pasadena, 2006.

[23] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU decomposition using FPGA," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.