Automatic Synthesis of Physical System Differential Equation Models to a Custom Network of General Processing Elements on FPGAs

CHEN HUANG and FRANK VAHID, University of California, Riverside
TONY GIVARGIS, University of California, Irvine

Fast execution of physical system models has various uses, such as simulating physical phenomena or real-time testing of medical equipment. Physical system models commonly consist of thousands of differential equations. Solving such equations using software on microprocessor devices may be slow. Several past efforts implement such models as parallel circuits on special computing devices called Field-Programmable Gate Arrays (FPGAs), demonstrating large speedups due to the excellent match between the massive fine-grained local communication parallelism common in physical models and the fine-grained parallel compute elements and local connectivity of FPGAs. However, past implementation efforts were mostly manual or ad hoc. We present the first method for automatically converting a set of ordinary differential equations into circuits on FPGAs. The method uses a general Processing Element (PE) that we developed, designed to quickly solve a set of ordinary differential equations while using few FPGA resources. The method instantiates a network of general PEs, partitions equations among the PEs to minimize communication, generates each PE's custom program, creates custom connections among PEs, and maintains synchronization of all PEs in the network. Our experiments show that the method generates a 400-PE network on a commercial FPGA that executes four different models on average 15x faster than a 3 GHz Intel processor, 30x faster than a commercial 4-core ARM, 14x faster than a commercial 6-core Texas Instruments digital signal processor, and 4.4x faster than an NVIDIA 336-core graphics processing unit. We also show that the FPGA-based approach is reasonably cost effective compared to using the other platforms. The method yields 2.1x faster circuits than a commercial high-level synthesis tool that uses the traditional method for converting behavior to circuits, while using 2x fewer lookup tables, 2x fewer hardcore multiplier (DSP) units, though 3.5x more block RAM due to being programmable. Furthermore, the method does not just generate a single fastest design, but generates a range of designs that trade off size and performance, by using different numbers of PEs.

Categories and Subject Descriptors: B.5.2 [**Design Aids**]: Automatic Synthesis; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; C.1.4 [**Parallel Architectures**]: Distributed Architectures

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: Physical system modeling, physical system simulation, FPGAs, embedded systems, processor network, synthesis, differential equations, cyber-physical systems, custom processors, application-specific processors

ACM Reference Format:
Huang, C., Vahid, F., and Givargis, T. 2013. Automatic synthesis of physical system differential equation models to a custom network of general processing elements on FPGAs. ACM Trans. Embedd. Comput. Syst. 13, 2, Article 23 (September 2013), 27 pages.
DOI: http://dx.doi.org/10.1145/2514641.2514650

This work was supported in part by the National Science Foundation (CNS-1016792, CPS-1136146) and by the Semiconductor Research Corporation (GRC-2143.001).
Authors' addresses: C. Huang (corresponding author) and F. Vahid, Computer Science Department, University of California, Riverside, CA; email: chuang@cs.ucr.edu; T. Givargis, University of California, Irvine, CA.

DOI: http://dx.doi.org/10.1145/2514641.2514650

1. INTRODUCTION

In cyber-physical systems [Lee 2008], computers or devices interact with the physical world. Developing digital models that accurately and quickly emulate physical systems has several uses. For instance, if a digital human lung model can execute in real time with sufficient accuracy, the lung model can be used to test medical ventilators in real time, to train medical professionals on ventilator use in real time, or even to run regression tests faster than real time. Another cyber-physical system scenario involves model-predictive control, wherein a controller like an aircraft flight system uses real-time predictions of physical system reactions to guide control actions [Gholkar et al. 2002]. Even beyond cyber-physical systems, simulating physical systems is commonly done in various branches of science, such as in physics and chemistry [Motuk et al. 2005; Osana et al. 2004]. Such simulations commonly require weeks or longer. Such use cases represent our main motivations.

Due to the complexity of physical models, previous approaches often use highly simplified models for emulation speed [Heart Simulator 2011]. Simplified models may not reflect the behavior of real physical systems with sufficient accuracy. Higher accuracy requires more complicated models and thus requires more computation capability. Physical models are often captured with Ordinary Differential Equations (ODEs). A complex physical model may contain thousands of ODEs (for simplicity, we refer to a system of ODEs of dimension m as m ODEs in this article). Iterative ODE solution methods require that all equations be evaluated frequently, such as every 1ms or faster, to maintain accuracy. Today's high-end desktop processors often cannot execute complex physical system models in real time, because the mostly serial software execution on a processor does not match the massively parallel nature of physical models.

A physical system model often contains inherent massive parallelism, which is intuitive as the physical world tends to involve items (e.g., human cells) operating in parallel. As such, a model's ODEs can be evaluated concurrently at each time step of an iterative solver. The ODEs also have high data locality and localized data transfer, again reflecting the physical world's tendency for local connectivity. High data locality and localized data transfer patterns are well suited to Field-Programmable Gate Arrays (FPGAs), which effectively support massively parallel computations, distributed data storage, and custom localized communication. The bottleneck in FPGAs is typically centralized data access, which is common in many typical computer applications but does not exist in most physical system models.

We propose a network of general processing elements (referred to as PEs throughout the article), intended for FPGAs, to efficiently solve the ODEs of a physical system. Each PE is a lightweight programmable processor whose design we optimized for solving ODEs. The data transfers between different PEs are implemented with synchronized point-to-point connections. The structure of the custom network mimics the real physical structure of a physical model, thus providing highly effective synthesized FPGA circuits in terms of both performance and resource usage compared to circuits generated by the standard automated approach of high-level synthesis [McFarland et al. 1990].

We created a PE synthesis method, embodied in a PE synthesis tool, to automatically convert a model's ODEs into VHDL (VHSIC Hardware Description Language) [VHDL 2011] files that can be synthesized to an FPGA. The tool automatically maps the ODEs to multiple PEs, and searches for the best mapping of ODEs to PEs such that the interconnections among PEs is minimized. Furthermore, rather than creating a single design, the tool generates a design space that shows trade-offs among design size and performance by using different numbers of PEs, so that the designer can choose a design based on a target application's requirements. Figure 1 shows an example of mapping

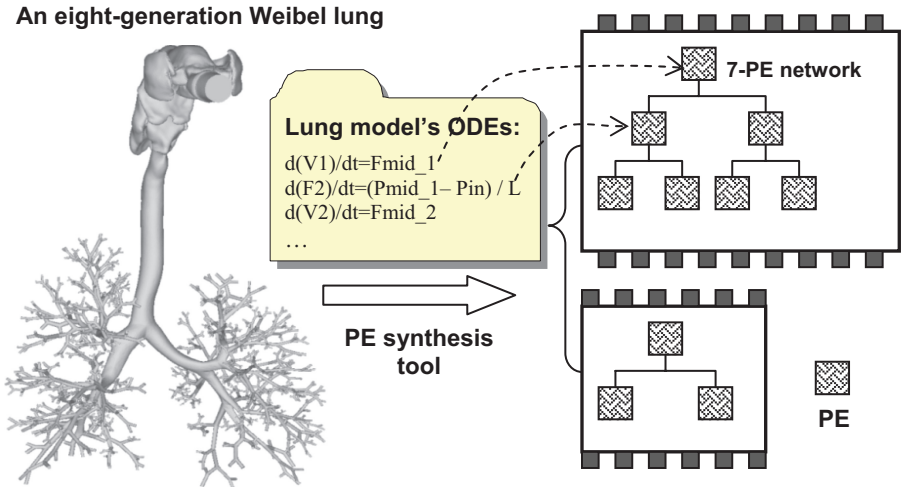


Fig. 1. Synthesizing an eight-generation Weibel lung model into networks of PEs on FPGAs.

ODEs of an eight-generation Weibel lung model [Lin et al. 2009] to networks of PEs. The PE synthesis tool reads the ODEs of the model and generates two designs that have different size and performance. Note that the generated networks have similar binary tree structures compared to the Weibel lung model.

The article is organized as follows. Section 2 reviews related work. Section 3 reviews modeling physical systems using ODEs. Section 4 describes the architecture of the network of PEs, and Section 5 describes the PE compiler and automatic ODE-to-PE mapping algorithm, these two sections containing the key contribution of this article. Section 6 shows experimental results of our PE synthesis method compared with numerous alternative devices and approaches for executing physical models, and Section 7 concludes.

2. RELATED WORK

Modeling and simulation of physical systems have been studied extensively in different fields. Physiological models are developed to help with understanding and analyzing mechanical, physical, and biochemical functions of the human body. Electromagnetic models are developed to understand and predict electromagnetic behavior, and are widely used in design of cellular phones, mobile computing, etc.

Languages have been introduced for modeling physical systems, such as the Mathematical Modeling Language (MML) [NSR Physiome Project 2011], the Systems Biology Markup Language (SBML) [Hucka et al. 2004], and CellML [2011]. Tools were developed for simulating physical systems, such as Matlab [Mathworks 2011], LabView [National Instruments 2011], JSim [2011], and Mathematica [2011]. These tools are usually aimed at producing accurate simulation results, rather than emphasizing real-time simulation.

Many efforts aim to increase execution speed for complex physical models using general-purpose processors or Graphics Processing Unit (GPUs) [ATI Graphics Cards 2011; Nvidia Corporation 2011]. Multicore processors [AMD 2011; Intel Corporation 2011] and supercomputers [Cray 2011; IBM Blue Gene 2011] have been utilized to exploit the parallelism inside physical models. For instance, a 768-core SGI machine executed a complex 2-billion equation heart model, simulating 0.4ms in 2 hours (18 million

times slower than real time) [MedGadget 2008]. An Nvidia GTX 295 GPU was used to execute a Flaim heart model 30x faster than OpenMP, with less than 1% error [Lionetti 2010]. Executing one heart beat (300ms) required 7.7 minutes. While multicore processors and GPUs are capable of doing parallel computation, their communication architectures do not match the local neighbor communication of many physical models, so the data transfer between different cores may cause memory contention or other communication bottlenecks. Designers also need extra design time and expertise to efficiently write multithreaded programs for multicore and GPU.

Many case studies using FPGAs to speed up simulation have been conducted. FPGAs were used to speed up fine-grained intra-cellular simulation [Salwinski and Eisenberg 2004], showing that an FPGA could hold 500 reactions related to gene expression. Yoshimi et al. [2004] obtained 100x speedups of a fine-grained biological simulation compared to a single processor, and showed why multicores are not suitable for speeding up fine-grained biochemical reactions. FPGAs have been used to simulate a heart-lung system model in real time for the purpose of testing medical devices [Pimentel and Tirat-Gefen 2006]. That work showed that a PC requires 1.5 hours to simulate 60 seconds of real time for that model, while an FPGA solution ran in real time. Their FPGA performance was calculated by a theoretical formula based on the number of multipliers and the performance of each multiplier of their target FPGA, rather than an implementation. Chen et al. [2009] used FPGAs to do an inductive dynamic simulation with a Runge-Kunta ODE solver. The custom FPGA implementation resulted in a 100x speedup over a 2.2 GHz desktop computer using Simulink [2001]. Osana et al. [2004] developed the ReCSiP tool to generate chemical models on FPGAs using the SBML language. The crossbar communication structure used in ReCSiP supports dozens of solvers but may not scale to larger models. Iwanaga et al. [2005] used FPGAs to simulate ODE-based multimodel biochemical simulations, proposing several scheduling and resource sharing methods to optimize implementation on FPGAs. The aforesaid efforts mostly used manual design to implement the physical models on FPGAs, requiring much human effort for design and test. This article proposes a systematic and scalable approach to synthesize physical system models into a custom network of general PEs on an FPGA.

A no-instruction-set-computer concept was introduced by Reshadi et al. [2005]. That work involved creation of a C-to-RTL synthesis tool to generate custom instructions on a given datapath, eliminating the need for instruction decoding logic. Our PE uses a similar idea to encode control words into instruction memory. Our PE is typically smaller and less flexible than a NISC processor due to our focus on differential equation solving, and we also emphasize synthesis of a custom communication structure for a network of PEs.

Another common approach for implementing applications on FPGAs uses high-level synthesis, also known as behavioral synthesis. Many tools have been developed to generate circuits from a high-level representation like C, Matlab, Java, etc. Major high-level synthesis approaches and tools include SA-C [Najjar et al. 2003], Streams-C [Gokhale et al. 2000], DEFACTO [Diniz et al. 2001], SPARK [SPARK Project 2005], ROCCC [Buyukkurt et al. 2006], Celoxica [2011], SynphonyC [2011], DRFM [Cong et al. 2008], etc. We use a commercial high-level synthesis tool in this work for comparison purposes. We show in Section 6.2 that our tool generates faster and smaller implementations due to our tool's specific focus on a network of ODEs.

3. PHYSICAL SYSTEM MODELING AND ODE SOLVING

This section reviews physical system modeling with ODEs, using a Weibel lung as a driver example. The section emphasizes the ODE solving process and data dependencies among ODEs, and describes four different physical models used in the article.

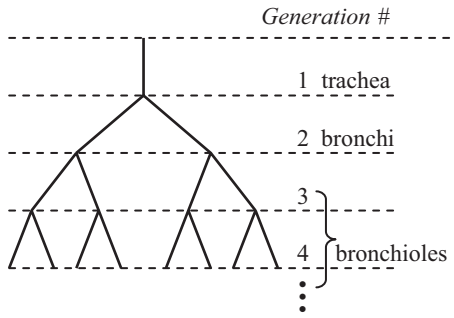
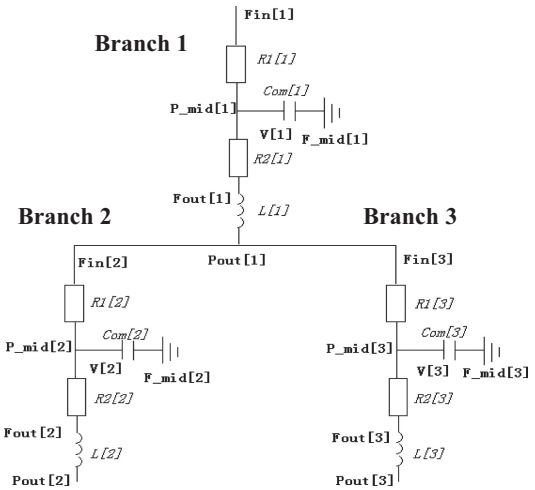


Fig. 2. The first four generations of a Weibel lung model.



ODEs for branch 1:

P_mid[1] = V[1] / Com[1]
Fout[1] = Fin – F_mid[1]
d(V[1]) /dt = F_mid[1]
d(Fout[1])/dt = (P_mid[1] –
*Pout[1] – Fout[1] * R2[1])/ L[1]*

ODEs for branch 2:

P_mid[2] = V[2] / Com[2]
Fin[2] = Fout[1] – Fin[3]
Fout[[2] = Fin[2] – F_mid[2]
d(V[2]) /dt = F_mid[2]
d(Fout[2])/dt = (P_mid[2] –
*Pout[2] – Fout[2] * R2[2]) / L[2]*

…

Fig. 3. RLC circuit modeling of a bifurcating airway.

3.1. Modeling Physical Systems with ODEs

As an example of modeling a physical system, consider modeling a human's lungs. Weibel [1963] proposed a lung model having a binary tree structure to reflect a human lung's anatomic structure. Figure 2 shows the structure of the first four generations of the Weibel lung model. Generation 1 represents the airway (or trachea). Generation 2 represents two bronchi. Generations 3–20 represent smaller bronchioles, and generations 20–23 contain millions of alveoli that handle gas exchange between the lung and capillaries. In the structure, each line segment within a generation is known as a *branch*. Splitting of a branch is called *bifurcating*. Due to the exponential increase in the number of branches for each generation, a Weibel model typically includes fewer than 23 generations. The total number of branches in the model will equal $2^n - 1$, where n is the deepest generation number.

A bifurcating airway of the Weibel lung structure can be modeled as the RLC circuit shown in Figure 3. The physical property of each branch is captured with R (*resistance*), *Com* (capacitance, or *compliance* in lung terminology), and L (*inductance*). Each branch has unique R, *Com*, and L values according to the branch's physical properties. For example, deeper generations have larger resistances and smaller capacitances. The relevant variables during simulation are F (flow), P (pressure), and V (volume) of each

branch. For instance: *Fin[1]* is the input flow for branch 1, *Fout[1]* is the output flow of branch 1, and *P_mid[1]* is the inner pressure of branch 1.

3.2. ODE Solving Process and Data Dependencies among ODEs

The equations illustrated in Figure 3 model a bifurcating airway. These equations include *ordinary equations*, for example, *P_mid[1] = V[1]/Com[1]* (middle pressure of branch1 is equal to volume of branch1 divided by capacitance of branch1), as well as *Ordinary Differential Equations (ODEs)*, for example, $d(V[1])/dt = F\_mid[1]$ (the derivative of branch1's volume is equal to the middle flow of branch1).

The variables on the left-hand side of ODEs are *state variables*, for example, *V[1]*, *Fout[1]*. The ordinary equations calculate temporary values that are used in the ODEs, for example, *P_mid[1]* is used for calculating $d(Fout[1])/dt$. Substituting temporary values yields the general ODE format: $d(X)/dt = Fun(X)$, where X is a vector of the state variables: *V[1], V[2], V[3], Fout[1], Fout[2], Fout[3]*. The derivative of X is a function of X.

To solve these ODEs, iterative solvers such as Euler [Atkinson 1993] or Runge-Kutta [Butcher 2003] are often used. Starting from time 0, iterative solvers move forward in time by a given *time step*, such as by 1ms. Note that there are data dependencies among ODEs. For instance, the ODE $d(Fout[1])/dt = (\mathbf{V[1]}/Com[1] - Pout[1] - Fout[1] * R2[1])/L[1]$ depends on the ODE $d(V[1])/dt = F\_mid[1]$, because *V[1]* is updated by the second ODE at each step, and the new value is needed by the first ODE. We use the Euler method to describe the ODE solving process here. At each step, we divided the solving process into the following three stages.

(1) *Evaluate.* Calculate the derivative of each state variable, such as $d(V[1]/dt) = Fin - Fout[1]$.
(2) *Update.* Update each state variable using the derivatives calculated in the evaluate stage, such as $V[1] = V[1] + d(V[1]/dt) * dt$, where dt is a time step.
(3) *Data transfer.* Propagate the new value of each state variable to the ODEs that depend on the state variable.

These three stages present the basic idea of the parallel version of the ODE solving process, where each ODE is mapped to a different processor. At the beginning of each time step, we assume each processor has the latest values of state variables on the right-hand side of the ODE, for example, *V[1], Fout[1]* in the following ODE.

$$d(Fout[1])/dt = (\mathbf{V[1]}/Com[1] - Pout[1] - \mathbf{Fout[1]} * R2[1])/L[1].$$

Thus the *evaluate* and *update* stages can be calculated in parallel on each processor. However, new state variable values need to be transferred between processors according to ODE data dependencies at the *data transfer* stage.

The Euler method has an error proportional to dt^2 per step. The Runge-Kutta method gives better accuracy. The classical RK4 method calculates the derivative of each state variable four times (at beginning, midpoint (twice), and end of the interval) per step, having error proportional to dt^5 per step (note that dt is less than 1, thus $dt^5 < dt^2$).

3.3. Physical Model Examples

This article uses four complex physical models as examples. The first is the Weibel lung model described earlier. The second is an entirely different lung model called the Lutchen lung [Lutchen et al. 1982]. The third is a wave model [Motuk et al. 2005]. The fourth is an atrial cell model [Zhang et al. 2001]. Figure 4 briefly shows the physical structure of the three additional models.

Figure 4(a) shows the structure of the Lutchen lung model, which contains three components: a nondispersive airway and two alveolar compartments. The Lutchen lung

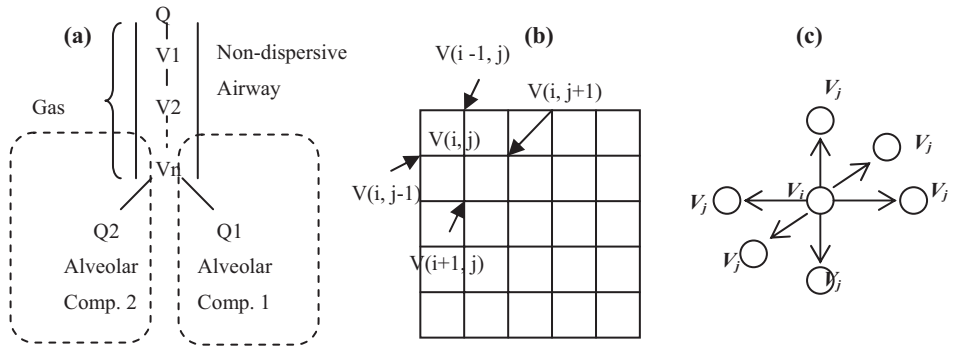


Fig. 4. Three additional physical model examples: (a) Lutchen lung model; (b) wave model; (c) atrial cell model.

emphasis is on modeling gas exchange in the nondispersive airways, each of which is segmented into a number of gas cells. Each gas cell only connects with two neighbor cells, for example, *V2* only connects with *V1* and *V3*.

The wave model, or Finite Difference Time-Domain (FDTD) model, is an important physical model in electromagnetics. The basic structure of the wave model is the grid shown in Figure 4(b). Each node in the grid has a value representing the amplitude of the wave at that point. Each node only communicates with its four neighbor nodes. For instance, $V(i, j)$ only communicates with nodes $V(i, j-1)$, $V(i, j+1)$, $V(i-1, j)$, $V(i+1, j)$.

Figure 4(c) shows a three-dimensinal atrial cell model intended to model a heart for interacting with a pacemaker [Zhang et al. 2001]. Each node represents an atrial cell, and *Vi* stands for the membrane potential for cell *i*. Each atrial cell only communicates with its six neighbor cells (*Vj*) in a three-dimensinal space, as shown in the figure.

The four physical model examples represent four different connection structures: linear (Lutchen lung), binary tree (Weibel lung), grid (wave model), and three-dimensional cubic (atrial cell). In this work, we use an 11-generation Weibel lung with 4094 ODEs, a 4000-cell Lutchen lung with 4000 ODEs, an 80×80 wave model with 6400 ODEs, and a $15 \times 15 \times 15$ atrial cell mode with 3375 ODEs. The four physical models are denoted as *Weibel 11*, *Lutchen 4000*, *Wave 80 × 80,* and *Atrial cell 15* throughout the article.

4. GENERAL PROCESSING ELEMENT ARCHITECTURE AND A CUSTOM NETWORK OF PES

This section summarizes the architecture of a general Processing Element (PE), and then introduces the communication architecture for a custom network of PEs.

4.1. Background: Basic PE Architecture

We earlier proposed an architecture for a single general PE optimized for general ODE solving purposes [Huang et al. 2011]. The goals included minimizing the PE's FPGA resource requirements while also maximizing the PE's performance for solving ODEs. The general PE has the flexibility of solving different types of ODEs.

Figure 5 reviews the basic (nonpipelined) PE architecture. The PE has multiple input ports (three in the figure) and an output port to handle communication between the PE and other PEs/external modules. The PE has a data RAM that works as a register file, a programmable instruction RAM that stores the control word for each instruction, and an ALU component that reads data from data RAM and performs an operation.

To reduce the PE's resource usage, we used microcoded control words [Agarwal et al. 1986] to eliminate instruction decoding logic and thus improving size and performance, similar to the idea of the No-Instruction-Set Computer (NISC) [Reshadi et al. 2005]. The PE has two types of control words: store and compute. A store control word stores data

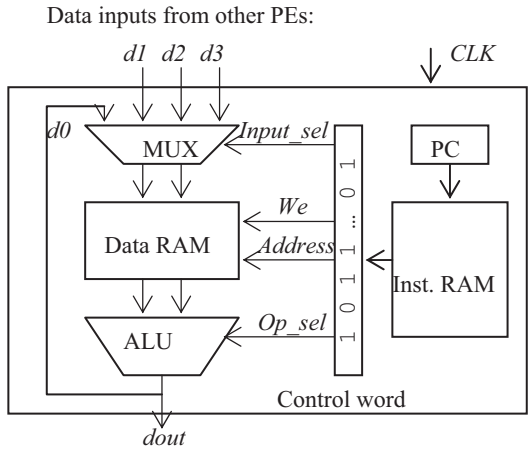


Fig. 5. Nonpipelined PE architecture.

from its own output, another PE, or from an external input. A compute control word performs a certain computation using data from the data RAM. A detailed discussion about these two operations appears in Huang et al. [2011]. The number of input ports, data, and instruction RAM sizes, and ALU operations can each be adjusted to the ODEs mapped to the PE. Since an ODE can be parsed into basic operations for the ALU, the general PE can solve different types of ODEs.

Currently, we manually convert floating-point numbers into 32-bit fixed-point numbers that can be executed efficiently with the integer ALU and shift operator in the PE. To obtain a precise conversion, we estimate the value range for each variable using a floating-point simulation [Kum et al. 2000]. The scale factor of each variable can be determined by the equation: MAX_{int32}/MAX_i, where MAX_{int32} and MAX_i stand for the maximal value of a 32-bit integer and the maximal simulated value of variable i.

To verify the accuracy of the fixed-point conversion, we tested the Weibel 11 model, in which the flow and volume change significantly at each generation. We compared the fixed-point version results with a double-precision floating-point implementation in Matlab. The maximal relative error among all the variables (from different generations) is within 0.5%, which shows the fixed-point computation gives nearly identical results compared to the floating-point implementation.

The worst-case manual fixed-point conversion time for the four models is approximately 1 hour. The manual conversion could be automated in the future using any one of several established techniques, such as Kum et al. [2000].

We also invested the possibility of using floating-point components in the FPGA, and found the floating-point implementation would yield approximately a 3x–8x performance decrease and a 3x–5x size increase. Since the fixed-point computation gives comparable accuracy compared to the floating-point implementation, we use fixed point in our current design.

4.2. PE Performance Optimization

The longest register-to-register delay, known as the critical path, determines a PE's maximum clock frequency. The critical path in a basic PE is from the data RAM through the ALU and back to the data RAM. This path can become especially long when large numbers of PEs are implemented on an FPGA, because block RAM (used in data RAM) and DSP units (used in ALUs) have fixed locations on the FPGA and thus a PE in a highly utilized FPGA may have to use distant block RAM and DSP

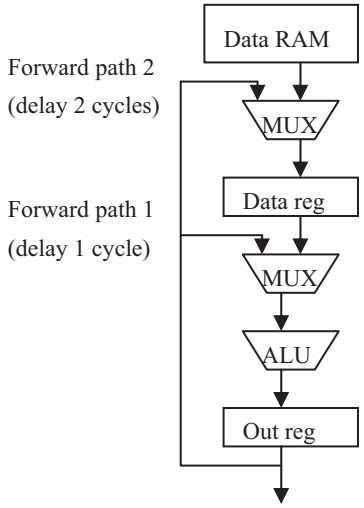


Fig. 6. Pipelined PE architecture.

units, resulting in long wire delays. To reduce the critical path, in this work we add pipeline registers as shown in Figure 6. We add two pipeline registers, Data reg and Out reg, into the PE datapath. The critical path of the pipelined design is: Data reg -> ALU -> Out reg, thus eliminating the wire delay between DSP units and block RAM. Based on our synthesis results, the clock frequency of the pipelined architecture's clock frequency increased from 100 MHz (for the nonpipelined version) to 170 MHz for the pipelined version, for some large designs.

However, pipelining increases the number of control words, requiring more instruction RAM. The nonpipelined PE can perform a computation and result write-back with one control word, for example, *data RAM[5]*data RAM[8] -> data RAM[10]*. However, the pipelined PE needs one control word to write the result into the Out reg, and then another control word to write the result to data RAM. To reduce the number of required words, we introduced two new paths, known as *forward paths*, such that temporary results can be directly forwarded to the ALU for the next computation rather than having to first be written back to the data RAM. Our experiments show that the two forward paths nearly eliminate the need to write temporary values back to data RAM, such that the pipelined architecture uses only 10% more control words than the nonpipelined architecture. In terms of resource usage, the pipelined architecture incurs a LUT penalty of 10% to 20%.

4.3. Custom Network of PEs and Communication

Using multiple PEs can improve performance by solving ODEs in parallel. A simple parallelization approach, used here for illustration, maps one ODE to one PE, that is, a one-to-one ODE-to-PE mapping. Due to data dependencies among ODEs, each PE must communicate with other PEs. Figure 7 shows a 3-generation Weibel lung model, having 7 ODEs, mapped to a 7-PE network. Note that the communication Global clock structure of the network has a similar binary tree topology compared to the 3-generation Weibel lung, because the single ODE of each Weibel lung branch only communicates with the ODE's parent and child branches.

For a given set of ODEs mapped to PEs, we create a custom point-to-point connection structure. All PEs are synchronized with a global clock, and the data transfers are statically scheduled at compile time. This statically synchronized communication

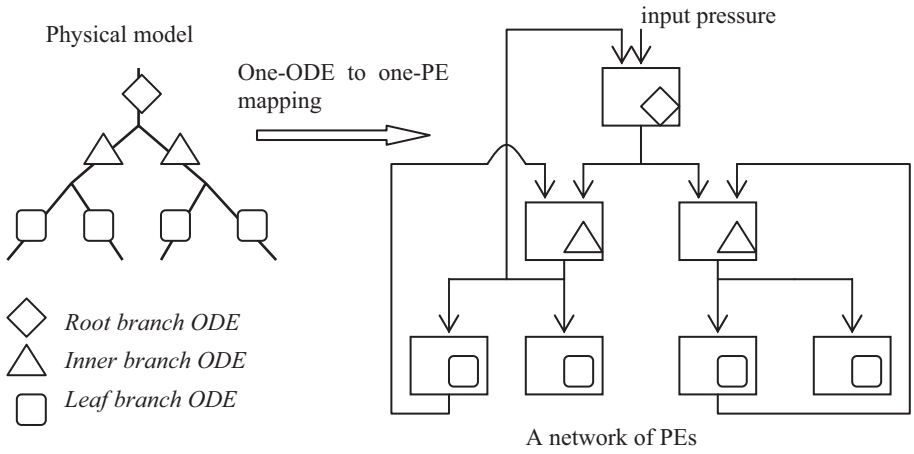


Fig. 7. Mapping a 3-generation Weibel lung to a custom network of PEs.

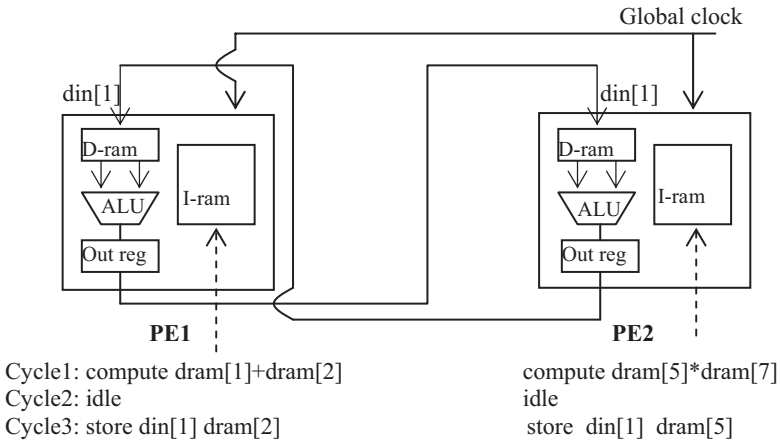


Fig. 8. Synchronized data transfer between PEs with global clock and point-to-point connection.

scheme eliminates resource-costly handshaking logic. A simple bidirectional data transfer between two PEs is illustrated in Figure 8. PE1 and PE2 each have its output connected to the other's input port. The data exchange between two PEs takes three clock cycles. For instance, PE1 and PE2 each perform a computation in Cycle 1, then are idle in Cycle 2 to let the result latch into the out register. In Cycle 3, PE1 and PE2 can read the data produced by each other. The point-to-point communication network is custom to each physical model (similar to the physical structure). Any pair of PEs can communicate in parallel, which is more efficient (in terms of size and performance) than general-purpose communication structures.

Rather than a one-to-one ODE-to-PE mapping, a many-to-one mapping can be considered, where multiple ODEs execute serially on a single PE. Mapping multiple ODEs to a single PE reduces performance by reducing parallelism, yet may better utilize the special locality of a physical model. For example, mapping nearby variables to the same PE would reduce the communication overhead. A many-to-one mapping also reduces the required FPGA resources, enabling larger models to fit onto an FPGA. The next section discusses the exploration of different mappings.

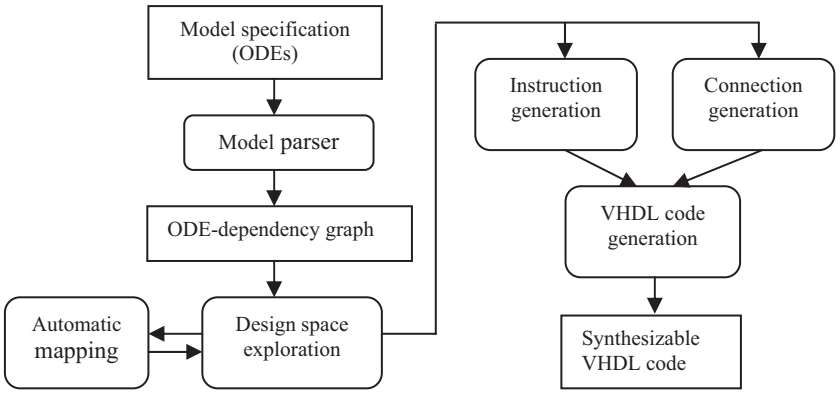


Fig. 9. PE synthesis and compiler tool overall structure.

Miller et al. [2011] describe how to integrate a PE or a network of PEs into a system for cyber-physical system testing where the PEs represent a digital mockup of the physical system.

5. SYNTHESIS AND COMPILATION OF THE CUSTOM NETWORK OF PES

We developed algorithms and tools to automatically synthesize a custom network of PEs for a given set of ODEs, and to automatically compile these ODEs into the control words for each PE. The tool's overall structure is illustrated in Figure 9.

The tool reads in a model consisting primarily of a set of ODEs. The model parser builds an ODE-dependency graph that describes the data dependencies among the ODEs. The tool then performs a custom design-space exploration with an automatic ODE-to-PE mapping algorithm, and generates designs with different sizes and performances. Finally, the tool generates PE control words, a custom network of PEs, and synthesizable VHDL files for FPGA implementation using standard synthesis tools.

5.1. Model Specification and ODE-Dependency Graph

Figure 10(a) shows a sample model specification file for a 2-generation Weibel lung model. The model specification file includes three sections. The *method-based parameters* include a solver method (Euler or RK4) and the solver time step in seconds. The second section includes model parameters (e.g., resistance and compliance of each branch), initial values of state variables (e.g., the initial volume and flow of each branch), and external inputs (e.g., input flow to the lung). The last section contains the ODEs and other equations that describe the model's behavior. The tool builds an ODE-dependency graph according to ODE data dependencies of the model, as in Figure 10(b).

5.2. Automatic Design-Space Exploration and ODE-to-PE Mapping

A complex physical model usually contains thousands of ODEs, so the corresponding ODE-dependency graph contains thousands of nodes. The FPGA used in this article is able to accommodate hundreds of PEs, thus the tool needs to map multiple ODEs to each PE. The mapping of ODEs to PEs determines the communication structure of the custom network. Figure 11 shows an example of mapping 31 ODEs to 8 PEs. A good mapping will generally reduce the design size and improve performance. We thus developed an automatic ODE-to-PE mapping algorithm to search for a good mapping.

Since each ODE can be mapped to any PE and all PEs are functionally equivalent, the total number of all possible mapping is: $m^n/m!$, where m is the number of PEs and n is the number of ODEs. An exhaustive search is not feasible for large models

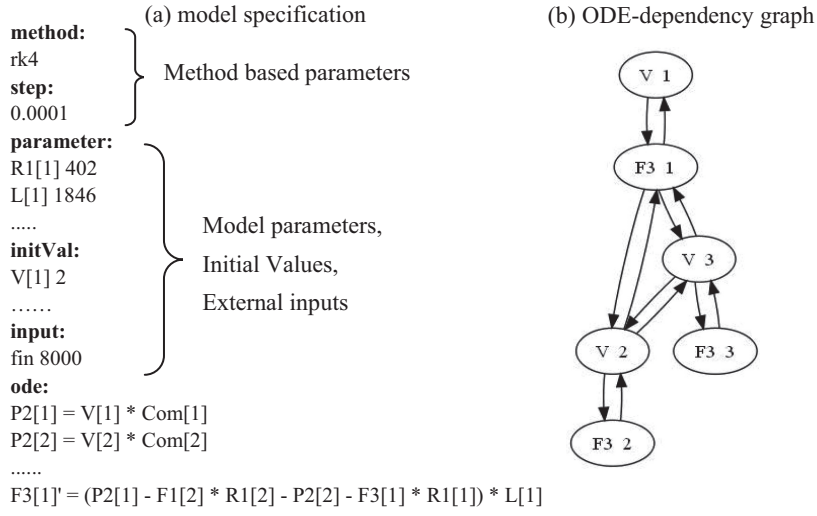


Fig. 10. A 2-generation Weibel lung model specification and ODE-dependency graph.

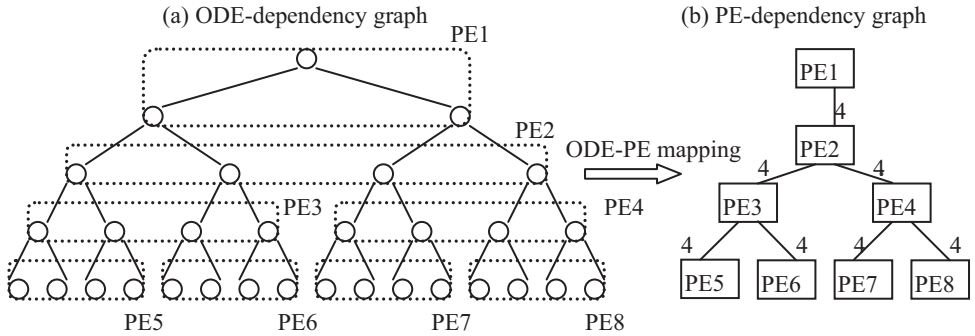


Fig. 11. ODE-to-PE mapping.

with thousands of ODEs. The ODE-to-PE mapping problem can be reduced to balanced graph partitioning, which is NP-complete [Andreev and Racke 2006]. Hence, we use an ODE-to-PE mapping algorithm that involves an iterative search heuristic based on simulated annealing.

5.2.1. Objective and Cost Function. The objective of the mapping algorithm is to minimize both ODE solving time and design size. Since the network of PEs uses a synchronized communication scheme, the system performance is mostly determined by the *bottleneck PE*, namely the PE requiring the most computation cycles. We denote #*cycle* as the number of cycles of the bottleneck PE. The design size is mostly determined by the total number of connections (abstract connections between PEs, not the physical wires in the FPGA implementation) in the network, denoted as #*connection*. To combine these two metrics, we use a cost function #*cycle* * #*connection*, with the objective being to minimize the cost.

To calculate the cost function, the mapping algorithm needs to calculate the number of cycles of each PE and find the PE with the highest cycles (#*cycle*). The number of cycles of each PE is determined by parsing the ODEs into PE instructions (including computation and communication instructions). To calculate the total number of

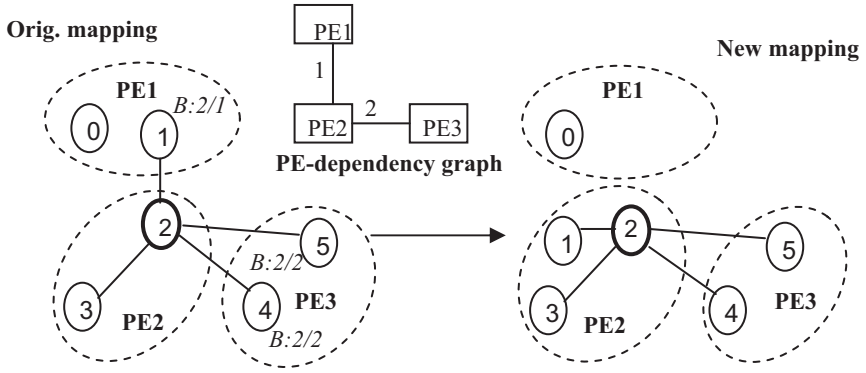


Fig. 12. Size neighbor generator (B: m/n stands for benefit: #ODE/edge weight).

connections, the mapping algorithm builds a *PE-dependency graph* according to the current ODE-to-PE mapping and the ODE-dependency graph. The PE-dependency graph building process is illustrated in Figure 11. An *edge* in the PE-dependency graph shows the physical connections, and *edge weights* reflect how many edges are in the original ODE-dependency graph (e.g., PE1 and PE2 have four edges in the ODE-dependency graph). The total number of connections is then the total number of edges in the PE-dependency graph (e.g., 7 in Figure 11).

The mapping algorithm iteratively generates a new mapping from the current mapping (denoted as a *neighbor mapping*), calculates the cost of the neighbor mapping, and decides whether to accept the neighbor mapping. The previously described steps are called an *iteration*. To speed up the algorithm, we developed an incremental cost function that modifies the PE-dependency graph based on the difference between the current mapping and the new mapping. This idea is similar to the incremental cost function in the Kernighan-Lin algorithm [Kernighan and Lin 1970]. The incremental cost function speeds up the original cost function by more than 100x.

5.2.2. Neighbor Mapping Generatio. We found that the mapping algorithm improves the cost slowly by generating the *random neighbors* (randomly chooses an ODE, and maps the ODE to a different PE). We thus developed two neighbor mapping generators (one for performance, the other for size) to guide the neighbor mapping generation. The performance neighbor generator chooses a random ODE from the bottleneck PE, and moves the ODE to a random PE that connects to the ODE in the ODE-dependency graph. The idea of the performance neighbor is to balance the number of ODEs among all PEs, thus improving overall performance. Note we also considered a convexity constraint (the performance neighbor generator should not increase size, and vice versa) by moving the ODE to a random PE which is connected with the ODE, thus the performance neighbor may also reduce the number of connections.

The size neighbor generator aims at reducing the total number of connections among all PEs. A size neighbor example is shown in Figure 12. The size neighbor generator first chooses a random ODE (2 in the figure), then finds which ODEs are connected to ODE2 and are resident in other PEs (1, 4, 5 in the figure). The generator then calculates the *benefit* of moving each candidate ODE (1, 4, 5) by *#ODE/edge weight,* and moves the ODE with the maximal benefit. The generator prefers to move an ODE with smaller edge weight (defined in the previous section) that has the most chance of reducing a connection. For instance, move ODE1 from PE1 to PE2 will reduce a connection. The #ODE stands for the total number of ODEs in the PE where the candidate ODE resides. The generator prefers to move an ODE from a PE that contains a larger number of

Objective: *min (#cycle * #connection)*

Input: PE number, total number of iterations and ODE dependency graph
Step 1: Generate a random ODE-to-PE mapping and calculate initial cost: *CostI*. Define current cost: *CostC* = *CostI*. Define best cost: *CostB* = *CostI*. Define temperature: *T* = *CostI*
Step 2: Generate a performance or size neighbor by the neighbor mapping generator.
Step 3: Calculate the neighbor mapping's cost: *CostN*. If *CostN* < *CostB*, then *CostB* = *CostN*
Step 4: Define: *D* = *CostN* – *CostC*. If *D* < *0*, accept the neighbor, else use possibility *exp(-D/T)* to accept it.
Step 5: Decrease *T* (*T=CostI / #iteration*) and go back to Step 2 until total number of iterations reached.
Output: *best* ODE-PE mapping found

Fig. 13. ODE-PE mapping algorithm.

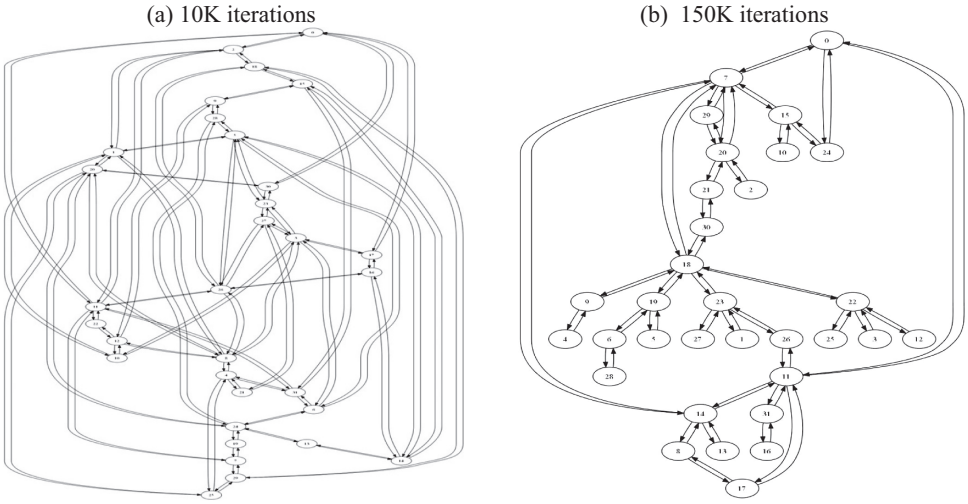


Fig. 14. PE-dependency graphs at different iterations.

ODEs, to balance the number of ODEs in each PE (to improve performance). Thus the size neighbor generator also considers the convexity constraint.

5.2.3. ODE-to-PE Mapping Algorithm. The ODE-to-PE mapping algorithm is illustrated in Figure 13. Note the algorithm inputs include the number of PEs and the number of iterations. Figure 14 shows two PE-dependency graphs at different iterations for mapping 510 ODEs to 32 PEs. Note that the 150K-iteration mapping contains fewer connections compared to the 10K-iteration mapping. The algorithm takes about 5–20 minutes to run on a 3.0 GHz Pentium 4 machine, because we chose a large number of iterations to generate a near-optimal mapping. The ODE-to-PE mapping algorithm's runtime is much less than the FPGA synthesis tool runtime, which usually takes more than 2 hours for large designs. In this work, we run the mapping algorithm multiple times with different numbers of PEs to generate a design space for each physical model.

To test the quality of the ODE-to-PE mapping algorithm, we compared the automatic mapping results to manual mappings. For the four physical models in this article, the

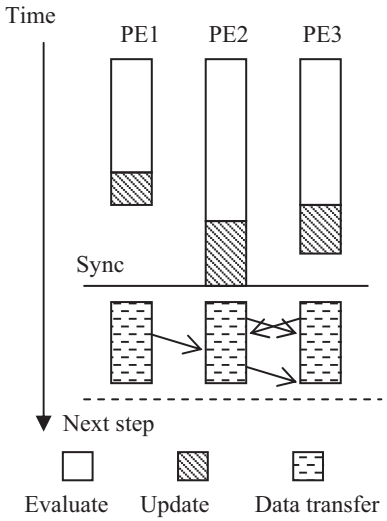


Fig. 15. ODE solving stages.

total number of cycles (performance) of the automatic mapping is usually within 15% of the manual partition. The only exception is the three-dimensional atrial model, where the automatic mapping has a 19% overhead. The total number of connections (size) of the automatic mapping is within 5% of the manual solution for all four models. We notice the automatic mapping algorithm performs worse for complex communication structures, such as the three-dimensional cubic atrial cell model. The mapping algorithm could be further improved for models with a complex connection structure.

5.3. PE Instruction Generation and VHDL Code Generation

5.3.1. PE Instruction Generation. After generating the mapping between ODEs and PEs, the next task of the tool is to generate PE instructions and VHDL files. Figure 15 shows ODE solving stages of three PEs. The ODE solving process has three stages: evaluate, update, and data transfer, as defined in Section 3.2. The evaluate and update processes can be executed independently within each PE, and they may finish at different clock cycles. Evaluate and update instructions are generated by parsing the ODEs mapped to each PE into PE instructions using expression evaluation. All PEs are synchronized at the point when the slowest PE finishes its update task (PE2 in this example). Extra "idle" instructions will be added to the PEs that finished earlier.

The tool statically schedules the data transfer between PEs, after a PE has updated local variables, based on ODE data dependencies. The tool tracks the availability of each PE for each clock cycle and schedules compute operations to load data from data RAM into out register, for example, PE1 executes *compute data RAM[5] + 0* at cycle 1, PE2 could execute *store din[2] -> data RAM[3]* at cycle 3 to store the result produced by PE1 (assume PE1's output is connected to PE2's input port2). Multiple data transfers could happen in parallel as long as they do not conflict with each other. The communication instructions are appended to the update instructions to complete the PE instructions for one solving step. The number of cycles for each PE is determined by the number of instructions.

5.3.2. VHDL Code Generation. To generate the VHDL files, we first created a PE component pool for different PE versions (discussed in detail in Section 6.1). Each PE version has a unique triple of number of input ports, instruction RAM size, and data RAM

size. Each PE also has a generic map for the data and instruction RAM. We developed a script to generate the VHDL code for the PE pool. The script took about 2 hours to build, because different PE versions have the same general architecture and only differ in input ports and instruction RAM/data RAM size. Once created, the PE pool can be reused by different physical models.

For each PE instance, the PE compiler first chooses a suitable PE version from the PE pool and then converts the PE instructions into the control words by a PE assembler. The control words and the model's initial value/parameters will then be injected into the target PE through the generic map.

Once each PE instance has been generated, the next step is to connect them based on the structure of the top-level network. Since we already know the network structure by the automatic mapping algorithm, the PE compiler will convert the edges (in the PE-dependency graph) into VHDL wires. The VHDL code generation flow is fully automated by the PE compiler without the help of any other tools.

6. EXPERIMENTAL RESULTS

This section provides experimental results for automatically synthesizing the four physical models described in Section 3.3 into custom networks of PEs. We first show the size and performance of different single PE versions, and then compare our approach to a high-level synthesis approach. We further compare the performance of our approach to other general-purpose processor platforms including a multicore PC, an ARM processor, DSP processors, and a modern GPU, and we provide a detailed case study with an 11-generation Weibel lung model.

The Weibel 11 (4094 ODEs), Lutchen 4000 (4000 ODEs), and atrial cell 15 (3375 ODEs) models use an RK4 solver with a 0.0001 second step, while the wave 80×80 (6400 ODEs) model has a much smaller step (1/44.1K second, audio sample rate) and thus uses an Euler solver.

Performance numbers are in milliseconds (ms) unless otherwise stated and represent the time for an implementation to execute 1000ms of simulated time. For example, "300" means an implementation executed 1000ms of simulated time in just 300 milliseconds (thus executing faster than real time).

The PE and HLS approaches targeted a Xilinx XC6VLX240T-2 FPGA, having 150,720 LUTs (lookup tables), 768 DSP units (built-in hardcore multipliers), and 416 BRAMs (built-in 32Kb hardcore block RAMs). We used the Xilinx ISE 12.3 tool [Xilinx ISE 2011] for synthesis. We fully synthesized and implemented each example on the target FPGA and report the maximum allowable clock frequency as determined by the Xilinx tools. We note that the work is not limited to a particular FPGA or synthesis tool.

6.1. Size and Frequency of Different Single PE Versions

A PE's data RAM size and instruction RAM size can be configured according to how many ODEs are mapped to the PE. A PE's input mux size is determined by the communication structure. We thus generate different PE versions labeled as: *PE(input no)_D(data RAM size)_I(inst BRAM no)*, for example, *PE3_D64_I2* means the PE has 3 input ports, a 64-word data RAM, and an instruction RAM with 2 BRAMs.

The number of PE input ports can be 1, 3, 7, or 15 in our current design (powers of 2 minus 1, where 1 is the reservation for self-output). The data RAM size can be 32, 64, 128, or 1024 words. The 32-, 64-, and 128-word versions are implemented with LUTs, while the 1024-word version is implemented with block RAM. The LUTs implementation of large data RAMs (larger than 128 words) is inefficient in terms of both performance and size. The instruction RAM is implemented with BRAM, because the number of PE instructions is usually more than 128 in our experiments. The

Table I. Synthesis Results of Different Single PE Versions

	LUTs	DSP	BRAM	freq. (MHz)
PE3_D32_I1	214	1	1	193
PE3_D64_I1	284	1	1	194
PE3_D128_I1	378	1	1	197
PE3_D1024_I1	184	1	2	193

(a) Different data RAM sizes

	LUTs	DSP	BRAM	freq. (MHz)
PE1_D64_I1	264	1	1	195
PE3_D64_I1	284	1	1	194
PE7_D64_I1	312	1	1	195
PE15_D64_I1	376	1	1	198

(b) Different numbers of input ports

instruction RAM contains 1, 2, 3, or 4 BRAMs. We added implementations for 4*4*4 = 64 PE versions into our PE pool to adapt to different physical models' requirements.

We show two sets of PE synthesis results in Table I by altering data RAM size and number of input ports, respectively. Table I(a) shows PEs with 3 input ports, 1 instruction BRAM, and with different data RAM sizes. We notice that a PE's number of LUTs increases rapidly with data RAM size. A *PE3_D128_I1* with a 128-word data RAM uses 70% more LUTs than a *PE3_D32_I1* with a 32-word data RAM. *PE3_D1024_I1* uses a BRAM to implement data RAM, thus using the fewest LUTs. Table I(b) shows how the number of input ports impacts PE size. Note that the number of LUTs increases slowly from 1 input port to 3 and 7 input ports. The 15-input port *PE15_D64_I1* shows a larger LUTs increase, because of the large input mux.

The eight PE versions have similar maximum clock frequencies of about 195 MHz. We compared our PEs with a Xilinx MicroBlaze soft-core processor [MicroBlaze 2011]. The default MicroBlaze version consumes 1,445 LUTs, 3 DSP units, and 8 BRAMs, and has a clock frequency of 123 MHz on the target FPGA. Our PE is thus about 5x smaller than the MicroBlaze, and has a 60% faster clock speed.

6.2. Network of PEs versus High-Level Synthesis

6.2.1. High-Level Synthesis with Custom Communication Architectur. A High-Level Synthesis (HLS) tool usually takes C code of a system/function and converts the C code into synthesizable VHDL code. Modern HLS tools perform extensive algorithm parallelization (e.g., loop unrolling, loop interchange) and create heavily pipelined designs. For comparison purposes, we implemented the four physical models onto an FPGA with a commercial HLS tool[1] with optimization for a custom communication architecture.

Since the four physical models each have a homogenous structure, the corresponding ODEs of each model have the same format. For instance, the wave model's ODEs have the following format: $d(v[i][j])/dt = a * (v[i-1][j] + v[i+1][j] + v[i][j-1] + v[i][j+1]) + b * v[i][j]$, where i, j represents the position of a node in a two-dimentional space. The HLS tool generates a custom datapath that efficiently calculates the ODE, noted as *ODE unit*. We use a *for loop* in the C code to capture this homogenous property, and choose different unrolling factors in the HLS tool to generate designs with different numbers of ODE units, thus generating designs that trade off size and performance.

We noticed that the HLS tool generates a unified memory with block RAM to store all the variables of the ODEs. The unified memory becomes a bottleneck for large designs with multiple ODE units. When using HLS, iteration between writing the input specification and synthesis is common to improve the implementation. We thus rewrote the input specification to include a custom communication architecture that used registers to store the data, referred to as *data registers*. All ODE units can write to corresponding data registers concurrently. Since multiple ODEs are mapped to each ODE unit using resource sharing, each ODE unit needs to access multiple data

[1]The tool name is not included due to the licensing agreement. The tool is commercially available and used by dozens of companies and universities, including the U.S. Dept. of Defense. Reproduction of our experiments using other high-level synthesis tools is highly encouraged.

Table II. Synthesis Results and Performance Comparisons between Network of PEs and High-Level Synthesis (*: implemented on a larger Virtex6 550T-2 FPGA)

Weibel 11	LUTs	DSP	BRAM	Euiqv. LUTs	Cycle/step	freq.(Mhz)	perf(ms)	imp. Time
HLS(10)	57,985	160	80	126,785	3,856	105	367	116
HLS(40)	104,190	640	80	292,990	964	112	86	356
PE(64)	13,421	64	320	144,621	3,900	181	215	69
PE(396)	89,724	396	396	331,284	780	178	44	277
Lutchen 4000								
HLS(20)	48,150	300	60	144,750	2,760	91	303	160
*HLS(80)	187,555	480	80	336,355	690	68	101	570
PE(63)	13,008	63	315	142,158	3,247	182	178	70
PE(397)	89,761	397	397	331,931	549	179	31	225
Wave 80*80								
HLS(20)	92,743	180	80	166,543	1,280	102	558	142
*HLS(80)	167,472	480	160	345,072	320	84	170	438
PE(63)	13,780	63	189	97,570	1,402	166	372	62
PE(380)	93,958	380	380	325,758	269	175	68	260
Atrial cell 15								
HLS(20)	36,881	100	180	126,681	8,250	85	971	216
*HLS(80)	133,558	400	160	291,158	2,025	104	195	402
PE(63)	29,696	63	315	158,846	6,225	140	445	80
PE(219)	87,452	309	309	275,942	1,320	145	91	272

registers. Input muxes are needed for each ODE unit. The custom communication architecture consumes extra FPGA resources for the data registers and input muxes, but avoids the block RAM bottleneck, thus improves performance.

6.2.2. Performance and Size Comparison. Table II shows synthesis results of the HLS and the network of PEs on the target FPGA. Each physical model has two networks of PEs' implementations and two HLS implementations obtained by using different numbers of PEs or ODE units, representing small and large designs. For instance, HLS(10) means that HLS design contains 10 ODE units. The FPGA resource usage of each implementation is shown. To ease comparisons by use of a single number, we also define an *equivalent LUTs* term assuming BRAM and DSP components are both implemented with LUTs. By implementing equivalent DSP multiplier and BRAM components using LUTs, we determined that a DSP unit is equivalent to 250 LUTs, while a BRAM unit is equivalent to 360 LUTs. The table also shows the clock cycles per (solver) step, maximum clock frequency, performance, and total implementation time.

The network of PEs and HLS yield comparable clock cycles per step for their designs. However, the network of PEs' clock frequency is on average 1.8x faster than HLS. In terms of FPGA resource usage, the networks of PEs use on average 0.5x LUTs, 0.57x DSP units, and 3.5x BRAMs compared to their counterpart HLS designs. The networks of PEs use more BRAMs due to PE instructions and data storage. Figure 16 shows the average size (in equivalent LUTs) and performance of the four models using HLS and networks of PEs. Note that for comparably sized implementations, our approach achieves 2.1x performance improvement over HLS.

The custom communication architecture in HLS consumes more than 70% of total LUTs for small designs, and around 40% for large designs. Our approach provides a more resource-efficient communication architecture by extensive ODE-to-PE mapping exploration for reducing the total number of connections among PEs.

We compared the total implementation time of the two approaches. The network of PEs' tool time includes: the PE compiler (10∼20 min), Xilinx ISE (1∼3 hour); the

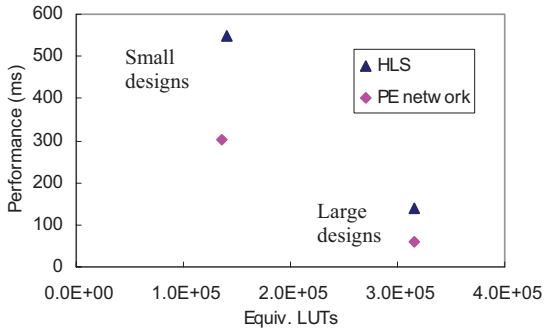


Fig. 16. Size and performance comparison between high-level synthesis and the network of PEs.

Table III. Synthesis Results and Performance Comparisons between the Network of PEs, HLS, and Manual Design for the Weibel 11 Model on the Target FPGA

Weibel 11	LUTs	DSP	BRAM	Equiv. LUTs	Cycle/step	freq.(Mhz)	perf(ms)	imp. time (min)
HLS(40)	104,190	640	80	292,990	964	112	86	356
Manual design	78,597	350	100	202,097	820	192	43	2 Days
PE(396)	89,724	396	396	331,284	780	178	44	277

HLS's tool time includes: HLS tool (5∼25 min), Xilinx ISE (1∼5 hour), and manual communication architecture coding (1∼3 hour). Our approach uses on average 50% less time than HLS.

6.2.3. Compare Network of PEs to a Hand-Tuned Implementation. To further test the quality of our approach, we implemented a hand-tuned version of the Weibel 11 model. We manually designed and optimized the ODE datapath for the Weibel lung model's ODE. We also manually optimized the communication architecture between the ODE datapath and the data registers. To reduce the size of the communication architecture, the input mux may be shared by multiple input ports using time multiplex. To reduce the size of the shared input mux, we also utilize the spatial locality of the ODEs by manually mapping nearby ODEs to an ODE datapath.

Table III shows the synthesis results. The manual optimized design uses around 10% fewer LUTs, DSP, and 4x fewer BRAMs compared to the network of PEs. In terms of equivalent LUTs, the manual design is about 40% smaller due to the large BRAM reduction. The clock frequency and performance of the manual design is comparable to the network of PEs. Compared to the HLS approach, the manual design reduces the size by 33%, and increases the performance by 100% due to a fully customized ODE datapath and communication architecture.

Although our approach used more FPGA resources compared to the manual design, the design flow of our approach is fully automated. The manual design took about 16 hours to design and implement. Note the ALU component in the PE can handle any type of ODE, while the manual design used a customized ODE datapath that can only calculate one type of ODE (Weibel lung ODE in the design). Thus the PE is more readily reusable, and can also be instrumented for future debug and monitor purposes.

6.3. Network of PEs vs. General-Purpose Processors and a GPU

6.3.1. Configuration of Each Approach. We compare the network of PEs with other modern general-purpose processor approaches. These approaches include a modern Intel I7 processor based on the X86-64 instruction set [Intel 64 2011], a TI ARM processor based on a RISC (reduced instruction set computer) instruction set [ARM RISC 2001], a TI

digital signal processor with an optimized architecture for the fast operational needs of digital signal processing, and an NVIDIA graphics processor unit with specialized circuits designed to accelerate video processing. The GPU is programmed with CUDA C [CUDA 2011] (C with NVIDIA extensions and restrictions). The configuration of each approach is listed as follows.

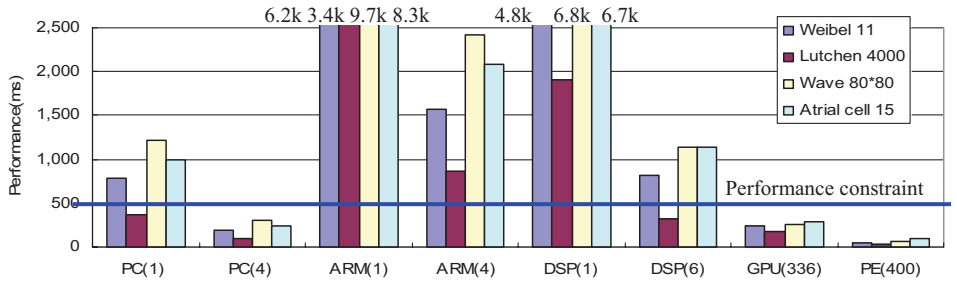
(1) *PC*: C code on a 3.06 GHz Intel I7-950 4-core processor, compiled using Visual C++ 2010 with –O3 flag.
(2) *ARM*: C code on a 1 GHz TI Cortex A9 4-core embedded processor, compiled using TMS470 compiler with –O3 flag.
(3) *DSP*: C code on a 700 MHz TI C6472 6-core digital signal processor, compiled using TI C6000 compiler with –O3 flag.
(4) GPU: CUDA C code on a 763 MHz NVIDIA GTX460 Fermi GPU with 336 CUDA cores, compiled using nvcc with –O3 flag.
(5) PE: A network with 400 PEs on the target Xilinx Virtex6 240T-2 FPGA.

We used a fixed-point C implementation for the four physical models for a fair comparison across all the general processor platforms. Although the general-purpose platforms all support floating point, the fixed-point implementation on FPGA gives nearly identical results. The C code is compiled with the –O3 flag intended to optimize performance. The ARM and DSP results are simulated by TI's CCS cycle-accurate simulator [TI CCS 2011]. For the multicore general-purpose processors, rather than conducting time-consuming multithreaded implementation, we instead measured single-threaded performance first and then calculated an *optimistic* performance bound for multicores simply by dividing the single-threaded performance by the number of cores; in reality, communication overhead will degrade multicore performance.

We also implemented the ODE solvers for the physical models on an Nvidia GTX460 GPU. The target GPU contains 336 cores in total, and cores are distributed onto different GPU blocks. Since each model only contains a few types of ODEs due to homogenous physical structure, we developed ODE kernel functions to calculate each type of ODEs. Different GPU threads are executing the ODE kernel functions against different model variables. To obtain the best performance, we tuned GPU implementation parameters, such as the number of blocks and the number of threads within each block. We also utilized the physical models' spatial locality and load the variables into the shared memory within each block. Utilizing the shared memory will reduce the expensive global memory access, which is commonly used in GPU ODE solving optimization [Ackermann et al. 2009; Hong and Kim 2009].

Unfortunately, communications are necessary between blocks, because a physical model is often globally connected. The only method to do inter-block communication/synchronization is through the GPU's global memory [CUDA Programming Guide 2011]. After each step, all the model variables need to be written back to the global memory. Thus for each step, a new function kernel invocation is necessary.

6.3.2. Performance Comparison. The performance and throughput results for different platforms are shown in Figure 17. To give slacks for future debugging and monitoring purposes, we define a *performance constraint* as 500ms or the pure emulation speed must be at least 2x faster than real time. The single-threaded PC or *PC(1)* just meets the performance constraint of the Lutchen 4000 model, while running slower than the performance constraint for other three models. The single-threaded ARM and DSP run on average 8.4x and 6x slower than the single-threaded PC, respectively, and both fail to meet the performance constraint for all four models. The GPU runs on average 3.4x faster than *PC(1)* due to parallel execution on multiple cores. The GPU runs on average 2.1x faster than the performance constraint.



Model (Perf)	PC(1)	PC(4)	ARM(1)	ARM(4)	DSP(1)	DSP(6)	GPU(336)	PE(400)
Weibel 11	783	196	6,288	1,572	4,876	813	242	44
Lutchen 4000	375	94	3,443	861	1,909	319	171	31
Wave 80*80	1,214	304	9,708	2,427	6,811	1,136	260	69
Atrial cell 15	989	248	8,363	2,091	6,798	1,134	281	91

Model (Throughput)	PC(1)	PC(4)	ARM(1)	ARM(4)	DSP(1)	DSP(6)	GPU(336)	PE(400)
Weibel 11	2.5	10.0	0.3	1.3	0.4	2.4	8.1	44.7
Lutchen 4000	3.2	12.8	0.3	1.4	0.6	3.8	7.0	38.7
Wave 80*80	1.7	6.7	0.2	0.8	0.3	1.8	7.8	29.4
Atrial cell 15	2.0	8.2	0.2	1.0	0.3	1.8	7.2	22.3

Fig. 17. Performance/throughput (GOPS) comparisons between several general-purpose processors, a GPU, and the network of PEs. The numbers in the parentheses indicate the number of threads. Note that thee ARM(1) and the DSP(1) numbers extend off the chart top.

The network of PEs runs on average 10x faster than the performance constraint and 15x faster than the single-threaded PC for the four models. Compared to the performance upper bound of multicore processors, our approach is still 3.6x faster than the 4-core PC, 30x faster than the 4-core ARM, and 14x faster than the 6-core DSP. Although the optimal *PC(4)* runs every model faster than the performance constraint, the implementation of a multithreaded ODE solver is nontrivial. For reference, we also reported the throughput of each platform in terms of giga-operations per second. The network of PEs has an average 34 GOPS, which is approximately 4x faster than GPU and PC(4).

Comparing to the GPU, the network of PEs is on average 4.4x faster. Although the GPU has a higher clock frequency (763 MHz), the general-purpose memory architecture in the GPU may not match the physical model's requirements. An ODE kernel function invocation is necessary at each step. The overhead of frequent ODE kernel invocation consumed 40%–70% of total GPU execution time.

We also compared our GPU results with other GPU ODE solvers for physical systems [Ackermann et al. 2009; Amorim et al. 2010; Mosegaard and Sørensen 2005]. Those papers use much larger physical models of around one million ODEs, and the execution speed is usually more than 100x slower than real time. The larger model executes each time step longer, thus hiding the penalty of the frequent kernel invocation, and gains larger speedups over CPU than our results. However, this article emphasizes real-time emulation. We note the GPU results may be further optimized, and we strongly encourage other researchers to strive to create faster implementations of our physical models on GPUs.

6.3.3. Total Implementation Time. To give a fair comparison to different approaches, we spent comparable amount of time for each approach. The C implementation took around

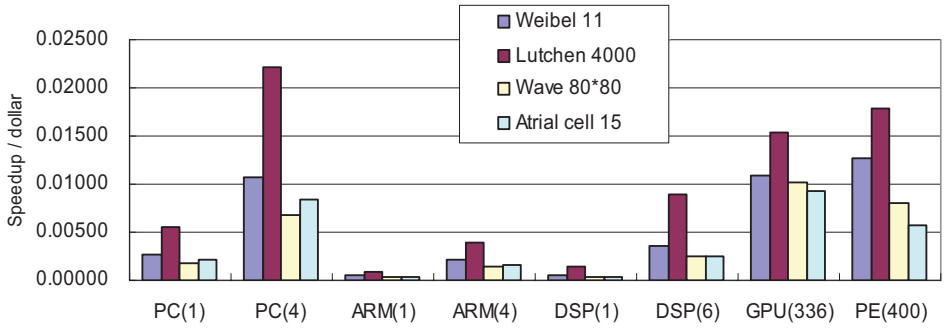


Fig. 18. Normalized speedup per dollar of different approaches, using optimistic (no communication overhead) estimates for all multicore devices except the PE approach.

2–3 hours to optimize the code for each model. The GPU approach took about 3–4 hours to optimize each model. The optimization includes the GPU parameter tuning (number of blocks, number of threads per block), choosing different mapping of the ODEs to the GPU blocks, and utilizing shared memory in each block. Our approach took 20 minutes to generate a custom network, and took another 2–3 hours to synthesize. Although all three approaches took comparable amount of time to implement, the network of PEs' design flow is fully automated.

6.3.4. Cost Comparison. Although comparing costs of the various compute platforms is difficult due to diverse pricing policies and rapidly changing costs, we nevertheless include some approximate cost comparisons, in particular to acknowledge that our FPGA-based approach is currently costlier, though within reason. We consider minimal required components for each approach; as such components would contain a complete system that could be used for purposes of physical modeling in scenarios suggested in this article. The approximate cost (as of January 2012, obtained via Web-based distributor pricing) of each board is as follows:

(1) CPU (I7-950 + Intel X58 board): $480
(2) ARM (Cortex 9A 4-core board): $300
(3) DSP (TI C6472 board): $350
(4) GPU (GTX460 + I3-540 + H55 board): $380
(5) FPGA (Xilinx Virtex6 240T-2 board): $1800

To compare these costs fairly, we consider the term: (speedup over real time)/(cost), written as speedup/dollar in Figure 18. One can see that, although the FPGA board itself is currently more expensive, the speedup obtained by that board is greater, leading to speedup/dollar that is competitive with the GPU and PC(4). However, we note again that the general-purpose processor and the GPU speedups are optimistic; communication overheads will degrade these speedups. In other words, five GTX460 GPUs will probably perform worse than our PE solution due to the communication overhead, though the two approaches have similar total cost. Furthermore, an FPGA may better support custom interfaces to the real physical world [Miller et al. 2011]. Also, FPGA cost trends may continue to improve the FPGA speedup/dollar relative to the other approaches, although the FPGA trend versus PC and GPU trends is hard to predict.

6.4. Case Study

This section highlights a case study of synthesizing an 11-generation Weibel lung model to a custom network of PEs. We also compare the computation accuracy of the simulation results between the network of PEs and a desktop floating-point implementation.

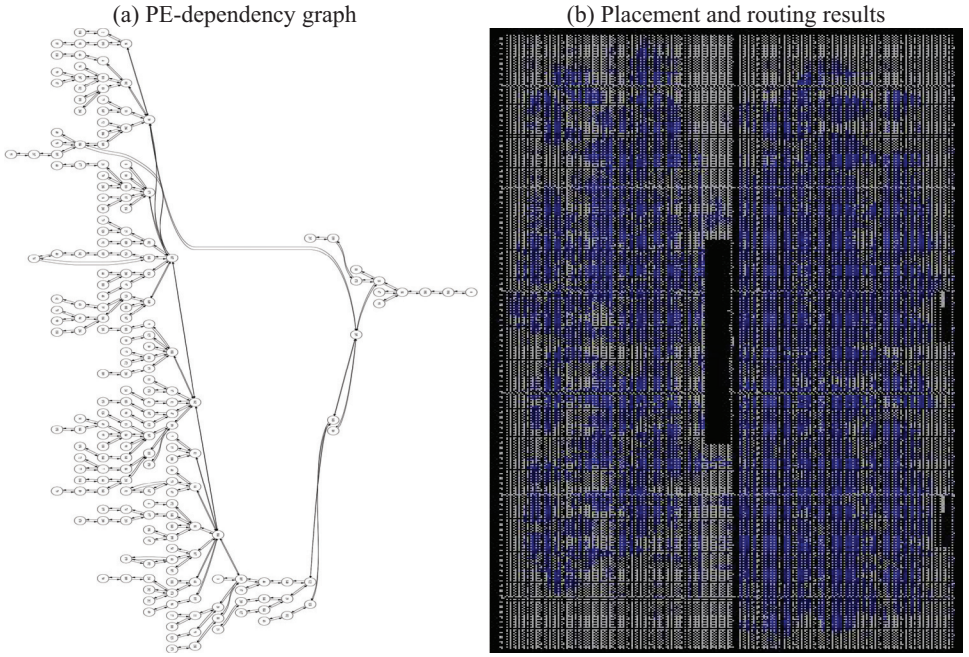


Fig. 19. Weibel 11 200-PE network on a Xilinx Virtex6 240T-2 FPGA.

The Weibel 11 model contains 2047 branches and each branch is captured with two ODEs (volume and flow), thus Weibel 11 contains 4094 ODEs. We input the specification file of Weibel 11 to the PE compiler and specify to use 200 PEs in the design (around 20 ODEs per PE). The PE compiler performed automatic ODE-to-PE mapping and generated the PE-dependency graph in Figure 19(a). Note that the custom network of PEs has a similar tree structure compared to the physical structure of the Weibel lung. This tree-like communication structure is quite scalable on FPGAs without wire congestion problems.

We implemented the VHDL code generated by the PE compiler onto the target FPGA. The final placement and routing results are shown in Figure 19(b), with the darker regions (blue if viewed in color) representing the regions used for implementation. Note the circuits of the network of PEs are almost evenly distributed on the target FPGA, representing the local connectivity of the physical model. The 200 PE implementation of Weibel 11 uses 57,586 of the available 150,720 LUTs (38%), 372 of the 416 BRAMs (89%), and 186 of the 768 DSPs (24%) of the target FPGA. The average size of each PE is 310 LUTs, 1 DSP, and 2 BRAMs. The maximal clock frequency is 164 MHz. Each PE contains 1590 instructions for each step, thus the network simulates 1 second of real time in 97ms, or 10.3x faster than real time.

Earlier we stated that all models used fixed-point computation rather than floating point. Fast floating-point computation is readily available on some general-purpose processors like PCs, but less so on FPGAs on some other general-purpose processors. To validate that the conversion to fixed point did not hurt the model accuracy, we compared the simulation results of the Weibel 11 lung model between our approach and a desktop PC floating-point implementation. The volumes of the first branch under a square wave and a sine wave pressure are illustrated in Figure 20. The total simulated time is 10 seconds. The fixed-point PE implementation has almost identical results compared to the desktop implementation. From the figures on the right, we notice

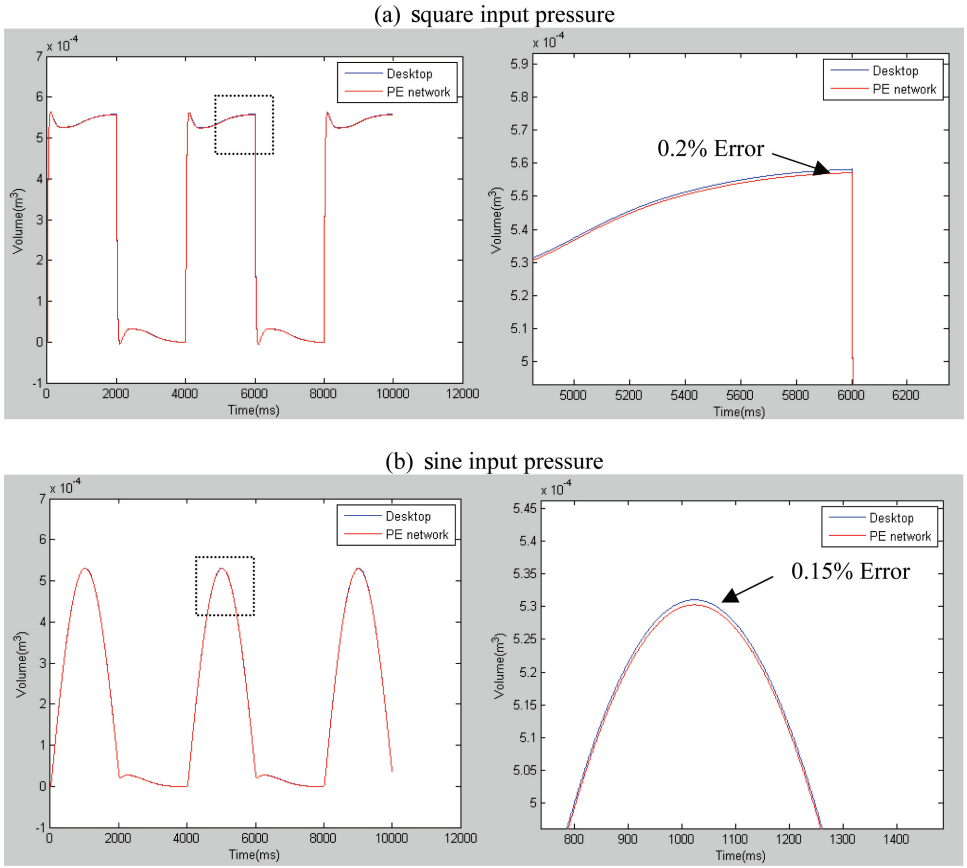


Fig. 20. First branch volume.

that the network of PEs' results has errors within 0.2% compared to the desktop implementation.

7. CONCLUSIONS

We described a general Processing Element (PE) for efficient solving of physical model ordinary differential equations, introduced a custom network of general PEs for parallelized solution of large models, and described a fully automated approach for implementing physical model ODEs on modern FPGAs. Comparing to a commercial high-level synthesis tool on several models, the networks of PEs were on average 2.1x faster with 2x fewer LUTs, 2x fewer DSPs, but 3.5x more BRAMs. Our approach was also 15x faster than a single-core Intel I7 processor and 4.4x faster than an NVIDIA GTX460 GPU. The speedups are obtained due to the excellent match between the local computation/communication structure of most physical models and the local computation/communication capabilities of FPGAs, avoiding the common routing or memory bottlenecks for many applications mapped to FPGAs. These speedups are from the first version of our physical model to FPGA approach; we anticipate that continued improvements will improve speedups versus the other more mature approaches. Currently, the network of PEs can handle around 5000 ODEs on the target FPGA, limited by the block RAM resource. Our future work includes reducing PE instruction and data storage overhead, such that our approach can handle larger models. To further

accelerate PE execution speed, a custom PE that specifically solves certain ODEs can be investigated. We also plan to add runtime control and debug capabilities into the network of PEs, so we can integrate the FPGA physical models into the cyber-physical system testing framework described in Miller et al. [2011].

REFERENCES

ACKERMANN, J., BAECHER, P., FRANZEL, T., GOESELE, M., AND HAMACHER, K. 2009. Massively-parallel simulation of biochemical systems. In *Proceedings of the Massively Parallel Computational Biology on GPUs Conference.* Jahrestagung der Gesellschaft fÃOEr Informatik e.V.

ADVANCED MICRO DEVICES (AMD). 2011. AMD opteron. http://www.amd.com/usen/Processors/ProductInformation/0,,30_118_8825,00.html.

AGARWAL, A., SITES, R., AND HORWITZ, M. 1986. ATUM a new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*.

AMORIM, R. M., ROCHA, B. M., CAMPOS, F. O., AND DOS SANTOS, R. W. 2010. Automatic code generation for solvers of cardiac cellular membrane dynamics in gpus. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'10).* 2666–2669.

ANDREEV, K. AND RACKE, H. 2006. Balanced graph partitioning. *Theor. Comput. Syst. 39*, 6, 929–939.

ARM RISC. 2001. http://www.arm.com/products/processors/technologies/instruction-set-architectures.php.

ASPX TI CCS. 2011. http://focus.ti.com/docs/toolsw/folders/print/ccstudio.html.

ATKINSON, K. 1993. *Elementary. Numerical Analysis* 2nd Ed. John Wiley and Sons, New York.

ATI GRAPHICS CARDS. 2011. http://ati.amd.com/support/driver.html.

BARBINI, P., BRIGHENTI, C., CEVENINI, G., AND GNUDI, G. 2005. A dynamic morphometric model of the normal lung for studying expiratory flow limitation in mechanical ventilation. *Ann. Biomed. Engin 33*, 4, 518–530.

BUTCHER, J. C. 2003. *Numerical Methods for Ordinary Differential Equations*. Wiley.

BUYUKKURT, B. A., GUO, Z., AND NAJJAR, W. 2006. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. In *Proceedings of the International Workshop on Applied Reconfigurable Computing.*

CELLML. 2011. http://www.cellml.org.

CELOXICA. 2011. http://www.celoxica.com/.

CHEN, H., SUN, S., ALIPRANTIS, D., AND ZAMBRENA, J. 2009. Dynamic simulation of electric machines on FPGA boards. In *Proceedings of the International Electric Machines and Drives Conference.*

CONG, J., FAN, Y., HAN, G., JIANG, W., AND ZHANG, Z. 2008. Platform-based behavior-level and system-level synthesis. In *Proceedings of the IEEE International SOC Conference*. 199–202.

CRAY. 2011. http://www.cray.com/Home.aspx.

CUDA 2011. http://developer.nvidia.com/cuda-downloads.

CUDA PROGRAMMING GUIDE. 2011. http://developer.download.nvidia.com/compute/cuda/4_0-/toolkit/docs/CUDA_C_Programming_Guide.pdf.

DINIZ, P., HALL PARK, M., PARK, J., SO, B., AND ZIEGLER, H. 2001. Bridging the gap between compilation and synthesis in the defacto system. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing Synthesis (LCPC'01).*

GHOLKAR, A., LSAACS, A., AND HEMENDRA, A. 2002. Hardware-in-loop simulator for mini aerial vehicle. In *Proceedings of the 6th Real-Time Linux Workshop.*

GOKHALE, M. B., STONE, J. M., ARNOLD, J., AND LALINOWSKI, M. 2000. Stream-oriented FPGA computing in the streams-C high level language. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'00).*

HEART SIMULATOR. 2011. http://www.columbia.edu/itc/hs/medical/heartsim/.

HONG, S. AND KIM, H. 2009. An analytical model for GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the International Symposium on Computer Architecture*.

HUANG, C., VAHID, F., AND GIVARGIS, T. A. 2011. Custom FPGA processor for physical model ordinary differential equation solving. *IEEE Embedd. Syst. Lett. 3*, 4, 113–116.

HUCKA, M., FINNEY, A., BORNSTEIN, B., KEATING, S., SHAPIRO, B. MATTHEWS, J. KOVITZ, B., SCHILSTRA, M., FUNAHASHI, A., DOYLE, J., AND KITANO, H. 2004. Evolving a lingua franca and associated software infrastructure for computational systems biology: The systems biology markup language (SBML) project. *IEEE Syst. Biol. 1*, 1, 41–53.

IBM BLUE GENE. 2011. Supercomputer. http://domino.research.ibm.com/comm/research_projects.nsf/pages/bluegene.index.html.

INTEL 64. 2011. http://www.intel.com/technology/intel64/index.htm.

INTEL CORPORATION. 2011. Multicore technology. http://www.intel.com/multi-core/.

IWANAGA, N., SHIBATA, Y., YOSHIMI, M., OSANA, Y., IWAOKA, Y., ET AL. 2005. Efficient scheduling of rate law functions for ODE-basedmultimodel biochemical simulation on an FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 666–669.

JSIM. 2011. http://nsr.bioeng.washington.edu/jsim/.

KERNIGHAN, B. W. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J. 49*, 291–307.

KUM, K., KANG, J., AND SUNG, W. 2000. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Analog Digital Signal Process. 47*, 9, 840–848.

LEE, E. A. 2008. Cyber physical systems: Design challenges. Tech. rep. UCB/EECS-2008-8, University of California, EECS Department.

LIN, C. L., TAWHAI, M. H., MCLENNAN, G., AND HOFFMAN, E. A. 2009. Multiscale simulation of gas flow in subject-Specific models of the human lung. *IEEE Engin Med. Biol. 28*, 3, 25–33.

LIONETTI, F. 2010. http://cseweb.ucsd.edu/groups/hpcl/scg/papers/2010/lionetti_ms_thesis.pdf.

LUTCHEN, F. P., PRIMIANO, J. R., AND SAIDEL, G. M. 1982. A nonlinear model combining pulmonary mechanics and gas concentration dynamics. *IEEE Trans. Biomed. Engin. 29*, 629–641.

MATHEMATICA. 2011. http://www.wolfram.com/.

MATHWORKS. 2011. Matlab and simulink. http://www.mathworks.com/.

MEDGADGET. 2008. Supercomputer creates most advanced heart model. *Int. J. Emerging Med. Technol.* Jan. 2008.

MCFARLAND, M. C., PARKER, A. C., AND CAMPOSANO, R. 1990. The high level synthesis of digital systems. *Proc IEEE 78*, 301–318.

MICROBLAZE. 2011. http://www.xilinx.com/tools/microblaze.htm.

MILLER, B., GIVARGIS, T., AND VAHID, F. 2011. Application-specific codesign platform generation for digital mockups in cyber-physical systems. In *Proceedings of the IEEE Electronic System Level Synthesis Conference (ESLsyn'11)*.

MOSEGAARD, J. AND SØRENSEN, T. S. 2005. Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the GPU. In *Proceedings of the Eurographics Virtual Environments Workshop*. 105–110.

MOTUK, E., WOODS, R., AND BILBAO, S. 2005. Implementation of finite difference schemes for the wave equation on FPGA. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'05)*.

NAJJAR, W., BOHM, W., DRAPER, B., HAMMES, J., RINKER, R., BEVERIDGE, R., CHAWATHE, M., AND ROSS, C. 2003. From algorithms to hardware - A high-level language abstraction for reconfigurable computing. *Comput. 36*, 8.

NATIONAL INSTRUMENTS. 2011. LabView FPGA module. http://www.ni.com/fpga/.

NSR PHYSIOME PROJECT. 2011. Mathematical markup language. http://nsr.bioeng.washington.edu/jsim/docs/MML_Intro.html.

NVIDIA CORPORATION. 2011. http://www.nvidia.com/object/gpu.html.

OSANA, Y., FUKUSHIMA, T., AND AMANO, H. 2004. ReCSiP: A reconfigurable cell simulation platform: Accelerating biological applications with FPGA. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'04)*.

PIMENTEL, J. AND TIRAT-GEFEN, Y. 2006. Hardware acceleration for real time simulation of physiological systems. In *Proceedings of the 28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS'06)*. 218–223.

RESHADI, M., GORJIARA, B., AND GAJSKI, D. 2005. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *Proceedings of the International Conference on Computer Design*.

SALWINSKI, L. AND EISENBERG, D. 2004. *Silico Simulation of Biological Network Dynamics*. Nature Publishing Group, 1017–1019.

SIMULINK. 2001. http://www.mathworks.com/products/simulink/.

SPARK PROJECT. 2005. http://mesl.ucsd.edu/spark/.

SYNPHONYC. 2011. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyCCompiler.

VHDL. 2011. http://www.vhdl.org.

WEIBEL, E. R. 1963. *Morphometry of the Human Lung*. Springer.

XILINX ISE. 2011. http://www.xilinx.com/support/documentation/dt_ise12-4.htm.

YOSHIMI, M., OSANA, Y., FUKUSHIMA, T., AND AMANO, H. 2004. Stochastic simulation for biochemical reactions on FPGA. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application (FPL'04).* 05–114.

ZHANG, H., HOLDEN, A. V., AND BOYETT, M. R. 2001. Gradient model versus mosaic model of the sinoatrial node. *Circulat. 103*, 584–588

Received February 2011; revised February 2012; accepted July 2012