Synthesis of Networks of Custom Processing Elements for Real-Time Physical System Emulation

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

Emulating a physical system in real-time or faster has numerous applications in cyber-physical system design and deployment. For example, testing of a cyber-device's software (e.g., a medical ventilator) can be done via interaction with a real-time digital emulation of the target physical system (e.g., a human's respiratory system). Physical system emulation typically involves iteratively solving thousands of ordinary differential equations (ODEs) that model the physical system. We describe an approach that creates custom processing elements (PEs) specialized to the ODEs of a particular model while maintaining some programmability, targeting implementation on field-programmable gate arrays (FPGAs). We detail the PE micro-architecture and accompanying automated compilation and synthesis techniques. Furthermore, we describe our efforts to use a high-level synthesis approach that incorporates regularity extraction techniques as an alternative FPGA-based solution, and also describe an approach using graphics processing units (GPUs). We perform experiments with five models: a Weibel lung model, a Lutchen lung model, an atrial heart model, a neuron model, and a wave model; each model consists of several thousand ODEs and targets a Xilinx Virtex 6 FPGA. Results of the experiments show that the custom PE approach achieves 4X-9X speedups (average 6.7X) versus our previous general ODE-solver PE approach, and 7X-10X speedups (average 8.7X) versus high-level synthesis, while using approximately the same or fewer FPGA resources. Furthermore, the approach achieves speedups of 18X-32X (average 26X) versus an Nvidia GTX 460 GPU, and average speedups of more than 100X compared to a six-core TI DSP processor or a four-core ARM processor, and 24X versus an Intel I7 quad core processor running at 3.06 GHz. While an FPGA implementation costs about 3X-5X more than the non-FPGA approaches, a speedup/dollar analysis shows 10X improvement versus the next best approach, with the trend of decreasing FPGA costs improving speedup/dollar in the future.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: Custom processor, field-programmable gate array (FPGA), ordinary differential equation (ODE) solving, high-level synthesis, physical models, real-time emulation

ACM Reference Format:
Huang, C., Miller, B., Vahid, F., and Givargis, T. 2013. Synthesis of networks of custom processing elements for real-time physical system emulation. ACM Trans. Des. Autom. Electron. Syst. 18, 2, Article 21 (March 2013), 21 pages.
DOI: http://dx.doi.org/10.1145/2442087.2442092

This work was supported in part by the National Science Foundation (CNS1016792, CPS1136146), the Semiconductor Research Corporation (GRC 2143.001), and a U.S. Department of Education GAANN fellowship.
Author's address: C. Huang; email: chuang@cs.ucr.edu.

DOI: http://dx.doi.org/10.1145/2442087.2442092

1. INTRODUCTION

In cyber-physical systems [Lee 2008], computers or other computing devices interact with physical systems. Fast accurate digital models that emulate physical systems can assist in designing cyber-physical systems. For example, a digital model of human lungs can be interfaced with a ventilator for testing during ventilator development (in real-time or faster), or for training clinicians on ventilator use across a variety of lung conditions. Furthermore, the model's parameters can be modified to reflect varied lung pathologies more easily than typical lung emulation approaches (e.g., using physical balloons). Such use cases motivate our work.

Accurate physical system emulation requires complex physical models. A physical model is often captured as a system of ordinary differential equations (ODEs) having thousands of state variables (or dimensions). The ODEs are often solved by iterative ODE solvers, which may compute each equation in the system 1000 times per second, or even faster if greater accuracy is required. Modern desktop processors may not be able to emulate complex physical models in real-time, because the mostly-serial program execution on a processor does not match the massively-parallel nature of a physical model.

To accelerate physical model emulation, we previously introduced an approach using a network of processing elements (PEs) optimized for ODE solving [Huang et al. 2012]. We first created a lightweight programmable ODE-solver PE for a field-programmable gate array (FPGA), where the PE was optimized for solving tens of ODEs [Huang et al. 2011]. We then developed a synthesis tool to automatically map thousands of ODEs onto a statically-scheduled custom network of tens or hundreds of PEs, each PE solving a subset of ODEs. Because physical systems are typically comprised of numerous physical objects communicating locally with neighboring objects, physical systems represent an excellent match for FPGAs, which excel at performing numerous parallel computations with local communication (avoiding the well-known centralized-data communication bottleneck in FPGAs). For example, Figure 1 shows how an atrial cell model, which simulates cardiac action potentials by propagating signals across neighboring cells, can be mapped to a network of custom PEs in which a cell or groups of neighboring cells each occupy a PE. The atrial cell model executes about 100x faster on a network of custom PEs on a mid-range FPGA than on a modern desktop processor.

In our earlier work, the PEs were optimized for solving general ODEs, and not for any specific ODEs. In this article, we examine the benefits of synthesizing PEs that are customized to the specific ODEs mapped onto each PE. The custom PEs are still programmable, thus enabling for example insertion of instructions for profiling or debugging. Many large physical systems are homogenous systems with only a few (<5) types of ODEs replicated many times, (e.g., atrial cell model of Figure 1). Thus, this article focuses on using one custom PE that is replicated in a network; future work will examine using heterogeneous PEs.

This article is organized as follows. Section 2 reviews related work. Section 3 describes how to model a physical system with ODEs, and how to map the ODEs to a network of PEs. Section 4 introduces custom ODE-solving PEs and their advantages over the previous general ODE-solving PE. Section 5 describes a custom-PE compiler. Section 6 discusses a related approach involving high-level synthesis with regularity extraction. Section 7 describes our efforts to map the ODE solving problem to a graphics processing unit (GPU). Section 8 provides experimental results comparing the custom-PE approach with other compute platforms and approaches. Section 9 concludes.

2. RELATED WORK

Physical system modeling and simulation have been studied extensively in different fields, such as physiology, chemistry, and biology. Languages have been introduced for

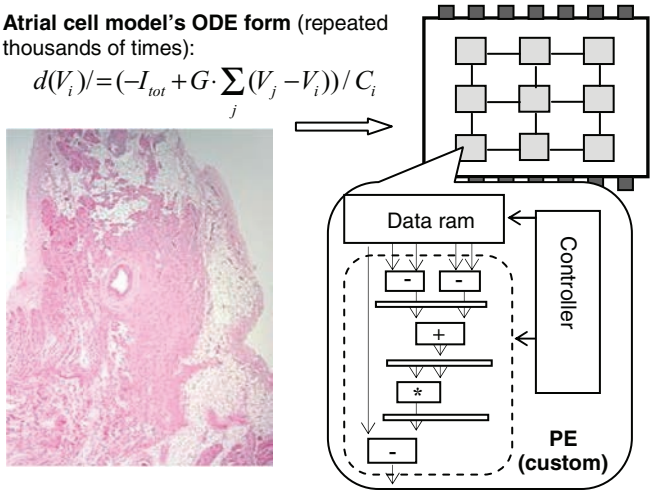


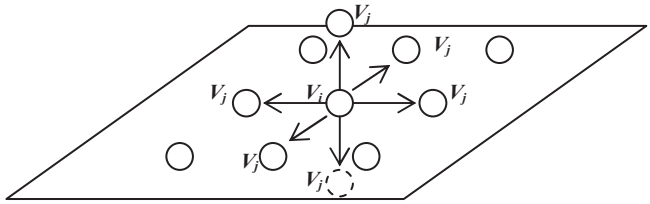
Fig. 1. Synthesizing an atrial cell model into a network of custom PEs on an FPGA.

modeling physical systems, such as MML [NSR 2011], SBML [Hucha et al. 2004], and CellML [CellML 2011]. Tools have been built for simulating physical models, such as Matlab [Mathworks 2011], JSim [Jsim 2011], LabView [National Instruments 2011]. These tools usually aim to produce accurate results rather than real-time emulation.

Many efforts seek to speed up emulation using multicore processors and GPUs [ATI 2011; Nvidia 2011]. For instance, a 768-core SGI machine executed a 2 billion equation heart model, simulating 0.4 ms in 2 hours [MedGadget 2008]. An Nvidia GTX 295 GPU was used to execute a Flaim heart model 30x faster than OpenMP running on an Intel I7 quad core processor at 2.93 GHz. Executing one heartbeat (300 ms) required 7.7 minutes [Lionetti 2010]. The main bottleneck of multicore processors and GPUs is the centralized memory architecture, which commonly does not match the local-neighbor communication pattern of a physical model.

Many case studies using FPGAs to speed up physical model emulation have been conducted. For example, Yoshimi et al. [2004] obtained 100x speedups of a fine-grained biological emulation compared to a single-core processor, and showed why multicore processors were not suitable. Pimentel et al. [2006] proposed an FPGA solution to emulate a heart-lung system model in real-time, while a PC required 1.5 hours to simulate 60 seconds of the same model. Their FPGA performance was estimated by a theoretical optimal formula, rather than via an implementation. 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 may not scale to larger models. Those previous efforts mostly used manual design approaches to implement the models on FPGAs.

A common automated design approach for implementing applications on FPGAs uses high-level synthesis (HLS). Recently, tools have evolved to synthesize from C code rather than hardware description language (HDL) code, such as Streams-C [Gokhale et al. 2000], SPARK [SPARK 2005], ROCCC [Villarreal et al. 2010], Celoxica [Celoxica 2011], SynphonyC [SynphonyC 2011], Impulse C [Impulse C 2011], and AutoESL [AutoESL 2011]. We compare our approach to a commercial C HLS tool to generate ODE datapaths, and due to the regularity of physical models, we incorporate the idea of regularity extraction [Rao and Kurdahi 1993], as will be described in Section 6.



$$d(V_i)/dt = (-I_{tot} + G.\sum_j (V_j - V_i))/C_i \quad (1)$$

Fig. 2. A 3-dimensional atrial cell model for modeling a heart, each circle representing a cell.

We earlier proposed an approach for synthesizing a physical model onto a network of general ODE-solving PEs on an FPGA [Huang et al. 2011]. The work in this article aggressively customizes the PE component to specific ODEs, utilizing a newly-developed PE compiler to support programming of a network of custom PEs, leading to performance and size improvements while still including some PE programmability.

3. PHYSICAL SYSTEM MODELING AND A NETWORK OF PROCESSING ELEMENTS

This section briefly describes how to model a physical system with ODEs using an atrial cell model as a driver example, and reviews our earlier network of general PEs approach.

3.1. Modeling Physical Systems Using ODEs

Physical system models are often captured as ordinary differential equations (ODEs). Figure 2 shows a 3-dimensional atrial cell model intended to model a heart for interaction with a pacemaker [Zhang et al. 2001]. Vi is the membrane potential of cell i, *Itot* is the total ionic current in a cell, G is the coupling conductance, and C is the cell capacitance. Vj stands for the neighboring cells (6 neighbors in the figure). The derivative of Vi with respect to time is determined by the sum of the membrane potential differences between the cell and neighboring cells (Vj).

The ODEs of a physical system can be written in the form: $d(X)/dt = Fun(X)$, where X is a vector of state variables. In the atrial cell model, X is the membrane potentials for all cells in the model (note *Itot*, G, and C are constant parameters of the system, but not state variables). The scale of a physical system is determined by the number of state variables (or dimensions). For this atrial cell model, the scale is determined by the number of cells in the system (e.g., 3,375 cells in our experiments).

To calculate the membrane potential of a cell at a given time, the ODEs can be solved using iterative solvers such as Euler [Atkinson 1993] or Runge-Kutta [Butcher 2003]. Starting from time 0, iterative solvers move forward in time by a given *time step*, such as by 1 ms. At each time step, the Euler methods performs two major tasks.

(1) *Evaluate*. Calculate each variable's derivative value using the equation, for instance, equation (1) in Figure 2.
(2) *Update*. Estimate the variable values for the next time step using current values and the derivatives calculated before, for instance, $Vi = Vi + d(Vi)/dt * h$, where h is the time step.

3.2. Properties of Physical System ODEs and a Network of PEs

General-processor-based ODE solvers often store all the variables of a physical system in a central memory, and calculate the ODEs sequentially. Thus the total simulation time increases linearly with the dimensions of the ODEs.

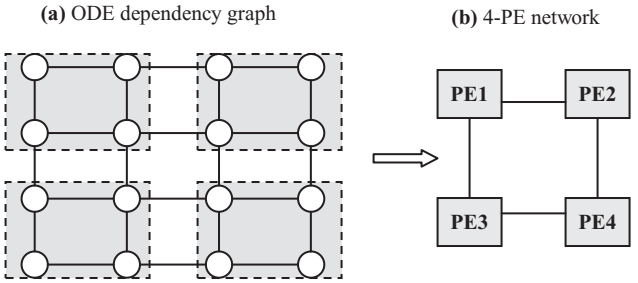


Fig. 3. Mapping a 4 × 4 2-dimensional atrial cell network to a 4-PE network.

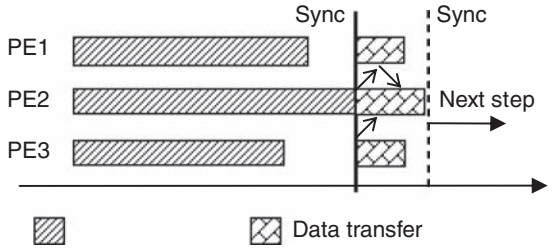


Fig. 4. ODE solving process in a network of 3 PEs

However, the ODEs of a physical system usually have spatial locality similar to their system's physical counterparts. For instance, the membrane potential of an atrial cell is determined only by itself and the cell's neighboring cells.

We thus define an *ODE dependency graph* to capture the data-dependencies among the state variables in a physical system. Each node in the ODE dependency graph represents one variable or a set of variables, while each edge represents the data dependency between two nodes. For instance, Figure 3(a) shows the ODE dependency graph of a 4 × 4 2-dimensional atrial cell model, where each node represents the membrane potential of each cell. Note we use undirected edges in the figure for simplicity. The actually implementation used directed edges.

To increase the speed of the physical system emulation, we parallelize the ODE solving process by mapping the ODEs to a network of PEs. For example, Figure 3 maps the 4 × 4 atrial cell model to a 4-PE network. The variables in each of the dashed rectangles are mapped to one PE. The interconnection among the PEs is determined by the variable-to-PE mapping and the ODE dependency graph. Instead of using a centralized memory, the variables and their relevant parameters are stored in the local memory of the PE. This distributed data model eliminates the bottleneck of a central memory.

At each time step, each PE performs an "evaluate" task, and an "update" task for the variables mapped to the PE, called the *resident variables*. To perform these two tasks, the PE may need the variables residing in other PEs (referred to as dependent variables). Instead of accessing the dependent variables directly from other PEs (which may introduce delay and inconsistency), each PE keeps local copies of the dependent variables. We introduce a "data transfer" task and synchronizations to ensure the dependent variables have the same value as the original copies.

The synchronization process of a 3-PE network is illustrated in Figure 4. The "evaluate" and "update" tasks can be done independently in each PE. The processing elements are synchronized at the point when all PEs have finished updating their resident variables. Then the updated variables are propagated to the PEs that depend on those

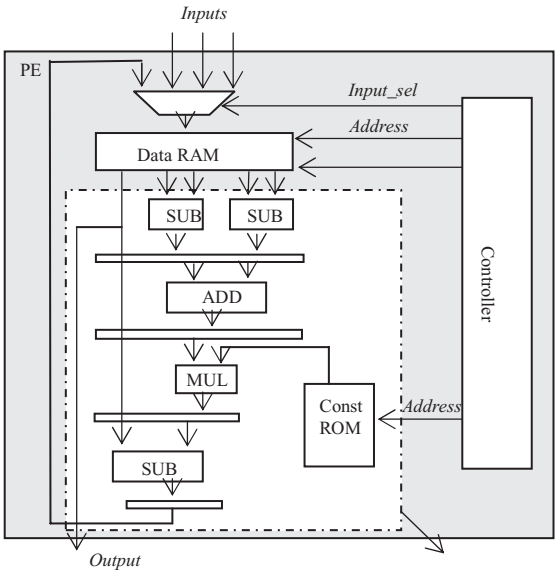


Fig. 5. Custom PE micro-architecture.

variables during the data transfer phase (store the dependent variables into local data ram). All PEs are synchronized again for the next step. Thus at the beginning of each time step, each PE has the latest values of all variables for the "execute" and "update" tasks. This synchronization process ensures the results of the network of PEs are identical to a desktop processor version (with centralized data storage).

4. CUSTOM PES

4.1. Custom PE Architecture

We earlier proposed a general processing element (PE) for solving different types of ODEs [Huang et al. 2011]. The general PE contains a general purpose ALU (arithmetic logic unit). Each ODE is parsed into basic operations (such as addition, subtraction, store, etc) for the ALU. Although the general PE has some optimizations intended for ODE solving, such as eliminating instruction decoding logic, absence of a jump operation, etc., the general purpose ALU doesn't capture the unique structure of a specific ODE. Further customization of the ALU can lead to better performance and less FPGA resource requirements.

In this work, we note that many physical models have homogenous structures. For instance, the atrial cell model has a 3-dimensional cubic structure, and a Weibel lung [Weibel 1963] has a binary tree structure, with each node in the structure performing the same behavior. The homogenous property of a physical model leads to the same structure of all its ODEs. For instance, the ODE for each atrial cell is: $d(V_i)/dt = (-I_{tot} + G \cdot \sum_j (V_j - V_i))/C_i$ where Vj means the membrane potential of its neighboring cells. Since every cell uses this equation to calculate the derivative of its membrane potential, we can build a custom datapath to calculate this equation. Figure 5 shows the micro-architecture of a custom PE with the custom ODE datapath for the given equation. The custom PE has a similar architecture compared to the general PE, except the ALU component is replaced with a custom ODE datapath. To support the needs of the custom ODE datapath, the data-ram of the custom PE usually contains more output ports than a general PE. Since the data-ram often contains 32 or 64 words

(depending on how many variables are mapped to a PE), the data-ram is built with FPGA LUTs (lookup tables).

The custom PE uses a fixed-point computation. Currently, we manually convert floating-point numbers into fixed-point using the method described by Kum et al. [2000]. The one time manual conversion to fixed-point for a model usually takes less than one hour.

We use a 32-bit fixed-point implementation for all models in this work. We compared the results of the 32-bit fixed-point implementation to a floating-point implementation in Matlab. The fixed-point results are nearly identical to the floating-point results, with less than 0.5% relative error. The number of bits of the fixed-point implementation is configurable for different models, if higher accuracy is required.

4.2. Abstract PE Tasks

In each time step, a PE needs to perform three tasks: evaluate, update, and data transfer, as discussed in Section 4.2. To map those tasks to the instructions of a custom PE, we define three abstract PE instructions: compute, store, and output.

The compute instruction combines the "evaluate and update" task of a certain variable. For the previous atrial cell example, the "*compute Vi*" task performs:

$$V_i(t+1) = V_i(t) + \left(-I_{tot} + G \cdot \sum_j (V_j(t) - V_i(t)) \right) / C_i \cdot dt.$$

where $(t + 1)$ stands for the value for the next step. To achieve the "*compute Vi*" instruction, the control word should set the data-ram addresses for each neighbor cell Vj and for Vi.

The store instruction stores the value of a certain variable. For instance, the "*store Vi*" instruction stores the new value of Vi from another PE or from the same-PE's ODE datapath (resident variable).

The output instruction outputs the value of a resident variable to other PEs. The store and output instructions are realized by setting the signals for the input mux and data-ram. The abstract PE instructions provide an interface between the PE compiler and the implementation details.

4.3. Advantages of Custom PEs

The major advantage of custom PEs over general PEs is the reduction of cycles to calculate one ODE. Since the custom ODE datapath is fully pipelined, the average number of cycles to calculate an ODE is $(n + \textit{ODE datapath delay})/n$, where n is the number of ODEs being calculated sequentially. The cycles per ODE will approach 1 if many ODEs are mapped to one custom PE. Since the general PE usually needs 5-20 cycles to calculate each ODE (for models in our experiment, such as ODE (1) in Figure 2), we obtain about a 5-20x speedup if we map the same set of ODEs (usually containing more than 10 ODEs) from the general PE to a custom PE. The custom PE also has a smaller instruction-ram, because the number of required instructions decreases.

In the custom PE, the "evaluate and update" tasks can be calculated in one pass using the custom ODE datapath with careful scheduling. Thus, the derivative of each variable no longer needs to be stored into the data-ram, thus reducing the data-ram size. We will discuss the scheduling detail in the next section. The constant parameters are stored into a less expensive constant ROM inside the ODE datapath, which further reduces the size of the data-ram.

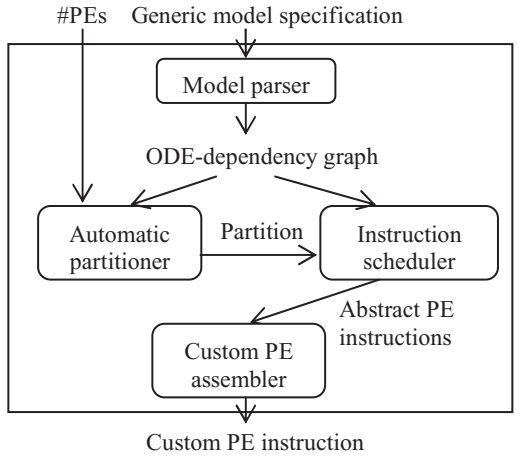


Fig. 6. Custom PE compiler overall structure.

The custom PE also eliminates the muxes and the forwarding logic in a general PE, which reduces the size of the PE. The deeply pipelined custom PE allows for synthesis tools to better optimize the circuit timing and yield higher clock frequencies than the network of general PEs.

5. CUSTOM PE COMPILER

5.1. Custom PE Compiler Overview

We developed a custom PE compiler to automate the design process. The tool's overall structure is illustrated in Figure 6. The model parser reads a generic model specification file that captures a physical model's homogenous ODEs, and generates an ODE-dependency graph. The automatic partitioner reads the ODE-dependency graph and outputs a partition file. The instruction scheduler then generates abstract PE instructions for each custom PE based on the partition and the ODE-dependency graph. The custom PE assembler translates the PE instructions into native PE control words. The following sub-sections discuss each task in more detail.

5.2. Generic Model Specification

Since a homogenous physical model contains ODEs with the same structure (or a small number of structures), the ODEs can be captured concisely using an iterator-like representation. For instance, a $15 \times 15 \times 15$ atrial cell model has a generic specification like:

$$\begin{aligned} i &= 1{:}3375 \\ V_i' &= (I + (V_{i-1} + V_{i+1} + V_{i-15} + V_{i+15} + V_{i-215} + V_{i+215} - 6 \cdot V_i) \cdot G) \cdot C, \end{aligned}$$

where Vi is the membrane potential of cell i, $Vi-1, Vi+1, Vi-15, Vi+15, Vi-215, Vi+215$ are the V of 6 neighboring cells. This generic specification file also implies the data-dependency among all the variables. The model parser will generate an ODE-dependency graph with 3,375 nodes based on this specification. The ODE-dependency graph of a physical system is often very sparse, because each variable only depends on the neighboring variables. Thus the total number of edges in the graph grows linearly with the number of nodes.

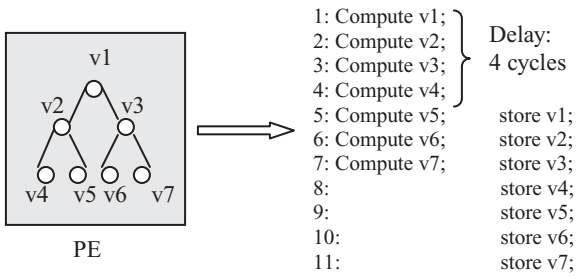


Fig. 7. Compute and store instruction scheduling.

5.3. Automatic Partitioner

The automatic partitioner partitions the ODE-dependency graph into n parts, where n is the number of PEs onto which we intend to map this ODE-dependency graph. Intuitively, we should take advantage of the graph's spatial locality by grouping nearby ODEs together to reduce communication costs between PEs.

We developed a partitioning heuristic based on simulated annealing. The main goal of the partitioning heuristic is to find a partition such that the total number of cycles per step is minimized and the total size of the network is minimized. Since the ODE solving process is executed concurrently on all PEs, the total number of cycles is determined by the PE that contains the most variables, and so the partitioner should balance the number of variables on each PE. The size of the network is mainly determined by the total number of interconnection wires among PEs. Thus, the goal of the partitioner is to minimize the cost function: *(# cycles per iteration) * (# wires)*

To speed up the partitioning heuristic, we developed custom functions to generate neighbor solutions that have a higher chance of reducing a wire or balancing the load among PEs. We also implemented an incremental cost function so that the cost is computed based on the difference of two solutions. With the incremental cost function, the partitioning algorithm usually finishes in less than 1 minute for graphs with 5,000 nodes. The quality of the resulting partition (in terms of design size and cycles per step) is usually within 20% of our manually-obtained partitions. Some further details of the automatic partitioner are discussed in Huang et al. [2012].

5.4. Instruction scheduler

Given the ODE-dependency graph and a partition, the next step is to schedule the abstract PE instructions defined in Section 4.2 for each cycle. Compute and store (resident variables) instructions are first scheduled, which correspond to the "evaluate and update" tasks in the ODE solving process. There may be data-dependencies within the resident variables. For instance, 7 variables with a dependency graph resembling a tree structure are mapped to a PE in Figure 7 Since *v2* and *v3* depend on the original value of *v1*, *v1* cannot be updated before *v2* and *v3* read the original value of *v1*. In other words, the "*store v1*" instruction can not execute before "*compute v2*" or "*compute v3*". The scheduler must take care of this "write after read" dependency.

The schedule in Figure 7 is a valid schedule, since each compute instruction can read the original value of the dependent variables. (Note that although *v3* is updated in the same cycle as "*compute v7*", we assume the write happens after read in the same cycle, as is the case in properly-clocked synchronous register-transfer-level circuits). The store instruction executes 4 cycles after the corresponding compute instruction. We call this delay *compute-to-store delay*. Since the custom ODE datapath has a pipeline delay, the actual delay can be calculated as

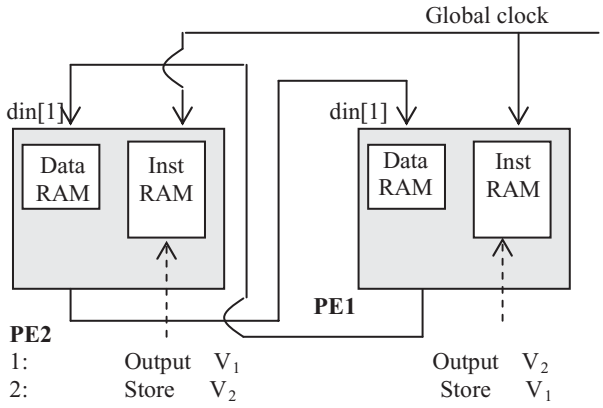


Fig. 8. Synchronized data transfer between PEs with a global clock and point-to-point connections.

Max (compute to store delay, ODE datapath delay).

To find the calculation order of all the resident variables and the compute-to-store delay, we can perform a breadth first search on the dependency graph of the resident variables. The compute-to-store delay can be found during the traversal.

The next step of the scheduler is to handle the data-transfer of dependent variables between PEs. Since we use a point-to-point connection between any two PEs, communications are performed by scheduling appropriate output and store instructions in the communicating PEs. A simple bidirectional data transfer between PE1 and PE2 is illustrated in Figure 8. PE1 and PE2 each has its output connected to the other's input port (din[1]). To exchange the value of *V1* and *V2*, PE1 and PE2 each outputs its resident variable *V1* or *V2* in the first cycle. In the next cycle, each PE can perform a store task to store the variable now located at the PE's input port.

Based on the partition and the ODE-dependency graph, the scheduler can determine a set of variables that need to be output by each PE. Multiple output instructions can be executed in parallel in the network, as long as the instructions do not conflict with each other.

Currently, all the data transfer instructions are scheduled after all PEs have updated all of their resident variables, so that the outputs are guaranteed to be the updated value.

5.5. Custom PE Generator and Custom PE Assembler

Since the architecture of a custom PE depends on the target ODE's structure, we cannot pregenerate all possible custom PEs. We thus built an automatic ODE datapath generator to facilitate custom PE generation. The ODE data-path generator reads an ODE string, and parses the ODE string into an expression tree. The generator then uses an ASAP (as-soon-as-possible) scheduling algorithm [Paulin et al. 1986] to schedule the operations, adds pipeline registers, and outputs a VHDL component for this ODE. We also developed a script to general a custom data ram for a custom PE, because the number of ports and depth is unique to each custom PE. With the ODE data-path generator, and the custom data ram generator, a custom PE can be generated on the fly.

We then built a custom PE assembler to translate each abstract PE instruction into the corresponding control word, and store the control words into the PE's instruction

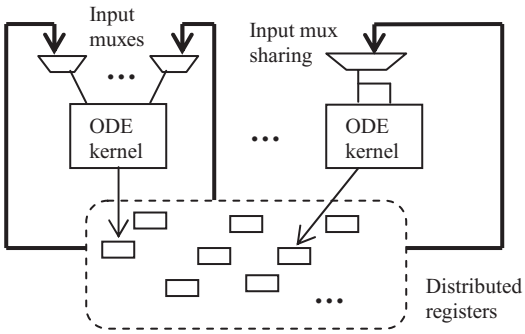


Fig. 9. Overall architecture of the ODE solver using regularity extraction.

ram. Note the control word is unique for each custom PE, thus the custom PE assembler also needs to read the specification of the custom PE component.

Once the custom PEs have been generated, a custom network can be generated automatically based on the partition obtained by the PE compiler. The final output consists of a synthesizable HDL description that loads memories and connects dependent PEs.

6. HLS WITH REGULARITY EXTRACTION

For comparison purposes, we implemented ODE solvers for physical systems using a commercial HLS tool.[1] We optimized the communication architecture by utilizing the spatial locality of each model. Since physical models exhibit much regularity, we incorporated the idea of regularity extraction into HLS as proposed by several past researchers [Rao and Kurdahi 1993], wherein an algorithm first seeks redundant sub-patterns in a dataflow graph, synthesizes an optimized component for the sub-pattern, and then strives to cover the graph using those components. For ODEs of a large homogenous physical system, the ODEs are the sub-patterns that are replicated many times to calculate different variables of the physical system. Thus, the dataflow graph of a physical system is composed of a large number of ODEs, and the ODEs are connected similar to the system's physical structure.

The ODE datapath is generated by a C representation of the equation. We tuned different parameters in the HLS tool (such as arithmetic balancing and copy reduction) intended to optimize performance, and generated fully pipelined ODE datapaths.

We tried to generate the entire ODE solver system with the HLS tool, but the tool used a unified memory with block RAMs to store all the variables. Since the block RAM only has 2 ports, the unified memory becomes a bottleneck. We instead optimized the communication architecture as illustrated in Figure 9. Instead of using a unified memory, we store each variable in its own distributed register. The input of each register comes from the ODE datapath that is responsible for calculating that variable.

Since each ODE datapath is shared by a set of variables, input muxes are needed for each ODE datapath. To reduce the size of the muxes, we used a graph partitioning algorithm (similar to the algorithm used for mapping ODEs onto a network of PEs). The partitioning algorithm utilizes the spatial locality of the ODE-dependency graph, such that the total number of inputs of an ODE datapath is reduced. Thus, the size of the input muxes is reduced. However, we still cannot guarantee that each input port of the

[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.

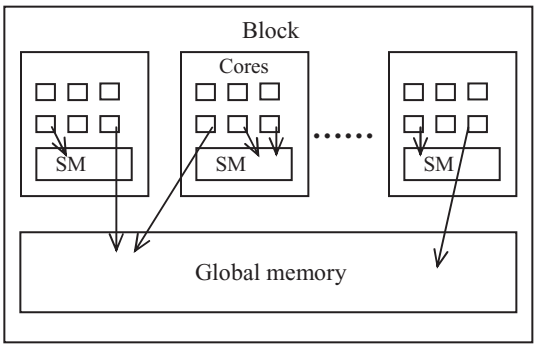


Fig. 10. Overall architecture of an Nvidia GTX460 GPU. (SM: shared memory).

ODE datapath has a dedicated input mux, due to the large FPGA LUTs consumption of the input mux. Thus an input mux may be shared by multiple input ports of an ODE datapath using time multiplex (illustrated in Figure 9). We fully automated the custom communication architecture generation for the HLS approach.

In comparison with the HLS regularity extraction approach, the network of custom PEs produces a more optimized and cleaner encapsulation of an ODE datapath. In a network of custom PEs, the variables of a physical system are stored in the local data-ram of each custom PE, instead of among distributed registers which incur added wire routing cost. The number of inputs of a custom PE is usually less than 10, while the ODE datapath in the regularity extraction approach usually has more than 50 inputs. The reduction of the number of inputs is due to the data encapsulation of the custom PEs. The data transfers occur between PEs in our approach. Thus the network of custom PEs uses the communication wires more efficiently.

7. GRAPHICS PROCESSING UNIT (GPU)

Prior to developing our custom PE approach, we originally sought to map the ODEs of physical systems to a GPU, believing such a mapping would yield competitive speedups while using relatively-inexpensive commodity parts. We investigated mapping onto an Nvidia GTX460 GPU. The overall architecture of the GPU is shown in Figure 10.

The GPU contains multiple blocks, where each block has multiple cores. The GTX460 GPU has 336 cores in total. The GPU has a global memory that can be accessed by any core. The global memory is usually composed of high-latency off-chip memory. Each GPU block contains a shared memory that can be accessed only by the cores within the block. The shared memory is an on-chip memory that has low latency and high throughput.

The basic idea is to map the variables of a physical system to different cores. The homogeneous equations are captured with the same ODE functions. Each GPU thread executes the same ODE function against different variables. To get the best performance, we tuned parameters specific to GPU implementations, such as the number of blocks and the number of threads in each block.

We also considered the spatial locality of each physical model, and mapped the variables nearby to the same block. Thus, we can load the data to the shared memory of each block at the beginning of each step, to reduce the expensive global memory accesses. Unfortunately, data exchanges are needed between blocks, because the ODEs in one block still have some communication with ODEs in other blocks. The variables must be written back to the global memory at the end of each step, because the only

way to do inter-block communication is via the global memory for the target GPU. Thus, the global memory becomes a bottleneck.

Compared to a GPU,[2] the main advantage of our approach is the custom communication structure. The data transfer tasks can be executed in parallel through the point-to-point communication wires that have a higher throughput than a global memory.

When interacting with the physical world, a GPU installed inside a PC would need to read the latest external inputs from the PC, and send out the emulated values via the PC. If real-time monitoring is necessary, the GPU-PC communication frequency would be high, thus incurring high communication overhead. In contrast, the PC and the network of PEs on an FPGA can interact with the external world by themselves, thus eliminating the extra communication step. An additional approach is to use a GPU integrated on an external data acquisition card, to reduce some of the cost/convenience benefits of the GPU versus an FPGA.

8. EXPERIMENTAL RESULTS

This section summarizes experimental results of the network of custom PEs using five physical system models. We compare the network of custom PEs with HLS including regularity extraction, networks of general PEs on an FPGA, a GPU, and other general purpose processors. Throughout this section, we execute the physical models using an Euler solver having a 10E-5 second step.

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

The FPGA-based approach 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 note that the work is not limited to a particular FPGA or synthesis tool.

8.1. Homogenous Physical System Models

We used five homogenous physical systems in the experiment, four of which are physiology models. The five models are briefly introduced in the following section, with the ODEs shown in their generic format. For simplicity, we use Ci to represent constant model parameters for each physical system. The detailed meaning of each constant parameter and the equation is omitted.

8.1.1. Weibel Lung. A Weibel lung has a binary tree structure that represents a human lung's anatomical structure. The branches at the i'th level are called generation i. The trachea in the first generation is connected to the mouth, and the last 20 to 23 generations of the Weibel lung contain millions of alveoli that handle gas exchange between the lung and capillaries. The Weibel lung's ODEs in the generic format are:

$$V_i' = F_{parent} \cdot C_1 + (V_{sib} - V_i) \cdot C_2 + F_i$$
$$F_i' = V_i \cdot C_3 - F_i \cdot C_4 - (V_{R\_child} - V_{L\_child}) \cdot C_5 - V_{R\_child} \cdot C_6 - F_i \cdot C_7$$

[2]The quality of a manually-obtained implementation, as for the GPU, is obviously related to the amount of effort applied and to the implementer's skill. In that light, we note that our original intent was to actually use the GPU, and not to compare with the FPGA approach, but we could not attain the desired speed on the GPU, and hence we resorted to creating the FPGA approach. Nevertheless, our results are merely one data point, and we strongly encourage other researchers to strive to create faster implementations of our physical models on GPUs.

(parent, sib, and child means the parent, sibling, right/left child branch of branch i, respectively), where $ViandFi$ are the volume and flow of branch i; Ci are constant parameters of each branch. Thus each branch contains two variables. We use an 11-generation Weibel lung model that contains 2,046 branches or 4,094 variables.

8.1.2. Lutchen Airway. A Lutchen airway model [Lutchen et al. 1982] contains thousands of gas cells to model the gas exchange within the nondispersive airway of human lung. The gas cells are connected in a linear structure. The ODEs of a Lutchen airway in the generic format are:

$$V_i' = C_1 \cdot (C_2 \cdot V_{i-1} - C_3 \cdot V_{i+1} + (C_3 - C_2) \cdot V_i)$$

where Vi is the volume of gas cell i, and $Vi-1$, and $Vi+1$ are two neighboring gas cells. We use a 4000-cell Lutchen model that has a system of ODEs with 4,000 dimensions.

8.1.3. Wave. A wave model [Motuk et al. 2005] has a 2-dimensional mesh network structure for modeling wave propagation, which is widely used in the areas of oil exploration seismology, laboratory ultrasonics, ocean acoustics, etc. The equation of the wave model in generic format is:

$$U_{i,j}^{t+1} = C_1 \cdot \left(U_{i+1,j}^t + U_{i-1,j}^t + U_{i,j-1}^t + U_{i+1,j+1}^t\right) + C_2 \cdot U_{i,j}^t - U_{i,j}^{t-1},$$

where $U_{i,j}^t$ means the amplitude of node(i, j) at time step i. To calculate the amplitude of a node for the next time step, we need the amplitude of four neighboring nodes. We use an 80×80 wave model that contains 6,400 variables.

8.1.4. Atrial Cell. The atrial cell model is discussed in Section 3.1. Each cell is connected to 6 neighboring cells in a 3-dimentional cubic structure. We use a $15 \times 15 \times 15$ atrial cell model that contains 3,375 variables.

8.1.5. Neuron Network. A neuron network model [Terman et al. 2008] contains a number of neuron cells to model the neuron system for the brain. The neuron cells are connected with synaptic connections. The ODEs of a neuron network in the generic format are

$$\begin{aligned} V_i' &= C_1 \cdot V_i + W_i - C_2 \cdot (V_i - C_3) \cdot \sum_j S_j, \\ W_i' &= C_4 \cdot W_i - V_i, \\ S_i' &= C_5 \cdot (1 - S_i) \cdot (V_i - C_6) - C_7 S_i, \end{aligned}$$

where Vi is the membrane potential of neuron cell i, W represents a channel gating variable, and S is a synaptic variable. Sj items are the synaptic values of the neighboring neuron cells. We use a 40×40 mesh network structure for the neuron network. Since each node contains three variables, the neuron network model has a system of ODEs with 4,800 dimensions.

These five physical system models represent four different homogenous connection schemes: linear (Lutchen), tree (Weibel), 2D mesh (wave, neuron), and 3D cubic (atrial).

8.2. Comparison with HLS with Regularity Extraction and a Network of General PEs

For comparison purposes, we implemented the ODE solver of each physical system using the HLS with regularity extraction approach discussed in Section 6, and the network of general PEs on the target FPGA.

8.2.1. Size and Performance of Each Approach. The detailed synthesis results are shown in Table I We recorded the FPGA resource utilization of each approach. For size-comparison purposes, we also define an 'equivalent LUTs' term that describes BRAM and DSP units in terms of a number of LUTs, as is commonly done [Meyer and

Table I. Synthesis results and performance comparisons between a network of general/custom PEs and high-level synthesis on a Virtex 6 FPGA. While the custom PE approach shows moderate improvement in size (subject to how one defines "size", with various size measures italicized below), the approach yields clear and substantial improvements in performance, showing 5x-10x speedups (normalized to HLS results). The approach also yields moderate improvement in implementation time due to compilation/synthesis, although such improvement is not the intent of this work.

weibel	LUTs	DSP	BRAM	Equiv.	PE/ODE kernel	Cycles	Freq. (MHz)	Perf (ms)	Speedup	Imp. Time(min)
HLS	85,693	700	50	*278,693*	50	184	130	*142*	*1.0*	330
General PE	89,761	396	396	*331,284*	396	158	179	*88*	*1.6*	277
custom PE	61,232	511	73	***215,262***	73	51	253	***20***	***7.0***	201
lutchen										
HLS	47,693	480	80	*196,493*	80	110	150	*73*	*1.0*	581
General PE	89,761	397	397	*331,931*	397	108	179	*60*	*1.2*	225
custom PE	33,933	450	150	***200,433***	150	35	305	***11***	***6.4***	170
wave										
HLS	94,046	320	80	*202,846*	80	405	140	*289*	*1.0*	340
General PE	93,958	380	380	*325,758*	380	269	175	*154*	*1.9*	260
custom PE	61,705	288	144	***185,545***	144	84	286	***29***	***9.8***	233
atrial										
HLS	123,742	365	80	*243,792*	80	368	113	*326*	*1.0*	458
General PE	79,518	219	219	*213,108*	219	418	140	*299*	*1.1*	391
custom PE	71,049	375	125	***209,799***	125	77	238	***32***	***10.1***	196
nueron										
HLS	91,459	672	50	*277,459*	50	230	110	*209*	*1.0*	281
General PE	74,632	294	294	*253,972*	294	290	150	*193*	*1.1*	227
custom PE	52,726	384	64	***171,766***	64	57	263	***22***	***9.6***	148

Kocan 2007]. By implementing equivalent DSP multiplier and BRAM components using LUTs, we give a DSP unit a value of 250 LUTs and a BRAM 360 LUTs.

Across all five models, the custom PEs use on average 30% fewer FPGA resources compared to the general PEs. The custom PEs are more efficient in ODE solving, thus fewer PE instances are required. The network of custom PEs uses on average 16% fewer FPGA resources than HLS with regularity extraction mainly due to the local data-RAM storage in the custom PE approach containing fewer input muxes and registers.

The performance number is mainly determined by the clock frequency and the cycles per step. Both general/custom PEs store the data in the local data-ram, thus the network of general/custom PEs contains fewer wires and input muxes than the regularity extraction approach. The network of general/custom PEs has better clock frequencies than HLS with regularity extraction due to less routing delays. The deeply pipelined network of custom PEs achieves the best clock frequency; on average 60% faster than the network of general PEs, and 100% faster than regularity extraction.

The cycles per step is an important factor that affects performance. The network of custom PEs uses on average 4X fewer cycles to calculate one ODE step compared to the network of general PEs, though the network of custom PEs contains fewer PEs. The gain in efficiency comes from the custom ODE datapath's fully pipelined design with single-cycle ODE calculation capability. Although HLS with regularity extraction also uses custom ODE datapaths, the large input muxes limit the number of ODE datapaths in the design. Furthermore, the "write after read" dependency discussed in Section 5.4 also exists in the regularity extraction solution, which further slows down the performance.

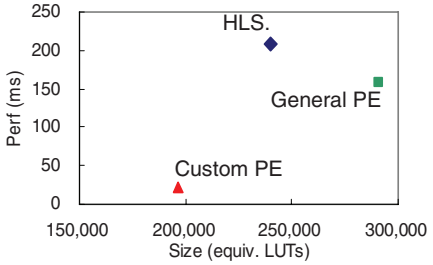


Fig. 11. Average performance and size comparison of three FPGA-based approaches; points toward the lower-left are preferred.

Comparing the performance of different physical models, we found that the Lutchen model with its linear structure has the best performance. The atrial, wave, and neuron models take longer to emulate one second, because they have a more complex physical structure, thus requiring more inter-connection wires and communications that impact overall circuit frequency.

Figure 11 shows the average size and performance comparison for the three approaches for the 5 physical models. Note the performance of the general PEs is 30% faster than regularity extraction, but it consumes 20% more equivalent LUTs. The custom PEs are on average 7X faster than the general PEs, and use 30% less LUTs. Compared to regularity extraction, custom PEs are 9X faster and 20% smaller.

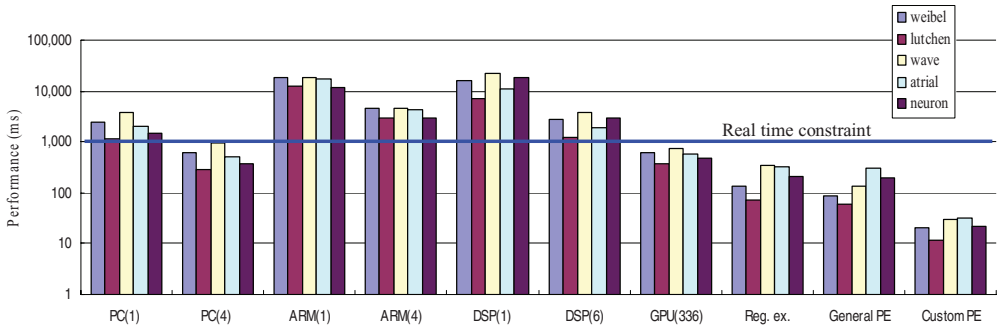
We also recorded the total implementation time of these three approaches in Table I, which includes the design and synthesis time. The three approaches have comparable design times (30 $\sim$ 90 min). The differences mainly come from the synthesis time. The HLS with regularity extraction approach consumes the most time due to more wires between the distributed registers and the input muxes. The network of custom PEs approach have the least synthesis time due to smaller sizes.

8.3. Comparing with a GPU and General Purpose Processors

8.3.1. Configuration of Each Approach. We also compared the network of custom PEs with general purpose processors and a GPU. The configuration of each processor and the GPU is listed as follows.

(1) *PC*: C code on a 3.06GHz Intel I7-950 4-core processor, compiled using gcc with –O2 flag
(2) *ARM*: C code on a 1GHz TI Cortex A9 4-core embedded processor, compiled using TMS470 compiler with –O2 flag
(3) *DSP*: C code on a 700MHz TI C6472 6-core digital signal processor, compiled using TI C6000 compiler with –O2 flag
(4) *GPU*: CUDA C code on a 763MHz NVIDIA GTX460 Fermi GPU with 336 CUDA cores, compiled using nvcc with –O2 flag

A fixed-point C implementation was used for all test cases for a fair comparison across all the general processor platforms. The C code is manually optimized and with the –O2 flags intended to optimize performance. For multicore processors, we first measure a single-threaded performance, and then calculate an optimistic performance bound for multicores by dividing the single-threaded result by the number of cores. In reality, communication overhead will degrade multicore performance, and thus the custom PE speedups would be even better.



Model	PC(1)	PC(4)	ARM(1)	ARM(4)	DSP(1)	DSP(6)	GPU(336)	HLS w/ reg. extr.	General PE	Custom PE
Weibel	2,393	619	17,843	4,460	16,435	2,739	616	139	89	**20**
Lutchen	1,182	296	12,327	3,026	7,309	1,217	378	70	61	**12**
wave	3,814	953	18,377	4,594	22,428	3,738	729	337	139	**29**
atrial	1,981	495	16,721	4,180	11,236	1,873	587	326	299	**32**
neuron	1,472	368	12,104	3,026	18,080	3,013	475	209	193	**22**

Fig. 12. Performance comparisons between general purpose processors, a GPU, HLS with regularity extraction, and networks of general and custom PEs. The numbers in the parentheses indicate the number of processor cores. Note that the plot's y-axis scale is logarithmic.

We wrote CUDA C code for each physical model on the GPU using the method discussed in Section 7. The CUDA C code is also compiled with –O2 flag.

8.3.2. Performance Comparison. Figure 12 shows the results of different approaches. Note that the plot's y-axis scale is logarithmic because of the large differences in performance on different platforms. For general purpose processors (PC, ARM, DSP), only the quad-core PC meets the real-time constraint for the physical models. The multicore DSP and ARM processors are around 5X-10X slower than the I7-950 processor. The network of custom PEs performs on average 100X faster than the single threaded PC. Compared to the optimistic multicore results of the PC, ARM, and DSP, the custom PEs still gained 24X, 183X, and 113X speedups, respectively (nonoptimistic speedups would be higher). Although the clock frequencies on the general purpose processors are higher than the network of custom PEs, the dedicated ODE datapath and custom connections among the custom PEs are more efficient in solving specific ODE systems.

Compared to the GPU results, the network of custom PEs runs on average 26X faster. The main bottleneck of the GPU implementation is the global memory. Currently, the GPU executes all five models faster than real-time. However, the communication overhead with a host PC could become limiting if the system must interact with external items. The custom PEs runs on average 50X faster than real-time, which gives much room for future monitoring and testing tasks.

We also recorded the total implementation time of each approach. Assuming a translator (from the model specification) with negligible translation time would be built, the single threaded C code took around 1 minute to compile on GCC with the –O2 flag. The GPU implementation time is around 2-3 hour, in which manual parameter tuning and optimization took most of the time. The network of custom PEs took 1-5 minutes to run through the custom PE compiler, and took another 1-2 hours to synthesize.

8.3.3. Cost Comparison. We included some approximate cost comparisons—in particular to acknowledge a limitation of our work, namely that our FPGA-based approach is currently costlier. We consider the minimal required components for each platform; as such components would contain a complete system used for emulating the physical

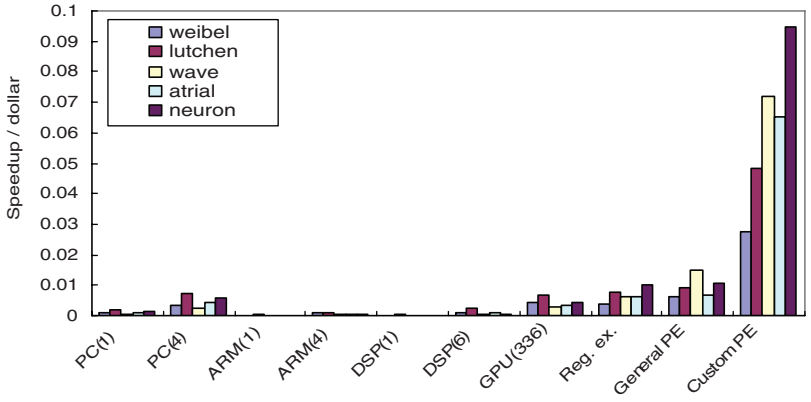


Fig. 13. Normalized speedup per dollar of different approaches.

models. The approximate cost (as of Dec 2011, obtained via Web-based distributor pricing) of each platform 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

Note that these costs are rough values, as cost is strongly dependent on customer/vendor relationships and purchase quantities.

We also consider a term that combines both cost and speedup, namely: (speedup over real-time)/cost. Figure 13 shows the results for each approach. Although the FPGA board costs the most, the speedup obtained by the FPGA based approach is greater, leading to a net benefit in terms of speedup/dollar. Among general purpose processor and GPU approaches, the multicore PC and the GPU are the best in terms of speedup/dollar, but still about 10X worse than the network of custom PEs.

8.4. Case Study

This section describes a case study of synthesizing the 11-generation Weibel lung model to a network of custom PEs. The corresponding ODE-dependency graph is a binary tree with 11 levels. To obtain the best performance, we manually partitioned the graph into 73 parts. After partitioning, the 11-level binary tree becomes a 3-level 8-ary tree as illustrated in Figure 14. One might think a leaf PE contains more variables and thus could become a bottleneck during the emulation. Actually, the leaf PE has fewer data-transfer instructions compared to the nonleaf PEs, due to not having children. Thus the total number of instructions of each PE is almost balanced.

The final placement and routing results are illustrated in Figure 15, with the blue regions used for implementation. Note the circuits of the network are almost evenly distributed in the FPGA fabric, representing the local connectivity of the physical model. We also highlighted 5 PEs by coloring the nets related to each PE. Note the nets of a PE are clustered within a small area of the FPGA. Key FPGA resource utilization information include: 61,232 LUTs (40%), 511 DSPs (67%), and 73 block-rams (18%). The minimal clock period is 3.94ns (or 253MHz maximal clock frequency). The total number of cycles per step is 51. The execution speed is 50x faster than real-time.

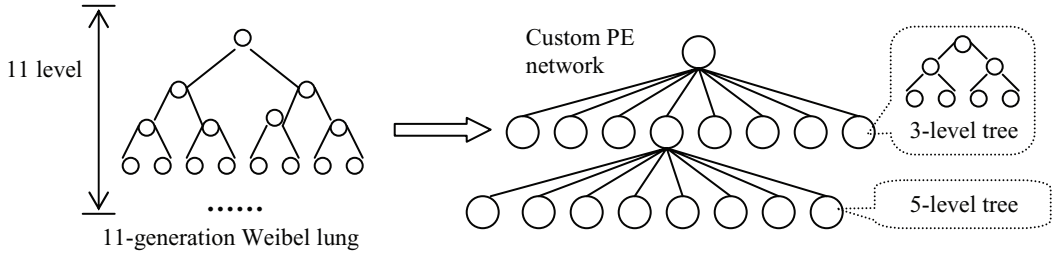


Fig. 14. The network of custom PEs of an 11 generation Weibel lung after partition. (Omitted some leaf PEs).

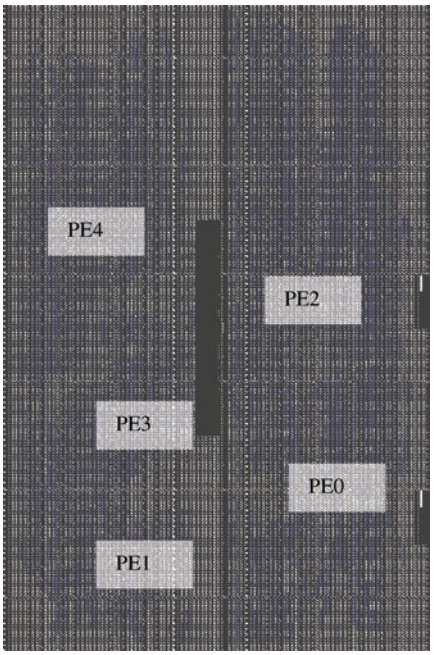


Fig. 15. Placement and routing results of the 11 generation Weibel lung on the target FPGA.

9. CONCLUSION

We described an approach for fast execution of physical models consisting of thousands of ordinary differential equations. The approach consists of synthesizing a network of processing elements customized to the model's particular ODEs, and is fully automatable. The experiments on five models, each model consisting of several thousand ODEs, and targeting a Xilinx Virtex 6 FPGA, show the custom PE approach achieves 4X-10X speedups versus previous general PEs and high-level synthesis, while using approximately the same or fewer FPGA resources. Furthermore, the approach achieves speedups of 20-30X compared to a quad-core I7-950 CPU and an Nvidia GTX460 GPU. Thus, the approach presently appears to yield the fastest execution of physical models on moderately-priced programmable platforms (excluding high-cost supercomputer platforms), and the approach appears to be robust across different types of models. Future work includes automatically synthesizing and mapping to heterogeneous PEs (especially for more comprehensive models with heterogeneous ODEs), and further comparisons with high-level synthesis, GPUs, and other evolving compute platforms.

REFERENCES

ATI Graphics Cards. 2011. http://ati.amd.com/support/driver.html.

Atkinson, K. 1993. *Elementary Numerical Analysis* 2nd Ed. John Wiley & Sons, Inc. New York, New York.

AutoEsl. http://www.xilinx.com/tools/autoesl.htm.

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

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

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

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*.

Huang, C., Vahid, F., and Givargis, T. 2011. A custom FPGA processor for physical model ordinary differential equation solving. *IEEE Embed. Sys. Lett. 3*, 4, 113–116,

Huang, C., Vahid, F., and Givargis, T. 2012. Automatic synthesis of physical system differential equation models to a processing element network on FPGAs. Trans. Embed. Comput. Syst. To appear.

Hucka, M., Finney, A., et al. 2004. Evolving a lingua franca and associated software infrastructure for computational systems biology: The Systems Biology Markup Language (SBML) project. *IEEE Syst. Biology*, 41–53.

Impulse C. 2011. http://www.impulseaccelerated.com/.

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

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, EECS Department University of California.

Lionetti, F. 2010. http://cseweb.ucsd.edu/groups/-hpcl/scg/papers/2010/Europ10-src-src-GPU.pdf.

Lutchen, F. P. Primiano, J. R., and Saidel, G. M. 1982. A nonlinear model combining pulmonary mechanics and gas concentration dynamics. *IEEE Trans. Bio-med. Electron. 29*, 629–641.

Mathworks. 2011. Matlab and Simulink. http://www.mathworks.com/.

medGadget. 2008. Supercomputer creates most advanced heart model. *Internet J. Emerg. Med. Tech*.

Meyer, J. and Kocan, F. 2007. Sharing of SRAM Tables Among NPN-Equivalent LUTs in SRAM-Based FPGAs. *IEEE Trans. VLSI Syst. 15*, 2, 182–195.

Motuk, E., Woods, R., and Bilbao, S. 2005. Implementation of finite difference schemes for the wave equation on FPGA. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.*

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*.

Paulin, P. G., Knight, J. P., and Girczyc, E. F. 1986. HAL: a multi-paradigm approach to automatic data path synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC '86)*. IEEE, 263–270.

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 Engineering in Medicine and Biology Society (EMBS '06)*. IEEE.

Rao, D. S. and Kurdahi, F. J. 1993. On clustering for maximal regularity extraction. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 12*, 8, 1198–1208.

Reshadi, M., B. Gorjiara, B., and Gajski, D. 2005. Utilizing horizontal and vertical parallelism using a no-instruction-set compiler and custom datapaths, In *Proceedings of the IEEE International Conference on Computer Design*.

Spark Project. 2005. http://mesl.ucsd.edu/spark/.

SynphonyC. 2011. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx.

Terman, D., Ahn, S., Wang, X., and Just, W. 2008. Reducing neuronal networks to discrete dynamics. *Physica D 237*, 3, 324–338.

VILLARREAL, J., PARK, A., NAJJAR, W., AND HALSTEAD, R. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*. 127–134.

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 *Field Programmable Logic and Application*, Lecture Notes in Computer Science, vol. 3203, 105–114.

ZHANG, H., HOLDEN, A. V., AND BOYETT, M. R. 2001. Gradient model versus mosaic model of the sinoatrial node. *Circulation 103*, 4, 584–588

Received January 2012; revised May 2012; accepted September 2012