Synthesis of Custom Networks of Processing Elements Onto Field-Programmable Gate Arrays for Physical System Emulation

A Dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Chen Huang

June 2012

Dissertation Committee:

Dr. Frank Vahid, Chairperson
Dr. Tony Givargis
Dr. Sheldon Tan
The Dissertation of Chen Huang is approved:

____________________________________

____________________________________

____________________________________

Committee Chairperson

University of California, Riverside
Acknowledgement

First of all, I would like to extend my sincere gratitude to my Phd advisor, Dr. Frank Vahid, for his instructive and kind guidance during my Phd process. Dr. Vahid is a very diligent and passionate advisor, who has given me not only great ideas for my research, but also good advices of being a responsible person. I am deeply grateful of his help in completion of this dissertation.

I would also like to extend my gratitude to Dr. Tony Givargis and Dr. Sheldon Tan, for being my dissertation committee member, and for their comments and helps during the completion of my dissertation. I am also deeply indebted to all the other professors who helped me during my graduate study.

I would like to thanks Dr. Scott Sirowy, Dr. David Sheldon, and Dr. Eric Cheung for their helps and advices during my early years of graduate school. I also like to thanks my other lab-mates for their instructive comments for my dissertation.

Finally, I would like to thanks my mother, Xianhua Peng, and my father, Siqing Huang, for supporting and encouraging me all the time. I am also grateful for my dear friends, and my landlord, Gladys, for their supports during my Phd process.
ABSTRACT OF THE DISSERTATION

Synthesis of Custom Networks of Processing Elements Onto Field-Programmable Gate Arrays for Physical System Emulation

by

Chen Huang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2012
Dr. Frank Vahid, Chairperson

Executing a complex physical system model in real-time or faster has numerous applications in cyber-physical system. For instance, if a human lung model can be executed in real-time, the lung model can be used to test a ventilator in real-time. A complex physical system can often be captured with thousands of ordinary differential equations (ODEs). We introduce an approach to map the ODEs of a physical system to a custom network of processing elements on a field-programmable gate array (FPGA). A processing element (PE) is a light-weight processor that solves a subset of the ODEs. The custom interconnection of the processing elements is based on the data dependencies of the ODEs. The processing elements can execute in parallel and communicate with each other. To automate the design process, we developed a compilation tool to find a good mapping between the ODEs and the processing elements, and to generate synthesizable HDL (hardware description language) code for the entire design.

We first investigated a general purpose processing element that can solve any type of
ODEs. The network of general PEs achieves 10-20x speedups against a single-threaded Intel I7-950 CPU, and 4x speedups against an Nvidia GTX 460 GPU. The network of general PEs also yields 2x speedups compared to a commercial high-level-synthesis tool. We further optimized our approach by building custom processing elements that can only solve certain type of ODEs. For homogeneous physical systems (contains only one or a few types ODEs), the network of custom PEs yields another 6x speedup compared to the network of general PEs, given comparable size. Finally, we introduced the network of heterogeneous PEs, where the network may contain both general PEs and different types of custom PEs. We developed an allocation and binding heuristic to explore the large design space. The network of heterogeneous PEs achieves 7x/6x speedup against the network of general PE/single-type custom PEs, and was on average 10x faster than the circuits generated by a high-level synthesis tool.
# Table of Contents

Section 1: Introduction ........................................................................................................ 1

Section 2: Related works .................................................................................................... 4

Section 3: Custom network of general processing elements .............................................. 8

  Physical system modeling and ode solving ................................................................. 10

  Architecture of a processing element (PE) and a network of PEs ............................... 15

  Converting floating-point numbers to fixed-point numbers ......................................... 20

  Synthesis and Compilation of a network of PEs .......................................................... 23

  Experimental results ................................................................................................... 34

  Discussion .................................................................................................................... 50

Section 4: Network of custom processing elements ......................................................... 52

  Physical system modeling and a network of processing elements approach ............... 53

  Custom processing element ....................................................................................... 58

  Custom PE compiler .................................................................................................... 62

  HLS with Regularity extraction ................................................................................. 68

  Graphics Processing Unit (GPU) ................................................................................ 70

  Experimental results .................................................................................................. 72

  Discussion ................................................................................................................... 85

Section 5: Network of heterogeneous processing elements ............................................. 87

  Physical system emulation with a custom network of processing elements ............... 89

  Network of Heterogeneous PEs .................................................................................. 94

  Experimental results ................................................................................................. 104
Lists of figures

Fig. 1. Synthesizing an eight-generation Weibel lung model into network of PEs on FPGAs..9

Fig. 2. The first four generations of a Weibel lung model.......................................................10

Fig. 3. RLC circuit modeling of a bifurcating airway. ............................................................11

Fig. 4. Three additional physical model examples: .................................................................14

Fig. 5. Non-pipelined PE architecture .....................................................................................15

Fig. 6. Pipelined PE architecture.............................................................................................16

Fig. 7: Mapping a 3-generation Weibel lung to a network of PEs...........................................18

Fig. 8. Synchronized data transfer between PEs with global clock and point-to-point
connection. ......................................................................................................................19

Fig. 9. PE synthesis and compiler tool overall structure ..........................................................23

Fig. 10. A 2-generation Weibel lung model specification and ODE-dependency graph .........24

Fig. 11. ODE-to-PE mapping ..................................................................................................25

Fig. 12. Size neighbor generator (B: m/n stands for benefit: #ODE / edge weight)...............28

Fig. 13. ODE-PE mapping algorithm......................................................................................29

Fig. 14. PE-dependency graphs at different iterations.............................................................31

Fig. 15. ODE solving stages.................................................................................................... 32

Fig. 16. Size and performance comparison between high-level synthesis and the networks of
PEs...........................................................................................................................................39

Fig. 17. Performance / throughput (GOPS) comparisons between several general purpose
processors, a GPU, and the network of PE approach..............................................................44
Fig. 34. A four generation Weibel lung with gas exchange model ........................................ 89

Fig. 35. Mapping a four generation Weibel lung model to a custom network of 5 PEs ....... 91

Fig. 36. ODE solving process in a network of 3 PEs ............................................................ 92

Fig. 37. Merging tightly coupled ODE types ...................................................................... 98

Fig. 38. Overall structure of the allocation and binding heuristic for the network of
heterogeneous PEs ............................................................................................................ 99

Fig. 39. The overall architecture of the ODE solver using an HLS tool with regularity
extraction .......................................................................................................................... 106

Fig. 40. GPU kernel function implementation options ................................................... 113

Fig. 41. The performance for each approach .................................................................... 114

Fig. 42. The normalized speedup for each approach ......................................................... 115
Lists of tables

Table I. Synthesis results of different single PE versions........................................................35

Table II. Synthesis results and performance comparisons between networks of PEs and high-level synthesis. ........................................................................................................38

Table III. Synthesis results and performance comparisons between the network of PEs, HLS, and manually design for the Weibel 11 model on the target FPGA .................................40

Table IV. Synthesis results and performance comparisons between networks of general/custom PEs and high-level synthesis on a Virtex 6 FPGA ........................................77

Table V. Synthesis results and performance comparisons for a small atrial model on the smaller Xilinx Virtex5 110T-1 FPGA. .............................................................80

Table VI. Synthesis results of networks of heterogeneous PEs, HLS, and general/single-type custom PEs. ........................................................................................................108
Section 1: Introduction

The capturing and execution of physical systems have gained extensive research attention in recent decades. A physical system is a mathematical representation of a physical phenomenon. Executing a physical model may help with understanding a physical process, such as a human heartbeat, or the respiration behavior of the human lung. The executing of a physical system could also be used for testing a control algorithm in a cyber-physical system, such as testing a ventilator with a human lung model, or testing a pacemaker with a human heart model. To make the physical system models interact with the physical world or devices, accurate and real-time execution are the two main constraints. We focus on addressing these two constraints in this work.

In order to represent a physical system accurately, a complex mathematical model should be used. A physical system’s mathematical model typically consists of thousands of ordinary differential equations, or ODEs. Execution consists of solving the ODEs using iterative techniques, such as Euler or Runge-Kutta [10]. In order to meet the accuracy and the real-time execution constraints, the execution contains 1000 iterations or more per second. Thus executing a complex physical model in real-time could be computationally expensive. Even today’s high end desktop processor or GPUs (graphic processing unit) could hardly meet the real-time constraint for executing a complex physical system.

We notice each ODE in a physical system model represents part of the physical space, each part connected with neighboring parts, and all parts executing in parallel. This
computation pattern matches the inner structure of FPGAs (field-programmable gate arrays) well, because the ODEs can be mapped to circuit components in the FPGA, each component connects to neighboring components, thus avoiding memory or input/output bottlenecks common in FPGA applications. The ODEs mapped to different FPGA components can execute in parallel to increase the execution speed. Furthermore, the interconnection of these FPGA components can be configured to match the communication structure of a physical system. The custom communication architecture in the FPGA often outperforms the general communication architectures found in CPUs and GPUs for physical system emulation.

We thus propose a custom network of processing elements for fast emulation of a physical system on the FPGA. Each processing element (or PE) is a light-weight custom processor that executes a subset of the ODEs in a physical system. The processing elements are connected based on the data dependencies among the ODEs. We note the structure of a custom network usually matches the physical connection structure of the model, which represents an efficient communication structure.

We developed a compilation tool (or the PE compiler) to automate the design process. We notice the mapping between ODEs and PEs has a huge impact on the performance and size of the design, thus we developed an automatic mapping heuristic based on simulated annealing. After a good mapping has been found, the compilation tool will instantiate the PE components, and generates instructions for each PE instance. Finally, the tool will connect all PE instances and outputs synthesizable VHDL files of the entire design.

The design of the processing element is critical for our approach. The size of a PE should
be minimized, thus more PEs can be instantiated to increase the throughput of the system. We also want a PE to be flexible for solving different types of ODEs. We first developed a general PE component that contains a general purposed ALU (arithmetic logic unit). The general PE has a NISC (no-instruction-set-computer) [52] architecture to reduce the size. The network of general PEs will be discussed in Section 3.

We notice many large physical systems are homogeneous systems, which contains thousands of ODEs of the same type (or a few types). We thus developed custom PE components. Each custom PE contains a custom datapath for a certain ODE type. The custom datapath is fully pipelined, thus the throughput of a custom PE can reach one ODE per cycle if many ODEs of the same type are mapped to the custom PE. The network of custom PEs achieved 5-10x speedups compared to the network general PEs. The network of custom PEs will be discussed in Section 4.

Finally, we introduce the network of heterogeneous PEs in Section 5, where the design may contain both general PEs and a few types of custom PEs. Thus the system has both the flexibility of the general PEs and the performance of the custom PEs. The network of heterogeneous PEs is especially useful for heterogeneous physical systems that contain many types of ODEs. However, the design space of a heterogeneous network increases, because we need to choose the PE types, the number of PEs for each PE type, and the mapping from ODEs to PEs. We developed an allocation and binding heuristic to explore the design space of the network of heterogeneous PEs.
Section 2: Related works

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 function of the human body. Electromagnetic models are developed to understand and predict electromagnetic behavior, and are widely used in cellular phones, mobile computing, etc.

Languages have been introduced for modeling physical systems, such as the mathematical modeling language (MML) [46], the systems biology markup language (SBML) [22], and CellML [11]. Tools were developed for simulating physical systems, such as Matlab [35], LabView [45], JSim [27], Mathematica [34], etc. Those 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 units (GPUs) [7][47]. Multi-core processors [2][25] and supercomputers [23] have been utilized to exploit the parallelism inside physical models. For instance, a 768-core SGI machine executed a 2 billion equation heart model, simulating 0.4 ms in 2 hours (18 million times slower than real-time) [36]. An Nvidia GTX 295 GPU was used to execute a Flaim heart model 30x faster than OpenMP, with less than 1% error [32]. Executing one heart beat (300 ms) required 7.7 minutes. While multi-core 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 multi-threaded programs for multi-core and GPU.

Many case studies using FPGAs to speed up simulation have been conducted. FPGAs were used to speedup fine-grained intra-cellular simulation [53], showing that an FPGA could hold 500 reactions related to gene expression. Yoshimi [63] obtained 100x speedups of a fine-grained biological simulation compared to a single processor, and showed why multi-cores 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 [50]. That work showed that a PC required 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 optimal formula based on how many multipliers and the performance of each multiplier of their target FPGA, rather than an implementation. Chen [13] used FPGAs to do an inductive dynamic simulation with a Runge-Kutta ODE solver. The custom FPGA implementation resulted in a 100x speedup over a 2.2 GHz desktop computer using Simulink [35]. Osana [48] 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 [26] used FPGAs to simulate ODE-based multi-model biochemical simulations, proposing several scheduling and resource sharing methods to optimize implementation on FPGAs. The above efforts mostly used manual design to implement the physical models on FPGAs, requiring
much human effort for design and test. This work proposes a systematic and scalable approach to synthesize physical system models into a custom network of PEs on an FPGA.

Custom processors have been studied extensively in the electric design automation field. Many tools and methods have been proposed to automatically implement an application as a custom processor. Cong [14] introduced a distributed register-file processor micro-architecture for FPGAs. The micro-architecture utilizes a platform featuring on-chip memory and register-file cores to reduce multiplexing logic and global interconnection. The micro-architecture is similar to the PE in this work, but is aimed at general applications, while our PE is optimized specifically for solving ODEs of a physical system model. A no-instruction-set-computer concept was introduced by Reshadi [52]. 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 [44], Streams-C [18], DEFACTO [16], SPARK [54], ROCCC [60], Celoxica [12], SynphonyC [55], etc. 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 [51]. We show that our tool generates faster and smaller implementations due to the tool’s specific focus on a complex network of ODEs.
Section 3: Custom network of general processing elements

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 custom network of differential equation processing elements (PEs), intended for FPGAs, to efficiently solve the ODEs of a physical system. Each PE is a light-weight 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 [37].
We created a PE synthesis method, embodied in a PE synthesis tool, to automatically convert a model’s ODEs (and other equations) into VHDL [59] code 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 tradeoffs 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. Fig. 1 shows an example of mapping ODEs of an eight-generation Weibel lung model [61] to a network 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 the networks have similar binary tree structures compared to the Weibel lung.

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

Modeling physical systems with ODEs

As an example of modeling a physical system, consider modeling a human’s lung. Weibel [61] proposed a lung model having a binary tree structure to reflect a human lung’s anatomic structure. Fig. 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. A split 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

![Fig. 2. The first four generations of a Weibel lung model.](image-url)
A bifurcating airway of the Weibel lung structure can be modeled as the RLC circuit shown in Fig. 3. The physical property of each branch is captured with R (resistance), Com (capacitance, a.k.a. compliance in lung terminology), and L (inductance). Each branch has unique R, Com, and L values according to the branch’s physical properties; 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.

ODE solving process and data dependencies among ODEs

The equations illustrated in Fig. 3 model the bifurcating airway. These equations include

ODEs for branch 1:
\[ P_{\text{mid}}[1] = \frac{V[1]}{\text{Com}[1]} \]
\[ \text{Fout}[1] = \text{Fin} - \text{F}_{\text{mid}}[1] \]
\[ \frac{d(V[1])}{dt} = \text{F}_{\text{mid}}[1] \]
\[ \frac{d(\text{Fout}[1])}{dt} = \frac{(P_{\text{mid}}[1] - P_{\text{out}}[1] - \text{Fout}[1] \times \text{R}_{2}[1])}{\text{L}[1]} \]

ODEs for branch 2:
\[ P_{\text{mid}}[2] = \frac{V[2]}{\text{Com}[2]} \]
\[ \text{Fin}[2] = \text{Fout}[1] - \text{Fin}[3] \]
\[ \text{Fout}[2] = \text{Fin}[2] - \text{F}_{\text{mid}}[2] \]
\[ \frac{d(V[2])}{dt} = \frac{(P_{\text{mid}}[2] - \text{Pout}[1] - \text{Fin}[1] * \text{R}_{2}[1])}{\text{L}[2]} \]
\[ \frac{d(\text{Fout}[2])}{dt} = \frac{(P_{\text{mid}}[2] - P_{\text{out}}[2] - \text{Fout}[2] \times \text{R}_{2}[2])}{\text{L}[3]} \]

Fig. 3  RLC circuit modeling of a bifurcating airway.
ordinary equations, e.g., $P_{\text{mid}[1]} = V[1] / \text{Com}[1]$ (middle pressure of branch1 is equal to volume of branch1 divided by capacitance of branch1), as well as ordinary differential equations (ODEs), e.g., $\frac{d(V[1])}{dt} = F_{\text{mid}[1]}$ (the derivative of branch1’s volume is equal to the middle flow of branch1).

This paper defines variables on the left-hand side of ODEs as ODE variables, e.g., $V[1]$, $F_{\text{out}[1]}$. The ordinary equations calculate temporary values that are used in the ODEs, e.g., $P_{\text{mid}[1]}$ is used for calculating $\frac{d(F_{\text{out}[1]})}{dt}$. Substituting temporary values yields the general ODE format: $\frac{d(X)}{dt} = \text{Fun}(X)$, where $X$ is a vector of the ODE variables: $V[1]$, $V[2]$, $V[3]$, $F_{\text{out}[1]}$, $F_{\text{out}[2]}$, $F_{\text{out}[3]}$. The derivative of $X$ is a function of $X$.

To solve these ODEs, iterative solvers such as Euler [6] or Runge-Kutta [10] are often used. Starting from time 0, iterative solvers move forward in time by a given time step, such as by 1 ms. Note that there are data dependencies among ODEs. For instance, the ODE $\frac{d(F_{\text{out}[1]})}{dt} = (V[1] / \text{Com}[1] - P_{\text{out}[1]} - F_{\text{out}[1]} * R_{2[1]})/ L[1]$ depends on the ODE $\frac{d(V[1])}{dt} = F_{\text{mid}[1]}$, because the $V[1]$ value updated by the second ODE at each step 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 three stages:

**Evaluation:** Calculate the derivative of each ODE variable, e.g., $\frac{d(V[1])}{dt} = \text{Fin} - F_{\text{out}[1]}$.

**Update:** Update each ODE variable using the derivatives calculated in the evaluation stage, e.g., $V[1] = V[1] + \frac{d(V[1])}{dt} * dt$, where $dt$ is time step.

**Data transfer:** Propagate the new value of each ODE variable to the ODEs that depend
on the ODE 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 ODE variables on the right-hand size of the ODE, e.g., \(V[1]\), \(Fout[1]\) in the following ODE:

\[
\]

Thus the evaluation and updating stages can be calculated in parallel on each processor. However, new ODE 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 ODE 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 \(dt^5 < dt^2\)).

Physical model examples

This section uses four complex physical models as examples. The first is the Weibel lung model described above. The second is an entirely different lung model called the Lutchen lung [33]. The third is a wave model [43]. The fourth is an atrial cell model [64]. Fig. 4 briefly shows the physical structure of the three additional models.
Fig. 4(a) shows the structure of the Lutchen lung model, which contains three components: a non-dispersive airway and two alveolar compartments. The Lutchen lung emphasis is on modeling gas exchange in the non-dispersive airways, each of which is each segmented into a number of gas cells. Each gas cell only connects with two neighbor cells, e.g., 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 Fig. 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).

Fig. 4(c) shows a 3-dimensional atrial cell model intended to model a heart for interacting with a pacemaker. 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 cell (Vj) in a 3-dimensional 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), 3D 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 80x80 Wave model with 6400 ODEs, and a 15x15x15 Atrial cell mode with 3375 ODEs. The four physical models are denoted as Weibel 11, Lutchen 4000, Wave 80x80, Atrial cell 15 throughout this section.

Architecture of a processing element (PE) and a network of PEs

This section summarizes the architecture of a single processing element (PE), and then introduces the communication architecture for a network of PEs.

General PE architecture

We earlier proposed an architecture for a single PE optimized for ODE solving [21]. The optimization goals included minimizing the PE’s FPGA resource requirements, while also

Fig. 5. Non-pipelined PE architecture
maximizing the PE’s performance for solving ODEs.

Fig. 5 reviews the general (non-pipelined) 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 use microcoded control words [3] to eliminate instruction decoding logic and thus improves size and performance, and is similar to the idea of the No Instruction Set Computer (NISC) [52]. The PE has two types of control words: store and compute. A store control word stores data 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. The detailed discussion about these two operations can be found in [ref]. The PE architecture components are adjustable to the ODEs mapped to the PE: The number of inputs port, data and instruction RAM sizes, and ALU operations can each be adjusted.

PE performance optimization

The longest register-to-register delay, known as the critical path, determines a PE’s maximum clock frequency. The critical

![Forward path 1 (delay 1 cycle)](Data reg) -> MUX -> data RAM

![Forward path 2 (delay 2 cycles)](Data RAM) -> MUX -> data reg

Fig. 6. Pipelined PE architecture.
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 units resulting in long wire delays. To reduce the critical path, we added pipeline registers as shown in Fig. 6. We added 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 non-pipelined version) to 170 MHz for the pipelined version, for some large designs.

However, pipelining increases the number and size of control words, requiring more instruction RAM. The non-pipelined PE can perform a computation and result write-back with one control word, e.g. 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 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 non-pipelined architecture. In terms of resource usage, the pipelined architecture incurs a LUTs penalty of 10% to 20%.
A 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, i.e., a one-to-one ODE-to-PE mapping. Due to data dependencies among ODEs, each PE must communicate with other PEs. Fig. 7 shows a 3-generation Weibel lung model, having 7 ODEs, mapped to a 7-PE network. Note that the communication 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 scheme eliminates

Fig. 7: Mapping a 3-generation Weibel lung to a network of PEs.
resource-costly handshaking logic. A simple bidirectional data transfer between two PEs is illustrated in Fig. 8. PE1 and PE2 each has 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 improve performance by increasing PE clock frequency due to better synthesis onto FPGA resources. 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.

Miller [41] describes 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.

**Converting floating-point numbers to fixed-point numbers**

Floating-point cores on an FPGA consume much more resources compared to a fixed-point core. For instance, the Xilinx floating-point IP core for a 32-bit add/sub/multiplication takes 636 LUTs and 2 DSPs on a Xilinx Virtex6 FPGA, while the fixed-point core only takes 48 LUTs and 1 DSP. The floating-point computation is also slower, in terms of latency. For example, the Xilinx floating-point core has a latency of 3–8 cycles (depends on different clock frequency requirements), while the fixed-point core can achieve single cycle latency. Given the size and latency advantages, we decide to use fixed-point number in our current design.

Currently, we manually convert floating-point numbers into 32-bit fixed-point numbers that can be executed efficiently with the integer ALU and shifter in the processing element. The manual conversion process is illustrated by the following Weibel lung ODE example:

\[
\]

(F: flow, V: volume, Com: Compliance, R: Resistance, the number in the square parenthesis is the branch index of the Weibel lung)

The first rule of the fixed-point operation is that the additions and subtractions must be
executed at the same scale level (or the radix point must be aligned). We define an *ODE scale factor*, where all the additions and subtractions in an ODE are executed at the same scale level. For instance, the operands \((F[1] \times R[1], V[3] \times Com[3], V[2] \times Com[2])\) for addition and subtraction operations in this ODE are the pressures of branch 1, 3, 2, respectively. Thus the ODE scale factor for this ODE is the scale factor for the pressure. For simplicity, we use the same ODE scale factor for all ODEs of the same type. For this example, the ODE scale factor can be calculated by: \(\text{the max value of INT32} / \text{max pressure in the system}\).

We notice that the value range of each variable may vary much for a physical model. For example, the flow and volume changes significantly at each generation for the Weibel lung model. We thus define a *variable scale factor* for each variable which is calculated by: \(\text{max value of INT32} / \text{max value of this variable}\). The max value of each variable can be obtained by floating point simulation [29].

The second rule of the fixed-point operation is related to multiplication. The scale factor of the multiplication result is the product of the scale factors of the operands. For instance, if A’s scale factor is 16 and B’s scale factor is 32, then the scale factor of \(A \times B\) is \(16 \times 32 = 512\). To maintain the ODE scale factor (if the multiplication result is an operand of an addition or a subtraction), we associate a shifter with the multiplier. The right shift amount is determined by: \(\log_2(\text{scale factor}(A) \times \text{scale factor}(B) / \text{ODE scale factor})\).

To verify the accuracy of the fixed-point conversion, we compared the fixed-point version results with a double precision floating-point implementation in Matlab for the Weibel lung example. 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.

One concern for the manual fixed-point conversion is the conversion time. For homogeneous physical systems (with only 1, 2 types of ODEs), we only need to calculate one or two ODE scale factors. Although a physical system may contain thousands of variables, we may just need to calculate one or a few variable scale factors. For instance, we can use the same variable scale factor for some models, e.g. the wave, atrial cell, because all variables in these physical systems have similar value ranges. For the Weibel lung model, variables at different generations have different value ranges. However, the variables at the same generation have very similar value ranges, thus we only need different variable scale factors for each generation. For homogeneous systems, the number of different variable scale factors is rarely exceeding 20 based on our experiments. The total manual conversion time (including floating-point simulation) for a homogeneous physical system is usually 20 min ~ 1 hour.

For a heterogeneous physical system (may contains more than one sub-systems), the manual fixed-point conversion time for each sub-system is comparable to a homogeneous system. Thus the total manual conversion time is usually within 1~4 hours. Note the floating-point to fixed-point conversion is a one time conversion. Once the conversion is done, the results can be used for different PE network implementations. The manual conversion could be automated in the future using any one of several established techniques, such as Kum [29], if complex heterogeneous models are often used. We could also explore other resource efficient floating-point cores on the FPGA in the future.
Synthesis and Compilation of a 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 those ODEs into the control words for each PE. The tool’s overall structure is illustrated in Fig. 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. The designer can choose a design based on the target application’s requirements. Finally, the tool generates PE control words, the connection structure of the network, and synthesizable VHDL files for FPGA implementation using standard synthesis tools.

![Fig. 9. PE synthesis and compiler tool overall structure.](image-url)
Model specification and ODE-dependency graph

Fig. 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 ODE 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 Fig. 10(b).

Fig. 10. A 2-generation Weibel lung model specification and ODE-dependency graph
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 section 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 a network of PEs. Fig. 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 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 with thousands of ODEs. Hence, the ODE-to-PE mapping algorithm uses an iterative search heuristic based on simulated annealing.

![Fig. 11: ODE-to-PE mapping](image-url)
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 $\#\text{cycle}$ as the number of cycles of the bottleneck PE. The design size is mostly determined by the total number of connections (abstract connection between 2 PEs, not the physical wires in the FPGA implementation) in the network, denoted as $\#\text{connection}$. To combine these two metrics, we use a cost function $\#\text{cycle} \times \#\text{connection}$, 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 ($\#\text{cycle}$). 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 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 Fig. 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 Fig. 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 above steps are called an iteration. To speedup the algorithm, we optimized the cost function. 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 [28]. The incremental cost function speeds up the original cost function by more than 100x.

**Neighbor mapping generation**

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 generation. The performance neighbor generator chooses a random ODE(a) from the bottleneck PE (the PE with the most cycles), and moves ODE(a) to a random PE who contains an ODE that connected with ODE(a) 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 the convexity constraint 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 Fig. 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 \( \#\text{ODE} / \text{edge weight} \) and moves the ODE with the maximal benefit. The generator prefers to move an ODE with smaller edge weight that having the most chance of reducing a connection. For instance, move ODE1 from PE1 to PE2 will reduce a connection. The \( \#\text{ODE} \) 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 ODEs, to balance the number of ODEs in each PE (to improve performance). Thus the size neighbor generator also considers the convexity constraint.
Objective:  \( \min (\#\text{cycle} \times \#\text{connection}) \)

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 \( T (T=CostI / \#\text{iteration}) \) and go back to Step 2 until total number of iterations reached.

Output: best ODE-PE mapping found

Fig. 13 ODE-PE mapping algorithm

ODE-to-PE mapping algorithm

The ODE-to-PE mapping algorithm is illustrated in Fig. 13. Note the algorithm inputs include the number of PEs and the number of iterations. Fig. 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 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 near-optimal manual mappings. For the four physical models in this section, the total number of cycles (performance) of the automatic mapping is usually within 15% of the manually partition. The only exception is the 3D 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 structure, such as the 3D cubic atrial cell model. The mapping algorithm could be further improved for the models with a complex connection structure.
Fig. 14. PE-dependency graphs at different iterations
PE instruction generation

After generating the mapping between ODEs and PEs, the next task of the tool is to generate PE instructions and the VHDL code for the network.

Fig. 15 shows ODE solving stages of three PEs, and how PEs are synchronized. The ODE solving process has three stages: evaluation, updating, and data transfer. The evaluation and updating processes can be executed independently within each PE, and they may finish at different clock cycles. Evaluation and updating 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 updating task (PE2 in this example).

The tool statically schedules the data transfer between PEs, after a PE has updated local variables, based on ODE data dependences. 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, e.g., 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’ 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 updating instructions to complete the PE instructions for one solving step. The number of cycle for each PE is determined by the number of instructions.

**HDL code generation**

To generate the HDL code for a custom network of PEs, we first created a PE component pool for different PE versions. Each PE version has a unique triple of input ports number, instruction ram size, data ram size. Each PE also has a generic map for the data and instruction ram. We developed a script to generate the HDL 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 port, 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 network of PEs. Since we already know the network structure by the automatic mapping algorithm, the PE compiler will convert the edges (in the network) into HDL wires. The HDL code generation flow is fully automated by the PE compiler without the help of any other tools.
Experimental results

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

The Weibel 11 (4094 ODEs), Lutchen 4000 (4000 ODEs), Atrial cell 15 (3375 ODEs) models use an RK4 solver with a 0.0001 second step, while the Wave 80x80 (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 1 second of simulated time. For example, “300” means an implementation executed 1 second of simulated time in just 300 milliseconds (thus executing faster than real time).

The PE 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 [62] 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.
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), e.g., 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 instruction RAM contains 1, 2, 3, or 4 BRAMs. We added implementations for $4 \times 4 \times 4 = 64$ PE versions into our PE pool to adapt to different physical models’ requirements.

<table>
<thead>
<tr>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>freq. (MHz)</th>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE3_D32_I1</td>
<td>214</td>
<td>1</td>
<td>1</td>
<td>193</td>
<td>PE1_D64_I1</td>
<td>264</td>
<td>1</td>
</tr>
<tr>
<td>PE3_D64_I1</td>
<td>284</td>
<td>1</td>
<td>1</td>
<td>194</td>
<td>PE3_D64_I1</td>
<td>284</td>
<td>1</td>
</tr>
<tr>
<td>PE3_D128_I1</td>
<td>378</td>
<td>1</td>
<td>1</td>
<td>197</td>
<td>PE7_D64_I1</td>
<td>312</td>
<td>1</td>
</tr>
<tr>
<td>PE3_D1024_I1</td>
<td>184</td>
<td>1</td>
<td>2</td>
<td>193</td>
<td>PE15_D64_I1</td>
<td>376</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Different data RAM sizes  
(b) Different numbers of input ports
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 [40]. 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.

Network of PEs versus high level synthesis

A high-level synthesis (HLS) tool usually takes C code of a system/function and converts the C code into synthesizable HDL 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 with optimization for a custom communication architecture.

Since the four physical models each has a homogenous structure, the corresponding ODEs of each model have the same format. For instance, the Wave model’s ODEs have the following format: \( \frac{d(v[i][j])}{dt} = a * (v[i-1][j] + v[i+1][j] + v[i][j-1] + v[i][j+1]) + b * v[i][j] \), where i, j represents the position of a node in a 2-dimensional 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 tradeoff 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 registers. Input muxes are needed for each ODE unit. The custom communication architecture consumes extra FPGA resource for the data registers and input muxes, but avoids the block RAM bottleneck thus improves performance.

\[1\] 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.
Performance and size comparison

Table II shows synthesis results of the HLS and networks 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

<table>
<thead>
<tr>
<th>Model</th>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>LUTs</th>
<th>Cycle/step</th>
<th>freq.(MHz)</th>
<th>perf.(ms)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weibel 11</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS(10)</td>
<td>57,985</td>
<td>160</td>
<td>80</td>
<td>126,785</td>
<td>3,856</td>
<td>105</td>
<td>367</td>
<td>116</td>
</tr>
<tr>
<td>HLS(40)</td>
<td>104,190</td>
<td>640</td>
<td>80</td>
<td>292,990</td>
<td>964</td>
<td>112</td>
<td>86</td>
<td>356</td>
</tr>
<tr>
<td>PE(64)</td>
<td>13,421</td>
<td>64</td>
<td>320</td>
<td>144,621</td>
<td>3,900</td>
<td>181</td>
<td>215</td>
<td>69</td>
</tr>
<tr>
<td>PE(396)</td>
<td>89,724</td>
<td>396</td>
<td>396</td>
<td>331,284</td>
<td>780</td>
<td>178</td>
<td>44</td>
<td>277</td>
</tr>
<tr>
<td><strong>Lutchen 4000</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS(20)</td>
<td>48,150</td>
<td>300</td>
<td>60</td>
<td>144,750</td>
<td>2,760</td>
<td>91</td>
<td>303</td>
<td>160</td>
</tr>
<tr>
<td>*HLS(80)</td>
<td>187,555</td>
<td>480</td>
<td>80</td>
<td>336,355</td>
<td>690</td>
<td>68</td>
<td>101</td>
<td>570</td>
</tr>
<tr>
<td>PE(63)</td>
<td>13,008</td>
<td>63</td>
<td>315</td>
<td>142,158</td>
<td>3,247</td>
<td>182</td>
<td>178</td>
<td>70</td>
</tr>
<tr>
<td>PE(397)</td>
<td>89,761</td>
<td>397</td>
<td>397</td>
<td>331,931</td>
<td>549</td>
<td>179</td>
<td>31</td>
<td>225</td>
</tr>
<tr>
<td><strong>Wave 80*80</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS(20)</td>
<td>92,743</td>
<td>180</td>
<td>80</td>
<td>166,543</td>
<td>1,280</td>
<td>102</td>
<td>558</td>
<td>142</td>
</tr>
<tr>
<td>*HLS(80)</td>
<td>167,472</td>
<td>480</td>
<td>160</td>
<td>345,072</td>
<td>320</td>
<td>84</td>
<td>170</td>
<td>438</td>
</tr>
<tr>
<td>PE(63)</td>
<td>13,780</td>
<td>63</td>
<td>189</td>
<td>97,570</td>
<td>1,402</td>
<td>166</td>
<td>372</td>
<td>62</td>
</tr>
<tr>
<td>PE(380)</td>
<td>93,958</td>
<td>380</td>
<td>380</td>
<td>325,758</td>
<td>269</td>
<td>175</td>
<td>68</td>
<td>260</td>
</tr>
<tr>
<td><strong>Atrial cell 15</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS(20)</td>
<td>36,881</td>
<td>100</td>
<td>180</td>
<td>126,681</td>
<td>8,250</td>
<td>85</td>
<td>971</td>
<td>216</td>
</tr>
<tr>
<td>*HLS(80)</td>
<td>133,558</td>
<td>400</td>
<td>160</td>
<td>291,158</td>
<td>2,025</td>
<td>104</td>
<td>195</td>
<td>402</td>
</tr>
<tr>
<td>PE(63)</td>
<td>29,696</td>
<td>63</td>
<td>315</td>
<td>158,846</td>
<td>6,225</td>
<td>140</td>
<td>445</td>
<td>80</td>
</tr>
<tr>
<td>PE(219)</td>
<td>87,452</td>
<td>309</td>
<td>309</td>
<td>275,942</td>
<td>1,320</td>
<td>145</td>
<td>91</td>
<td>272</td>
</tr>
</tbody>
</table>
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 networks of PEs and HLS yield comparable clock cycles per step for their designs. However, the networks of PEs have about a 1.5x-2x faster clock frequency. In terms of FPGA resource usage, our approach 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. Fig. 16 shows the average size (in equivalent LUTs) and performance of the four models using HLS and the networks of PEs. Note that for comparably-sized implementations, the networks of PEs achieve 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. The network of PEs 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 HLS’s tool time includes: HLS tool (5~25 min), Xilinx ISE (1~5 hour), and manual communication architecture coding (1~3 hour). The network of PEs uses on average 50% less time than HLS.

Compare networks 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 Weibel lung’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.

Table III. Synthesis results and performance comparisons between the network of PEs, HLS, and manually design for the Weibel 11 model on the target FPGA

<table>
<thead>
<tr>
<th>Weibel 11</th>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>Eqiv. LUTs</th>
<th>Cycle/step freq.(Mhz)</th>
<th>perf(ms)</th>
<th>imp. time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS(40)</td>
<td>104,190</td>
<td>640</td>
<td>80</td>
<td>292,990</td>
<td>964</td>
<td>112</td>
<td>86</td>
</tr>
<tr>
<td>Manual design</td>
<td>78,597</td>
<td>350</td>
<td>100</td>
<td>202,097</td>
<td>820</td>
<td>192</td>
<td>43</td>
</tr>
<tr>
<td>PE(396)</td>
<td>89,724</td>
<td>396</td>
<td>396</td>
<td>331,284</td>
<td>780</td>
<td>178</td>
<td>44</td>
</tr>
</tbody>
</table>
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 increase the performance by 100% due to fully customized ODE datapath and communication architecture.

Although the networks of PEs used more FPGA resource 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 types 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 flexible and can be easily instrumented for future debug and monitor purposes.

Network of PEs vs. general purpose processors and a GPU

We compare the networks of PEs with other modern general purpose processor approaches. These approaches include a modern Intel I7 processor based on the X86-64 instruction set [24], a TI ARM processor based on a RISC (reduced instruction set computer) instruction set [5] a TI digital signal processor with an optimized architecture for the fast operational needs of digital signal processing, and a NVIDIA graphic processor unit with specialized circuits designed to accelerate video processing. The GPU is programmed with ‘CUDA C’ [15] (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++
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**: networks of PEs 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 [57]. For the multi-core general purpose processors, rather than conducting time-consuming multi-threaded implementation, we instead measured single-threaded performance first and then calculated an optimistic performance bound for multi-cores simply by dividing the single-threaded performance by the number of cores; in reality, communication overhead will degrade multi-core performance.

For comparison purpose, we also implement 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 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 [1][20].

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 global memory, according to CUDA programming guide[15]. 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.

Performance comparison

The performance and throughput results for different platforms are highlighted in Fig. 17. To give slacks for future debugging and monitoring purposes, we define a performance constraint as 500 ms 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 single threaded PC respectively, and both fail to meet the real time constraint for all four models. The GPU runs
on average 3.4x faster than PC(1) due to parallel execution on multiple cores.

The networks of PEs runs on average 15x faster than the single threaded PC for the four models. Note that the network of PEs’ average clock frequency is around 170 MHz, which is 18x slower than the I7-950 processor running at 3.06GHz. The 15x speedup is gained by extensive parallelization on hundreds of PEs, and also by efficient ODE solving on each PE.

Compared to the performance upper bound of multi-core 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 multi-threaded PC(4) runs every model faster than performance constraint, the implementation of a multi-threaded ODE solver is non-trivial. Careful partitioning of the ODEs of a physical model is needed for a multi-threaded version. 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 are approximately 4X larger than GPU and PC(4).

Comparing to the GPU, the network of PEs is on average 4.4x faster. Although the GPU has a much higher clock frequency (763 MHz), the general purpose memory architecture in the GPU may not match the physical model’s requirements. We utilized the shared memory in each GPU block to reduce global memory access, however all the variables of a model needs to be written back to the global memory after each time step as discussed earlier. An ODE kernel function invocation is necessary at each step. The overhead of frequent ODE kernel invocation took 40% - 70% of total GPU execution time.

We also compared our GPU results with other GPU ODE solvers for physical systems [1][4][42]. Those papers use much larger physical models usually around 1000K ODEs, and the execution speed is usually more than 100X slower than real-time. The larger model executes each time step longer, thus hides the penalty of the frequent kernel invocation, and gained larger speedups over CPU than our results However, this work emphasized on real-time emulation. We note the GPU results may be further optimized, and we strongly encourage other researchers to create faster implementations of our physical models on GPUs.
To give a fair comparison to different approaches, we spent comparable amount of time for each approach. The C implementation took around 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. The network of PEs 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.

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 work. The approximate cost (as of Jan 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 Fig. 18. One can see that, although the FPGA board itself is currently more expensive, the speedup obtained by that board is greater, leading to a net benefit in terms of speedup / dollar, competitive with the GPU and PC(4). However, we note again that the general-purpose processor and the GPU speedups are optimistic; communication overhead will degrade those 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 the similar total cost. Furthermore, an FPGA may better support custom interfaces to the real physical world [41]. Also, FPGA cost trends may continue to improve the FPGA speedup/dollar relative to the other approaches, although the FPGA trend vs. PC and GPU trends is hard to predict.

![Fig. 18. Normalized speedup per dollar of different approaches, using optimistic (no communication overhead) estimates for all multi-core devices except the PE approach.](image-url)
This section shows 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.

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 (20 ODEs per PE). The PE compiler performed automatic ODE-to-PE mapping and generated the
PE-dependency graph in Fig. 19(a). Note that the custom network of the Weibel 11 model 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 Fig. 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

(a) Square input pressure

(b) Sine input pressure

Fig. 20. First branch volume
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 of PEs simulates 1 second of real time in 97 ms, 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 Fig. 20. The total simulated time is 10 seconds. The fixed-point network of PEs implementation has almost identical results compared to the desktop implementation. From the figures on the right, we notice that the network of PEs’ results have errors within 0.2% compared to the desktop implementation.

Discussion

We described a processing element (PE) for efficient solving of physical model differential equations, introduced a custom network of PEs for parallelized solution of large models, and described a fully-automated approach for implementing physical model equations on modern FPGAs. Comparing to a commercial high-level synthesis tool on several
models, our approach were on average 2.1x faster with 2x fewer LUTs, 2x fewer DSPs, but 3.5x more BRAMs. The network of PEs 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 block RAM resource. Our future work includes reducing PE instruction and data storage overhead, such that a network of PEs 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 [41].
Section 4: Network of custom processing elements

To accelerate physical model emulation, we introduced an approach using a network of processing elements (PEs) optimized for ODE solving in the previous section. 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. 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

Atrial cell model’s ODE form
(repeated thousands of times):  
\[
d(V_i)/dt = (-I_{tot} + G \cdot \sum_j (V_j - V_i))/C_i
\]

Fig. 21. Synthesizing an atrial cell model into a network of custom PEs on an FPGA.
well-known centralized-data communication bottleneck in FPGAs). For example, Fig. 21 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 the previous section, the PEs were optimized for solving general ODEs, and not for any specific ODEs. In this section, we examine the benefits of synthesizing PEs customized to the specific ODEs mapped onto each PE, which is mainly used for large homogenous physical systems. The custom PEs are still programmable, thus enabling for example insertion of instructions for profiling or debugging. Because many large physical systems are homogenous systems with only a few (<5) types of ODEs replicated many times, like the atrial cell model in Fig. 21), this section focuses on using one custom-synthesized PE that is replicated in a network; future work may examine heterogeneous PEs.

Physical system modeling and a network of processing elements

Physical system modeling and a network of processing elements

Physical system models are often captured as ordinary differential equations (ODEs). Fig. 22 shows a 3-dimensional atrial cell model intended to model a heart for interaction with a pacemaker [64]. \(V_i\) is the membrane potential of cell \(i\), \(I_{tot}\) is the total ionic current in a cell, \(G\) is the coupling conductance, and \(C\) is the cell capacitance. \(V_j\) stands for the neighboring
The derivative of \( V_i \) with respect to time is determined by the sum of the membrane potential differences between the cell and neighboring cells \( (V_j) \).

The ODEs of a physical system can be written in the form: \( \frac{d(X)}{dt} = \text{Fun}(X) \), where \( X \) is a vector of the variables. In the atrial cell model, \( X \) is the membrane potentials for all cells in the model (note \( I_{tot}, G, \text{ and } C \) are constant parameters of the system, but not variables). The scale of a physical system is determined by the number of variables (or dimensions) of the physical system’s ODEs. 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 [6] or Runge-Kutta [10]. 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, e.g., equation (1) in Fig. 22.

(2) *Update*: Estimate the variable values for the next time step using current values and the derivatives calculated above, e.g., \( V_i = V_i + \frac{d(V_i)}{dt} \cdot h \), where \( h \) is the time step.

![Atrial cell model](image-url)
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.

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 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, Fig. 23(a) shows the ODE dependency graph of a 4x4 2-dimensional atrial cell model, where each node contains the membrane potential of each cell. Note that we use undirected edges, because the dependencies are typically bi-directional in a physical system.

Fig. 23. Mapping a 4x4 2-dimensional atrial cell network to a 4-PE network.
To increase the speed of the physical system emulation, we can parallelize the ODE solving process. More specifically, we can map the ODE dependency graph to a network of PEs. For example, Fig. 23 maps the 4x4 atrial cell model to a 4-PE network. The variables in each of the dashed rectangles are mapped to one PE. The interconnection among those 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 persist in the local memory of their 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 access to variables residing in other PEs. For example, to calculate \( \frac{d(V_1)}{dt} \) in Fig. 24 (a simplified version of equation (1)), PE1 needs the latest value of variables \( V_2 \) and \( V_3 \) that reside in PE2 and PE3. We refer to the variables needed in the evaluate task but not resident in the PE as dependent variables.

\[
\frac{d(V_1)}{dt} = V_2 + V_3 - 2V_1 \tag{2}
\]

Fig. 24. Data transfer task (variables in parentheses are dependent variables.)
Since the values of dependent variables are produced in other PEs, we let a PE store a copy of each dependent variable. At the end of each time step, each PE must update the dependent variables by reading them from other PEs, and must output the latest values of its resident variables to the PEs that depend on them. We call this a “data transfer” task. A data transfer example is illustrated in Fig. 24 with directed arrows.

General PE

We earlier proposed a processing element (PE) for general ODE solving. The PE is a no-instruction-set-processor (NISC) [52], which eliminates the instruction decoding logic to reduce the size of the processor.

The PE’s micro-architecture is illustrated in Fig. 25. Each instruction is directly mapped to a control word that controls three major components: An input mux, a data-ram, and an arithmetic logic unit (ALU). The PE contains a few input ports (three in the figure) and one output port, for communicating with external input/output or with other PEs. The data ram

---

Fig. 25. General ODE-solving PE micro-architecture.
stores variables and parameters of a certain part of a physical model. The ALU calculates the equations by the single cycle “compute” and “store” operations. For instance, the equation:

\[ \frac{d(V1)}{dt} = V2 + V3 - 2 \cdot V1 \]

is compiled into the following four operations:

Cycle 1, compute \( V2 + V3 \) \( \rightarrow \) reg1
Cycle 2, compute \( 2 \cdot V1 \) \( \rightarrow \) reg2
Cycle 3, compute \( \text{reg1} - \text{reg2} \) \( \rightarrow \) reg1
Cycle 4, store \( \text{reg1} \) \( \rightarrow \) dram[0] (the \( \frac{d(V1)}{dt} \) ’s location)

The PE is optimized by inserting pipeline registers and forward paths to increase the clock frequency and to reduce temporary-value write backs. Since the PE can handle different ODEs by compiling the equations into basic compute operations, we call this type of PE a general PE.

**Custom processing element**

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 reflect 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, or a Weibel lung [61] 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:

\[
\frac{d(V_i)}{dt} = (-I_{\text{tot}} + G \sum_j (V_j - V_i)) / C_i
\]

Where \(V_j\) 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. Fig. 26 shows the micro-architecture of a custom PE with the custom ODE datapath for the atrial cell. Note that 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, the data-ram is built with FPGA LUTs (lookup tables).
Abstract PE tasks

In each time step, a PE needs to perform three tasks: evaluate, update, and data transfer, as discussed earlier. 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 \( V_i \)” task performs:

\[
V_i(t + 1) = V_i(t) + (-I_{\text{tot}} + G \sum_j (V_j(t) - V_i(t))) / C_i \cdot dt
\]

Where \((t + 1)\) stands for the value for next step. To achieve the “compute \( V_i \)” instruction, the control word should set the data-ram addresses for each neighbor cell \( V_j \) and for \( V_i \).

The store instruction stores the value of a certain variable. For instance, “store \( V_i \)” instruction stores the new value of \( V_i \) from other PE or from the 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.

Advantages of custom PEs

Comparing to a general PE, the major advantage of the custom PE 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 \((n + \text{ODE datapath delay}) / n\), 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 5-10 cycles (for models in our experiment) to calculate each equation, we obtain about a 5x speedup. The custom PE also has a smaller instruction-ram, because the number of clock cycles 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.

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 general ODE-solving PE.
Custom PE compiler

We developed a custom PE compiler to automate some of the design process. The tool’s overall structure is illustrated in Fig. 27. 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.
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 15x15x15 atrial cell model has a generic specification like:

\[ i = 1 : 3375 \]

\[ V'_i = (I + (V_{i-1} + V_{i+1} + V_{i-15} + V_{i+15} + V_{i-215} + V_{i+215} - 6 \cdot V_i) \cdot G) \cdot C \]

Where \( V_i \) is the membrane potential of cell \( i \), \( V_{i-1} \), \( V_{i+1} \), \( V_{i-15} \), \( V_{i+15} \), \( V_{i-215} \), \( V_{i+215} \) 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.

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 the advantage of the graph’s spatial locality by grouping ODEs nearby together to reduce communication costs between PEs.

We developed a partitioning heuristic. 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 of PEs 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 custom 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 Section 3.

Instruction scheduler

Given the ODE-dependency graph and a partition, the next step is to schedule the abstract PE instructions defined earlier for each cycle. The scheduler first schedules the compute and store (for resident variables) instructions, which correspond to the “evaluate and update” tasks in the ODE solving process. We notice 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 Fig. 28. Since v2 and v3 depend on the original value of v1,
v1 cannot be updated before v2 and v3 read the original v1. In other words, the “store v1” instruction cannot execute before “compute v2” or “compute v3”. The scheduler must take care of this “write after read” dependency.

The schedule in Fig. 28 is a valid schedule, since each compute instruction can read the original value of its 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 its own pipeline delay, the actual delay can be calculated as: 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.

Fig. 28. Compute and store instruction scheduling.
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 2 PEs, the communications between 2 PEs are performed by the output and the store instructions. A simple bidirectional data transfer between PE1 and PE2 is illustrated in Fig. 29. 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 must be output by each PE. Multiple output instructions can be executed in parallel in the network of PEs, as long as the instructions do not conflict with each other.

Currently, all the data transfer instructions are scheduled after each PE has updated all of its resident variables, so that the outputs are guaranteed to be the updated value.

Fig. 29. Synchronized data transfer between PEs with a global clock and point-to-point connections.
Custom PE assembler and custom ODE datapath

A custom PE assembler translates each abstract PE instruction into the corresponding control word, and stores the control word into the instruction ram. Since the control word is partly determined by the custom ODE datapath, the assembler needs to handle each type of ODE datapath in the system. The custom PE assembler is different for each physical model, because the custom ODE datapath of each model is different.

Currently, we manually build and optimize each custom PE. The critical component of a custom PE is the custom ODE datapath. We first convert an equation to an expression tree with minimal depth, and use an ASAP [49] scheduling algorithm to schedule the operations. To maximize the clock frequency and throughput of the ODE datapath, we use a fully pipelined design and assume enough computation components (adder, multiplier, etc) exist on the target platform. Once the custom ODE datapath has been built, a data RAM is allocated that has enough output ports to support the ODE datapath, and the instruction RAM is loaded with the control words.

A homogenous physical system often contains only 1 or 2 types of ODEs, reducing the total manual design time of the corresponding custom PEs to about 1 or 2 hours. The custom PE design process could be automated in the future.

Once the custom PE (or PEs) to be used in the network has been designed, a network design can be generated automatically based on partition obtained by the PE compiler. The final output consists of a synthesizable HDL description that loads memories and connects dependent PEs.
HLS with Regularity extraction

For comparison, we implemented ODE solvers for physical systems using a commercial HLS tool. In previous work, we ran the HLS tool with a custom communication architecture. In this work, we further optimized the communication architecture by utilizing the spatial locality of each model. Since physical models exhibit much regularity, we incorporate the idea of regularity extraction into HLS as proposed by several past researchers, wherein an algorithm first seeks redundant sub-patterns in a dataflow graph, synthesizes an optimized component for the sub-pattern, and then strives the cover the graph using those components.

For ODEs of a large homogenous physical system, the ODEs are the sub-patterns and 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 manually optimized the communication architecture as illustrated in Fig. 30. Instead of using a unified memory, we

---

2 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.
used distributed registers to store the variables, each register storing a variable. 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 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 Fig. 30).

In comparison with the HLS regularity extraction approach, the network of custom PEs implementation 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 network of PEs. The data transfers occur between PEs in the network. Thus the network of custom PEs shares the communication wires more efficiently.

**Graphics Processing Unit (GPU)**

Prior to developing our custom PE on FPGA 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 Fig. 31.

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 of solving the ODEs using a GPU 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 for 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 still needed between blocks, because the ODEs in one block still have some communication with ODEs in other blocks. Even though such inter-block communication is typically with neighboring blocks, the variables still 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, the main advantage of the network of custom PEs 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 this extra communication step. Using a GPU integrated on an external data acquisition card, representing another approach, would reduce some of the cost/convenience benefits of the GPU versus an FPGA.

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, and with previous network of general PEs. We further compared the network of custom PEs with a GPU and with other general purpose processors. Throughout

---

3 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.
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 1 second of simulated time. For example, “300” means an implementation executed 1 second 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 [62] for synthesis. We note that the work is not limited to a particular FPGA or synthesis tool.

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 $C_i$ to represent constant model parameters for each physical system. The detailed meaning of each constant parameter and the equation is omitted.

Weibel lung

A Weibel lung has a binary tree structure that represents 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:

\[
\begin{align*}
V_i' & = F_{\text{parent}} \cdot C_i + (V_{\text{sib}} - V_i) \cdot C_2 + F_i \\
F_i' & = V_i \cdot C_3 - F_i \cdot C_4 - (V_{L_{\text{child}}} - V_{R_{\text{child}}}) \cdot C_5 - V_{R_{\text{child}}} \cdot C_6 - F_i \cdot C_7
\end{align*}
\]

(paren parent, sib, and child means the parent, sibling, right/left child branch of branch i, respectively), where \(V_i\) and \(F_i\) are the volume and flow of branch \(i\); \(C_i\) 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.

Lutchen airway

A Lutchen airway model [33] contains thousands of gas cells to model the gas exchange within the non-dispersive airway of human lung. The gas cells are connected in a linear structure. The ODEs of a Lutchen airway in the generic format are:

\[
\begin{align*}
V_i' & = C_1 \cdot (C_2 \cdot V_{i-1} - C_3 \cdot V_{i+1} + (C_3 - C_2) \cdot V_i)
\end{align*}
\]

where \(V_i\) is the volume of gas cell \(i\), and \(V_{i-1}\) and \(V_{i+1}\) are two neighboring gas cells. We use a 4000-cell Lutchen model that has a system of ODEs with 4,000 dimensions.

Wave

A wave model [43] 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:

\[
U_{i,j}^{t+1} = C_1 \cdot (U_{i+1,j} + U_{i-1,j} + U_{i,j-1} + U_{i,j+1}) + C_2 \cdot U_{i,j} - U_{i,j}^{t+1}
\]
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 80x80 wave model that contains 6,400 variables.

Atrial cell

The atrial cell model is discussed in the previous section. Each cell is connected to 6 neighboring cells in a 3-dimensional cubic structure. We use a 15x15x15 atrial cell model that contains 3,375 variables.

Neuron network

A neuron network model [56] 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:

\[
\begin{align*}
V_i' &= C_1 \cdot V_i + W_i - C_2 \cdot (V_i - C_3) \cdot \sum_j S_j \\
W_i' &= C_4 \cdot W_i - V_i \\
S_i' &= C_5 \cdot (1 - S_i) \cdot (V_i - C_6) - C_7 S_i
\end{align*}
\]

where \( V_i \) is the membrane potential of neuron cell \( i \), \( W \) represents a channel gating variable, and \( S \) is a synaptic variable. \( S_j \) items are the synaptic values of the neighboring neuron cells. We use a 40x40 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).

Compare to HLS with regularity extraction and network of general PEs

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

Size and performance of each approach

The detailed synthesis results are shown in Table IV. We recorded the FPGA resource utilization of each approach. For size-comparison purposes, we also define an equivalent LUTs term, wherein assign to BRAM and DSP units a size value in terms of a number of LUTs, as is commonly done [39]. 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 network of custom PEs uses on average 30% fewer FPGA resources compared to the network of general PEs, because the custom PE is more efficient in ODE solving, thus fewer custom PEs are required. Note that the custom PEs used fewer BRAM components compared to the general PEs due to fewer PEs. 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 network of custom PEs containing fewer input muxes and registers.
Table IV. Synthesis results and performance comparisons between networks 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. The approach also yields moderate improvement in implementation time due to compilation/synthesis, although such improvement is not the intent of this work.

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>Equiv. LUTs</th>
<th>PE/ODE datapath</th>
<th>Cycles</th>
<th>Freq. (MHz)</th>
<th>Perf. (ms)</th>
<th>Imp. Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weibel</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS w/ reg. extr.</td>
<td>85,693</td>
<td>700</td>
<td>50</td>
<td>278,693</td>
<td>50</td>
<td>184</td>
<td>130</td>
<td>142</td>
<td>330</td>
</tr>
<tr>
<td>general PE</td>
<td>89,761</td>
<td>396</td>
<td>396</td>
<td>331,284</td>
<td>396</td>
<td>158</td>
<td>179</td>
<td>88</td>
<td>277</td>
</tr>
<tr>
<td>custom PE</td>
<td>61,232</td>
<td>511</td>
<td>73</td>
<td><strong>215,262</strong></td>
<td>73</td>
<td>51</td>
<td>253</td>
<td>20</td>
<td>201</td>
</tr>
<tr>
<td><strong>Lutchen</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS w/ reg. extr.</td>
<td>47,693</td>
<td>480</td>
<td>80</td>
<td>196,493</td>
<td>80</td>
<td>110</td>
<td>150</td>
<td>73</td>
<td>581</td>
</tr>
<tr>
<td>general PE</td>
<td>89,761</td>
<td>397</td>
<td>397</td>
<td>331,931</td>
<td>397</td>
<td>108</td>
<td>179</td>
<td>60</td>
<td>225</td>
</tr>
<tr>
<td>custom PE</td>
<td>33,933</td>
<td>450</td>
<td>150</td>
<td><strong>200,433</strong></td>
<td>150</td>
<td>35</td>
<td>305</td>
<td>11</td>
<td>170</td>
</tr>
<tr>
<td><strong>wave</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS w/ reg. extr.</td>
<td>94,046</td>
<td>320</td>
<td>80</td>
<td>202,846</td>
<td>80</td>
<td>405</td>
<td>140</td>
<td>289</td>
<td>340</td>
</tr>
<tr>
<td>general PE</td>
<td>93,958</td>
<td>380</td>
<td>380</td>
<td>325,758</td>
<td>380</td>
<td>269</td>
<td>175</td>
<td>154</td>
<td>260</td>
</tr>
<tr>
<td>custom PE</td>
<td>61,705</td>
<td>288</td>
<td>144</td>
<td><strong>185,545</strong></td>
<td>144</td>
<td>84</td>
<td>286</td>
<td>29</td>
<td>233</td>
</tr>
<tr>
<td><strong>atrial</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS w/ reg. extr.</td>
<td>123,742</td>
<td>365</td>
<td>80</td>
<td>243,792</td>
<td>80</td>
<td>368</td>
<td>113</td>
<td>326</td>
<td>458</td>
</tr>
<tr>
<td>general PE</td>
<td>79,518</td>
<td>219</td>
<td>219</td>
<td>213,108</td>
<td>219</td>
<td>418</td>
<td>140</td>
<td>299</td>
<td>391</td>
</tr>
<tr>
<td>custom PE</td>
<td>71,049</td>
<td>375</td>
<td>125</td>
<td><strong>209,799</strong></td>
<td>125</td>
<td>77</td>
<td>238</td>
<td>32</td>
<td>196</td>
</tr>
<tr>
<td><strong>neuron</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS w/ reg. extr.</td>
<td>91,459</td>
<td>672</td>
<td>50</td>
<td>277,459</td>
<td>50</td>
<td>230</td>
<td>110</td>
<td>209</td>
<td>281</td>
</tr>
<tr>
<td>general PE</td>
<td>74,632</td>
<td>294</td>
<td>294</td>
<td>253,972</td>
<td>294</td>
<td>290</td>
<td>150</td>
<td>193</td>
<td>227</td>
</tr>
<tr>
<td>custom PE</td>
<td>52,726</td>
<td>384</td>
<td>64</td>
<td><strong>171,766</strong></td>
<td>64</td>
<td>57</td>
<td>263</td>
<td>22</td>
<td>148</td>
</tr>
</tbody>
</table>
The performance number is mainly determined by the clock frequency and the cycles per step. Both networks of general/custom PEs approaches store the data in the local data-ram, thus the network of PEs contain much less wires and input muxes than the regularity extraction approach. The network of general/custom PEs had 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 contain 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 number of input muxes may not satisfy the needs of the ODE datapath. Furthermore, the “write after read” dependency discussed earlier also exists in the regularity extraction solution, which further slows down the performance.

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. Fig. 32 shows the size and performance comparison for the three approaches for the 5 physical models. Note the average performance of the network of general PEs is 30% faster than regularity extraction, but it
consumes 20% more equivalent LUTs. The network of custom PEs is on average 7X faster than the network of general PEs and uses 30% less LUTs. Compared to regularity extraction, the network of custom PEs is 9X faster and 20% smaller.

We also recorded the total implementation time of these three approaches in Table II, which includes the design and synthesis time. Three approaches have comparable design times (30 ~ 90 min). The differences mainly come from the synthesis time. The regularity

![Diagram](Image)

**Fig. 32.** Performance and size comparison of three FPGA-based approaches; points toward the lower-left are preferred.
extraction consumes most time due to more wires between the distributed registers and the input muxes. The networks of custom PEs have the least synthesis time due to smaller sizes.

Resource constrained scenario

To test each FPGA based approach under a resource constrained scenario, we implemented an atrial model on a smaller Xilinx Virtex5 110T-1 FPGA (69120 LUTs, 64 DSPs, and 148 BRAMs). The Virtex5 110T FPGA is 3.6X smaller than the Virtex6 240T FPGA in terms of total equivalent LUTs, and contains 10X fewer DSP modules. The original atrial model greatly exceeded the capacity of this FPGA, and thus we used a small atrial model with 1000 cells (10 x 10 x 10), which still exceeded the capacity but by a more reasonable amount. The small atrial model is 3.4X smaller than the original atrial model.

The synthesis results are shown in Table V. The number of PE/ODE datapaths for all three implementation approaches are mainly constrained by the available DSP modules. The performance of the small atrial model of each approach is comparable to the large atrial model on the target Virtex6 FPGA, showing that each approach adapts to the constrained scenario well.

Table V. Synthesis results and performance comparisons for a small atrial model on the smaller Xilinx Virtex5 110T-1 FPGA.

<table>
<thead>
<tr>
<th>atrial(small)</th>
<th>LUTs</th>
<th>DSP</th>
<th>BRAM</th>
<th>Equiv. LUTs</th>
<th>PE/ODE datapath</th>
<th>Cycles</th>
<th>Freq. (MHz)</th>
<th>Perf (ms)</th>
<th>Imp. Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS</td>
<td>31,322</td>
<td>60</td>
<td>12</td>
<td>50,642</td>
<td>12</td>
<td>367</td>
<td>100</td>
<td>367</td>
<td>73</td>
</tr>
<tr>
<td>general PE</td>
<td>21,813</td>
<td>57</td>
<td>57</td>
<td>56,583</td>
<td>57</td>
<td>458</td>
<td>140</td>
<td>327</td>
<td>97</td>
</tr>
<tr>
<td>custom PE</td>
<td>18,215</td>
<td>64</td>
<td>27</td>
<td><strong>43,935</strong></td>
<td>27</td>
<td>90</td>
<td><strong>38</strong></td>
<td>239</td>
<td><strong>94</strong></td>
</tr>
</tbody>
</table>
Comparing to GPU and general purpose processors

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 3.06 GHz Intel I7-950 4-core processor, compiled using gcc with –O2 flag.
2. ARM: C code on a 1 GHz TI Cortex A9 4-core embedded processor, compiled using TMS470 compiler with –O2 flag.
3. DSP: C code on a 700 MHz TI C6472 6-core digital signal processor, compiled using TI C6000 compiler with –O2 flag.
4. GPU: CUDA C code on a 763 MHz 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 multi-core processors, we first measure a single-threaded performance, and then calculate an optimistic performance bound for multi-cores by dividing the single-threaded result by the number of cores. In reality, communication overhead will degrade multi-core performance, and thus the custom PE speedups would be even better.
We wrote CUDA C code for each physical model on the GPU using the method
discussed earlier. The CUDA C code is also compiled with –O2 flag.

Fig. 33 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 4 processor PC meets the real-time
constraint for the physical models. The multi-core DSP and ARM processors are around
5X-10X slower than the 17-950 processor. The network of custom PEs performs on average
100X faster than the single threaded PC. Compared to the optimistic multi-core results of the
PC, ARM, and DSP, the network of custom PEs still gained 24X, 183X, and 113X speedups,
respectively (non-optimistic 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 network of custom PE 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 1 hour, in which manual parameter tuning and optimization took around 50 minutes. The network of custom PEs required 1 hour to manually build the custom PE (which could easily be automated in the future), took 1 minute to run through the custom PE compiler, and took another 1-2 hours to synthesize.

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 models. The approximate cost of each platform is as follows:

- **CPU (I7-950 + Intel X58 board):** $480
- **ARM (Cortex 9A 4-core board):** $300
- **DSP (TI C6472 board):** $350
- **GPU (GTX460 + I3-540 + H55 board):** $380
- **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. Fig. 34. 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 multi-core PC and the GPU are the best in terms of speedup / dollar, but still about 10X worse than the network of custom PEs.

Discussion

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 (although we manually created the micro-architecture of the custom PEs in this work). The Experiments on five models—a Weibel lung model, a Lutchen lung model, an atrial heart model, a neuron model, and a wave model, each model consisting of several thousand ODEs—and targeting a Xilinx Virtex 6 FPGA, show 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, with 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 still shows roughly 10X improvement, with decreasing FPGA costs improving that metric. 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 and different model and FPGA sizes. 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.
Section 5: Network of heterogeneous processing elements

In the previous two sections, we proposed a network of homogeneous processing-elements on FPGAs (field-programmable gate arrays) approach for even faster emulation of a physical system. A set of lightweight custom processors each solve a subset of ODEs in the physical system. The structure of the network and interconnections of the PEs is based on ODE interdependencies. Models with widely-varying structures can be efficiently implemented on FPGAs due to FPGA configurability. Initially, a custom network of general PEs was used, with each general PE able to solve any ODEs, yielding 10-20x speedups over a single-threaded Intel I7 desktop processor running at 3.07 GHz. Later, a custom network of custom PEs was introduced, where a single custom PE was first created to best match the needs of the model’s ODEs and then a network of those custom PEs was used to achieve 5x-10x speedups over the network of general PEs.

In contrast, we propose to create a custom network of heterogeneous PEs consisting of general PEs and multiple custom PEs together, as shown in Fig. 35. The proposed method greatly expands the exploration problem associated with mapping model equations to the correct type of PE, but can yield substantial speedups of model emulation. A simplified hemodynamic model [58] for modeling a human circulatory system is shown in the figure, containing six ODE types for modeling different sub-systems; each type is represented as a different shape in the figure. Each ODE type is unique, and requires a particular ordered set of
operations that differ from the operations of other ODE types (e.g., ODE 1 and ODE 2 in the figure are different types). There can be different numbers of ODEs per type. For example, the left four sub-systems in Fig. 35 consist of unique ODE types with a single ODE per type, while the two sub-systems on the right have four ODEs per type. We can build a custom network of 5 PEs that contains two custom PEs for the types with four ODEs, and three general PEs for the left four ODEs (one of the PEs solves 2 ODEs). A general PE is usually smaller than a custom PE, thus a custom PE is only worth being built if multiple ODEs of one type exists. Note the custom network of PEs has a similar circular connection structure as the circulation model as the communication between model sub-systems remains unchanged.

Heterogeneous PEs substantially changes the synthesis problem due to introducing an enormous design space, involving not just the number of PEs and mapping of ODEs to PEs, but also instantiation of different types of PEs, numbers of each types, and mappings of ODEs to different types. We propose an allocation and binding heuristic for this problem and build
an HDL (hardware description language) code generation tool to generate synthesizable VHDL code from the obtained solution.

Physical system emulation with a custom network of processing elements

This section briefly describes how to convert a physical system into ordinary differential equations (ODEs), using a Weibel lung model [61] with gas exchange as a driver example, and reviews how to use a custom network of PEs to solve ODEs.

Model physical systems using ODEs

Fig. 36 shows the structure of a four generation Weibel lung model with basic gas exchange at the leaf branches. The Weibel lung has a binary tree structure. The first generation represents the airway (or trachea) between the mouth and lung. The 2nd and 3rd generations represent two bronchi and smaller bronchioles, and the fourth generation represents alveoli that handle gas transfer between lung and capillary cells.

The Weibel lung can be captured as RLC circuits, where each branch contains two state
variables (flow and volume). The ODEs of each branch can be written in a general format (for
simplicity we use Ci to represent constant parameters):

\[ V_i' = F_{parent} \cdot C_1 + V_i \cdot C_2 + F_i \]  \hspace{1cm} (1)

\[ F_i' = V_i \cdot C_3 - F_i \cdot C_4 - V_{R\_child} \cdot C_5 - V_{L\_child} \cdot C_6 \]  \hspace{1cm} (2)

Equation (1) and Equation (2) show that the derivative of Vi and Fi are linear functions
of the volume and flow of branch i and neighboring branches. The ODE that governs the
model gas exchange is:

\[ Po_{2,\:\prime} = C_1 \cdot ( P_{co\_2,\:\prime} - Po_{2,\:\prime} ) + C_2 \cdot ( V_i \cdot C_3 - Po_{2,\:\prime} ) \]  \hspace{1cm} (3)

The derivative of O2’s pressure is a linear function of the pressure of CO2, volume of the
alveoli, and the pressure of current O2. The Weibel lung with gas exchange model contains
three types of ODEs, because equations (1) (2) (3) each have a uniquely ordered set of
operations.

To calculate the value of a state variable at a given time, the ODEs can be solved using
iterative solvers such as Euler [6] or Runge-Kutta [10]. Starting from time 0, iterative solvers
move forward in time by a given time step such as 1 ms. At each time step, two major tasks
are performed by the Euler method:

*Evaluate*: Calculate each state variable’s derivative value, e.g.,

\[ V_i' = F_{parent} \cdot C_1 + V_i \cdot C_2 + F_i \]

*Update*: Estimate the value of each state variable for the next time step using the current
values and the derivatives calculated above, e.g., \( V_i = V_i + \frac{d(V_i)}{dt} \cdot h \), where h is the time
step.
Custom network of processing elements

To increase the speed of physical system emulation, we proposed an approach to map the ODEs of a physical system to a homogeneous network of processing elements. For example, Fig. 37 maps the ODEs of the four-generation Weibel lung to a custom network of 5 PEs. The state variables in each of the dotted rectangles are mapped to one PE. The custom network of PEs uses a point-to-point communication scheme between PEs. The structure of a network of PEs is determined by the ODE-to-PE mapping and the ODE data-dependency graph, as in Fig. 37. The state variables and their relevant parameters persist in each PE’s local memory. Such distributed data storage eliminates the bottleneck of a centralized memory.

At each time step, a PE performs the “evaluate” and “update” tasks for the state variables mapped to the PE, known as the PE’s resident variables. In order to evaluate each resident variable, the PE may need the state variables that reside in other PEs. For instance, PE1 needs the latest value of V4 and V5 in order to calculate F2’ according to Equation (2). We call such

Fig. 37. Mapping a four generation Weibel lung model to a custom network of 5 PEs.
state variables dependent variables. At the end of every time step, each PE outputs the latest values of the PE’s resident variables, and stores local copies of the PE’s dependent variables from other PEs. We call this additional task data transfer.

The ODE solving process of a network of PEs is illustrated in Fig. 38. The system has three PEs. The “evaluate” and “update” tasks can be done independently in each PE. All PEs are synchronized at the point when the slowest PE has finished updating the PE’s resident variables (PE2 in this example), to ensure that every state variable is updated. Then communications between PEs are performed according to a static schedule built from the ODE data dependency graph. When all the data-transfer tasks have finished, all PEs are synchronized again for the next time step.

Processing element architecture

Two types of processing element components were proposed in the previous sections to tradeoff performance and flexibility. A general PE can solve any type of ODE. The architecture of the general PE is illustrated in Fig. 25. An instruction is directly mapped to a
control word that controls an input mux, a data-ram and a general purpose arithmetic logic unit (ALU). The PE contains a few input ports and one output port for communication purposes. The data ram stores state variables and parameters mapped to the PE. The ALU calculates the equations by the “compute” and “store” operations. The general PE can handle different types of ODEs by parsing the equations into basic compute operations; thus the general PE is flexible.

Since a physical system often exhibits homogenous properties (only contains a few types of ODEs and each type appears many times), a custom PE that captures the unique structure of a certain ODE type can provide significant speedup (>10x) over a general PE. A custom PE illustrated in Fig. 26. Note that the custom PE has a similar architecture compared to the general PE, except the ALU components is replaced with a custom ODE data-path. The custom ODE data-path is fully customized for one type of ODE. With the fully-pipelined design, the maximal throughput of the custom PE is one ODE per cycle. However, the custom PE is not flexible and can only solve a certain type of ODE. The custom PE also consumes more FPGA resources (1.5-3x) compared to the general PE due the ODE data-path and wider data ram.

Both the general and custom PE use fixed point computation, because a floating point core in an FPGA is inefficient in size and latency. Floating point physicals models are manually converted into fixed point using the method described by Kum [29]. Automated conversion from floating point numbers to fixed point numbers is desired in the future.
Network of Heterogeneous PEs

The network of custom PEs is 5-10x faster than the network of general PEs and consumes less FPGA resources (due to fewer number of PE instances). However, the models examined in previous work were all homogenous models that contained only 1 or 2 types of ODEs.

A heterogeneous physical system model contains many types of ODEs, such as the hemodynamic model in Fig. 35, where each type can have different numbers of ODEs. Using custom PEs where possible can yield performance improvements, however if the number of ODEs of a certain type is small than using general PEs to solve those ODEs is likely more efficient in terms of size because multiple ODEs types can use the same general PE. Thus, a custom network of heterogeneous processing elements can provide improvements in performance and area due to more fine-grained control over the network implementation.

Different PE types

A network of heterogeneous PEs may contain different types of processing elements. A general PE can solve any type of ODE, but performance is slow. A custom PE can only solve one type of ODE, but performance is 10x faster than a general PE. However, in a real physical system, two or more types of ODEs might be tightly coupled. For instance, the Weibel lung model described contains two types of ODEs, for volume and for flow, for each branch, and the volume and flow have data dependencies with each other as shown in Equations (1) and (2). If the system only has custom PEs for flow and volume respectively, there will be too
much communication overhead. We thus extend the custom PE to support multiple ODE types, called multi-type custom PEs.

The idea is to put multiple ODE data-paths of different ODE types into one custom PE. To fully utilize those components, these ODE data-paths can execute in parallel, thus the theoretic throughput of multi-type custom PE can be $\geq 2$ ODEs / cycle. The idea is similar to VLIW (very long instruction word) architectures. The multi-type custom PE is larger than the single type custom PE, because of extra ODE data-paths and wider data ram. With the multi-type custom PE, different types of ODEs that are tightly coupled may be mapped to one custom PE to reduce communication overhead.

**Problem definition**

Given the ODEs of a physical system and different PE options, the question is how to select the number of each PE type and how to map the ODEs to the PEs in order to build a custom network of PEs that efficiently emulates the physical system. We formally define an allocation and binding problem as follows. Given are:

- A set of ODEs in the physical system: $O = \{o_1, o_2, ..., o_n\}$.
- A set of ODE types of the physical system: $T = \{t_1, t_2, ..., t_n\}$.
- A mapping function from $O$ to $T$: $\text{ode2Type}$, e.g., $\text{ode2Type}(o_i)$ returns the ODE type of $o_i$.
- A ODE data-dependency graph of the physical system: $G$, where $G(i, j) = 1$ stands for $o_j$ depends on $o_i$. 


A set of all possible PE types, PT = \{pt_1, pt_2, \ldots, pt_n\}, which includes general PE, single type custom PEs, and all possible multi-type custom PEs.

A Boolean function determines if an ODE to PE mapping is valid: Valid(pt, t). E.g., Valid(pt_i, t_j) = true means t_j (ODE type j) can be mapped to pti (PE type i).

A FPGA resource consumption function for each PE type: RES, e.g., RES(pt_i) returns the FPGA resource consumption for PE type i. (Current we developed a function to estimate the resource consumption of a custom PE based on the ODE types the PE can solve. More accurate result can be obtained by actually synthesis each PE type)

The total available FPGA resource: T_RES.

A computation cost function: CompCost(pt_i, set<ODE>), which returns the number of cycles to evaluate and update a set of ODEs with PE type i. (For a general PE, the computation cost is obtained by parsing the ODEs into basic computation operations. For a custom PE, the computation cost is depends on the number of ODEs of each type)

A communication cost function: CommCost, the total communication cycles of the network of PEs. The communication cost depends on G and the mapping function pe2Ode defined below.

The solution is:

A set of allocated PEs: PE = \{pe_1, pe_2, \ldots, pe_3\}.

A mapping function from PE to PT: pe2Type, e.g., pe2Type(pei) returns the PE type
The ODEs mapped to each PE: pe2Ode, where pe2Ode(pei) return a set of ODEs mapped to pei.

Constraints are:

- FPGA Resource constraint:
  \[
  \sum_i (RES(\text{pe2Type}(\text{pei}))) \leq T \_RES
  \]

- ODE Binding constraint: \( \forall oi \in pe2Ode(\text{pei}) \)
  \[
  \text{valid}(\text{pe2Type}(\text{pei}), \text{ode2Type}(oi)) = \text{true}
  \]

- Each ODE has been mapped to one PE.

The Objective is (minimize cycles per step):

\[
\text{Min} \{\max (\text{CompCost(\text{pe2Type(pei)}, \text{pe2Ode(pei)}))) + \text{CommCost}\}
\]

In other words, the objective is to allocate a valid set of PEs (satisfy the FPGA resource constraint), and to find a valid mapping from ODEs to PEs (satisfy the ODE binding constraint) such that the throughput of the system is maximized.

The throughput of the network of PEs is equal to the number of ODEs divided by the time to emulate one step. Since the number of ODEs is a constant, maximizing the throughput is equivalent to minimizing the time to emulate one step. Assuming the clock frequency is a constant, the time per step is determined by the cycles per step. According to the ODE solving process illustrated in Fig. 38, the total cycles per step is equal to the maximal computation cost of the PEs plus the communication cost, or the objective function.
Allocation and binding heuristic

Choosing PE types

Since a multi-type custom PE can be built for any subset of the ODE types of a physical system, the number of all possible custom PE types increases exponentially with the number of ODE types. To consider all possible custom PEs might be too expensive in terms of algorithm runtime. However, not all ODE types need a custom PE. For instance, if an ODE type contains only a few ODEs (e.g. < 5 ODEs), using a general PE is more size efficient. We thus define a custom PE threshold, such that we only build a custom PE for the ODE types with more ODEs than the custom PE threshold.

The multi-type custom PE is only needed when two or more ODEs are tightly coupled, which is reflected by the number of connections between 2 ODE types in the ODE dependency graph. Our solutions is to keep merging the ODE types that pass the custom PE threshold, using a custom PE merge criteria. The criteria is that two sets of ODE types can be merged only if the number of connections between the two sets is greater than (or equal to) the number of ODEs of any type in the two sets. Fig. 39 shows an example of merging 5 ODE types (Ta, Tb, … Te), the number below each ODE type is the number of ODEs of that type. The line between two types shows the number of connections.

Fig. 39. Merging tightly coupled ODE types
between two ODE types. Note after the merge process, two multi-type custom PE can be built
\(\{Ta, Tb, Tc\}, \{Td, Te\} \). Note \(\{Ta, Tb, Tc\}\) can not merge with \(\{Td, Te\}\) because the number
of connections between these two sets (5) do not satisfy the merge criteria. With the custom
PE threshold and the merge criteria, the custom PE types are chosen based on the ODE
data-dependency graph of the target physical system.

Allocation and binding heuristic overview

Once the PE types have been chosen, we need to allocate the number of each PE type,
and map the ODEs to each PE. The overall structure of the allocation and binding heuristic is
illustrated in Fig. 40. The allocation and binding heuristic includes two major components: an
ODE-to-PE mapper and a PE allocator. The ODE-to-PE mapper tries to find the best
ODE-to-PE mapping for current PE allocation based on the objective function defined earlier.
The PE allocator adjusts the PE allocation based on the feedback from the ODE-to-PE
mapper.

![Fig. 40. Overall structure of the allocation and binding heuristic for the
network of heterogeneous PEs](image-url)

99
The heuristic first generates a random PE allocation, which includes at least one instance for each PE type. Then the ODE-to-PE mapper will try to find the best mapping based on current PE allocation, and return the number of computation cycles for each PE to the PE allocator. The PE allocator will adjust the number of each PE type based on the mapper’s feedback, and generate a new PE allocation. This iterative improving process will terminate when the mapper cannot find a better solution.

ODE-to-PE mapper

The ODE-to-PE mapping algorithm is based on the mapping algorithm for the network of general PEs in Section 3, with the objective as the cost function. The basic idea of the mapping algorithm is to move one ODE from one PE to another PE, which is called a neighbor mapping. To speedup the mapping algorithm, two types of neighbor functions are developed to generate neighbor mapping that has higher changes of improving the solution.

The “performance neighbor function” tries to balance the load (or total number of cycles) among PEs by moving ODEs from a heavily loaded PE to a light loaded PE. The “size neighbor function” tries to utilize the spatial locality of the ODEs, and group ODEs nearby to one PE. Thus the total number of connections and communications among the PEs would be reduced.

To adapt the original mapping algorithm to networks of heterogeneous PEs, we added constraints when generating the neighbor mappings such that the neighbor mapping still satisfies the ODE binding constraint (custom PEs can only solve a sub-set of ODE types).
ODE-to-PE mapper will output the number of cycles for each PE for the best mapping has been found.

**PE allocator**

The PE allocator adjusts the number of PEs for each PE type based on the feedback from the ODE-to-PE mapper. The PE adjustment algorithm is shown as follows:

**Step 1:** Choose top K% PEs with most loads (#Cycles).

**Step 2:** Allocate a new PE (same PE type) for each PE chosen in step 1, move half of the ODEs mapped to the original PE to the new PE.

**Step 3:** Remove the PE with least loads (#Cycles), and randomly redistribute the ODEs mapped to the removed PE to other PEs. (Satisfying the ODE binding constraint)

**Step 4:** Repeat step 3 until the resource constraint is satisfied.

The basic idea is to look at the number of cycles for each PE, and duplicate the most heavily loaded PEs. The allocator allocates new PEs for top 10% loaded PEs in this work. We notice two possibilities why a PE is heavily loaded. The first possibility is that this PE type doesn’t have enough instances because some ODEs can only be mapped to this PE type. The second possibility is that this PE type is very efficient in solving the ODEs, thus many ODEs are mapped to it. Either possibility means that more instances of this PE type are required. However, adding new PEs may violate the FPGA resource constraint, thus we remove the least loaded PEs until the resource constraint is satisfied.

Since the PE adjustment algorithm depends on the ODE-to-PE mapping results, the
quality of the mapping algorithm is critical for the entire allocation and binding heuristic. We let the ODE-to-PE mapper run long enough (200K-500K iterations) in order to produce a good solution.

HDL code generation

Once the processing elements have been allocated, and the ODEs are mapped to the PEs, the next task is to generate HDL code for the entire network. The code generation includes two steps:

1. Generate PE components for each PE type.
2. Generate and inject instructions for each PE instance, and connect them at top level.

Since the general PEs only differ in the data-ram size, instruction ram size, and input ports number, we developed a script to generate PE components with all possible combinations of the three parameters.

A custom PE contains unique ODE data-paths designed for the ODE types that the PE can solve. Thus we cannot pre-generate all possible custom PE components. Instead, we developed an automatic ODE data-path generator to facilitate custom PE generation. The ODE data-path generator reads an ODE, and outputs a fully pipelined ODE data-path component for calculating this ODE. The data-path generator parses the input ODE into an expression tree, and uses an ASAP (as-soon-as-possible) scheduling algorithm [49] to schedule the operations. The data-path generation task can be done with a high-level synthesis tool.
We also developed a custom data-ram generator to generate data-ram components with different number of ports and depth, because different ODE data-path may require different custom data ram. With the ODE data-path generator and the custom data-ram generator, a custom PE component of any type can be generated on the fly.

The PE instructions can be divided into two parts: (1) “evaluate and update” instructions, and (2) “data transfer” instructions, as shown in Fig. 38. The “evaluation and update” instructions are scheduled independently on each PE instance based on the ODEs mapped to the PE, while the “data transfer” instructions are scheduled globally.

The general PE handles the “evaluate and update” instructions by parsing the ODEs into basic instructions for the general purpose ALU component. The general instructions are then converted into the control word by a general instruction assembler. The custom PE schedules the “evaluate and update” instructions by an instruction scheduler, and converts the instructions into control words by a custom instruction assembler.

The general PE and the custom PE have the same input/output interface. At each clock cycle, one PE can output a variable and store a variable concurrently. Thus the “data transfer” instructions are handled globally with a communication scheduler. The inputs to the communication scheduler are the ODE data-dependency graph (G) and the each PE’s ODE set (pe2Ode). The scheduler tries to schedule as many “data transfer” instructions as possible if those instructions do not conflict with each other (each PE can only output one variable and store one variable at one cycle). The “data transfer” instructions are appended to the back of the “evaluate and update” instructions for each PE. The complete instructions are then
injected into each PE from the generic interface for the instruction ram.

The structure of the top level network is determined by the ODE data-dependency graph and the ODE-to-PE mapping. The code generation tool will finally connect all PE instances according to the network structure, and output synthesizable VHDL files.

Experimental results

This section describes experimental results using five physical system models as benchmarks. The section compare our network of heterogeneous PEs with a high-level-synthesis approach including regularity extraction, networks of general/custom PEs, a GPU implementation, and a modern desktop processor.

Performance numbers are in milliseconds (ms) and represent the time for an implementation to execute one second of simulated time. Throughout this section, we execute the physical system using the Euler solver with a 0.01 ms step.

All FPGA based 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 [62] for synthesis. Note that this work is not limited to a particular FPGA or synthesis tool.

Physical system models

The five physical system models are all physiology models, and each contains more than one type of ODE. The five models have different connection structures and contain different
numbers of ODE types.

Weibel lung: 11 generation Weibel lung model that contains 4094 ODEs and calculates internal lung states. The Weibel lung has two types of ODEs (flow and volume) as discussed in earlier. The Weibel lung has a binary tree structure.

Neuron network: a neuron network model [56] contains a number of neuron cells for modeling a neuron system in the brain. The neuron network contains three types of ODEs for membrane potentials, channel gating and synaptics. We use a 40 x 40 2-dimensional neuron network, which contain 1600 neurons or 4800 ODEs. The neurons are connected in a 2-dimensional mesh network.

Weibel lung with gas exchange: 11 generation Weibel lung model connected with 500 capillary cells connected with the leaf branches. The model contains 3 types of ODEs as, and contains 4594 ODEs and has a binary tree structure.

Hemodynamic: the hemodynamic model [58] is a system circulation model which contains pulmonary tissues, systemic tissues and left/right ventricles. The model contains 36 types of ODEs for modeling different types of organs/cells. Within the 36 types of ODEs, 12 ODE types each with more than 100 ODEs. The remaining 24 ODE types each with only 1 ODE. The hemodynamic model contains 4800 ODEs, and has a circular connection structure.

Weibel lung with hemodynamic model: We combine the Weibel lung model with the hemodynamic model, because the pressure of the lung is an input to the hemodynamic model. We use a 10 generation Weibel lung with the hemodynamic model. The entire model contains 4266 ODEs, and 38 types of ODEs. The model has a hybrid connection structure with a
binary tree connected with a circular structure.

Heterogeneous networks vs. HLS and homogeneous networks

We implemented the ODE solver for each model using a commercial high-level synthesis tool. Since physical systems exhibit much regularity, we incorporate the idea of regularity extraction [51] into the design. Each ODE type represents a sub-pattern in the system, thus we generate a fully pipelined ODE data-path to compute each ODE type with the HLS tool.

To eliminate the memory bottleneck of the system, we manually optimized the communication structure of the HLS approach (Fig. 41). The state variables of a physical system are persisted in the distributed registers. The system contains different types of ODE

![Diagram of ODE solver](image-url)

**Fig. 41.** The overall architecture of the ODE solver using an HLS tool with regularity extraction.

---

4 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 HLS tools is highly encouraged; we will provide our models for such purposes upon request.
data-paths. Each ODE data-path is mapped with a sub-set of ODEs of the physical system, thus each ODE data-path is responsible of updating multiple registers as shown in the figure. To utilize the spatial locality of the ODEs, one input mux is shared by all ports of the ODE datapath (shown in the figure) using time multiplex. Using time multiplex will decrease the performance. However, using the shared input mux significantly reduces the number of wires and results in synthesizable designs (one mux per port cannot be fully synthesized due to a large number of wires). We use the same allocate and binding heuristic to allocate the ODE data-path, and balance the number of ODEs in each ODE data-path. Since each ODE data-path is mapped with multiple ODEs (usually >50 ODEs), the shared input mux is often very large (64-256 inputs).

The networks of heterogeneous PEs are generated by the PE allocation and binding heuristic, and the code generation tool discussed in the previous section. The total algorithm runtime to generate the VHDL code for each model is about 5-10 minutes. For comparison purpose, we implemented the 5 models using networks of general PEs. We also included the results for networks of single-type custom PEs. All FPGA results are fully synthesized and implemented on the target FPGA.

The summary of the results is shown in Table VI. We recorded the resource utilization of each approach. For easy comparison purposes via a single number, we define an equivalent LUTs term as is commonly done for FPGA designs [39]. By implementing of equivalent DSP and BRAM components using LUTs, we assign a DSP unit a value of 250 LUTs and a BRAM of 360 LUTs. The bottleneck of each design is highlighted with underlines.
Table VI. Synthesis results of networks of heterogeneous PEs, HLS, and general/single-type custom PEs.  
PES: the number of PE or ODE datapath in the design.  
Cycles: total clock cycles to compute one time step.  
Freq: Maximum clock frequency after place and route.  
Syn. time: total synthesis time (including place and route) of a design. The underlined entries show the FPGA resource bottleneck of a design, i.e., the issue that prevents further improvement via more PEs.

<table>
<thead>
<tr>
<th></th>
<th>Luts</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Equiv. Luts</th>
<th>PEs.</th>
<th>Cycle (Mhz)</th>
<th>Perf. (ms)</th>
<th>Syn. Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HLS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weibel</td>
<td>101,031</td>
<td>131</td>
<td>245</td>
<td>209,441</td>
<td>67</td>
<td>296</td>
<td>112</td>
<td>264</td>
</tr>
<tr>
<td>Neuron</td>
<td>139,118</td>
<td>156</td>
<td>190</td>
<td>242,778</td>
<td>80</td>
<td>370</td>
<td>102</td>
<td>363</td>
</tr>
<tr>
<td>Weibel_gas</td>
<td>106,008</td>
<td>144</td>
<td>225</td>
<td>214,098</td>
<td>72</td>
<td>332</td>
<td>110</td>
<td>301</td>
</tr>
<tr>
<td>Hemodynamics</td>
<td>82,007</td>
<td>145</td>
<td>246</td>
<td>195,707</td>
<td>69</td>
<td>420</td>
<td>131</td>
<td>321</td>
</tr>
<tr>
<td>Weibel_hemo</td>
<td>79,735</td>
<td>143</td>
<td>230</td>
<td>188,715</td>
<td>75</td>
<td>340</td>
<td>115</td>
<td>295</td>
</tr>
<tr>
<td><strong>General PEs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weibel</td>
<td>89,761</td>
<td>396</td>
<td>396</td>
<td>331,321</td>
<td>396</td>
<td>184</td>
<td>130</td>
<td>142</td>
</tr>
<tr>
<td>Neuron</td>
<td>74,632</td>
<td>294</td>
<td>294</td>
<td>253,972</td>
<td>294</td>
<td>290</td>
<td>150</td>
<td>193</td>
</tr>
<tr>
<td>Weibel_gas</td>
<td>78,178</td>
<td>281</td>
<td>281</td>
<td>249,588</td>
<td>281</td>
<td>302</td>
<td>105</td>
<td>288</td>
</tr>
<tr>
<td>Hemodynamics</td>
<td>75,311</td>
<td>281</td>
<td>281</td>
<td>246,721</td>
<td>281</td>
<td>364</td>
<td>190</td>
<td>192</td>
</tr>
<tr>
<td>Weibel_hemo</td>
<td>74,956</td>
<td>281</td>
<td>281</td>
<td>246,366</td>
<td>281</td>
<td>326</td>
<td>165</td>
<td>198</td>
</tr>
<tr>
<td><strong>Custom PEs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weibel</td>
<td>48,579</td>
<td>56</td>
<td>205</td>
<td>119,989</td>
<td>56</td>
<td>221</td>
<td>129</td>
<td>171</td>
</tr>
<tr>
<td>Neuron</td>
<td>49,762</td>
<td>71</td>
<td>129</td>
<td>107,572</td>
<td>53</td>
<td>397</td>
<td>123</td>
<td>323</td>
</tr>
<tr>
<td>Weibel_gas</td>
<td>50,491</td>
<td>62</td>
<td>202</td>
<td>123,311</td>
<td>61</td>
<td>253</td>
<td>123</td>
<td>206</td>
</tr>
<tr>
<td>Hemodynamics</td>
<td>50,743</td>
<td>111</td>
<td>386</td>
<td>187,203</td>
<td>111</td>
<td>185</td>
<td>143</td>
<td>129</td>
</tr>
<tr>
<td>Weibel_hemo</td>
<td>72,828</td>
<td>154</td>
<td>555</td>
<td>266,518</td>
<td>175</td>
<td>84</td>
<td>167</td>
<td>50</td>
</tr>
<tr>
<td><strong>Heterogeneous PEs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weibel</td>
<td>65,036</td>
<td>160</td>
<td>560</td>
<td>262,636</td>
<td>80</td>
<td>52</td>
<td>219</td>
<td>24</td>
</tr>
<tr>
<td>Neuron</td>
<td>63,458</td>
<td>133</td>
<td>462</td>
<td>226,838</td>
<td>67</td>
<td>69</td>
<td>204</td>
<td>34</td>
</tr>
<tr>
<td>Weibel_gas</td>
<td>58,800</td>
<td>141</td>
<td>494</td>
<td>233,060</td>
<td>86</td>
<td>72</td>
<td>222</td>
<td>32</td>
</tr>
<tr>
<td>Hemodynamics</td>
<td>48,652</td>
<td>118</td>
<td>444</td>
<td>202,132</td>
<td>142</td>
<td>56</td>
<td>250</td>
<td>22</td>
</tr>
<tr>
<td>Weibel_hemo</td>
<td>58,795</td>
<td>141</td>
<td>494</td>
<td>233,055</td>
<td>86</td>
<td>71</td>
<td>213</td>
<td>33</td>
</tr>
</tbody>
</table>
Bottleneck of each approach

We tried to implement the fastest circuit for each design. We notice that performance each design may be constrained by one of the FPGA resource (LUTs, DSPs, BRAMs), or by the clock frequency.

For instance, the HLS designs are constrained by the available LUTs (the FPGA has 150,720 LUTs), because each ODE datapath requires a large input mux as shown in Fig. 41, requiring many LUTs. The networks of general PEs are mainly constrained by BRAMs (the FPGA has 406 BRAMs), because each general PE requires a BRAM instance. The networks of single-type custom PEs are mainly constrained by the clock frequency (or the number of wires in the system). Putting more single-type custom PEs into the network will result in long routing time and lower clock frequency, thus adding PEs, while achieving more parallelism, yields overall performance decrease due to the slower clock frequency.

The networks of heterogeneous PEs are mainly constrained by DSPs (the FPGA has 768), because the custom ODE datapath (especially multi-type custom PEs) requires more DSPs than a general PE. Since the custom ODE datapaths are fully pipelined, these DSPs are highly utilized in the design. Thus the networks of heterogeneous PEs have the best performance. Note that the FPGA resource utilization of each design may not be close to the upper-bound of the total FPGA resource, because we also consider the clock frequency. Increasing the size of each design will decrease the clock frequency, thus the overall performance will decrease.
Comparison with high level synthesis

Compared to the HLS approach, the network of heterogeneous PEs uses around 40% fewer LUTs, a comparable number of BRAMs, and 2x more DSPs. In terms of equivalent LUTs, the network of heterogeneous PEs uses on average 10% more FPGA resources than the HLS approach.

The performance of the network of heterogeneous PEs approach is on average 10.8x (9x-14x) faster than the HLS approach because the network of heterogeneous PEs makes better usage of memories and computational components like DSPs. Since a large input mux consumes many LUTs in the HLS design, the HLS approach can only put a limited number of ODE data-paths, which limits the performance of the HLS approach. The time multiplex of the shared input mux further decreases the performance of the HLS approach. Our approach using encapsulated processing elements better utilizes the spatial locality of a physical system. The network of heterogeneous PEs also obtained higher clock frequencies and shorter synthesis times due to fewer wires in the design.

Comparison with networks of general/single-type custom Pes

The network of general PEs consumes on average 15% more equivalent LUTs compared to networks of heterogeneous PEs due to having more PE instances. The performance of the network of heterogeneous PEs is on average 7x (6x-8.8x) faster than the network of general PEs, because the former contains custom PEs that solve certain types of ODEs faster than the general PEs. The networks of heterogeneous PEs also obtained 50% faster clock frequency,
due to containing fewer PE instances and fewer wires in the network.

The networks of single-type custom PEs consumed on average 30% fewer equivalent LUTs than the networks of heterogeneous PEs because the former contained fewer PE instances. The reason is that all five models contain tightly coupled ODEs of different types, and the tightly coupled ODEs are split into different single-type custom PEs. Thus the network requires more connections and communications. The wire congestion problem limits the clock frequency and the number of single-type custom PEs in the network.

The networks of heterogeneous PEs contain multi-type custom PEs, which reduce the number of connections and communication in the network. The network of heterogeneous PEs may also contain general PEs in case the number of ODEs of a type is too small. For instance, the hemodynamic model has 24 ODE types with only one ODE. Assigning the 24 individual ODEs to a few general PEs instead of creating a custom PE for each type is reasonable. With the multi-type custom PEs and general PE options, the network of heterogeneous PEs is on average 6x (1.5x-9.4x) faster than the network of single-type custom PEs. Note the performance of single-type custom PE depends on the model; while our approach provides much larger design space, and gives much better solution for certain models. (e.g., Neuron)
Heterogeneous network vs. CPU and GPU

We further compared the network of heterogeneous PEs with a modern desktop processor and a GPU. The configurations of the processor and the GPU are listed as follows:

- **PC**: C code on a 3.06 GHz Intel I7-950 quad-core processor with 16G DDR3 RAM, compiled with Microsoft VS2010 with –O3 flag
- **GPU**: CUDA C code on a 763 MHz NVIDIA GTX460 Fermi GPU with 336 CUDA cores, compiled using nvcc with –O3 flag.

A fixed-point implementation was used for all test cases for a fair comparison. The C code on the desktop was manually optimized. The time to optimize each model is around 1-3 hours, which is comparable to the synthesis time of the network of heterogeneous PEs. We first obtained the single threaded performance (PC(1)) for the PC, and calculated an optimistic performance bound for multi-cores (PC(4)) by dividing the single threaded result by the number of cores.

We implemented a GPU kernel function for calculating the ODEs for each model. The kernel function may contain multiple branches for different ODE types as illustrated in Fig. 42(a). Another approach is to implement different GPU functions for each ODE type, shown in Fig. 42(b). However, the second approach executes each function sequentially, which is slower than the first approach. We thus decided to use the first approach that better utilizes the resource on the GPU.

The ODE kernel function is executed on multiple GPU blocks, and each block contains multiple threads. We tuned the number of blocks and the number of threads to obtain the best
performance. We also considered the coalesced access pattern when reading the global memory on the GPU. Since global memory access is more expensive than accessing the shared memory within each GPU block. We utilized the spatial locality of each model by mapping related ODEs to one GPU block. Thus nearby state variables are loaded to shared memory to reduce global memory accesses. The total GPU implementation time for each model is around 2-4 hours, which is comparable to the synthesis time of the custom network of heterogeneous PEs. To ensure our GPU implementations are good, we asked a student who has many GPU coding experience to optimize our CUDA code. The student further improved the results by 5-15% mainly by utilizing the coalesced access for the global memory. We used the further optimized code in the experiment.

The performance of each approach is illustrated in Fig. 43. The single threaded PC failed the real-time constraint for four models, while the optimal multi-threaded version runs each model 2-4x faster than real-time. The average speed of the network of heterogeneous PEs is 45.3X (36x-60x) faster than PC(1), and 11.3x (9x-15x) faster than PC(4). Note the performance is the pure execution time of each model. Further monitoring and debugging logic will cause extra overhead. The network of heterogeneous PEs gives more slack than the
C implementation on the PC, which provides opportunity for real-time tracing of model state variables. The network of heterogeneous PEs is also more preferable in case fast-forward emulation is needed.

The GPU performs comparably to the multi-threaded PC approach when the number of ODE types is small (e.g., Weibel lung and Neuron models). The model with larger number of ODE types requires more branches in the kernel function, which decreases the performance.

The network of heterogeneous PEs is on average 20.7X (13.7X–29X) faster than the GPU implementation. The major advantage of our approach is the custom communication network.

For the target GPU, the only method to synchronize the GPU blocks is through a new function invocation, according to the CUDA programming guide [15]. Thus the frequent function invocation (105 times per second) greatly impacts the GPU’s performance. According to the profile information, the pure function invocation overhead consumes 30% -
60% of the total GPU execution time.

We included some approximate cost comparison for different approaches, in particular to acknowledge that FPGA platforms are costlier than PCs and GPUs. We consider the minimal required components for each platform in order to performance the emulation. The approximate cost of each platform is as follows:

1. CPU (I7-950 + Intel X58 board): $480
2. GPU (NV GTX460 + I3-540 processor + H55 board) $380
3. FPGA (Xilinx Virtex6 240T-2 board): $1800

We consider a normalized speedup term, namely: (speedup over real-time) / cost. The normalized speedup of each approach is shown in Fig. 44. The network of heterogeneous PEs obtained the best normalized speedup (3x over the I7-950 CPU and 4.4x over the GTX460 GPU), because of much higher emulation speed. The FPGA based solution has other
advantages, such as smaller device size and the flexibility to build custom interfaces to the physical world [41].

Discussion

We introduced a network of heterogeneous processing-elements for real-time emulation of complex physical systems. We developed an automated tool to generate a custom PE for a given set of ODEs. We developed an automatic allocation and binding heuristic for allocating different types of PEs, and mapping the ODEs of the target physical system to the PEs. We created a tool to generate synthesizable VHDL code from the allocation and binding results.

Comparing to a commercial high-level synthesis tool with regularity extraction, the networks of heterogeneous PEs were on average 10.8x faster and of comparable size. Compared to the networks of general and single-type custom PEs, the networks of heterogeneous PEs were on average 7x and 6x faster. The network of heterogeneous PEs was also on average 45x faster than a single threaded I7-950 processor, and 20x faster than an Nvidia GTX460 GPU given comparable implementation time. The speedups are due to the custom communication structure and also due to the custom datapath for each type of ODE.
Section 6: Conclusions

We introduced a systematic approach to execute ordinary differential equations (ODEs) of a physical system on FPGAs. The ODEs are mapped to a set of processing elements (PEs). Each PE is responsible for calculating a subset of the ODEs, and handles the communication with other PEs. The interconnection of the processing elements is customized for the target physical system, based on the data dependencies of the ODEs. Thus the custom network is efficient in terms of size and performance.

Each processing element can either be a general PE or a custom PE. A general PE can solve any types of ODEs, but is 5-10x slower than a custom PE. A custom PE can only solve one or a few types of ODEs, but is 5-10x faster than a general PE. We developed an automatic allocation and binding tool to search the design space, and to generate synthesizable VHDL files. The experimental results show the custom networks of processing elements were on average 12x faster than a quad-core I7-950 desktop processor running at 3.07 GHz, 20x faster than an Nvidia GTX460 GPU running at 763 MHz, and 10x faster than the circuits generated by a commercial high-level-synthesis tool. Our approach ran the benchmark models on average 35x faster than real-time, while other approaches could not or just be able to meet the real-time constraint.
References

Massively-parallel simulation of biochemical systems. In Proceedings of Massively  
Parallel Computational Biology on GPUs, Jahrestagung der Gesellschaft fÃœr  
Informatik e.V., LÃœbeck, Germany, 2009.

http://www.amd.com/usen/Processors/ProductInformation/0,,30_118_8825,00.html

Address Traces Using Microcode. Proceedings of the 13th International Symposium on  
Computer Architecture.

generation for solvers of cardiac cellular membrane dynamics in GPUs, Engineering in  
Medicine and Biology Society (EMBC), 2010


Inc. New York, New York.


doi: 10.1109/LES.2011.2170152


[38] METIman. 2012 available online: http://www.meti.com/products_ps_metiman.htm


doi: 10.1109/TVLSI.2007.893581


[42] Mosegaard, J., Sørensen, T.S., Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU. Eurographics Virtual Environments Workshop 2005; 105-10


