Synthesis of Custom Networks of Heterogeneous Processing Elements for Complex Physical System Emulation

Chen Huang[1], Bailey Miller[1], Frank Vahid[1, 3], Tony Givargis[2]
[1] Department of Computer Science and Engineering – University of California, Riverside
{chuang, bmiller, vahid}@cs.ucr.edu
[2] Center for Embedded Computer Systems – University of California, Irvine; givargis@uci.edu
[3] Also with the Center for Embedded Computer Systems – University of California, Irvine

ABSTRACT
Physical system models that consist of thousands of ordinary differential equations can be synthesized to field-programmable gate arrays (FPGAs) for highly-parallelized, real-time physical system emulation. Previous work introduced synthesis of custom networks of homogeneous processing elements, consisting of processing elements that are either all general differential equation solvers or are all custom solvers tailored to solve specific equations. However, a complex physical system model may contain different types of equations such that using only general solvers or only custom solvers does not provide all of the possible speedup. We introduce methods to synthesize a custom network of *heterogeneous* processing elements for emulating physical systems, where each element is either a general or custom differential equation solver. We show average speedups of 45x over a 3 GHz single-core desktop processor, and of 11x and 20x over a 3 GHz four-core desktop and a 763 MHz NVIDIA graphical processing unit, respectively. Compared to a commercial high-level synthesis tool including regularity extraction, the networks of heterogeneous processing elements were on average 10.8x faster. Compared to homogeneous networks of general and single-type custom processing elements, heterogeneous networks were on average 7x and 6x faster, respectively.

Categories and Subject Descriptors
B.5.2 [**Design Aids**]: Automatic synthesis;
C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms
Design, Performance, Experimentation

Keywords
Real-time emulation, field-programmable gate array (FPGA), ordinary differential equation (ODE) solving, physical models, cyber-physical systems, differential equation synthesis, high-level synthesis, system-level synthesis, processing elements.

1. INTRODUCTION
Fast accurate digital emulation of physical systems is useful in various cyber-physical systems [11]. For instance, a complicated patient simulator that emulates a human's circulatory, respiratory, and nervous system can be used for training clinicians [15]. Compared to mechanical systems used for emulation purposes, the parameters of a digital model can be easily modified. For instance, a digital lung model's parameters can be changed for capturing varied lung pathologies, which is more flexible than typical lung emulation approaches using mechanical balloons.

Most physical systems can be modeled with ordinary differential equations (ODEs). A heterogeneous physical system may contain thousands of ODEs. Because ODEs for real physical systems typically cannot be solved exactly, simulations/emulations typically step forward using small time steps and estimate the next values of all variables using the current variable values and the slope of the variables' function at that time (the slope coming from the derivative terms). Such iterative solvers typically solve 1000 times per second or faster to achieve sufficient accuracy, thus real-time emulation of a physical system can be computationally intensive. A modern multi-core desktop computer may not satisfy the real-time constraint, because of the mostly-sequential computation of a CPU (centralized processing unit). GPU (graphical processing unit) platforms perform better than CPUs in emulating a physical system because of more parallel computation resources. However, we have found that the memory architecture of a GPU may not match the communication pattern of a physical system model, thus greatly limiting the performance of a GPU.

We previously [7][8] proposed a network of *homogeneous* processing-elements on FPGAs 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, we used a custom network of *general* PEs [7], with each general PE able to solve any ODEs, yielding 10x-20x speedups over a single-core Intel I7 desktop processor running at 3.07 GHz. Later, we introduced a custom network of *custom* PEs [8], where a single custom PE was first created to best

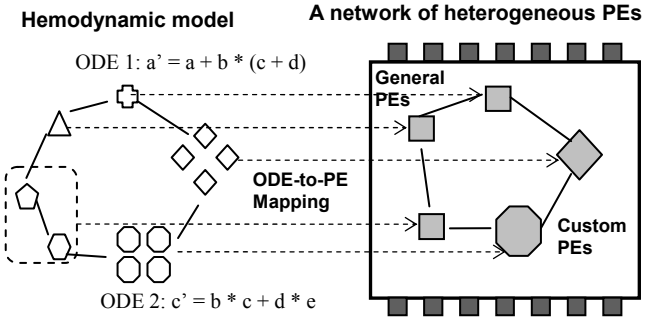

Fig. 1. Synthesizing a hemodynamic model into a network of heterogeneous PEs on an FPGA. Each shape type in the model represents a different type of ODE. Two simplified ODEs are shown for illustration purpose

CODES+ISSS'12, October 7–12, 2012, Tampere, Finland.

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 this paper, we propose to create a custom network of *heterogeneous* PEs consisting of general PEs and multiple custom PEs together, as shown in Fig. 1. 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 [25] 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. 1 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). 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.

This paper is organized as follows. Sections 2 reviews related works. Section 3 reviews a processing element network approach. Section 4 introduces an allocation and binding heuristic for a network of heterogeneous PEs and HDL code generation. Section 5 summarizes experimental results, and Section 6 concludes.

2. RELATED WORK

Simulation of physical system models has been studied extensively in the past decades. Tools have been developed to facilitate physical system simulation, such as Matlab [13], JSim [9], LabView [18], etc. These tools usually aim at producing accurate results rather than real-time emulation, commonly executing models much slower than real time.

Researchers have sought methods to speed up simulation by using multi-core processors and GPUs (Graphic Processing Units). For example, a 768-core SGI super-computer executed a 2-billion equation heart model, simulating 0.4 ms in 2 hours [14]. A Flaim heart model was executed on an Nvidia [19] GTX 295 GPU, running 30x faster than OpenMP running on an Intel I7 quad-core processor at 2.93 GHz, but still much slower than real-time, with execution of one heartbeat (300 ms) requiring 7.7 minutes. Amorim [1] used an Nvidia GTX 285 GPU to solve cardiac membrane dynamics, claiming 22-86x speedups over an Intel I7 quad-core processor at 2.8 GHz, though still running >100x slower than real-time.

FPGAs have been used for emulating physical systems. Yoshimi [29] obtained 100x speedups using an FPGA for fine-grained biological emulation compared to a single-core processor. Salwinski and Eisenberg [22] used an FPGA to speedup fine-grained intra-cellular simulation, showing that an FPGA could hold 500 reactions related to gene expression. However, these implementations are mostly manually designed and optimized, which requires significant design time. High-level synthesis (HLS) tools are another way to implement physical system solvers on an FPGA. Various tools have evolved that perform synthesis from C code, such as Stream-C [6], AutoESL [3], ROCCC [26], SynphonyC [23], etc. We compare our approach to a commercial HLS tool. Due to the regularity of most physical systems, we incorporate the idea of regularity extraction [21] in the design for a more fair comparison.

We previously [7][8] proposed a homogeneous processing element network approach for physical system emulation, and built a compiler to convert a model specification to synthesizable VHDL code. The previous custom PE networks contained homogenous processing elements, either general PEs, or custom PEs, but not both. The work in this paper allows different PE types in the network, greatly expanding the solution space and thus requiring additional exploration heuristics, but yielding substantial speedups.

3. PHYSICAL SYSTEM EMULATION WITH A CUSTOM NETWORK OF PROCESSING ELEMENTS

3.1 Modeling physical systems using ODEs

Fig. 2 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 [7], 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 C_i to represent constant parameters):

$$V_i' = F_{parent} \cdot C_1 + V_i \cdot C_2 + F_i \quad (1)$$

$$F_i' = V_i \cdot C_3 - F_i \cdot C_4 - V_{R\_child} \cdot C_5 - V_{L\_child} \cdot C_6 \quad (2)$$

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:

$$Po2_i' = C_1 \cdot (Pco2_i - Po2_i) + C_2 \cdot (V_i \cdot C_3 - Po2_i) \quad (3)$$

The derivative of O_2's pressure is a linear function of the pressure

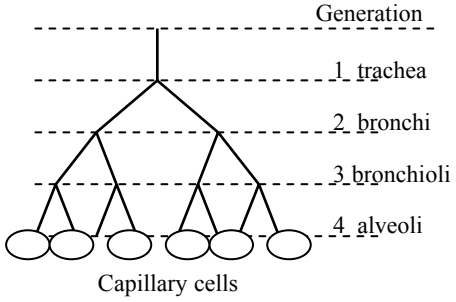

Fig. 2. A four generation Weibel lung with gas exchange model

of CO_2, volume of the alveoli, and the pressure of current O_2. 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 [2] or Runge-Kutta [4]. 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., *Vi = Vi + d(Vi) / dt * h*, where h is the time step.

3.2 Physical system ODE properties

The ODEs of a physical system often exhibit spatial locality similar to the system's physical counterparts. For instance, the volume of each branch in the Weibel lung is determined only by itself and the neighboring branches.

An *ODE data-dependency graph* [8] captures data-dependencies among the state variables. Each node represents one state variable (or a set of variables), while each edge represents the data dependency between two nodes. For instance, Fig. 3 shows the ODE data-dependency graph of the four-generation Weibel lung. Each node represents a branch which contains two state variables (volume and flow). Note the ODE data-dependency graph has a similar binary tree structure as the Weibel lung model. Since a state variable in a physical system only depends on the variable's neighboring variables, the ODE data dependency graph of a physical system is often very sparse. The number of edges increases linearly with the number of nodes.

3.3 Custom network of processing elements

To increase the speed of physical system emulation, we previously [7][8] proposed an approach to map the ODEs of a physical system to a homogeneous network of processing elements. For example, Fig. 3 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 PE network is determined by the ODE-to-PE mapping and the ODE data-dependency graph, as in Fig. 3. 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.

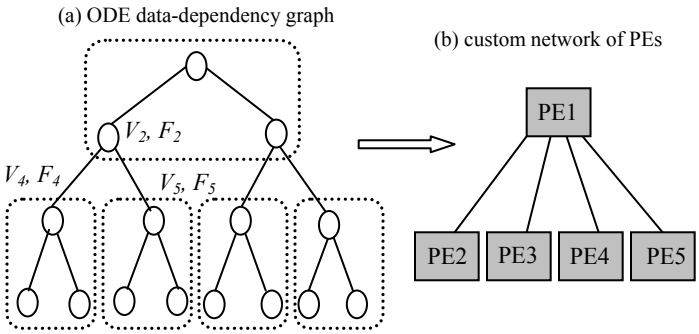

Fig. 3. Mapping a four generation Weibel lung model to a custom network of 5 PEs.

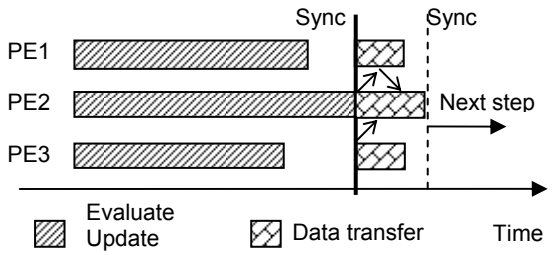

Fig. 4. ODE solving process in a network of 3 PEs

At each time step, a PE performs the "evaluate" and "update" tasks for the state variables mapped to the PE, which are 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 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. 4. 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.

3.4 Processing element architecture

Two types of processing elements were proposed in the previous works that trade off performance and flexibility. A general PE [7] can solve any type of ODE. The architecture of the general PE is illustrated in Fig. 5. 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 like ADD, MUL, etc.; 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

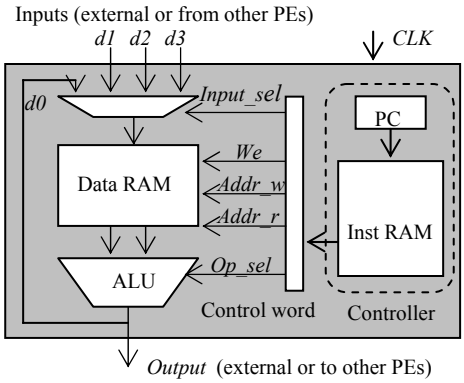

Fig. 5. General PE architecture.

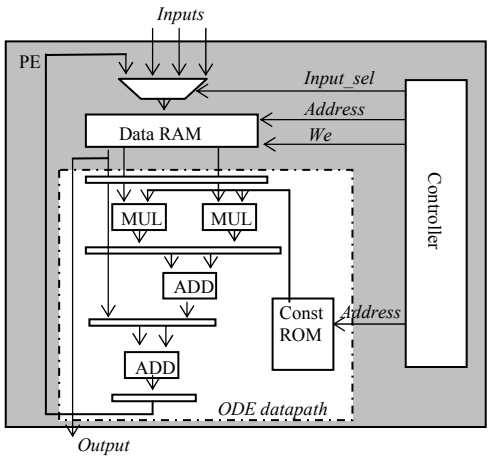

Fig. 6. Custom PE architecture.

times), a custom PE that captures the unique structure of a certain ODE type can provide significant speedup (>10x) over a general PE [8]. A custom PE for equation (1) is illustrated in Fig. 6. Note that the custom PE has a similar architecture compared to the general PE, except the ALU components is replaced with a *custom ODE datapath*. The custom ODE datapath is 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 datapath 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 [10]. The fixed-point results are nearly identical compared to a double precision floating point implementation (the relative error is usually within 0.5%) [7]. Automated conversion from floating point numbers to fixed point numbers is desired in the future.

4. NETWORK OF HETEROGENEOUS PES

The custom network of custom PEs is 5-10x faster than the custom network of general PEs and consumes fewer 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. 1, where each ODE 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 ODE types can use the same general PE. Thus, a custom network of *heterogeneous* processing elements, both general and custom, can provide improvements in performance and area due to more fine-grained control over the network implementation.

4.1 Different PE types

A custom 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 in Section 3.1 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 excessive 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 datapaths of different ODE types into one custom PE. To fully utilize those components, these ODE datapaths can execute in parallel, thus the theoretical throughput of a multi-type custom PE can be >= 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 datapaths 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.

4.2 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 = {o1, o2, ..., on}*.
- A set of ODE types of the physical system: *T = {t1, t2,.., tn}*.
- A mapping function from *O* to *T*: o*de2Type*, e.g., *ode2Type(oi)* returns the ODE type of *oi*.
- An ODE data-dependency graph of the physical system: *G*, where *G(i, j) = 1* stands for *oj* depends on *oi*.
- A set of all possible PE types, *PT = {pt1, pt2, ..., ptn},* 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(pti, tj) = true* means *tj* (ODE type *j)* can be mapped to *pti* (PE type *i)*.
- A FPGA resource consumption function for each PE type: *RES*, e.g., *RES(pti)* 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(pti, 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 = {pe1, pe2, ..., pe3}*.

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

Constraints are:

- FPGA resource constraint: $\sum_i (RES(pe2Type(pei))) <= T\_RES$
- ODE Binding constraint: $\forall oi \in pe2Ode(pei)$
- $valid(pe2Type(pei), ode2Type(oi)) = true$
- Each ODE has been mapped to one PE.

The objective is (minimize cycles per step):

Min {max (CompCost(pe2Type(pei), pe2Ode(pei))) + 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. 4, the total cycles per step is equal to the maximal computation cost of the PEs plus the communication cost, or the objective function.

4.3 Allocation and binding heuristic

4.3.1 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 solution 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. 7

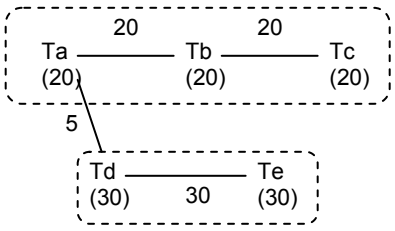

Fig. 7. Merging tightly coupled ODE types

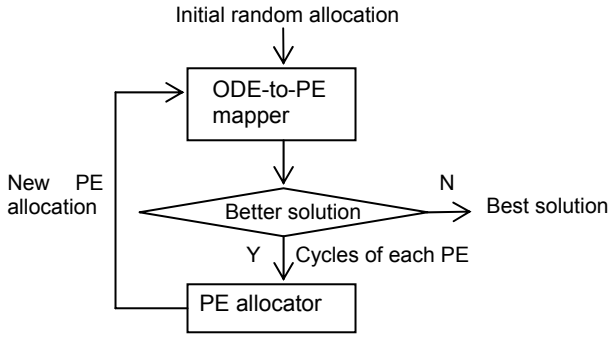

Fig. 8. Overall structure of the allocation and binding heuristic for the network of heterogeneous PEs

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 between two ODE types. 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.

4.3.2 Allocation and binding heuristic overview

Once the PE types have been chosen, we allocate a 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. 8. 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 the current PE allocation based on the objective function defined in Section 4.2. The PE allocator adjusts the PE allocation based on the feedback from the ODE-to-PE mapper.

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 improvement process will terminate when the mapper cannot find a better solution.

4.3.3 ODE-to-PE mapper

The ODE-to-PE mapping algorithm is based on the mapping algorithm for the network of general PEs [7], 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) defined in Section 4.2. The ODE-to-PE mapper will output the number of cycles for each PE for the best mapping has been found.

4.3.4 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. The overall time complexity of our approach is: *O(#allocator iterations * #mapper iterations * C)*, where *C* is the cost of generating a new mapping and re-computing the cost. C is mainly determined by the average number of edges of an ODE (not by the total number of ODEs), because the neighbor function and the incremental cost function only modifies one ODE and the neighbors.

4.4 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.

4.4.1 PE components generation

Since the general PEs only differ in the data-ram size, instruction ram size, and input ports number [7], we developed a script to generate PE components with all possible combinations of the three parameters.

A custom PE contains unique ODE datapaths 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 datapath generator to facilitate custom PE generation. The ODE datapath generator reads an ODE string, and outputs a fully pipelined ODE datapath component for calculating this ODE. The datapath generator parses the input ODE into an expression tree, and uses an ASAP (as-soon-as-possible) scheduling algorithm [20] to schedule the operations. The generator adds pipeline registers for each stage. An ODE datapath usually contains 5-15 stages (or delays), depending on the ODE. The datapath 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 datapath may require different custom data ram. With the ODE datapath generator and the custom data-ram generator, a custom PE component of any type can be generated on the fly.

4.4.2 PE instruction generation and top level network generation

The PE instructions can be divided into two parts: (1) "evaluate and update" instructions, and (2) "data transfer" instructions, as shown in Fig. 4. 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 [7]. 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 [8].

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.

5. EXPERIMENTAL RESULTS

This section describes experimental results using five physical system models as benchmarks. The section compares 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 an 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 [28] for synthesis. Note that this work is not limited to a particular FPGA or synthesis tool.

5.1 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 such as binary trees, 2-dimensional meshes, rings, etc.

1. *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 Section 3.1. The Weibel lung has a binary tree structure.
2. *Neuron network*: a neuron network model [24] 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.
3. *Weibel lung with gas exchange*: 11 generation Weibel lung model with 500 capillary cells connected to leaf branches. The model contains 3 types of ODEs as discussed in Section 3.1, and contains 4594 ODEs and has a binary tree structure.
4. *Hemodynamic*: the hemodynamic model [25] 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.
5. *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 ring.

5.2 Heterogenous networks vs. HLS and homogeneous networks

We implemented the ODE solver for each model using a commercial high-level synthesis tool[1]. Since physical systems exhibit many commonly recurring patterns we incorporate the idea of regularity extraction [21] 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.

We first attempted to input the entire PE network design into the HLS tool, however the tool used a centralized block ram to store all the data and the created design was not efficient. To eliminate the memory bottleneck of the system, we manually optimized the communication structure of the HLS approach (Fig. 9) to provide a better comparison to our custom network approach. The state

[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 HLS tools is highly encouraged; we will provide our models for such purposes upon request.

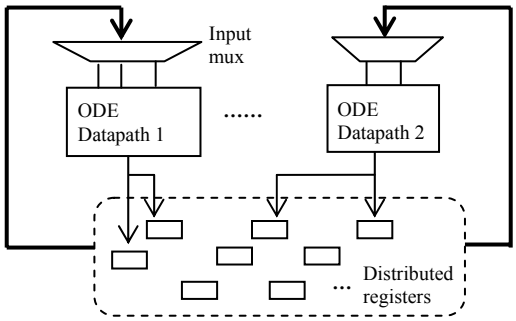

Fig. 9. The overall architecture of the ODE solver using an HLS tool with regularity extraction.

variables of a physical system are placed into distributed registers throughout the design. Each ODE datapath is mapped with a subset of ODEs of the physical system, thus each ODE datapath 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 a time multiplexing scheme. Using time multiplexing decreases 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 in Section 4.3 to allocate the ODE datapath, and balance the number of ODEs in each ODE datapath. Since each ODE datapath 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 Section 4. 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 I. 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 [16]. 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.

5.2.1 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. 9, 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 (the reason will be discussed in Section 5.2.3).

Table I. Synthesis results of custom 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.

HLS	LUTs	BRAMs	DSPs	Equiv. LUTs	PEs.	Cycle	Freq (Mhz)	Perf. (ms)	Syn. Time (min)
Weibel	101,031	131	245	*209,441*	67	296	112	*264*	752
Neuron	139,118	156	190	*242,778*	80	370	102	*363*	664
Weibel_gas	106,008	144	225	*214,098*	72	332	110	*301*	746
Hemodynamics	82,007	145	246	*195,707*	69	420	131	*321*	356
Weibel_hemo	79,735	143	230	*188,715*	75	340	115	*295*	520
General PEs									
Weibel	89,761	396	396	*331,321*	396	184	130	*142*	230
Neuron	74,632	294	294	*253,972*	294	290	150	*193*	147
Weibel_gas	78,178	281	281	*249,588*	281	302	105	*288*	123
Hemodynamics	75,311	281	281	*246,721*	281	364	190	*192*	107
Weibel_hemo	74,956	281	281	*246,366*	281	326	165	*198*	106
Custom PEs									
Weibel	48,579	56	205	*119,989*	56	221	129	*171*	72
Neuron	49,762	71	129	*107,572*	53	397	123	*323*	114
Weibel_gas	50,491	62	202	*123,311*	61	253	123	*206*	78
Hemodynamics	50,743	111	386	*187,203*	111	185	143	*129*	90
Weibel_hemo	72,828	154	553	*266,518*	175	84	167	*50*	146
Heterogeneous PEs									
Weibel	65,036	160	560	*262,636*	80	52	219	***24***	174
Neuron	63,458	133	462	*226,838*	67	69	204	***34***	185
Weibel_gas	58,800	141	494	*233,060*	86	72	222	***32***	122
Hemodynamics	48,652	118	444	*202,132*	142	56	250	***22***	69
Weibel_hemo	58,795	141	494	*233,055*	86	71	213	***33***	130

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.

5.2.2 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 place a limited number of ODE datapaths, which limits the performance of the HLS approach. The time multiplexing 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.

5.2.3 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 as discussed in Section 4.1. 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 networks depends heavily on the model; certain models may have faster performance when implemented as general PE networks (e.g., Nueron).

5.3 Heterogeneous network vs. CPU and GPU

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

1. 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
2. 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. 11(a). Another approach is to implement different GPU functions for each ODE type, shown in Fig. 11(b). However, the second approach executes each function sequentially, which is slower than the first approach. We 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 that our GPU implementations are of high quality, we asked an experienced GPU programmer to optimize our CUDA code. Runtime was reduced an additional 5-15%, mainly by utilizing coalesced global memory accesses.

The performance of each approach is illustrated in Fig. 10. 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

```
For each step                    For each step
  ODE_kernel() {                   ODE_kernel1()  {type1}
    branch1:  type1                ODE_kernel2()  {type2}
    branch2:  type2                ODE_kernel3()  {type3}
    branch3:  type3                ...
    … }
        (a)                              (b)
```

Fig. 11. GPU kernel function implementation options

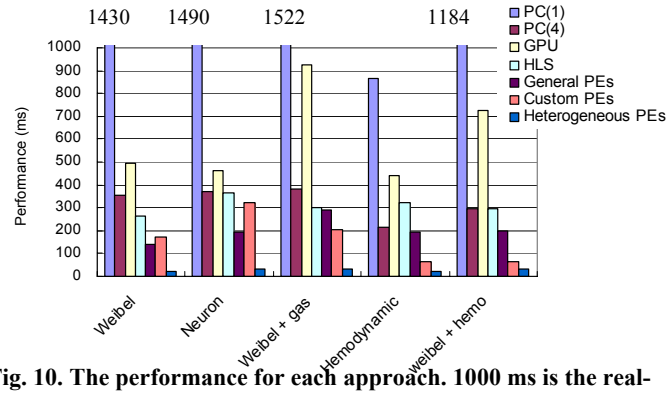

Fig. 10. The performance for each approach. 1000 ms is the real-time constraint and some PC(1) numbers extend off the chart top.

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 [5]. Thus the frequent function invocation (10^5 times per second) greatly impacts the GPU's performance. According to profiling of the GPU-accelerated program, the pure function invocation overhead consumes 30% - 60% of the total 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. 12. 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

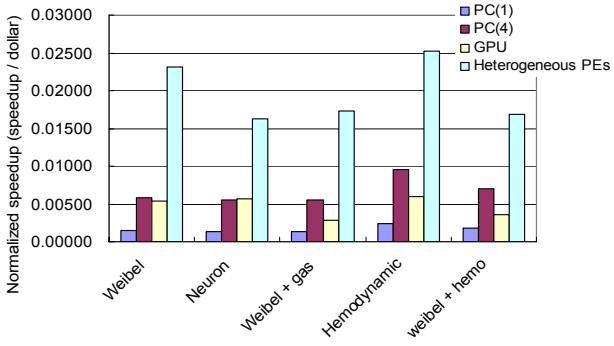

Fig. 12. The normalized speedup for each approach

size and the flexibility to build custom interfaces to the physical world [17].

6. CONCLUSION

We introduced a custom network of heterogeneous (both general and custom) processing-elements for real-time emulation of complex physical systems. We developed an automated tool to generate a custom PE network 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 custom networks of heterogeneous PEs were on average 10.8x faster and of comparable size. Compared to the custom 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 3GHz I7-950 processor, and 20x faster than a 763 MHz 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.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (CNS1016792, CPS1136146), the Semiconductor Research Corporation (GRC 2143.001), and a U.S. Department of Education GAANN fellowship.

8. REFERENCES

[1] Amorim, R.M., Rocha, B.M., Campos, F.O., dos Santos, R.W. 2010. Automatic code generation for solvers of cardiac cellular membrane dynamics in GPUs, EMBC.

[2] Atkinson, K. 1993. Elementary. Numerical Analysis, 2nd Edition, John Wiley & Sons, Inc. New York, New York.

[3] AutoESL 2012. http://www.xilinx.com/tools/autoesl.htm

[4] Butcher, J.C. 2003. Numerical methods for ordinary differential equations, ISBN 0471967580.

[5] CUDA programming guide 2011. http://developer.download.nvidia.com/compute/cuda/4_0-/toolkit/docs/CUDA_C_Programming_Guide.pdf

[6] Gokhale, M. B., Stone, J.M., Arnold, J., and Lalinowski, M. 2000. Stream-oriented FPGA computing in the Streams-C high level language. FCCM.

[7] Huang, C., Vahid, F., Givargis, T., 2011. Automatic synthesis of physical system differential equation models to a processing element network on FPGAs. Under submission.

[8] Huang, C., Miller, B., Vahid, F., Givargis, T., 2012. Synthesis of networks of custom processing-elements for real-time physical system emulation. Under submission.

[9] JSIM. 2012. http://nsr.bioeng.washington.edu/jsim/

[10] Kum, K., Kang, J., Sung, W. 2000. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. IEEE Transactions on Analog and Digital Signal Processing, vol. 47, no. 9, pp. 840-848, September 2000.

[11] Lee, E. A. 2008 Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, University of California, EECS Department, 2008.

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

[13] Mathworks 2012. Matlab and Simulink. http://www.mathworks.com/.

[14] MedGadget,2008. Supercomputer Creates Most Advanced Heart Model, Internet Journal of Emerging Medical Technologies.

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

[16] Meyer, J., Kocan, F. 2007. Sharing of SRAM Tables Among NPN-Equivalent LUTs in SRAM-Based FPGAs, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol.15, no.2, pp.182-195, Feb. 2007 doi: 10.1109/TVLSI.2007.893581

[17] Miller, B., Givargis, T., Vahid, F. 2011. Application-Specific Codesign Platform Generation for Digital Mockups in Cyber-Physical Systems IEEE Electronic System Level Synthesis Conf. (ESLsyn)

[18] National Instruments. 2011. LabView FPGA Module. http://www.ni.com/fpga/

[19] Nvidia Corporation. 2011. http://www.nvidia.com/object/gpu.html.

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

[21] Rao, D.S., Kurdahi, F.J. 1993. On clustering for maximal regularity extraction, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.12, no.8, pp.1198-1208, Aug 1993

[22] Salwinski, L., and Eisenberg, D. 2004. In silico simulation of biological network dynamics. Nature. Nature Publishing Group. pp 1017-1019.

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

[24] Terman, D., Ahn, S., Wang, X., Just, W. 2008. Reducing neuronal networks to discrete dynamics, Physica D: Nonlinear Phenomena, Volume 237, Issue 3, March 2008.

[25] Van Meurs, WL. 2011. Modeling and Simulation in Biomedical Engineering: Applications in Cardiorespiratory Physiology. McGraw-Hill Professional

[26] Villarreal, J., Park, A., Najjar, W., and Halstead, R. 2010 Designing modular hardware accelerators in C with ROCCC 2.0, FCCM, 2010.

[27] Weibel, E.R, Morphometry of the Human Lung. Berlin, Germany: Springer-Verlag, 1963

[28] Xilinx ISE. 2011. http://www.xilinx.com/

[29] Yoshimi, M., Osana, Y., Fukushima, T., and Amano, H. 2004. Stochastic Simulation for Biochemical Reactions on FPGA. FPL. pp 105-114.