IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), Toronto, Canada, April 29 - May 1, 2012

A Heterogeneous Architecture for Evaluating Real-Time One-Dimensional Computational Fluid Dynamics on FPGAs

Isaac Liu, Edward A. Lee
Electrical Engineering and Computer Science
UC Berkeley
Berkeley, California, USA
{liuisaac, eal}@eecs.berkeley.edu

Matthew Viele
R&D
Drivven, Inc.
Elizabeth, Colorado, USA
mviele@drivven.com

Guoqiang Wang, Hugo Andrade
R&D
National Instruments Corp.
Berkeley, California, USA
{gerald.wang, hugo.andrade}@ni.com

***Abstract*—Many fuel systems for diesel engines are developed with the help of commercial one-dimensional computational fluid dynamics (1D CFD) solvers that model and simulate the behavior of fluid flow through the interconnected pipes off-line. This paper presents a novel framework to evaluate 1D CFD models in real time on an FPGA. This improves fuel pressure estimation and closes the loop on fuel delivery, allowing for a cleaner and more efficient engine. The real-time requirements of the models are defined by the physics and geometry of the problem being solved. In this framework, the interconnected pipes are partitioned into individual sub-volumes that compute their pressure and flow rate every time step based upon neighboring values. We use timing-based synchronization and multiple Precision Timed (PRET) processor cores to ensure the real-time constraints are met. Leveraging the programmability of FPGAs, we use a configurable heterogeneous architecture to save hardware resources. Several examples are presented along with the implementation results after place and route for a Xilinx Virtex 6 FPGA. The results demonstrate the resource savings and scalability of our framework, confirming the feasibility of our approach – solving 1D CFD models in real time on FPGAs.**

I. Introduction

In order to meet the ever tightening worldwide emissions standards, diesel engines are becoming more and more complex. In particular, the diesel engine's fuel system must now support as many as 5 injections per cylinder event [1]. The fuel system consists of a high pressure pump, fuel injectors, and a network of connecting pipes commonly known as the ”fuel rail”. Each time an injection event happens, pulsations are sent through the fuel rail. The high pressure, around 2000 bar, in the fuel system, is often generated by a piston pump that also induces pulsations. These pulses need to be damped and/or modeled before the subsequent injection event to ensure a correct amount of fuel injection [2].

Currently most fuel systems use an ad-hoc model of fuel pressure for subsequent injection events [3]. Since many fuel rails are developed using commercial one-dimensional computational fluid dynamics (1D CFD) solvers like GT-SUITE [4], it seems a natural approach to use the same technique to model their behavior in real time. To meet the stringent real-time requirement, the solution obtained on-line usually ignores second order effects such as cavitation and thermal gradients that are taken into account in the GT-SUITE calculations. The second order effects are small, but important for designing a well-behaved system. However, there is a salient distinction between an off-line research-oriented approach like GT-SUITE and a real-time approach like the one presented here. So long as the real-time code is sufficiently accurate to allow improved fuel pressure estimation, it can close the loop of fuel delivery, allowing for a more precise air/fuel ratio control and thus a cleaner and more efficient engine.

1D CFD is used when the system to be evaluated can be described as a network of pipes. The advantage of 1D CFD over its 2D and 3D cousins is the greatly reduced number of nodes to be solved, and the simplified equations in each node. This makes it common for use in simulating transient operation of internal combustion engines [5]. This also makes it possible to solve these problems in real time using a highly parallel approach. We specifically examine the area of fluid flow, but heat transfer, mechanical dynamics, and electrical circuit simulation all represent similar problems. In each of these problems, it may be possible to represent the set of equations to be solved as a graph, where each node of the graph represents a physical quantity to be modeled, such as a sub-volume of fluid in a pipe. The information path communicated by nodes is represented as the interconnect of the graph.

When an explicit fixed time step solver is used to solve the governing equations, the time step is determined by the modeling granularity of the application. The requirement is that the solver must run faster than the speed of information flow. This is expressed as

$$\frac{\Delta t}{\Delta x} a = C,$$

where a is the wave speed, C is the Courant number, Δt is the time step, and Δx is the spatial discretization step. For stability purpose, the Courant number needs to be less than 1, and a number below 0.8 is recommended [4]. For instance, if a fluid has a wave speed a of 1 $cm/\mu s$ and a discretization step Δx of 1 cm, then we require a time step Δt of less than 1 μs. The discretization step of a pipe network is dominated by its smallest sub-volume. For a diesel fuel system, a 1 cm discretization step is common. For our purposes in this paper, we treat the speed of sound (wave speed) as 1500 m/s [6]. During each time step, each node reads the neighboring data from the previous time step and computes the new data to be sent to its neighbors. In this periodic and parallel execution of all nodes, the performance of the system is limited by the node with the longest execution time. We only require adequate performance so that this slowest node can complete in Δt.

1D CFD fits into the class of problems that are *heterogeneous* and *micro-parallel*. The *micro-parallel* modifier emphasizes small, dedicated portions of the problem being solved in each computational element, while the *heterogeneous* modifier emphasizes a number of distinct node types. This distinguishes it from homogeneous, micro-parallel problems often found in image processing, which lend themselves to graphics processor unit (GPU) and SIMD solutions with large common memories [7].

The advent of caches, out-of-order executions, branch prediction, and other performance improvements in modern processors improves their average execution speed at the cost of determinism. As a result, worst-case execution time (WCET) analysis often gives imprecise and overestimated results. For applications that must meet the execution time deadline requirements, this causes overprovisioning of hardware resources in order to guarantee no timing violations. The Precision Timed (PRET) architecture [8] is a processor architecture designed to provide timing determinism and good worst-case performance. It contains multiple hardware threads with predictable timing, and provides timing instructions to gate execution time of code blocks on the hardware threads.

This paper presents a novel framework for real-time execution of a 1D CFD solver on a Field Programmable Gate Array (FPGA) using PRET cores. We map the heterogeneous computation nodes onto the hardware threads of multiple PRET cores. The deterministic execution time ensures that each node completes within the specified time step and the timing instructions ensure the synchronization of data communication between threads and cores. We show that a timing deterministic design allows us to minimize the synchronization overhead and processor core footprint while our heterogeneous evaluation architecture further optimizes the FPGA area and leads to a practical and scalable solution.

II. Related Work

The computing power of FPGAs has enabled accelerated simulation in many application domains. In this paper, we focus on real-time CFD problems. FPGAs have been used to accelerate off-line 2D and 3D CFD computations with millions of nodes [9]. In these examples, there is no real-time constraint and the number of computation nodes is huge, which makes a common practice to reuse FPGA elements for many fluid nodes. Soft real-time CFD has been used for video games for quite a while [10]. These cases differ from our application in several important respects. First, they operate on the order of milli-seconds (e.g. $25ms$) as opposed to micro-seconds (e.g. $5\mu s$) for our case. Second, the soft real-time simulation results just need to look good to game players, so accuracy is not all that important. Lastly, being soft real-time, they can be allowed to miss a deadline and degrade gracefully if they cannot complete the calculations in time.

From a hardware architecture perspective, we need to consider evaluating our problem via the alternatives of traditional processor or GPU. Traditional desktop processors, like Intel's Pentium, have a great deal of non-deterministic behavior brought about by caches, out-of-order instruction executions etc., which makes it difficult to implement deterministic hard real-time systems. GPUs are gaining ground in scientific computing because of the large total throughput they offer. While in aggregate a GPU can outperform an FPGA-based implementation, it has some disabling limitations [11]. GPUs do well with relatively homogeneous tasks. E.g., NVIDIA's GTX 280 has 30 streaming multiprocessors. Each streaming multiprocessor is effectively a 32-channel SIMD processor. Communication to the global memory takes hundreds of cycles. Our application has many different types of nodes interconnected and it requires an ultra-low latency. This is why we chose not to pursue a GPU-based approach.

Besides their powerful computing capability, FPGAs have additional advantages. They can contain other structures unrelated to the CFD code like actuator control logic or sensor interfaces that can be connected to the correct part of the CFD model with a single-cycle latency.

There may exist different implementation options for our application on FPGAs. Without leveraging the benefits from PRET cores, we could attempt the problem in discrete FPGA blocks. In order to make the application fit in a practical FPGA like the one we tested, we would need to re-use the hardware multipliers, adders, and other functional units. This would require a state machine to run it, begins to look a great deal like a processor. Along this line, the highly parallel property of this application lends itself nicely to a multi-threaded architectures that favor throughput over latency. The real-time requirements propels us to choose a deterministic architecture, and the configurability of FPGAs allows us to optimize for the heterogeneity nature of our implementation. The PRET cores, with a deterministic multi-threaded architecture and good compiler support, offer a flexible solution, and is a perfect architecture for the job. To the best of our knowledge, we believe this is the first attempt to attack real-time CFD on this time scale and complexity of problem.

III. Background

A. One-Dimensional Computational Fluid Dynamics

Solving 1D CFD problems begins with the Navier Stokes equations for compressible flow. We construct a library of computational elements for the type of pipe segments we use in the form of first order finite difference equations [12]. We start with momentum equation (1) and continuity equation (2):

$$\frac{P_x}{\rho} + \dot{V} + \frac{fV}{2D}|V| = 0, \tag{1}$$

$$a^2 V_x + V\left(\frac{P_x}{\rho} + g\sin\alpha\right) + \frac{P_t}{\rho} = 0, \tag{2}$$

where P is pressure, ρ is fluid density, V fluid velocity, f is the Darcy-Weisbach friction factor, D is pipe diameter, a is the wave speed, and $g\sin\alpha$ is the directional force of gravity. A dot $(\cdot)$ over a variable indicates the total derivative with respect to time. Subscripts x and t indicate partial differentiation along the pipe length and with respect to time, respectively.

These equations can be expanded and simplified by leaving out the body force and convective terms because these are negligible given the pressure and flow regime. A method of characteristic solution is used to explicitly evaluate the pressure and flow at the next time step through a first order finite difference method. The left graph in Fig. 1 shows the evaluation of pressure and flow at point I, where I is i at $t_0 + \Delta t$, based on the pressure and flow of the adjacent points at time t_0. The equations to be evaluated at each point are

$$V_I = \frac{1}{2}\left[V_{i-1} + V_{i+1} + \frac{1}{a\rho}\left(P_{i-1} - P_{i+1}\right) - \frac{f\Delta t}{2D}\beta\right], \tag{3}$$

$$P_I = \frac{1}{2}\left[P_{i-1} + P_{i+1} + \rho a\left(V_{i-1} - V_{i+1}\right) - \frac{\rho a f\Delta t}{2D}\beta\right] \tag{4}$$

where $\beta = (V_{i-1}|V_{i-1}| + V_{i+1}|V_{i+1}|)$.

One critical part in evaluating the value at point I is that there is a fixed relationship among Δx, Δt, and the wave speed a such that $a \leq \Delta x/\Delta t$. Δx is fixed by the geometry of the problem and a is fixed by the properties of the working fluid such as $a = \sqrt{K/\rho}$, where K is the bulk modulus. Therefore, Δt is defined. All computations must complete within Δt and the results must be posted in exactly Δt. Because the wave speed varies and the geometry of the problem may not work out evenly for all pipe segments, a modified method is implemented with an interpolation step. This is shown by the right graph in Fig. 1 and described by equations

$$\zeta_R = \zeta_i - \theta a(\zeta_i - \zeta_{i+1}) \text{ and } \zeta_L = \zeta_i - \theta a(\zeta_i - \zeta_{i-1}),$$

where θ represents the amount of interpolation desired and subscript R (resp. L) denotes the right (resp. left) point of i. These equations are evaluated for both pressure $\zeta = P$ and velocity $\zeta = V$. While we use this method to give calculation leeway, it is important to realize that this adds complexity to every block in the system as well as decreases Δt.

Let $B = a\rho/A$ and $E = \rho f\Delta x/2DA^2$, where A is the pipe cross sectional area. Plugging them into (3) and (4) and further rearrangement give the following characteristic equations:

$$C_p = P_{i-1} + Q_{i-1}\left(B - E|Q_{i-1}|\right),$$

$$C_m = P_{i+1} - Q_{i+1}\left(B - E|Q_{i+1}|\right),$$

where Q is the flow rate along the pipe and subscript p (resp. m) denotes the plus (resp. minus) branches of the characteristic equation.

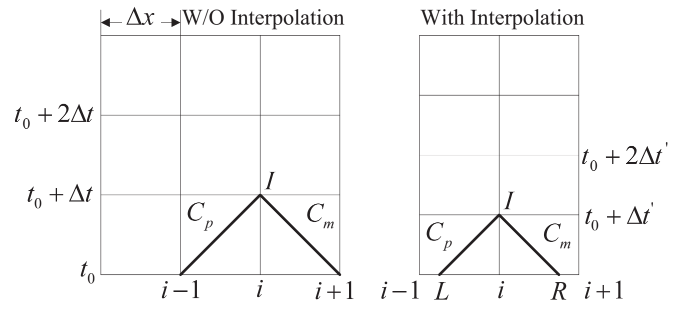


Fig. 1: First Order Difference

Table I shows the equations for each of the supported flow elements. From them we can generate a network of flow elements that represents our fuel system. Subscript Bnd denotes a boundary condition. C_V is the flow coefficient which is a function of the nominal open flow (Q_0), the downstream pressure (P_0), and the fraction the valve is open (τ).

TABLE I: Equations for Supported Types

Type	$P_I =$	$Q_I =$
Pipe segment	$\frac{(C_p+C_m)}{2}$	$\frac{(P_I+C_m)}{B}$
Imposed pressure	P_{Bnd}	$\frac{(P_{Bnd}-C_m)}{B}$
Imposed flow	C_m+BQ_{Bnd}	Q_{Bnd}
Valve	C_p-BQ_I	$-BC_V+\sqrt{(BC_V)^2+2C_VC_p}$ $C_V=\frac{(Q_0\tau)^2}{2\cdot P_0}$
Cap	C_p-BQ_I	0
Pipe "T"	$\frac{\frac{C_{p_1}}{B_1}+\frac{C_{m_2}}{B_2}+\frac{C_{m_3}}{B_3}}{\sum_{j=1}^{3}\frac{1}{B_j}}$	$-\frac{P_I}{B_1}+\frac{C_{p_1}}{B_1};-\frac{P_I}{B_2}+\frac{C_{m_2}}{B_2};$ $-\frac{P_I}{B_3}+\frac{C_{m_3}}{B_3}$

B. Precision Timed Architecture

For real-time applications that need timing determinism, Edwards and Lee [13] propose Precision Timed (PRET) architectures. These architectures are designed for timing predictability and determinism, rather than average-case performance. Lickly et al. [8] present a multi-threaded implementation of the PRET architecture using a thread-interleaved pipeline and scratchpad memories. The thread-interleaved pipeline removes data and control hazards within the pipeline by interleaving multiple hardware threads in a predictable round robin fashion. Scratchpad memories are single-cycle access on-chip memories which are managed in software. They are used in place of a cache to improve on predictability because the contents on the scratchpads are controlled in software.

Along with the predictable architecture, Lickly et al. [8] also introduce a timing instruction to control the temporal behavior of programs. The timing instruction gives programmers a way to specify a lower bound on execution time for a specific code block, guaranteeing that the code block will not complete until at least the specified execution time. [8] also outlines a producer consumer example that synchronizes communication through the use of timing instructions, to ensure an ordering between shared data accesses. Our implementation of the 1D CFD solver uses a similar mechanism to synchronize communication across computation nodes and align the computation with real-world events.

IV. Design Flow and Architecture

A. System Description and Design Flow

The process of generating a system is outlined in Fig. 2. Starting from a description of the flow system and our library of elements, we can construct a graph to describe the system. The flow system dictates the maximum time step of the system. With this information we can instantiate processors and interconnects tailored to our needs and apply the library code to them. From there, we build and deploy our system onto an FPGA target.

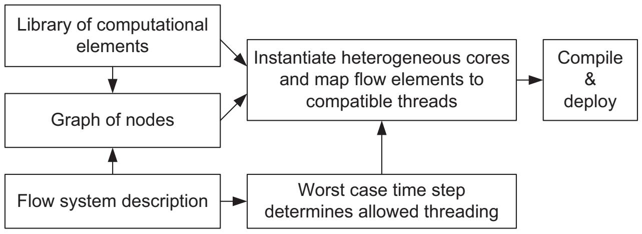

Library of computational elements
Graph of nodes
Flow system description
Instantiate heterogeneous cores and map flow elements to compatible threads
Worst case time step determines allowed threading
Compile & deploy

Fig. 2: Design Flow

Fig. 3 shows an overview of a representative system that models fuel rails. The 1D CFD model is bounded inside the dashed rectangle. External to that is the real-world sensor and actuator interfaces which provide boundary conditions or consume model output variables. The small blue squares inside the dashed rectangle represent the network of flow elements. In a practical simulation of a diesel fuel system, the total number of flow elements can range from around 50 to a few hundred.

Each pipe element is a computation node, and their graphical representation is shown in Table II. The top 3 rows of the table represent the flow elements described in Table I. *Mechanical calculation* elements compute the inputs to *valve*,

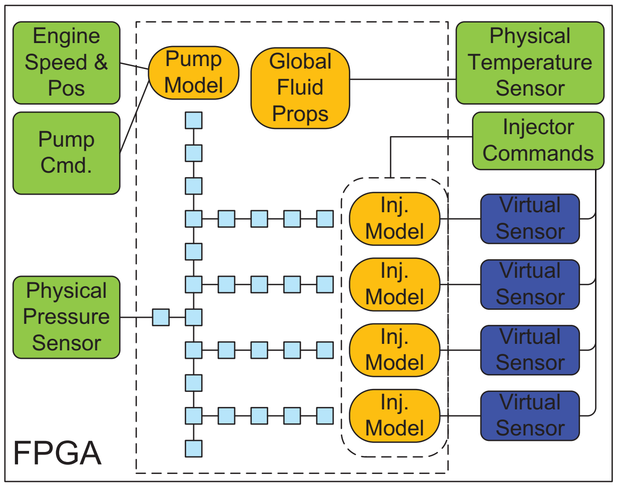
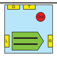
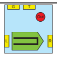
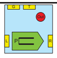
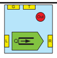
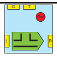
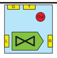
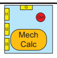
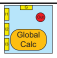
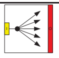
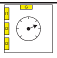


Fig. 3: High Level System Diagram

TABLE II: Library of Computation Node Elements

Pipe segment		Cap	
Imposed pressure		Imposed flow	
Pipe “T”		Valve	
Mechanical calculation		Global calculation	
Global distribution		Output	

imposed flow, and *imposed pressure* blocks. They serve as an interface between the flow model and the real world. *Global calculation* elements are used to compute the temperature dependent variables of density and wave speed. They post their data to a global distribution node, broadcasting it to all other nodes. Blocks with white backgrounds in the last row of Table II, i.e., *Global distribution* and *Output*, are implemented directly in FPGA fabric, not mapped to a processor thread. Global distribution elements in the graph are purely used to show the distribution of input values to each of the computational elements. Output elements are used when data needs to be communicated out of the model to other parts of the FPGA.

For illustrative purposes, we show a simplified sample pipe network with an imposed flow input (P1) in Fig. 4. Fluid travels through a few pipe segment nodes (P2 and P3) to a “T” intersection (P4), where it splits off to a second branch of the network. The “T” node is also measured by the outside world (D1) through an output port. Flow going up the new leg ends in a cap (P8), while flow continuing down the original path exits the system through a valve (P6). Temperature dependent variables of density and wave speed are computed by global calculations (G1, G2, and G3) and delivered by global distributions (GD1, GD2, and GD3) to each of the computational elements every time step.

B. System Hardware Architecture

The hardware architecture of real-time 1D CFD evaluation consists of multiple PRET cores connected through point-

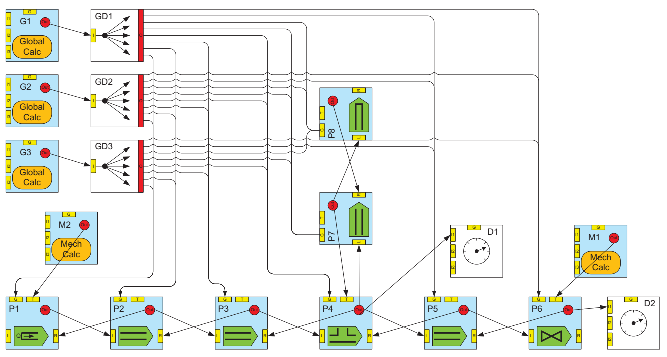


Fig. 4: Detailed System Diagram

to-point connections and a global distribution circuit. Fig. 5 shows a block-level view of the hardware architecture. The PRET core implementation used here is slightly different from the one presented in [8]. Instead of the SPARC ISA [14], our PRET cores implement the ARMv4 instruction set, without thumb mode. We switched to ARM due to its popularity in the embedded domain and the unpredictable nature of register windows in the SPARC ISA.

Similar to what is mentioned in Section III-B, each core consists of hardware threads interleaved through the pipeline in a predictable round robin fashion. This provides two major advantages. First, multi-threaded architectures maximize the throughput over latency, favoring highly parallel applications. When a hardware thread is waiting on a long latency operation (such as a floating point calculation or memory access) to complete, other threads can continue to execute in the pipeline, effectively hiding the latency of the operation. E.g., in our implementation, floating-point addition and subtraction appear as single-cycle instructions for each thread in timing analysis. Second, the thread interleaving mechanism allows for a simpler pipeline design. Since data and control hazards are no longer present in the pipeline [8], the logic used for handling them can be stripped out, greatly reducing the cost of the core. Multiple threads share the same datapath, so the cost of adding threads is far less than adding a core, further reducing the cost of the system. We discuss in more details the trade-offs involving adding threads later in Section V.

Instead of each being implemented as its own core, computation nodes are mapped onto hardware threads. The memory footprint required for each node is small enough (roughly a hundred assembly instructions) so the scratchpad is sufficient for memory use, and no main memory is needed.

Our architectural design supports configurations which exclude certain floating point units, since not all computation nodes require all floating operations. For example, as shown in Table I, square root is only used by the valve node, and divide is only used by the "T" node. The floating point divide and square root hardware are the most resource intensive units, but the valve and "T" nodes usually represent only a few percent of overall system. The common fuel rail system we present later contains 234 nodes, but only 5 are "T" nodes and 4 are valves. To save on hardware resources, we could use software emulation for the complex operations at the cost of increasing in the execution time of the "T" and valve nodes. As all nodes synchronize communication points at the end of each time step, the overall performance of our system is bounded by the slowest computational element. As a result, the performance hit from using software emulation for these small percent of nodes would limit the overall system performance. Instead, by allowing different configurations of ARM-based PRET cores within the system, we can include the hardware implementations of complex operations only on cores that require them, getting the performance boost without a huge resource overhead. This leads to substantial resource savings, which we show in Section V.

Our implementation establishes point-to-point communication channels across nodes, and a global distribution circuit to distribute temperature dependent variables. Each node in our system has one to four input ports and one to four output ports. One input port is always dedicated as the global port, which receives broadcasts from the global distribution circuit. Nodes mapped to the same core (*intra-core* communication) can communicate through the shared local memory within the core. Nodes mapped to different cores (*inter-core* communication) communicate through the point-to-point interconnect, as illustrated in Fig. 5. We use shared dual-port Block RAMs (BRAMs) to implement the inter-core communication. This serves two purposes. First, it provides single-cycle deterministic communication, as BRAM access is single cycle. This allows the timing analysis to be simplified as there is no hardware protocol that needs to be accounted for when accessing data through the inter-core communication channels. More importantly, the timing analysis for each node is now independent of the node mapping because both intra- and inter-core communication mechanisms are single cycle via BRAMs. Second, by using the dedicated BRAM blocks on the FPGA for interconnects, we save the logic slices to be used for computation nodes. This is useful because the limiting resource in our implementation is logic slices, not BRAMs. This is justified later in Section V. Each core only requires a small number of BRAMs to be used for registers and scratchpads, so the BRAM utilization ratio is far less than the logic slice utilization ratio. Our *timed periodic* execution of computation nodes (described next in Section IV-C) ensures that we only need a buffer size of one for each of the words. Because the communication bandwidth is small, we only need one BRAM block to establish an interconnect that allows all threads from one core to communicate with all threads on the other.

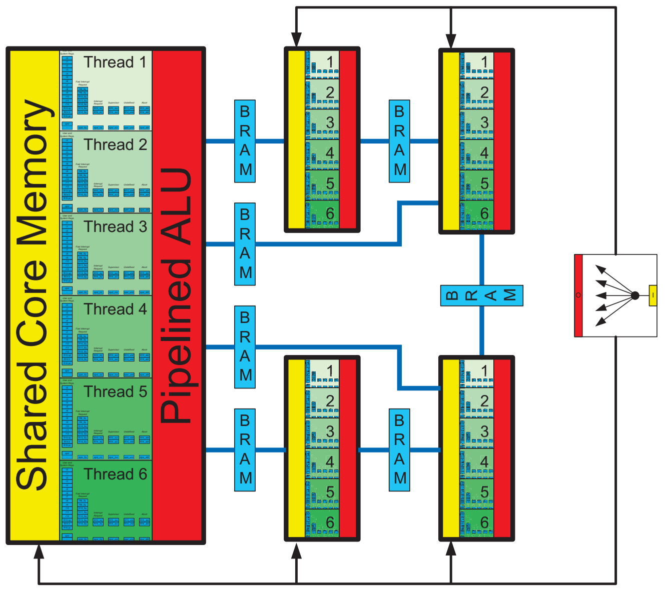


Fig. 5: System of PRET Cores and Interconnects

All of our flow elements have a dependency on density and wave speed that are functions of temperature. Temperature is assumed to be the same throughout the system, so these parameters are computed in a single computational element and broadcast to all pipe elements through the global distribution circuit, as illustrated in Fig. 4. Leveraging this, the global distribution circuit is implemented with a single broadcaster that writes to dedicated memories local to each core. This broadcast receiving memory is also synthesized to a small dual-port BRAM, with a read-only side connected to the core, and a write-only side connected to the broadcaster. This memory is shared amongst all threads in a core so all threads can access the global variables. This architecture allows us to save on the resources needed to implement a full fledged interconnect routing system or any network protocol to be used for broadcasting.

C. Software Design

We implement the equations in Table I in the language C, and compile it with the GNU ARM cross compiler [15] to run on our cores. Columns 2-6 in Table III show the number of Multiply, Add/Subtract, Absolute Value, Square Root, and Divide operations required by each computational element after optimization. Each computation node is executed on a hardware thread and data is exchanged only at the boundaries of time steps to avoid data races. Fig. 6 shows an example timeline view of the operation for each node. The execution for computation nodes (top of Fig. 6) during each time step consists of three phases: (1) Read in the pressure and flow rate value from neighbors, and global value from the global port; (2) Compute the output values; (3) Send output values to neighbors to be used for next time step. The global and mechanical calculation nodes do not read data from other nodes, but might gather data from physical sensors to be broadcast or sent to other nodes.

TABLE III: Computational Intensity of Supported Types

	Without Interpolation / With Interpolation					
Type	Mul	Add/ Sub	Abs	Sqrt	Div	Thread cycles
Pipe segment	10 / 18	5 / 13	2 / 2	0 / 0	0 / 0	81 / 51
Imposed pressure	6 / 10	3 / 7	1 / 1	0 / 0	0 / 0	50 / 38
Imposed flow	5 / 9	3 / 7	1 / 1	0 / 0	0 / 0	51 / 40
Valve	13 / 17	5 / 9	1 / 1	1 / 1	0 / 0	64 / 55
Cap	4 / 8	2 / 6	1 / 1	0 / 0	0 / 0	48 / 39
Pipe “T”	16 / 28	13 / 25	3 / 0	0 / 0	4 / 4	111 / 72

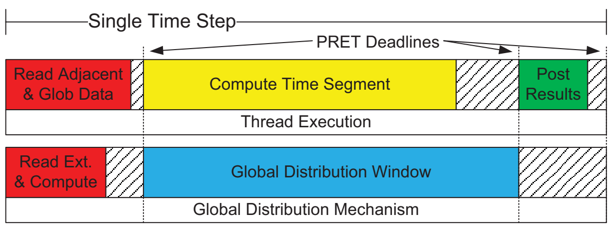


Fig. 6: Execution of Nodes at Each Time Step

The computation done by each node consists of only a single path of execution, voiding the need for complex software analysis. Data synchronization is handled by the synchronized periodic communication points, which enforces an ordering between the writing and reading of shared data. This voids the need of any explicit synchronization method, preventing any overhead or unpredictability for communication. These properties allow us to statically obtain an exact execution time on our predictable architecture for each computation node. We show this in the last column of Table III. A *thread cycle* is defined as a thread’s perceived clock cycle. To get the physical execution time, multiply the thread cycles by the number of hardware threads in the pipeline to get processor clock cycles, then convert the clock cycles to physical time according to the processor clock speed.

In addition to statically assuring that the worst-case execution time meets the timing constraints specified, we also need to enforce that node executions remain synchronized. Our approach uses specialized timing instructions provided by the PRET architecture to enforce the synchronized communication points for all nodes. When a timing instruction is decoded, it first enforces the previously specified timing constraint, then it specifies a new timing constraint for the next code block. In the code, one timing instruction is used during initialization to specify the first timing constraint. Then, timing instructions are inserted within the loop iteration to separate the execution phases. When a phase completes and the timing instruction is reached, the processor enforces the previously specified time bound by stalling if needed. Once code continues execution, the next timing specification is set. Each timing instruction takes 2 cycles because it manipulates a 64-bit value representing time. For our computational elements, 3 timing instructions are used during each computation iteration, thus 6 cycles of overhead are introduced per time step. The overhead is already included in our execution time analysis presented in Table III. Fig. 6 shows the program synchronization points that our timing instruction enforces. The hatched area in the figure denotes slack time that is generated by the timing instructions.

The same effect can possibly be achieved with instruction counting and NOP insertions. This can certainly be done on any deterministic architecture, such as PRET. However, NOP insertions are brittle and tedious. Any change in the code would change the timing of the software and insertions need to be adjusted to ensure a correct number of NOPs. Designs now are mostly written in programming languages like C and compiled into assembly code, making it extremely difficult to gauge the number of NOPs needed at design time. The timing instructions allow for a much more scalable and flexible approach. In a system with heterogeneous nodes that exhibit different execution times, the timing instructions allow us to set the same timing constraints for all nodes, regardless of its contents.

V. Experimental Results and Discussion

A. Setup

We use three examples to evaluate our framework. Our first example is a simple waterhammer example taken from Wylie and Streeter [16]. Its configuration is similar to the one shown in Fig. 4, but without the "T" element and the nodes that branch up. This example contains an imposed pressure, 5 pipe segments, a valve, and two mechanical input blocks which provide both the reference pressure and the valve angle as a function of time. We use this simply as a sanity check for the correctness of functionality for our framework.

The second and third example cover two common diesel injector configurations: the unit pump and common rail. The data for configuring these cases was taken from reference examples provided by Gamma Technologies' GT-SUITE software package [4]. The unit pump is much like the simple waterhammer example in that there are no branches in the system. The input is a defined flow specified by an electronically controlled cam driven pump. The output is a single valve. There are a total of 73 fluid sub-volumes in this system. The common rail example is more complex, where the topology is roughly that described by the computational elements in Fig. 4. It has a total of 234 sub-volumes, including 5 "T" intersections and 4 valves. Both the GT-SUITE-based models use a 1 cm discretization step. Using a wave speed of 1500 m/s and a stability factor of 0.8 yields a 5.33 μs timing constraint for the slowest computational element.

We implement all our cores and interconnects on the Xilinx Virtex 6 xc6vlx195t [17] with speed grade 3. Each Virtex-6 FPGA logic slice contains 4 LUTs and 8 flip-flops, and this FPGA contains 31,200 logic slices and 512 18-KB BRAMs. Each PRET core is clocked at 150 MHz and contains 6 threads. All floating point units are generated from the Xilinx Coregen tool [18], and are configured to use the least amount of logic slices possible to meet the timing constraint. With our current ARM-based PRET implementation, our C code is compiled using the GNU ARM cross compiler [15] with the optimization compiler flag set to level 3.

For these examples, we used a mapping heuristic that groups nodes requiring same computation onto the same core. The results show that this heuristic allows us to save hardware resources by synthesizing less floating point units.

The main metric used to evaluate our approach is the resource usage after place and route on Xilinx Virtex 6 FPGAs. The speed of the system is limited by the application parameters, which determine the execution time constraints for a time step. Once those constraints are met, there is no benefit to continue to improve system speed. Instead, we focus on optimizing resource usage to improve the scalability of our approach.

B. Timing Requirement Validation

We need to ensure that the worst-case computational element can meet the timing requirements for our examples. In our thread-interleaved pipeline, a hardware context switch occurs every processor cycle and threads are scheduled in a round robin order. Given a 150 MHz clock rate, each thread executes at 25 Mhz, so thread cycles are 40 ns long. Table III shows that the "T" element, which takes 111 thread cycles with interpolation, is the worst-case node. The unit pump and common rail have a requirement of 5.33 μs, which converts to a 133 thread cycle time step requirement. For the simple waterhammer example, a bigger discretization Δx is used, which leads to a longer time step than the two complex examples. This validates that we can safely meet the timing requirements for all three of our examples, ensuring the correctness of functionality.

C. Resource Utilization

Table IV shows the resource usage for the different configurations of a core. We include the fixed point configuration in Table IV only for reference purposes. It is not used in our system as it does not contain any floating point units. Instead, the baseline configuration used in our implementation is "basic float", which contains a floating point add/subtracter, a floating point multiplier, and float to fix conversion units. The "sqrt", "div" and "sqrt & div" configurations add the corresponding hardware units onto the "basic float" configuration respectively. Besides the effect of hardware units, we also show the area impact of adjusting the thread count on a core.

TABLE IV: Number of Occupied Slices per Core on the Virtex 6 (xc6vlx195t) FPGA

Threads per core	6	8	9	16
Fixed point only	572	588	764	779
Basic float	820	823	1000	1022
Float with sqrt	987	992	1146	1172
Float with div	1039	1051	1231	1237
Float with div & sqrt	1237	1249	1403	1413

An interesting observation is that the area increase is approximately proportional only to the number of bits required to represent the thread count. E.g., 6 and 8 threads, which both require three bits to represent, have a similar area usage. Once a 9th thread is introduced, the resource usage noticeably increases, but remains similar for up to 16 threads. This can be explained by the synthesis of multi-threaded processors onto an FPGA. Multi-threaded processors maintain independent register sets and processor states for each thread, while sharing the datapath and ALU units amongst all threads. The size of the multiplexers used to select thread states and registers is determined by the number of bits encoding the thread IDs, not the number of threads. Since the register sets are implemented onto BRAMs, the number of bits used to encode thread IDs is also what determines how big a BRAM is used for the register set, not the number of threads. As a result, increasing the thread capacity of cores can potentially reduce the number of cores required to fit a fixed number of nodes, because it is possible to increase the thread count with only a small increase of area. However, since hardware threads share the processor pipeline, adding threads slows down the running speed of the individual threads. Thus, the number of threads used should be tailored to each application, depending on its performance and resource requirements. Our implementation uses 6 threads, which is the maximum number of threads allowing us to meet our timing constraint for flow elements.

As shown in Table IV, for 6 threads on a core, the "square root" configuration uses 20.3% more slices than the "basic float" configuration. The "division" configuration uses 26.7% more. A core with both square root and division requires 50.8% more slices. These are estimates because the slices occupied might vary slightly based on how the implementation tool maps LUTs and flip flops to logic slices. However, they give an intuition to the resource difference used for each configuration.

Each core uses 7 BRAMs: 3 for the integer unit register set (3 read and 1 write port), 2 for floating point register set (2 read and 1 write port), 1 for the scratchpad, and 1 for the global broadcast receiving memory.

The actual resource impact can be seen from Table V, which shows the total slices occupied after placement and route for the three examples implemented. In the homogeneous (hom. suffix) configuration, all the cores contain the square root and divide hardware. In the heterogeneous (het. suffix) configuration, only necessary cores contain square root and divide, the rest use the basic float configuration.

TABLE V: Total Resource Utilization of Examples Implemented on the Virtex 6 (xc6vlx195t) FPGA

Example		Nodes	Cores / Conn.	Slices / BRAM	
				Absolute	Relative (%)
Water Hammer	het.	12	2 / 1	1805 / 15	5.7 / 2.1
	hom.			2379 / 15	7.6 / 2.1
Unit Pump	het.	73	13 / 12	10566 / 103	33.0 / 15.0
	hom.			16635 / 103	44.0 / 15.0
Common Rail	het.	234	39 / 38	29134 / 311	93.4 / 45.0
	hom.			N/A	

For the simple waterhammer example, since only 2 cores are used, the savings are less noticeable. But as the application size scales up, the resource savings become more apparent. The homogeneous approach uses roughly 1.5 times the number of slices the heterogeneous approach uses, which is consistent with the findings in Table IV. The results also show that our system scales linearly with the number of nodes. This proved to be critical for the 234-node common rail example as only our heterogeneous architecture could implement the design on the xc6vlx195t FPGA while the homogeneous design simply could not fit. These results also reflect our decision to use a heuristic that groups nodes with the same computation together. By doing so, we can synthesize less hardware computation units overall, saving hardware resources.

Table V also shows the BRAM usage for the implemented examples. Each interconnect uses 1 BRAM and each core uses 7 BRAMs. We see that the BRAM utilization ratio is far below the logic cell utilization, validating our design choice of using BRAMs for interconnects and broadcasts.

VI. Conclusions and Future Work

In this paper we presented a novel framework for solving a class of heterogeneous micro-parallel problems. Specifically, we showed that our approach is sufficient to model a diesel fuel system in real time using the 1D CFD approach on FPGAs. We use the PRET architecture to ensure timing determinism and implement a timing based synchronization on a multi-core system. We set up a configurable heterogeneous architecture that leverages the programmability of FPGAs to efficiently implement designs for efficient area usage. Our experimental results show ample resource savings, and prove that our approach is practical and scalable to larger and more complex systems.

We plan to continue to extend this work along several lines. From the application perspective, we look to add more flow elements to our library to compare our results to more complex flow systems. We also plan to examine more closely the integration of mechanical and electrical nodes in our library. For the hardware architecture, we look to explore multi-rate timing of nodes to allow for differences in electrical, fluid, and mechanical time steps.

References

[1] *DI Driver Module Kit User's Manual*, Drivven, rev. E.

[2] H. Bauer, *Diesel-Engine Management*, 3rd ed. Society of Automotive Engineers, 2004.

[3] E. Winward, J. Deng, and R. K. Stobart, "Innovations in Experimental Techniques for the Development of Fuel Path Control in Diesel Engines," *SAE International Journal of Fuels and Lubricants*, vol. 3, no. 1, pp. 594–613, 2010.

[4] *GT-Suite Flow Theory Manual*, 7th ed., Gamma Technologies.

[5] M. Sellnau, J. Sinnamon, L. Oberdier, C. Dase, M. Viele, K. Quillen, J. Silverstri, and I. Papadimitriou, "Development of a Practical Tool for Residual Gas Estimation in IC Engines," in *SAE, Paper 2009-01-0695*, 2009.

[6] M. E. Tat and J. H. V. Gerpen, "Measurment of Biodiesel Speed of Sound and Its Impact on Injection Timing," Dept. of Mechanical Engineering, Iowa State University, Tech. Rep., 2003.

[7] B. Ylvisaker, B. V. Essen, and C. Ebeling, "A Type Architecture for Hybrid Micro-Parallel Computers," *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 99–110, 2006.

[8] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable Programming on a Precision Timed Architecture," in *CASES '08*, 2008, pp. 137–146.

[9] W. Smith and A. Schnore, "Towards an RCC-based Accelerator for Computational Fluid Dynamics Applications," *The Journal of Supercomputing*, vol. 30, no. 3, pp. 239–261, 2004.

[10] Q. Yu, F. Neyret, E. Bruneton, and N. Holzschuch, "Scalable Real-time Animation of Rivers," in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 239–248.

[11] K. Kothapalli, "The GPGPU Phenomenon : Understanding its Scope, Applicability, and its Limitations," ICDCN, 2011, Jan. 2011.

[12] J. Desantes, J. Arreglt, and P. Rodriguez, "Computation Model for Simulation of Diesel Injection Systems," in *SAE, Paper 1999-01-0915*, 1999.

[13] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Proceedings of DAC '07*, 2007, pp. 264–265.

[14] SPARC International Inc., "SPARC Standards," Website: http://www.sparc.org.

[15] "GNU ARM Toolchains." [Online]. Available: http://www.gnuarm.com/

[16] E. B. Wylie and V. L. Streeter, *Fluid Transients*. McGraw-Hill, 1978.

[17] *Xilinx Virtex-6 Family Overview*, Xilinx, March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[18] *Core Generator Guide*, Xilinx. [Online]. Available: http://homepages.cae.wisc.edu/~ece554/website/Xilinx/Coregen_user_guide.pdf