A Spatial Computing Architecture for Implementing Computational Circuits

David Grant and Guy G. F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, V6T 1Z4
Email: davidg,lemieux@ece.ubc.ca

***Abstract*—To accelerate many computational software algorithms, designers are implementing them as computational circuits. These algorithms are diverse and include molecular dynamics, weather simulation, video encoding, and financial modelling. Circuit designers repeatedly synthesize and simulate circuits for debugging and incremental design, but due to the size of computational circuits these steps are slow and waste designer productivity. In this paper we present an architecture and tool flow for rapidly compiling and simulating/executing computational circuits. We use a motion estimation circuit to demonstrate the performance vs. capacity scalability of our architecture, and show that the performance is comparable to an FPGA-based design.**

I. Introduction

To realize performance gains in many computationally intensive software algorithms, designers are implementing them in hardware as *computational circuits*. This is being done for a wide range of algorithms, including molecular dynamics, weather simulation, video encoding, financial modelling, rendering, and nuclear particle simulation. These computational circuits are word-oriented and are often very large, requiring millions of gates.

Unfortunately, creating a custom computational circuit is challenging and slow. A designer must repeatedly *synthesize* and *simulate* the circuit while debugging and incrementally adding to the design. As circuit size grows, it takes longer to synthesize and simulate, thereby reducing the productivity of the circuit designer. This paper presents a custom architecture—and a simulator for that architecture—for simulating/executing computational circuits. This paper also presents a methodology to show how an automatic tool could quickly map a computational circuit onto the architecture.

Field Programmable Gate Arrays (FPGAs) and similar reconfigurable devices solve the simulation half of the problem. They use emulation to actually implement the circuit [1]. However, these devices have two major problems for computational circuit designers. First, they require a circuit synthesized at the gate level, which takes several hours for a large circuit. Second, they suffer from a strict capacity limit on the number of gates that can be implemented. If a circuit does not fit within this limit, the designer must resort to software simulation (slow), buy a bigger device (if one exists), or partition the circuit into two or more FPGAs. These two problems, slow synthesis time and a strict capacity limit, are major obstacles for using FPGAs to create computational circuits.

To solve these two problems and retain the speed of an FPGA, we propose a custom architecture and a tool flow for computational circuits. The architecture is an array of processors that uses time-multiplexing to achieve a soft capacity limit where capacity can be traded for performance. The tools use a technique called behavioural compilation [2], [3], [4] and leverage the coarseness of the architecture to improve synthesis speed.

The research goals for the architecture and tools are:

1) To compile a circuit 10x faster than FPGA CAD tools, requiring minutes instead of hours for synthesis.
2) To have a capacity 10x that of an FPGA, and a performance no less than $\frac{1}{10}$th an FPGA at full capacity.
3) To be able to automatically trade capacity for speed, matching the speed of an FPGA at "low capacity".

Section II presents our architecture, and Section III presents the simulator for the architecture. Section IV give an outline of our tool flow for the architecture. We present the results of mapping a motion estimation circuit to the architecture in Section V, and conclude and present future work in Section VI.

II. Architecture

This section presents our architecture for simulating/executing computational circuits. The architecture is an array of processors, where each processing element (PE) only communicates with its four immediate neighbours. The architecture overview is shown in Figure 1. Each PE consists of a router and a core; both follow a pre-programmed static schedule. The architecture does not require a global low-skew clock, and avoids the long routing wires used in many coarse-grained reconfigurable arrays (CGRAs). The design presented here serves as a starting point for further research which will investigate parameters such as the size of the memories.

In this design, all PEs receive the same clock frequency, but neighbouring PEs can have a small, known clock skew between them. From the point of view of a PE, the bounded skew gives the appearance that the clock is synchronized with its four neighbours. Since communication is restricted to neighbouring PEs, a larger skew between distant PEs does not matter. With this design, the architecture is readily scalable to high clock frequencies on the order of 3 GHz or higher.

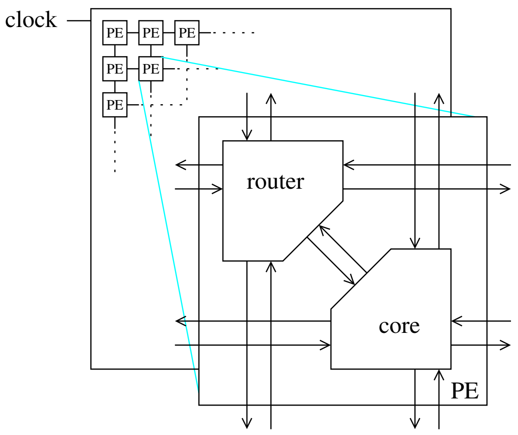

Fig. 1. Architecture overview

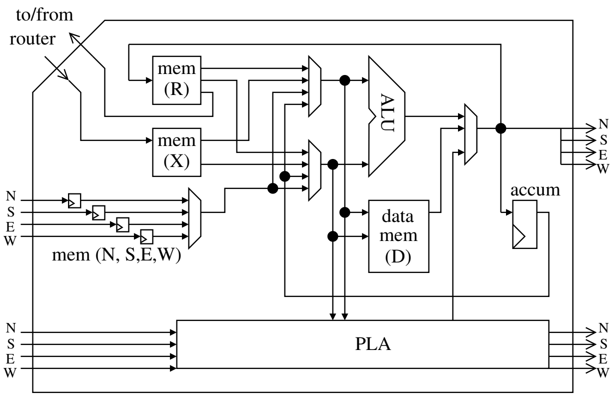

Fig. 2. PE core

When a circuit is mapped to this architecture, the user clock is different from the system (3 GHz) clock. Each PE router and each PE core contain a schedule with exactly n instructions to be executed in an infinite loop to implement the overall circuit. One pass of this schedule is equivalent to one user clock cycle, so the user clock has a frequency of $\frac{1}{n}3$ GHz. Each rising clock edge causes routers and processors to advance to the next instruction in their schedule, which is similar to how other CGRAs behave. These pre-determined schedules mean the entire architecture is deterministic. It is the responsibility of the tools to orchestrate the code for each processor and the schedule for each router so that data is always in the correct place at the correct time. Non-deterministic delays, such as waiting for input data from an external device, must be handled at the user-circuit level.

The PE router component is a 5x5 crossbar with a register on each output. In a user circuit, all potential communication paths (wires) are known at compile time and are statically scheduled. Each communication must have a timeslot in each router between its source and sink(s). Each router follows an individual schedule to pass data between its five links; five messages may be passed in a single cycle provided the destinations are all different. The router can also delay a message by holding the value in the output register. This will be used to avoid data collisions. Given a circuit, the tools determine when every signal needs to be generated, and when it needs to arrive. This information is used to create the configuration (schedule) for each router.

The processor core is shown in Figure 2. It is a simple processor with several additions. The ALU and data memory D are used to implement user-level circuit behaviour. Node memory R is used to temporarily store ALU results—emulating a wire for data used in the same user clock cycle and a flip-flip for data needed in the next user cycle. Node memory X is used by the router to store values for the core without being perfectly synchronized to its operation. There are also four buffered, direct links to adjacent PEs (node memory N, S, E, and W) which are used as a lower-latency alternative to the router for short connections.

Computational circuits will also contain single-bit signals, such as control logic, that do not map well to word-oriented processors. Our architecture provides a mechanism to deal with such signals so they do not degrade the performance of a circuit. The PE core in Figure 2 includes a PLA for generating and communicating these bit signals. This logic is not time-multiplexed. The PLA communicates with neighbouring PEs over dedicated wires, and is connected to the PE core datapath for decision-making logic. The tools must identify bit-level signals in the source circuit and generate the configurable logic for it. If a circuit stresses the capacity of these resources, the tools can fall back to processor instructions to ensure there is no strict capacity limitation in the architecture.

Initially all buses are 32 bits wide, the node memory X and R are each 16x32-bit, the data memory is 8 kB (2048x32-bit), the remaining node memories are 4x32-bit, and we assume the ALU contains a single-cycle multiplier. These are all parameters that will be changed and tested as future work.

III. Architecture Simulator

This section presents the design of a simulation platform for our architecture. The simulation platform was designed to be flexible to facilitate the exploration of the various architectural parameters. The simulator currently implements all features of the architecture except the bit-level resources (the PLA in Figure 2). This is not critical because the PLA merely accelerates bit-level operations that can also be executed (inefficiently) in the ALU.

To speed up the simulator, the code for each PE is compiled directly to the host processor using a native compiler. This gives the fastest possible simulation (a good thing when simulating 1,000 cores on a desktop computer), but requires the tool flow to insert `cycle()` calls in the PE code to denote instruction boundaries. This means that the simulator does not run the same binary as the proposed architecture, but it is still cycle-accurate. The simulator switches to a different PE when it encounters a `cycle()` call.

The simulator executes each system (3 GHz) clock cycle in three phases:

1) Copy all registers (in the router and the PE core) to temporary locations for reading. This simulates a rising clock edge where all registers are latched and stable for the duration of the clock cycle. The next steps write to the original registers and read from these stable values. PEs can be simulated in any order within each cycle.
2) Execute one system cycle in every PE core. The simulator passes program control to the PE, and allows it to run until `cycle()` is called. The simulator uses the POSIX `swapcontext()` call to implement low-level, lightweight context switching among PEs using a deterministic scheduler. This is done because full context-switching between thousands of threads is wasteful and because the OS scheduler is non-deterministic.
3) Execute one system cycle in each PE router. The router context memory is advanced to the next configuration, and values on each communication link are copied to the output registers according to the new configuration.

The simulator simulates input and output by attaching communication channels to files instead of other PEs (usually to PEs along the edge of the device). The above three steps are executed until the data from all input files is depleted.

IV. Tools

This section outlines our method to automatically map computational circuits to our architecture. Currently the tools are incomplete, so designs must be hand-mapped to our architecture. The example design presented in Section V was hand-mapped following the method presented here. The flow is a modified version of an FPGA CAD flow because many similar problems are being solved. The input is a circuit, specified in Verilog, and the output is a bitstream for the architecture or simulator. The tool flow is separated into five steps: **Parallelize**, **Combine**, **Placement**, **Schedule and Route**, and **Code Generation**.

The **Parallelize** step parses the Verilog source and partitions the circuit into a large number of parallel operations, where each parallel partition *could* be implemented on an individual PE. The tool begins by constructing a graph that represents the control and data flow of the design at the behavioural level. To construct this graph, the circuit is transformed into an equivalent network of combinational logic by separating the circuit so that flip-flop outputs appear as virtual inputs to the circuit, and all flip-flop inputs appear as virtual outputs. In one user clock cycle, operations (nodes in the graph) and communication (edges) are mapped to processors and interconnect. At this stage, the tools assume a latency of 1 for each communication hop. This determines a lower bound for the number of system clock cycles (3 GHz) required to implement a user clock cycle. The **Schedule and Route** tool may insert additional system cycles (by increasing edge delays or adding operations) to resolve resource conflicts, so the number of system clock cycles in a user clock cycle may increase.

At this point, the tool separates the bit-level and word-level operations, and could duplicate logic to improve performance. User instantiated memory is mapped into the 8 kB data memory (see Figure 2). To handle a larger user memory, the tools must coordinate several PEs to implement the memory required.

It is anticipated that the number of parallel operations in a computational circuit will exceed the number of PEs available. The **Combine** step groups the parallel operations from the **Parallelize** step into code clusters, with one cluster for each PE. **Combine** is similar to clustering in CAD tools: short nets and heavy communication nets are absorbed into a single PE and kept off the communication network. It is this ability to combine code that trades off area (PEs) for performance and achieves a soft capacity limit.

The **Placement** step assigns the code clusters to physical PEs so that communication is minimized. Using a rough code schedule, which is computed for each code cluster, the placer prioritizes which communication paths to minimize. This step is similar to FPGA placement, so a modified version of the VPR [5] simulated annealer can be used. However, the cost function will have to be modified due to the pipelined interconnect in the architecture. Since all communication links are time-multiplexed, the Manhattan distance from every source to every sink is a good estimate for communication latency. This is different from traditional bounding-box approaches where the location of the source is less important.

After placement, the **Schedule and Route** step orders the code on each PE and routes data between the PEs for correct overall operation. There are several factors not present in current FPGA architectures that need to be considered here:

1) The architecture contains direct links to neighbouring PEs that avoid the routing network.
2) The architecture contains both bit-level and word-level resources for routing data.
3) The routing network is time multiplexed, so the Schedule and Route steps must work together. For example, due to congestion, data may be delayed along the routed path for a few cycles. If the router does this the scheduler must delay the code that depends on the data by the same number of cycles, which may affect when subsequent output results are available for routing.

Schedule and Route must iterate to finalize the order of the code and router schedules. The scheduler must accept a partially completed route (or no route on the first pass) and schedule the code within each PE to minimize the user clock period. The router must take the code schedule and route all sources to all sinks over the time-multiplexed communication network. If the router succeeds, this step is complete. If it fails, the scheduler is re-invoked with a partially completed route. Excessive iterations can be avoided by always making forward progress (*i.e.*, previous connections are never ripped up or re-routed).

In the final step, **Code Generation**, the code assigned to each PE is compiled for the target architecture, and the routing schedule for each PE is assembled and packed into a bitstream.

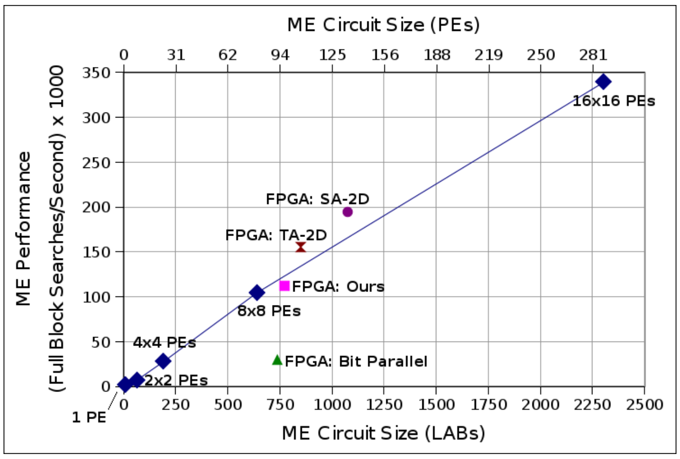

Fig. 3. Graph of motion estimation performance vs. size

V. Example: Motion Estimation

To the study the architecture, a motion estimation (ME) algorithm was implemented based on an ASIC ME circuit [6]. Motion estimation is computationally intensive and highly parallelizable, yet requires a distributed control and communication network, so it is a good candidate for a computational circuit. Motion estimation takes a reference block R of an image (16x16 pixels) and sweeps a search space M (32x32 pixels) of a second image to find the best match of that block. This is called a full block search. For each position, the sum of absolute differences (SAD) is computed, and the lowest SAD is the best match. The SAD at search position (x, y) is computed using:

$$\sum_{i=0}^{15}\sum_{j=0}^{15} abs(R_{i,j} - M_{x+i,y+j})$$

We first mapped the ME circuit to a 16x16 PE array. Then, the code from each group of 4 PEs was inlined onto a single PE to create the 8x8, 4x4, 2x2, and finally 1 PE implementations. This scaling demonstrates the performance vs. capacity tradeoff of our architecture, and can be easily done automatically by a tool. Finally, to compare circuit size between our architecture and FPGA-based ME circuits, we computed the equivalent number of FPGA LABs based silicon area estimates where one PE is equal to 8 LABs (Stratix-III).

Figure 3 plots the performance vs. area of five scaled versions of a motion estimation circuit implemented on our architecture (1, 2x2, 4x4, 8x8, 16x16). It also plots the same circuit on the FPGA (FPGA: Ours), and the results of three other FPGA-based motion estimation designs (Bit parallel [7], TA-2D [8], and SA-2D [8]). The TA-2D algorithm is similar to the implemented algorithm (FPGA); they both use data movement and an adder tree to sum the SADs.

The performance and area requirements for the 8x8 PE implementation on the proposed architecture are comparable to the FPGA implementation, and also comparable to the published research. This means that 8x8 PEs of the proposed architecture achieves a similar area and speed as an FPGA, which gives us confidence in proceeding with further testing.

A linear interpolation between the 2x2 and 4x4 datapoints gives a point at 77 LABs operating at 12437 searches per second. This matches the stated objectives of 10x the density and $\frac{1}{10}th$ the performance. Unfortunately, this point is not attainable for this ME circuit since PEs are discrete entities that cannot be subdivided.

VI. Conclusion

In this paper we have presented a custom architecture for simulating/executing computational circuits. The architecture is based on an array of processors, and uses time-multiplexing to trade circuit capacity for performance. We also presented an outline of a fast tool flow for quickly compiling computational circuits to the architecture.

We have presented an example implementation of motion estimation that was hand-mapped to the architecture. The motion estimation circuit showed that the architecture has the desired capacity vs. performance tradeoff, and also that it is able to match the speed of an FPGA at a lower capacity. This demonstrates that the goals of the project are reasonable.

For future work, we plan to implement the automatic tool flow so that the two remaining goals (10x tool speed over an FPGA CAD tool, and 10x the capacity of an FPGA) can be achieved.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and CMC Microsystems.

References

[1] M. Larouche. (2007, January) Infusing speed and visibility into ASIC verification. [Online]. Available: http://www.synplicity.com/literature/whitepapers/pdf/totalrecall_wp_1206.pdf

[2] Tenison Design Automation. (2006, October) Tenison VTOC RTL to SystemC/C++ synthesis. [Online]. Available: http://www.tenison.com/images/stories/Brochures/vtoc-10-25.pdf

[3] W. Snyder. (2007, June) Verilator-3.652. [Online]. Available: http://www.veripool.com/verilator_doc.pdf

[4] D. Greaves, “A verilog to C compiler,” in *Proc. Rapid System Prototyping (RSP)*, 2000, pp. 122–127.

[5] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” in *Proc. Field-Programmable Logic and Applications (FPL)*, 1997, pp. 213–222.

[6] N. Roma and L. Sousa, “A new efficient VLSI architecture for full search block matching motion estimation,” in *Proc. Very Large Scale Integration of Systems On a Chip (VLSI-SOC)*, 2002, pp. 253–264.

[7] C. Wei and M. Z. Gang, “A novel SAD computing hardware architecture for variable-size block motion estimation and its implementation with FPGA,” in *Proc. Conference on ASIC (ASICON)*, vol. 2, Oct 2003, pp. 950–953.

[8] B. M. Li and P. H. Leong, “Serial and parallel FPGA-based variable block size motion estimation processors,” *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 77–98, 2008.