Extending Course-Grained Reconfigurable Arrays with Multi-Kernel Dataflow

Robin Panda*, Aaron Wood*, Nathaniel McVicar*,Carl Ebeling†, Scott Hauck*
*Dept. of Electrical Engineering and †Dept. of Computer Science and Engineering
University of Washington, Seattle, WA 98195
email: *{robin, arw82, nmcvicar, hauck}@ee.washington.edu †ebeling@cs.washington.edu

Abstract—Coarse-Grained Reconfigurable Arrays (CGRAs) are a promising class of architectures for accelerating applications using a large number of parallel execution units for high throughput. While the model allows for tools that can automatically parallelize a single task across many processing elements, all processing elements are required to perform in lock step. This makes applications that involve multiple data streams, multiple tasks, or unpredictable schedules more difficult to program and inefficient in their use of resources. These applications can often be decomposed into a set of communicating kernels, operating independently to achieve the overall computation. Although competing accelerator architectures like Massively Parallel Processor Arrays (MPPAs) can make use of this communicating processes model, it generally requires the designer to decompose the design into as many kernels as there are processors to be used. While this is excellent for executing unrelated tasks simultaneously, the amount of resources easily utilized for a single task is limited.

We are developing a new CGRA architecture that enables execution of multiple kernels of computation simultaneously. This greatly extends the domain of applications that can be accelerated with CGRAs. This new architecture poses two problems that we describe in this paper. First, the tools must handle the decomposition, scheduling placement and routing of multiple kernels. Second, the CGRA must include new resources for coordinating and synchronizing the operation of multiple kernels. This paper is a digest of the project's previously published results [17,18,19,20,21].

I. INTRODUCTION

Field programmable gate arrays (FPGAs) have long been used for accelerating compute intensive applications. FPGAs avoid the high development and fabrication costs of custom ASICs, and are far faster than a general purpose CPU for many parallel or pipelined applications. The FPGA's programmability comes at a cost, though. The functionality of an FPGA is implemented using lookup tables to compute every simple Boolean function. Signals between operations require large and complex routing switches and common arithmetic operations are programmed down to each individual bit instead of a word at a time. These inefficiencies result in lower speed and higher power consumption compared to an ASIC implementation.

The vast majority of most computations are multi-bit operations. Coarse-grained configurable arrays (CGRAs) take advantage of this by sharing configuration data across word-wide operations, which mitigates much of the inefficiency of an FPGA. CGRAs use coarse-grained computation units like ALUs and multipliers instead of LUTs, and move data as words instead of bits. Some support for configurable Boolean functions is usually included to allow for control functionality.

Several different CGRAs have been developed, including MorphoSys [1], ADRES [2], VEAL [3], and Mosaic [4] which have a sea of ALUs connected with an word-based FPGA-like interconnect. Time-multiplexing computation and routing resources decreases logic area and energy required for applications and speeds execution [5], so we assume they have this capability. Configurations are generated using FPGA-like placement and routing algorithms such as SPR [6] for automatic parallelization. The move to word-width resources is assisted by moving the programming language from Hardware Description Languages (HDL) typically used for FPGAs to more C-like alternatives. The FPGA-like tools and configuration of CGRAs can use the parallelism and pipelining in the algorithm to map a single task to several processing elements automatically. However, the design is completely scheduled at compile time so they are poor at handling complex control flow and require predictability from their workflow.

The traditional processors used in Massively Parallel Processor Arrays (MPPAs) are naturally much better for applications with more complex control structures. MPPAs like ASAP2 [7] Ambric [8], and RAW [9] contain independent processors that communicate by passing messages. Each is programmed by the user individually, with a traditional instruction set. These processes use only memory local to each processor, and explicit communication over the network, instead of the large shared memory of a multicore CPU. These MPPAs are great for control and variable workloads, but the programmer is required to manually split a computation into 100's of CPU-sized programs.

The aim of our research is to extend CGRA architectures with some of the flexibility that is inherent in MPPA architectures. This will provide some of the control and workload flexibility of MPPAs, but with individual tasks still automatically parallelized over multiple processing elements. We use a CGRA's lock-step operation for each independent task, allowing processing elements and communication resources to execute on a fixed schedule produced at compile time. We synchronize between kernels via MPPA-style message passing, allowing processing regions to operate at their own rate, and possibly supporting branching and other data-driven control flow inside individual kernels.

Creating the extended architecture requires several modifications over a basic CGRA. The programming language must be extended to specify multiple kernels and describe their communication. The mapping tools must allocate the kernels to regions of the CGRA so each kernel has its own region that can execute according to its own schedule.

Supported by grants: NSF #CCF-1116248 and DOE #DE-FG02-08ER64676

Additionally, the tools must map the new inter-kernel communication. Finally, the CGRA architecture must provide the additional resources that allow the configured regions to operate independently using different schedules. Because inter-kernel communication does not occur according to a schedule, the hardware must now be able to detect when the communication is actually occurring, and have each kernel properly react to full and empty communication streams.

II. Previous Architecture

A generalized CGRA, such as our base architecture Mosaic, is composed of various word-width functional units, which can include ALUs, shifters, or other special-purpose processing elements [10], connected with a programmable, word-width interconnect (Fig. 2). It is often useful to include some LUTs and single bit communication channels to form a basic FPGA within the architecture for control and bitwise logic [5]. All memory is local, like in an FPGA, with no native coherency mechanisms for shared memory [4]. Block memories are typically explicitly managed by the application code, while registers required for timing and synchronization are managed by the CAD tools as necessary.

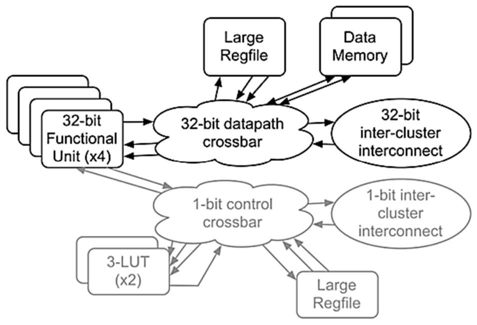

Figure 1. CGRA execution cluster components

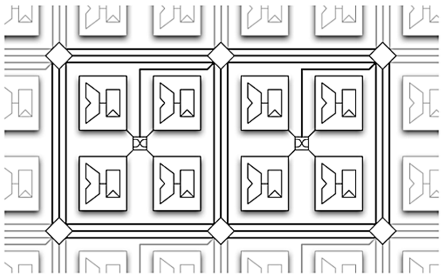

Figure 2. CGRA showing two clusters in a grid interconnect

The configuration, consisting of the functional units' opcodes and addresses requested from register banks, is sent to the functional units each cycle. The interconnect (Fig. 3) is controlled in a similar manner. An incoming bus (**A**) fans out to multiplexers (**B**), (**C**), (**D**) in all other directions. A phase counter cycles through the configurations in the configuration memory (**E**) to select the appropriate inputs to the multiplexer. This loads the different routes in configuration memory, synchronized with the incoming data, to time-multiplex the interconnect. After passing through the multiplexer (**B**), the bus is registered before being driven across the long wires to the next switchbox. Resources like the configuration memory, decode, and clock gating are shared by all the wires in a bus.

There are two main ways to think about these configurations. The most straightforward is as a set of predefined contexts, cycling after each clock cycle to simulate additional hardware, similar to a word-width version of a Tabula 3PLD [11]. The other is as an instruction word for the core of a clustered VLIW, but without the complex instruction processing components required for conditional branches.

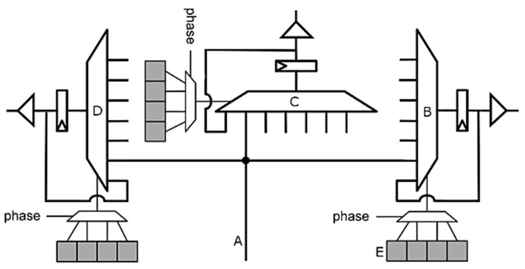

Figure 3. Switchbox schematic

An accelerator is useless if it is too difficult to program, so the programming mechanism is an important consideration. Traditionally, FPGAs are programmed using a hardware description language (HDL), but newer C-like methods have been developed. The majority of word-width accelerators have programming languages that are like standard programming languages, usually with extra directives to specify communications, placement, or similar. For example, the Macah language for Mosaic [12] has specific directives for specifying a kernel, its input and output streams, and helping extract parallelism. The programmer can flag inner loops where the execution flow is always the same so the compiler can completely flatten it for extra pipeline parallelism.

Extracting parallelism from this C-like code is not a trivial task. A common step in this process is producing a dataflow graph (DFG). The DFG contains all the operations as nodes, and the edges represent dependencies. Some accelerators can execute this DFG directly [13], but most need more processing to convert it into instructions. In the Scheduling, Placing, and Routing (SPR) tool for Mosaic [6] each operation is mapped to a functional unit in an appropriate context and the edges are used to configure the routing. To maximize throughput and the amount of processing that can be accomplished, smaller loops are flattened to utilize additional processing elements. The resulting dataflow graphs can be hundreds of cycles long. These are scheduled to the time-multiplexed computation and routing resources using Iterative Modulo Scheduling [6], a technique for loop pipelining commonly used in VLIW compilers. In the case of CGRAs, this scheduling must be done in both time and space.

In Iterative Modulo Scheduling a small number of instructions are iterated repeatedly to execute a much longer pipeline. Table I shows how 6 instructions, where each is dependent on the previous one, could be scheduled across 3 execution units. This schedule executes all 6 instructions in only 2 clock cycles; however, it does still take 6 instructions to complete one iteration (in bold). The 2-cycle length of this schedule is known as the iteration initiation interval (II). If instruction 6 in the table is the output stage, it follows that the schedule shown can only produce one value every other cycle because that instruction is only executed once per initiation interval. Therefore, the schedule with the maximum throughput would be to fully pipeline the entire graph across the 6 processors and iterate one cycle repeatedly.

Table I. Simulation configuration

Phase	Processor 1	Processor 2	Processor 3
A	**Instruction 1**	Instruction 3	Instruction 5
B	**Instruction 2**	Instruction 4	Instruction 6
A	Instruction 1	**Instruction 3**	Instruction 5
B	Instruction 2	← **Instruction 4**	Instruction 6
A	Instruction 1	Instruction 3	**Instruction 5**
B	Instruction 2	Instruction 4	**Instruction 6**

Arrow represents additional dependency

However, in actual applications, one iteration of a loop often depends on the result of an instruction in a previous iteration. If instruction 2 of one iteration depends on the result of instruction 4 of the previous iteration, (as shown by the arrow) this schedule will not work because both instructions run at the same time. This means our schedule must be at least 3 phases long. This will allow instructions 3 and 4 to run and then provide the results to instruction 2 of the next iteration on the following cycle. In this situation, we say that the recurrence II of the algorithm is 3, and this will be the optimal II for this dataflow graph.

The computing model of CGRAs is promising because tools such as SPR, can automatically spread a single computation across a large array of computation units from only a single program. However, many common styles of computation run into problems with this model:

- *Multiple tasks sharing the hardware share a single static schedule*. Because CGRA tools generally take only a single computation and spread it across the entire array, we must combine all tasks into one integrated computation. Thus, multiple independent tasks (such as processing on different streaming inputs), or multiple tasks for a single computation (such as the stages in an image-processing pipeline) must be combined into one loop. This is time-consuming, inefficient, and hard to support. On the other hand, this lockstep operation is what allows the architecture and CAD tools to be as efficient as they are.
- *They use predication for data-dependent execution*. Individual tasks usually have data-dependent operation, such as the choices in an IF-THEN-ELSE construct, or different modes of processing at different times in a computation (such as the phases in K-Means clustering). Since a CGRA requires every operation to occur at exactly the same time and place in each iteration, CGRAs use predication to handle data-dependent operation. This means that a large fraction of the issue slots in each iteration are consumed by operations that are simply predicated away.
- *All schedules run at once must be the same length*. Computation pipelines often have some tasks that are more complex, and therefore have a longer recurrence loop that limits their natural computation rate. Every task has a minimum achievable II, but a CGRA generally forces all tasks to use the same II (often the highest II of the tasks) to coordinate communication between them. If communication rates were identical, this is not a big problem. For computations with long tasks that are executed sporadically (such as PET [14]), or long tasks on lower-bandwidth paths in the computation, this imposes a significant performance penalty on the entire computation.

III. MPPAs

One can think of a Massively Parallel Processor Array (MPPA) as a CGRA where the hundreds of ALUs of the CGRA are replaced with small processors with full branching capability independent of other functional units. This makes it relatively inexpensive to handle small control tasks on chip, because predication is not required. The processors are individually programmed, often in a traditional language. However, since the processors and network are no longer executing in a lock-step manner, this complicates the coordination of the architecture. The interconnect multiplexers can no longer select based simply on clock cycle, and all memory blocks are coupled tightly with an individual processor or have a dedicated processor to sequence data.

MPPAs are dynamically synchronized by using communication channels with flow control between the processors. This flow control identifies when a valid data word is on the channel downstream and provides backpressure upstream. It is straightforward to understand that processors should stall until they see valid data arrive. However, if the process transmitting data can transmit faster than the receiver can receive, signals from full buffers prevent the sender from sending when the receiver is not ready. In this manner, the data synchronizes processing instead of a global program counter. This can be accomplished with a handshake adding two signals. One simply acts like an additional data bit, but represents whether or not the other 32 bits are *valid* data. The other goes in the opposite direction and indicates if the receiving processor or interconnect stage is *ready* to consume data.

The Ambric MPPA has such a network with routing, configured at compile time, that is used for the duration of execution. This network passes data between special sets of registers. These are wired together through a programmable interconnect (not shown) that appears, logically as shown in Fig. 4. A sequence of words, **A**, **B**, **C**, **D**, and **E** are being sent in the channel. When the receiver is no longer able to receive, it deasserts *ready* and the next register upstream retains its data. Because the *ready* will take time to propagate upstream, each register will need to store additional data when stalling. In this case, the special register set is storing both **A** and **B** while the deassertion of *ready* propagates backwards.

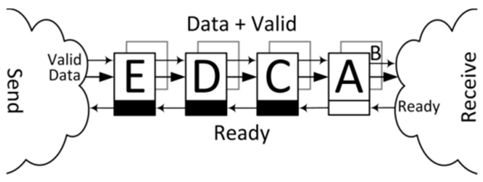

Figure 4. Dedicated register channel logical view

While some MPPA architectures such as RAW have included full dynamic communication routing, RAW required additional networks to avoid deadlock. More recent architectures, such as ASAP2 and Ambric, configure all their routing statically at compile time. Because the processors themselves are also configured at compile time this does not result in a significant loss in flexibility. In an architecture with hundreds of processors, some of them can be programmed by the user to act as soft-routers for the remaining cases [15].

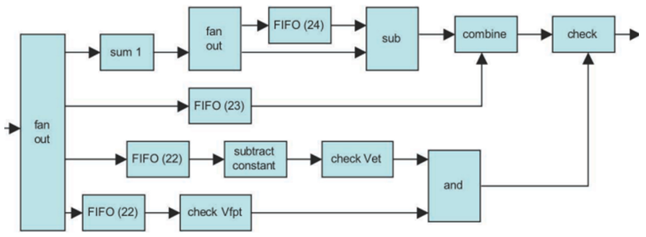

Figure 5. Block diagram for PET thresholding computation on an MPPA

Because an MPPA has multiple, independent processors loosely synchronized through communication channels, they avoid most of the previously mentioned problems with a CGRA. Each processor can have its own schedule, so different computations can have different schedule lengths, and independent tasks do not need to be combined into a single program. In addition, since the processors have true program counters, they can use branching for IF-THEN-ELSE constructs, and looping for supporting different modes. However, MPPAs have their own challenges:

- *MPPAs require the programmer to manually split computations into processor-sized chunk.* CGRAs leverage their system-wide synchronous behavior to provide tools that can automatically spread a computation across numerous processors. Thus, tools like SPR can take a single task and efficiently spread it across tens to hundreds of CPUs. MPPAs, with their more loosely coupled CPUs, do not provide the same functionality or tools, and instead force the application developer to write programs for each individual processor in the system. This is a huge task. For example, in [14], mapping a simple running sum threshold test to the Ambric MPPA required manually breaking the short loop into 8 processors and 4 FIFOs, all manually specified (Fig. 5). This implementation still took 6 clock cycles per input where a CGRA only needs one or two.
- *MPPAs generally keep most computations and results local to a single processor.* Although there are abundant resources connecting the individual processors together, communication between two processors in an MPPA is still noticeably more expensive than between CGRA ALUs operating in lockstep. This limits the achievable pipeline parallelism for a given task; thus many processors are lightly loaded while the processor with the most complicated task runs constantly [16].

IV. Multikernel Hybrid

CGRAs prove to be quite good at utilizing many processing elements for a single kernel of execution, but are inefficient for control and handling multiple tasks in an application. MPPAs are great for control and a large number of tasks and/or applications, but are less efficient for individual pipelined tasks that are more difficult to spread across multiple processing units. Our goal is to merge the benefits of these two technologies. Our devices is a 2D computing surface, like a CGRA or MPPA, that can be split (on a per-mapping basis) into multiple communicating kernels. Each individual kernel computes in the CGRA model, where multiple processors operate in lockstep performing a unified modulo schedule, created by SPR. For control-dominated kernels we can create a 1-processor region, and compile the kernel to it using standard VLIW techniques [17], making use of branching and other standard control flow operations. Between the kernels we communicate via MPPA-style messages, with flow control to ensure that writes to full streams, or reads from empty streams, stalls that kernel until the condition is resolved. Users of the system write multi-kernel computations in Macah, and they are automatically translated to implementations on the hybrid device.

Achieving this vision requires innovation on multiple fronts. The control domain size and placement must have some flexibility to maximize the utilization of the compute resources for designs with wildly different kernel complexities. The CGRA model requires architectural and tool modifications to execute different, yet related tasks simultaneously. Within a control domain, routing and computation should still operate according to compiler-determined schedules for efficiency. Communication between control domains must be dynamically flow-controlled like in an MPPA and not scheduled. In the sections that follow we discuss several of these challenges, and the techniques we are developing to address them.

A. Language and compiler modification

The Macah language must be modified so that multiple kernels can be specified along with their communication patterns. Fig. 6 shows an example of 3 kernels communicating. Kernel **A** handles input data and sends outputs to kernel **B** and **C**, and **B** also sends to **C,** who sends out the final answer. Since our original Macah language already had a single kernel that communicated with the external environment via streams, it was relatively simple to extend this model to have multiple kernels connected by explicit streams. The new compiler takes multi-kernel code and it emits a separate DFG for each kernel to downstream tools. These modifications are detailed in [18].

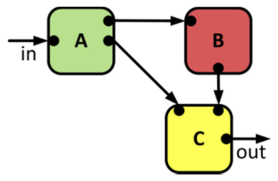

Figure 6. Control domain communication pattern

B. Architecture for dynamic communication

The hardware must be augmented to handle dataflow communication within the CGRA. Instead of communicating according to a schedule, the interconnect must be able to handle excess or insufficient data and signal producers and consumers accordingly. In keeping with the configurable nature of the device, as little special-purpose hardware as possible should be added.

Specific details on how resources in a control domain are used to implement flow-controlled communication between unrelated control domains are described in [19]. Several solutions were evaluated for area and energy as a function of utilization. Most types of dataflow control from existing MPPAs were considered, including several ways of building distributed FIFOs out of existing interconnect components.

All solutions involved the addition of at least two bits, one indicating the presence of data and one in the reverse direction indicating if the receiver would be able to consume data on the following clock. Because these communication channels needed to remain independent of the execution of any control domains they passed through, the channels were scheduled for all time-slots.

Because it is very difficult to write code that results in a 1-cycle-long schedule, it is rare to encounter a kernel that can send or receive on every clock. If the maximum bandwidth required is one word every-other clock or less, then half-bandwidth channels are sufficient. These ½ bandwidth channels can be implemented using little more than the word-wide channel and two 1-bit channels that already exist in the interconnect. For longer channels, and at high channel utilization, the best solution is creating small, dedicated FIFOs at the receiver and using credits accounted at the sender for backpressure. However, we have found that for most designs our floorplanner (discussed below) can place communicating kernels next to each other, making typical inter-kernel streams very short. Overall, the added expense of providing ½ bandwidth channels amounts to only a 0.16% increase in full-chip area-energy product, a trivial cost.

C. *Dataflow-controlled execution*

Individual control domains must be able to modify their execution according to the signaling from the communication channel. If there is no data available for a read in the next instruction, the control domain should stop executing (stall) until the data is available. Similarly, if a channel fills up with too much data, the control domain should stall until space frees up in the FIFO. As an example, in Fig. 6, **B** cannot work until **A** provides the data to it. Similarly, anything executed in **C** before **A** and **B** have provided appropriate inputs is useless at best and often results in improper functionality.

We design each end of a dynamic communication channel to have a streaming communication port (black circles in Fig. 6) to interface with the control domain and buffer data to or from the channel. Fig. 7 zooms in to the top right corner of control domain C to show a read port and a write port. Each read or write port takes a predicate from within the control domain that indicates whether the kernel actually wishes to perform the scheduled communication that cycle. Stalls occur when an active port's predicate is true, and there is no data in the buffer for reads, or there is a full buffer for writes. In these cases, the stream port indicates to the control domain that it should stall and not proceed to the next instruction. If any stream port within a control domain signals a stall, the entire control domain halts at the exact same place in the schedule. Therefore, we also need a network to aggregate all stall signals from the ports and deliver it to the mechanism that actually stalls the control domain.

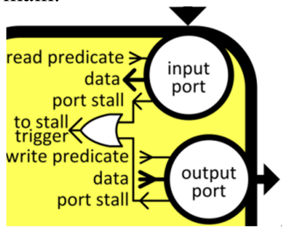

Figure 7. Stream port interfaces

Implementation details for these mechanisms are covered in [20]. Rather than build a new configurable clock-gating network, the stalls are implemented by stopping the phase counter and adding an additional stall configuration in the configuration memory. This configuration instructs registers to hold their value and disables writes to register files, which gates their clocks and effectively freezes the control domain. This increase area by 0.86% and increases energy by a maximum of 3.7%, which is significantly less than that used by a configurable clock gating network.

All processors in a control domain must remain on the same phase or data will be lost. This means that the stalls must be coordinated to trigger on the same clock cycle, regardless of stall source. Accomplishing this requires a new network. As this network will always be used, there is nothing to gain by building it from existing configurable components. However, by making this network programmed by SPR, only two single-bit communication links are required between processors (one in each direction). The logic for implementing this function is only a handful of program bits, some ORs and a small shift register for each processor. This amounts to an area overhead of only 0.75%.

The latency of this stall propagation is not negligible, especially for a kernel spread over 10's to 100's of processing units. Thus, there will be a noticeable latency from when a stall condition is detected to when it can actually take effect. However, we have developed techniques to detect the stall in advance of the communication that actually triggers it. This includes saving space in outgoing FIFOs via a "high-water mark" to maintain space for writes occurring during the stall latency, and separating the empty stream check from the data usage in reads. Better operation can also be supported by adding maximum read and write frequencies determined by the CAD tools. This transformations make the prediction operations into standard operations, which can allow the modulo scheduling algorithms in SPR to automatically hide much of these latencies.

D. *Resource assignment*

New floorplanning/global placement tools [21] re required to assign specific chip resources to each kernel in the computation. This operation is broken into two phases: an optimal resource assignment algorithm that sizes the regions assigned to each kernel to maximize throughput, and a simulated-annealing based global placer to break the computing surface into contiguous regions for each kernel.

For resource assignment we start by allocating each kernel its minimum resources, dictated by the number of non-time-multiplexed elements (memory capacity and the like), as well as the maximum time multiplexing allowed by the architecture. We then seek the bottleneck in the system by computing the steady-state operating rate of the mapping. This employs per-stream reading and writing rates, which establishes a limit on the performance of interconnected kernels, and transitively on the entire connected computation. Once the bottleneck is identified, exactly enough resources are added to speed up those kernel(s), and we iterate back to finding the new bottlenecks. This continues until either all the resources in the system are assigned, or the minimum II of some bottleneck kernel is reached, indicating the maximum possible performance has been achieved. We have demonstrated that this simple procedure takes less than a second on realistic examples, and produces optimal results.

Resource assignment determines the number of processing elements to assign to each kernel, but not their actual position on the chip. For that, we use a simulated-annealing based global placement phase. The heuristic attempts to minimize two cost functions to optimize processor assignment for this step. A bounding box is drawn around all the resources for any pair of control domains that communicate with one-another for one cost representing the inter-kernel communication. A similar box is used for each individual control domain (or the perimeter of the domain if larger). This represents the intra-kernel costs of the current placement

Two floorplans from example applications are shown in Fig. 8; each color is a separate CGRA region implementing a substep (such as interpolation or gradient-correction for demosaicing a Bayer filter image) and each square is a processor. Within a region, SPR is used to handle data routing, instruction scheduling, and detailed placement, just like a normal CGRA.

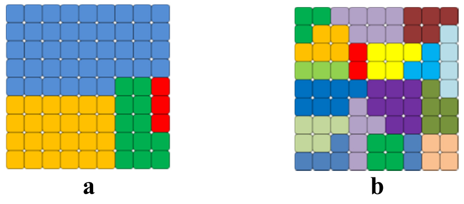

Figure 8. Floorplans for **a**: Bayer filter and **b**: discrete wavelet transform

V. Conclusion

While MPPAs and CGRAs each have their strengths, the details of their operations hold them back from supporting important styles of operation. In this paper we have presented a novel architecture and computation model. We use multiple communicating kernels like an MPPA, allowing for differing communication rates and IIs to be balanced to achieve the best possible overall performance. However, instead of requiring a user to program at the individual processor level, difficult for a streaming computation run on hundreds or thousands of processors, we use CGRA techniques to spread computation-heavy kernels across multiple processors.

As part of this paper we have mentioned many of the challenges that must be faced to create such a system, and presented approaches to solve these issues. We are currently developing the Mosaic 2.0 architecture, which should bring this computation substrate to future stream processing applications. We believe it can provide very power-efficient computation, with better area and power consumption than multicore systems (with overly complex dynamic scheduling support), FPGAs (with their bit-oriented inefficiencies), and MPPAs (which have difficulty spreading computations across multiple processors).

References

[1] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh and E. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *Computers, IEEE Transactions on,* vol. 49, no. 5, pp. 465 -481, May 2000.

[2] B. Mei, S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2003.

[3] N. Clark, A. Hormati and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, Washington, DC, USA, 2008.

[4] B. Van Essen, R. Panda, A. Wood, C. Ebeling and S. Hauck, "Managing Short-Lived and Long-Lived Values in Coarse-Grained Reconfigurable Arrays," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2010.

[5] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling and S. Hauck, "Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays," in *Field Programmable Logic and Applications, (FPL) International Conference on*, 2009.

[6] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling and S. Hauck, "SPR: An Architecture-Adaptive CGRA Mapping Tool," in *FPGA '09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2009.

[7] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia and B. Baas, "A 167-Processor Computational Platform in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of,* vol. 44, no. 4, pp. 1130 -1144, April 2009.

[8] Ambric, Inc, Am2000 Family Architecture Reference, 2008.

[9] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *Micro, IEEE,* vol. 22, no. 2, pp. 25 - 35, Mar/Apr 2002.

[10] B. C. Van Essen, R. Panda, A. Wood, C. Ebeling and S. Hauck, "Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Array," in *FPGA '11: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011.

[11] T. R. Halfhill, "Tabula's Time Machine," *Microprocessor Report,* Mar. 29 2010.

[12] B. Ylvisaker, A. Carroll, S. Friedman, B. Van Essen, C. Ebeling, D. Grossman and S. Hauck, "Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators," Dept. Elect. Eng., Univ. of Washington, Seattle, WA, Tech. Rep., 2008.

[13] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald and W. Yoder, "Scaling to the end of silicon with EDGE architectures," *Computer,* vol. 37, no. 7, pp. 44-55, July 2004.

[14] M. Haselman, N. Johnson-Williams, C. Jerde, M. Kim, S. Hauck, T. Lewellen and R. Miyaoka, "FPGA vs. MPPA for Positron Emission Tomography pulse processing," in *Field-Programmable Technology, 2009. (FPT) International Conference on*, 2009.

[15] R. Panda, J. Xu and S. Hauck, "Software Managed Distributed Memories in MPPAs," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2010.

[16] Z. Yu and B. Baas, "High Performance, Energy Efficiency, and Scalability With GALS Chip Multiprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 17, no. 1, pp. 66 -79, Jan. 2009.

[17] N. McVicar, "Architecture and Compiler Support for a VLIW Execution Model on a Coarse-Grained Reconfigurable Array," M.S. Thesis, Dept. Elect. Eng., University of Washington, Seattle, 2011.

[18] A. Knight, "Multi-Kernel Macah Support and Applications," M.S. Thesis, Dept. Elect. Eng., University of Washington, Seattle, 2010.

[19] R. Panda and S. Hauck, "Dynamic Communication in a Coarse Grained Reconfigurable Array," in *Field-Programmable Custom Computing Machines (FCCM), IEEE 19th Annual International Symposium on*, 2011.

[20] R. Panda, C. Ebeling and S. Hauck, "Adding Dataflow-Driven Execution Control to a Coarse-Grained Reconfigurable Array," in *Submitted to: Field Programmable Logic and Applications, (FPL) International Conference on*, 2012.

[21] A. Wood, A. Knight, B. Ylvisaker and S. Hauck, "Multi-Kernel Floorplanning for Enhanced CGRAs," in *Submitted to: Field Programmable Logic and Applications, (FPL) International Conference on*, 2012.