Versat, a general purpose hardware accelerator

***Abstract*—This paper introduces Versat, a novel architecture for a general purpose reconfigurable hardware accelerator. The motivation for the new architecture is implementing fast self-reconfiguration and simplifying programming. Fast self-reconfiguration is implemented using finest-grain partial reconfiguration. Programming is simplified by adopting a fully connected graph topology. Preliminary experimental results are presented.**

I. Introduction

In this work we present Versat, a new reconfigurable hardware accelerator which is suitable for low-cost low-power devices.

Versat can effectively replace a number of dedicated hardware accelerators in the system, making it smaller, more power efficient and safer to design (the development risk of designing dedicated hardware accelerators is eliminated).

With the eminent demise of Moore's law and the advent of the Internet of Things (IoT), power consumption and device price (which is linked to silicon area) are becoming main concerns [1]. Reconfigurable hardware, which is known to be extremely efficient in terms of power consumption and silicon area utilization, has been used in mid-range to high-end applications. However, applications to low-power low-cost devices still need to be researched.

The most suitable type of reconfigurable hardware for small programmable devices is the Coarse Grain Reconfigurable Array (CGRA) [2].

The main problems that we have identified with existing CGRAs are the slow and limited reconfiguration control and the difficulty in programming. Therefore, we propose some architectural improvements to address these problems which follow three basic ideas.

The first idea is to make the CGRA a fully connected graph. Normally, graphs with constrained connectivity are employed in CGRAs to avoid decreasing the frequency of operation. However, low power devices do not need to operate at a high frequency, so using a fully connected graph is a possibility. In terms of silicon area, fewer compute nodes can be used if every node is connected to every other node but more routing resources are needed. In terms of parallelism, more instruction level parallelism and data level parallelism can be achieved due to more flexibility in building datapaths. Finally, programming is drastically simplified as there is no need to *place & route* compute resources as in FPGAs.

The second idea is to make the configuration register addressable to the finest level. The configuration is divided in spaces, where each functional unit gets a configuration space. The configuration spaces are further divided in configuration fields which are made individually addressable. Partial reconfiguration is useful to keep reconfiguration to a strict minimum and to exploit the similarity between successive configurations. Reducing reconfiguration time has a dramatic influence on improving the performance, which can compensate for the lower frequency of operation.

The third idea is to integrate a controller in the CGRA able to access and manage all its resources in a multi-threaded fashion. The controller is in charge of the main program flow of the accelerator, sequencing the configurations and using partial reconfiguration whenever possible. The controller can spawn data stream compute threads in the CGRA and data movement threads using a DMA. While these threads are running, the controller can prepare the next configurations.

The controller makes the CGRA independent and very easy to use in a modern heterogeneous computing environment. A clean procedural interface to a host becomes possible. With such an interface the host can have tasks executed in the CGRA by simply calling procedures and passing arguments. This is in the spirit of the Open Computing Language (OpenCL) initiative [3].

Versat enables digital signal processing at a low power budget, which can be attractive for biometrics, speech recognition, artificial vision, security, etc. The overall goal of the project is to create an Intellectual Property (IP) core and a library of useful procedures.

II. Background

CGRAs have gained increasing attention in the last 2 decades both in academia and industry [4], [5], [6], [7], [8]. CGRAs are programmable hardware mesh structures that operate on word-length variables and are efficient in terms of operations per unit of silicon area. CGRAs can be built with RISC processor arrays [9] or simpler components such as adders, subtractors, multipliers, shifters, etc [10], [8]. We have identified the latter CGRAs as the most suitable architecture for a vast range of low power devices. A good survey of CGRAs is given in [2].

The main contribution in [8] was the invention of an address generation scheme able to support groups of nested loops in a single machine configuration. The idea, aimed at reducing reconfiguration time, was inspired by the use of cascaded counters for address generation [11]. This represented a major improvement from other works that focus exclusively in supporting the inner loops of compute kernels [2]. However, as this paper shows, more can be done to reduce the reconfiguration overhead.

A critical aspect for achieving performance speedups is dynamic reconfiguration of the CGRA. Static reconfiguration where the array is configured once to run a complete kernel is far less flexible [12]. Some arrays are dynamically reconfigurable but they only iterate over a fixed sequence of configurations [7], [4], [5].

The question of heterogeneity versus homogeneity of the functional units inside a CGRA is an important one. Some CGRAs are homogeneous [13], whereas others support a diversity of FUs [14]. A careful analysis in [15] has favored heterogeneous CGRAs as the performance degradation when going from homogeneous to heterogeneous is greatly compensated by the silicon utilization rate and power efficiency of heterogeneous solutions. Thus, we adopt heterogeneous CGRAs in this project.

Another important problem is that of the interconnect topology [16]. Fully connected graph topologies have been avoided as they scale poorly in terms of area, wire delays and power consumption. In this work this technique is nevertheless used to exploit the fact that frequencies of operation are anyway low.

Compiler support for CGRAs is probably the most difficult aspect. Not only a compiler has to make use of standard compilation techniques, especially the well known modulo scheduling approach used in VLIW machines [17], but also CAD techniques are needed, such as those used in FPGA compilation [18]. One attempt to circumvent the compiler difficulties is to formulate CGRAs as vector processors [19], [20], [21]. In those approaches, instructions are the equivalent of small configurations, and their authors claim several orders of magnitude speedup in certain applications. However, the user has to work at a very low level to make use of vector instructions. We propose as a solution to this problem the adoption of programming interfaces such as OpenCL [3], now very popular for GPUs and FPGAs in heterogeneous computing environments.

III. Architecture

The top level entity of the Versat module is shown in Fig. 1. Versat is designed to carry out computations on data arrays using its Data Engine (DE). To carry out computations the DE needs to be configured using the Configuration Subsystem. The data to be processed is read from an external memory using a DMA engine, also used to store the processed data back in the memory. A typical computation uses a number of DE configurations and a number of external memory transactions. The Controller executes programs stored in the Program Memory. Each program embodies the main flow of an algorithm, coordinating the reconfiguration and execution of the DE, as well as the external memory accesses. The controller accesses the modules in the system by means of the Read/Write (RW) bus.

The Versat top-level entity has a host interface and a memory interface. The host interface is used by a host system to load the Versat program containing a set of procedures or kernels and to command Versat to execute those kernels. The memory interface is used to access data from an external memory using the DMA. The host interface is used only to exchange command and status information. Data exchange is reserved to the memory interface. Exceptionally, if the amount of data is small, the host interface may be used for data exchange. Debug data is likely to use the host interface.

The host interface can have two formats selectable at compile time: a Serial Peripheral Interface (SPI) and a parallel bus interface. The SPI interface is used when an off-chip device

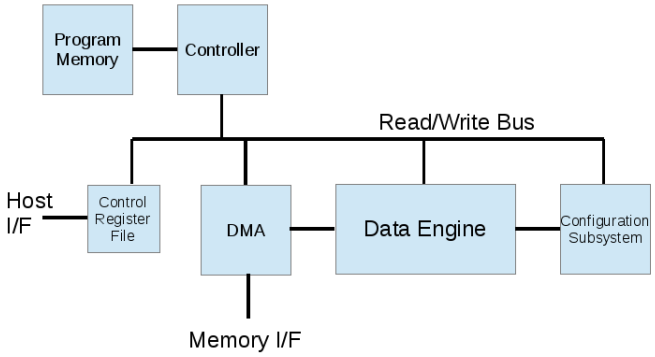

Fig. 1. Versat top-level entity

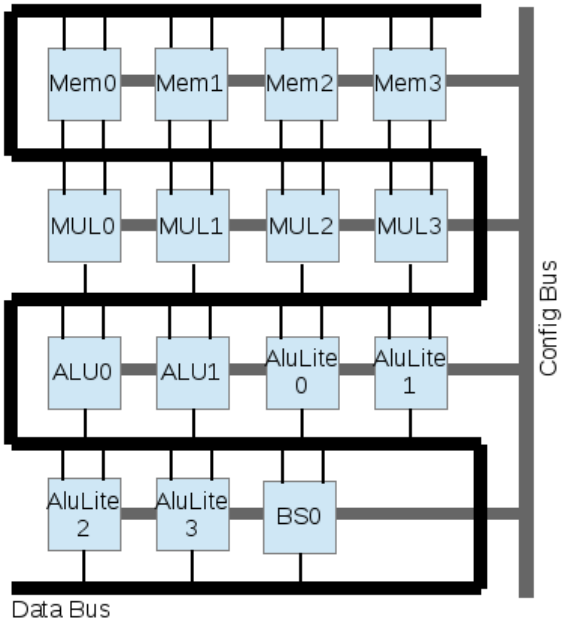

Fig. 2. Data engine

is the host. Versat is a slave SPI device and the host is a master SPI device. The parallel bus interface is used when the host is some embedded processor. This bus has a generic format which may need to be replaced with a proprietary interface. For example, an Altera Avalon interface may be used if a NIOS processor is the host; an AMBA AXI interface may be used in case the host is an ARM processor.

A. Data engine

The Data Engine (DE) currently has a fixed topology using 15 functional units (FUs) as shown in Fig. 2. However, it is relatively easy to change its structure to better accommodate the application. It is a 32-bit architecture and contains the following types of FUs: embedded memories (MEM), multipliers (MUL), arithmetic and logic units (ALU), including lightweight versions (AluLite). The Versat controller can read and write the outputs of any FU and can read and write to the embedded memories.

Each FU contributes its 32-bit output into a wide Data Bus. Additionally the data bus has two fixed entries for driving the constants 0 and 1, which are needed in many datapaths. Each

FU is able to select one data bus entry for each of its inputs. This creates a fully connected graph pattern to simplify the programming of Versat.

The FUs are configured by the Config Bus. Each FU is configured with a mode of operation and with its input selections. There are no global configurations – configuration data is always linked to a particular FU. The Config Bus is divided in configuration spaces, one for each FU. Each configuration space contains several configuration fields. Configuring a set of FUs results in a custom datapath for a particular computation.

Datapaths can have parallel execution lanes to exploit Data Level Parallelism (DLP) or pipelined paths to exploit Instruction Level Parallelism (ILP). An important type of FU is the dual-port embedded memory. Each port is equipped with an address generator to enable data-flow computing. If two memories are part of the same datapath, they will have its address generators working in a synchronized fashion. Given enough resources, multiple datapaths can operate in parallel with independent address generation. This corresponds to having multiple concurrent threads in Versat – Thread Level Parallelism (TLP).

The fact that each memory port has a dedicated address generator is in contrast with the address generation scheme in [8], where each address generator is built by connecting basic address generation units. The motivation behind the approach in [8] is to support nested loops of arbitrary depth in a single configuration. However, it makes the task of creating datapaths more complex, as a significant number of basic address units and connections between them need to be configured.

The discussion of the details of address generation in Versat falls out of the scope of this paper. We will simply state the following properties of the address generators: (1) the address generators are compact enough to be dedicated to each memory port; (2) only two levels of nested loops with a short inner loop is supported, as reconfiguring after each short loop causes excessive reconfiguration overhead; (3) if the inner loop is long then Versat will work in single loop mode and rely on its fast reconfiguration time to implement the outer loops.

B. Configuration subsystem

The Configuration Subsystem is illustrated in Fig. 3. It contains a fully addressable configuration register (`config_reg`) and a configuration memory (`config_mem`). The controller accesses the Configuration Subsystem from the RW bus using the following signals: write data (`rw_wdata`), address (`rw_addr`), read-not-write (`rw_rnw`) and request signal (`rw_req`). The Configuration Subsystem produces the wide Config. Bus, used to configure the DE.

The next configuration of the DE is stored in `config_reg` where one can write each field of the configuration space of an FU using the RW bus. There is a shadow configuration register (not shown) that holds the current DE configuration, so that `config_reg` can be changed while the DE is processing.

Building a configuration of the DE for the first time requires several writes to the fields of the configuration space of the involved FUs. In most applications there is a high likelihood that one configuration will be reused again in a nearby

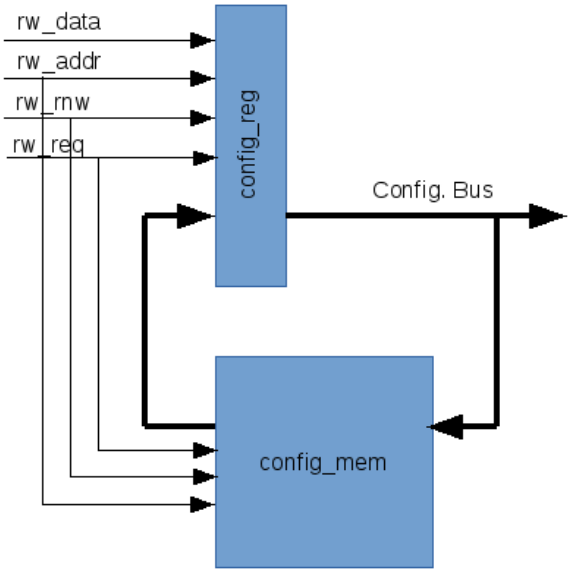

Fig. 3. Configuration subsystem

instant of time (time locality). It is also likely that other configurations will differ little from the current configuration. Thus, it is useful to save certain configurations in `config_mem` to later reuse them. A space for 64 configurations is provisioned for this buffer, organized in several 32-bit wide embedded RAM slices. The whole configuration word is currently 660 bits wide.

Each of the 64 positions of `config_mem` is addressable by the controller. A read access by the controller will read the configuration from `config_mem` into `config_reg`. A write access by the controller will write the contents of `config_reg` into `config_mem`.

C. DMA

One of the crucial factors to guarantee acceleration is the rapidity at which data is moved in and out of Versat. Addressing data words one by one in the external memory is out of the question. Data must be moved in blocks to amortize the memory latency that is always present when an external memory device has to be used.

The DMA engine is operated by the Versat controller to transfer a data burst from external memory to one of the Versat's data engine memories, or from one of Versat's data engine memories to the external memory.

From the Versat controller point of view the DMA is memory mapped and the following DMA registers can be accessed: the command register, the external address register and the status register. The command register contains the following fields: the internal memory address (14 bits), the direction of the transaction (1 bit stating if the transaction is from the external memory to the internal memory or otherwise) and the size of the transaction in 32-bit words (8 bits to support transactions of up to 256 words). The status register tells the controller whether the DMA is busy or ready to initiate a new transaction.

D. Program memory

The instruction memory is designed to contain 2048 instructions. This is deemed enough for most compute tasks (kernels) that we have in mind. In fact, the idea is to place multiple compute kernels in this memory. However, historically, memory needs always increase beyond expected. If we develop compiler tools, it is likely that the memory capacity to contain these compiled programs will have to be larger. Note that compiler produced code is never as optimized as the code that results from handwritten assembly instructions.

The program memory is divided in two parts: boot ROM and execution RAM. The boot ROM contains a hardwired program which runs after hardware reset. The execution RAM is where the user program to be run is loaded. The Versat controller can write to the program memory to load it and then can read each instruction to execute it; it cannot read words from the program memory as data.

The code in the boot ROM is called the boot loader. It is used for loading programs and data files into Versat; it is also used to download data files from Versat and to start programs previously loaded in the execution RAM. If the size of the programs increase it is likely that the execution RAM will evolve into an instruction cache. The instruction cache is likely to share the external memory with the DMA.

E. Control register file

Host and Versat exchange information and synchronize using the Control Register File (CRF). Both can read and write to the CRF which has 16 general purpose registers.

After hardware reset the boot-loader program waits a host command. Three registers of the CRF are used for this purpose: one as a command register, another as an address register and yet another as a data register.

The command and address registers are written by the host to indicate whether a read, write or execute operation is to take place and the respective internal address. The data register can be written by both host and Versat. If the host wants to write a word to Versat it places this word in the data register and Versat will transfer it to its address. If the host wants to read a word from Versat it commands Versat to transfer this word from its address into the data register and then proceeds to read it.

For the execution of regular Versat programs the CRF is used by the host to pass parameters to Versat and by Versat to return status information to the host.

F. Controller

The Versat controller has a minimal architecture to support reconfiguration, data movement and host interaction. Its architecture is shown in Fig. 4.

The controller architecture contains 3 main registers: the program counter (register PC), the accumulator (register A) and the data pointer (register B). The instruction whose address is pointed by the PC is decoded so that opcodes, immediate values and addresses are extracted from it.

Register A is the main system register and the destination of operations having as arguments register A itself, implicitly,

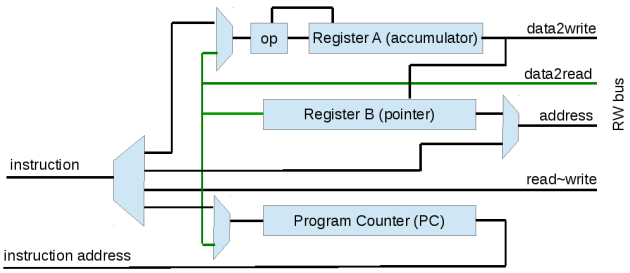

Fig. 4. Controller

and an immediate or addressed value. Register B is used to implement indirect loads and stores; its contents is the load/store address. Register B is addressable by the controller for reading and writing. Many other addresses are accessible for reading and writing by the controller by means of the RW bus. Fig. 4 shows the RW bus signals whose names are self-descriptive: `data2write`, `data2read`, `address` and `read~write`.

The controller handles host procedure calls. For each procedure it is necessary to read parameters from the control register file, load data vectors into Versat's memories using the DMA, configure and execute the DE a few times, and finally store data vectors in the external memory using the DMA. DE and DMA execution threads can be spawned, hiding part of the controller execution time.

The instruction set contains just 16 instructions used to perform the following actions: (1) loads/stores to/from the accumulator; (2) operations; (3) branching. The loads can be of immediate constants, of a direct address or of an indirect address stored in register B. The stores can be direct or indirect. Operations such as *add, subtract, shift, and* involve register A itself and an immediate or addressed value. One can branch if the accumulator is zero or non-zero, to an immediate address or to an indirect address.

Computations can be accomplished in two ways: (1) transport triggered and (2) data flow. For transport triggered computations the FUs are configured to form a datapath, values are written to the outputs of the source FUs and results are collected at the outputs of the destination FUs. For data flow computations a datapath is similarly set up but the embedded memories stream the data through the FUs and store the result data.

IV. Programming

Versat is programmed using its own assembly language. So far there is no compiler from a high-level language to this assembly language. The Versat assembler is implemented by a Python script and converts assembly code into the 16 machine code instructions supported by the Versat controller.

Programs written in the Versat assembly can control basic program flow, data movement using the DMA and DE configuration and operation. Using the assembly language, compute threads using the DE and data movement threads using the DMA can be spawned.

The assembly language incorporates the naming of instructions and the naming of the different parts of Versat that are memory mapped. For example, the instruction

```
wrw DMA_CTRL_REG
```

will write the contents of register A into the control register of the DMA, instructing it to do something. The contents of register A can be set by load instructions and manipulated by operations. For example,

```
ldi 0xFF00
add ALU0
```

will load the constant 0xFF00 to register A and add the output of ALU0 to it.

To set the DE to run one issues the instruction `wrw ENG_CTRL_REG`
To check whether the DE is still running one must read its status register:

```
rdw ENG_STATUS_REG
```

Another example is an instruction that writes to the *delay*[1] configuration field of the configuration space of one of the embedded memory ports (MEM1A):

```
wrw MEM1A_CONFIG_ADDR,MEM_CONF_DELAY_OFFSET
```

Branch instructions implement loop control and flow control in general, with special emphasis on conditional DE configurations. Since the current DE implementation is not capable of executing conditionals, the program decides if the DE is configured in one way or another. An example of a branch statement is the following

```
beqi TARGET
```

where `beqi` is a branch if the accumulator is zero, to the position labeled by the user as TARGET.

The assembler is implemented in two passes. The first pass reads the code labels inserted in the assembly code by the programmer and adds them to a lookup table. The second pass converts the instruction mnemonics into machine code.

V. Preliminary Results

This section presents results on implementing Versat on FPGA and performance results comparing Versat to FPGA embedded processors.

A. FPGA results

FPGA implementation results are given in Table I. These results show that in terms of size Versat is several times bigger than a "typical" (common configuration) embedded processor such as Altera's NIOS or Xilinx's Microblaze. Depending on the configuration of the embedded processor Versat can be 3-7 times bigger in size. Also in terms of frequency of operation Versat will use a 2-3 times lower clock frequency. However, as the results in the next section show, Versat can be considerably faster than these embedded processors.

TABLE I. FPGA implementation results

Architecture	Logic	Regs	RAM(kbit)	Mults	Fmax(MHz)
Cyclone IV	19366 LEs	4673	351	32 (9 bits)	64
Virtex V	12510 LUTs	4396	1062	16 (18 bits)	102

[1]The *delay* configuration field specifies the number of clock cycles after the initial instant to start generating the address sequence.

B. Execution results

This section presents execution results of running a set of example kernels on Versat and on the embedded processors Microblaze and Nios II. Hardware timers have been used to measure the time in elapsed clock cycles and the results are summarized in Table II.

Clock cycle counts have been obtained after code and data are already loaded in local memories. Since the DMA is still under development it was impossible to do otherwise. Nevertheless this is a reasonable comparison as the processors tend to operate multiple times on cached data and Versat tends to operate multiple times on data previously loaded into its embedded memories.

The results for Versat are obtained using a regular PC as the host, connected to an FPGA via a FTDI FT4232H USB module. The results for the FPGA embedded processors have been obtained on a Xilinx ML505 board and an Altera DE0 Nano board.

The purpose of these preliminary is to show that Versat is capable of acceleration but the examples we have tested so far do not yet illustrate the partial self-reconfiguration capabilities of Versat. The four examples are basic single configuration kernels. Execution times of C implementations running on the Microblaze and NIOS processors have been obtained and are very similar. Thus, in column *Processor* in Table II, the average cycle counts for these two processors is given. The total cycle counts for Versat are given in the homonymous column, and column *Configuration* gives just the configuration cycles. Two speedup results are presented: Speedup1 is the measured speedup and Speedup2 is the speedup discounting the configuration time. The purpose for the latter is to have an idea of the additional acceleration that can be obtained in more complex kernels that use multiple configurations, assuming individual configuration times can be hidden.

TABLE II. Execution results

Kernel	Processor	Versat	Configuration	Speedup1	Speedup2
`vec_add`	10001	304	45	32.90	38.61
`lpf1`	72273	2627	61	27.51	28.17
`lpf2`	104750	4194	89	24.98	25.52
`cdp`	52241	1144	112	45.67	60.62

The kernel `vec_add` is a vector addition, `lpf1` and `lpf2` are 1^{st} and 2^{nd} order IIR filters and `cdp` is a complex dot product. Kernel `vec_add` is very simple but it is fully pipelined and shows a considerable speedup in Versat; it produces one result vector element per cycle. Kernel `cdp` is more complex with 4 multiplies in parallel followed by pipelined adders. Due to a deeper pipeline the speedups for `cdp` are greater, despite the feedback loop needed to implement the accumulation in the end; a new result vector element is accumulated every other cycle. Kernels `lpf1` and `lpf2` show more modest speedups due to the feedback loops needed to implement the filters; they produce new vector elements every 5 and 8 cycles, respectively, in the Q1.31 fixed-point format.

VI. Conclusion

In this paper we have presented Versat, a new reconfigurable architecture that can take care of the reconfiguration

process itself as well as the data movement operations to and from an external memory. Versat is programmed in its own assembly language. It is designed to handle host procedure calls using a clean interface. For example, the complex dot product example characterized in the results section could be invoked by the host processor to run on Versat with the prototype function:

```
void cdp(int *c,int *a, int *b, n);
```

where c is a pointer to the complex result, a and b are the operand vector pointers and n is the size of a and b.

The results show that in general Versat can accomplish speedups in the order of a few tens, depending on how pipelined the datapaths are. Even for non-pipelined datapaths, speedups of a couple of tens are possible. In terms of area and operation frequency, Versat is 3-7 times larger than a commonly configured embedded processor and uses a 2-3 times lower clock rate. Reconfiguration times are small and can be hidden when more than one configuration is run in a kernel.

One question to be answered is whether Versat should always be programmed in assembly or a higher level language will be required. On the one hand Versat is designed to be a general purpose hardware accelerator and a simple assembly language should be enough to describe accelerators. The fact that it is programmable is already a big distinction compared to hardwired accelerators. On the other hand, Versat is flexible enough for building rich computing structures, and in that case using assembly language can limit the productivity of programmers.

ACKNOWLEDGMENT

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

REFERENCES

[1] Claudia Canali, Michele Colajanni, and Riccardo Lancellotti. Performance evolution of mobile web-based services. *Internet Computing, IEEE*, 13(2):60–68, 2009.

[2] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010.

[3] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[4] Bingfeng Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *Design & Test of Computers, IEEE*, 22(2):90–101, March 2005.

[5] Ming hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, and Fadi J. Kurdahi. Design and implementation of the MorphoSys reconfigurable computing processor. In *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*. Kluwer Academic Publishers, 2000.

[6] V. Baumgarte, G. Ehlers, F. May, A. Nckel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.

[7] M. Quax, J. Huisken, and J. Van Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004*, volume 3, pages 230–235 Vol.3, Feb 2004.

[8] J.T. de Sousa, V.M.G. Martins, N.C.C. Lourenco, A.M.D. Santos, and N.G. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, September 25 2012. US Patent 8,276,120.

[9] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sep 1997.

[10] Justin L Tripp, Jan Frigo, and Paul Graham. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. *Proc. of HPEC*, 2007.

[11] Salvatore M. Carta, Danilo Pani, and Luigi Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006.

[12] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.

[13] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 126–135, London, UK, 1996. Springer-Verlag.

[14] P.M. Heysters and G.J.M. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 6–, April 2003.

[15] Yongjun Park, J.J.K. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *International Conference on Field-Programmable Technology (FPT), 2012*, pages 335–342, Dec 2012.

[16] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM.

[17] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[18] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[19] A. Severance and G.G.F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.

[20] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 117–126, New York, NY, USA, 2014. ACM.

[21] M. Naylor and S.W. Moore. Rapid codesign of a soft vector processor and its compiler. In *24th International Conference on Field Programmable Logic and Applications (FPL), 2014*, pages 1–4, Sept 2014.