CHAPTER 10

PROGRAMMING DATA PARALLEL FPGA APPLICATIONS USING THE SIMD/VECTOR MODEL

Maya B. Gokhale
Lawrence Livermore National Laboratory

In the Single Instruction Multiple Data (SIMD) model, aggregate operations on arrays and vectors can be mapped to arrays of function units. A single instruction stream is dispatched from a control unit to the function units, which operate in lockstep on the data sequences. Reconfigurable hardware is well suited to perform SIMD (also called *vector* or *data parallel*) computation (see Section 5.1.5). Groups of lookup tables (LUTs) can be configured as function units, and the data local to each unit can be stored in distributed memories. This chapter explores parallel processing on reconfigurable computers using the SIMD/vector model.

Reconfigurable computers can exploit parallelism at many different levels of granularity, from coarse-grained parallel tasks to fine-grained instruction-level parallelism. The massive amount of parallelism available in the reconfigurable computer more than compensates for its slow clock rate—one-tenth the clock rate of modern microprocessors. Raw spatial parallelism is plentiful in reconfigurable processors, especially those based on FPGAs. The challenge is to partition and map the application onto the inherently parallel fabric of lookup tables, DSP blocks, and memories. Parallel activities can be explicitly described and scheduled by the programmer or hardware designer, or can be inferred through analysis of the source code. SIMD/vector parallelism is very well suited to the spatial parallelism of FPGAs and other coarse-grained arithmetic logic units (ALU) arrays. In this programming model, aggregate data such as vectors and matrices are processed in parallel on arrays of function units.

10.1 SIMD COMPUTING ON FPGAS: AN EXAMPLE

As an introduction to SIMD computing on FPGAs, Figure 10.1 shows an SIMD array customized to perform two vector operations. A vector A is scaled by a constant factor, and then the dot product $A \cdot B = \Sigma a_i \times b_i$ of vectors A and B is performed. In this example, the number of SIMD processors is equal to the size of the vectors. Each processor holds one element of A and one of B. There is an additional storage location in each processor to hold the result of the $\times$ operation.

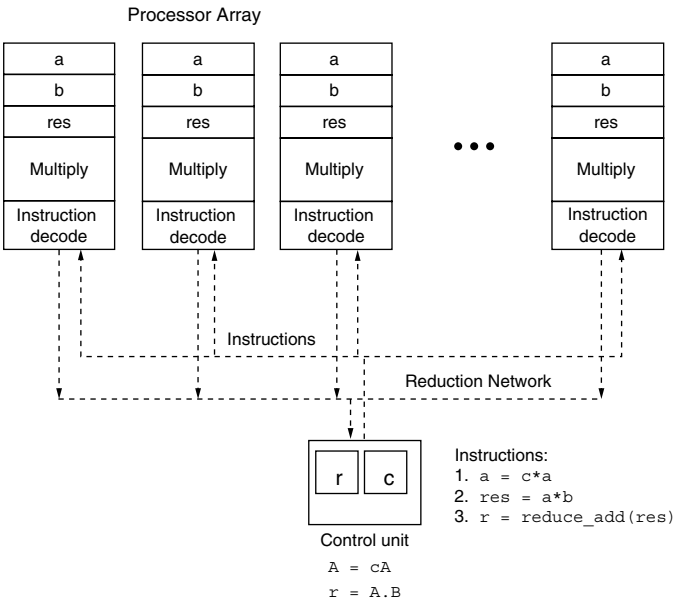

FIGURE 10.1 ■ An SIMD dot-product machine.

The control unit generates the instruction stream. An instruction can be executed on the control unit itself, on each processor of the processor array, or cooperatively on both. In this example, the control unit sends the constant *c* as part of the first multiply instruction. The constant appears as an immediate operand in the instruction to each processor. Next the control unit sends the second multiply instruction to the processor array, and all processors perform the operation `res=a*b`. The final instruction performs a *reduction*, a global combining operation, in which each processor sends its instance of `res` into the reduction network. Because the operation is a global sum, all the `res` instances are summed and the result is stored in the control unit variable *r*. While the example shows the control unit sending three separate instructions to the processor array, on an FPGA it is very possible that the controller will send a single instruction that results in a multi-cycle sequence of multiply operations followed by the global sum.

In this idealized example, the number of processors exactly matches the size of the vectors. In real applications, there are many different vectors of different sizes. The vectors must be distributed to the processors in blocks, and each processor must multiply subvectors of elements. If the number of processors doesn't evenly divide the vector size, some processors must remain inactive when the tail ends of the vectors are multiplied. Each processor must keep a subaccumulation, and, when the entire vector has been processed, the global sum is performed over the partial sums. When the processor array is on an FPGA, the compiler must synthesize state machines (FSMD subsection of Section 5.2.2) to

control the sequence of operations and iterate over the blocks of data. Designing algorithms for reconfigurable computers in the SIMD model in the face of these real-world complicating factors will be addressed in Section 10.4.

10.2 SIMD PROCESSING ARCHITECTURES

SIMD/vector machines were among the first parallel processors to be designed. From the days of the Iliac IV, with 64 processing elements (PEs) receiving instructions from a control processor, this parallel-processing architecture has gone through myriad incarnations. Notable among SIMD arrays are the Connection Machine, which had thousands of simple PEs operating in synchrony [1], as well as DAP and MasPar (late 1980s [2]). The Terasys Integrated Circuit [3] and the Clearspeed SIMD array [4] both included an SIMD processing array on a single integrated circuit.

Historically, supercomputers with dedicated floating-point function units used for processing arrays and vectors were called vector supercomputers, while massively parallel, highly interconnected arrays of function units were referred to as SIMD, or data parallel. More recently, as small arrays of function units have been incorporated into the architecture of scalar processors, the terms *SIMD*, *vector*, and *data parallel* have become interchangeable. This is especially apropos to reconfigurable computers, in which arbitrary numbers and types of function units may be used with many different kinds of interconnect patterns.

An SIMD processing array, illustrated in Figure 10.2, consists of a collection of identical processing elements operating in lockstep. The PEs all execute exactly the same instruction, which is broadcast to them from a control unit, or "sequencer," as indicated by the dotted lines in the figure. Each PE has a local memory from which to fetch data operands and store results. On an SIMD array, control flow instructions, such as branching, conditional branch, and subroutine call, are executed on the control unit.

Data-dependent branching represents a particular challenge when different instances of the data are resident in each PE's memory. Depending on the data value, some PEs might evaluate the branch predicate to true and others to false. Because they all must execute the same instruction at the same time, each PE has a predicate mask flag (the `M` in the corner of each ALU) indicating whether the PE should execute or ignore the current instruction.

The PE sets the predicate mask to the result of evaluating the predicate on its data items, and then either executes subsequent instructions or is inactive. The control unit can reset PEs to the active mode by issuing "unconditional" instructions to them, directing them to ignore the predicate mask. The notion of predicated instructions, which is essential to SIMD processing, is also used in some microprocessor instruction sets [5], particularly in wide-word explicitly parallel architectures.

In SIMD processing, PEs exchange data synchronously. The PE interconnection network may be arranged as a linear array, as in Figure 10.2, or as a two-dimensional (or even three-dimensional) mesh or torus. In addition to

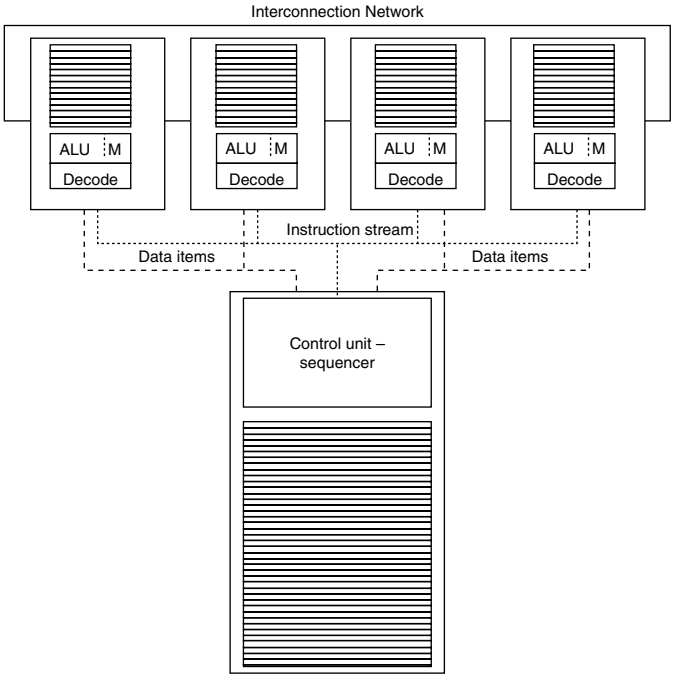

FIGURE 10.2 ■ An SIMD processing array.

nearest-neighbor communication (illustrated with solid lines in the figure), data parallel arrays usually include global combining networks for global reduction (sum, product, min, max, and logical) operations. The control unit can retrieve data from the memory of individual PEs and can also receive the result of the global combining operations (dashed lines in the figure).

A global combining network is illustrated in Figure 10.3, which shows a network organized as a binary tree with a combining operator at each interior tree node. Global combining networks can be used for any associative operation. With parallel tree operations, an `O(n)` operation is reduced to `O(log(n))`.

10.3 DATA PARALLEL LANGUAGES

High-level data parallel languages for SIMD machines were popularized in the late 1980s with the emergence of the Connection Machines CM-1, CM-2, and CM-5, and were adopted by other vendors. In the CM approach, a base language such as Fortran or C was extended with new keywords, syntax, and semantics. In the C* language, a data parallel extension to ANSI C, new data type modifiers `mono` and `poly` were introduced. A `mono` variable resides in the control unit memory, while a `poly` variable occupies memory local to each PE,

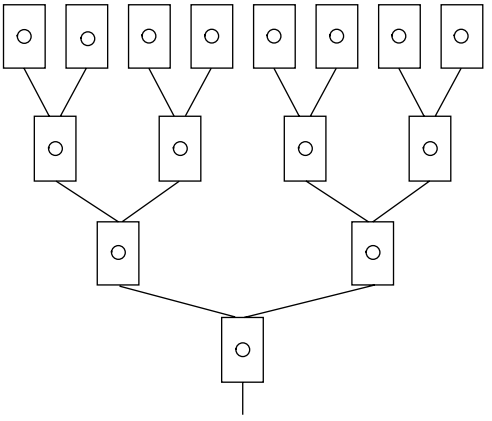

FIGURE 10.3 ■ A global sum network.

implicitly defining a vector or higher-dimension array. Operation on a `mono` variable is performed on the control unit, while a `poly` expression is evaluated independently on each PE.

Also in the 1980s, new syntax and intrinsic functions were introduced to express global combining operations, inter-PE communication, and unconditional execution.

Declaration of `poly` variables in most data parallel languages implicitly defines an aggregate object whose length is the number of PEs in the physical array. Unfortunately, most datasets do not conform in size or shape to the physical PE array, and therefore the programmer must arrange the data arrays in blocks distributed among the PEs' memories, and then loop over the blocks on each PE. The Connection Machines, however, supported "virtual" processors in microcode. The programmer could define an array of processing elements larger than the size of the physical PE array that better matched the size of the datasets, and microcode in each PE looped over the block of data in its memory.

10.4 RECONFIGURABLE COMPUTERS FOR SIMD/VECTOR PROCESSING

In contrast to specific physical implementations of SIMD arrays in silicon, a large variety of data parallel machines may be mapped onto FPGA-based reconfigurable computers. The data parallel model maps naturally to the physical structure of FPGAs, with dedicated hardware blocks of arithmetic units and memories tiled regularly in a two-dimensional array, as well as a flexible interconnect. In addition, there are many degrees of freedom in an FPGA implementation. The data parallel engine can be customized to the datasets being processed in terms of geometry (one versus multidimensional arrays), interconnect (linear, mesh, torus), and even PE instruction set.

An early experiment in data parallel computing on FPGAs was the dbC project [6] in which a data parallel language was compiled onto the Splash 2 reconfigurable logic array [7]. dbC was modeled on the Connection Machines' C* language. Like C*, dbC included the `mono` and `poly` data type modifiers to denote data on the control unit and SIMD array, respectively.

The size of the SIMD array could be specified at the language level by setting a predefined variable to the number of PEs. The linear array thus defined was automatically partitioned among the 16 FPGAs of the Splash system.

Instructions were broadcast to the FPGAs from the Sun workstation host, which served as the control unit. Unlike conventional SIMD arrays, the PE instruction set was not fixed. Rather, the compiler created a unique instruction set for each dbC program, generating a behavioral VHDL module (see Chapter 6) that was synthesized through the normal CAD tool flow. An instruction, rather than being a simple arithmetic or load/store operation, was synthesized as a predicated block. This could be a simple basic block—a straight-line sequence of code with a single entry and a single exit. If the C code contained `if` statements, the compiler transformed control dependence into data dependence [8], creating sequential predicated blocks that contained first the true branch and then the false branch of the `if`. Thus, a single instruction dispatched from the control unit to the SIMD array could result in a multi-clock-cycle block of logic executing a predicated hyperblock.

To exploit the flexibility of FPGAs to perform arithmetic on arbitrary bit-length operands, dbC allowed `poly` variables to be of user-specified bit length. dbC extended C integer data types by permitting C bit field syntax to be used to define the bit length of signed and unsigned integer variables. This ability was particularly valuable on early FPGAs with limited logic and interconnect. The arithmetic units synthesized within the SIMD PE were customized to the precision required, and the programmer specified that precision by the choice of data types.

In keeping with the SIMD interprocessor communication model, a runtime hardware library was built to implement global communications instructions such as min/max and a small set of logic operations, which were performed bit-serially by the Splash 2 control FPGA.

The dbC language and compiler thus combined a parallel language, traditional compiler transformations, and a simple form of hardware synthesis to generate a control program and FPGA bitstream for the Splash system.

To illustrate the dbC data parallel language and its mapping onto FPGAs, Figure 10.4 expands on the vector multiply example in Section 10.2. Line 3 illustrates the use of bit field syntax to define a new data type, a 24-bit integer, `my_int`. `DBC_net_shape` (line 6) is a predefined variable used to set the number of processors and their shape. (On Splash, the shape was limited to a linear array.) The vector multiply is divided into two sections. First there is a loop over the blocks of vectors resident on each PE (lines 31–34). The control unit handles the loop control and iteratively issues instructions in the loop body to the SIMD array. The += operation on line 33 is executed by each PE and accumulates the partial product into the `poly` variable `res`.

```
1   #define ISIZE 24
2
3   typedef poly int my_int:ISIZE;
4
5   /* specify 64 processors in a linear array */
6   unsigned in DBC_net_shape[1] = {64};
7
8   /* Each PE can hold up to 500 elements of the vector,
9      so maximum vector size is 500*64                  */
10
11  #define VEC_MAX 500
12  void main() {
13
14    /* vectors A, B, res are on each PE */
15    poly my_int A[VEC_MAX];
16    poly my_int B[VEC_MAX];
17    poly my_int res[VEC_MAX];
18
19    /* r, c, and vec_size are on the control unit */
20    mono unsigned long long int r;
21    mono int c;
22    mono int vec_size;
23    int i;
24
25    /* first initialize vec_size, vectors A and B, constant c */
26
27    /* next, compute vector multiply on the vector elements up to
28       the index that evenly divide the total number of PEs.    */
29
30    res = 0;
31    for (i=0; i<vec_size/DBC_nproc; i++) {
32      A[i] = A[i] * c;
33      res += A[i] * B[i];
34    }
35
36    /* now multiply the remaining elements of the vectors */
37
38    if (DBC_iproc < vec_size % DBC_nproc) {
39      A[i] = A[i] * c;
40      res += A[i]*B[i];
41    }
42
43    r += res;
44
45    /* continue computation */
46
47  }
```

FIGURE 10.4 ■ A vector multiply program in dbC.

The second section of code finishes the multiplication of final residue, potentially on a smaller number of PEs (lines 38–41). The `if` statement on line 38 sets the predicate mask bit to true in each PE whose processor number is less than the number of remaining elements of the vectors, and to false in all the other

PEs. The comparison of `vec_size` to `DBC_nproc` involves only `mono` variables and so is performed on the control unit and sent to the PE array as a constant in the instruction. Line 43 is a global accumulation of intermediate results from each PE into the control unit variable `r`.

There are some unique aspects to compiling SIMD algorithms to FPGA-based reconfigurable computers. For one, the compiler can synthesize an instruction set customized to the application. In our example, there need be only three instructions:

- `A[i] = A[i] * c; res += A[i] * b[i];`
- mask bit ← `DBC iproc < vec size % DBC nproc`
- `r += res;`

For another, the ALU can be customized to the operations used in the code. In this example, only a 24-bit multiplier, adder, and comparator are required. If different precision is needed, the PE can be resynthesized. In fact, if floating-point data types are necessary, floating-point, rather than integer arithmetic units can be instantiated. Finally, the PE array can be easily resynthesized to hold more or fewer PEs.

10.5 VARIATIONS OF SIMD/VECTOR COMPUTING

The SIMD programming model is attractive in its simplicity of parallel operation. There is a single instruction stream; inter-PE communication is global and synchronous; and the global reduction operations allow operations across the entire PE array. However, SIMD also has some deficiencies. Often there are cases in which some PEs perform slightly different operations than others, particularly with boundary conditions. The SIMD model requires that all PEs participate in all alternatives. This can result in poor performance in the presence of deeply nested `if` statements, as the instruction stream follows all possible control flows. For this reason, SIMD processing is often used in conjunction with other programming models on reconfigurable computers.

10.5.1 Multiple SIMD Engines

It is possible to map multiple SIMD engines onto an FPGA, with a controller for each engine synthesized in the reconfigurable logic. Such a system is illustrated in Figure 10.5. This capability was offered by the Fabric-based System [9], and demonstrated on a system-on-a-chip using the Altera Excalibur FPGA. In this framework, the on-chip microprocessor controls a flexible, runtime reconfigurable computing fabric of mesh-connected processing cells. Each cell has a separate local data memory and a small program memory that holds DSP-like microcode instructions. In SIMD mode, a group of cells all contain the same program and are sequenced through it by a customized control unit that is also in the reconfigurable logic.

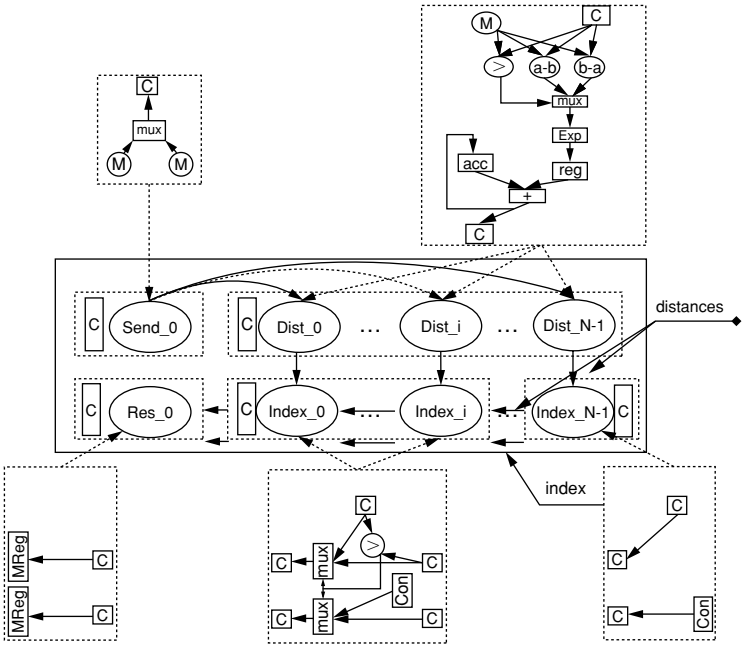

FIGURE 10.5 ■ An extended SIMD architecture.

The fabric illustrated in Figure 10.5 shows a multi-SIMD implementation of a compute-intensive kernel of the K-Means clustering algorithm. In this iterative data-mining algorithm, the dataset is partitioned into a predetermined number of classes. Initially, elements of the dataset are randomly assigned to classes, and a center C_i (where i ranges over the number of classes) of each class is computed. Then, for each element E_j (j ranges over the number of data elements) and each class C_i, the distance between E_j and C_i is computed. E_j is moved to the class closest to it in the distance metric, and the process repeats either for a fixed number of iterations or until there is no change from the previous iteration. In the example, the distance metric is $|E_j - C_i|$ (i.e., the absolute value [10] is used).

This design (Figure 10.5) implements the distance calculation, in which the distance between E_j and C_i is computed, and finds the class closest to each element E_j. There are five cell types—send, distance calculation, two for index calculation, and receive—each with its own control unit (labeled "C" in the figure). The "Dist" SIMD engine controls the distance calculation PEs; the "Index" SIMD engine, the index calculation PEs. The last index calculation has its own controller because its interconnect is slightly different from the others. Similarly, the send and receive cells have unique datapaths, so each has a dedicated controller.

In the figure, the computation is parallelized across classes, with one distance/index pair per class. A microprocessor controls the outer K-Means loop and updates class centers by loading new values into the `Send_0` cell's local memory. `Send_0` reads from one of two memories and sends the data element

out its communication channel. This allows the microprocessor to load one memory while the fabric is computing with the other. The distance calculation cells compute the distance between the pixel and the class centers. Their datapath is shown in the upper right box. The index calculation cells calculate the index of the class having the minimum distance to the pixel (the middle and right boxes at the bottom). The receive cell (`Res_0`) stores the class index corresponding to the minimum distance. It accepts data from two channels and writes into two memories.

Thus, an efficient parallel architecture for the K-Means clustering algorithm combines two SIMD arrays with three additional specialized processing units and a control microprocessor.

10.5.2 A Multi-SIMD Coarse-grained Array

In addition to FPGA-based data parallel systems, the Morphosys system [11] was designed as a coarse-grained SIMD array. Morphosys was an 8×8 array of reconfigurable logic cells controlled by a small RISC processor. Each row or column of the array operated in SIMD mode, executing the same instruction on different data instances. The RISC processor could dynamically load configurations into the array on a row/column granularity. This versatility in data parallelism and dynamic reconfiguration made it possible to map a combination of data parallel and control parallel algorithms onto Morphosys.

10.5.3 SPMD Model

A popular generalization of SIMD is the Single Program Multiple Data (SPMD) model (see Single program multiple data subsection of Section 5.2.4 and [12]) in which all processes independently execute the same program and can take different paths through it. SPMD differs from SIMD in that, rather than execute a global, synchronized communication step, programs use send/receive message passing to communicate with each other, and may employ other synchronization primitives such as barrier synchronization, in which each process waits at the barrier until all processes have reached it in their control flow.

SPMD is most common in parallel processing clusters. However, elements of it have also been adapted to FPGA computing. For example, in the Streams-C language, a CSP-like [13] parallel programming language for FPGAs [14], the programmer can define a parallel processor composed of an "array of processes," with each having the same hardware logic and control program, operating independently from the others, and using unidirectional channels to communicate.

10.6 PIPELINED SIMD/VECTOR PROCESSING

Pipeline processing can often be incorporated into SIMD/vector reconfigurable computing. This technique in essence synthesizes customized vector units that are replicated on the FPGA. Pipelined SIMD processing is especially beneficial on

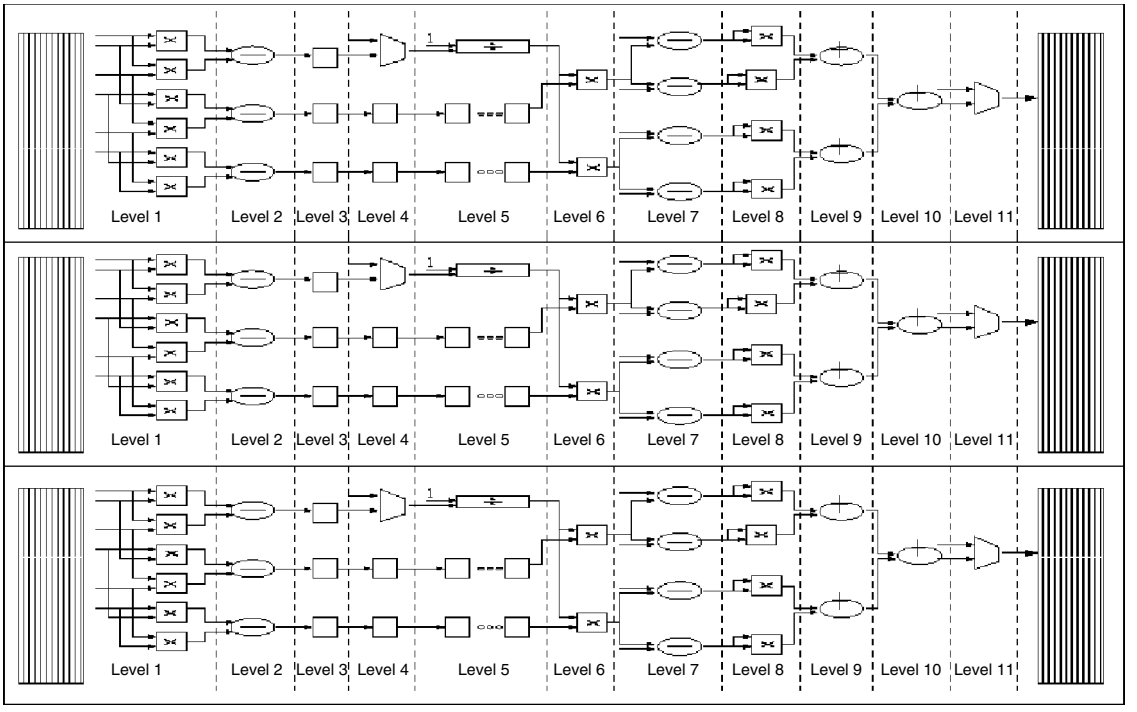

FIGURE 10.6 ■ SIMD with pipelined vector units.

FPGAs when complex arithmetic operations such as floating-point calculations must be performed.

Figure 10.6 shows an SIMD pipelined processing system in reconfigurable logic [15] having three PEs. Three pipelines are instantiated, with each receiving input parameters from a local memory and returning results to another memory. Each pipeline has 11 stages of floating-point operations, and each floating-point operation is, in turn, pipelined, resulting in a 43-stage pipeline. This pipeline implements the inner loop of a Monte Carlo simulation of radiative heat transfer in a two-dimensional chamber. In this case, three single-precision floating-point pipelines could be accommodated on a Xilinx Virtex-II Pro 100.

10.7 SUMMARY

In the SIMD/vector model, a tightly coupled ensemble of processors execute a single instruction stream issued by a control unit. The model can be synthesized onto an FPGA fabric. Having programmable hardware makes it possible to synthesize an instruction set tailored to the specific computations in the application. Customized data widths are naturally accommodated, as there is no fixed-width

ALU. Global combining operations utilizing parallel prefix networks can also be synthesized.

On FPGAs, the SIMD/vector model can be flexibly extended. Collections of SIMD subunits can be assembled and interconnected. This permits portions of the application that map naturally to the SIMD programming model to use it while still allowing other more irregular, control flow-dominated code to be synthesized on the same device. Pipeline processing can also be incorporated into the SIMD/vector processor, increasing the spatial parallelism available to the application.

Acknowledgments The contributions of Christophe Wolinski to Section 10.5 and of Jan Frigo to Section 10.6 are gratefully acknowledged.

References

[1] W. D. Hillis. *The Connection Machine*, MIT Press, 1989.
[2] R. M. Hord. *Parallel Supercomputing in SIMD Architectures*, CRC Press, 1990.
[3] M. Gokhale, B. Holmes, K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer* 23–31, 1995.
[4] Clearspeed. *http://www.clearspeed.com/*.
[5] D. I. August, et al. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *International Symposium on Computer Architecture*, 1998.
[6] M. Gokhale, B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing* 9(3), 1995.
[7] D. A. Buell, J. M. Arnold, W. J. Kleinfelder (eds.). *Splash 2: FPGAs in a Custom Computing Machine,* Wiley-IEEE Computer Society Press, 1996.
[8] J. R. Allen, K. Kennedy, C. Porterfield, J. Warren. Conversion of control dependence to data dependence. *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983.
[9] C. Wolinski, M. Gokhale, K. McCabe. *Polymorphous Fabric-based Systems: Model, Tools, Applications*, Elsevier Science, 2003.
[10] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. Szymanski, J. Theiler. Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery. *Proceedings SPIE* 4480, 2001.
[11] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers* 49(5), 2000.
[12] T. G. Mattson, B. A. Sanders, B. Massingill. *Patterns for Parallel Programming*, Addison-Wesley, 2004.
[13] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, 1985.
[14] M. B. Gokhale, J. M. Stone, J. Arnold, M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *IEEE International Symposium on FPGAs for Custom Computing Machines*, April 2000.
[15] M. Gokhale, J. Frigo, C. Ahrens, J. L. Tripp, R. Minnich. Monte Carlo radiative heat transfer simulation on a reconfigurable computer: An evaluation. *Proceedings Field-Programmable Logic and Applications (FPL)*, 2004.