SIZING OF PROCESSING ARRAYS FOR FPGA-BASED COMPUTATION*

Tom VanCourt and Martin Herbordt

Boston University, Department of Electrical and Computer Engineering
8 St. Mary's St., Boston MA USA 02215
email: {tvancour, herbordt} @ bu.edu

ABSTRACT

Computing applications in FPGAs are commonly built from repetitive structures of computing and/or memory elements. In many cases, application performance depends on the degree of parallelism – ideally, the most that will fit into the fabric of the FPGA being used. Several factors complicate determination of the largest structure that will fit the FPGA: arrays that grow polynomially and trees that grow exponentially, coupled structures that grow in different polynomial order, multiple design parameters controlling different aspects of the computing structure, and interlocked usage of different hardware resources. Combined with resource usage that depends on application-specific data elements and arithmetic details, these factors defeat any simple approach for scaling the computing structures up to the FPGA's capacity. We present an analytic framework for maximizing FPGA utilization, including features for efficient exploration of array size alternatives. We also report on design tools containing extensions that support automated sizing of FPGA-based computation arrays.

1. INTRODUCTION

Recent product announcements by Cray [2] and Silicon Graphics, Inc. [12] show that FPGA acceleration is entering the main stream of performance computing. Tools for FPGA application development are still oriented toward traditional logic rather than computation, however, leading to the observation that "*10×-100× of performance ... has been at the cost of 10×-100× increase in difficulty in application development*." [6] Much of this difficulty comes from the need to manage an FPGA's biggest advantage, its inherently massive, fine grained parallelism. In many applications, however, well understood repetitive structures such as systolic arrays can be used to manage the parallelism [15][16]. These offer the possibility of effectively unbounded growth of computation arrays as FPGA sizes increase, with the promise of increased application throughput or of increased problem sizes.

One challenge to tools for FPGA-based computation lies in determining the size of the largest possible array for a given computation. Three factors affect the array size: resource limits set by the FPGA hardware, details of the specific application at hand, and the algebraic law governing permissible sizes for the application's computing array.

The structure of a computing array is defined by the particular application at hand. Some applications use linear arrays, but others use square or cubical arrays, trees, or multiple structures coupled to each other. Good design practice often requires partitioning the structure into multiple components, coupling between components visible only at the highest design levels. Because of non-linear growth laws for computing arrays and because of the need for global analysis in determining the array's total resource utilization, local approaches for linear scaling (e.g. loop unrolling) are not sufficient for determining how large an array can be instantiated.

Contributions in this paper address automated growth or sizing of FPGA-based computation arrays in the presence of nonlinear allocation of FPGA resources, multiple different growth-limiting resources in the FPGA (including logic, RAM, and hardware multipliers), and customization of the arrays for different applications. The remainder of this introduction presents the idea of application families, or reusable communication structures that can be instantiated with different data communicated data elements and with different inner components. Section 2 briefly reviews related work in design space exploration (DSE). It will be seen that the current problem has different goals than previous DSE research, but that many valuable ideas from DSE are applicable to automated sizing of computation arrays. Section 3 describes problem of optimizing the computation array in more detail, with examples of FGPA and application complications that must be addressed. This section ends with a formal statement of array sizing as a multidimensional optimization problem. Section 4 presents one solution to the array sizing problem, as implemented in the prototype LAMP tool set. This includes language primitives for heuristic resource estimation, and guidelines for implementing the optimization formalisms in terms of LAMP language features. We show how this addresses the reusable portion of the application accelerator, the portion unique to each instance of the accelerator, and the idiosyncrasies of each FPGA, with respect to allocation of its computing resources. Finally, section 5 describes factors

* This work was supported in part by the NIH through award RR020209-01, and facilitated by donations of software and equipment from Xilinx Corporation

not addressed in the current implementation, suggests ways in which the current LAMP strategy may be strengthened.

1.1. Application Families

The problem of scaling an array of processing elements (PEs) arises when creating computation accelerators for *families of applications*. A family of applications is defined by a reusable computing pipeline and communication structure, within which the details of application-specific data type and leaf computations are free to vary.

One example of an application family addresses approximate string matching [15]. In that case study, the character strings may be DNA with a four-letter (two bit) alphabet, protein sequences composed of 20-letters (five bits each), IUPAC wildcards (four bits), codons (six bits), or any other type the user chooses. Character comparison functions also vary according to application choice: exact equality tests, tests for synonymous codons, wildcard matches, or a function returning graded goodness-of-match scores. Another application family arises in generalized 3D correlations used for screening molecules as potential drug leads [16]. In this case, the voxel data values and the intermolecular scoring functions vary widely according to the kinds of chemical interactions being modeled. Both the number of bits in the data values and the numbers of gates needed for the scoring functions vary widely across the range of family members.

In cases like these, the basic computation is performed by an array of hundreds or thousands of PEs. Larger arrays support higher parallelism, so support higher throughput or larger problem sizes. One design goal, therefore, is to implement the biggest array possible for a given problem. Efficient implementation depends on the FPGA's capacity for fine-grained resource allocation. Each member of each application family allocates only as many gates as needed for its data elements and PEs. As a result, family members with lower resource allocation per PE can generate more PEs, allowing larger computations to be performed.

It is understood that we extend the conventional term 'PE' to refer to any of the repeatable elements of the computing structure. This definition includes PEs containing RAM buffers, and even PEs with RAM only and no arithmetic processing.

2. RELATED WORK

Standard FPGA design tools attempt to increase FPGA resource utilization through better placement and routing choices [3] or through new algorithms for choosing between available resources within an FPGA [8]. In these cases, the design tools perform low-level tradeoffs between resource instances or resource types in order to implement a fixed logic design. These optimizations sometimes replicate logic structures in order to meet timing requirements or to improve performance. This kind of replication, however, does not change the apparent degree of parallelism in the system. Array sizing, however, changes the number of PEs in visible ways, in order to increase the size or speed of the computation.

Design space exploration (DSE) is another family of approaches to increasing FPGA utilization. This generally considers an application of strictly defined function, and tries multiple implementations of the application subsystems. In some forms, DSE proposes alternative implementations of a fixed system definition with different space-time tradeoffs [3], or with maximum speed [5].

Another DSE approach seeks to reduce hardware costs through temporal partitioning of one algorithm into multiple logic patterns for one FPGA, subject to the constraint that the FPGA computation and reload times meet stated timing constraints [11].

DSE differs from the current study in one major respect: it holds the application to perform constant. Various DSE approaches then try to reduce hardware cost or improve other performance criteria, possibly within some performance constraint. In the current study, a single parameterized implementation is used, but the application is allowed to vary up to the capacity of the chosen FPGA. Although automated sizing relies on many of the same estimation techniques as DSE, its goals are very different.

3. INPUTS TO THE OPTIMIZATION PROBLEM

Whatever the computation array and FPGA capacity, the accelerator designer generally wants one thing that no current design tools are able to state explicitly: as many PEs as possible, in order to maximize parallelism in between the PEs. This indefinite number depends on the resource utilization per PE, permissible sizes for computation and memory arrays, and FPGA capacity.

Three major sources of information affect an application accelerator's implementation. First, the choice of FPGA specifies the available amounts of each computing resource. Second, resource utilization specific to a particular member of the application family specifies the amount of FPGA fabric needed for each PE. Third, the geometry of the computing array, shared across all members of the application family, specifies the numbers of PEs in each valid configuration. The remainder of this section discusses how those factors combine to define an application-specific accelerator.

3.1. FPGA resources

The FPGA resources are simply the programmable logic, hardware multipliers, block RAMs, and other features accessible to the logic designer. (Connectivity resources are usually allocated by development tools, and not directly available to the logic designer.) A larger FPGA in a given

product family contains more of some or all resources, potentially allowing a larger computation array for a given application's accelerator. The resources of interest are expected to differ between applications or application families; one family member may require hardware multipliers where another does not, for example.

Care must be taken in creating the resource abstraction since some resources are available only in specific quanta, such as block sizes for RAM bits. Extra care must be taken when the abstraction must cross FPGA product lines, since resources from different vendors are not always directly comparable. Block RAMS typify resource differences between vendors: the Xilinx Virtex-II Pro products contain 18Kb block RAMs, but comparable Altera Stratix-II chips offer a combination of 512b, 4Kb, and 512Kb RAMs.

3.2. Application-specific resource usage

Different members of an application family share structures for control, communication, and synchronization, but differ in the numbers of bits in their operand data types and in the complexity of the "leaf" calculations used within the pipeline. For example, the case study of [16] uses a generalized 3D correlation of the form:

$$S_{xyz} = \sum_{i,j,k} F(a_{x+i,y+j,z+k}, b_{ijk}) \qquad (1)$$

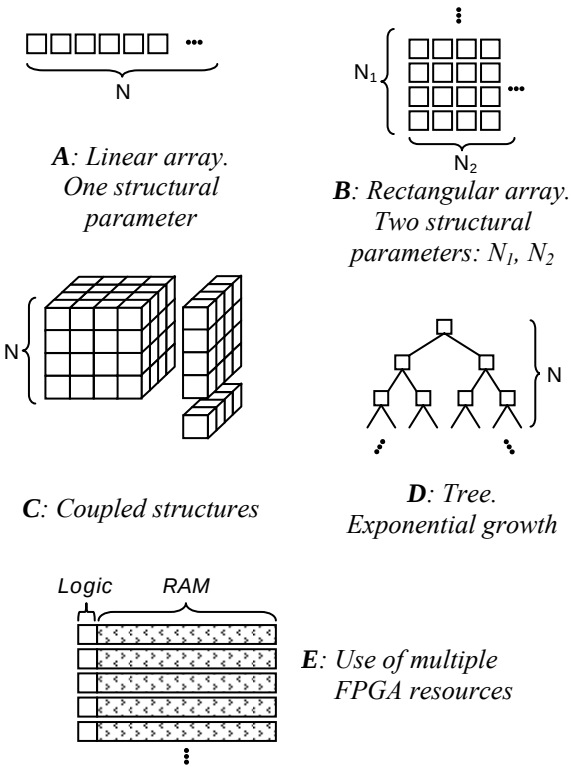
The data types of voxels *a* and *b* and the logic of function *F* differ in each member of the application family, in order to represent the chemical phenomena used in that instance of the calculation. As a result, the numbers of data bits and the size of each PE in the computing array differ between family members. This means that, for a given FPGA, different members of an application family can instantiate different numbers of PEs.

3.3. Growth laws in computing arrays

A *growth law* is the set of arithmetic rules defining the allowable array sizes. A crucial feature of the growth laws, explained in the next section, is that they invert the sense of the structural parameters commonly used for specifying numbers of component instances in some logic design. In this analysis, the set of parameter values is not supplied by the designer, but chosen by the tools to create the most useful accelerator possible given the resources available.

For generality and for ease of discussion, the term "processing element" and the same formalisms address arrays of memory elements, computing elements, and combinations. The goal of the discussion is to characterize repeatable arrays of computing resources, whatever the resource may be.

Figure 1 suggests growth laws for several kinds of computation array. Figure 1A, a linear array, is the simplest. It allows an array of size *N* for any positive



***A**: Linear array. One structural parameter*

***B**: Rectangular array. Two structural parameters: N_1, N_2*

***C**: Coupled structures*

***D**: Tree. Exponential growth*

***E**: Use of multiple FPGA resources*

Figure 1. Growth laws for computing arrays specified in terms of structural parameters

integer value of the structural parameter *N*. If, as in Figure 1B, an array has arbitrary rectangular shape, then there are two different structural parameters, N_1 and N_2, giving the dimensions of the array. Of course, an accelerator family may have any number of structural parameters. Figure1D demonstrates an exponential rather than polynomial growth law, as just one example of growth laws of arbitrary complexity. For example, [14] uses a bus structure based on combinatorial Steiner systems. In that example, a term in the growth law involves an expression of the form $^{N!}/_{(N-K)!K!}$, for structural parameter *N* and application parameter *K*.

Figure 1C illustrates multiple coupled structures, possibly representing a cubical computation array, a square array of row reductions, and a linear array of column reductions, where the sizes of the structures are locked to each other. Each structure is assumed to have a different unit cell with different resource requirements. The growth law is represented by a polynomial with separate terms for each of the inter-related structures: $k_1N+k_2N^2+k_3N^3$. Figure 1E shows a linear structure like that in 1A, but with the added complication of consuming two different FPGA resources: logic and RAM. Depending on the specific FPGA's resource availability and the specific application's resource consumption, either of the two resources could be the one that limits the size of the array.

Of course, all of these features can occur in the growth law for any one system. One computing array can involve

multiple structural parameters describing nonlinear relationships between coupled structures, with interlocking terms for different types of FPGA resource.

3.4. Array growth – not just loop unrolling.

Loop unrolling is a standard compilation technique, both in software compilation and in synthesis from high level languages [1]. It could, in principle, be one approach to expanding a linear structure up to the capacity of the FPGA.

The factors illustrated in Figure 1 show some of the reasons why loop unrolling is problematic at best and impossible in the general case. The first problem is that linear techniques for loop unrolling poorly represent non-linear computing structures such as cubes or trees. The second problem, suggested by Figure 1B, occurs when multiple loops are candidates for unrolling, with no clear way to decide how much unrolling should be performed along each axis.

A third problem is illustrated in Figure 1C. Typical design decomposition will break the compound structure into multiple communicating components, in any over several meaningful ways. One designer might separate the three substructures into separate components; another designer might group a plane of the cube with a column of the 2D array and a cell of the linear array. In either case, the structural parameter creates a coupling between multiple design components. Typical loop analysis operates locally, within a single component. Total resource allocation, however, depends on global analysis of multiple, coupled design components.

3.5. Optimizing the computation array

Equation (2) summarizes the problem of optimizing the computing array for a given member of some application family, subject to the constraints of a specified FPGA.

$$N_{opt} = \underset{N \mid V(N) \wedge \{ \forall j : r_j^F \geq S_j(N, B) \}}{argmax\ U(N)} \qquad (2)$$

Symbols in Equation (2) have the following meanings, defined by the FPGA, the application family, or the specific member of the application family being instantiated:

$N = (n_1, n_2, \ldots n_I)$ The set of structural parameters that define an accelerator configuration, normally a tuple of non-negative integer values. These parameters and their meanings are defined by the application family. It is worth noting that this approach inverts the usual sense of the structural parameters. Normally, they are design inputs; in this usage, they are consequences of other design decisions.

N_{opt} defines the most desirable configuration possible, or one of the configurations tied for most desirable

$V(N)$ This predicate determines whether structural parameters $(n_1, n_2, \ldots n_I)$ meet architectural validity criteria. Validity constraints on structural parameters apply not only individual parameters in N, but to arbitrary relationships between sets of parameters It expresses the idea that, in some application families, particular combinations of structural parameter values may be disallowed, without respect to the specific member of the application family or capacity of the particular FPGA in use.

$B = (b_1, b_2, \ldots b_K)$ These application-specific values represent the resource utilization for each element of the application accelerator. Although the set of structural parameters is specific to an application family, the actual values of each parameter depend only on the details of the family member. It is assumed, for now, that the values describing resource utilization are independent of the FPGA platform.

$R^F = (r_1^F, r_2^F, \ldots r_J^F)$ These values specify the integer amounts of resource j available in FPGA F. This assumes that all models of FPGA under consideration support the same types of computing resources, independent of any other model of FPGA.

$S_j(N, B)$ These functions state the consumption of FPGA resource j for design parameters N and application-specific usage coefficients B. These functions capture the growth laws of the application family. For current purposes, these functions are assumed independent of the FPGA platform.

$U(N)$ The utility function is a scoring function such that higher values indicate preferable accelerator configurations. If $U(N_1) = U(N_2) \wedge N_1 \neq N_2$, then configurations *1* and *2* are different but equally desirable; either could be chosen. Absolute values of the utility function have no significance. The ranks of utility values simply establish the ordering of more and less desirable accelerator configurations for an application family.

Equation (2) suggests that the most desirable accelerator is the largest one that fits the FPGA-specific resources, once the application family growth laws and application-specific usage demands have been specified. Maximizing $U(N)$ for configuration parameters N is, in general, a difficult problem, the exact solution of which is beyond scope of this discussion. Because realistic parameter sets N have modest numbers of structural parameters and modest integer ranges, exhaustive search of the configuration space defined by V is assumed to be acceptable.

3.6. Simplifying assumptions

Without loss of generality, predicate V and functions U and S_j are assumed to be monotonic in the following senses. Let N and N' be any tuples of structural parameters, such that N' is identical to N in all positions except one where $n_i < n_i'$. Predicate $V(N)$ is said to be monotonic if $\neg V(N) \Rightarrow \neg V(N')$. Holding all other $n_{j \neq i}$ constant, there is some limit for n_i below which all configurations are valid and above which they are not.

It is also assumed that $U(N') \geq U(N)$, i.e. that the value of an accelerator increases with at least weak monotonicity

in all components (and subsets of components) of N. Functions S_j are assumed monotonic in the same sense as U, for any given application family member characterized by some fixed B.

These constraints are not necessary for Equation (2) to be valid. There is, however, intuitive appeal in the idea that larger n_i represent larger computation structures so have utility U at least as high. There is also appeal in the intuition that larger structures consume at least as much of each FPGA resource according to S_j. The real reason for monotonicity is pragmatic, however. Monotonic objective functions $U(N)$ are far easier to maximize than non-monotonic functions. Monotonicity of $S_j(N, B)$ helps in limiting the number of configurations examined. Once some resource limit is exceeded, it is no longer necessary to continue examining configurations with larger values of n_i, since larger accelerators would consume at least as much of any resource j, and would continue to violate resource constraints. Also, non-negativity of n_i and monotonicity of predicate $V(N)$ take the place of some alternative mechanism for setting lower and upper bounds on values for structural parameters, i.e. for limits to the parameter space to be searched.

4. LAMP IMPLEMENTATION

Equation 2 provides an analytic model for exploring the space of array sizes, but it can not be readily expressed in current hardware description languages. Experience suggests that function U and predicate V often have convenient representation in closed form, but that resource estimation functions S_j are relatively complicated and must incorporate knowledge of many design components. Values B that represent resource utilization of any given family member depend on the numbers of bits in the data elements and on the complexity of the member's unique logic elements. No existing hardware design languages make these values available explicitly.

The Logic Architecture by Model Parameterization (LAMP) tool set supports parameter search using two basic mechanisms: heuristic leaf estimation, and extensions to the underlying HDL on which LAMP operates. LAMP tools acknowledge that an FPGA based accelerator involves at least two developers with different skills, design responsibilities, and preferred programming tools: a logic designer who creates the communication and pipeline structure, and an application specialist who uses the accelerated computation.

The logic designer uses a standard HDL such as VHDL, with XML-based markup language (LAMPML) to parameterize the leaf functions and data types in the design. Standard hierarchical logic design already creates a tree structure. The tree's root is the outermost design component; branches and sub-branches represent instances of components and of their inner components. LAMP allows the logic designer to overlay the call graphs of resource estimation functions onto this design hierarchy. The logic designer uses LAMP markup to define functions in each component that estimate the usage of each resource for a given set of structural parameters. That estimation function is defined in terms of estimation functions exported by the inner components, numbers of component instances, knowledge of the logic structures outside of LAMP control, and leaf resource estimates. Each component uses only local knowledge of its own structure and of symbols exported by its immediate inner components, as recommended by the Law of Demeter [9], but the recursive structure makes a global resource estimate available at the root level.

The application specialist defines application-specific data types (e.g. the kinds of characters used in string comparison) and leaf functions (e.g. the function that rates quality of match between characters). These type and function definitions are written in a C-like syntax unique to LAMP, and coupled to LAMP markup in the HDL portion of the accelerator design. Since these definitions are written in LAMPML they are accessible to the LAMPML language processor, and in particular to LAMP's resource estimation heuristics.

LAMPML provides two related primitives for resource estimation:

```
(1) n = synthSize(typeName);
```

and

```
(2) n = synthSize(fctn, typeName, …);
```

The first form of `synthSize` returns the number of bits allocated to the data type named, somewhat the way the C language's `sizeof` operator works on types or values. The second form of `synthSize` estimates the number of logic elements needed to implement function `fctn`. Since LAMPML supports polymorphic function definitions, the caller must provide parameter type declarations to disambiguate the function implementation.

LAMPML provides a second level of polymorphism at the level of a HDL design component. A FIFO, for example, may be instantiated repeatedly with different data types of different widths. The `synthSize` function works with that type parameterization, so that form (1) of `synthSize` returns a value appropriate to the actual type bound to symbol `typeName` in each different instance of the component in which it occurs. Also, because LAMPML can treat function implementations as parameters, form (2) of `synthSize` returns potentially different values according to the bindings of its `typeName` parameters and depending on the local binding of actual function definition to symbol `fctn`.

Exact resource utilization figures can only come from actual synthesis, so LAMP uses conservative heuristics to estimate the numbers of logic elements needed for a function. The `synthSize` function is part of the LAMP language tool, but is loosely coupled to other parts of

LAMP logic. That makes it easy to modify the `synthSize` logic as improved estimation techniques become available. The current implementation of LAMP supports only estimates of logic utilization, not of RAM, block multipliers, or other resources.

Additional input to LAMP describes the FPGA itself, and provides another place in which resource estimation functions may be defined. FPGA-dependent estimation functions are especially helpful for estimating block RAM utilization, which depends not only on the number of bits to be stored but also on the word width. As an example, consider a RAM logically organized as 128 words of 128 bits each. Block RAMs in the Xilinx Virtex family contain 18Kb, in word widths to 36 bits. Even though one of those block RAMs holds enough bits to contain the 128×128 bits, it take four Virtex block RAMs to implement the full word width. Block RAM sizes and supported word widths are different in different models of FPGA, so device-specific estimation functions help in retargeting of the application-specific part of an accelerator without changes to the code representing the application logic.

Referring to Equation (2), it can be seen that the `synthSize` function fills the role of application-specific usage values B, and LAMPML functions embedded in each logic component support recursive construction of estimation functions S_j. The LAMP input representing FPGA specifics is expected to define the constants R^F for the resources in FPGA F. Validity test V and utility scoring function U can be written in LAMPML with the same notation used for the other functions described above.

Case studies have shown that a modest number of structural parameters N describe many useful accelerator structures, and that useful values of structural parameters tend not to exceed a few hundred or a few thousand. Given the simplifying assumptions of section 3.6, exhaustive search of the parameter space is a practical approach to solving for N_{opt}, the most desirable accelerator configuration, in Equation (2).

Thus, LAMP primitives and programming features support automated array sizing in terms of the application family, details of the application family member, and FPGA-specific resources.

5. CONCLUSION AND FUTURE DIRECTIONS

Although sizing is very different from traditional DSE, there appear to be common low-level components in their logic. In particular, DSE typically involves synthesis estimation based on heuristics [5] or measurements of component synthesis [13]. The initial implementation of sizing logic uses only crude synthesis estimation. We look forward to improving this phase of the sizing calculations using techniques from the DSE literature.

6. REFERENCES

[1] R. Camposano. “From Behavior to Structure: High-level synthesis,” *IEEE Design and Test of Computers* pp 8-19, Oct. 1990.

[2] Cray, Inc. “Cray XD1 Supercomputer,” www.cray.com/products/xd1, 2005 (verified 3 Mar. 2006)

[3] R. Dutta, J. Roy, and R. Vemuri. “Distributed Design-Space Exploration for High-Level Synthesis Systems.” *Proc. Design Automation Conference (DAC)* 1992.

[4] D. Gajski, F. Vahid, S. Narayan, and J. Gong. “System-Level Exploration with SpecSyn.” *Proc. DAC* 1998.

[5] J. Gerlach and W. Rosenstiel. “Development of a High-Level Design SpaceExploration Methodology,” Technical Report WSI-98-13, University of Tübingen, 1998.

[6] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, M. “Stream-oriented FPGA computing in the Streams-C high-level language.” *Proc*. *FCCM*. 2000.

[7] M. Goosman. “Improve Design Performance using PlanAhead Design Tools.” *XCell Journal*, Q4 2005.

[8] S. L. Graham, M. Snir, and C. A. Patterson. Getting Up to Speed: The Future of Supercomputing. National Academies Press, Washington DC 2004.

[9] K. Lieberherr, I. Holland, and A. Riel. “Object-Oriented Programming: An Objective Sense of Style.” *Proc. OOPSLA 1988*, p.323-334

[10] D. Rautela and R. Katti. ”Design and Implementation of FPGA Router for Efficient Utilization of Heterogeneous Routing Resources,” *Proc. Annual Symp. on VLSI. 2005*.

[11] B. Schoner, J. Villanesor, S. Malloy, and R. Jain. “Techniques for FPGA Implementation of Video Compression Systems.” *Proc. FPGA* 1995.

[12] Silicon Graphics, Inc. “Extraordinary Accleration of Workflows with Reconfigurable Application-Specific Computing from SGI.” www.sgi.com/pdfs/3721.pdf 2004 (verified 3 Mar. 2006)

[13] B. So, P. Diniz, and M. W. Hall. “Using Estimates from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration.” *Proc. DAC* 2003.

[14] T. VanCourt, M. Herbordt, and R. Barton “Microarray data analysis using an FPGA-based coprocessor,” *Microprocessors and Microsystems* 28(4):213–222. 2004.

[15] T. VanCourt and M. Herbordt. “Families of FPGA-based algorithms for approximate string matching,” *Proc. ASAP* 2004.

[16] T. VanCourt, Y. Gu, V. Mundada, and M. Herbordt. “Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws.” European Journal of Applied Signal Processing, to appear 2006