

Requirements for any HPC/FPGA Application Development Tool Flow

*(that gets more than a small fraction of potential performance)**

Tom Van Court

Martin C. Herbordt

Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215
EMail: {tvancour|herbordt}@bu.edu

1 Introduction

One of the most exciting innovations in computer architecture in many years is the introduction by several major vendors of FPGA-based processing nodes. FPGAs are fundamentally different from von Neuman (vN) processors: applications are configured in circuitry, rather than programmed into software. The critical problem to be solved with respect to High Performance Computing using FPGAs (HPC/FPGA) is determining an appropriate method for configuring the FPGAs; this problem involves the often inherent conflict between performance and development cost.

The argument is currently being made, in analogy to the historic assembly language versus high level language (HLL) debate—and to a lesser extent, the similarly historic quest for portable parallel programs—that the expected mode for creating HPC/FPGA applications is with HLLs, including the direct compilation of HLL programs into gates. Moreover, the argument continues, this so-called C2gates approach can be to the exclusion of logic-aware design, such as, e.g., would require using a Hardware Description Language (HDL).

In this abstract, we argue that the conflict between C2gates and logic-aware approaches is far from settled; in particular, that C2gates as so described is fundamentally flawed; and that although C2gates will be highly useful in most design flows, *logic-aware design is essential to obtain even a fraction of the possible capability of HPC/FPGA*. Putting this another way: we argue that the tremendous performance advantage (potentially) obtained by freeing the node architecture from the vN model is not likely to remain an advantage after the vN model is reimposed back onto that node architecture.

*This work was supported in part by the NIH through award #RR020209-01 and facilitated by donations from Xilinx Corporation. Web: <http://www.bu.edu/caadlab>.

2 Design Methods

In the rest of this abstract we present a number of fundamental design methods that are both essential for creating efficient HPC/FPGA applications and likely to be beyond C2gates. Or at least beyond C2gates in the sense that the developer is not particularly “FPGA-aware” in addition to not being logic-aware. These potential monkey wrenches in the FPGA machinery (particularly 1-10 below) each have the capacity of reducing performance by an order of magnitude or more.

1. Use the correct programming model. In particular, exactly the wrong programming model for FPGA configuration is the one used for serial computing. In the classic example, a program is created that has a FOR loop with a harmless, but unnecessary, dependency among its iterations. There is little effect on serial performance, but on the FPGA, little parallelism can be extracted. There are other not-so-blatant examples: parallel programming models, such as **global shared memory**, **message passing**, and **data parallel** are likely to be better than serial, but still far from ideal. In particular, none is likely to inspire an appropriate FPGA algorithm. In contrast, certain programming styles work particularly well on FPGAs: 1D and 2D systolic arrays; associative computing, which is dominated by broadcast, matching, and reduction; and complex heterogeneous pipelines, i.e. pipelines of operations, rather than pipelines to execute single operations. These programming styles, collectively, could be an appropriate HPC/FPGA programming model.

2. Use an appropriate (FPGA) algorithm. Assuming that the programming model is correct, it is still possible to use a suboptimal FPGA algorithm. HPC/FPGA implementors often create efficient FPGA versions of the corresponding serial algorithm, e.g. by getting rid of spurious inter-loop dependencies. There is often, however, a different algorithm that is supe-

rior to the one originally chosen. One example comes from modeling molecular interactions: the standard algorithm uses FFTs, but on FPGAs direct correlation is preferred. Another example comes from BLAST: the original uses tables of pointers and random access; a preferred algorithm uses streaming and a 2D systolic array.

3. Speed match computations. Even a good algorithm can be implemented so that it does not use resources properly. For example, (pipelined) stages of a computation can have drastically different operating frequencies. If they are not "speed-matched," then they will all run at the slowest frequency. This problem can be solved by replicating stages for parallel execution so that the data production per unit time matches. This item is related to load balancing in parallel programming.

4. Use all chip resources. A related issue is that of leaving chip resources unused. Sometimes this is unavoidable, such as when an application simply does not call for a large number of multipliers or memory ports to be used. Other times, however, the design system simply will not let repeating elements in a computing array expand to fill available resources. This item is related to scalability in parallel programming.

5. Hide latency of independent functions. For example, let's say a random number is required for a particular operation. There is probably no reason that the random number cannot be computed so that it is available in time. In another example, a generalized correlation for 3D object matching, a large number of 3D rotations is required. These are executed virtually; i.e., data are fetched in the order that they would be traversed had the rotation occurred. But this only gets good performance if the computation of the rotation parameters is completely hidden. This item also has an obvious correlation with parallel programming.

6. Use appropriate constructs. (Overlaps with Use Correct Programming Model; related – do not use inappropriate constructs.) One of best features of FPGAs is their ability to do *hardware computing*. Broadcast, reduction, and leader election all can be done at electrical speeds, resulting in a large number of instruction-equivalents being executed by a single hardware structure in a single cycle. On the other hand, some operations—sparse data structures, pointer following, and non-pipelined random data access—are very slow. Also related to this item is the use of the correct hardware construct for the correct structure: For example, there are well-known ways to implement FIFOs in hardware that yield performance far superior to naive implementations.

7. Use FPGA resource types appropriately. Modern FPGAs contain much more than just loose gates. In particular, they contain hundreds of embedded block RAMS and multipliers. In a molecular dynamics application, we access 400 separate memories for two reads and two writes on every cycle, yielding on overall on-chip memory bandwidth of 20Tb/s. Just as important is to not use resources incorrectly. For example, configuring chip resources into complete CPUs (softcores) for high performance applications is unlikely to be beneficial. The small number of such CPUs that can be configured into a high-end FPGA is unlikely to match the performance of even a single microprocessor, much less offer significant speed-up.

8. Arithmetic 1: Use appropriate precision. Certain applications—notably those involving biological sequences, but also many others—require only a few bits of precision. Using appropriate precision allows a proportional increase in parallelism and so performance.

9. Arithmetic 2: Use appropriate operations. The most obvious of these (unfortunately) is to limit the use of floating point. An FPGA implementation of a floating point unit following the complete IEEE standard requires the bulk of a high-end FPGA. There are often substitutes, however, that cost little performance, and use far fewer resources. For example, in a molecular dynamics application, floating point was successfully replaced with a combination of table look-up, "semi-fixed floating point," and higher order interpolation.

10. Overall 1: Use good logic design. Poor HDL-level design leads to even poorer logic. One of the most common pitfalls in modern logic design is the use of HDLs without an understanding of what logic will actually be generated. Inexperienced designers quickly learn that seemingly trivial oversights or imprecisions lead to logic that is correct, but that runs at a small fraction of projected performance. One vendor of a commercial C2gates tool reports that 180 lines of HLL code generated 150,000 lines of VHDL code.

11. Overall 2: Recognize that there are inherent limits to logic synthesis (compilation). There exist formal results proving that an optimal design can be unreachable even from a semantically equivalent but syntactically different description.

12. Overall 3: Recognize the brittleness of high-performance solutions. As we know from Amdahl's Law, the greater the potential speed-up, the more difficult it is to obtain a large fraction of that speed-up.