Abstract:
Making FPGAs a Cost-Effective Computing Architecture

Tom VanCourt, Yongfeng Gu, Martin Herbordt
Department of Electrical and Computer Engineering, Boston University
{tvancour, maplegu, herbordt} @ bu.edu

Although Field Programmable Gate Arrays (FPGAs) are still used primarily for signal processing and glue logic, it is well known that a skilled circuit designer can get a 1000-fold speed-up over a PC for a variety of applications. Recent work by our group has shown that such speed-ups are possible not just for isolated solutions, but for entire classes of computations. Moreover, these techniques point to methods for achieving the "Holy Grail" of FPGAs a cost-effective architecture: enabling the creation of high-performance FPGA-based applications by High-Level Language (HLL) programmers.

In this abstract we briefly sketch (1) the state of HW art that enables the performance, (2) previous attempts to raise the level of design abstraction to enable HLL programmers to develop general applications, and (3) our work on this problem.

FPGA characteristics and appropriate applications
Modern high-end FPGAs have not only millions of configurable gate equivalents and interconnects, but also large numbers of hard-wired components. The latter may include several hundred independent caches and ALUs, and 1 or more complete microprocessors. The former can be configured various ways, including 10,000-bit wide processors, pipelines with hundreds of stages, collections of cellular, associative or SIMD processors, systolic arrays, and many other alternatives. In addition, FPGAs have several hundred of conventional I/O pins typically used for access to memory or I/O devices, and up to 30 8Gb/s transceivers for high-speed network interconnect. FPGAs represent a mature and flourishing area of technology development, and together with graphics processors, are the driving application in new generations of silicon process technology.

This computational capability makes them attractive for computing numerous types of applications. Ideal candidate applications involve heavy computation and significant re-use of input data or intermediate results. Applications may combine high initiation latency with very high memory bandwidth, as in typical vector operations. But unlike clusters, the massive on-chip interconnect means that algorithms with heavy use of fine-grained communication are especially appropriate. Furthermore, the greatest advantage of MPPs, the large number of "memory pipes" is replicated in the many on-chip RAM blocks. This allows complex interaction among the data-fetch streams, including the kind we presented at BARC04. FPGAs tend not to work well when large amounts of precision are required over wide dynamic ranges, as vendors have as yet not emphasized incorporated floating-point support into the FPGA fabric. Although FPGAs typically process more payload data per cycle by eliminating loop overhead and access delays, they work well only when applications contain enough parallelism to compensate for their relatively low clock speeds.

Previous attempts at raising the level of abstraction

The fundamental problem in FPGA-based application development is programmability. Previous attempts have generally been based on hardware description languages (HDLs) or on high level languages (HLLs). The former, even at the so called behavioral or architectural level still have the circuit as unit of abstraction. Even at their highest levels, their primary benefit is in project management and simulation. Use of HLLs has rarely achieved even a 10 fold speed-up for any application, making them unacceptably inefficient.

The basic failure of HLLs in achieving high performance on FPGAs is the well-known *semantic gap*. An HLL's basic operations have only an indirect and ambiguous mapping to the operations and physical structure in an FPGA. Worse, many aspects of FPGA computation have no representation at all in typical HLLs. Timing, synchronization, and parallelism are recognized problems in standard parallel programming, and worse in FPGAs because the level of parallelism is so much higher relative to the size of the parallelized operations. The semantic gap works both ways: anyone who has mastered FPGA-based design is unlikely to know much about the application domains, including bioinformatics and computational biology (BCB), that are likely to benefit from hardware acceleration.

Our work

Our work separates the system- and timing-dependent aspects of designing a hardware accelerator from the domain-specific logic of the application being accelerated. We observe that wide ranges of applications have common patterns of data access and communication, irrespective of the application details being accelerated. Given that units of application behavior can be 'parameters', it is possible to parameterize the memory and synchronization structures in a reusable way. Instead of creating reusable leaf components to be combined with application-specific glue logic, we create reusable pipelines in terms of unpopulated black boxes representing application-specific data and computations.

We are developing application libraries and tool sets that implement this design approach. New tools are required for several reasons. First, although VHDL supports limited kinds of type polymorphism and component redefinition, VHDL language primitives lack the flexibility required by our applications. Equally important, VHDL has no inherent way of accepting application logic created in other, more user-friendly representations.

The development package consists of XML applications libraries and a set of tools, to be used with any standard synthesis tool flow. XML is used as an intermediate-level representation, allowing great flexibility in the actual syntax or appearance of the user's design environment. Each application library consists of a set of parameterized VHDL files coupled to abstract classes representing the interface to the application-specific logic. A FPGA-based accelerator is the result of filling in concretions of the application logic, sizing the accelerator to the FPGA platform at hand, and synthesizing the resulting logic. Initial versions of the tool set require manual sizing, but the XML representation can also work with heuristic or synthesis-based estimates of the number of processing elements that a given platform can support for the given application's logic.