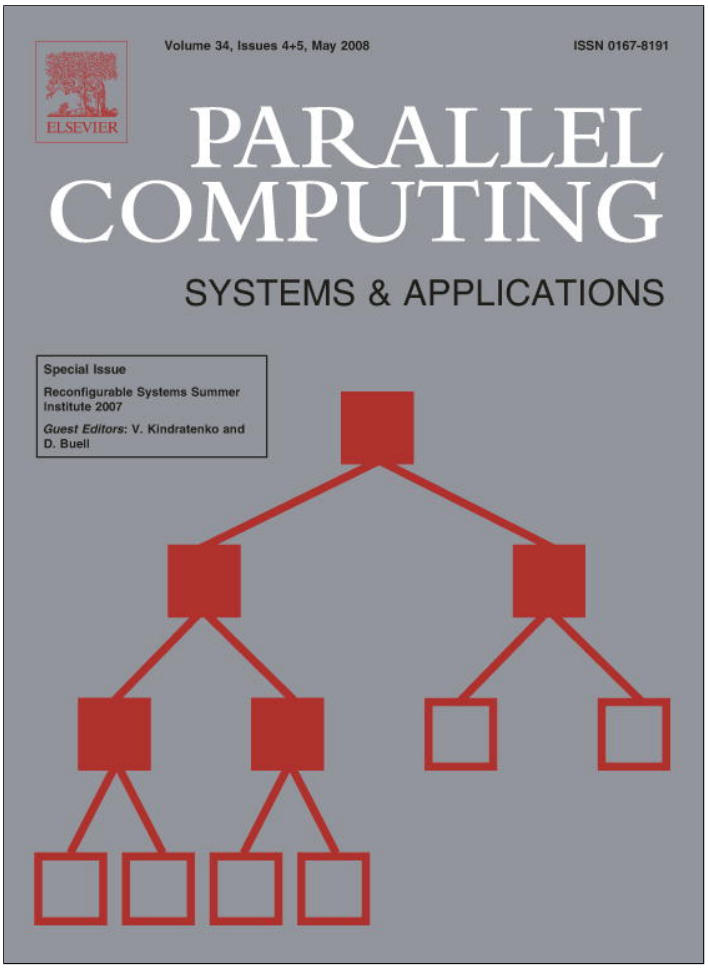

Parallel Computing 34 (2008) 245–260

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Portable library development for reconfigurable computing systems: A case study

Proshanta Saha [a,*], Esam El-Araby [a], Miaoqing Huang [a], Mohamed Taher [a], Sergio Lopez-Buedo [a], Tarek El-Ghazawi [a], Chang Shu [b], Kris Gaj [b], Alan Michalski [c], Duncan Buell [c]

[a] Department of Electrical and Computer Engineering, The George Washington University, 801 22nd St. NW, Rm. 624A, Washington, DC 20052, United States
[b] Department of Electrical and Computer Engineering, George Mason University, 230 Science and Technology II, 4400 University Drive, Fairfax, VA 22030, United States
[c] Department of Computer Science and Engineering, University of South Carolina, 3A01 Swearingen Engineering Center, Columbia, SC 29208, United States

ARTICLE INFO

Article history:
Received 11 December 2007
Received in revised form 4 March 2008
Accepted 26 March 2008
Available online 28 March 2008

Keywords:
Reconfigurable computing
Portable libraries
FPGA
Hardware design methodologies

ABSTRACT

Portable libraries of highly-optimized hardware cores can significantly reduce the development time of reconfigurable computing applications. This paper presents the tradeoffs and challenges in the design of such libraries. A set of library development guidelines is provided, which has been validated with the RCLib case study. RCLib is a set of portable libraries with over 100 cores, targeting a wide range of applications. RCLib portability has been verified in three major High-Performance reconfigurable computing architectures: SRC6, Cray XD1 and SGI RC100. Compared to full-software implementations, applications using RCLib hardware acceleration cores show speedups ranging from one to four orders of magnitude.

1. Introduction

When reconfigurable computers first appeared in the late 1980s [1], they were basically FPGA coprocessing boards connected to a conventional computer. The logic resources of the FPGA were used to create coprocessors that accelerated a given application, obtaining significant speedups by exploiting hardware fine-grain parallelism. However, these machines implied a dedicated design of coprocessors. The application was analyzed in order to find the sections of the code that performed poorly on the processor. Those sections, especially the ones that showed a high degree of parallelism, were ported to the FPGA. As a result, each application required a custom hardware design. Due to the customization process, these machines were also known as custom computers [2].

The concept of High-Performance Reconfigurable Computer (HPRC) appeared in the early 2000s, when the idea of reconfigurable computing was first applied to High-Performance Parallel Computing (HPC) systems [3]. HPRCs are parallel computing machines that have multiple processors and multiple FPGAs, so they combine fine-grain hardware parallelism with system-level HPC parallelism. The benefits that reconfigurable computing provides in terms of hardware acceleration are amplified with the scalability offered by HPC systems. In fact, speedups reaching three or four orders of magnitude have been reported, as well as one order of magnitude power consumption reductions and improvements in both cost and area [4,5]. However, the design methodology for HPRCs still resembles their custom computing origins. For each application, a

* Corresponding author. Tel.: +1 703 726 8363; fax: +1 202 994 0227.
E-mail addresses: sahap@gwu.edu (P. Saha), esam@gwu.edu (E. El-Araby), mqhuang@gwu.edu (M. Huang), mtaher@gwu.edu (M. Taher), sergiolb@gwu.edu (S. Lopez-Buedo), tarek@gwu.edu (T. El-Ghazawi), cshu@gmu.edu (C. Shu), kgaj@gmu.edu (K. Gaj), michalsk@engr.sc.edu (A. Michalski), buell@engr.sc.edu (D. Buell).

0167-8191/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.03.005

dedicated hardware accelerator has to be designed and implemented. The scalability achieved at the system level is not reflected at the tool level, and the execution speedups achievable with HPRCs might easily be eclipsed by long development cycles. Moreover, this methodology does not take advantage of the significant amount of overlap among different applications in a field. For example, many image processing applications use the same kernels, and many public-key cryptography algorithms use the same modular operations. This lack of efficient design reuse strategies causes a significant part of the development time to be wasted in implementing hardware similar to that of previous projects.

This paper proposes the use of portable libraries of highly-optimized hardware cores to address these issues. The benefits of optimized portable libraries have been demonstrated at the software level by the many proprietary and open source packages available to the HPC community. This work is aimed at expanding this idea to the HPRC arena, that is, to the hardware level. The goal is to provide the end-user with a collection of acceleration cores that can be used among different HPRC machines. These portable libraries allow end-users to develop applications by simply instantiating hardware cores, significantly reducing development time and the requirements for hardware design skills. In order to be efficient, the contents of the library have to be chosen in such a way that the available cores can apply to many different algorithms in a given application field or fields. In addition, cores should be highly optimized in order to obtain the maximum possible performance.

This paper is based on the experience gained in the Reconfigurable Computing Library (RCLib) project, developed by The George Washington University (GWU) and academic partners such as George Mason University (GMU) and University of South Carolina (SC). To the best of our knowledge, this is the first set of libraries that provide a comprehensive collection of optimized hardware cores that can be ported among different HPRC systems (SRC, Cray, SGI). Although other open source hardware repositories exist, such as Opencores [6], they are neither geared towards HPRC systems, nor do they provide a systematic approach to portability. Throughout the development cycle of RCLib, different tradeoffs and challenges in the design of such libraries were identified. Additionally, a library development guideline was created to assist developers throughout the process. This guideline addresses key concerns such as performance vs. portability, code reuse vs. application specific needs, collaboration between multiple sites, generic vs. architecture specific library distribution, licensing issues.

This paper is broken into 7 sections. Section 2 introduces the related work in this field. Section 3 provides a brief introduction about HPRC and how to develop hardware acceleration cores for these systems. Section 4 discusses the tradeoffs and challenges in the design of portable libraries of hardware cores. Section 5 presents the proposed library development guideline, and shows how it was applied to the RCLib case study. Section 6 presents usage and performance results of the RCLib implementation on the SRC6, Cray XD1, and SGI RC100 platforms. Finally, section 7 concludes the paper and presents future work.

2. Related work

The use of optimized libraries is a common practice in the development of high-performance scientific applications. Libraries provide the basic mathematical building blocks necessary to construct these applications, such as linear algebra packages, matrix operations, transforms, convolutions, etc. HPC vendors provide users with comprehensive libraries such as SGI/Cray SCSL [7], HP MLIB [8] or IBM ESSL [9]. Most of them are based on classic projects such as BLAS [10] or LAPACK [11]. Vendor's libraries are not the only resource available to users, open-source projects such as GNU's Scientific Library (GSL) [12] or GNU's Multiple Precision Library (GMP) [13] also provide good alternatives. The use of optimized libraries still has a number of open issues, such as productivity and maintenance. In [14],Van De Vanter et al. proposed a HPC software productivity infrastructure, placing an emphasis on portability. Using Component-Based Software Engineering in HPC scientific applications has also been suggested [15]. Other authors such as Ratiu et al. [16] stress the importance of libraries, focusing on their utility in actual applications rather than trying to create a generalized library.

In the reconfigurable computing domain, the use of pre-built hardware components to address the difficulties of FPGA design is also a common practice. FPGA vendors provide libraries of highly parameterizable components. For example, the Xilinx Coregen tool [17] provide cores ranging from low-level functions, such as memories and logic blocks, to complex communication blocks such as Ethernet controllers and a wide collection of encoders and decoders. Altera provides a similar solution called IP Base Suite [18], but both focus more on communications and signal processing applications than in scientific ones. An interesting approach to component-based hardware design is the use of graphical tools. One example is System Generator from Xilinx [19], which uses a set of Coregen parameterizable cores in the context of Matlab's Simulink tool. A comparable solution, RCToolbox [20], is provided by DSPLogic. However, all these Simulink-based tools are strongly geared towards DSP applications. Some reconfigurable computing vendors also provide graphical, component-based tools. Two good examples of such tools are Viva from Starbridge systems [22] and CoreFire from Annapolis Micro [23].

In summary, the existing libraries of hardware cores provided by either FPGA or RC vendors certainly ease application development, but they lack two important characteristics. First, portability, as the designs created with the libraries and tools provided by one vendor cannot be used in a system supplied by another vendor. Moreover, it is not uncommon to see libraries that are not portable between different architectures of a same vendor. A major application redesign is needed when the application is ported to a newer system. A second problem is the granularity of the operations provided by the cores. In most cases, cores offer a basic and limited functionality, making the development of complex scientific applications difficult.

Apart from the core libraries and associated tools provided by FPGA and RC vendors, there are a number of open-source hardware core repositories. One of the most prominent ones is Opencores [6]. However, these repositories work as big warehouses where contributors freely download their designs, without having to adhere to any coding or interfacing conventions. Although the cores usually are vendor-agnostic, portability is jeopardized because of the lack of a common interface. Opencores tried to address this problem by defining the Wishbone interface, but this is a bus oriented towards System-on-a-Chip designs, with limited use in HPRC systems. Unfortunately, standard core interfaces are still lacking in the HPRC arena, but there are a number of ongoing efforts in the HPRC community geared towards this standardization, most notably OpenFPGA [24] and The Spirit Consortium [25].

3. Hardware acceleration in HPRC systems

This section provides a brief overview of how hardware acceleration is implemented in HPRC systems. HPRC architectures are introduced in section 3.1. It is then followed in Section 3.2 by the description of core services, the components provided by vendors to implement the communication between the core and rest of the system. Finally, the broad spectrum of design tools available HPRC users of systems is briefly outlined in Section 3.3.

3.1. HPRC Architectures

Fig. 1 shows the two basic types of HPRC systems [29], uniform node/non-uniform system (UNNS)seen on Fig. 1a [5], and non-uniform node/uniform system (NNUS) on Fig. 1b [5]. In the former, nodes are uniform because they contain only one type of computing device, but the system is non-uniform because it has two different kinds of nodes, either processors or FPGAs. This is the approach used for example in the SGI RC100 and SRC6 platforms. Nodes containing just shared memory are also possible, as in the SRC6 platform. Non-uniform node/uniform system architectures have nodes containing two different computing devices, namely processors and FPGAs (non-uniform nodes) but there is only one type of compute node (uniform system). This is the approach used in the Cray XD1 platform.

All HPRC systems have two important points in common. First, the communication between the microprocessors and the FPGAs is performed via high-speed, usually proprietary links. Standard busses such as PCI-X do not provide the bandwidth and low latency required by HPRC systems. Second, FPGAs have a local memory at their disposal, typically 2 to 6 banks of high-speed SRAM.

3.2. Core services

HPRC vendors provide users with mechanisms to support core development, commonly known as core services. They come in the form of pre-built software and hardware components that implement the communication between microprocessor and FPGA (that is, data exchange and synchronization) and provide access to FPGA local memories.

On the hardware side, core services free developers from having to deal with the complex protocols used by the proprietary communication links and local memories. Core services hide all the communication details, providing a simpler interface to the core. As it can be seen in Fig. 2, the implementation of cores in an HPRC system follows a layered approach. The

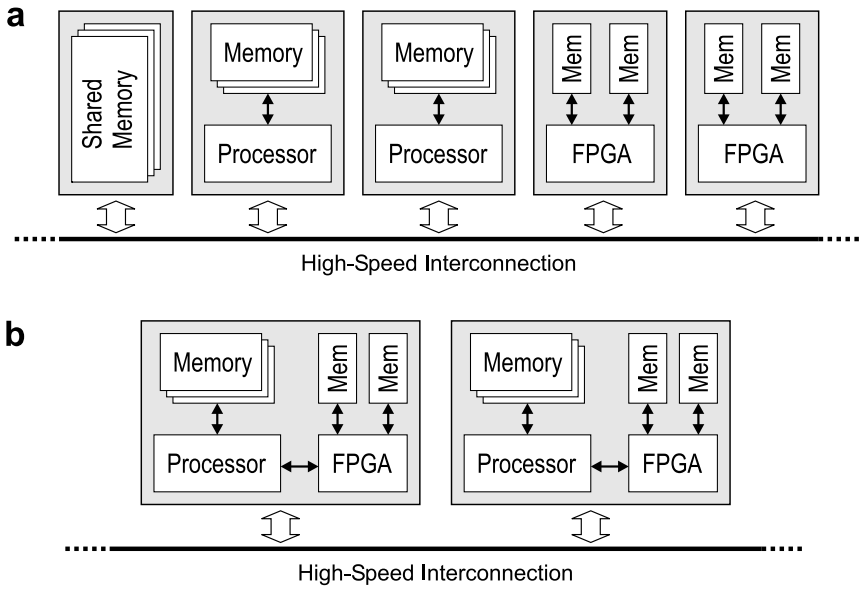

Fig. 1. Basic HPRC architectures: (a) uniform node/non-uniform system and (b) non-uniform node/uniform system [5].

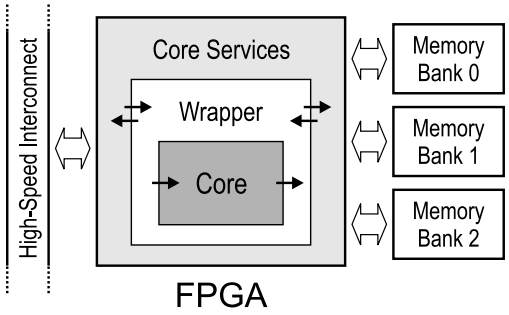

Fig. 2. Implementation of hardware acceleration cores in a HPRC system.

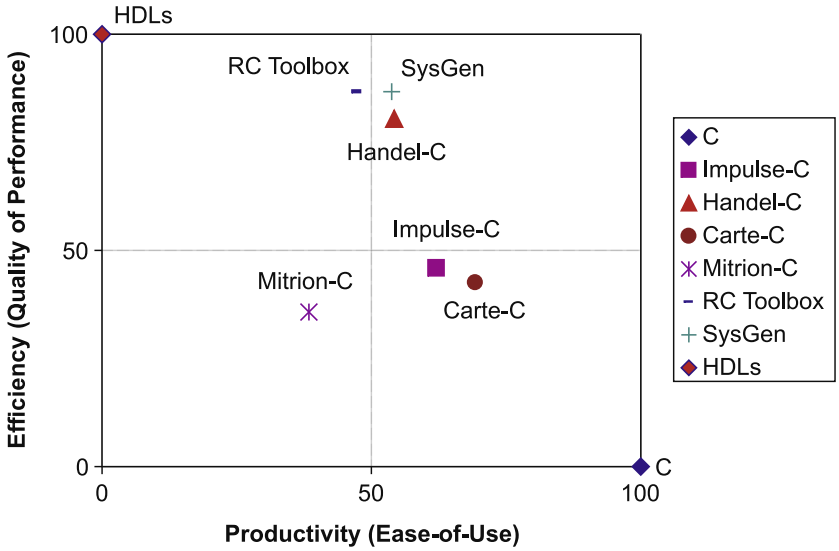

Fig. 3. Tradeoff between efficiency and ease-of-use for several development tools and languages [30].

first step is to design the core that implements the algorithm that has to be accelerated. This core is instantiated in a wrapper, which adapts the core interface to the signals provided by the core services. Finally, the wrapper is connected to the core services provided by the vendor, which creates the top level that is synthesized and implemented in the FPGA.

On the software side, core services provide an API which offers procedures to download the bitstreams, to manage data movements between microprocessor and FPGA, and to control hardware execution.

3.3. Development tools

To implement their cores, users can choose among many different languages and design tools. The reason for this diversity comes from the growing interest to provide easy-to-use alternatives to classic HDLs. Tools such as Carte-C [21], Impulse-C [26], Mitrion-C [27], and Handel-C [28] offer text-based high level language (HLL) synthesis. The other alternative, presented earlier in Section 2, are graphical tools based on the instantiation of elements from a library of parameterizable cores. Some examples of these tools are DSPlogic's RC Toolbox [20], Xilinx's SysGen [19], Starbridge Viva [22] and Annapolis CoreFire [23].

However, none of these higher-level solutions provide performance similar to that of HDLs. Fig. 3, as presented in [30], summarizes the results of a comprehensive study we undertook at The George Washington University. This figure shows the tradeoff between ease-of-use and performance. The metrics are normalized, so that 100% efficiency and 0% ease-of-use corresponds to an optimal HDL design, an 100% ease-of-use and 0% efficiency relates to a plain C implementation.

4. Tradeoffs in the development of hardware libraries

This section is aimed at identifying the design tradeoffs, as well as listing the similarities and differences in the development process of hardware and software libraries. In particular, five key issues related to building portable libraries are discussed. The first issue is domain analysis, required to establish the library scope and contents. The second issue is with respect to adhering to a standard interface definition. This enables quick porting of cores to new architectures. The third

issue addresses code portability. Balance between portability and performance needs to be taken into consideration when defining guidelines for library core development. The fourth issue is in regards to quality control. Verifying the library at each step of the development process helps alleviate integration issues, particularly when dealing with a large number of contributors. And finally the issue of library distribution, licensing, and collaboration is addressed. The management of intellectual property (IP) and navigation among the complex license agreements are necessary to ensure that the distribution of the library is done correctly.

4.1. Domain analysis

Hardware libraries can take advantage of the domain analysis techniques used in software engineering for specifying library components [31]. Finding common elements in a given domain can be very valuable for identifying the cores that a hardware library should contain. There are two important issues that have to be taken into account. First, not all algorithms are suitable for hardware implementation. Prohibitive resource requirements or poor speedup can negate the benefits of hardware. Once the library scope and the target applications are specified, the developers should identify the performance bottlenecks and analyze how they can be improved with hardware acceleration. The existing literature in reconfigurable computing can provide valuable guidelines to ease this process.

Another important difference between hardware and software comes when considering the tradeoff between generality and application-specific functionality. This tradeoff will determine the granularity of the cores. Breaking down the functionality into finer and more generic units allows for more re-use. However, the performance of an application-specific implementation is expected to be better than the one constructed out of generic blocks. Moreover, it might be difficult for the end-user to build a real application out of small and generic cores.

4.2. Standard interfaces

The use of standard interfaces is key [32] to ensuring portability. This is easy to achieve in other domains where standard busses exist, such as the CoreConnect, AMBA or Wishbone standards used for SoC connectivity. However, in the HPRC field there are no standard interfaces. Since the interfaces are unique for each vendor, the end user is responsible for creating the necessary wrappers to communicate with the library cores for each platform. These wrappers implement the interface to the vendor core services used for sending/collecting data to/from the core, see Section 3.2. Here there is a tradeoff between portability and end-user design effort. In order to ensure portability, architecture-specific details should not be defined inside the core, but instead should be specified in the core wrapper by the end-user.

As long as standard interfaces are not available, the library designer will have to specify a common interface to be used by all cores. As it was mentioned in Section 2, this is an active topic being addressed by groups such as OpenFPGA [24] and the Spirit Consortium [25]. Two basic requirements for interfaces have been identified in this work, namely simplicity and modularity. Simplicity facilitates the design of the core wrappers. Modularity, as previously mentioned, is important since only the basic building blocks are implemented and not complete algorithms. Depending on the resources available and the performance required, several blocks may have to be cascaded in order to implement part of or the full algorithm.

To simplify the interface definition it is necessary to identify the minimum requirements of the core interface. First, cores should be synchronous, so a clock and reset signal will always be present. Second, the developer should use a dataflow model, where a core has several data inputs and an output for the results of the processed data. Processing can be either pipelined or non-pipelined. In the former case, results appear at the output after a given number of clock cycles. In the latter case, execution takes a certain number of cycles so a handshake protocol is needed.

4.3. Code portability

The first step to ensure portability is to use a design flow available in all reconfigurable platforms. As discussed in Section 3.3, there are many different development tools in the market, including hardware compilers for high-level languages such as C or Matlab. However, currently HDL (VHDL or Verilog) is the only design flow that exists in all reconfigurable platforms. The same is also true for software development, standard languages such as C or Fortran plus MPI are most commonly used in reconfigurable platforms due to their ubiquity.

HDLs are relatively low-level languages and are known for their lengthy design time and steep learning curves, see Section 3.3. Here the tradeoff is design time vs portability. The only common design tools in all reconfigurable computers are HDL compilers. Compilers for high-level languages such as Handel-C or Mitrion-C are not available as part of the standard development tools and must be purchased separately. Moreover, some of these tools rely on the existence of drivers for each platform. As a result, a reduced development time may be offset by the increased effort in the necessary wrappers and interfaces for the target system.

HDL alone does not ensure portability. Optimized HDL cores that rely on the instantiation of FPGA components like embedded memories and multipliers are not portable. The tradeoff here is between speed and portability. The best performance is attained when proprietary component instantiations and placement constraints are used. However, the only portable coding style HDL is RTL behavioral.

Another key issue in code portability is setting a minimum clock speed. The goal is to achieve an effective portability, that is, not only guarantee that the core compiles and executes in all systems, but also that a uniform performance is achieved among all target architectures.

4.4. Quality control

The process of ensuring the quality of a library follows the normal software engineering principles, namely unit testing, regression testing, release builds, and benchmarks. The main difference is that the testing will be performed at two levels, HDL simulation and actual hardware implementations. HDL simulation is useful during the early stages of the design to support code development with relatively basic functional simulations. As HDL simulation is known to be very slow, more systematic tests need to be performed on real implementations of the core. This approach is known to be several orders of magnitude faster than HDL simulation, but requires the core to be synthesized and physically implemented in the FPGA. This is a very time consuming process and it is not uncommon to see cores requiring hours to be synthesized, placed and routed.

Frequent unit testing and regression testing ensure that the implementation quality is maintained throughout the development process. The unit tests are made up of short verification tests designed to detect functional and compilation errors. Regression tests allow for a comprehensive capture of errors and failures in a hardware library. Builds and regression tests help developers quickly determine the stability of their contributions and verify that previously working modules still function as expected. Although frequent builds are encouraged, they may be cost prohibitive due to time taken by the FPGA toolchain (synthesis, place and route). A typical build process can take several hours to properly run a single core through the toolchain.

Finally, benchmarks serve as a quality assurance check to ensure that the current release exceeds and/or meets the performance of previously released libraries. Here, the term benchmark refers to the performance measurements of the hardware library components, most notably speedup, and does not necessarily reflect any industry specific benchmark. Benchmarks can also be utilized by the end-user to find out which HPRC platform is more adequate for their applications.

4.5. Collaboration, distribution and licensing

Libraries tend to be large projects with a large number of contributors working on multiple branches and deliverable schedules. To avoid any conflicts, access and submission procedures and policies are essential. Fortunately, these issues are similar for both hardware and software developments, so existing software engineering practices also apply here. An important management and policy enforcement resource is source control software. This is a topic that can easily fill volumes, but is often underestimated by hardware developers. Developers not familiar with software engineering practices, often overlook the simplicity and resource management provided by source control software such as CVS [33] and SVN [34]. Licensing and distribution policy is yet another often overlooked aspect in library development.

There are two key decisions to be made when releasing a library. First is the distribution format, that is choosing between HDL source code or precompiled binaries. While precompiled binaries are convenient to the end user, it is challenging for library developers to keep up with the various HPRC platforms and the different releases of the vendor core services and APIs. Perhaps the more straightforward choice is to provide the HDL source code of the core along with corresponding documentation. Here the user is responsible for creating the wrapper required to utilize the library on their target system. Although this solution has clear advantages in terms of library development time and portability, the main drawback is the hardware design skills required from the end-user. The end-user is not only required to implement the core wrappers, but also to modify the core in order to meet vendor-specific requirements such as timing, port lists, registers, and etc.

The second major decision is selecting a license to use. Licensing is perhaps one of the most confusing aspects of a software distribution or library distribution in the open source community. Unfortunately, violating or ignoring the original consent by the authors not only diminishes the value and efforts of the open source contributors, but can also inadvertently create legal worries for further distribution of work spawned. It is important to understand the level of freedom the project wishes to bestow upon the community, and still respect the original contributors' intent and intellectual property. Comprehensive information about licenses and licensing models can be found at the Open Source Initiative [35].

The incorporation of various different types of licenses brings about the confusing part in building a library. While chip manufacturers (Xilinx, Altera, etc) and/or system vendors (SRC, AMS, Cray, SGI, etc.) provide basic cores freely to their users, license restrictions allow the use only on particular platforms and architectures. This may require that certain soft cores, regardless of how trivial they may be, be re-written to prevent litigation and to allow for free distribution. Soft cores from open source communities may also impose a different license agreement, requiring careful understanding of parts that may or may not be bundled with other license agreements. The end user may also suffer from similar confusion as they may not be able to determine which license agreement holds precedence.

5. RCLib a case study

This section presents the proposed library development guideline using the RCLib as a case study. Fig. 4 presents the development flow suggested in this guideline, including main inputs to the design decisions and actions that should be taken to validate the processes.

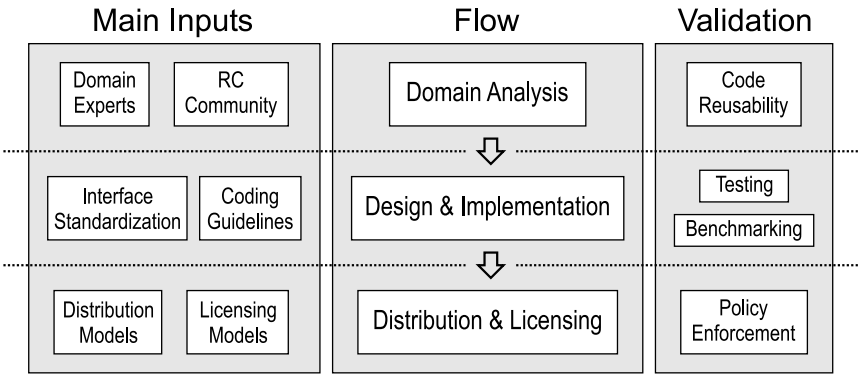

Fig. 4. Library Development Guidelines. Main Flow, inputs and validation actions.

5.1. Domain analysis

During domain analysis, it is necessary to take into consideration application acceleration, resource utilization and study of design patterns. The goal is that all cores satisfy three basic principles namely, speedup, generality, and simplicity. The first ensures that the acceleration provided by the core is high enough so that even considering Amdahl's law, the final application speedup will be significant. The second, ensures that the core is useful for a wide range of applications in a given domain, as stated by the design pattern analysis. Finally, the third ensures portability. The resource analysis identifies cores whose resource requirements compromise their portability. The objective is to avoid cores so complex that they can only be implemented in the highest-end FPGA families.

The application scope of RCLib was chosen based on input from the RC community as well as domain experts in the respective fields. Various different domains were targeted including secret key ciphers, binary Galois field arithmetic, elliptic curve cryptography, long integer arithmetic, image processing, bioinformatics, sorting, and matrix arithmetic, see Table 1. A comprehensive list of cores implemented in RCLib is detailed in Appendix A.

Often times, efficient implementations of an application keep the cores at too high a level, thereby diminishing the reuse of the core. In such cases the domain applications were re-examined to find overlap between applications and were redesigned to utilize the common cores. RCLib also utilized novel approaches such as variable size operands and common skeletons to efficiently instantiate hardware implementations of various image processing and elliptic curve cryptography applications.

5.2. Standard interfaces

Cores should have a standard interface, as this is a key issue to ensure portability, see Section 4.2. The RCLib core interface followed a few guidelines. First cores utilized single clock domain and reset ports. Second, the interface was kept as simple as possible, typically one or many data inputs and one data output. Operand size and coding were made compatible to existing C types such as 16, 32 or 64-bit integers or IEEE-754 floating point numbers. Finally, handshaking and/or synchronization mechanisms were utilized by adding two more signals, one to start processing the data, and another to signal that the processing is complete. More complex protocols were not necessary and thus were avoided.

5.3. Code portability

A simple set of coding guidelines is proposed to ensure portability. First, use HDLs (Verilog or VHDL) to develop the cores, writing only behavioral RTL descriptions. Higher-level tools do not currently enjoy ubiquity among platforms, so they are not

Table 1
Scope of RCLib

Library	Domain	Typical Cores
Secret Key Cipher	Cryptography	Encryption, decryption, breaking, key scheduling
Bioinformatics	Computational biology	Scoring, find max
Elliptic curve cryptography	Cryptography	Scalar, projection, point arithmetic
Binary Galois field arithmetic	Cryptography	Squaring, multiplication, inversion
Image processing	Image and signal processing	Filtering, buffering, transformation, correlation, histograming, remote sensing
Long integer arithmetic	Cryptography	Modular arithmetic
Matrix arithmetic	Vector processing	Bit manipulation, Bit transpose
Sorting	Data processing	Merging, sorting algorithms

recommended for library development. Additionally, HDLs still provide the best performance (see section 3.3), at the cost of longer development times.

A standard packaging format should be adhered to, which includes HDL code, debug/simulation code, black box file with interface and constraint definitions, core documentation, and a Makefile. By maintaining a minimal set of files, future library maintainers and developers have enough information required to modify the library package should constraints change in the future. A good example of this is when the timing constraints are changed to adapt to faster clock speeds. This also helps in the automation of library builds and distribution.

FPGA technology specific components should not be instantiated within the core. Instead the developer should use inference from behavioral HDL instead. However, code written in a manner that infers underlying resources is allowed and favored. For example, adjusting the size of the operands will cause the synthesis tool to use the embedded multipliers in the FPGA. Advanced vendor-specific features, for example Xilinx's RPMs, should be avoided.

Libraries included in the RCLib follow a minimum clock speed of 100 MHz (10 ns) as accepted in the SRC, Cray XD1, and SGI RASC RC systems. The RCLib also provides detailed documentation as to the pipeline latency and a timing diagram to help the end user and library developers get a better understanding as to the usage of the core. A standard set of extensions for the file names and a consistent naming convention was chosen to facilitate library integration. Following a naming convention prevents any name space clashes, ensures that library cores are not overwritten when they are built/archived, and avoids runtime and compile time errors during the integration phase.

Auxiliary functions with identical names posed a significant problem during our library integration step. Several libraries utilized similar performance measurement routines and test functions and thus clashed with the inclusion of multiple libraries due to the lack of support for overloading in C. To prevent this from occurring, a common library was created for the auxiliary functions.

Although all hardware cores in the RCLib project currently target Xilinx FPGAs, they were designed with very little to no assumption of available peripherals, memory, hard cores. This ensures that future porting to different reconfigurable elements such as Altera or future architectures can be done with minimal effort.

5.4. Quality control

Correctness and performance are essential qualities of a library. To control the quality of a library through the development cycle, it is important that guidelines are in place at each step. Software engineering practices such as unit testing, regression testing, builds, and benchmarking are essential for ensuring quality.

The tests can be performed utilizing simulation and software benchmark suites, but it is still important to execute on the target HPRC. This section proposes simple guidelines to allow verification of the library cores, see Fig. 5. First, test suites and benchmarks should be designed for the respective domains rather than adopting a more generalized test suite. Second, HDL-based simulations are only valid to help in developing the cores, and should not be used for their systematic testing due to the low speed. Third, advanced verification techniques like coverage-driven and assertion-based simulations are encouraged and should be used to identify bugs before comprehensive testing. And finally cross library tests should be designed to identify any compatibilities issues between library cores from different application domains.

To carry out the systematic testing of the cores, tests should be run on a target HPRC system, and test software program(s) can be used to verify functionality, typically using large databases of input vectors and expected results. This approach has been chosen because it provides two important advantages. First, executions on real systems are known to be orders of magnitude faster than HDL simulations. Second, it verifies the core in real conditions, so FPGA implementation and HPRC integration are also covered by this test. All developed software test programs make up the regression test suite that every release build must pass.

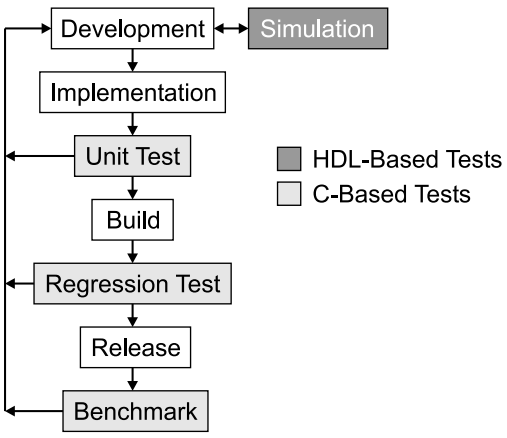

Fig. 5. Quality control flow.

Unit tests were created to quickly verify that the RCLib hardware cores compiled, built, and passed simple functionality tests. This step was sufficient to catch simple failures. Regression tests were used to go through an entire suite of tests to ensure that there are no conflicts. For a comprehensive check against an entire library, regression tests checked for compatibility with each individual library core, the runtime environment, compiler options, tool chain. Regression tests detected more subtle faults that would otherwise go undetected in simple tests, such as identifying cross library errors. Frequent builds were done to ensure the stability of user contributions and verify the functionality of previously working modules.

5.5. Collaboration, distribution and licensing

A strict methodology for collaborative work is mandatory to keep the project manageable. Well-known software engineering guidelines are proposed. First, identify the source control software supported by the target platforms. Second, establish a policy for setting access privileges to the repositories. This is essential for build schedules as they tend to be quite lengthy for a hardware library. Finally, decide on a framework for contributions to allow for users to easily extend existing library cores.

The collaboration and development framework for the RCLib took into consideration vendor compatibility, to ensure that the libraries created were readily usable. The source control software CVS was largely chosen due to support by SRC Computers. SRC's scripts allowed the pull, parsing, and building of the library for unit testing, regression testing, and finally release build.

Distribution format of the library is a key decision in the library build process and should be addressed early on. The recommended method is to provide HDL source code. Only the cores will be supplied, not the vendor-specific wrappers. Although it implies an extra effort for the end-user to implement the wrappers, its main advantage is that it does not require any maintenance effort by the library developers.

Alternatively, offering binary solutions is also possible. Binaries such as architecture specific netlists and pre-compiled bitstreams ease development efforts for the end-user. With the use of simple APIs a developer is able to instantiate a design without needing to write the necessary wrappers to interface with the library core. An added advantage of binary releases is that tool chain changes will not affect the behavior of the hardware library core. Even if a limited number of reconfigurable systems are to be supported, the problem is that core services and APIs might be changed with a new version of the development tools, so a continuous effort to keep core wrappers up to date has to be done. This is not feasible for most projects, which have a limited time scope. Providing the behavioral HDL source code for the cores will have minor backward compatibility issues.

The RCLib project provides two package formats. The first format is a binary release designed for the SRC6 platform. The library build and distribution process was done via the vendor's guidelines to ensure usability across available SRC platforms. The second format provided is a source code package which includes the HDL source code, black box interface file, documentation file, test code, test vectors. This package format was used in porting the library on to the Cray XD1 and SGI RASC platforms.

Addressing licensing and IP (intellectual property) is an important aspect in the distribution of the library. There are various open source licenses available, and picking one was not a trivial task. Choosing a popular licensing model can help in modifying, utilizing, and integrating other existing work with similar licensing agreements, such as GNU General Public License (GPL) [36]. GNU GPL, written and maintained by the Free Software Foundation, is widely used by the open source community, particularly for software developed for the Linux environment. The terms of this license are quite simple and aimed at complete freedom of use. GPL only requires that the parts modified be shared with the community for wider adoption and to also prevent derivative work from becoming proprietary. Another popular license in the open source community is the Berkeley Software Distribution (BSD) license [37]. This license is particularly friendly to commercial users who do not wish to release their modified source back into the community and allows the packages to be re-licensed into a proprietary license if needed.

To reduce the litigation burden and offer freely distributable libraries, the recommended method is to avoid utilizing third-party cores all together. When doing so is not an option, providing pre-requisites for utilizing the hardware library will alleviate the need to bundle third-party libraries and packages. This includes requiring the end user to pre-install the necessary hardware and software libraries, and in effect will satisfy the third-party licensing requirements.

Licensing was perhaps one of the most daunting aspects of the RCLib project. The mixing of proprietary licenses with open source libraries was especially challenging, as shown in Table 2. During our library development process, we were posed

Table 2
Third-party tools and licensing models utilized in RCLib

Tool/Core	License	Redistributable
Xilinx Soft Cores	Xilinx Free Software	Yes, for use with Xilinx FPGAs only
SRC Carte Library Support	SRC Proprietary	Available directly from SRC only
GMP	GNU	Yes
LiDIA	LiDIA-Group Proprietary	Yes, non-commercial only
OpenSSL	BSD	Yes

with several licensing decisions such as whether to use freely available cores from Xilinx and/or system vendors (SRC, Cray, SGI, etc.) and cope with the license restrictions that allow use only on particular platforms and architectures. The alternative was to re-write the cores, regardless of how trivial they may be to allow for open distribution. We were also faced with similar licensing concerns with soft cores, such as those provided free for use from tools such as Xilinx SysGen and others, which at times provided vague licensing restrictions. Licensing concerns also crept up during the development of the library regression testing and benchmarking suites. Open source licenses such as GNU Multi-precision (GMP) library with GPL licensing, LiDIA's non-commercial use license, as well as OpenSSL BSD style license provided similar concerns. It was often difficult to determine which license agreement held precedence.

Trying to discover other third-party tools that are compatible with the preferred license may not be trivial either. Not all third-party tools are alike, and require the scrutiny of thorough research as to their ease of use, features, documentation, performance, development language, and other tradeoffs. Although it is possible to recreate software that can enable common functionality offered by the third-party tools, it may not meet the specifications or performance requirements necessary for the library.

To alleviate the need to include the necessary tools with the library, the RCLib assumes that the end user is responsible for the availability of the necessary third-party libraries and tools on their systems. All effort is made to ensure that the tools are available to the public and are well maintained.

Due to the complex nature of the library development which includes the use tools with varying degrees of restrictions, the chosen license format is fashioned after the GNU LGPL [38] license. RCLib assumes that the end user is in compliance with the license restrictions imposed by the third-party components.

6. Usage and performance of RCLib hardware libraries

In this section, two application examples are used to describe the usage models of portable hardware libraries in HPRC applications. Section 6.1 presents a single-node remote sensing application, where different RCLib cores are used to accelerate the critical section of a wavelet-based dimension reduction algorithm. Section 6.2 presents a bioinformatics algorithm where parallelism is achieved at two levels. First, multiple RCLib hardware cores are instantiated in each FPGA accelerator. Second, conventional HPC techniques are used to run multiple hardware-accelerated nodes in parallel. Finally, Section 6.3 summarizes the benefits in terms of speedup, power and cost arising from the use of hardware libraries such as RCLib.

6.1. Single-node usage of RCLib

The first application example is Wavelet-Based Dimension Reduction of Hyperspectral Imagery. Dimension reduction is a transformation that brings data from a high order dimension to a low order dimension. This transformation is used in remote sensing applications and is known to have large computational requirements. The general description of the automatic wavelet dimension reduction algorithm is shown in Fig. 6. "PIXEL LEVEL" is the most computational intensive function on traditional platforms. Based on the separability property of individual pixels, the algorithm lends itself to parallelization and full pipelining within this specific function. Fig. 7 shows the top hierarchical level of the implementation architecture. A major component is the DWT_IDWT module. This module utilizes both the Discrete Wavelet Transform (DWT) and the Inverse Discrete Wavelet Transform (IDWT) cores of RCLib, thus producing respectively the decomposition spectrum, i.e. L1-L5, as well as the reconstruction spectrum, i.e. Y1-Y5, see Fig. 7. The second major component utilized from RCLib was the Correlator. It implements a correlation function (ρ) between the original spectral signature (X) and its reconstructed approximation (Y) which results from applying the DWT and IDWT. The final RCLib component used is the Histogram core. This module is implemented with counters which are updated according to the correlation-threshold inequality, i.e. $\rho \geqslant \mathrm{TH}$. All these cores use fixed-point arithmetic.

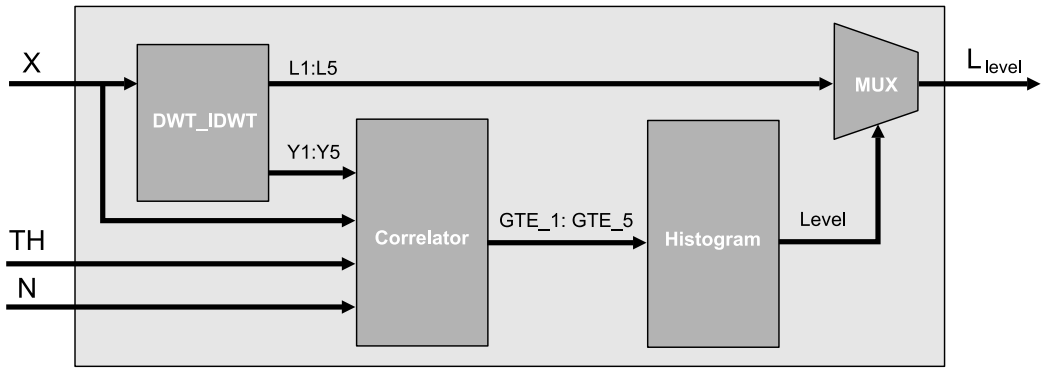

Fig. 6. Automatic Wavelet Spectral Dimension Reduction Algorithm [39].

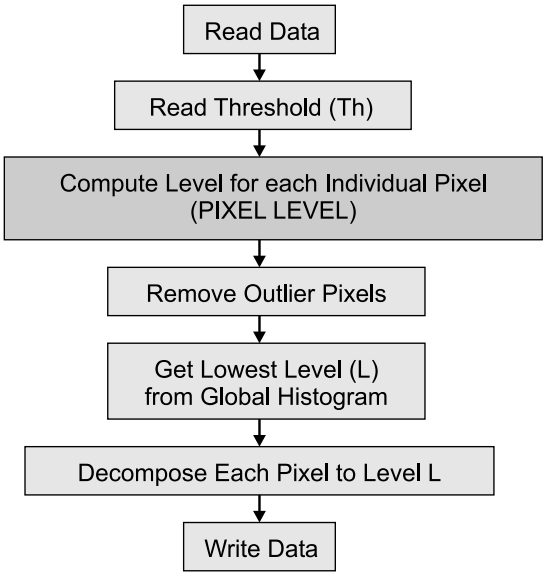

Fig. 7. Architecture of the hardware accelerator for the PIXEL LEVEL function [39].

The results from this application example, detailed in [39], show that management of the data transfers between the microprocessor memory and FPGA onboard memory becomes critical to achieve a significant speedup. In a full-software implementation, data movements only take 8% of the execution time. On the contrary, I/O might reach 75% of the total execution time on a SRC-6 HPRC implementation. It is therefore mandatory to use techniques such as sharing of memory banks or overlapping of memory transfers with computations in order to reduce the I/O overhead. Using such techniques, a speedup of 32.04x in comparison to a full-software version of the algorithm was measured for the SRC6 implementation.

6.2. Multi-node usage of RCLib

This section presents a bioinformatics application example where the two parallelism levels of a HPRC computer, chip-level and system-level, are exploited in order to obtain the maximum speedup. Chip-level parallelism is achieved through the instantiation of multiple RCLib cores, system-level parallelism is gained using conventional HPC techniques. The selected bioinformatics application is the Smith-Waterman algorithm,. This is a well-known algorithm to perform local sequence alignments, that is, to determine similar regions between two nucleotide or two protein sequences [40].

Two RCLib cores, i.e. Scoring Maximum and Score Search, were utilized to calculate the similarity matrix. A 2-bit encoding was used in DNA sequencing to represent the four nucleotides (AGCT), whereas Protein sequencing required 5-bits to encode the twenty amino acids. RCLib cores are highly parameterized, easily allowing for different schemes of encoding. A virtualization and scheduling technique was needed to handle the large sequences, and a 2-D sliding window was used to traverse the entire virtual similarity matrix. A 16 by 16 sliding window was initially proposed but proved to be inefficient since it does not engage the system to its full capacity [41]. Taking advantage of the full parameterization of RCLib cores, the window geometry was changed to 32 by 1, a size that proved to be more efficient by decreasing the stalling time of reloading the sequences.

Table 3 shows the performance of the Protein and DNA sequencing over the Cray XD1 and SRC6 as compared to the FASTA open source software running on a 2.4 GHz Opteron processor. The throughput metric used was Giga cell updates per second (GCUPS). The XD1 had an advantage with its FPGA chips running at 200 MHz, while the SRC machine had a restriction to run the FPGAs only at 100 MHz. The table shows both the expected and measured speedups, the difference is mainly due to the communication overhead. The results also show the throughput when system-level parallelism is used. In the case of SRC6 (a UNNS system), one microprocessor allocates four FPGA accelerators and distributes the workload among them. In Cray XD1, a NNUS system, nodes only have one FPGA available. Conventional MPI-based parallel programming techniques are used to distribute the workload among the six nodes available in the chassis. In this case, performance does not scale as good as in the SRC6 example due to the MPI communications overhead.

The maximum speedup of 2794x was achieved by means of the two levels of parallelism and the higher clock frequency of Cray XD1. Chip-level parallelism was accomplished by packing 8 kernels on each FPGA chip, providing a 695x speedup. Then, system-level parallelism was used to make the six nodes in the Cray XD1 chassis work in parallel, thus achieving the final 2794x speedup.

6.3. Cost, power and space savings of HPRCs using RCLib

This section estimates the required FPGA resources, the speed, power consumption and size improvements that can be achieved by HPRCs when using RCLib. In order to do so, the performance of the HPRC implementations is maximized using

Table 3
Speedup of protein and DNA sequencing [4,5]

			Expected		Measured	
			Throughput (GCUPS)	Speedup	Throughput (GCUPS)	Speedup
Opteron 2.4 GHz	DNA		NA	NA	0.065	1
	Protein		NA	NA	0.130	1
SRC 100 MHz (32 × 1)	DNA	1 Engine/Chip	3.2	49.2	3.19 → 12.2 1 → 4 Chips	49 → 188 1 → 4 Chips
		4 Engines/Chip	12.8	197	12.4 → 42.7 1 → 4 Chips	191 → 656 1 → 4 Chips
		8 Engines/Chip	25.6	394	24.1 → 74 1 → 4 Chips	371 → 1138 1 → 4 Chips
	Protein		3.2	24.6	3.12 → 11.7 1 → 4 Chips	24 → 90 1 → 4 Chips
XD1 200 MHz (32 × 1)	DNA	1 Engine/Chip	6.4	98	5.9 → 32 MPI 1 → 6 nodes	91 → 492 PI 1 → 6 nodes
		4 Engines/Chip	25.6	394	23.3 → 120.7 MPI 1 → 6 nodes	359 → 1857 MPI 1 → 6 nodes
		8 Engines/Chip	51.2	788	45.2 → 181.6 MPI 1 → 6 nodes	695 → 2794 MPI 1 → 6 nodes
	Protein		6.4	49	5.9 → 34 MPI 1 → 6 nodes	45 → 262 MPI 1 → 6 nodes

a two-level parallelism approach similar to the one presented in the previous section. The size of a computer cluster that can achieve the same level of performance is estimated, and the equivalent power consumptions, size and cost are deduced. From all of that, the level of improvements in performance, cost, power consumption, and size that can be achieved by a HPRC over a PC cluster is obtained. Tables 4–6 summarize those results.

In this analysis, a 100x speedup of an HPRC, means that the power, size and cost of the HPRC are compared to that of a 100 processor Beowulf cluster. The estimates are conservative, because when parallel efficiency is considered, a 100 processor

Table 4
Sample RCLib cores on an SRC 6 system compared with Opteron 2.4 GHz microprocessors

Application	Core Information		Speedup	Savings		
	Resource Utilization (% Slices per core)	Cores per Chip		Cost	Power	Size
Smith Waterman	6	8	1138	6	313	34
DES Breaker	7	10	6757	34	1856	203
IDEA Breaker	18	4	641	3	176	19
RC5 Breaker	99	1	1140	6	313	34

Table 5
Sample RCLib cores on a Cray XD1 system compared with Opteron 2.4 GHz microprocessors

Application	Core Information		Speedup	Savings		
	Resource Utilization (% Slices per core)	Cores per Chip		Cost	Power	Size
Smith Waterman	12	8	2794	28	140	29
DES Breaker	23	6	12162	122	608	127
IDEA Breaker	27	5	2402	24	120	25
RC5 Breaker	87	1	2321	23	116	24

Table 6
Sample RCLib cores on an SGI RC100 system compared with Opteron 2.4 GHz microprocessors

Application	Core Information		Speedup	Savings		
	Resource Utilization (% Slices per core)	Cores per Chip		Cost	Power	Size
Smith Waterman	4	30	8723	22	779	253
DES Breaker	3	38	38514	96	3439	1116
IDEA Breaker	7	2	961	2	86	28
RC5 Breaker	39	2	6838	17	610	198

cluster will likely produce a speedup that is much less than 100 folds. In other words, the competing cluster was assumed to have 100% efficiency. It was also assumed that one cluster node consumes about 220 W, and that 100 cluster nodes have a foot print of 6 square feet. Based on the actual prices for these systems, the cost was assumed to be 1:100 in the case of Cray XD1 and 1:200 in the case of SRC6.

The results show that HPRCs using RCLib might achieve many orders of magnitude improvements in performance, cost, power and size over conventional high performance computers. Moreover, when the cost factors associated with power (e.g. cooling and size) are taken into account, the reduction in cost is much more significant than even the reported in Tables 4–6. However, the speedups, power savings, cost savings and size savings shown in these tables can be viewed as realistic upper bounds of the performance of HPRC technology using RCLib. This is because the selected applications in these tables are all compute intensive integer applications, a class of applications for which HPRCs clearly excel and hence around which RCLib was developed. With additional FPGA chip improvements in size, floating point support, and with improved data transfer bandwidths between FPGAs and local memory or microprocessor, a much wider range of applications will be able to harness similar levels of benefits with minimal modifications to RCLib.

7. Conclusions

In the recent years, the hardware capabilities of reconfigurable computers in general, and HPRC systems in particular, have grown exponentially. Unfortunately the development tools have not evolved at the same pace. Among other problems, current tools have poor support for design reuse. This paper proposes the use of portable libraries of hardware optimized cores to solve this issue. The idea is to expand the well-known HPC core libraries to the hardware level. However, the development of hardware libraries poses many specific challenges. This paper provides a list of the tradeoffs and challenges that the developer will face. Unlike software libraries, there is to the best of our knowledge no previous experience in the development of a comprehensive and portable library of hardware cores. Therefore a new set of guidelines for development of such libraries had to be conceived, that should address the major concerns. These include domain analysis, code reuse, packaging, interface definition, portability, collaboration, benchmarking and testing, distribution and licensing. This guidelines have been fully validated by means of a comprehensive case study covering more than 100 different cores, targeting applications such as image processing, cryptography, bioinformatics, and long integer arithmetic. Using the RCLib case study, portability was demonstrated by implementing these libraries in machines from major HPRC vendors. The speedup results, as well as the cost, area and power savings, confirmed the benefits of this approach.

Future work include assessing portability in other non-Xilinx architectures, such as Xtreme Data XD2000i and SRC7. Work is underway for automatic generation of the vendor-specific wrappers, to help end-users in portability. Additionally, co-scheduling algorithms are currently being developed to optimize the synergy of microprocessors and hardware acceleration cores.

Finally, to request a copy of the RCLib hardware library package or for more information please visit http://hpc.gwu.edu/library.

Acknowledgements

We would like to thank the various contributors to the library. We would also like to thank Dan Poznanovic and Paul Gage of SRC Computers for their tireless efforts in integrating the necessary components to allow for end user library development and deployment in the Carte-C environment.

Appendix A. RCLib cores

See Tables 7–14.

Table 7
Bioinformatics cores in RCLib

Application	Cores
Bioinformatics	
Smith Waterman	Scoring Maximum Score Search

Table 8
Secret-key ciphering and hashing cores in RCLib

Application	Cores
Secret Key Ciphers	
IDEA	Encryption
	Decryption
	Breaking
DES	Encryption
	Decryption
	Breaking
RC5	Key Scheduling
	Encryption
	Decryption
	Breaking

Table 9
Elliptic curve cryptography cores in RCLib

Application	Cores
Elliptic curve cryptography	
Scalar multiplication	Normal basis
	Polynomial basis
Project to affine	Normal basis
	Polynomial basis
Point addition	Normal basis
	Polynomial basis
Point doubling	Normal basis
	Polynomial basis

Table 10
Binary galois field arithmetic cores in RCLib

Application	Cores
Binary Galois Field Arithmetic	
Polynomial basis	
Trinomial	Squaring
	Multiplication
	Inversion
Pentanomial	Squaring
	Multiplication
	Inversion
Special Field	Squaring
	Multiplication
	Inversion
NIST	Squaring
	Multiplication
	Inversion
Normal basis	
NIST	Squaring
	Multiplication
	Inversion

Table 11
Image processing cores in RCLib

Application	Cores
Image processing	
Wavelet	Discrete Wavelet Transform
	Inverse Discrete Wavelet Transform
	Correlation and Histograming
Filtering	Gaussian filtering
	Smoothing filtering
	Sharpening filtering
	Blurring filtering
	Prewitt filtering
	Sobel edge filtering
	Median filtering
Buffering	Line buffer

Table 12
Long Integer Arithmetic cores in RCLib

Application	Cores
Long Integer Arithmetic	
1024 bit	Montgomery Multiplier with Carry Save Adder
	Montgomery Multiplier with Carry Propagate Adder
	Modular Exponentiation with Carry Save Adder
	Modular Exponentiation with Carry Propagate Adder
1536 bit	Montgomery Multiplier with Carry Save Adder
	Montgomery Multiplier with Carry Propagate Adder
	Modular Exponentiation with Carry Save Adder
	Modular Exponentiation with Carry Propagate Adder
2048 bit	Montgomery Multiplier with Carry Save Adder
	Montgomery Multiplier with Carry Propagate Adder
	Modular Exponentiation with Carry Save Adder
	Modular Exponentiation with Carry Propagate Adder
3027 bit	Montgomery Multiplier with Carry Save Adder
	Montgomery Multiplier with Carry Propagate Adder
	Modular Exponentiation with Carry Save Adder
	Modular Exponentiation with Carry Propagate Adder

Table 13
Matrix arithmetic cores in RCLib

Application	Cores
Matrix arithmetic	
Bit matrix	Bit matrix multiplication
	Bit matrix transpose

Table 14
Sorting cores in RCLib

Application	Cores
Sorting	
Sorting	Bitonic Sorting
	Stream Sorting
	Odd–Even Sorting
	Heap Sorting
	Quick Sorting
Scheduling	Sorting Scheduler

References

[1] J.P. Gray, T.A. Kean, The case for custom computation, in: Proceedings of the 2nd International Specialist Seminar on the Design and Application of Parallel Digital Processors, 1991, pp. 60–64.
[2] D.A. Buell, J.M. Arnold, W.J. Kleinfelder (Eds.), Splash 2: FPGAs in a Custom Computing Machine, Wiley-IEEE Computer Society Press, 1996.
[3] D. Buell, T. El-Ghazawi, K. Gaj, V. Kindratenko, Guest Editors' Introduction: high-performance reconfigurable computing, IEEE Computer 40 (3) (2007).
[4] T. El-Ghazawi, K. Gaj, D. Buell, V. Kindratenko, Reconfigurable Supercomputing, in: IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC07), Reno, NV, USA, November 2007, Tutorial M04.
[5] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, D. Buell, The Promise of High-Performance Reconfigurable Computing, IEEE Computer 41 (2) (2008) 69–76.
[6] OpenCores, www.opencores.org.
[7] SGI Scientific Computing Software Library (SCSL), <http://www.sgi.com/products/software/scsl.html>.
[8] HP Mathematical Software Library (MLIB), <http://h21007.www2.hp.com/portal/site/dspp/PAGE.template/page.document?ciid=c008a8ea6ce02110a8ea6ce02110275d6e10RCRD>.
[9] IBM Engineering and Scientific Subroutine Library (ESSL), <http://www-03.ibm.com/systems/p/software/essl/>.
[10] Basic Linear Algebra Subprograms (BLAS), <ttp://www.netlib.org/blas/>.
[11] Linear Algebra Package (LAPACK), http://www.netlib.org/lapack/.
[12] GNU Scientific Library (GSL), <http://www.gnu.org/software/gsl/>.
[13] GNU MP Library (GMP), <ttp://gmplib.org>.
[14] M.L. Van De Vanter, D.E. Post, M.E. Zosel, HPC needs a tool strategy, in: Proceedings of the Second international Workshop on Software Engineering For High Performance Computing System Applications (St. Louis, Missouri, May 15–15, 2005), SE-HPCS'05, ACM, New York, NY, pp. 55–59.
[15] N. Wang, R. Pundaleeka, J. Carlsson, Distributed component support for integrating large-scale parallel HPC applications, in: Proceedings of the 2007 Symposium on Component and Framework Technology in High-performance and Scientific Computing (Montreal, Quebec, Canada, October 21–22, 2007).

[16] Daniel Ratiu, Jan Jurjens, The reality of libraries, software maintenance and reengineering, 2007, in: 11th European Conference on CSMR'07, 21–23 March 2007, pp. 307–318.
[17] Xilinx Coregen, <http://www.xilinx.com/ipcenter/coregen/>.
[18] Altera IP Base, <http://www.altera.com/products/ip/design/basesuite/ip-basesuite_qanda.html>.
[19] Xilinx Inc., System Generator, <http://www.xilinx.com/products/software/sysgen/features.htm>.
[20] DSPLogic Reconfigurable Computing Toolbox, <http://www.dsplogic.com/home/products/rctb>.
[21] SRC Computers Inc., Carte Programming Environment, <http://www.srccomputers.com/CarteProgEnv.htm>.
[22] Starbridge Systems Inc. Viva development software, <http://www.starbridgesystems.com/viva-software/what-is-viva/>.
[23] Annapolis Microsystems Inc. CoreFire Design Suite, <http://www.annapmicro.com/corefire.html>.
[24] OpenFPGA, <http://www.openfpga.org/>.
[25] The Spirit Consortium, <http://www.spiritconsortium.org/home>.
[26] Impuse CoDeveloper FPGA Compiler, <http://www.impulsec.com/fpga_c_products.htm>.
[27] Mitrionics Mitrion SDK, <http://www.mitrionics.com/default.asp?pId=23>.
[28] Celoxica DK Design Suite, <http://www.celoxica.com/products/dk/default.asp>.
[29] J. Harkins, T. El-Ghazawi, E. El-Araby, M. Huang, Performance of Sorting Algorithms on the SRC 6 Reconfigurable Computer, in: IEEE International Conference on Field-Programmable Technology (FPT 2005), Singapore, December, 2005.
[30] E. El-Araby, P. Nosum, T. El-Ghazawi, Productivity of High-Level Languages on Reconfigurable Computers: an HPC Perspective, in: IEEE International Conference on Field-Programmable Technology (FPT 2007), Japan, December, 2007.
[31] R. Prieto-Díaz, Domain analysis: an introduction, ACM SIGSOFT Software Engineering Notes 15 (2) (1990).
[32] M. Keating, P. Bricaud, Reuse Methodology Manual for System-On-A-Chip Designs, third ed., Springer, 2002.
[33] Concurrent Versions System (CVS), <http://www.nongnu.org/cvs/>.
[34] Subversion(SVN) Source Control system, <http://subversion.tigris.org/>.
[35] Open source Initiative, <http://www.opensource.org/>.
[36] GNU's not Unix General Public License, <http://www.gnu.org/copyleft/gpl.html>.
[37] BSD License, <http://www.opensource.org/licenses/bsd-license.php>.
[38] GNU Lesser General Public License, <http://www.gnu.org/copyleft/lgpl.htmlA>.
[39] E. El-Araby, T. El-Ghazawi, Le J. Moigne, K. Gaj, Wavelet spectral dimension reduction of hyperspectral imagery on a reconfigurable computer, IEEE FPT 2004, Brisbane, Australia, December, 2004.
[40] D. Mount, Bioinformatics: Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2004.
[41] M. Abouellail, E. El-Araby, M. Taher, T. El-Ghazawi, G.B. Newby, DNA and protein sequence alignment with high performance reconfigurable systems, in: NASA/ESA Conference on Adaptive Hardware and Systems 2007 (AHS2007), August 5–8, 2007, Scotland, UK.