Parallel Programming of High-Performance Reconfigurable Computing Systems with Unified Parallel C

Tarek El-Ghazawi, Olivier Serres, Samy Bahra, Miaoqing Huang and Esam El-Araby
Department of Electrical and Computer Engineering,
The George Washington University
{tarek, serres, sbahra, mqhuang, esam}@gwu.edu

Abstract

High-performance Reconfigurable Computers (HPRCs) integrate nodes of either microprocessors and/or field programmable gate arrays (FPGAs) through an interconnection network and system software into a parallel architecture. For domain scientists who lack the hardware design experience, programming these machines is near impossible. Existing high-level programming tools such as C-to-hardware tools only address designs on one chip. Other tools require the programmer to create separate hardware and software program modules. An application programmer needs to explicitly develop the hardware side and the software side of his/her application separately and figure out how to integrate the two in order to achieve intra-node parallelism. Furthermore, the programmer will have to follow that with an effort to exploit the extra-node parallelism. In this work, we propose unified parallel programming models for HPRCs based on the Unified Parallel C programming language (UPC). Through extensions to UPC, the programmer is presented with a programming model that abstracts hardware microprocessors and accelerators through a two level hierarchy of parallelism. The implementation is quite modular and capitalizes on the use of source-to-source UPC compilers. Based on the parallel characteristics exhibited at the UPC program, code sections that are amenable to hardware implementation are extracted and diverted to a C-to-hardware compiler. In addition to extending the UPC specifications to allow hierarchical parallelism and hardware-software co-processing, a framework is proposed for calling and using an optimized library of cores as an alternative programming model for additional enhancement. Our experimental results will show that the proposed techniques are promising and can help non-hardware specialists to program HPRCs with substantial ease while achieving improved performance in many cases.

1. Introduction

High-Performance Reconfigurable Computers (HPRCs) are parallel architectures that integrate both microprocessors and field programmable gate arrays (FPGAs) into scalable systems that can exploit the synergism between these two types of processors. HPRCs have been shown to achieve up to several orders of magnitude improvements in speed [1], size, power and cost over conventional supercomputers in application areas that are of critical national interest such as cryptography, bio-informatics, and image processing [2]. The productivity of HPRCs, however, remains an issue due to the lack of easy programming models for this class of architectures. Application development for such systems is viewed as a hardware design exercise that is not only complex but may also be prohibitive to domain scientists who are not computer or electrical engineers.

Many High-Level Languages (HLLs) have been introduced to address this issue. However, those HLLs address only a single FPGA [4], leaving the exploitation of the parallelism between resulting hardware cores and the rest of the resources, and scaling the solution across multiple nodes to the developer using brute force, with no tools to help. This has prompted the need for a programming model that addresses an entire HPRC with its different levels and granularity of parallelism. In this work, we propose to extend the partitioned global address space (PGAS) scheme to provide a programming model that presents the HPRC users with a global view that captures the overall parallel architecture of FPGA-based supercomputers. PGAS provides programmers with a global address space which is logically partitioned such that threads are aware of what data are local and what are not. Therefore PGAS programming languages are characterized with ease-of-use. Two different usage models are proposed and separately investigated and then integrated to provide application developers with easy access to the performance of such

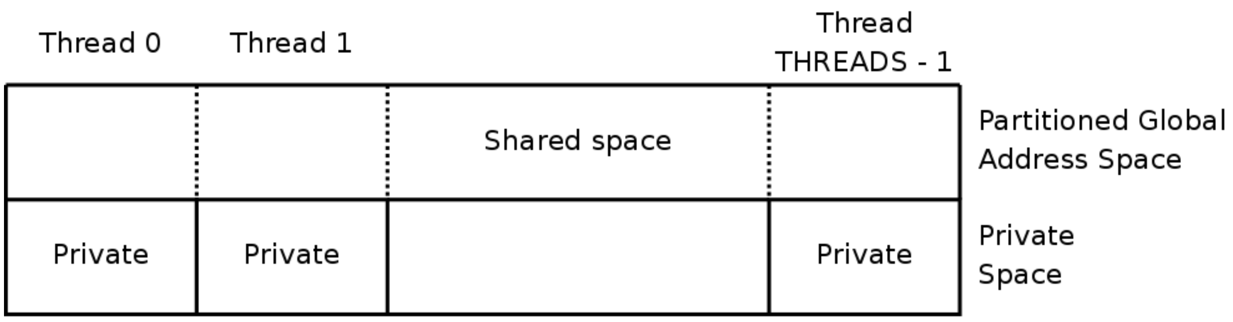

Fig. 1. The UPC memory and execution model

architectures without the need for detailed hardware knowledge. One model simply calls a library of cores from a program already written in a parallel language, where fine grain FPGA level of parallelism is handled by the library and the node and system parallelism are handled through the parallel programming mode. Another methodology considered identifying sections of codes that can benefit from hardware execution and diverting those sections to a C-to-hardware compiler. Such candidate sections can be more easily identified from a parallel language syntax and semantics than that from a sequential language. This slightly modifies the UPC programming models to allow a two-level hierarchy of parallelism, where original UPC threads are used for coarse grain parallelism across the system nodes. Another fine level parallelism is used to spawn activities within each UPC thread where such activities may be handled by an accelerator such as an FPGA. Proof of concept experiments using UPC coupled with a C-to-hardware programming interface using Impulse C have been conducted. UPC or Unified Parallel C, is a parallel extension of ISO C using the PGAS model [5].

2. Overview of Unified Parallel C (UPC)

Fig. 1 illustrates the memory and execution model as viewed by UPC programmers. In UPC, memory is composed of a shared memory space and a private memory space. A number of threads work independently and each of them can reference any address in the shared space, and also its own private space. The total number of threads is `THREADS` and each thread can identify itself using `MYTHREAD`, where `THREADS` and `MYTHREAD` can be seen as special constants. The shared space, however, is logically divided into portions each with a special association (affinity) to a given thread. The idea is to allow programmers, with an appropriate design, to keep the shared data that will be dominantly processed by a given thread (and occasionally accessed by others) close to that thread and efficiently exploit data locality in applications.

Since UPC is an explicit parallel extension of ISO C99, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control of the distribution of data across the threads. Among the interesting and complementary UPC features is a work-sharing iteration statement, `upc_forall`. This statement helps distribute independent loop iterations across the threads, such that iterations and data that are processed by them are assigned to the same thread. Such parallel processing constructs become very handy when the code is examined for fine grain parallelism that suits hardware execution. UPC also defines a number of rich private and shared pointer concepts. The language also offers a rich range of synchronization and memory consistency control constructs.

3. Proposed Solutions

The productivity problem of HPRCs can be stated as follows. At present there is not a single programming model that provides an integrated view of HPRCs with the FPGAs, the node architecture with its microprocessor(s), and the system level architecture. All available tools, even those that are C-like, allow only the programming of a single FPGA chip. In some cases they provide a view that can allow the programming of one FPGA and one microprocessor but programmers have to deal explicitly with hardware, software and their integration. This leaves many gaps in the development process for the application developer to fill with brute force. A programming model is therefore needed to provide a unified view of the overall parallel HPRC architecture, which allows exploiting parallelism at the FPGA level, exploiting the synergy between the FPGAs and the microprocessors and exploiting the multiplicity of nodes through parallel processing.

3.1 The HPRC PGAS Programming Model

In UPC. work load sharing is facilitated by the upc_forall construct. Upc_forall(a, b, c, d); has four fields a, b, c, d or initialization, test, incrementation, and affinity respectively. The iterations of a upc_forall must be independent and the last field, affinity, determines which thread executes the current iteration. Threads typically correspond to processors, as in MPI processes. The number of threads in UPC remains fixed during execution. In this work, we are proposing to extend the upc_forall such that the outer-most iteration of the loop stills correspond to major UPC threads as defined is the specifications, but the activities that take place within the next level represents finer grain activities that take place within the executing thread. Those activities can be associated by the compiler with the available accelerating hardware. In the case of an FPGA, the number of such activities as expressed by the programmer may exceed the available resources. In such cases, techniques such as striping can be used. In the prototype that was created for proof of concept only simple case of upc_forall are considered.

3.2 Implementation techniques

In this work, we are proposing a solution that provides application developers with a unified programming model that addresses the entire parallel architecture of a HPRC, through the partitioned global address space (PGAS) paradigm.

Under this model, each UPC thread, running on a processor, will be paired with a reconfigurable engine (FPGA) to provide hardware co-processing capabilities for that thread. A simple library call and/or a diversion in the compiling process will provide the capability of creating hardware that is seamlessly interfaced to UPC. On the other hand, UPC will continue to maintain a global view for the whole HPRC system in the PGAS/SPMD context. Ideas for expanding UPC specifications to exploit the power of such architectures will be explored.

A testbed and a reference implementation will be used to provide a proof of concept as well as insights. In the rest of this document we will provide more details on our approach and some of its elements. In this work, the UPC interface is facilitated through two basic ideas: 1. the use of an optimized library and 2. the use of source-to-source UPC compliers and the diversion of selected code sections to a C-to-Hardware compiler.

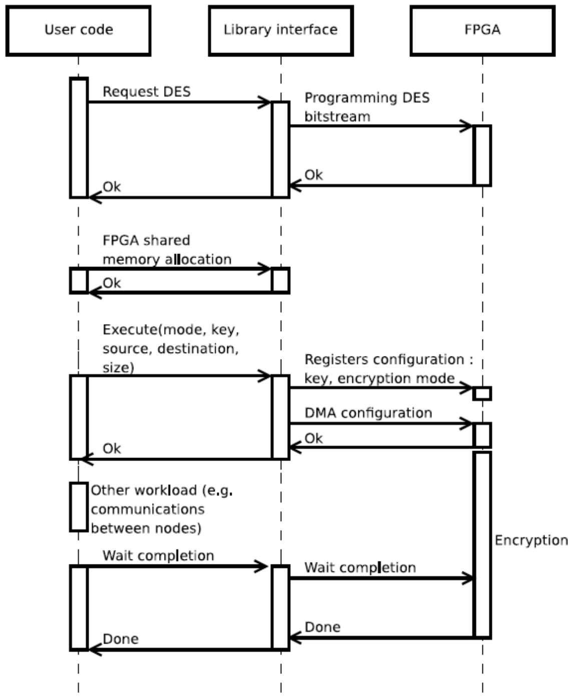
User code
Library interface
FPGA
Request DES
Programming DES bitstream
Ok
Ok
FPGA shared memory allocation
Ok
Execute(mode, key, source, destination, size)
Registers configuration : key, encryption mode
DMA configuration
Ok
Ok
Other workload (e.g. communications between nodes)
Encryption
Wait completion
Wait completion
Done
Done

Fig. 2. The UPC library approach, calling a DES core

3.2.1 The Application Library Approach

In high-performance reconfigurable computing, application libraries that wrap around vendor-specific FPGA libraries are typically made available to application developers. As such, programmers of parallel applications are only required to link their applications with these libraries and make use of traditional function calls that will transparently offload computational tasks to the FPGA(s) of a system. Though this approach itself is simple, it does come with high maintenance cost. The library itself has to directly make use of vendor APIs in order to implement sometimes complex core invocation semantics. Changes to these semantics can sometimes lead to substantial changes in the code actually supporting the specific FPGA core.

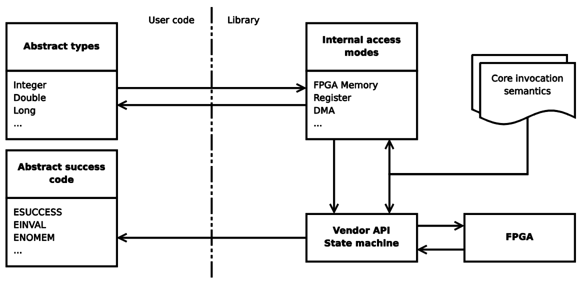

Fig. 3. Library abstract types

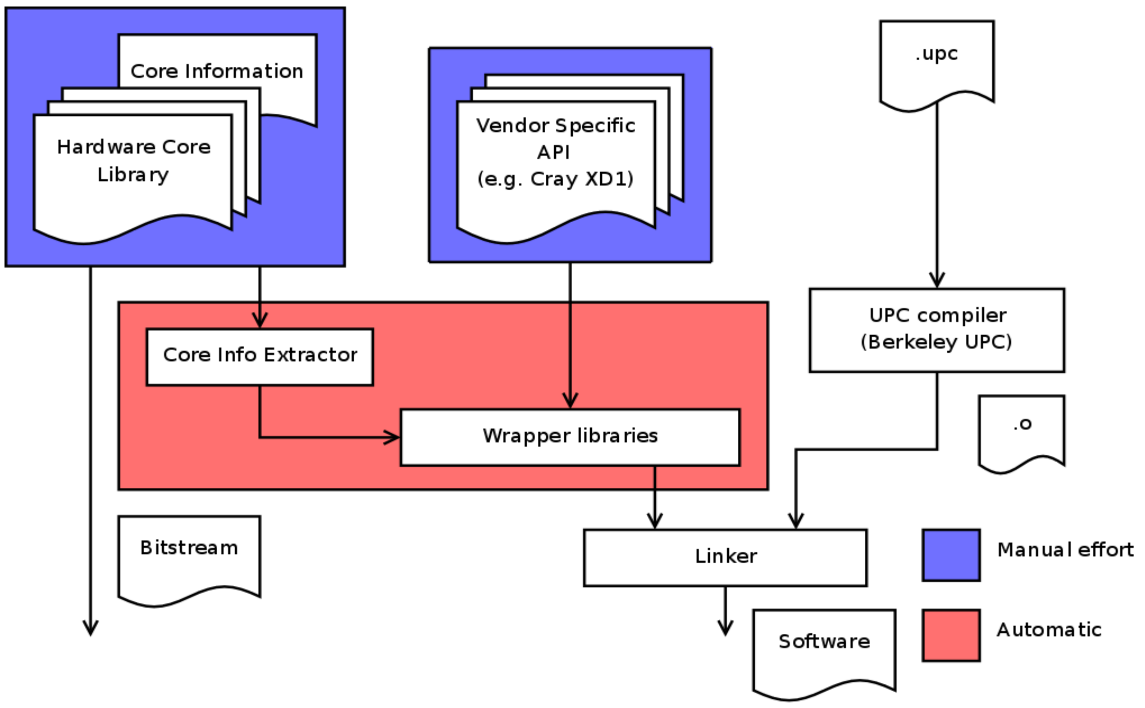

Fig. 4. Proposed core library approach

The work proposed here implements a language-agnostic approach with support for UPC collectiveness requirements. A high-level language is proposed which allows software and hardware engineers to define core invocation semantics without implementing the state machine for the actual invocation. The library is able to map the core invocation definitions to a custom state machine at run-time which is able to properly pass software's arguments to the hardware. The run-time capabilities of this approach are powered by a powerful type translation system. This can be seen in Fig. 3. The interface provided to the end-user is simple and can be seen in Fig. 4.

Fig. 2 shows that when calling a hardware function, there is more involved in terms of configuring the needed hardware on the FPGA. The function call should result in sending input data and configuring the right core if available, and initiating the processing. The API was designed to be asynchronous in nature in order to meet the requirements of overlapping computation and I/O in high performance computing. Upon return of a function call, the output data must be deposited in the correct location then control must be returned to the application in a seamless manner. All this implies some overhead compared to the standard approach. In the worse test case scenario, making many calls to a test core which is doing very little computation, the overhead is around 16% of the total time. Taking into consideration than in most cases, one call will handle an important amount of data, it is a fairly acceptable trade-off.

These wrappers also provide the benefit of being able to properly map compute problems to either hardware or software depending on FPGA availability. This can be decided based on pre-established hardware-software co-scheduling rules that take into consideration configuration time, context switching overhead, and area constraints [6]. In this work, we propose to establish a software framework for allowing UPC applications to call and use hardware functions over the FPGAs in a seamless fashion just as calling any other library. To do so, invocation mechanisms are integrated such that all of this work is hidden from the programmer.

3.2.2 The C-to-Hardware Compilation Approach

In many cases, the user may not have an optimized library that addresses his/her application needs. In these cases, since a source-to-source UPC compiler produces C code, then some of this code can be diverted to a C-to-hardware compiler such as Impulse C.

Also it as been shown the FPGA circuit design can be well captured by a language such as C [7]. In their study, 82% of the 35 circuit chosen could be effectively described using standard C code. If C code can describe circuits, UPC code will be easily able to describe and be an effective means for distributing a whole HPRC application.

Fig. 5 outlines some of the elements of the approach to tackle this problem. Code sections that are amenable to hardware implementation are discovered by taking advantage of some of the UPC constructs such as `upc_forall`, which expresses the presence of data parallelism.

In our implementation, a nesting of two `upc_forall` as in the code:

```
upc_forall(i=0; i<n; i++, i){
    ...
    upc_forall(j=0; j<m; j++, j)
        result[i][j] = function(i, j);
    ...
}
```

allow us to exploit different levels of parallelism. The first `upc_forall` describes the system level parallelism. The second one indicates that data parallelism can still be exploited inside one thread; it will be inferred during the compilation that the second `upc_forall` could be implemented by instantiating cores for `function` in each FPGA. If the *m* core required by the user can fit in the chip, then execution will go as requested. If *m* excess the number of cores than can fit, then striping is used. Fine-grain parallelism will be extracted during the function compilation by the C-to-Hardware compiler, thus allowing us to exploit parallelism at all the possible levels.

4. Experimental Testbed and Results

Our testbed for the initial investigation in this paper used the Cray XD1, the Impulse C C-to-hardware compiler and the Berkeley UPC compiler along with the GWU reconfigurable computing library.

Impulse Accelerated Technologies provides a C-based development kit, Impulse-C, that allows the users to program their applications in a language strongly based on ISO C extended with a library of functions for describing parallel hardware or software processes communicating with streams. The kit provides a compiler that generates HDL code for synthesis from the hardware parts of the application targeting different HPRC platforms.

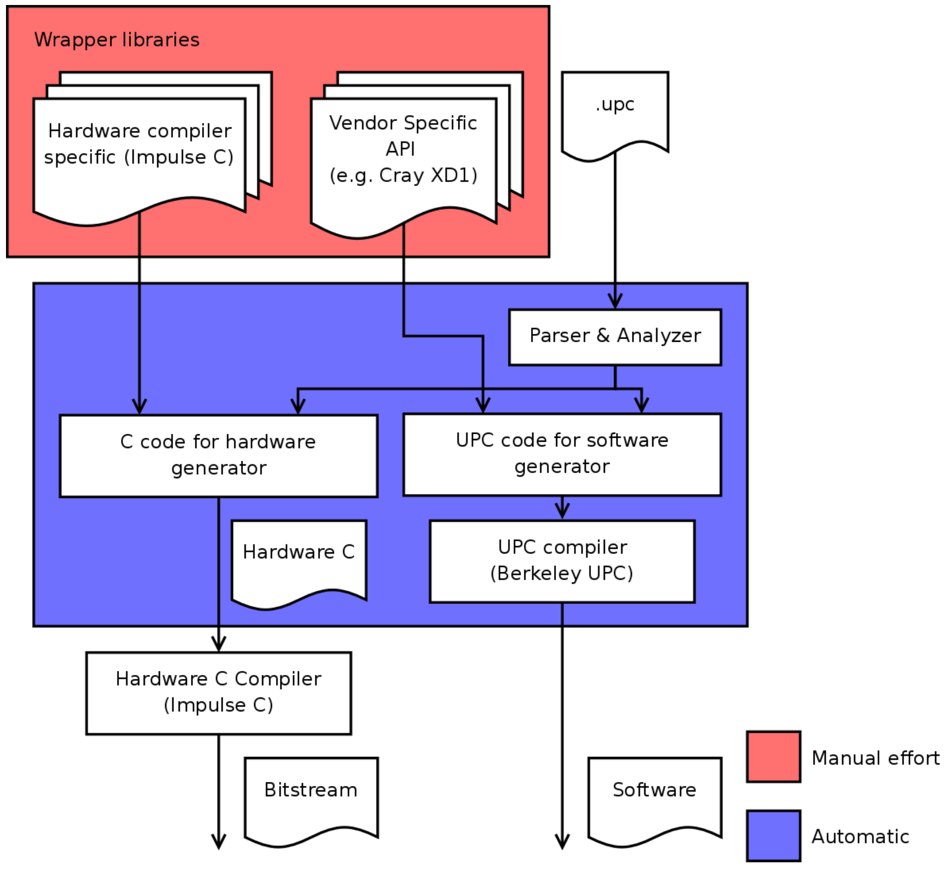
Wrapper libraries
Hardware compiler specific (Impulse C)
Vendor Specific API (e.g. Cray XD1)
.upc
Parser & Analyzer
C code for hardware generator
UPC code for software generator
Hardware C
UPC compiler (Berkeley UPC)
Hardware C Compiler (Impulse C)
Bitstream
Software
Manual effort
Automatic

Fig. 5. Proposed C-to-Hardware compilation approach

In our proof of concept, a UPC parser (See Fig. 5), based on “ANother Tool for Language Recognition” (ANTLR) [8] generate the abstract syntax tree of the UPC code; this tree is analyzed and code section amenable to hardware are identified. The identified functions are extracted and transformed to a valid hardware Impulse C code by adding the streams and processes handling code based on a template. A software replacement of the functions is also generated to handle data transfers between the CPU and the FPGA. Finally, an initialization and bitstream loading code is added to the UPC program.

The general structure of the Cray XD1 system we used is as follows: one chassis houses six compute cards. Each compute card has two AMD Opteron microprocessors at 2.4 GHz connected by AMD's HyperTransport with a bandwidth of 3.2 GB/s, on every compute board, a Xilinx Vertex II FPGA can be provided. All the compute cards are interconnected using a RapidArray switch offering 24 4 GB/s links between the compute nodes and 24 4 GB/s links to other chassis [9].

Table 1 shows the throughput experimental results for the library approach. Speedups over the software implementation running on one Opteron at 2.4GHz are included. One can note that data-intensive applications from image processing, e.g. Discrete Wavelet Transform (DWT) and Sobel edge detection, and from cryptography, e.g. DES encryption, could achieve an order of magnitude improvement over traditional software implementation using only one FPGA. This improvement increased to two orders of magnitude when the full multi-node XD1 was utilized.

Furthermore, compute-intensive applications such DES breaking could achieve as high as two orders of magnitude improvement over traditional software implementations using only one FPGA. This improvement increased to almost four orders of magnitude when the system-level parallelism was exploited.

Table 2 summarizes the results for the C-to-Hardware compilation approach. It is worthy to mention that using these approach improvements over microprocessor implementation were possible for compute-intensive applications. Data-intensive applications showed some slowdown. This is due to the fact that Impulse-C C-to-Hardware compiler supports [10] only the simplest, but yet the least efficient transfer scenario on XD1. This was the case as further improvements for XD1 were abandoned once Cray discontinued it. Therefore, for data-intensive applications, this had a dramatically negative impact on the throughput while for compute-intensive applications such as DES breaking the tool achieved comparable results to HDL.

Also, the Impulse-C compiler was not able to generate valid pipelines on XD1, thus reducing again the expected throughput as the code was left un-pipelined. These experiments showed the fact that the compilation approach is heavily dependent on the support features of the underlying C-to-Hardware compiler for the target platform. In addition, the capabilities of the C-to-Hardware compiler, e.g. the efficiency of parallelism extraction with the least resources, can also impact the achievable results using the compilation approach. This fact can be seen when considering the compute-intensive applications, DES breaking, under both approaches, see Tables 1 and 2, and Fig. 6.

Table 1. Throughput and speedup over an Opteron 2.4GHz for the library approach on XD1

	Opteron 2.4GHz	1 FPGA	6 FPGAs
DWT (MB/s)	83	1,104 - (13.3x)	6,273 (75.5x)
Sobel Edge detection (MB/s)	101	1,095 - (10.9x)	6,216 - (61.1x)
DES encryption (MB/s)	16	1,088 - (68.8x)	6,185 - (391.3x)
DES breaking (M keys/s)	2	1,194 - (600x)	7,112 - (3,574x)

Table 2. Speedup results for the compilation approach on XD1

	1 FPGA 1 engine/FPGA	1 FPGA 6 engines/FPGA	6 FPGAs
DWT	1/13.6	N/A	1/2.4
Sobel Edge detection	1/15.9	N/A	1/2.7
DES breaking	2.3	10.2	61.2

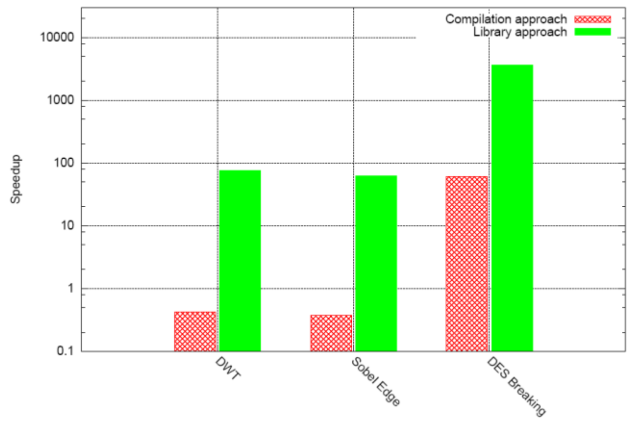

Fig. 6. Experimental results for the proposed approaches

5. Conclusions and Future Work

This paper describes a seamless end-to-end model for productively programming HPRCs by application developers. Initial implementation was conducted as a proof of concepts rather than to provide a fully working tool. These preliminary investigations did shed a lot of light on the promise of this approach.

UPC can provide a productive environment for HPRC programmers. Integrated libraries and SW-HW branching of Source-to-Source UPC compilers are two good methods for the implementation of this approach. These two techniques are not mutually exclusive: they can be integrated into one common interface that gives programmers a powerful environment.

The library approach has shown very good results, yet it can be improved dramatically. Some of the future improvements will include forwarding and chaining of hardware calls to reduce data movements and increase pipeline depths.

The SW-HW branching of Source-to-Source UPC Compilers can also achieve good results in some of the cases. However, the performance is limited by the ability of the chosen C-to-hardware compiler on our testbed. However, many HLL-to-Hardware are already available and show good performance results.

Three main issues with the compilation approach remain as open questions for further investigation: 1. How can we, more automatically, select the part of the code that goes to the hardware compiler, 2. Which extensions to the current UPC specifications are needed to convey sufficient parallelism and locality information that can help automate the process of discovering and compiling part of the application for hardware execution? Are future languages such as X10, Chapel or Sequoia [11] capable of exposing those opportunities more effectively? and 3. To what extent the productivity can be improved by the UPC language on reconfigurable systems.

References

[1] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors", *ACM/SGIDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA*,2004.

[2] T. El-Ghazawi, D. Buell, K. Gaj, and V. Kindertenko, "Reconfigurable Supercomputing, Tutorial Notes", *IEEE/ACM Supercomputing Conference, Tampa Florida*, 2006.

[3] S. Singh, "Integrating FPGAs in high-performance computing: Programming models for parallel systems - the programmer's perspective", *Fifteenth ACM/SGIDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA*, 2007.

[4] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. B. Newby, "Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study", *III Southern Conference on Programmable Logic (SPL 2007), Mar del Plata, Argentina*, 2007.

[5] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, "*UPC: Distributed Shared Memory Programming",* May 2005.

[6] P. Saha and T. El-Ghazawi, "A methodology for automating co-scheduling for reconfigurable computing systems", *ACM/IEEE Conference on Formal Methods and Models for Co-design, Nice, France*, May 2007.

[7] S. Sirowy, G. Stitt and F. Vahidi, "C is for Circuits: Capturing FPGA Circuits as Sequential Code for Portability", *Sixteenth ACM/SGIDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA*, February 2008.

[8] "Another Tool for Language Recognition (www.antlr.org)"

[9] Cray Inc., "Cray XD1TM FPGA development (S-6400-14)," 2006.

[10] Impulse C, "Impulse accelerated technologies (www.impulsec.com)."

[11] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy", *ACM/IEEE Conference on Supercomputing SC*, 2006.

[12] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "Guest editors introduction-high-performance reconfigurable computing," *IEEE Computer*, vol. 40, no. 3, 2007.