# The HARWEST Compiling Environment: Accessing the FPGA World through ANSI-C Programs

**Alessandro Marongiu** *and* **Paolo Palazzari**
*Ylichron Srl and ENEA*

**ABSTRACT:** *FPGA computing is always thought as a media to dramatically improve computational performances. The real obstacle to its widespread diffusion is primarily due to the lack of compiling tools which allow to use common specification languages (like the ANSI C); on the contrary, FPGAs have to be programmed either through very low level HDL languages or through some not standard languages which are dialects derived from the C but which are very far from the standard C-language. In order to overcome previous drawbacks, Ylichron developed a compiling chain, the HARWEST Compiling Environment (HCE), which allows to specify algorithms to be mapped onto FPGAs through standard C programs: as a consequence, no special skills are required to access the power of FPGA computing and no special efforts have to be spent to learn proprietary languages. The HCE Design Flow and some performance figures are presented in the paper.*

**KEYWORDS:** FPGA, High Level Synthesis, DRC blades, ANSI C

## 1. Introduction

FPGA computing is always thought as a media to dramatically improve computational performances. The real obstacle to its widespread diffusion is primarily due to the lack of compiling tools which allow to use common specification languages (like the ANSI C); on the contrary, FPGAs have to be programmed either through very low level HDL languages (such as VHDL or Verilog) or through some not standard languages which are dialects derived from the C but which are very far from the standard C-language: in both the cases, the effort to develop an application cannot be kept and reused to port the same application onto different compiling environments. In order to overcome previous drawbacks, Ylichron developed a compiling chain, the HARWEST Compiling Environment (HCE), which allows to specify algorithms to be mapped onto FPGAs as standard C programs: as a consequence, no special skills in electronic system design are required to access the power of FPGA computing and no special efforts have to be spent to learn proprietary languages.

HCE is a set of compilation tools which transform an ANSI C program into an equivalent, optimized VHDL ready to be compiled and run onto a prefixed target board: the DRC FPGA blades, used by the Cray XT5h systems, are among the supported targets.

The choice of the ANSI C is aimed at allowing the adoption of FPGA computing devices to a widespread and heterogeneous community of users: as it is a standard de facto language, its adoption allows many technicians and scientists from many fields (biology, chemistry, physics, …) to port the kernel of their C applications onto FPGA devices.

The main characteristic of the Ylichron approach is that the programs given as input to the HCE flow can be previously refined and debugged within a well known C compiling environment (the Microsoft VisualStudio) and, once fixed, they are passed to the HCE flow without changing or adding any line of code: this fact ensures that no use of not standard functionalities related to the hardware synthesis flow (exploiting the application parallelism, guiding the scheduling process, …) is required. Thanks to the effort spent in ensuring such compliance to the standard C, no special skills (electronic design, system architectures) are required to use the HCE flow.

In order to allow the implementation of area efficient designs, the set of the standard ANSI C data types has been extended to include also two of the SystemC data types – the sc_bv and the sc_fixed, being SystemC [1] a set of standardized C++ classes defined to efficiently model and simulate digital systems. For such a reason, the debug and test phase requires a C++ compiler.

## 2. HCE Design Flow

The HARWEST Compiling Environment (HCE) is one of the first outcomes of the HARWEST research project, funded by the Italian Ministry for University and Research, aimed at creating a fully automated HW/SW co-design environment. The HCE flow is reported in Fig. 1.

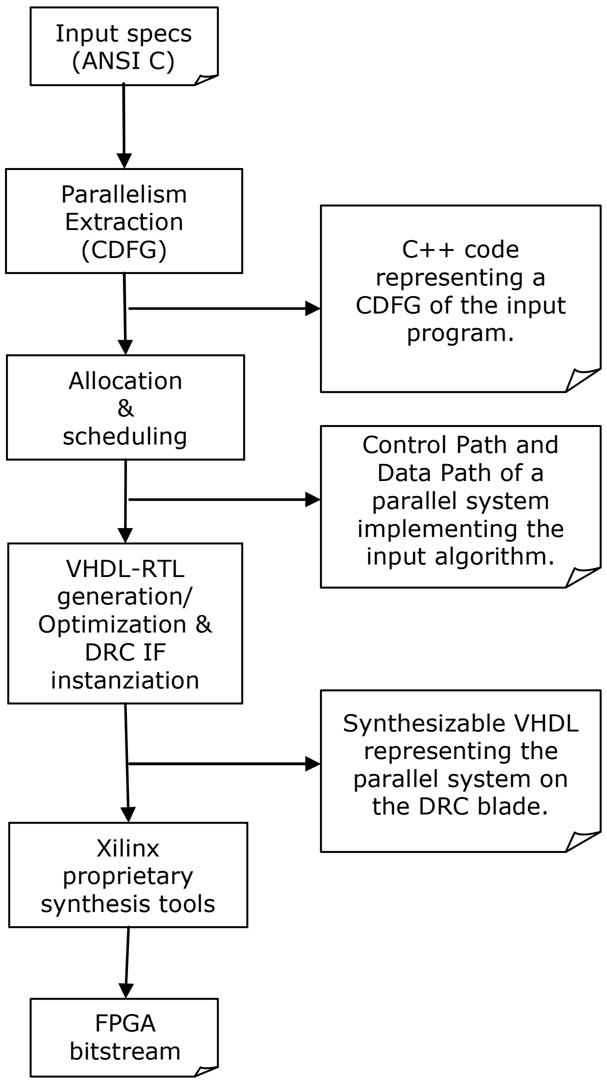

Input specs (ANSI C)
Parallelism Extraction (CDFG)
C++ code representing a CDFG of the input program.
Allocation & scheduling
Control Path and Data Path of a parallel system implementing the input algorithm.
VHDL-RTL generation/ Optimization & DRC IF instanziation
Synthesizable VHDL representing the parallel system on the DRC blade.
Xilinx proprietary synthesis tools
FPGA bitstream

Figure 1: HCE Flow

The system specifics are given by means of an ANSI C program: actually we adopt both a subset of the ANSI C, not allowing the use of pointers and of recursive functions, and a small extension toward C++, supporting the two SystemC sc_bv and sc_fixed data types. As these two data types are implemented as C++ objects, with their methods and dedicated operations, a C++ compiler is needed to run the HCE C input programs. This, together with the & operator used when declaring a function parameter, are the only C++ features accepted by the HCE.

Starting at a very high level of abstraction (C language), the correctness of the specifics is checked by running and debugging, on a conventional system, the C program (of course, we do not afford the task of proofing the correctness of a C program: we consider a program to be correct once it works correctly on some given sets of input data). Once we are satisfied with the program behavior, i.e. after the debugging and testing phase, the input specifics are translated into the Control and Data Flow Graph (CDFG) model of computation. Given a set of resources (number of basic building block modules), the CDFG is mapped into a Data Path and a Control Finite State Machine (FSM) which enforces the behavior expressed by the starting C program. Such an architecture, represented by the Data Path and the Control FSM, is further optimized (some redundant logic is removed and connections are implemented and optimized) and translated into an equivalent synthesizable VHDL code. Finally, the VHDL program is processed through the standard proprietary design tools (translation and map into the target technology, place and route) to produce the configuration bit stream.

It is worth to be underlined that the debugging phase is executed only at very high level of abstraction (on the C program), while all the other steps are fully automated and result to be correct by construction: this fact ensures that, once we have a working C program, the final architecture will show the same program behavior.

***CDFG Extraction from C programs***

As reported in Fig. 1, the first step of the design flow transforms the C program into an equivalent algorithm expressed by means of the, intrinsically parallel, CDFG computational model.

When designing HCE, we decided to use a CDFG model with blocking semantics (i.e. a computing node does not terminate until all its outputs have been read by the consumer nodes) because it does not require the insertion of buffers along the communication edges. In [2] we individuated a set of rules which transform C programs into CDFGs and ensure that deadlocks never arise.

Once the CDFG of the original program has been determined, it is analyzed to implement some basic optimizations aimed at reducing the complexity of the graph (constant folding/propagation, common sub-expression elimination, invariant code motion, …).

### *Allocating and scheduling CDFGs*

Resorting to the design flow in Fig. 1, after having derived the CDFG representation of the original algorithm, we are faced with the problem of allocating and scheduling such computations onto a set of predefined computing resources which represent a subset of the input constraints (further constraints could be involved by the need of a real-time behavior or by throughput requirements).

In order to perform the allocation and scheduling operations, within the HCE we created a library of computing modules which are the building blocks to be used to set-up the final parallel architecture. Each module is constituted by

- a collection of static information regarding
  - the area requirements (basic blocks needed for the implementation of the module onto a certain technology - LUT, memory modules, DSP blocks, ...),
  - the module behavior (combinational, pipelined, multicycle, asynchronous),
  - the response time (propagation delay, setup time, latency, ...)
- a collection of files (EDIF format) to implement the module onto different target technologies (for instance, FPGAs from different vendors).

The HCE library of pre-designed modules contains:

- ✓ Pipelined (*, +) and multi-cycle (/) floating point operators
- ✓ All the family of operators to support char, int and fixed point data types
- ✓ The basic math functions (sinf, cosf, tanf, sqrtf, logf, expf)
- ✓ The rand() function
- ✓ Modules to manage pipelined memory banks with any address and word length; address length is constrained to be <= 64 and word length has to be a power of 2.
- ✓ Modules to support the management of the program control flow.

The library of HCE modules can be extended by inserting user defined modules.

Allocating and scheduling a computation requires a time instant and a computing resource (i.e. a module) to be assigned to each operation of the computation. In order to do this, the HCE flow implements a list scheduling heuristic, derived from the algorithm presented in [3]. It is worth to be underlined that the algorithm in [3] has been deeply modified to allow the sharing of the same HW module among different SW nodes.

When the original C program is structured with different functions, the program can be flattened through the inline expansion of the functions, thus generating a unique CDFG and a unique FSM. In order to reduce the complexity of the scheduling operation, as a simplifying option, each function - starting from the deepest ones - can be translated onto a different CDFG. The HCE flow initially schedules the CDFGs corresponding to code which has not inner functions, generating the corresponding FSMs; successively such FSMs are scheduled as asynchronous modules when the CDFG of the calling function is scheduled; these asynchronous modules are synchronized with the FSM of the calling CDFG through a basic start/stop protocol.

Before performing the actual allocation and scheduling phase, HCE allows the user to define the structure of the target architecture. In fact it is possible to specify, for each C function, the multiplicity of the modules that will be used to implement that function: in such a way the user may control at which level of granularity the parallelism is exploited.

As illustrative example of previous point, let us consider a portion of code to implement the Cannon algorithm [4] to multiply two matrices:

```
void BlockMatrixMAC(float BA[N][BS],
     float BB[N][BS],float BC[N][BS],int step)
{
 int i,j,k, base_k, BN;
 for (BN = 0; BN < NP; BN++){
  base_k = ((BN+step)*BS)%N;
    for (i=BN*BS;i<(BN+1)*BS;i++){
      for (j=0;j<BS;j++)
         for (k=0;k<BS;k++)
            BC[i][j]+=BA[i][k]*BB[base_k+ k][j];
    }
 }
}

void rotateMatrixBA(float BA[NP][N][BS]
                    /*#HWST split 1 NP */)
{...}

void CannonMM(
        float  BA[NP][N][BS]  /*#HWST  split  1
NP*/,
       float BB[NP][N][BS] /*#HWST split 1 NP*/,
       float BC[NP][N][BS] /*#HWST split 1 NP*/)
{
    for (int s = 0; s < NP; s++){
        int i;
        /*#HWST split */
        for (i=0; i<NP; i++)
            BlockMatrixMAC(BA[i],BB[i],BC[i],
s);
       rotateMatrixBA(BA);
    }
}
```

Figure 2: The Cannon algorithm the for matrix product

As mentioned before, the inner `BlockMatrixMAC` function will be initially scheduled. Before entering the scheduling step, the user is asked about the number of Floating Point multipliers and adders that have to be used: wanting not to exploit parallelism at this level, 1 FP add and 1 FP mul can be selected. Once that `BlockMatrixMAC` has been scheduled, it becomes an asynchronous module which is used by the `CannonMM` function. Again, before scheduling the `CannonMM` function, user is asked about the multiplicity of the `BlockMatrixMAC` module: looking at the algorithm, where `BlockMatrixMAC` is called NP times without any dependence, it is natural to ask for NP modules of the previously synthesized `BlockMatrixMAC` module: in such a case NP sequential modules `BlockMatrixMAC` will run in parallel.

***Post optimization and VHDL generation***

Once obtained the allocation and the scheduling for the CDFG, the final architecture has still to be defined in the part concerning the interconnection/control network. In fact, during the scheduling step, all the details about the connections and the structures to support HW sharing are neglected. In this final phase all the connections are implemented, introducing and defining the multiplexing logic necessary to implement the communication lines; furthermore, a set of registers and multiplexers on the control lines are inserted, in order to allow a coherent management of iterations in presence of HW sharing.

As basic optimization, the control lines driven by the same logic conditions are joined together. In order to meet the fan-out constraints imposed by the target technology, a buffer tree is eventually inserted.

Once the original C program has been translated into a synthesizable parallel architecture, the proper interface with the DRC environment is added.

## 3. HCE IDE

As already discussed in the introduction, when designing HCE we had in mind two main usability objectives:

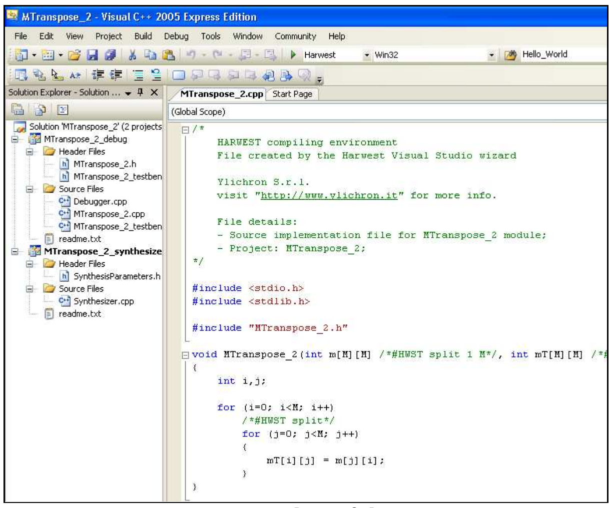
1- use only C standard statements, avoiding the introduction of constructs to directly manage parallelism or instruct the scheduler in a way which is not compliant with the ANSI C;
2- minimize the effort, for a not specialist, to use HCE.

In order to fulfill previous requirements we decided to embed HCE within a very well known C compiler. Our choice fall on the VisualStudio from the Microsoft for

a. its strict compliance with the ANSI C and its nearly complete coverage of the ISO C++ standard (required by the HCE engine)
b. its debug capabilities and its widespread diffusion
c. its possibility to be customized to host and execute external programs
d. the availability of the free Express edition.

Consequently, HCE has been completely embedded within the VisualStudio environment. Figure 3 reports a screenshot of a typical HCE program.

Thanks to the deep integration with a C compiler, HCE programs are written and debugged using the native compiler: this fact will ensure that the programs accepted by HCE adhere to the ANSI C / ISO C++ standards, because they have to be compiled by the VisualStudio C++ compiler.

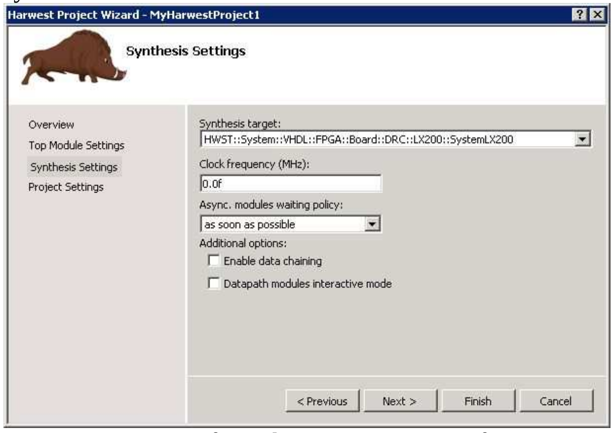


```
/*
    HARWEST compiling environment
    File created by the Harwest Visual Studio wizard

    Ylichron S.r.l.
    visit "http://www.ylichron.it" for more info.

    File details:
    - Source implementation file for MTranspose_2 module;
    - Project: MTranspose_2;
*/

#include <stdio.h>
#include <stdlib.h>

#include "MTranspose_2.h"

void MTranspose_2(int m[M][M] /*#HWST split 1 M*/, int mT[M][M] /*#
{
    int i,j;

    for (i=0; i<M; i++)
        /*#HWST split*/
        for (j=0; j<M; j++)
        {
            mT[i][j] = m[j][i];
        }
}
```

Figure 3: Screenshot of the HCE IDE

Remaining in the same environment, switching from the (VisualStudio standard) Debug modality to the (HCE proprietary) HARWEST modality, the same C code is passed through the whole HCE flow sketched in Figure 1.

The execution of the HCE flow generates, other than the final VHDL program corresponding to the original C outermost function, a set of report files that allow to have a deeper comprehension of what HCE did: in both graphical and XML formats the user finds the structure of the CDFGs generated, of the FSMs produced and of the system architecture.



Synthesis target: HWST::System::VHDL::FPGA::Board::DRC::LX200::SystemLX200

Clock frequency (MHz): 0.0f

Async. modules waiting policy: as soon as possible

Additional options:

☐ Enable data chaining

☐ Datapath modules interactive mode

Figure 4: A step from the HCE project configuration wizard

When creating a new HCE project, the user is assisted by the HCE Project wizard which allows:

1- to define the signature of the function;
2- to select the HW target which will be addressed to generate the final architecture (FPGA family – i.e. Virtex2, Virtex4, Virtex5, board type – i.e. DRC LX200, XD1, …);
3- to decide whether allowing or not data chaining in the scheduling phase (i.e. if two dependent operations could be scheduled in the same state);

4- to select if being asked to define the structure of the parallel architecture (multiplicity of modules and of registers) or letting HCE to generate a simple parallel architecture;
5- to add support to the SystemC data types;
6- to ask for XML/postscript graphical reports (CDFG, FSM, Architecture)
7- to automatically generate a debug project which contains the skeleton of a testbench application aimed at stimulating the inputs of the function to be synthesized by HCE.

## 4. DRC Co-processing systems

DRC provides a co-processor system which fits on a free Opteron socket. Due to the tight interconnection to the host buses it provides high communication bandwidth between the host and the co-processor system. At the moment of writing the system is provided in two versions: RPU110-L60 and DRC RPU110-L200. In the following we refer to the last version which is able to provide more performances.

DRC co-processor system is equipped with:

- a Xilinx Virtex4 FPGA (XC4VLX200). See Xilinx documentation for details on such a FPGA;
- HyperTransport (HT) interconnection;
- Up to 2GB of RPU DRAM with two independent physical buses each with 3.2GB/s peak bandwidth;
- 128 MB of RPU Low Latency RAM (LLRAM) with two independent physical buses each with 800Mb/s peak bandwidth;
- 256 Mbits of not volatile Flash RAM

DRC co-processor system can be optionally equipped with up to 4GB of motherboard DRAM with one physical bus with 6.4GB/s peak bandwidth.

DRC provides all the cores needed to drive the resources outside the FPGA device:

- HT bus driver core;
- DRAM and LLRAM driver cores

## 5. DRC co-processors and the HCE flow

Writing an application for the DRC co-processor system requires skills in electronic system design. In fact the first step is to design and implement the core function to be executed on the hardware. Once designed, the hardware block must be interfaced with the external resources of the FPGA, i.e. with the HT, the DRAM and the LLRAM. Finally the host software application must be re-designed to include the calls to routines which allow to transfer input data on the storage resources of the co-processor system, to activate the hardware core function and to get the computation results.

HCE allows the automation of the previous steps.

First of all the core function to be implemented in hardware must be selected. Usually such a function is selected among the more computationally intensive functions which require the less data transfers.

Once the core function (for example myFunc()) has been selected, it is synthesized through the HCE selecting the DRC co-processor system as target technology. HCE produces a hardware block (described in VHDL/Verilog language) which reflects the function signature. In fact, the hardware block has a set of I/O ports generated according to the following rules:

- if the function returns a value (i.e. the function is not a void function) the hardware block has an output port named out_myFunc_var;
- for each scalar input parameter passed “by value” in the function signature, the hardware block has an input port named in_parameter_name;
- for each scalar input/output parameter passed “by reference” (&) in the function signature, the hardware block has:
    - o an input port named in_parameter_name if the parameter is only read in the function body;
    - o an output port named out_parameter_name if the parameter is only written in the function body;
    - o both the previous input and output port if the parameter is read and written in the function body;
- for each array parameter in the function signature, the hardware block has a set of port which allows to drive an external memory. The actual protocol implemented on these ports depends on the target technology. Referring to the DRC co-processor system, at the moment of writing HCE supports only the Block RAM of the FPGA device. Support for the LLRAM and DRAM will be added in the next release of HCE;
- finally the set of signals implementing a simple handshake protocol to synchronize with other devices.

HCE synthesis process automatically generates an interface which allows the hardware block to interact with the resources outside the DRC co-processor system.

Analyzing the signature of the function to be synthesized, HCE maps the I/O parameters into the addressing space of the DRC co-processor system and to each parameter associates a storage resource: for each scalar parameter a register is generated and for each array parameter a memory is generated.

Then a decoding block is designed and instantiated to map the DRC co-processor system addressing space to the FPGA storage resources. Such a decoding block is designed in order to allow the direct interconnection with the HT bus.

The interface generated by HCE also contains an application register which allows the synchronization with the host system.

Finally a simple Finite State Machine is generated to manage the handshake protocol of the hardware block implementing the user function synthesized by HCE.

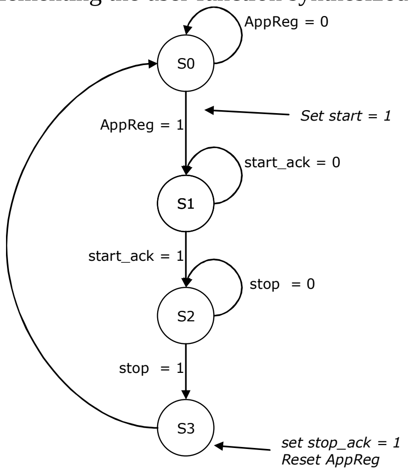


Figure 5: Handshake FSM

The FSM has four states. In S0 the FSM wait for the application register. As soon the application register value is set to “1”, the start signal is asserted and the FSM goes in the S1 state waiting for the start_ack signal from the HCE block. When the start_ack signal as been asserted, the FSM goes in the S2 state and waits for the stop signal to be asserted. When the hardware block finishes the computations it asserts the stop signal. Then the FSM goes in the S3 state, asserts stop_ack and reset the application register. Then the FSM goes to the S0 state waiting for a new computation cycle.

On the software side, HCE generates a wrapper function which allows to use blindly the FPGA hardware. The wrapper function performs the following operations:

- checks the DRC blade to be correctly configured and available to the current user;
- transfers the input data via HT bus through the DRC API functions;
- writes “1” into the application register to trigger the execution of the hardware implemented function;
- waits for the end of computation and gets the output data;

Three wrapper functions are provided:

- a blocking wrapper function which stops the execution of the application program running on the host until the end of the hardware computation;
- two non blocking wrapper functions: the first allows to start the hardware computation and, while the FPGA hardware is running, to perform in parallel some computation to be run on the host side; the second waits for the computation end.

The usage of the wrapper function is trivial. Suppose to have the following fragment of code which uses the function myFunc();

```
#include <myFunc.h>
…
…
myFunc();     //call to myFunc()
…
…
```

Once myFunc() has been synthesized through HCE, we obtain:

- the VHDL files which represent the hardware implementation of myFunc() on the DRC blade. Such VHDL files have to be synthesized with the Xilinx synthesis tool chain in order to obtain a configuration bitstream file. HCE produces a batch file to automatically perform the low level synthesis step with default Xilinx synthesis options. If different synthesis options are needed the low level synthesis step has to be performed manually;
- the C++ file which contains the wrapper to call the hardware implementation of myFunc()

In order to use the hardware implementation of myFunc(), the bitstream must be downloaded onto the DRC blade and the original program must be modified so that it calls the wrapper function. The original code becomes:

```
#include <myFunc-hwst-wrapper.h>
…
…
//call to the wrapper of myFunc() which
//activates the hardware implementation
// and waits for the output data
myFunc();
…
…
```

The previous example uses the blocking version of the wrapper function. The non-blocking version is the following:

```
#include < myFunc-hwst-wrapper.h >
…
…
//call to the wrapper of myFunc() which
//activates the hardware implementation
//without waiting for the output data
myFunc_start();
…
… code to be executed on the host side
…
```

```
//call to the wrapper of myFunc() which
//waits for the output data
myFunc_end();
```

Obviously the code executed between the myFunc_start and myFunc_end calls to the wrapper functions must be executed on the host side and must be not dependent on the results of myFunc() computation. It is up to the user to satisfy such requirements.

## 6. Performances

In order to quantify the performances achievable through HCE, in this paragraph we report the synthesis results achieved in several test cases.

In order to be meaningful, we consider tests to measure different aspects of the global system. We implemented a test (test 1) to measure the Read/Write bandwidth from/to the DRC accelerator, a test (test 2) to measure the effective utilization of the internal memory bandwidth, a third test (test 3) to show the performances in the case of Boolean computation, a test (test 4) to measure performances when processing DNA sequences, a test (test 5) to show how HCE behaves with fixed point computations and, finally a test (test 6) to analyze the case of floating point computations. In all the tests, whenever the number of employed slices is reported, the number of slices used by the DRC interface has to be added (~12000, 13% of the total slices available in a Virtex4 LX200 FPGA)

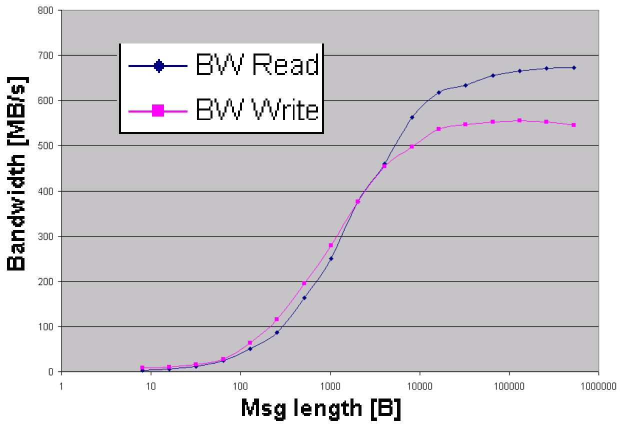


Figure 6: Host – DRC I/O Bandwidth

***Test 1: measuring DMA bandwidth***

In order to measure DMA bandwidth from an Opteron Linux System, we wrote a simple application which required the transfer of a message from the host to the FPGA (DMA Write) and the corresponding read back from the FPGA to the host memory (DMA Read). The test was performed 10 times for different message length.

In Figure 6 we report, for each message length, the average bandwidth obtained averaging the results of the 10 runs. The bandwidth saturates at 670 MB/s in the read case and at 550 MB/s in the write case. The time necessary to write (read) the shortest message (8 bytes) is equal to 0.9 µs (2.5 µs ).

***Test 2: measuring the exploitation of internal bandwidth***

One of the main advantages of FPGAs rely on their extremely high memory bandwidth. We wrote a simple test to measure the ability of HCE to exploit such a huge bandwidth. The test performs the transposition of a 2D square matrix, copying the original matrix m[M][M] into the transposed matrix mT[M][M]. The C code is the following:

```
#define M 128
void MTransp(float m[M][M] /*#HWST split 1 M*/,
           float mT[M][M] /*#HWST split 2 M*/)
{
      int i,j;
      /*#HWST unroll 16 */
      for (i=0; i<M; i++)
            /*#HWST split*/
            for (j=0; j<M; j++)
            {
                  mT[i][j] = m[j][i];
            }
}
```

In the code the `split` keywords in the comments are special keys used to inform HCE that matrix m[M][M] is divided into M row vectors of size M and mT[M][M] is divided into M column vectors of size M. Thanks to this matrix organization, each vector will be mapped on a different FPGA block memory bank (the LX200 FPGA of the Xilinx Virtex4 family has 336 block RAM modules, each one with size of 18Kbits) and HCE will be able to parallelize the accesses to these memory banks.

After the synthesis of this simple code, HCE produces the VHDL files corresponding to an architecture which employs 136 clock cycles to transpose the matrix. As the clock frequency is equal to 100 MHz, MxMx4 bytes are read and written in 1.36 µs, corresponding to an internal bandwidth $BW_{Read}=BW_{Write}$=45 GB/s. The number of slices used by such an architecture is 6601 (7% of the total slices available in a Virtex4 LX200 FPGA). The architecture uses 256 Block RAM modules.

***Test 3: computing the transitive closure of a graph***

Let us consider a directed graph G with N nodes, represented through the incidence matrix a[N][N] ($a_{ij}$=1 when node $n_i$ is connected to node $n_j$, $a_{ij}$=0 otherwise).

The transitive closure of G is a direct graph which has an edge between two any nodes if the two nodes are connected by a path in G. The algorithm to compute the incidence matrix of the transitive closure of G is the following:

```
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
         a_ij=a_ij | (a_ik & a_kj);
```

We implemented the algorithm to compute the transitive closure in the case of graph size N = 2048. The architecture produced by HCE computes the transitive closure in $T_{EXE}$ = 250 ms, corresponding to $2*N*N*N/T_{EXE}$ = 68 x $10^9$ op/s, being one op an elementary 1 bit boolean operation. The number of slices used by the architecture is 3695 (4% of the slices available in a V4LX200 FPGA). The architecture requires 256 Block RAM modules.

***Test 4: the Smith Watermann algorithm***

Bioinformatics, and DNA sequencing in particular, has always been seen as a good candidate to adopt FPGAs. To test HCE in such environment, we decided to implement the computation of the scoring matrix of the Smith Waterman algorithm [5]. This algorithm uses a dynamic programming approach to find the best alignment (with insertions, deletions and mismatches) between a DNA sequence of size m (pattern) and another sequence of size n (text). The algorithm returns the position in the text where the pattern is contained and how the pattern has to be stretched to best match the text. In our coding, we fixed the pattern size m=255 and the text size n=1024. After the HCE flow we obtained an architecture which requires ~4(n+m) clock cycles to run, being $f_{ck}$ = 100 MHz. The number of slices used by the architecture is 20897 (23% of the slices available in a V4LX200 FPGA).

***Test 5: implementing a FIR filter***

Digital signal processing is one of the fields more involved in embedded processing. To test how HCE behaves on DSP algorithms, we considered the implementation of a FIR filter – a typical, paradigmatic DSP application. We coded a filter with 128 taps and fed it with a signal x[N] constituted by N = 1024 samples. Both the filter taps and the samples (input and output) has been represented in fixed point, using the sc_fixed<16,8> SystemC data type: this data type is composed by 16 bits, 8 used as integer part and 8 as fractional part. The HCE flow produced an architecture which ran at 60 MHz and employed 17 μs to produce the output vector: such timing corresponds to a sustained computation rate of 14 Gop/s, being one op a 16 bit fixed point operation. The number of slices used by the architecture is 30489 (34% of the slices available in a V4LX200 FPGA); furthermore, the architecture uses all the 96 available DSP blocks.

***Test 6: the Cannon algorithm for the matrix product***

To test the flexibility of HCE in the case of algorithms requiring the floating point representation, we implemented the Cannon Algorithm [4] to perform the matrix product C = AxB, being A, B and C NxN matrices (in the example N = 128). A sketch of the code is shown in Figure 2, being the block size dimension BS=8 and the number of `BlockMatrixMAC` modules NP=16. The HCE flow produced an architecture which ran at 80 MHz and employed 3.7 ms to produce the output vector: such timing corresponds to a sustained computation rate of 1.1 GFlop/s. The number of slices used by the architecture is 40799 (45% of the slices available in a Virtex4 LX200 FPGA); furthermore, the architecture uses all the 96 available DSP blocks and 98 memory blocks (29% of the total memory blocks available in a V4LX200 FPGA).

## 7. Conclusions

The HARWEST Compiling Environment (HCE), an high level synthesis tool developed by Ylichron Srl in the framework of the HARWEST industrial research project, has been presented. HCE is a collection of compiling tools which automatically translate, in a nearly optimal way, an ANSI C program into a corresponding, synthesizable parallel architecture. In order to make really easy the adoption of FPGA technology to users with experience in software development (and without experience in electronic system design), the HCE flow has been completely embedded within the Microsoft VisualStudio compile framework.

HCE supports, among its targets, the DRC co-processing boards and allows the seamless integration of programs running in the host processor node with the kernel allotted to the co-processing FPGA node: HCE translates the kernel into a synthesizable parallel architecture, adds the interface to the DRC FPGA environment, automatically generates the bitstream to configure the FPGA and generates also the C wrapper functions to invoke, together with the necessary DMAs to pass data and to read the results, the accelerated kernel from a C program running on the host processor.

Thanks to the HCE approach, the porting of computationally demanding applications onto the DRC FPGA co-processing boards has become a task which can be accomplished in a completely automated way.

## References

[1] www.systemc.org

[2] G. Brusco: “Generation of CDFGs from C programs and their scheduling”. Master Thesis in Electronic Engineering, University “La Sapienza” (Rome), 2004 (in Italian).

[3] G. Lakshminarayana et al.: “Wavesched: a novel scheduling technique for control flow intensive designs”. IEEE Trans on CAD, 18, 5, 1999.

[4] A. Grama, A. Gupta, G. Karypis, V. Kumar: “Introduction to parallel computing”, Addison Wesley, 2003 – Paragraph 8.2.2

[5] TF Smith, MS Waterman: “Identification of Common Molecular Subsequences”. Journal of Molecular Biology 147, 1981.