# Evaluation of the capabilities of Handel-C considering the implementation of a classification algorithm as example

Uwe Düffert, Frank Winkler
Institut für Informatik, Humboldt-Universität zu Berlin
<dueffert,fwinkler@informatik.hu-berlin.de>

July 16, 2003

**Abstract**

Field-Programmable Gate Arrays (FPGAs) are widely used to produce very small runs of user defined circuits that can efficiently process occurring data in parallel, because this is a relatively fast and inexpensive technology.

The usage of Handel-C for programming FPGAs is supposed to shorten the development time further by using common programming concepts and algorithms leading to easier portability of existing software solutions to hardware and avoiding the need to learn a classic hardware description language.

We will analyze a concrete example of classifying multi spectral image data in this paper to find out to what extent Handle-C satisfies these expectations.

## 1 Introduction

A Field-Programmable Gate Array (FPGA) is a chip consisting of an array of logic blocks, surrounded by a ring of programmable input/output blocks, connected together via programmable interconnect. A typical FPGA contains thousands of logic blocks and an even greater number of flip-flops. The logic is placed and routed by sophisticated software.

Usually an FPGA design is specified in a hardware description language such as Verilog, VHDL, or ABEL. A different approach is Celoxicas language Handel-C which is based on ANSI-C with extensions required for hardware development (*http://www.celoxica.com/tech/handel-c/*). It allows hardware designers to take advantage of software design techniques, abstracts from physical details and aims for faster training of developers and easier porting of existing C algorithms.

To evaluate the capabilities of Handel-C an example of remote sensing application was chosen. In 2001 the German Aerospace Center (DLR, *http://www.dlr.de*) started a satellite called BIRD (BiSpectral InfraRed Detection) that orbits the earth at an altitude of 600 km. Its main purpose is testing new infrared sensors as well as earth monitoring with these sensors. One of the monitoring tasks is automatic forest fire detection.

The current satellite uses a neural network processor for classifying the sensor data. This is quite inflexible, therefore the next version will probably have an FPGA on board. So it is interesting to test whether Handel-C is a good choice for programming FPGAs for such tasks or not.

## 2 The Example Application - An Euclidean Distance Classifier

The aim of the research was evaluating a working example implementation - not just a piece of Handel-C source and a resulting net list with theoretical resource usage and timing constraints.
The DLR has lots of experience in forest fire detection from satellites, e. g. [4], [1] or [2].
The test data from BIRD were provided by the DLR[1]. They consisted of 3 different infra-red channels and were already converted to 8 bits per channel and preprocessed in a way that the same pixel in different channels belongs to the same area observed on earth. This can be seen as sensor calibration.

### 2.1 Architecture

In the real world there is a satellite and a ground station, in this example there is a PCI-FPGA-card and a PC. Image data, raw as well as classified ones, will be transfered line by line (512 pixels per line).
Furthermore the ground station can request the FPGA to operate in a certain mode, e. g. calculate everything but only send a certain class. The ground station can send a different set of sample vectors to the FPGA too. So it is possible to reduce the amount of data to be sent and change everything that is variable at run time.

### 2.2 Classification

An Euclidean distance classifier was implemented, because this is common practice in such image analysis. A class is represented by 16 sample vectors. In order to design the classifier it is neccessary to estimate the unknown parameters (typical sample vectors and a common acceptance radius) in a learning or training phase. For that a sample of patterns has to be observed. In the case of BIRD images the forest fire had to be verified by additional information.
If the pixel to be tested is inside a (hyper) sphere of given radius around one of these sample vectors, the pixel is classified to belong to the sample vectors class. This comparison is done with 8 different classes. It is explicitly allowed for one pixel to belong to different classes, e.g fire and vegetation. If this is not desired, sample vectors and hyper sphere radius have to be chosen in a way, that hyper spheres around sample vectors of different classes do not overlap.

## 3 Handel-C

Celoxica (*http://www.celoxica.com*) has developed an FPGA PCI card called RC1000 with a Xilinx Virtex FPGA. This is delivered with an IDE (*DK1 Design Suite*) including a Handel-C compiler. All further tests are done with that board and compiler.
Celoxica provides a good documentation [3]: Handel-C basics, the use of provided utilities and the communication with the PCI card are explained including working examples.
The DK1 Design Suite allows an easy start with Handel-C, especially when using debug features like single step execution. I/O channels can be used to exchange data with such a simulated FPGA, but that concept is quite different from communication with the real hardware, so it is useful to switch to communication with an FPGA design running on the PCI FPGA card quite early.
Using the compiler handelc.exe directly from the command line instead of using the design suite became more interesting when the project grew, e. g. for batch builds and stopping build times.

[1] many thanks to the DLR

Handel-C debug build, Handle-C FPGA build and PC code to access the FPGA card all use different header files with the same name, so the right one for the right task has to be included.

### 3.1 Portability of C code

The main reason for using Handel-C is to re-use existing algorithms and familiar programming concepts. Therefore converting C code to Handel-C is quite important. In general, porting algorithms written in C is possible, though there are a few things to notice.
All data types have a certain bit length that can be specified explicitly, e. g. `int 8 my8Bit;`. Casting between these types is not possible, but there are operators available for bit concatenation (`int 10 my10Bit = 0 @ my8Bit;`) and bit truncation.
Arrays should be declared `ram`, e. g. `ram int 8 myArray[10];` unless there is a good reason for not doing that (see section 3.2 and 4.1), because registers are generated otherwise.
Structs can be used too, but not classes, just as one would expect in a language called Handel-C instead of Handel-C++.
Recursive function calls are not allowed, the build process responds to the attempt with: `'Design contains an unbreakable combinational cycle'`. This is a quite hard constraint that avoids generating hardware for stack of unknown size. It might be possible to use recursive macros instead.
There is no floating point support at all in Handel-C.

### 3.2 Optimizability

The main source of optimization is parallelizing parts of the program flow. Handel-C supports that with the keyword `par{}`, that declares all commands in the following braces to be executable in parallel, even if this is not possible regarding the content or for technical reasons.
One such technical reason is the access to addressable memory (arrays declared `ram`). This is not possible in parallel, at least in general, because dual port ram is used. So no more than two concurrent accesses to explicitly defined ports can be realized. Otherwise the build process fails `'Routing active signals'`.
Handel-C projects grow quite rapidly, optimization flags increase compile times rigorous but might be needed do avoid exceeding FPGA resources and single processes use up to 320 MB of memory during compilation. So very small changes in Handel-C source code, e. g. increasing an array, can mean much more effort for the build process. Section 4.1 shows how difficult optimization can become that way.
Instead of a complete FPGA design build, VHDL output can be generated from Handel-C. That is not human readable any more, but might be useful for further processing with existing VHDL optimizers.
The following remarks refer to handelc.exe version 3.0.2505. Unfortunately there are some code constructs that cause unexpected error outputs or even crashes. If `sample` is declared `ram`, then `if (Val0>sample[0]) { Abs0 = (int)(0 @ (Val0 - sample[0])); }` causes the error message `'1 unrouted active'` just as one would try to access `sample[0]` several times in parallel.
The used version of handelc.exe can be improved, especially for larger tasks. A typical build scenario needs 30 minutes to compile, another 35 minutes for handelc.exe to free memory in 8 KB blocks without any new output, and 15 minutes for the remaining build process (e. g. routing).
Complete compile system crashes occurred on high FPGA resource usage ($\geq$ 75% slices) at the end of the build process in map.exe.

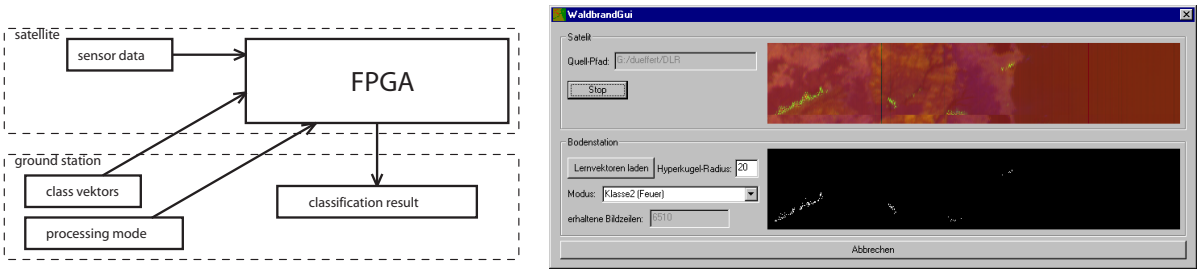


Figure 1: Data Flow and resulting ForestFireGui at work

# 4 ForestFireGui

Figure 1 shows the example implementation ForestFireGui, a realization of the task introduced in section 2. It consists of a GUI on a PC and the classification algorithm written in Handel-C running on a Celoxica FPGA card.

## 4.1 Optimization Process

An 1 GHz PC with sufficient memory was used for compiling for all the following measurements. The resulting program (GUI on PC, algorithm on FPGA card) was executed on a 500 MHz PC with an FPGA card running at 60 MHz (if not stated otherwise).
The average time of 20 runs was determined, each run classifying the same image line 128 times completely. The balance time (loading an image line from file, writing it to the FPGA card, reading the result from the FPGA card and displaying both) was measured for comparison too. The run times in the following table pass for once-only classification of one complete image line and are therefore calculated by (20*run time - 20*balance time)/128.

| No. | configuration | compile time [min] | FPGA slices | avg. run time [ms] |
|---|---|---|---|---|
| 0 | RAM, -O- oder -O+high | 4:03 | 16% | 49 |
| 1 | RAM, 2× par, -O+high | 2:39 | 9% | 32 |
| 2 | RAM, 2× par, optimized | 3:29 | 9% | 32 |
| 3 | FF, 2× par, -O+high | 19:07 | 44% | 32 |
| 4 | FF, 16× nearly flat, -O+high | 79:56 | 70% | 11 |
| 5 | FF, 16× flat, optimized | 97:31 | 65% | 1.6 |
| 6 | FF, 16× flat, optimized (100 MHz) | 97:31 | 65% | 1.0 |
| 7 | for comparison: 500 MHz PC | 0:03 | — | 7 |

0) The first test was completely sequential. In 3 nested loops, the innermost loop calculated the distance between one pixel of the image line and one test vector of one class. High level optimization had no effect. The resulting execution time was much too high: nearly 6000 clock cycles for completely classifying one pixel. Therefore the Handel-C implementation was optimized without changing the algorithm.

1) Everything parallelizable within the innermost loop was marked with `par{}`, the compile time even decreased because high level optimization simplified further steps (like generating net list and routing). Compared to this little change the execution time decreased significantly.

2) Further optimization flags only influenced compile time in this case.

3) To allow higher parallelism, the sample vector array had to be realized with flip-flops instead of addressable RAM by removing the keyword `ram`. This lead to significantly increased compile time and used FPGA resources, but did not influence run time, because the program flow has not changed at all.
4) Unrolling the biggest part of the innermost loop by hand by calculating additional temporary variables 16 times in parallel started to exploit the potential of parallel hardware. At the considerable expense of compile time and FPGA resources the resulting FPGA program ran 3 times faster.
5) Unrolling the rest of the innermost loop was harder than expected, because using only high level optimization reproducibly crashed the PC at the end of the compilation process. But using all optimization flags finally produced a pretty fast implementation.
6) This final design was able to run at 100 MHz too. So it was possible to speed up the same algorithm by factor 50 in total by parallelizing it as far as the resources allowed it. Unfortunately, this needed more manual work then expected.

# 5 Conclusion

The language Handel-C allows for fast porting recursion free non-floating point algorithms already existing or easily implementable in C to an FPGA platform. If minimum development time is the main criterion, the usage of Handel-C can be recommended. But such a Quick'n'Dirty solution largely discounts the possibilities of FPGA chips: initially the program flow is completely sequential.
Handel-C provides a concept for parallelization, but the source code may have to be restructured to be able to use it. Whereas addressable and sequentially accessible only memory can easily be converted to FPGA structures by the Handel-C compiler, parallelly accessible memory realized by single flip-flops needs large amounts of central memory during compilation and FPGA resources and leads to super-proportionally rising compile times and optimization times. Unfortunately this is hard to notice from the C syntax, and the multiple-stage compile process does not give much information about its remaining run time. Therefore you need some experience even for smaller projects to be able to judge which maximum number of RAM bits and flip-flops can be compiled, optimized and routed within a reasonable time without exceeding the capacity of the target FPGA at the end.
The need to subdivide Handel-C variables into RAM and flip-flops as well as unroll loops manually complicates optimization very much, because it demands restructuring the source code. It would be preferable having to define only which C statements could hypothetically be executed in parallel. Then an automatic optimizer should decide, how much of that is realized in parallel hardware and how much should be done sequentially to save compile time and FPGA resources at the expense of execution time.

# References

[1] Scientific Assessment of Space-borne High Temperature Event Observing Mission Concepts.

[2] The ECOFIRE Study.

[3] Celoxica Limited. RC1000-PP Software User Guide. Version 1.20.

[4] W. Skrbek, E. Lorenz. HSRS - An Infrared Sensor for Hot-Spot-Detection.