Good Technology Makes the Difficult Task Easy

Andrey Terekhov
Saint-Petersburg State University
Russia, Saint-Petersburg,
Universitetsky pr. 28
+78124287109
ant@math.spbu.ru

Abstract

A new language for chip design is presented. The main advantages of the language are explicit conveyer and parallel features fully controlled by the author of chip design. Non trivial industrial example is under discussion. There are run-time estimations and comparison with traditional programming in C.

Keywords

Hardware design, CoDesign, VHDL, Verilog, pipeline

1. Introduction

It is well known that new technologies could intensify the appearance of new, previously unknown methods of calculation, solutions of equations and other complex problems.

There are many examples where the existence or the absence of a particular technology changes the situation dramatically. Let's start with the most familiar to us. The emergence of high-level programming languages Fortran, Algol 60, Cobol and other in 50-60th years led to increased programmers' productivity, but would not be possible without the appearance of the source code analysis technology, optimization, code generation and other compiler techniques.

Another example – it's well known that if one implements the microprogram to perform some "time-consuming" function, one can reduce computation time in 10-20 times. All computers IMB/370 which have been also produced in the Soviet Union, had the opportunity of dynamic microprogramming, but nobody used this opportunity, as it was hard work to write firmware for mainframe even though everyone knew that it can reduce calculation time. In the mid-80s a member of our System programming laboratory of the Faculty of Mathematics and Mechanics – Nikolay Fominykh invented a solution how to create microprogramms in high level languages [1] and, as if by magic, the problem of a very complex turned into a problem which we did not even will to accept as a thesis - it became the task of the student-level investigation work on 3rd-4th courses.

Exactly the same thing is happening now with graphical programming languages. It is difficult to push the designer to use only standard UML, it is hard work even to correctly understand where and which of 13 types of UML diagrams to be used. So to do a good project (not only "images for bosses" but the project, which will automatically be converted into an executable computer program) - is a very difficult job. And here a way out eventually found - programmers began to create and use languages for each specific domain (Domain Specific Languages, DSL). Again, it was found that very difficult tasks are converted into common ones using well-chosen DSL for each subject area.

It is clear that it is very expensive to develop image diagram editors, repositories, code generators, debugging tools, etc. for each DSL. And here there was a solution – metatechnologies - technologies that are able to automatically generate the necessary technology for DSL from a set of not too complex formal descriptions [2].

2. Problem statement

This article focuses on the process of chip design. Here (we'll talk about flexible chips FPGA - Field Programmable Gate Array) chip algorithm is fundamentally different from those of traditional algorithms that programmers write every day in C or similar languages. It is believed that the time advantage of using programmable chip is achieved by the use of parallelism. 100-200 activities and processes can run in the chip at the same time, and thereby greatly benefit in execution time compared to traditional programming. But our experience has shown that the main gain is achieved not through a direct parallelism, but mainly due to pipelining. Many actions can not be performed in one clock cycle, so they have to perform for several consecutive cycles. If they use different devices, it is always advantageous to arrange the so-called conveyer - a sequence of actions in which the first step of a piece of accounts performed by a group of devices, interim results are transferred to another group of devices that in the next cycle will continue to do so, but the first group of devices in the next cycle can also take another piece of data, and to do its job, and the depth of the pipeline can be quite large. It is common knowledge, and everyone knows it, but really deep pipelines are very difficult to implement.

Traditional programming languages for the chip design VHDL and Verilog have no special features for pipelines. A VHDL programmer should create data communication through auxiliary registers, to synchronize (if desired portion of the data did not come, the conveyer must pause, and then, at the time the right data received, to resume its work). All of this is so troublesome and difficult, that conventional VHDL programmers do not create deep pipelines.

The main motive of this report is the appearance on the market of the product Vivado of Xilinx company [3], which is advertised as an automatic converter from C language into VHDL and Verilog. Upon closer inspection reveals that the real limitations to the input program, which can be converted to VHDL, are very high - no dynamics, loops and arrays with dynamically computed bounds. Everything is designed for the fact that modern large computers with large memory can build a syntactic parse tree, build control flow graphs and data flow graphs to make the necessary optimization and when such graphs are static (fully known at compile time), one can rely on the use of various optimization mechanisms and, as the authors of the tool say, by almost automatically obtain decent VHDL programs. It is still difficult to assess the real capabilities of the tool, how it is really effective.

It is better to go the other way, that is to give the programmer comfortable tool of expression and powerful efficient technology. Then we can solve a much wider class of problems.

3. Non trivial example

Our team has many years of experience on development a tool for design hardware implementations of complex applications on the basis of language HaSCoL (Hardware and Software CoDesign Language) [4, 5, 6, 7, 8]. This language is a convenient tool for the programmer to explicitly specify parallelism - a few steps one can write with a separator «|», thus the compiler is instructed that they must be carried out in parallel in a single cycle, and a means of specifying pipeline - a few steps separated by ";" set the conveyer, the system provides the auxiliary registers storing intermediate results, synchronization, the interaction of pipelines, their suspension and resumption. We believe that languages without explicit parallelism and explicitly set the pipeline may not be as effective as the languages in which such actions are predetermined, and we hope to demonstrate the advantages of our platform on a real industrial example.

As the example, we chose the image search in the screen. In order not to torment the reader the intricacies of the algorithm, we chose a standard correlation algorithm, assuming that the screen consists of 512 by 512 pixels, where is necessary to find an image (frame) of 128 by 128 pixels. It's pretty time consuming algorithm, but works well. Briefly describe the nature of the algorithm. Viewing the initial screen, each piece of screen (size of 128x128) must be normalized, and then the inner product of the resulting normalized values of pixels with a normalized value of the frame is computed. Maximum of these inner products indicates the fragment found.

Define screen size as M*N (in this example M=N=512). We must find an etalon in this screen – the array of pixels m*n (in this example m=n=128).

Normalization is performed by the following formula:

$$a'_{ij} = \frac{a_{ij} - \bar{a}}{\sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(a_{ij} - \bar{a}\right)^2}}$$

where $\bar{a}$ is average from all screen areas a_{ij}. The sum plays an important role in the proposed calculation scheme

$$\sum_{i}\sum_{j}(\mathrm{a_{ij}} - \bar{\mathrm{a}})^2$$

It is easy to show $\sum_i \sum_j (\mathrm{a_{ij}} - \bar{\mathrm{a}})^2 = \sum\sum \mathrm{a_{ij}^2} - \mathrm{m} * \mathrm{n} * \bar{\mathrm{a}}^2$

Let's estimate the complexity of the algorithm. Fragments of the screen to be normalized and compared with the etalon will be (512-128) * (512-128) = 384 * 384.

In each fragment to find the sum of the elements to calculate the average and the sum of their squares, i.e., with 2 * 128 * 128 actions to reduce the number of additions, we apply the standard for this kind of problems approach when the sums are stored in rows "accrual basis", there is an array in which each element is the sum of the previous n elements. Each new line item is obtained by adding to previous one and subtracting the previous element that comes n items back. So to get the sum of all elements of the next fragment we need only 2*128 operations (add elements of the last column), instead of 2*128*128.

To calculate the square root of a 32-bit integer, it was possible to find an algorithm with a cycle of 16 repetitions of 5-7 actions.

For the normalization of the elements and get the inner product any simplifications are not applied.

Thus, the resulting complexity of the algorithm can be estimated

384 * 384 * (2 * 128 + 6 * 16 + 10 * 128 * 128) = 384 * 384 * 164192 action.

C program for this algorithm works on MacAir about 30 seconds. At our request, graduate student Stanislav Sartasov using the system CUDA, applied the GPU, which parallelized the inner loop of 128 repetitions. He received a counting time of 0.5 seconds on a good processor with 2 cores.

4. Proposed solution

We now describe the program (main program elements provided at the end of the report) by using identifiers of its code in HaSCoL (s, s2, az, etc.). At first, we introduce an array of a (etalon) and consider the sum of all the pixels and the sum of squares of pixels

$$zv = \sum_{\mathrm{i=0}}^{\mathrm{m-1}}\sum_{\mathrm{j=0}}^{\mathrm{n-1}} \mathrm{a_{ij}} \quad s2 = \sum_{\mathrm{i=0}}^{\mathrm{m-1}}\sum_{\mathrm{j=0}}^{\mathrm{n-1}} a_{ij}^2$$

Then we calculate denominator

$$\mathrm{az} = \sqrt{\mathrm{s2} - \mathrm{m} * \mathrm{n} * \left(\frac{zv}{m * n}\right)^2} = \sqrt{\mathrm{s2} - \frac{zv^2}{m * n}}$$

and change etalon pixels be normalized values

$$\mathrm{ap_{ij}} = \frac{a_{ij} - \frac{zv}{m * n}}{az} = \frac{m * n * a_{ij} - \mathrm{zv}}{\mathrm{m * n * az}}$$

If $\mathrm{a_{ij}}$ is unsigned 8 bits, then $\sum_{\mathrm{i=0}}^{\mathrm{m-1}}\sum_{\mathrm{j=0}}^{\mathrm{n-1}} \mathrm{a_{ij}}$ request 22 bits, $\mathrm{a_{ij}^2}$ request 16 bits, and $\sum_{\mathrm{i=0}}^{\mathrm{m-1}}\sum_{\mathrm{j=0}}^{\mathrm{n-1}} \mathrm{a_{ij}^2}$ - 30 bits (m=n=128).

Similar calculations are performed on all the m*n rectangles of the screen, the upper left corner is defined be the indices k and l, in each of these rectangles $zp_{k+i,l+j}$ is calculated, similarly to ap_{ij} where k=0..M-m+1, l=0..N-n+1.

The total value of the estimated function for the rectangle k, l

$$\mathrm{F_{kl}} = \sum_{\mathrm{i=0}}^{\mathrm{m-1}}\sum_{\mathrm{j=0}}^{\mathrm{n-1}} \mathrm{zp_{k+i,l+j}} * \mathrm{ap_{ij}}$$

We are interested in the maximum of this function and the indices k and l, where it was achieved.

Sum S collected by rows in the array

s [m, N] of 15 bits

the sum of squares of lines going in the array

s2 [m, N] of 23 bits.

All crystal arrays stored in bram (special memory), each of which allows a maximum of two read operations per cycle.

Let's start demonstration from a simple example (Listing 1). In the line m-1 (counts from 0) reached column n-1, so the first piece formed for counting - a rectangle whose top left coordinates of k = 0, l = 0. For the sum of all elements of the rectangle we must add elements in the last column (which already has the sums of lines). This piece is not the most critical, so implement it on a lot of hardware resources is not desirable.

Specially for such occasions, if the author of the program believes that there is no sense to parallelize some not very important actions on a large number of chip elements in the language HaSCoL there are while-statement and a few other similar operators. This is normal loop which is repeated several times on the same fragment of the apparatus. While statement inhibits the current conveyer, for example, to fragment Listing 1, the algorithm issued by one intermediate result every clock, but since the beginning of operator while will produce a result in 128 clock cycles. But the cycle being once started, will run to the end, without the need for additional information.

Due to hardware limitations result of reading bram memory can not be used in the same clock cycle. It turns out that the complexity of the loop is 128 * 2 cycles. Proceed differently: with each repetition cycle increase counter and start subprocess sum1, with the launch of a subprocess and execution of his first action within one clock cycle.

Subprocess sum1 (Listing 4) starts the conveyer of 2 stages. Thus the complexity of 128 * 2 is replaced by 128 cycles!

Next is the square root (Listing 2). This while loop runs for 16 repetitions each time when the previous segment finishes its work. This is also the conveyer - while the square root of the result of the first conveyer element is counting, in parallel with it the first loop is calculating for the second element and so on. Thus, the computation time of this section may be neglected.

Now let's discuss the most interesting and time-consuming program fragment (Listing 3 and Listing 5). Here, we use macro generator realized many years ago, by our researcher Anton Moskal [9]. The idea is to replace the linear addition of 128 elements of the column by logarithmic addition in 7 iterations. But even these seven iterations we arrange as the conveyer in the form: in step 1 add together the adjacent conveyer elements in step 2 - amounts pairs, fours later and so on. We need 127 summators (64 +32 +16 + ...), but the chips are now more powerful, so it's not a problem.

Thus, in the Listing 3 a cycle of 128 repetitions is described, the loop body (Listing 5) takes exactly one clock, with 128 running parallel processes with the working depth of the conveyer equal 10 (I just counted the number of characters ";" in the generated program).

Thus, the body of the outer loop with 384 * 384 iterations has 3 parts - 128, 96 and 128 clocks, arranged in a pipeline. They practically run in parallel with the exception of a small "tail" of the plot, running after the completion of the outer loop. This means that the complexity of the body of the outer loop is approximately 128 clocks, and the complexity of the program 384 * 384 * 128 plus a small constant which is more than 1200 times less than the original version!

In our view, Xilinx Vivado can not create so effective program, because there is no conveyer driven by the programmer.

5. Conclusion

This program in HaSCoL was written by me during one week, plus a couple of working days spent on the coordination of interfaces, improved code generator for VHDL, bug fixes, etc. It is even difficult to imagine how long it takes for an engineer who owns VHDL, to write a similar program with the same deep pipelines.

We have conducted several industrial experiments. For example, the program for calculation the signatures of computer stereo vision in the language HaSCoL was written just during 1 week, and an engineer working full time, wrote in VHDL the same program more than six months. In HaSCoL already implemented neurocomputer containing 500 neurons in a single crystal VIRTEX VI, Macroblaze processor and several other large systems. This allows us to conclude that there are really big opportunities for technology.

Listing 1

```
if pk >= m1 && pl >=n1
then
ii=0;

while ii < m
do
 inform (sum1(ii,pl)) | ii:=ii + 1
done;
skip;
```

Listing 2

```
x:=ext(zz - (((zv >* zv) >> sh){0:29} :uint(30)), 32) | y:=0;
j:= (30 :uint(5));
while j >= 0
do
 zzz:=((ext(y,32)<<2)+1)<<j | y:=y << 1;
 if x >=zzz
 then y:=y + 1 | x:=x – zzz
 fi;
 j:=j-2
done;

if y == 0 then y:=1 fi;
```

Listing 3

```
while ii < m
 do
  inform(sum2(ext(ii, 10), iii, y, flagend, kk, ll)) | ii:=ii + 1
 done
```

Listing 4

```
sum1(i,l)
{
sil = s[i][l] | s2il = s2[i][l];
zv:=if i == 0 then 0 else zv fi + ext(sil,22) |
zz:=if i == 0 then 0 else zz fi + ext(s2il,30)
}
```

Listing 5

```
sum2(ii,iii, y, flagend, kk, ll)
{
#define mj 0
#while mj < n
  aij(mj) = a[mj][ii] |
 let imj = iii + mj in
 { zij(mj) = z[if imj >= m then imj - m else imj fi][ii] } |
#if mj == 0
  F2 = (ext(az >* y, 44) :int(44)) |
#endif
#set mj $eval(mj+1)
#endw
skip;
```

6. References

[1] «Using a standard Extensible algorithmic language in microprogramming», Nikolay Fominykh, Dialog Microcomputer systems, Moscow, Moscow State University, 1986 (in Russian).
[2] "Architecture of visual modeling environment QReal», System Programming, Vol. 4, 2009, A. Terekhov, T. Bryksin, Y. Litvinov, K. Smirnov, G. Nikandrov, I. Ivanov, E. Takun, number of pages 26, (in Russian).
[3] Manual: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf
[4] "The evolution of our views on CoDesign over the past 15 years," a personal website: http://www.math.spbu.ru/user/ant/History_Evol_CODESIGN.pdf, (in Russian).
[5] "Using Hardware-Software Codesign Language to implement CANSCID", Oleg Medvedev, Ilya Posov, Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on, pp 85 -88, http://oops.math.spbu.ru/dours/HaSCoL/MedvedevPosovMEMOCODE2010.pdf
[6] "Accelerating multiple alignment on FPGA with a high-level hardware description language", Oleg Medvedev, Central Eastern European Software Engineering Conference in Russia, 2011, http://oops.math.spbu.ru/dours/HaSCoL/SECR11.pdf
[7] "Overview of high-level language development equipment HaSCoL the example of a clone processor Xilinx Microblaze", Oleg Medvedev, the Second Scientific Conference of Young Specialists "Start the Future", 2011, pp. 231-234, http://oops.math.spbu.ru/dours/HaSCoL/KBSM.pdf
[8] "Hardware Description Language Based on Message Passing and Implicit Pipelining", Dmitri Boulytchev, Oleg Medvedev, EWDT 2009, http://oops.math.spbu.ru/dours/HaSCoL/EWDT09.pdf
[9] "Presentation of communication source and target texts in the automatic generation of program texts," A. Moskal, Automated software reengineering, St. Petersburg, St. Petersburg University,2000, (in Russian).