OPTIMIZING A 3D IMAGE RECONSTRUCTION ALGORITHM: ANALYZING THE CAPABILITIES OF A MODERN COMPILER

Tom Vander Aa
ESAT, KULeuven
Belgium
tvandera@esat.kuleuven.ac.be

Rudy Lauwereins
IMEC
Belgium
rudy.lauwereins@imec.be

Geert Deconinck
ESAT, KULeuven
Belgium
gdec@esat.kuleuven.ac.be

ABSTRACT

Today's DSP processors are so complex, it has become impossible to program them using assembly. To get the maximum performance out of the applications running on such devices very good compilers are needed. This paper analyzes the capabilities of those compilers by optimizing a compute-intensive 3D-image reconstruction algorithm on the TMS320C6701 ('C67) DSP processor from Texas Instruments.

Because the 'C67 is a VLIW processor, performance depends on the ability of the compiler to detect parallelism. By rewriting the C source code, we made it clear to the compiler which code was not data dependent, and thus could be executed in parallel. Over all optimizations the average instructions per cycle rose from 0.41 to 2.61 ($\times 6$) and the number of instructions to be executed was divided by 3.6. The net result was a performance increase of 2200%.

For every discussed optimization step we state the problem that prevented efficient code generation by the compiler and say how we overcame this problem. We show that for a lot of the steps the performance problem was caused by a lack of provisions to efficiently communicate between the user and compiler. We had to *trick* the compiler in doing the optimizations we wanted by writing the program the right way. This was a long and tedious process. Therefore, we look at what provisions should be added to improve communication and reduce development time and time to market.

1. INTRODUCTION

The TMS320C6701 DSP from Texas Instruments is a VLIW processor that can execute up to 8 RISC-like instructions in parallel. It is difficult to get high performance when programming for a VLIW processor like this one, because its performance depends heavily on the compiler. If the compiler is able to extract the parallelism available in the program, it can use the true power of this processor. If the compiler cannot detect the parallelism (even though it might be available), the performance will be crippled. Getting the compiler to understand your program is a long and tedious process.

On the other hand time-to-market is becoming increasingly important, so any means to speed-up this process is welcome. In this paper, we make suggestions to make the programmer and the compiler better understand each other and come to a satisfying solution faster.

There are two models to build a compiler. The first model looks at a compiler as a black box: source code goes in; object code comes out. The second model considers the compiler as white box: total exposure of the inner workings. Neither model is useful. The black box provides no information at all, so the user does not know what is going wrong. The white box compiler is too complex and would take months to understand. What is needed is a compiler that outputs only the information that is relevant to the programmer. In the rest of this paper we will try to find out what this relevant information is.

The rest of this paper is organized as follows. In section 2 we introduce the algorithm and in section 3 the hardware platform. Section 4 defines the cost function we want to use. The optimizations are described in section 5 and evaluated in section 6 on the 'C67. The conclusion is section 7.

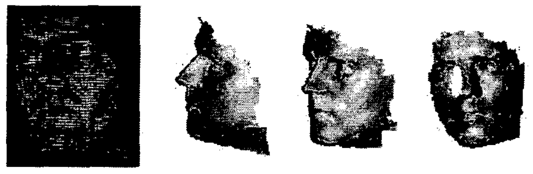

Fig. 1. A one shot 3D acquisition algorithm; a. image, with projected grid; b. reconstructed third dimension

2. A 3D-IMAGE RECONSTRUCTION ALGORITHM

The algorithm under consideration is a one-shot 3D acquisition system, which generates 3D shape descriptions from a single image, taken of a scene on which a simple grid is projected [5]. Viewed from a different angle, the grid appears deformed in the image, from which the three-dimensional shape can be extracted (see Figure 1).

The algorithm can be divided in roughly three steps. First, the crosspoints of the grid in the image are detected with pixel precision. A second step improves the accuracy of the grid with subpixel precision by applying an iterative energy minimizing snake-like process. The last step is the actual 3D shape extraction from the detected crosspoint coordinates. A description of the different steps in the algorithm, and how the crosspoint coordinates yield 3D shape can be found in [5] and [6].

Snake Algorithm: The snake process, essential for a good 3D reconstruction, is the most time consuming part of the algorithm. It changes the grid to let it more closely follow the pattern lines,

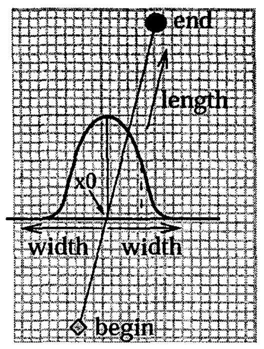

Fig. 2. Inner loops

```
1  slope = compute_slope_of_begin_end_line ();
2  for ( length = begin; length <= end; length++)
3  {
4    x0 = compute_x0(slope, begin, end);
5    pix\_x = ( int)x0;
6    while ( ( w = gauss(x0−pix_x)) > 0.0001 ) {
7        sum += w∗image[width][length];
8        norm += w;
9        width−−;
10     }
11
12   width = ( int)x0 + 1;
13   while ( ( w = gauss(width−x0)) > 0.0001 ) {
14       sum += w∗image[width][length];
15       norm += w;
16       width++;
17   }
18
19    integral = sum / norm;
20 }
```

Fig. 3. Inner loops, (pseudo-) source code

minimizing the energy E, where

$$E = \sum_{p=1}^{N_E} I(C_P, E_P) + \sum_{p=1}^{N_S} I(C_P, S_P) + ST$$

with N_E and N_S the numbers of crosspoints with E and S neighbors, respectively, and where $I(C_P, X_P)$ denotes the average intensity along the line segment between the center crosspoint C_P and the neighboring crosspoint X_P in E or S direction. Smoothing terms (ST) are added to keep the grid from becoming too irregular. Maximization of E moves the grid towards the darker positions in the image, i.e. the line centers.

Inner Loops: 98% of the total runtime in the SNAKE is spent in its two most inner loops. In these two loops one integral (I( ) in the above formula) is calculated. Therefore all optimizations focus the inner loops. We will explain this code briefly.

As depicted in Figure 2 graphically and in 3 as source code, the second most inner loop iterates over the connection line between two crosspoints (`length` loop, statement 2 in Figure 3), while the most inner loop iterates perpendicular to this connection line (`width` loop, statements 6 and 13 in Figure 3). The `width` loop calculates the average intensity, weighted with a gauss-like function, around one pixel (at coordinate [width][length], line 7). The `length` loop sums all these values to make the complete integral $I()$. Both iteration spaces are data dependent: the outer one depends on the position of the crosspoints at pixel level, the inner one on the position of the connection line at sub pixel level. Note that `x0`, which is the intersection of the interconnection line between the two crosspoints and the horizontal 'pixel' line, see Figure 2, is calculated as a floating-point value. This results in sub-pixel level calculations.

3. THE TARGET PLATFORM

The 'C67 is a modern VLIW processor [3]. It has two identical banks of four functional units. Each unit has a specific task (branch unit, logic unit, multiplication unit and shift unit) and executes a RISC-like instruction every clock cycle. Each bank is connected to a file of 16 32-bit registers. The register files are connected to the internal memory via a very high bandwidth bus.

The pipeline of the 'C67 is divided into three stages [9]: fetch, decode and execute. In the fetch stage a fetch packet that contains 8 instructions of 32 bits is retrieved from the program memory. In this packet there is indicated what instructions have to be executed in parallel. A new packet is not fetched until all instructions from the previous fetch packet are executed. The execute stage varies in length: 5 cycles for integer operations and 10 cycles for floating point operations. This means that a floating point result is only written to the register file 9 cycles after the source operands were read. Internally these stages are fully pipelined, such that a new floating point instruction can be started every cycle. A branch operation (5-cycle execution phase) has 4 delay slots: 4 slots of each 8 instructions that immediately follow the branch, are always executed, whether the branch is taken or not. This means the compiler has to fill these slots with instructions that are independent of the outcome of the branch.

There is no hardware support for detecting data dependencies which are all left to the compiler to solve. This makes it almost impossible to program this processor in assembly, since the programmer has to keep track of all the instructions that are being executed. With a pipeline depth of 10 and up to 8 instructions executing in parallel this is too much information to handle.

4. THE COST FUNCTION

We want to minimize the runtime of our 3D-image reconstruction algorithm. The factors involved in this cost function are:

1. The chosen algorithm. The algorithm says in what way the output of the program for a given input is calculated. We might be able to improve here, if we are able to *change the algorithm* but still can keep the output within the desired limits.

2. The implementation, which is the description (in our case in C) of the algorithm in such a way that the compiler can understand it. The better the compiler understands our implementation, the faster it will run on the target platform. We can divide our optimizations here into two categories. First of all we can change the implementation in such a way that

it will run faster on any target platform (*platform independent optimizations*), but we can also take the target platform into account and do *platform specific optimizations*.

3. The target platform. Given that we cannot change this, we cannot alter this part of the cost function.

So the three categories we will divide our optimizations in are: algorithmic changes (cat. 1), platform independent implementation changes (cat. 2) and platform specific implementation changes (cat. 3).

Comparing the gain for those three categories, we saw we had the most benefit from optimizations of category 3 (66%), while categories 1 and 2 only contributed 19% and 15%, respectively. From now on we will only consider these platform specific optimizations.

5. OPTIMIZATIONS

We use an approach with a closed loop interaction between the programmer (who knows the algorithm) and the compiler (that knows the hardware) in which the feedback in both ways is essential. The compiler should report to the user what optimizations it can and cannot do and why. The user uses this feedback to change and annotate the source code so the compiler can do a better job.

For every discussed optimization we first state the *problem* that prevented efficient code generation by the compiler and say how we *solved* this problem. For a lot of the steps the performance problem is caused by a lack of provisions to efficiently communicate between the user and compiler. Often the compiler has to assume worst case in places where the user knows better. The user is unaware of such assumptions because the compiler does not tell him. Other times the user knows what the compiler is doing wrong, but has no means of telling him otherwise. We say what provisions should be added to improve communication (*suggestion*). Finally, we also mention the resulting *speed up* (relative to the previous step).

Note that (as said in section 4), all the optimizations are *platform specific optimizations*.

5.1. Signal Type Refinement

Specification of the algorithm includes *signal type refinement*. In this part the important signals are analyzed. The minimal number of bits to achieve sufficient precision is determined. If necessary this also includes specifying rounding modes, overflow behavior, etc.

Signal type refinement should be done in a platform independent way. Once the platform is known an efficient way has to be found to map these refined types into types supported by the platform. The next paragraph will show that both signal type refinement and its mapping have great influence on performance.

5.1.1. 32-bit Floating Point

Problem: The main calculations in the inner loops are floating point calculations. Originally the algorithm was implemented on variables of the type *double*. Later the variables were converted to *floats* (32bit, floating point, with less precision than *doubles*, but still sufficient for this algorithm).

However, only the declarations were changed from *double* to *float*. This created very inefficient code where constants were used: by default constants, written in the form *1234.5678*, are treated as 64-bit floating point constants of the type *double* [4]. This would lead to superfluous conversions from *double* to *float* and vice versa. For example,

(w = gauss(width−yo)) > 0.0001

would be treated by the c-compiler as if it said

(**double**)(w = gauss(width−yo)) > 0.0001

Solution: The correct way is for the user to write *1234.5678F* such that the compiler recognizes the constant as being 32-bit *float*.

Suggestion: The compiler should be more verbose in such cases.

Speed-Up: 17%

5.1.2. Fixed Point

Problem: On this platform (and most other), floating point operations are more expensive than integer operations, because the pipelines of the floating point functional units are much deeper (cf. Section 3).

Solution: Fixed point variables have more precision than their floating point counterparts when using an equal number of bits. The drawback is that for fixed point numbers the dynamic range is more limited. So when the dynamic range of the variables is known beforehand, fixed point numbers should be used. If the range is unknown, floating point has to be used. Fixed point numbers are implemented as integers with an imaginary binary point on this platform.

There are a number of cases in the algorithm where fixed point can be used (cf. Figure 2):

1. The *length* loop iterates over the pixels between two cross-points. This is done by setting up the line between the cross-points in floating point calculus and determining the pixels nearest to this line. But since pixel-coordinates are sufficiently small (i.e. their range is equal to the image dimensions) you can use fixed point calculation. As a bonus you avoid expensive floating point (`float`) to integer (`int`) conversions (line 5 and 12 of Figure 3).

2. Every integral is a weighted sum, where the weights *w* are stored in a look-up table. The look-up table has a granularity of 200 elements per unit and a range between *-5.0F* and *5.0F*.

3. The values in the look-up table are also floating point and limited (from *-1.0* to *1.0*). If we convert them to fixed point we'll go from a float-integer multiplication to an integer-integer multiplication (line 7 and 14 of Figure 3).

Speed-Up: 58%

Suggestion: The compiler should provide a mechanism to refine and map the type of each variable. A pragma could be use to annotate the variable declaration with the refined type and the mapping onto the hardware supported type. In this way the original algorithm is not changed but merely extended with pragmas.

5.2. Using Intrinsics

Problem: The 'C67 does not have a division instruction. If one would write, for example,

$$a = \frac{y2 - y1}{x2 - x1}$$

the compiler would generate a function call to the *_fdiv* function, which would cause significant overhead.

Solution: A better way is to use *intrinsics* [7]. They instruct the compiler to use an instruction that otherwise could have not been expressed as a regular C expression. There is an instruction for the *reciprocal* (_rcp). The above example becomes

$$a = (y2 - y1) * \_rcp(x2 - x1)$$

Suggestion: The compiler should give a warning when it generates the *_fdiv*-call. Furthermore the use of intrinsics should be avoided as much as possible, because it makes the code less portable. Better would be to use a pragma to say this division should be done using the *_rcp* instruction.

Speed-Up: : 30%

5.3. Software Pipelining

5.3.1. Enabling software pipelining

Software pipelining [1] is the most important step to get maximum performance out of the 'C67 (as documented in [8]). If the compiler detects that it is possible it overlaps the different iterations of the most inner loop of the program to get optimal parallelism. The only condition that has to be satisfied is that it has to know the minimal amount of iterations in the loop, so that it can determine the amount of overlap.

Problem: The original inner loop (*width*) contains a data-dependent *while* statement, which makes it impossible for the compiler to software pipeline it. If we convert the *while* loop to a *for* loop, with constant bounds the compiler instantly does a much better job.

We will show why we can do this. For sufficiently accurate results the loop has to be executed 3 or 4 times, depending on the input data. If we take the upper bound (4) we will certainly be accurate enough. The loss due to the extra iteration in some (rare) cases is heavily compensated by the improved parallelism, as can be seen from the speed-up of 60%.

Speed-Up: 60%

5.3.2. Improving performance of the pipeline

The performance of the "software pipeline" depends on two things:

1. **The number of iterations in parallel** : the more stages in parallel the faster of course. This is limited by a number of hardware parameters (e.g. the number of functional units in the 'C67, the number of memory paths) and by the program (e.g. how well the instructions can be mapped onto the different functional units)
2. **The total number of iterations** : if this is too low the pipeline will never be completely filled up and operate at full speed, because most of the time the loop will be in the *prologue* or *epilogue* phase, where parallelism is low.

```
   [!A1]  B     .S1    L12
||        STW   .D2T2  B3,*+SP(136)

          STW   .D2T1  A12,*+SP(4)
          STW   .D2T1  A10,*+SP(12)
          STW   .D2T1  A6,*+SP(16)

          MV    .L1X   B4,A15
||        STW   .D2T2  B11,*+SP(144)

          MV    .L2    B6,B10
||        MV    .L1X   B8,A13
||        MV    .S2X   A8,B13
||        LDW   .D1T1  *+A4(12),A14
||        STW   .D2T2  B12,*+SP(148)
          ; BRANCH OCCURS
```

Fig. 4. Branch with delay cycles

```
for (i = begin; i <= end; i++) {
  x = (float)i;
  yo = y1 + a*(x-x1);
  j = (int) floor (yo);
  sum = norm = 0.0;

  for(k=0; k<4; k++) {
    w = gauss(yo-j);
    sum += w*PIX(i, j--);
  }
}
```

Fig. 5. Code before branch optimization

Problem: Our software pipeline is too small because the loop body only contains one multiplication, one addition, while the 'C67 contains 2 multiply and 2 add units.

Solution: We could remedy this by unrolling the inner loop once and doing two iterations in parallel but then the number of iterations (4) would be too small and pipeline would not get filled.

A better solution is to unroll the *length*-loop and merge two iterations of this loop into one. This way the compiler can put twice as much instructions in the inner loop, possible doubling the parallelism.

Speed-Up 7% for this last solution (unrolling the *length:* loop).

Compiler feedback: Here the compiler provided excellent feedback about how it did software pipelining. This way the programmer can easily make the trade-offs.

5.4. Branches

Problem: Branches on the 'C67 have 4 delay slots [9]. This means that the instruction flow changes 4 cycles after the branch occurs in the code. This also means that the 4 instructions after the branch are executed whether the branch is taken or not. Figure 4 shows example code.

If there are two nested loops, like in Figure 5, the outer loop will not be software pipelined. Therefore it is better to move as

```
x = (float)i;
yo = yl + a*(x-xl);
j = (int) floor (yo);
sum = norm = 0.0;

for (i = begin; i <= end; i++) {
  for (k=0; k<4; k++) {
    w = gauss(yo-j);
    sum += w*PIX(i, j--);
  }
  x = (float)i;
  yo = yl + a*(x-xl);
  j = (int) floor (yo);
  sum = norm = 0.0;
}
```

Fig. 6. Code after branch optimization

much code as possible before the branch of the outer for-loop. These instructions can then be used to fill the delay cycles of the branch (Figure 6).

Speed-Up: 2%

Suggestion: Better feedback is needed from the compiler. The compiler can reorder instructions to be placed in the delay slots, but its knowledge is limited. If the programmer knows where exactly the problem lies, he might be able to identify and mark (using pragmas) independent instructions.

6. RESULTS

In this section we will discuss the impact of the optimizations described in the previous section.

For compilation, we used the Code Composer Studio version 1.00 [10], which uses the TI compiler version 3.01. This was the compiler available at the time these experiments were done. When the newer compiler (v4.00) became available we also tested our optimizations with it. The new compiler provided a consistent gain of approximately 10% on each version of the code. The original version was 10% faster and the final version too, so the overall speed up was the same. Compiler options used are: -mv6700 to generate 'C67-specific code, full optimizations (-o3) and all the options to get compiler feedback (-kss -alsx -mw -os -on2).

To obtain the results, we used the built-in hardware counters of the 'C67 to measure clock cycles, NOP and non-NOP instructions. We also analyzed the executables and counted the functional unit usage for the different versions of the program. The execution count of each loop was calculated using the basic block profiling tool bprof [2]. This way we could calculate the Instructions Per Cycle (IPC) for the 'C67.

The numbers reported are for an image containing 465 × 320 pixels.

6.1. General Speed-Up

The total speed-up is a factor is 22. This means that the total algorithm, which originally took 9 seconds per reconstructed image, now can do 2.5 images per second. This makes it suitable for daily use, in – for example – a hand held 3D camera.

Nr.	*Optimization Step*	*Relative Speed Up*
1	Software Pipelining	60%
2	Fixed Point	58%
3	Using Intrinsics	30%
4	Floating Point Annotation	17%
5	Improved Software Pipelining	7%
6	Avoiding NOPs	2%

Table 1. Performance improvements of the different optimizations. Numbers shown are the relative improvements compared to the result of the previous optimization step.

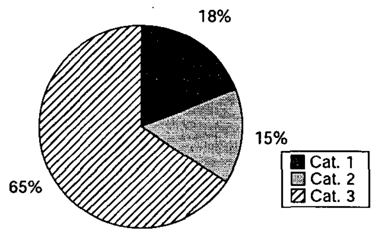

Fig. 7. Relative speed-up per category; Cat. 1: Change the algorithm; Cat. 2: Change the implementation, HW independent; Cat. 3: Change the implementation, HW dependent

Table 1 shows an overview of the Platform Specific Optimizations (cat.3). The biggest improvement is *Software Pipelining*, which is also the biggest gain overall (all categories). This confirms our theory that architecture specific optimizations give the best results. Indeed if we divide the gain in to the same three categories (Figure 7), we see that 65% is due to architecture specific optimizations.

We will see in the next section why the 'C67 is so sensitive to these optimizations.

6.2. Improved parallelism

The number of NOPs in the initial version (Figure 8a) is over 60%. This means that more than 60% of the time none of the 8 functional units is performing an operation. NOPs are inserted because a re-

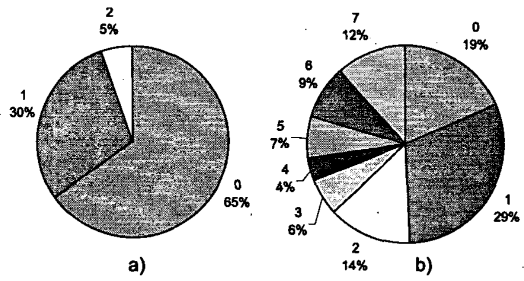

Fig. 8. IPC: original (a) versus optimized (b) version. The *integer number* indicates the number of functional units that are used. The *percentage* indicates in how many times this occurs.

sult that is needed for the next calculation is still in the pipeline. Since pipelines are so long on the 'C67, a lot of NOPs are inserted. These two architectural facts (deep pipelines and heavily parallel architecture) combined with the big amount of NOPs, indicate much potential of the 'C67 is unused in the initial program.

The final version has much fewer NOPs (Figure 8b). While in the original version the IPC ((average) Instructions per Cycle) is only 0.41, the optimal has an IPC of 2.61, with peaks of 7 in 12% of the code.

7. CONCLUSION

Because of the high complexity of today's DSP processors, it has become impossible to program them using assembly. To get the maximum performance out of the applications running on such devices very good compilers are needed. This paper has analyzed the capabilities of those compilers by implementing a compute-intensive 3D-image reconstruction algorithm on the TI 'C67 DSP processor.

The conclusion was, that if the programmer is not careful, the output of the compiler is disappointing. While with assistance from the programmer, the compiler can do a very good job. Indeed, 66% of the total gain was achieved by the programmer's combined knowledge of both the algorithm and the architecture. Because of its limited view of the program, the compiler has to make suboptimal or even worst-case assumptions. The programmer has better knowledge of the application. If he is able to communicate this information to the compiler, this will result in higher parallelism and performance. We we able to bring this information to the compiler by doing source-to-source transformations on the program. This way we were able to speed up the implementation with a factor of 22, making it usable in real-time applications.

Because exposing the parallelism this way is a long and cumbersome task, we wrote down several possible improvements to the way the compiler works. These improvements should facilitate the communication between the programmer and the tools. This speeds up development and reduces the time to market.

8. ACKNOWLEDGMENTS

Tom Vander Aa is supported by the FWO-project G.0036.99 and the TI Elite project. Geert Deconinck is a postdoctoral fellow of the Fund for Scientific Research - Flanders.

9. REFERENCES

[1] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[2] C. Fraser and D. Hanson. *A Retargetable Compiler: Design and Implementation.* Addison-Wesley, Menlo Park, CA, 1995.

[3] Jim Jurely and Harri Hakkarainen. TI's new 'C6x DSP screams at 1.600 MIPS. *Microprocessor Report*, 11(2), 1997.

[4] B. Kernighan and D. Ritchie. *The C programming language, second edition.* Prentice-Hall Inc., 1988.

[5] M. Proesmans, L. Van Gool, and A. Oosterlinck. Active acquisition of 3D shape for moving objects. In *Proceedings ICIP International Conference on Image Processing*, Lausanne, Switserland, 1996.

[6] M. Proesmans, L. Van Gool, and A. Oosterlinck. One shot active 3d shape reconstruction. In *Proceedings 13th ICPR International Conference on Pattern Recognition: applications & robotic systems*, volume Vol.III C, pages pp.336–340, Vienna, Austria, Aug. 1996.

[7] Texas Instruments. *TMS320C6x Optimizing C Compiler User's Guide*, Februari 1998.

[8] Texas Instruments. *TMS320C62x/C67x Programmer's Guide*, February, 1998.

[9] Texas Instruments. *The TMS320C6000 CPU and Instruction Set Reference Guide*, March, 1999.

[10] Texas Instruments. *Code Composer Studio White Paper*, May 1999.