Optimizing CPU Performance: Image Creation in Computer Graphics

Carlos Manuel Ferreira Silva
Manuel António Machado Carvalho

Departamento de Informática, Universidade do Minho
4710 – 057 Braga, Portugal
mi7007@di.uminho.pt
cei7075@ di.uminho.pt

Abstract. This work aimed to explore techniques to reduce the execution time of the creation of an image in a single-processor computer, using ray tracing software (PIRT 1.0). Applying a profiler to the source code (in C), the main bottlenecks were identified, machine-independent optimization techniques were applied to the longer functions – namely function inline and assembly inline – and the obtained results were analysed.

1 Introduction

The case study is PIRT 1.0. In a very simple way, program reads a file with a representation of a scene with objects, trace light ray on these objects, verify where the light ray intercepts object, verifies possible reflections of the ray on the part of object intercepted and construct an image file (type TARGA) where are recorded the image of the process of rendering. [1] [2]

2 Optimizations

Speed is the first thought which comes to mind if one thinks of optimizing. Everyone thinks speed and only few are left talking about disk-space and memory requirements, no doubt they all together make a well optimized program but unlike in DOS days - when 640K of memory was the matter of concern. Today processor time is coming out to be the scarcest resource. However, that is the case of PC programming, in case of embedded systems, memory and code size come out as important issues.

Optimizations are usually small modifications saving a few micro seconds only but these savings are magnified and made apparent when the optimized code fragment is used multiple times like in a loop or in a function called multiple times.

3 Optimizations Techniques

3.1 Using an Editor and Compiler

Some compilers can generate code for common tasks saving to the programmer a lot of effort. They can optimize the generated executables for speed and/or size usually one at the cost of other. For example in Microsoft Visual C++, it is possible to choose optimized executables either for speed or for size. When optimizing for size, the compiler chooses the smallest code sequence possible and when optimizing for speed, compiler chooses the fastest sequence. [3] [4]

3.2 Faster Cycles

Iteration is a very common element in any program and there are many simple and effective optimizations that can be applied to loops. Usually, any optimization becomes effective only when the statements take place inside a loop.

Example:

```
for( int i = 0; i < 3; i++ ) array[i] = i;
```

this is logically the same as

```
array[0] = 0; array[1] = 1, array[2] = 2;
```

3.3 Reducing Calculations inside Loops

Sometimes the same expression appears many times in the code and as a result it has to be evaluated multiple times wasting time. This can be avoided be using a temporary variable to store the result and then use this temporary variable instead. Although this technique improves execution speed, it makes the code less readable [7].

Consider the example:

```
 if ((dataStructPointer->ExpensiveFunctionCall()) < 10)
{
   /* code */
}
else if ((dataStructPointer->ExpensiveFunctionCal()) > 30)
{
   /* code */
}
```

This code can be rewritten as

```
Inttemp = dataStructPointer->ExpensiveFunctionCall() ;
if(temp < 10)
{
     /* code */
}
else if(temp > 30)
{
     /* code */
}
```

The second code statement is faster than the first one.

3.4 Using Assembly Code

It can be useful to look at the assembly generated by the compiler to see what is going on and exactly how many instructions are generated for a set of C statements. This way, we can figure out which set of C statements will run faster. For example in Microsoft Visual C, the statement to access value of a variable by using its name generates two instructions, whereas the same value if accessed by a pointer generates tree instructions. [5] [6]

4 Some Applied Tecniques

The Microsoft Visual C++ Profiler tool was used to find the slowest functions. The profiler generates a list with the execution times for each function.

4.1 Inline[1]

In a call to a function, a context switch occurs, that implies a push/pop of registers in stack (Fig.1).

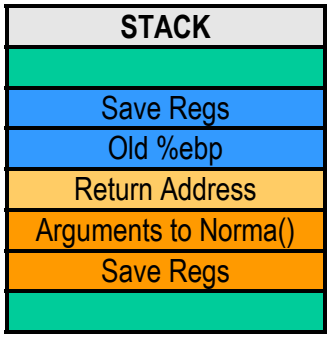

Fig.1. Stack evolution in context change.

In PIRT code the *Norma()* function was replaced by the code that implements its body. By doing it, the additional code to manage the function call/return was reduced.

Original code:

```
extern double Norma (Vector *v)
{
   return(sqrt(v->X*v->X + v->Y*v->Y + v->Z*v->Z));
}

extern void Normalize (Vector *v)
{
   double n;
   n = Norma (v);
   if (fabs(n) > FZERO) {
         v->X /= n;
         v->Y /= n;
         v->Z /= n;
   }
}
```

Optimized code:

```
extern void Normalize (Vector *v)
{
   double n;
   n = sqrt(v->X*v->X + v->Y*v->Y + v->Z*v->Z);
   if (fabs(n) > FZERO) {
         v->X /= n;
         v->Y /= n;
         v->Z /= n;
   }
}
```

[1] http://www-106.ibm.com/developerworks/library/l-ia.html

4.2 Using Assembly Code

Some referred techniques were implemented in the PIRT 1.0 code. The *Norma()* and *fabs()*functions, used several times in the source code, were modified with the code statements referred in the next paragraph. [5] [6]

```
double myfabs(double x)
{
  _asm
  {
        FLD [x];
        FABS;
       FSTP [x];
}
  return x;
}

double mysqrt(double x)
{
  _asm
  {
        FLD [x];
        FABS;
       FSTP [x];
}
  return x;
}
```

5 Conclusions

The implemented optimizations had significant improvements in the execution of the PIRT 1.0 code. Fig.1 shows the execution times due to the different optimizations. The optimization technique of Pointer Differencing led to the fastest execution, excluding the technique where assembly were used. It is verified that these techniques help the compiler to generate more efficient code.

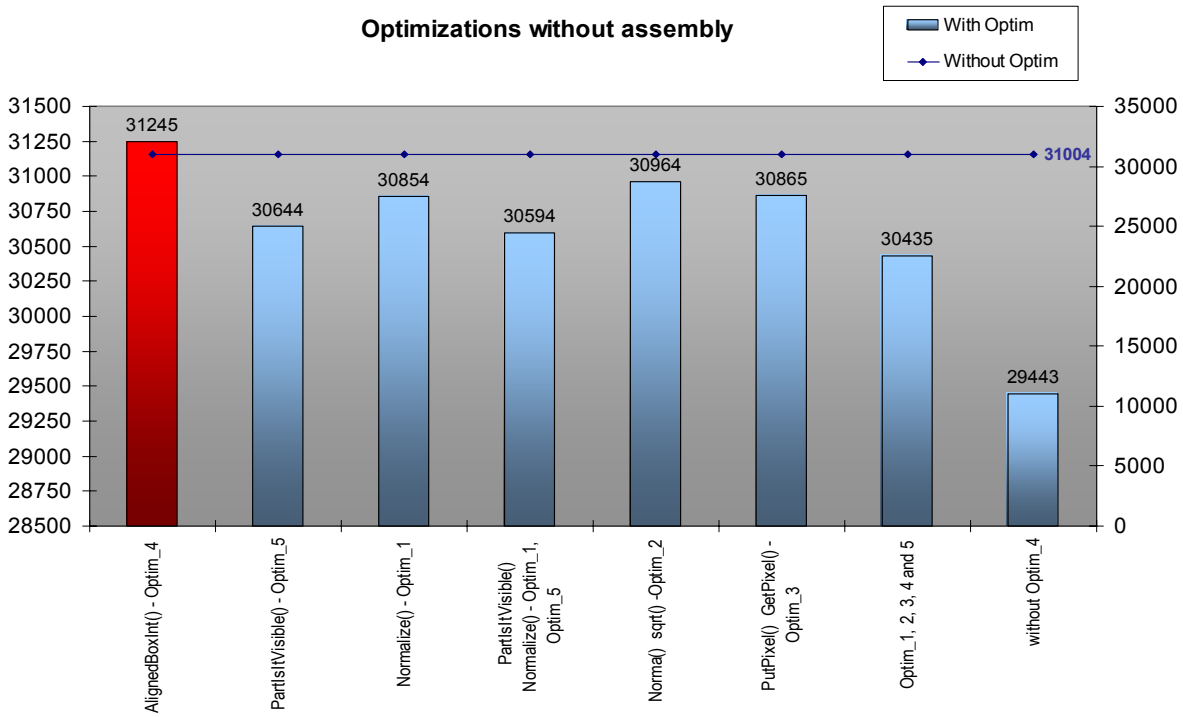

Fig. 2. Optimizations without using assembly code.

When assembly code was inserted in the C program, the PIRT 1.0 execution time reduced 6.33% (Fig.3). It achieved a reduction of two seconds in thirty and one seconds that the PIRT 1.0 program completes its tasks without the optimizations [7].

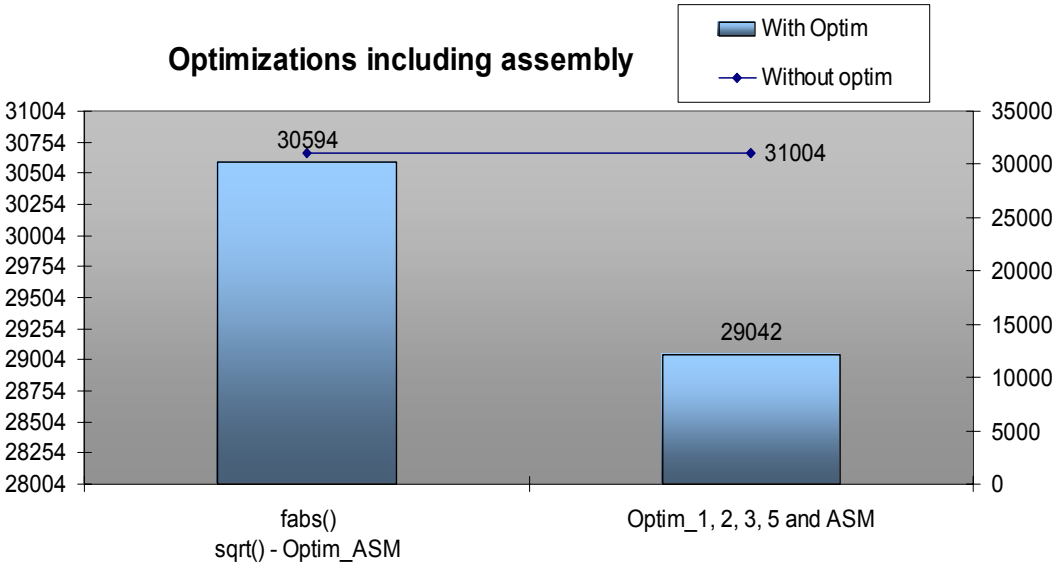

Fig.3. Optimizations using assembly code.

Fig.4 shows the execution times for the following configurations:

- Without any optimizations;
- Compiler optimization for maximum speed;
- Compiler optimization for maximum speed and ASM optimization;
- Compiler optimization for maximum speed and optimizations 1,2,3,5.

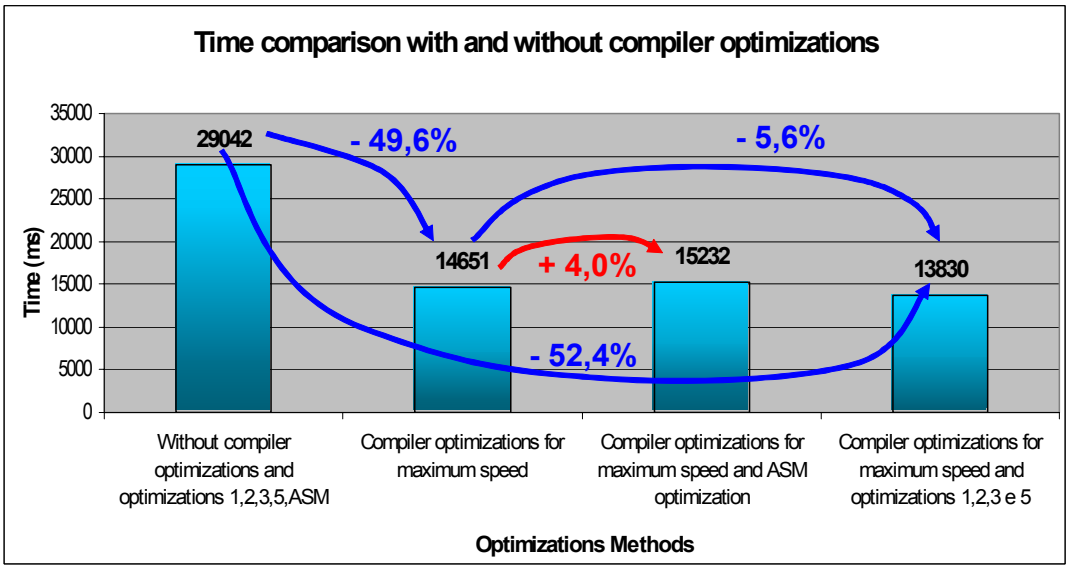

Fig.4. Time comparison.

Using full optimizations on the compiler options, the execution time reduced to 50%. Using the compiler optimizations and the optimizations 1, 2, 3, 4, 5, and ASM, the execution time increased 4%.

The best execution time was achieved when using together optimizations 1, 2, 3, 4, 5 (without ASM) and compiler optimizations.

The compiler generates more efficient code than the one used in the ASM optimization. In situations where the context matters, the compiler cannot optimize the code. In that case it is possible to obtain better execution times.

References

[1] Santos, Luis Paulo P., Parallel Intelligent Ray Tracer (PIRT) 2.0, Internal Report, Department of Informatics, University of Minho, Portugal (1999), also in http://gec.di.uminho.pt/psantos,

[2] TGA File Format, Technical Manual Version 2.2 January, http://www.ludorg.net/amnesia/TGA_File_Format_Spec.html (1991).

[3] IA32 - Intel Architecture Software Developer's Manual, Volume 2 Chap 3 http://www.intel.com (2002).

[4] Randal Bryant and David O'Hallaron, Computer Systems: A Programmer's Perspective, Prentice Hall (2002).

[5] Hennessy, John L., Patterson, David A., Computer Architecture- A Quantitative Approach, Third Ed, Morgan Kaufmann Publishers (2003)

[6] Proença, Alberto José, Introduction to Machine-Level Representation of C Programs - Version 2, Lecture Notes, Department of Informatics, University of Minho, Portugal (2001)

[7] Manuel Carvalho e Carlos Silva, Técnicas de Optimização de Código C, Internal Report, http://gec.di.uminho.pt/micei/ac/ICCA04/Proc/W1-PIRT.zip, Department of Informatics, University of Minho, Portugal (2004)