Optimizing and generating report for a monothread encoding library MPEG-4 AVC Livrable PARA 7.3

Henri-Pierre Charles
and Sajjad Khawar
and Denis Barthou (UVSQ)
In collaboration with Françoise Préteux and and Son-Minh Tran
and Marius Preda (INT) and F. Bodin (CAPS)

Mars 2008

Abstract

Optimizing and code generating report for a mono-thread encoding library MPEG-4 AVC for the PARA project

The multimedia kernel and the technical platform has been chosen in the first part of the project. Some preliminary results has been obtained on the two kernel families (SAD and SATD).

This report presents global results for all the kernels and for the global application.

Contents

1 Introduction

This work has been done in collaboration with :

Université de Versailles Saint-Quentin en Yvelines (UVSQ)

- Denis Barthou
- Henri-Pierre Charles
- Khawar Sajjad

INT Évry

- Françoise Préteux
- Son-Minh Tran
- Marius Preda

2 Introduction and Background

This report is devoted to the MPEG-4 AVC mono-thread optimization.

Following is a brief overview of the experimentatios environment and the performed optimizations.

2.1 Previous work

The first [1] report was devoted to the study of the X264 benchmark and different experiments. These experiments has shown that speedup opportunities exist but are difficult to reach because

- multimedia instructions are difficult to use
- the most important optimizations (vectorization, load vectorization) are difficult to use with an optimizing compiler.

But we were able to choose the experimentation platform :

Hardware the Itanium II platform because it has a large spectrum of optimization (multimedia ISA, multiple load versions, etc).

Software X264 which is an MPEG 4 encoder / decoder

Benchmarks the benchmark we use are the standards video used by the MPEG expert group, and our goal is to reach a real time (25 frames per second) compression for all image sizes.

Multimedia Kernels Profiling reports have shown two codelet families SAD and SATD. These codelets are statically specialized with 7 differents sizes (4x4, 4x8, 8x4, 8x8, 8x16, 16x8 and 16x16). These sizes correspond to portions of images that are processed by the X264 algorithm in the detection phase.

On this platform we have get promising results with the SAD kernel using different optimizing techniques :

Vectorization

Multimedia instructions

Load optimization

These results has been presented in the previous report.
The following work was remaining at the end of the previous report :

- Parallelization of SATD
- Experimentation with SATD kernels
- Comparaisons using newer compiler versions
- Inclusion of SAD and SATD in the application
- Improve the previous results
- Try to automatize these optimization

All these tasks have been done and are presented in this report. Improvements in the register allocation is presented in the section 3, the vectorization of SATD are presented in the section 4, the section 5 explains the methodology applied which is a step toward a vectorization automation and the new results are presented in the section 6.

3 Register Allocation

Register allocation has a profound impact on the performance of X264. The number of registers consumed by a program generally imposes an upper limit on the level of parallelism in the code. The number of consumed registers can be reduced by reusing registers. This register reuse on the one hand improves the level of parallelism but on the other hand may introduce false dependencies, resulting in a slow down. Apart from resolving these issues such a register allocation scheme needs to be efficient in terms of the runtime overhead.

We have incorporated a scheme that keeps track of the available registers at runtime and tries to avoid any false dependencies. The scheme has been kept as simple as possible to be suitable for a runtime system. The scheme allows registers to be allocated in the form of *register sets*. These register sets can be made *alive* or *dead*, anywhere in the program, by the help of two function calls *makeAlive* and *killSet*. Physical register numbers of the registers in an alive register set can be obtained by the function *reg_num*. The *reg_num* function accepts as input the reference to a register set and the *register index* of the desired register. The register indexes are similar to the indexes of elements in an array.

Figure 1: Global available registers data structure

Figure 2: A *regSet* data structure

3.1 Register Representation

In our system we have two kinds of registers: the *available registers* and the *occupied registers*. These two kinds of register-classes have been organized in the form of two data structures: the *global available registers* data structure and *regSet* data structures.

3.1.1 The Global Available Registers Data Structure

In this data structure the registers are represented in the form of an *ordered table* of available *contiguous register intervals*. This table has been implemented using a linked list to allow for low overhead insertion/deletion operations. The register intervals have been represented as simple data-structures that contain the first and last register number of each interval. The data structure has a *Max Reg* field that keep track of the current highest register number used during the execution of a certain code fragment. This highest register number can be useful for getting hardware registers allocated, e.g. through an *alloc* instruction on the Itanium IA64 architecture. These representations have been shown in figure 1.

3.1.2 The *regSet* Data Structures

Each *regSet* data structure has a representation similar to that of the global available register data structure. The register intervals occupied by a *regSet* are represented in the form of an ordered table implemented as a linked list. In addition these data structures have fields that indicate the size of the *regSet* and whether or not the *regSet* is alive. A sample *regSet* data structure has been shown as figure 2.

3.2 Register allocation API

A register set has the type *regSet*. A new register set is created by the *new_ regSet* function call. This function takes the size of the desired register set as input and returns a register set of the required size. A register set can be activated or *made alive* with the function call *makeAlive*. This call acquires a set of registers from the global set of available registers and renders them useable. The set of acquired registers may consist of several non contiguous register intervals. The *killSet* function call disactivates the referred register set and renders the acquired registers available. With the help of the *makeAlive* and *killSet* calls, we can easily manage the utilisation of registers in a code fragment. In other words *makeAlive* and *killSet* explicitly mark the interval of code where a register set is alive.

Register sets can have symbolic names thereby making them similar to program variables. The *reg_ num* function call returns the physical register number of the indexed register in the referred register set. This makes register set indexing similar to array indexing. These few function calls make register manipula-

Figure 3: A compilette code fragment using the register manipulation API

tion transparent, manageable and easier for the assembly language programmer. The use of all these functions have been shown in the code fragment given in figure 3.

4 Vectorization

Vectorization is a technique that permits arithmetic calculations or data manipulation on groups of data called *vectors*. Most modern processors have 64/128 bit registers and they allow the treatment of a single register or a pair of registers as a vector. They also provide vector instructions to perform different vector operations. Vectorization can thus be used to reduce the number of operations in a program. For example if a certain processor allows vector operations on vectors of four elements then, we may reduce the number of operations by a factor of 4. This may translate into a speedup of the application.

4.1 Vectorization of *SAD* and *SATD*

Vectorization is usually effective on array manipulation code. Therefore *X264* functions *sad* and *satd* are good candidates for this optimization. These functions operate on two arrays of pixels and calculate their *sum of absolute difference* and their *Hadamard transform* respectively. We have implemented the vectorized versions of these functions on the Itanium IA64 architecture. The Itanium architecture provides a number of different vector arithmetic instructions like *padd*, *psub*, *pshl*, *pshr* etc. etc. There are also a number of vector load instructions available on this architecture e.g. *ld2*, *ld4* *ld8* etc. etc. The Itanium architecture requires data to be aligned on the natural boundary of the concerned load instruction. For example, for *ld4* the data needs to be aligned at the boundary of 4 bytes etc.

In the case of the *sad* and the *satd* functions, the elements of the first array of pixels are always aligned on the respective natural boundaries. But the elements of the second array are not necessarily aligned. Hence, the loads for the first array can be vectorized without any problem, but to vectorize the loads for the second array we have to load two consective vectors and then perform some vector manipulation to obtain the desired vector. Once the vectors have been loaded we can vectorize the calculations without any problem.

We have used the Itanium IA64 instruction *psad* for vectorizing the *sad* operations. This instruction is capable of calculating the *sum of absolute difference* between two 8 element vectors in just 2 cycles. Further we can launch upto 6 *psad* instructions in one cycle, as we will see in the following section.

We have used various powerful IA64 vector instructions for the implementation of *satd*. These include *padd*, *psub*, *pmax* and *mix* instructions. The *mix* instructions have been used for various vector re-structuring operations.

Figure 4: ILP exploitation in SAD

Figure 5: ILP exploitation in SATD

5 ILP Improvement

Modern processors allow several assembly instructions to be issued in parallel, this phenomenon is called *Instruction Level Parallelism* or *ILP*. The Itanium IA64 architecture allows up till 6 instructions to be launched during one cycle. Itanium organizes its instructions in the form of *instruction groups*. An instruction group is a sequence of instructions that may be executed in any order(sequential or parallel) to give the same end-result. In other words there is no dependence between any two instructions in an instruction group. Instruction groups are delimited by assembly language *stops* that are represented as ';;' in assembly language code.

High levels of ILP can be obtained by merging up small independent instruction groups into larger combined instruction groups. This reduces the number of stops in a program, thus reducing unnecessary barriers in a program. Good instruction organization in the form of instruction groups may also help us overcome the latencies of high latency instructions.

5.1 ILP exploitation in *SAD* and *SATD*

Both SAD and SATD functions have no inter-iteration data dependence. Different iterations of these functions can, therefore be, launched in parallel, wherever possible. Furthermore the different computations of each iteration may be grouped together to form larger independent instruction groups to exploit parallelism.

The iterations of the SAD family of functions are small enough to be launched together in parallel. To achieve higher levels of ILP in the SAD functions the operations have been grouped together as shown in the figure 4.

In case of the SATD functions the internal loop iterations are larger and consume a large number of registers. Hence, it is not feasible to launch all the iterations in parallel. So, instead of parallelizing all the iterations together, we have parallelized all the internal loop iterations within one single external loop iteration. The external loop has then been unrolled completely to give the final code. The code structure within one external loop iteration of a sample SATD function has been shown in figure 5.

6 Optimization Results

The performance of the optimized code has been tested against that of the unoptimized code. The performance results have been divided up into two categories: performance of kernels and performance of the overall application. In the following sub-sections we discuss these performance results.

Figure 6: Speedups of SAD kernels

Figure 7: Speedups of SATD kernels

6.1 Kernel Performance Results

As mentioned earlier in the introduction, the *SAD* and the *SATD* functions have been identified as the most time consuming or *hot* functions. We call these hot functions as the kernels. The speedups of different *SAD* kernels have been shown in figure 6, whereas those of the *SATD* kernels have been shown in figure 7.

The code of each kernel has been compiled with *ICC 9.1* option *-O3* & *GCC 4.2* option *-O3* and then generated with the help of compilettes. The speedups of all these kernels are w.r.t the naive versions compiled by *GCC 4.2* option *-O0*.

This way of computing speedup is more accurate because the *-O0* has stable results and is a stable base.

From these plots it is clear that *GCC 4.2* optimizes the code better for small kernel sizes whereas *ICC 9.1* optimizes well for the larger kernel sizes.

For *SATD* *ICC 9.1* uses a powerful code generator able to wrap multiple loop iteration into one loop but this technique is not efficient for small loop size.

On the other hand the code generated by the compilettes works fine for all kernel sizes but get better results for largeer sizes using a better schedule between load and computations.

6.2 Overall Application Performance Results

The application performances have been shown in the form of *frames per second* which is our main goal. A "real time" compressor should compress at 25 frames per second.

Different composures of the overall application have been tested for different benchmark videos. All these combinations have also been tested separately for *ICC 9.1* and *GCC 4.2* compilers. The different results have been shown in the figures 8 to 11. The different application composures are as follows:

1. Application without any specialized functions (compiled with option *-O3*)
2. Application with only *SAD* functions specialized (compiled with option *-O3*)
3. Application with only *SATD* functions specialized (compiled with option *-O3*)
4. Application with both *SAD* and *SATD* functions specialized (compiled with option *-O3*)

For both *ICC 9.1* and *GCC 4.2* the overall gain in frame per seconds is about 20-25%, when both the kernels have been optimized. In case of *GCC 4.2* both the kernels contribute 50-50 to achieve the gain. But in case of *ICC 9.1* , most of the gain is due to the contribution from the *SAD* kernels.

Figure 8: Application frame rate for different benchmark videos using *GCC 4.2*

Figure 9: Application frame rate for different benchmark videos using *GCC 4.2*

7 Conclusion and perspectives

7.1 Obtained results

This report explains the results we have reached on optimization of the X264 video compressor.

Our optimization enables real time only on some videos and some formats. For example the video "mobile" with `qcif` size become real time with *GCC 4.2* (on figure 9). The optimized version is under 25 frame per second, the optimized one is up.

The global results with *ICC 9.1* are generally better, but our optimization is able to reach the "real time" goal locally : news/cif, sean/cif, weather/cif (see figure 11).

Unfortunately the most "dynamic" video with larger size are far from our goal. The video `children/sif`, `mobile/sif`, `table/sif` need an acceleration factor beween 3 and 4. The following options can be followed in order to get these factors :

- Vectorize the following major kernels in the code
- Go higher in the code, i.e. vectorize and optimize higher level fonction which aggregate multiple kernel calls.
- Enable multithreading, `X264` has a mutithreading option which is not used in our experiment because we focus on monothread optimizations.

Thanks to the dynamic compilation, we are able to build an efficient library more efficient than start of the art compiler, highly specialized but without the large memory occupancy drawback. Our technology is able to build efficient code on demand, for example we can generate and specialize only the kernels needed by a specific run.

7.2 Interaction with ASTEX

ASTEX is a tool developped by CAPS. It partitions C codes into code kernels that are the most executed code fragments (found in hotpaths). Moreover, these kernels can be automatically extracted and a memory analysis is performed in order to find opportunities for specialization. This helps in particular a vectorization phase, by finding most frequent data alignment, since they may a high impact on performance.

Figure 10: Application frame rate for different benchmark videos using *ICC 9.1*

Figure 11: Application frame rate for different benchmark videos using *ICC 9.1*

The execution of ASTEX on x264 code shows finds the code kernel described in this deliverable. We believe that ASTEX would first help to focus on code fragments that need the approach presented in this deliverable, and then it offers a framework for a run-time optimization as described here, since the kernels considered for vectorization correspond actually to "codelets" in the HMPP framework used by CAPS.

7.3 Perspectives

The actual technologies we compete with are hand tuned libraries and "intrinsic extensions" in compilers.

Our optimization is able to build automatically and dynamically libraries without the memory occupancy drawback. For example if a specific run does not need 4x4 block, the corresponding kernels will not be generated.

The major advantage of our technology is to be sufficiently generic to take advantage of more optimization opportunities (constant values, memory alignments, multimedia instructions, etc) that can not be taken into account by a classical library building tool like Atlas, SPIRAL of FFTW3 ([4], [3], [2]).

The "intrinsic extensions" in compilers allow to use specialized instructions but are not generic enough in a library building context.

We are also able to use specialized register allocation that can take advantage of algorithms specificities and target processors.

References

[1] Denis Barthou and Henri-Pierre Charles. Livrable para 5.1. rapport d'analyse et de restructuration de code monothread des modules p, t, q et f de l'encodage mpeg-4 avc. Technical report, Université de Versailles Saint-Quentin en Yvelines, November 2006.

[2] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[3] Markus Püschel, Bryan Singer, Jianxin Xiong, Jose M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms.

[4] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.