8—12 Evaluation of Two Real Time Low Level Image Processing Architectures

Stelian Persa Cristina Nicolescu Pieter Jonker
Pattern Recognition Group, Technical University Delft
Lorentzweg 1, Delft, 2628 CJ
The Netherlands
+31 15 278 3727
{stelian,cristina,pieter}@ph.tn.tudelft.nl

Abstract

The paper presents a study on the impact of using SIMD (Singe Instruction Multiple Data) techniques and architectures in low level image processing. Speedups obtained on a SIMD parallel architecture (IMAP-VISION board) and a single Intel MMX processor computer are presented for different low-level image processing operators. While the IMAP-Vision system performs better because of the large number of processing elements, the MMX processor remains a good candidate for low-level image processing.

1 Introduction

Because of the massive amounts of data involved, computer vision algorithms can be extremely computationally expensive. In order to make this processing run in real-time we need to process data in parallel and often a great deal of optimization needs to be utilized.

It can be observed that most of the image processing operators exhibit natural parallelism in the sense that the input image data required to compute a given area of the output is spatially localized. This high degree of natural parallelism exhibited by most of the low-level image processing operators can be easily exploited using SIMD parallel architectures or techniques. A SIMD architecture consists of a linear array of simple processors capable of applying in parallel the same instruction on different elements of the data.

The paper is organized as follows. Section 2 describes a SIMD architecture - the IMAP-VISION board. Section 3 presents the way in which a SIMD technique was integrated into the MMX technology of Intel processors, and how to get most speed out of MMX. Execution times and implementations of different low-level image processing operators computed on both architectures (SIMD and MMX processor) are presented in Section 4. Section 5 concludes the paper.

2 The IMAP-VISION - a SIMD architecture

IMAP-VISION is a SIMD Linear Processor Array (LPA) on a PCI board. It is a parallel architecture for real-time image processing, that includes a high level language for data parallel programming: 1DC (One Dimensional C). The IMAP-VISION card contains 256 8-bit Processing Elements (PEs), controlled by a 16-bit Control Processor (CP) and has 10 GIPS peak performance.

Although the IMAP chip has high levels of computational ability by integrating one-dimensional SIMD processors on a single chip, its on-chip memory is sometimes insufficient for flexible execution of complex algorithms. For that reason the IMAP-VISION board has a high bandwidth external memory with an efficient data transfer mechanism. The bus controller allows the host PC to access data not only in data memory, but also in on-chip memory and the external memory without collisions, even when a real-time vision computation is running. This mechanism not only allows the host PC to collect the results created by the processor array, but also allows the host PC to change the parameters in run-time easily.

1DC is designed as an enhanced C language to support virtual LPAs. The enhancement of 1DC from C is straightforward: extended declaration of entities which associated to the PE array (SEP or distributed variables), extended constructors for selecting active processor groups, and extended operators (like *mif*) for manipulating data on the PE array. To define a distributed variable the 1DC keyword *separate*, or *sep* can be used. A separate variable is stored in internal memory and can have a different value on each processor. A 256x240 image, distributed column-wise over the PEs is defined as *separate unsigned char img[240]*.

To make the value of a sep variable on a specific processor globally available it can be addressed using the :[pe_num:] operator. The operators for accessing variables on other processors, :< and :>, can be used in two ways. As a unary operator, *sep_var* = :< *sep_var*; will rotate the values of *sep_var* 1 processor to the left. As a binary operator *sep_var* = *sep_var* :< *n*; will rotate the values of *sep_var n* processors to the left.

The IMAP-VISION board comes with a MS Windows or X window (e.g. Linux) programming environment based on 1DC, which makes this parallel board a powerful tool. The X window based debuggers provide not only assembly and source level debugging facilities, but also functionality's such as interactive adjustment of variables and constants, which is useful for parameter tuning in real-time imaging applications.

2.1 Algorithms Implementation in 1DC and Optimizations

Writing a program in 1DC is not very difficult, but writing a good program is. A program that works is not necessarily efficient. In order to write efficient, i.e. fast, programs certain guidelines should be followed. The first is to always try to exploit parallel processing to the maximum. If you have 256 processors you should use all of them. The second is to avoid using nested loops.

This sometimes necessitates a workaround, which is not trivial. A simple but very expressive example is the histogram computation, which is a global image processing operation. First every processor makes a local histogram of the image column it has in its memory. Probably most people would write something like:

```
for(i=0;i<256;i++)
  for(j=0;j<PENO;j++)
    histo:[I:] += tmp[i]:[j:];
```

But in that case we have a nested loop in which only 1 processor is working at a time. A more efficient use of processors would be to do a prefix addition, one of the most basic forms of parallel programming. It uses only log(256) = 8 steps:

```
for(i=0;i<256;i++){
  for(j=0;j<8;j++)
    tmp[i] += tmp[i] :< (1<<j);
    histo:[i:] = tmp[i]:[i:];}
```

But we can do still better. Every processor could add its local value and then pass it on to its neighbor, which will add its local value while the first one adds the next value. Since the :< operator takes only one clock cycle we don't lose any time at all. Then we find the fastest code possible:

```
for(i=0;i<256;i++)
  histo = :<( histo + tmp[(PENUM+i)&255);
```

Performing execution timing in the debugger gives conclusive results [5,6]:

- classical histogram: 60.10 ms
- using prefix addition: 2.91 ms
- optimal program: 0.29 ms

Another class of image processing operation is the recursive neighborhood operations (RNO). A few examples of RNO are the morphological operations like the opening, closing, thinning or distance transform. For updating each pixel, RNOs refer the pixel value of its neighborhood pixel, which have already been updated. A parallel method for efficient implementation of RNO can be used [7]. We use overlapping waves where each wave is a line of activated PE. PE are activated successively in a fixed direction, while each activated PE updates the corresponding pixel after every fixed time interval. A distance transform implementation in 1DC will take 8ms on an image of dimension 256 x 240.

The distributed bucket processing [8,9] (stack-based) method is extensively used. It consists of two processing phases during which every PE of the LPA simulates a software stack in its local memory. In the first phase, all pixels are visited once in order to find points with specific features such as contour pixels or peak pixels and push them into the stack. In the second phase we pop them and process them. In the processing phase, we can push them back in the stack corresponding for instance to an object contour from where they belongs (contour tracking operation) or we can use them to accumulate votes like in circular Hough transform. We further process all the pixels from the stack till all interest PE stacks are empty. A circular Hough transform implemented using this method will perform in 6 to 11ms. This image transform is used to detect in real time the ball in robot soccer game using the vision system.

3 The MMX Technology

A set of MMX routines that allow us to compare the MMX SIMD and IMAP-VISION system has been developed; they include some low-level and some intermediate level image processing routines. MMX technology for accelerating multimedia applications has become quite commonplace in the recent months. Many of the core requirements of multimedia processing overlap with industrial machine vision requirements, and so it is natural that the vision community benefits from this new computational capacity.

Intel's MMX Technology adds several new data types and 57 new instructions specifically designed to manipulate and process video, audio and graphical data more efficiently. These types of applications often use repetitive loops that, while occupying 10 percent or less of the overall application code, can account for up to 90 percent of the execution time. A process called Single Instruction Multiple data (SIMD) enables one instruction to perform the same function on multiple pieces of data.

The new data types allow handling of 64-bit data. This is accomplished by reassigning 64 bits of each of the eight 80-bit floating-point registers as MMX register. The 64-bit registers may be thought as eight 8-bit bytes, four 16-bit words, two 32-bit double words, or one 64-bit quadword [1].

A. MMX Optimization Issues

MMX coding currently implies the use of Assembly language. We will focus this discussion on a couple of strategies which are impose by the architecture and that guarantee the highest performance. *Data alignment*: a misalignment access in the data cache or on the bus cost at least three extra clock cycles on the Pentium processor. This is a serious performance penalty if we think that an aligned memory access might take only 1 cycle to execute. An 8-byte datum should be aligned on an 8-byte boundary. Also if the code hushes MMX registers into stack, it is necessary to replace the entry and exit code of the procedure to ensure that the stack is aligned too [2]. Data alignment is very important when writing MMX code because the execution speed can be boosted by more than 30%.

Instruction Scheduling: to get the most speed out of MMX, we have to think in term of instruction scheduling. The most critical involve output operand collision. The Pentium processor is an advanced superscalar processor. It is built around two general-purpose integer pipelines and a pipelined floating-point unit, allowing the processor to execute two integer instructions simultaneously. The first logical pipe is the U-pipe, and the second is the V-pipe. During decoding of an instruction, next two instructions are checked, and, if is possible, they are issued such that the first one executes in the U-pipe and the second in the V-pipe. If this is not possible, only one instruction will be issued to the U-pipe and no instruction is issued to the V-pipe. That means that the execution of two instructions in one clock cycle might double the performance. For this reason is advised to keep both pipes busy. But we are limited by the hardware because the Pentium has only one shift register, only one multiplier, and only U-pipe can execute instructions that access the memory or integer register. A short summary:

- Two MMX instructions which both uses MMX multiplier unit (pmull, pmulh, pmadd) cannot pair. Multiply operations may be issued in either U-pipe or V-pipe but not in the same cycle.
- Two MMX instructions, which both uses MMX shifter unit (pack, unpack, and shift instruction), cannot pair.
- MMX instructions, which access either memory or integer, register can be issued in the U-pipe only.

B. Tuning MMX code

This section presents the loop variables reduction, loop unrolling and loop interleaving techniques. One optimization is to consider elimination of an inner loop. For loops are complex structures requiring counter variable. Counter variables place an additional strain on the CPU register pool, which in the Intel architecture is uncomfortably small. Using a pointer increment with an end-of-line limit greatly increases speed since the incrementing pointer is also the counter loop. Usually the MMX code consists of short loop iterated many times over an array of data. Often there is no relationship between two iterations of the same loop, so the output of the second iteration does not depend on the output of the first. Having calculated the highest number of iteration that can be executed in parallel (N), the loop can be unrolled by N and then instructions can be moved around to maximize the use of both pipelines:

- MMX instructions that access memory are always executed in the U-pipe, so it is better to spread them around and try to exploit the V-pipe with ALU or shift instructions.
- Two shift or multiply instructions cannot be issued in the same cycle. We can introduce a small delay between the iterations, so that the group of MMX instructions that are physically near belong to different stages of their respective iterations.

For operations on binary images like noise filter on binary image, morphological operations or measurements on binary images we can pack the image. In this way 64 pixels will fit in a single MMX register and if we can gain advantage from loop interleaving optimization and issue 2 operations in one cycle, we can process 128 pixels at once. Since a typical image is 256 x 256, the use of one bit per pixel shrinks the output data down to 8k bytes, allowing a more efficient use of the Pentium's L1 cache memory. Unfortunately this packing process requires quite a lot of instructions to be implemented, and an additional phase of bit swapping. Also careful instruction scheduling is required due to the limitations on the use of shift instructions: only one shift operation can be issued in each clock cycle.

3.1 MMX Image Processing implementation and benchmarks

An interesting problem is how to measure the execution time of MMX routine. This could be done using the information from RDTSC (real time stamp counter) which contains the cycle counter. The detailed description of the RDTSC counter may be find in [3]. The measured timing is approximate and depends on many factors as OS overheads, number of processes running, cash situation if MMX code contains read/write instructions, etc. Various conditions, as cache warming prior reading or writing from/to the same memory blocks, a particular write strategy implemented in the processor and L2 cache, most significantly affect the performance. For that reasons we need to carefully consider the results and run multiple test and average out the results excluding the values far from the mean, which may occur due to a particular running condition.

4 Implementation and Results

Due to the succinct language design and the RISC like instruction-set of IMAP-VISION, the IDC compiler has achieved codes competitive with hand-written assembly code as can be seen in table 1.

Algorithm	Assembly code steps	Compiler code steps	Ratio
Average filter	5600	6430	1.15
Histogram	4039	4872	1.21
Rotation 90 deg.	20696	23326	1.13

Table 1. 1DC compiler performance

Because of this we will use in comparison the code written in 1DC. The assembly should be used only to optimize the most time consuming parts of the code.

Also for complicate algorithms we used MMX technology intrinsics developed by Intel. Intrinsics are highly optimized routines written in assembly, from which the compiler generates inline code. The intrinsics allow for relative quick prototyping of code and is much easier to maintain than assembly. On the other hand only Intel compiler supports this technique and the performance is about 15-25% slower than equivalent assembly code.

In table 2 we make a comparison of the execution times between MMX code on a single Pentium II 300MHz processor and 1DC on IMAP-VISION system. We used in our measurements a 256 x 256, 8 bits per pixel image.

Operation Type	MMX PII 300MHz	IMAP-VISION
Image Binarization	0.3ms	0.036ms
Images add	0.5ms	0.1ms
Image mean	0.9 ms	0.04ms
Image Multiplication	1.1ms	0.09ms
Images Bit And	0.8ms	0.07ms
Convolution 3x3 kernel	5.5ms	0.42ms
Sobel Edge Detection	2.4ms	0.4ms
Image Variance	14.8ms	0.65 ms
Histogram	10,6ms	0.29ms
Dilation	12,4ms	0.31 ms

Table 2. MMX versus IMAP-VISION timings

5 Conclusions

Low level image processing performs very well on a single MMX processor architecture. The IMAP-VISION still performs in average 10 times faster, mostly because it has 256 PE. The drawback of the IMAP-VISION system is the 40MHz operating frequency. Also, both systems have a major drawback. The IMAP-VISION system has no floating point operations, and within MMX code we cannot use floating point (FP) operations either (in fact we can, but the cost is 50 processor cycles to switch between FP mode and MMX mode).

In this study we do not take into account the acquisition and the transfer time. We can mention that we can expect that the IMAP will perform far better because the bandwidth between on-chip and external memory is 1.28Gyte/s, which enables the transfer of 644 frames in 33.3ms (the NTSC frame rate).

While the IMAP-Vision system performs better because of the large number of processing elements, the MMX processor remains a good candidate for low-level image processing, if we take into account also the price.

ACKNOWLEDGMENTS

This work was done in the framework of Ubiquitous Communications Program (www.ubicom.tudelft.nl).

REFERENCES

[1] Intel Corporation, Intel Architecture MMX Technology Developer's Manual, Intel Corporation 1997. Available at http://www.intel.com.

[2] Intel Corporation, MMX Technology Programmers Reference Manual, Intel Corporation 1997. Available at http://www.intel.com.

[3] Intel Corporation, Using the RDTSC Instruction for Performance Monitoring, Available at http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM.

[4] Y. Fujita et al., IMAP-VISION: An SIMD Processor with High-Speed On-chip Memory and Large Capacity External Memory, Proc. of IAPR Workshop on Machine Vision Applications (MVA), pp.170-173, 1996.

[5] S. Kyo and K. Sato, Efficient Implementation of Image Processing Algorithms on Linear Processor Arrays using the Data Parallel Language 1DC, Proc. of IAPR Workshop on Machine Vision Applications (MVA), pp.160-165, 1996.

[6] M. van der Molen and S. Kyo, Reference guide for the IMAP-VISION assembly language, NEC Incubation Center, 1997.

[7] P. P. Jonker, Architectures for Multidimensional Low- and Intermediate Level Image Processing, Proc. of IAPR Workshop on Machine Vision Applications (MVA), pp.307-316, 1990.

[8] J.G.E. Olk and P.P. Jonker, Bucket processing: A paradigm for image processing, ICPR13, Proc. 13th Int. Conf. on Pattern Recognition (Vienna, Austria, Aug.25-29) Vol. 4, IEEE Computer Society Press, Los Alamitos

[9] J.G.E. Olk and P.P. Jonker, Parallel image processing using distributed arrays of buckets, Pattern Recognition and Image Analysis, vol. 7, no. 1,1997