Implementation of Convolution Operation on General Purpose Processors

Ernest Jamro, Kazimierz Wiatr

AGH Technical University, Institute of Electronics

Mickiewicza 30, 30-059 Krakow, Poland

tel. +48 12 6173033, fax +48 12 6332398, email: wiatr@uci.agh.edu.pl

Abstract

Convolution (FIR filtering) is a computationally demanding operation, especially when performed as a two-dimension (2D) operation in a real time image processing system. Consequently, a great amount of research has been done to perform this operation more and more efficiently. This paper reviews different architectures of general-purpose processors giving the example of Pentium family. In recent years a rapid development of microprocessors have been observed, however, it can be seen that architecture of microprocessors is already complex and a further rapid increase of the calculation speed rather limited. Dedicated processors implement convolution operation very efficiently, nevertheless, the design is not flexible which makes design development, test and upgrade difficult.

Topics: *image processing, real-time vision systems*

1 Introduction

Convolution may be a computationally demanding operation. For example, for image parameters: resolution *512×512* (N_X*= 512,* N_Y*= 512*), number of frames N_F*= 25/s* and kernel size $N \times M$*= 3×3*, a real-time image convolution [Cas96, Gon87, Wia97] requires $L_M = N_X \cdot N_Y \cdot N_F \cdot N \cdot M =$ *58 982 400* multiplies and $L_A = N_X \cdot N_Y \cdot N_F \cdot (N \cdot M-1) =$ *52 428 800* additions per second. Such the amount of operations is a challenge for nowadays' architectures. Furthermore, the parameters of the convolution can change; e.g. the size of the convolution kernel 3×3 is one of the smallest and often larger kernels are adopted, e.g. 7×7. Correspondingly, the image resolution and the refresh rate can increase [Wia97], which significantly increases computational requirements of the convoler.

Convolution, in spite of its mathematical simplicity, is a great challenge for engineers because of its computational requirements. Two-dimensional convolution (or a 2D FIR filter) is specified as follows:

$$b_{y+N/2,\ x+M/2} = \frac{1}{D}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1} w_{i,j} \cdot a_{y+i,x+j} \qquad (1)$$

where: N, M – the size of the convolution kernel (usually odd numbers), $a_{y,x}$*- an input,* $b_{y,x}$ *– an output,* $w_{i,j}$*- a coefficient of the convolution, D- a common denominator.*

In this paper, all variables in eq. 1.1 are integers, which significantly simplifies the architecture of the convoler. This assumption seldom confines the filter characteristic which can be adjust by a proper change of the coefficients value and the value of the common denominator *D*. Furthermore the denominator *D* is a power of two, consequently the division can be substituted by a bit shift. In some solutions, even coefficients $w_{i,j}$ are a power of two [Gon87, Cas96, Russ95] and multiplication can be substituted by addition. The examples of these filters are given in Table 1-1.

a) D=16

1	*2*	*1*
2	*4*	*2*
1	*2*	*1*

b) D=1

-1	*-2*	*-1*
0	*0*	*0*
1	*2*	*1*

c) D=1

1	*1*	*1*
1	*-8*	*1*
1	*1*	*1*

Table 1-1. Examples of standard image processing convolution kernels which do not require multiplication: a) low-pass b) Sobel gradient c) Laplacean edge detection

It should be noted that the convolution is a common operation and the example of the real-time image convolution is given hereby to illustrate the problem. For example, a convolution operation is often performed for prediction in artificial neural networks [Meh97]. Besides arithmetic operations for convolution and neural networks are similar. In conclusion, there is a great pressure to produce faster and faster convolers to cope with new, more computationally demanding requirements. Consequently, this paper is an overview of architectures that are capable of performing the real time image convolution.

2 General purpose processor

The most commonly used general-purpose processor is family of 8086 [And95, Bre97]. This processor has a complex architecture which is not optimal for convolution, however it is commonly used and therefore can be quickly and easily adopted as a convolution processor.

To illustrate the convolution process on the general-purpose processor, an example of C-language procedure is given in Listing 2-1.

```
const int M= 3, N= 3; //  the size of the convolution kernel: M- horizontal; N- vertical
const int Nx= 512, Ny= 512; // image resolution: Nx- horizontal, Ny- vertical
const int D= 16; // the common denominator (see eq. 1.1)
BYTE a[Ny+1+N][Nx]; // the source image which has been enlarged to eliminate padding effect. The actual image
      is from line 1+N/2 to Ny+N/2. The rest of image a is specified to minimise the padding effect.
BYTE b[Ny][Nx]; // the destination image
int w[N][M]={1,2,1, 2,4,2, 1,2,1}; // coefficients of the convolution given in Table 1-1a

void
convol2()
{   BYTE *pa= &a[0][Nx-M/2]; // the pointer to the source pixels (points top-left pixel of the convolution window)
    BYTE *pb= &b[0][0];  // the pointer to the destination pixel
    for(int y= 0; y<Ny; y++)  // for every line of the image
    {   for(int x=0; x<Nx;  x++, pb++, pa++) // for every pixel in the line
        {   register BYTE *pw=w[0]; // the pointer to the coefficients.
```

```
            register BYTE *pa1=pa; // pointer to the current source pixel
            register int sum= D/2; // accumulation result – initially D/2 to minimise division rounding error
            for(int i= 0; i<N; i++)  // vertical convolution
            {   for(int j=0; j<M; j++) // horizontal convolution
                    sum+= *pw++ * *pa1++; // the kernel of the convolution
                pa1+= Nx-M; // pa1 will point the first pixel in the next line
            }
            sum/= D; // division by the common denominator (D is a power of two so it is substituted by a bit-shit)
            *pb= (BYTE) sum; // conversion from int (4 bytes) to 1 byte variable, save the result.
        }
    }
}
```

Listing 2-1. C-language procedure for the convolution.

Listing 2-1can be further optimised by the following procedures:

1. the loops unrolling
2. rewriting the convolution procedure in the assembler language
3. rewriting the assembler language procedure with respect to the superscalar architecture of the Pentium processor
4. employing MMX processor and its Single Instruction–stream Multiple Data-stream (SIMD) [Fly66] architecture

2.1 Loop unrolling

In most cases, the size of the convolution kernel is fixed and therefore the convolution loops (loops: *i* and *j* in Listing 2-1– horizontal and vertical part of the convolution) can be easily unrolled by writing down $N \times M$ times the multiplication and addition operations. The loop instruction contains two assembler instructions:

- *dec*: decrement the loop counter
- *jnz*: conditional jump.

The benefit from the loop unrolling is not only fewer instructions to be executed but also fewer processor stalls. The stalls are caused by the conditional jump instructions which interfere with pipeline architecture of the processor [Mad95, Hwa93]. The pipelining effects that every instruction is executed partially in subsequent clock cycles. For example, Pentium 75 executes an instruction in 5 stages [And95]:

1. fetching
2. decoding (stage 1)
3. decoding (stage 2)
4. execution
5. updating registers

As a result of pipelining, the branch is finally executed in the last stage, therefore, instructions following the branch shall not be executed unless the branch is not taken. Consequently in 80486 processors, these instructions were not fetched until the branch instruction was finally executed. This caused the processor stalls. To improved processor performance in Pentium, branch prediction together with Branch Prediction Buffer (BPB) [And95, Int97a] have been introduced. This allows the processor to decode instructions beyond branches to keep the instruction pipeline full [Int97b]. The drawback of the branch prediction is that branches are

predicted incorrectly with a certain probability, $p>0$. Each misprediction causes a restart of the pipeline, which has similar effects as not fetching the instructions until the branch is executed.

Therefore to decrease misprediction ratio, Pentium 75 uses two up-down counter with saturation to keep track of the direction a branch is more likely to take [Smt81, And95]. Taking into account the convolution process (Listing 2-1) and the above branch prediction procedure, an assumption can be made that the processor will predict the loop to be executed infinitely, therefore every loop-braking causes the processor stalls. It should be noted that penalty for misprediction is even greater for the latest processors as the number of pipelining stages is increasing, e.g. Pentium Pro has 12 stages [Int97b]. This is confirmed by implementation results presented in Table 2-1 where $t_a/t_b = 1.42$ for P75, and 2.22 for Athlon 800MHz; where: t_a - calculation time without loop unrolling, t_b - with loop unrolling. The above prediction scheme is a basic one and nowadays more sophisticated branch prediction procedures have been demonstrated, e.g. [Eve98] where branch prediction scheme can detect loops and an additional loop counter is included.

It should be noted that loop unrolling not only decreases the number of instructions to be executed, eliminates branches and branch misprediction effects but also improves instruction level parallelism, which will be approached in the next section.

2.2 Superscalar architecture

In a superscalar processor [Hwa93] multiple instructions pipelines are used. This implies that multiple instructions are issued per cycle and multiple results are generated per cycle. Superscalar processors are designed to exploit more instruction level parallelism in a user program.

The superscalar architecture was introduced in Pentium 75 (P75) [And95] which incorporates two parallel integer processing units:

- Integer unit *U*
- Integer unit *V*

The number of parallel units has increased in the latest processors. For example, Pentium Pro incorporates three-way superscalar architecture [Int97b], Pentium 4 incorporates Rapid Execution Engine [Int00] for which the ALUs run two times the frequency of the processor core.

Taking into account the P75, two integer instructions can be executed in a single clock cycle. However, some instructions cannot be executed in parallel, e.g. ‘V’ unit cannot execute shift instructions or two multiplication cannot be executed in parallel [And95, Int97a]. Besides, some instructions cannot be executed in parallel because of register contention, e.g. the result of a ‘U’ instruction which is currently executed, is input to a ‘V’ instruction.

In conclusion, seldom all units of the superscalar architecture are fully exploited. Only independent instructions can be executed in parallel without causing a wait state and therefore the superscalar processor depends strongly on an optimising compiler to exploit parallelism.

As a result of the above conclusions, Listing 2-2b shows an assembler code which better exploits the superscalar architecture of the P75. Nevertheless, optimised code (Listing 2-2b) is executed only 10% quicker (see Table 2-1 for the P75) than non-optimised code (Listing 2-2a). This is as the multiplication is a complex instruction which requires several clock cycles to be executed and cannot be carried out in parallel. Besides non-optimised code can somehow exploit the superscalar architecture of the processor. For convolution filters without multiplication the optimised code is executed 25% quicker.

a)

```
// pixel 3 (top-right pixel in the convolution window)
xor edx,edx // clear edx
mov dl, byte ptr [ecx+2] // load data a (pixel 3)
imul edx,dword ptr [edi+8h]//multiply: pixel3 * w[0][2]
add eax, edx  // accumulate the result of the multiplication
// pixel 4 (left middle pixel)
xor edx,edx // clear edx
mov dl, byte ptr [ecx+200h] // load pixel 4
imul edx, dword ptr [edi+0Ch] // edx= pixel4 * w[1][0]
add eax, edx // accumulate the result of the multiplication
```

b)

```
xor edx, edx //start of calculation for pixel 3: clear
imul ebx, dword ptr [edi+4] //pixel2: ebx=pel2*w[0][1]
mov dl, byte ptr [ecx+2] // pixel 3: dl=pel3
add eax, ebx // end of calculation for pixel2: eax+= ebx
imul edx, dword ptr [edi+8]  // pixel 3: edx=pel3*w[0][2]
xor ebx, ebx // start calculation for pixel 4: clear
add eax, edx // end of calculation for pixel 3: eax+= edx
mov bl, byte ptr [ecx+200h] // pixel 4: bl= pel4
```

Listing 2-2. A fragment of the 3×3 convolution (Table 1-1a) assembler code for a) scalar, b) superscalar architecture.

To further exploit the superscalar architecture, speculative execution has been introduced in the latest processors, which allows for out-of-order execution, and therefore a code is internally optimised for the superscalar architecture of a particular processor. Consequently, a programmer need not optimise a code every time the number of parallel units increases. Nevertheless for speculative execution and three parallel units, the calculation time is shorter (up to 8%, see Table 2-1) for the two-way optimised than non-optimised code. This is probably because the speculative execution algorithm is not perfect and the optimised code is easier to be executed out-of-order. Besides the number of instructions in the loop has been reduced from 37 to 35 for optimised code.

It has been observed that the average value of instructions executed in parallel is around 2 for code without loop unrolling [Hwa93]. Even with loop unrolling, instruction-issue degree in a superscalar processor has been limited to 2 to 5 in practice [Hwa93, Tul95]. Let consider the case of the Pentium processors, it can be seen from Table 2-1 and Table 2-2 that average number of instructions executed by the P300 and Athlon 800 in a single clock cycle is up to the 2.3; for P166 it is about 1.17 (option without multiplication). It should be however noted that the improvement has been also achieved by reducing the number of clock cycles required to perform the multiplication. In conclusion, superscalar architecture quickly gets saturated, i.e. increasing the number of parallel units requires much greater hardware expense but causes little improvement.

An alternative solution is simultaneous multithreading, a technique permitting several independent threads to issue instructions to a superscalar's multiple functional units in a single cycle [Tul95]. This techniques allows better utilisation of superscalar units as different threads can issue their instructions in such a way that register contentions, memory miss or conflict and

even branch misprediction penalty is significantly reduced. For example, while one thread waits for data transmitted from external memory, others can issue their instructions to keep all processing units busy.

2.3 Very Long Instruction Word (VLIW)

The superscalar architecture requires complex instruction decoding, dispatching and speculative execution units. An alternative solution is the VLIW [Hwa93] architecture for which different fields of the long instruction word correspond to different functional units and therefore decoding and dispatching instructions is much easier. Unfortunately, a VLIW code has to be recompiled for a specified machine. Furthermore, for a superscalar processor the code density is greater as the fixed VLIW format includes bits for nonexecutable operations, while the superscalar processor issues only executable instructions.

The VLIW architecture is seldom employed in general purpose processors as tasks of the processors are unknown and therefore it is difficult to optimise functions of the processors units. On the contrary, the convolution operation is well defined and requires basically four (six - if processor does not support addressing with offset) different operations:

- load the coefficient (and increment the coefficient pointer),
- load the input pixel (and increment the input pixel pointer),
- multiply,
- accumulate.

Consequently, these four (six) operations can be executed in a single VLIW instruction in DSPs which are optimised for digital signal processing.

An interesting solution has been implemented in Crusoe processor by Transmeta [Kla00]. This processor has a VLIW architecture which is able to perform 3 integer and one floating point operations in parallel (see Figure 2-1).

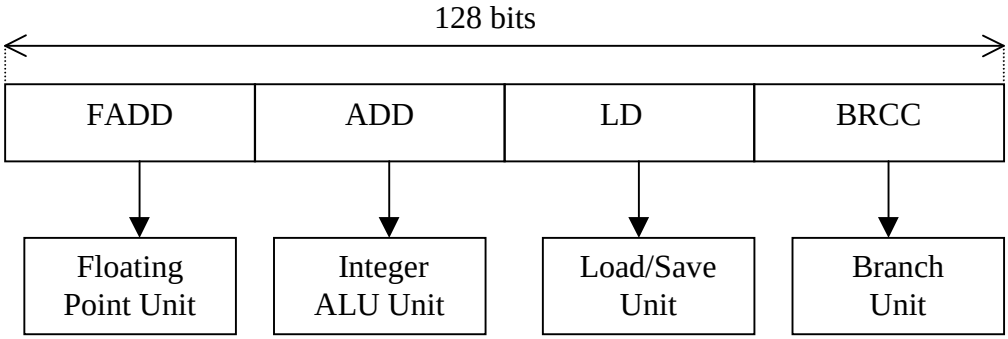

Figure 2-1. Crusoe processor can execute up to four operations in parallel.

In comparison to the Pentium processor, Crusoe processor does not require the complex decoding and dispatching module. Therefore, it requires simpler hardware and consumes much less power. Furthermore, Crusoe processor incorporates dynamic translation system, denoted as Code Morphing which complies the x86 instruction set into the host VLIW instruction set. The processor can therefore run standard x86 programmes because code translation is invisible to the external system. It should be noted that Pentium processors decode and dispatch instructions

every time they are executed. Conversely, Transmeta's software translates instructions once, saving the resulting translation in a translation cache [Kla00]. The next time the (now translated) x86 code is executed, the system skips the translation step and directly executes the existing optimised translation. As most code is executed several times in a loop, translation overheads have little influence on the system performance. Furthermore, the translation is carried out only once and therefore it can implement a complex algorithm which better optimises code than e.g. the Pentium processor does. Besides, as an application is executed, Code Morphing 'learns' more about the program and improves it so it will execute faster and faster. For example, aware of the branch history, the programs can favour the most frequently taken path, or execute code from both paths and select the correct result later if both paths are taken with equal probability. It is important to note that Crusoe hardware can achieve excellent performance because it has been designed specifically with dynamic translation in mind.

New Pentium 4 processor adopted a similar but much simpler instruction decoding solution. The hardware instruction decoder can decode maximum one instruction per clock cycle. The decoded instructions are stored in the execution trace cache (TC) [Int00] and then are executed directly from the TC. This removes decoding costs on frequently-executed code. In Pentium 4 processor implementation, the TC can hold up to 12K µops and can deliver up to three µops per cycle.

2.4 SIMD

A single Instruction stream over Multiple Data stream (SIMD) [Fly72] architecture allows a single instruction to be executed on several independent data simultaneously. This significantly simplifies the instruction decoding process, as only a single control unit is required.
Consequently, in the Pentium processor, a MultiMedia eXtension (MMX) [Int97b] coprocessor has been introduced which operates on 64 bits data and therefore can process eight independent 8-bit-wide data simultaneously. In the case of the image convolution, input pixels are 8 bits unsigned data, however intermediate results are 16-bit wide, and therefore up to four data can be processed simultaneously. Examples of MMX instructions are given in Figure 2-2.

a)

MM0	S	T	U	V
	×	×	×	×
MM1	W	X	Y	Z
=	=	=	=	=
MM0	S·W	T·X	U·Y	V·Z

b)

MM0	S	T	U	V
	×	×	×	×
MM1	W	X	Y	Z
=	=	=	=	=
MM0	S·W+T·X		U·Y+V·Z	

Figure 2-2. Example of MMX instructions: a) multiplication PMULLW MM0, MM1, b) multiply and accumulate (MAC) PMADDWD MM0, MM1

Additional computation power is obtained by superscalar architecture of the MMX coprocessor, as two MMX instructions can be executed in parallel provided that different MMX resources are employed.

Convolution operation can be carried out in two different ways. In the first one, given in Listing 2-3m only one result is obtained at the time. This solution exploits MMX multiply and accumulate (MAC) instruction (PMULLW), therefore it might seen that this is the best solution. Unfortunately, input data format causes that every MAC operation performs four multiplies (for every input row) but only three are used. Besides two partial results are obtained in two halves of the MMX register, and therefore integer units have to be used to carry out the final addition. The alternative solution is presented in Listing 2-3n, for which four results are obtained simultaneously. This option carries out multiply and add instructions independently, and input and output data better suit the convolution process and therefore all instructions except the loop instructions are carried out employing only MMX instruction which are fully exploited. The latest solution reduces calculation time in comparison to the former solution (see Table 2-1). It should be noted that option *n* allows for saturating the result (e.g. setting the output to the maximum value *(255)* if the result is overflowed *(≥256)*) during conversion from the word to byte format.

m)

```
beg:  // label for loop start
movd mm0, dword ptr [ecx] // load row 0
movd mm1, dword ptr [ecx+200h]  // load row 1
movd mm2, dword ptr [ecx+400h]  // load row 2

punpcklbw mm0, mm5 // convert byte to word
punpcklbw mm1, mm5 // mm5 – contains only zeros
punpcklbw mm2, mm5

pmaddwd mm0, mm6 // MAC; mm6= 0, 1,2,1
pmaddwd mm1, mm7 // mm7= 0,2,4,2
pmaddwd mm2, mm6

paddd mm0, mm1 // accumulating the MAC results
paddd mm0, mm2 // result in mm0
// integer units operations
movq [edi], mm0 // save register MMX to memory
movd eax, mm0 // load LSB half of the register MMX
add eax, [edi+4] // add LSB and MSB half of the register MMX
add eax, 8 // add 8 to reduce rounding error
inc esi // increment pointer to the destination
sar eax, 4 // division by 16 – prescaling
inc ecx // increment pointer to the source pixel
mov byte ptr [esi], al. // load the result to memory
dec ebp  // decrement the loop count
jnz beg // finish the loop?
```

n)

```
beg: // label for loop start
movd mm0, dword ptr [ecx] // load pixel (0,0)
punpcklbw mm0, mm7 // convert byte to word; mm7= 0

movd mm1, dword ptr [ecx+1] // load pixel (0,1)
pmullw mm0, mm6 // multiplication for pixel (0,0), mm6= 1,1,1,1
punpcklbw mm1, mm7 // convert byte to word, pixel (0,1)

movd mm2, dword ptr [ecx+2] // load pixel (0,2)
pmullw mm1, mm5 // multiplication for pixel (0,1); mm5= 2,2,2,2
punpcklbw mm2, mm7 // convert byte to word for pixel (0,2)
paddw mm0, mm1 // add products for pixels (0,0) and (0,1)
// continue for the rest of pixels
...
pmullw mm2, mm6 // multiplication for pixel (2,2), mm6= 1,1,1,1
paddw mm0, mm2 // the final result

paddw mm0, mm3 //add to reduce rounding error, mm3= 8,8,8,8
psrlw mm0, 4 // divide by 16
packuswb mm0, mm7 // convert the result from word to byte
add ecx, 4 // increment source pixels pointer
movd dword ptr [esi], mm0 // save the result
add esi,4 // increment the result pointer
dec ebp  // decrement the loop count
jnz beg // quit the loop?
```

Listing 2-3. Fragments of 3×3 convolution programs for different options and convolution kernel given in Table 1-1a

The calculation time when the MMX coprocessor is employed, is significantly reduced especially for the P166 (by 75%, see Table 2-1). For the latest processors, however, more sophisticated superscalar integer units are incorporated, therefore the speed-up is less significant.

New Pentium 4 can operate on 128 bits-wide data using SSE2 instructions which are similar to the MMX instructions [Int00]. Consequently, the speed-up by the use of SIMD instructions will be even greater, and it seems that in the future, new releases of processors will be able to process greater and greater data width in SIMD instructions.

2.5 Implementation results

Table 2-1 and Table 2-2 gives implementation results for different processors and different options. The following convolution options have been implemented:
a) standard algorithm written in C language (Listing 2-1),
b) after unrolling the convolution kernel (loops *i, j* in Listing 2-1) written in C language,
c) like option *b* but program is written directly in assembler language,
d) like option *c* – after optimisation for the superscalar architecture,
e) like option *c* but without multiplication (only shifts are implemented, the convolution kernel is given in Table 1-1a),
f) like option *e* – after optimisation for the superscalar architecture,
g) like option *f* but input and output data pointers are not incremented to avoid cash misses,
m) employing MMX coprocessor (Listing 2-3m),
n) employing MMX coprocessor, four pixels are calculated simultaneously (Listing 2-3n).

All options, except option *g*, have been referred in the previous sections. Therefore only option *g* will be now approached. It can be seen from Listing 2-2 and Listing 2-3 that approximately every second instruction communicates with memory and consequently memory-transfer might be a bottleneck of the system. Fortunately, all tested processors incorporate internal cache memory [Hwa93, And95], which significantly reduces external memory transfers. Nevertheless, cache misses might still cause deterioration of the processor performance. Consequently, in option *g* input and output data pointers are not incremented and therefore the convolution operates only on 9 input and 1 output pixels. This causes that the result of the convolution is corrupted; conversely the external memory transfer is not needed. Option *g*, in comparison to the corresponding option *f*, gives up to 10% improvement. For the latest versions of the processors, however, the drawback of the cache miss is increasing (see Table 2-1). It should be noted that SSE instructions allow software controlled data-perfecting, which can eliminate the cache misses.

Option	A	b	c	d	e	f	g	m	n
Number of instructions in the loop	-	-	42	42	37	35	35	21	43/4
Time [ms] 486DX4 –100	670	465	460	435	147	134	131	-	-
P75	587	414	403	366	95	70.3	67.6	-	-
P166	247	175	169	159	45	32	30	60	26
P300	48.6	25.8	22.6	22.8	16.6	15.6	14.0	22.1	9.1
Athlon 800MHz	25.8	11.6	10.4	9.9	6	5.5	5	12.6	7.2

Table 2-1. Number of assembler instruction in the loop and calculation time for different processors and options

Option	a	B	c	d	e	f	g	m	N
486DX4 –100	256	177	175	166	56	51	50	-	-
P75	168	118	115	105	27.2	20.1	19.3	-	-
P166	156	111	107	101	28.5	20.3	19.0	38.0	16.5
P300	55.6	29.5	25.9	26.1	19.0	17.9	16.0	25.3	10.4
Athlon 800MHz	78.7	35.4	31.7	30.2	18.3	16.8	15.3	38.5	22.0

Table 2-2. Number of clock cycles required to calculate a single output pixel for different processors and options

3 Conclusions

A general-purpose processor, in spite of its complex architecture, is the easiest solution for implementation of the convoler because the processor and its development environment are commonly available. Conversely, optimisation of the convolution code requires assembler and system level programming which knowledge is limited. However the primary drawback of the general-purpose processor is tasks sharing; the convolution operation may interfere with other tasks and vice versa. Besides in spite of its rapid speed-up, the processor is still not able to process large convolution kernels and image resolutions.

Architecture of microprocessors begins to saturate; a significant increase of hardware complexity results in a much less significant seed-up. Therefore in the future, further development of the complex superscalar processors is unlikely. Instead many processors will operate in parallel or a simultaneous-multithreading processor will be introduced. This however will increase the demand for cache memory which occupies significant chip area. Furthermore, parallel processing has several drawbacks like memory access contention, multiple-threads synchronisation, etc. [Hwa94], which significantly complicates the architecture and programming of the parallel system. Besides, obtained speed-up is often not proportional to the number of additional parallel units.

Field Programmable Gate Arrays (FPGAs) solution is an alternative to the microprocessors. FPGAs are more and more commonly implemented in regions originally reserved for DSPs or even ASICs. Furthermore FPGAs' density grow surpasses the counterpart grow. FPGAs are very scalable on highly concurrent tasks. Furthermore, taking into account the silicon area and throughput, the FPGAs significantly outperform microprocessors [DeH98], and in the future the performance gap will further increase. Microprocessors must confront overheating effect, which significantly constrains design of the microprocessors. For FPGAs, however, this problem is much less significant, and in most cases, is not considered at all.

It should be noted that FPGAs density increases very rapidly (about 10 times in two years) and FPGAs expand much quicker than the microprocessors do. According to Xilinx Inc., the FPGA convolution processor is capable of performing roughly 100 times more MACs per second than DSPs do. In conclusion, FPGAs seems to be the most efficient and prosperous architecture for the future.

References

[And95] Anderson D.: Shanley T. *Pentium Processor System Architecture.* Addision-Wesley 1995

[Bre97] Brey B. B.: *The Intel Microprocessors*. Prentice-Hall 1997
[Cas96] Castleman K. R.: *Digital Image Processing*. Prentice Hall 1996
[DeH98] DeHon A.: *Comparing Computing Machines,* SPIE Conference on Configurable Computing, Technology and Applications, Boston, Massachusetts, Nov. 1998
[Eve98] Evers M., Patel S. J., Chappell R. S., and Patt Y.N.: *Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work,* Proc. of the 25th International Symposium on Computer Architecture, Barcelona, June 1998
[Fly66] Flynn M. J.: *Very High-Speed Computing Systems.* Proc. IEEE, Vol 54, New York 1966, Nr 12
[Gon87] Gonzalez R., Wintz P.: *Digital Image Processing,* Addision-Wesley 1987
[Hwa93] Hwang K.: *Advanced Computer Architecture. Parallelism, Scalability, Programmability* McGraw-Hill Singapore, 1993
[Int97a] Intel Co.: *Intel Architecture Software Developer's. Manual Volume 3: System Programming Guide*. (Order Number 243192). Intel Corporation 1997
[Int97b] Intel Co.: *Intel Architecture Software Developer's. Manual Volume 1: Basic Architecture. (Order Number 243190).* Intel Corporation 1997
[Int00] Intel Co.: *A Detailed Look Inside the Intel NetBurst Micro-Architecture of the Intel Pentium 4 Processor,* Intel Corporation 2000
[Kla00] Klaiber A.: *The Technology behind Crusoe Processors.* Transmeta Co. Jun 2000 http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf
[Mad95] Madisetti V. K.: *VLSI Digital Signal Processors.* Butterworth Heinemann 1995
[Matrox] MATROX: *Matrox MIL-32 Library*. Matrox Electronic Systems Ltd, http://www.matrox.com /imgweb/
[Meh97] Mehrotra K., Mohan C. K., Ranka S.: *Elements of artificial neural networks,* MIT Press, Cambridge, London, 1997
[Omo94] Omondi A.R.: *Computer Arithmetic Systems. Algorithms Architecture and Implementations,* Prentice Hall, UK, 1994
[Russ95] Russ J. C.: *The Image Processing Handbook,* Boca Raton FL, CRC Press 1995
[Smt81] Smith J. E.: *A study of branch prediction strategies,* Proc. of the 8th Annual International Symposium on Computer Architecture, 1981, pp. 135-148
[Tul95] Tullsen D., Eggers S., Levy H. *Simultaneous Multithreading: Maximizing On-Chip Parallelism* Proceedings of the 22rd Annual International Symposium on Computer Architecture, June 1995, pp. 392-403
[Wia97] Wiatr K.: *Dedicated Hardware Processors for a Real-Time Image Data Pre-Processing Implemented in FPGA Structure*. Lecture Notes in Computer Science - no 1311 (Ed. Alberto Del Bimbo), Proc. of the 9th International Conference on Image Analysis and Processing - ICIAP'97, Florence - Italy 1997, Berlin, Springer-Verlag 1997, vol. II, pp. 69-75
[Wia98a] Wiatr K.: *Pipelined Architecture of Specialised Reconfigurable Processors in FPGA Structures for Real-Time Image Data Pre-Processing*. Proc. of the EUROMICRO International Conference: Digital System Design: Architectures, Methods and Tools, Vasteras - Sweden 1998, IEEE Computer Press, Washington-Brussels-Tokyo 1998, pp. 131-138
[Wia98b] Wiatr K.: *Dedicated System Architecture for Parallel Image Computation used Specialised Hardware Processors Implemented in FPGA Structures*. International Journal of Parallel and Distributed Systems and Networks, vol. 1, No. 4, Pittsburgh, 1998, pp. 161-168
[Wia00a] Wiatr K., Jamro E.: *Implementation Image Data Convolution Operations in FPGA Reconfigurable Structures for Real-Time Vision Systems*, Proc. of the IEEE International Conference on Information Technology: Coding and Computing ITCC'2000, Nevada 2000, IEEE Computer Society – Washington – Brussels – Tokyo 2000, pp. 152-157
[Wia00b] Wiatr K., Jamro E.: *Constant Coefficient Multiplication in FPGA Structures.* Proc. of the 26th Euromicro Conference on Digital Systems Design: Architecture, Methods, and Tools, Maastricht, Netherlands 2000, IEEE Computer Society – Washington – Brussels – Tokyo 2000, pp. 252-259

[Wia01a] Wiatr K. Jamro E.: *Implementation of Multipliers in FPGA Structures*, Proc. of the IEEE International Symposium on Quality Electronic Design, San Jose, March 2001