Journal of Electronic Imaging 9(3), 327–335 (July 2000).

Mapping of two-dimensional convolution on very long instruction word media processors for real-time performance

Ravi Managuli
George York
Donglok Kim
Yongmin Kim
University of Washington
Image Computing Systems Laboratory
Departments of Electrical Engineering and Bioengineering
Box 352500
Seattle, Washington 98195-2500
E-mail: ykim@u.washington.edu

Abstract. *Programmable media processors have been emerging to meet the continuously increasing computational demand in complex digital media applications, such as HDTV and MPEG-4, at an affordable cost. These media processors provide the flexibility to implement various image computing algorithms along with high performance, unlike the hardwired approach that has provided high performance for a particular algorithm, but lacks flexibility. However, to achieve high performance on these media processors, a careful and sometimes innovative design of algorithms is essential. In addition, programming techniques, e.g., software pipelining and loop unrolling, are needed to speed up the computations while the data flow can be optimized using a programmable DMA controller. In this paper, we describe an algorithm for two-dimensional convolution, which can be implemented efficiently on many media processors. Implemented on a new media processor called the MAP1000, it takes 7.9 ms to convolve a 512×512 image with a 7×7 kernel, which is much faster than the previously reported software-based convolution and is comparable with the hardwired implementations. High performance in two-dimensional convolution and other algorithms on the MAP1000 clearly demonstrates the feasibility of software-based solutions in demanding imaging and video applications.* © *2000 SPIE and IS&T.* [S1017-9909(00)00203-8]

1 Introduction

To satisfy the continuously increasing demand for higher media quality, tighter integration, new services, and more sophisticated functionality, the multimedia systems and applications require more and more computational power. Many multimedia applications involve real-time filtering, image/video compression, image segmentation, object tracking, etc., which are very compute intensive. To meet this increasing computing demand at an affordable cost, new advanced digital signal processors (DSPs) known as media processors have been emerging, typically with a very long instruction word (VLIW) architecture, e.g., Texas Instruments TMS320C80,[1] Philips Trimedia TM1000,[2] and Hitachi/Equator Technologies MAP1000.[3] A media processor typically has several arithmetic units and load/store units, which can operate concurrently. Many arithmetic units can support partitioned operations, e.g., a 64-bit computing engine can be divided into eight 8-bit units to perform eight operations in parallel. Figure 1 shows an example of *partitioned-add* where eight pairs of 8-bit pixels are added in parallel by a single instruction. Several of the media processors also have an on-chip programmable direct memory access (DMA) controller, which can move the data between on-chip and off-chip memory in parallel with the core processor's computations.

In designing multimedia systems, these programmable media processors offer an alternative to the traditional hardwired approach based on ASICs or fixed-function chips. The media processors allow efficient implementations of various functions while the hardwired approach is limited to a predetermined set of functions. This media processor-based system approach provides developers with the flexibility to optimize their products, upgrade them with new features, and quickly respond to the changing requirements of their customers.

Even with such a flexible and powerful architecture of media processors, to achieve good performance necessitates a careful design of algorithms that can make good use of the newly available parallelism.[4] Generic image computing algorithms developed without any considerations of the underlying media processor architecture often do not produce the desired level of performance. Thus, the programmer needs to understand the architecture well to be able to develop an algorithm for optimal performance. Another important issue for good performance is the efficient data handling. Cache-based processors can incur a large overhead due to data cache misses, which can frequently stall the core processor while waiting for the data to be moved into

Paper 99-037 received June 21, 1999; revised manuscript received Feb. 14, 2000; accepted for publication Apr. 26, 2000.
1017-9909/2000/$15.00 © 2000 SPIE and IS&T.

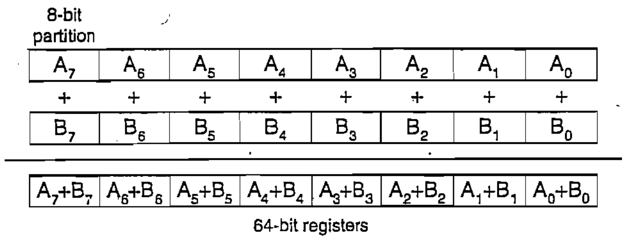


Fig. 1 Example partitioned operation: *partitioned add.*

an on-chip cache. With the careful use of a DMA controller, the I/O time can be hidden behind the computation time, which leads to a significant performance improvement in many algorithms.

In this paper, we will explain how we have utilized the media processor architecture to maximize the performance for one representative image processing algorithm: convolution. Along with the algorithm description, we will present several programming techniques, such as loop unrolling, software pipelining, and *double buffering* of data flow. We will present the performance of convolution on the MAP1000 and compare it with that of other approaches.

2 2D Convolution

In convolution, each output pixel is computed to be a weighted average of several neighboring input pixels. In the simplest form, generalized 2D convolution of an $N \times N$ input image with an $M \times M$ convolution kernel is defined as

$$b(x,y)=\frac{1}{s}\sum_{i=x}^{x+M-1}\sum_{j=y}^{y+M-1} f(i,j)h(i-x,j-y), \tag{1}$$

where f is the input image, h is the kernel, s is the scaling factor, and b is the convolved output image. This is computationally expensive. The total number of operations required, excluding address generation and other operations, are M^2N^2 multiplications, $(M^2-1)N^2$ additions, and $(2M^2N^2+N^2)$ loads/stores. For convolving a 512×512 image with a 7×7 kernel, for example, over 50 million operations are required. Thus, for a large-size kernel, the generalized convolution becomes even more computationally expensive because of the quadratic growth of M^2 in the amount of computations. Since convolution operations are employed in the preprocessing or early processing stage of many imaging applications, the support for fast convolution is essential.

2.1 Convolution Algorithm on Media Processors

All modern VLIW media processors are designed to execute multiply, add, and load/store operations in a single instruction. Furthermore, arithmetic units can perform partitioned operations, such as the *partitioned-add* instruction shown in Fig. 1. Some VLIW processors also feature a very powerful instruction called *inner product*, which in one instruction performs multiple multiplications as well as accumulation of the multiplied results. Equation (2) is an *inner-product* example where x and c are the 8-bit or 16-bit vectors and the multiplied results are accumulated in y:

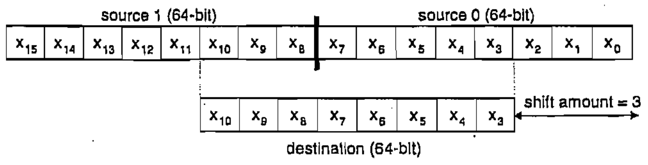


Fig. 2 *Align* instruction to extract the non aligned eight bytes.

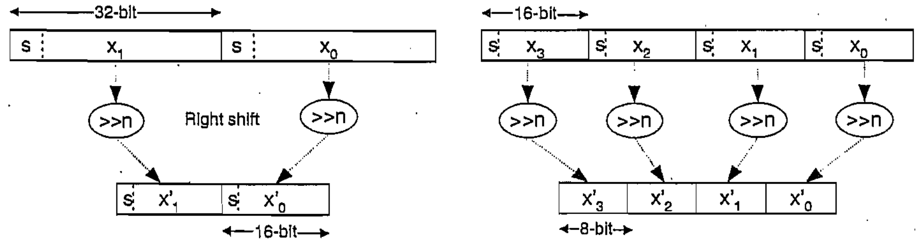
$$y=\sum_{i=0}^{n-1} c_i x_i, \tag{2}$$

where n depends on the number of partitions in the registers. If the register size is 64 bits (8 bytes), then the address of the data loaded from memory into a register holding an input operand (x or c) should be a multiple of 8.[5] When a data word is not aligned, extra overhead cycles are needed in loading two adjacent data words and then extracting the desired input operand by performing shift and mask operations. An instruction called *align* is typically provided to perform this extraction. Figure 2 shows the use of *align*, where the desired nonaligned input operand, x_3 through x_{10}, is extracted from the two adjacent aligned data words (x_0 through x_7 and x_8 through x_{15}) by specifying a shift amount of 3. To optimally use the *inner-product* instruction, the kernel width needs to match the number of partitions in the register that contains the kernel coefficients. If the kernel width is smaller than the number of partitions, then the extra partitions are filled with zeros. If the kernel width is greater than the number of partitions, then the kernel is subdivided into several sections, each section being equal to the number of partitions.

The generalized convolution has one division operation for normalizing the result as shown in Eq. (1). To avoid this time-consuming division operation, we multiply the reciprocal of the scaling factor with each kernel coefficient beforehand and then represent each scaled coefficient in 16-bit sQ15 fixed-point format (one sign bit followed by 15 fractional bits).[6] With this fixed-point representation of coefficients, a right-shift operation is now necessary after the computation of both inner and outer sums in Eq. (1) instead of division. The right-shifted result is saturated to 0 or 255 for the 8-bit output, i.e., if the right-shifted result is less than 0, it is set to 0; if it is greater than 255, then it is clipped to 255; otherwise it is left unchanged.

To perform right shift and saturate operations as well as packing multiple data (so that multiple packed 8-bit output pixels can be stored using a single instruction), two types of compress instructions are available as shown in Fig. 3. *compress1* in Fig. 3(a) packs two 32-bit values into two 16-bit values and stores them into a partitioned 32-bit register while performing the right-shift operation by a specified amount. *compress2* in Fig. 3(b) packs four 16-bit values to four 8-bit values while performing the right-shift operation by a specified amount. After compressing, *compress2* also saturates the individual partitioned results to 0 or 255. Once the multiple results are packed in a single register, then a single store instruction can be utilized to store packed multiple (e.g., eight) output pixels.

The pseudo code for the 2D convolution algorithm with the instructions discussed above is shown below, which



(a) compress1 : packs 32-bit values into 16-bit values while performing the right-shift operation

compress2 : packs four 16-bit values to four 8-bit values while performing the right-shift and saturation operation.

Fig. 3 Partitioned 64-bit *compress* instructions.

generates eight output pixels that are horizontally consecutive. In this pseudo code, we assume that (1) the number of partitions is eight (the data registers are 64 bits with eight 8-bit pixels), (2) the kernel register size is 128 bits (eight 16-bit kernel coefficients), and (3) the kernel width is less than or equal to eight.

```
        for (i = 0; i < kernel_height ; i++){
            /* Load 8 pixels of input data x0 through x7 and kernel coefficients c0 through c7 */
            image_data_x0_x7 = *src_ptr; kernel_data_c0_c7 = *kernel_ptr;
            /* Compute the inner-product for pixel 0 */
            accumulate_0 += inner-product (image_data_x0_x7, kernel_data_c0_c7);
            /* Extract data x1 through x8 from x0 through x7 and x8 through x15 */
            image_data_x8_x15 =*( src_ptr + 1);
            image_data_x1_x8 = align(image_data_x8_x15 : image_data_x0_x7, 1);
            /* Compute the inner-product for pixel 1 */
            accumulate_1 += inner-product (image_data_x1_x8 : kernel_data_c0_c7);
            /* Extract data x2 through x9 from x0 through x7 and x8 through x15 */
            image_data_x2_x9 = align(image_data_x8_x15 : image_data_x0_x7, 2);
            /* Compute the inner-product for pixel 2 */
            accumulate_2 += inner-product (image_data_x2_x9 : kernel_data_c0_c7);
            .......
            accumulate_7 += inner-product (image_data_x7_x15 : kernel_data_c0_c7 );
            /* Update the source and kernel addresses */
            src_ptr = src_ptr + image_width; kernel_ptr = kernel_ptr + kernel_width;
        } /* end for i */
        /* Compress eight 32-bit values to eight 16-bit values with right-shift operation */
        result64_ps16_0 = compress1(accumulator_0 : accumulator_1 , scale);
        result64_ps16_1 = compress1(accumulator_2 : accumulator_3, scale);
        result64_ps16_2 = compress1(accumulator_4 : accumulator_5 , scale);
        result64_ps16_3 = compress1(accumulator_6 : accumulator_7, scale);
        /* Compress eight 16-bit values to eight 8-bit values. Saturate each individual value to 0 or 255
        and store them in two consecutive 32-bit registers */
        result32_pu8_0 = compress2(result64_ps16_0 : result64_ps16_1, zero);
        result32_pu8_1 = compress2(result64_ps16_2 : result64_ps16_3, zero);
        /* Store 8 pixels present in two consecutive 32-bit registers and update the destination address */
        *dst_ptr++ = result32_pu8_0_and_1;
```

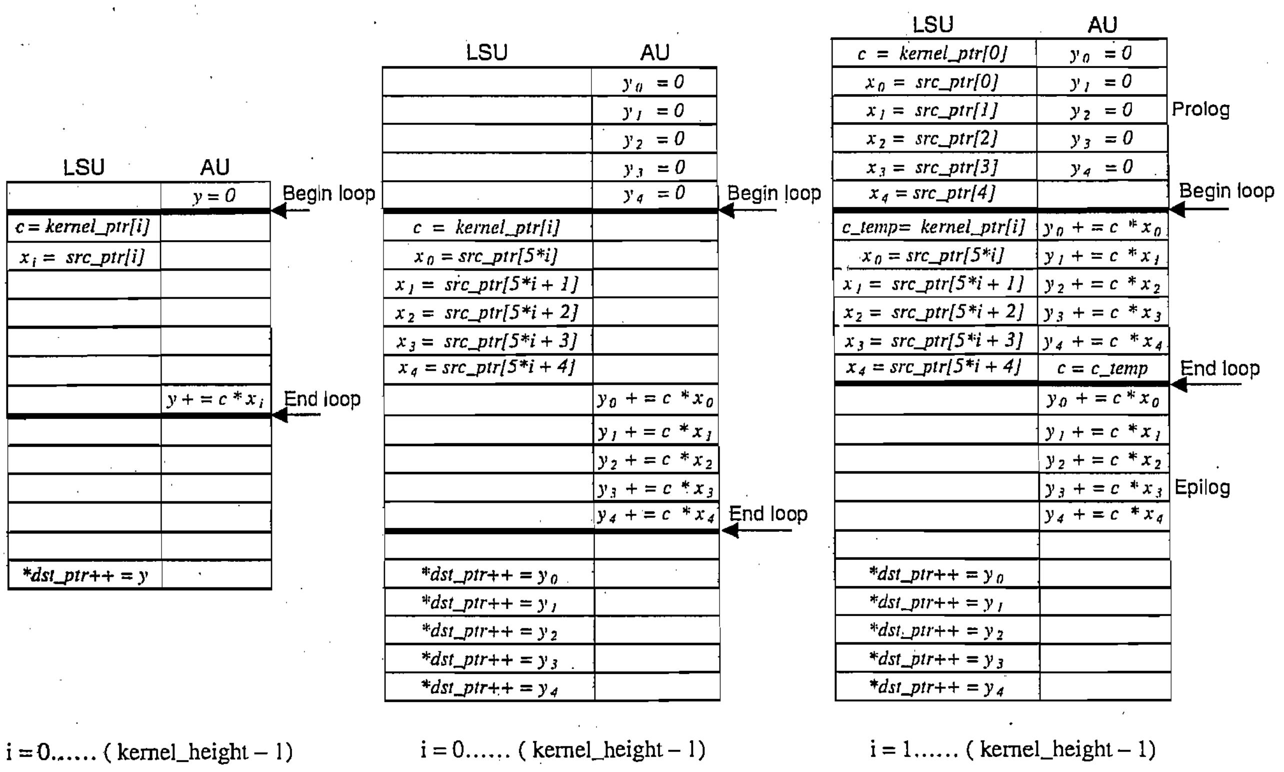
With this implementation, for every set of eight output pixels, 139 (56 *inner product*, 21 *load*, 49 *align*, 6 *add*, 6 *compress*, and 1 *store*) instruction slots are needed, which amounts to an average of 17.4 instruction slots per pixel. If the kernel width is greater than eight, then the kernel is subdivided as explained previously. Some media processors also have an *inner-product-add* instruction, which can add the current inner-product result to the specified register, thus eliminating the need for 6 add instructions for accumulating the multiplication results in the inner loop. Thus, by using these powerful VLIW instructions judiciously, it is possible to significantly improve the convolution performance.

However, these instructions take multiple cycles to complete (latency). For example, load/store instructions have a five-cycle latency while the *inner-product* instruction has a six-cycle latency in the MAP1000. To achieve the best performance, all the execution units on the media processor need to be kept as busy as possible every cycle, which is difficult due to these latencies. However, these instructions have a single-cycle throughput (i.e., another identical operation can be initiated in the next cycle) due to hardware pipelining. This characteristic can be utilized in overcoming the latency problem via loop unrolling and software pipelining.

2.2 Loop Unrolling and Software Pipelining

Let us consider the implementation of a simplified convolution algorithm using *inner-product-add* instructions (for simplicity *compress*, *align*, and *loop branching* are ignored) on a processor having a load/store unit (LSU) and an arithmetic unit (AU) with the latency of 5 for *load/store* and 6 for *inner-product-add* as shown in Fig. 4(a). The kernel (c) and image data (x) are loaded by the LSU and an *inner-product-add* instruction is performed in the AU with the result accumulated in y. Since *load* has a five-cycle latency, *inner-product-add* is issued after five cycles. After iterating over the kernel height, the final result is stored six cycles later since *inner-product-add* takes six cycles. Only three instruction slots are utilized out of 14 possible LSU and AU slots in the inner loop, which results in wasting 78.6% of the instruction issue slots and leads to disappointing computing performance.

To address this latency problem and underutilization of instruction slots, loop unrolling, and software pipelining can be utilized.[7] In loop unrolling, multiple sets of data are processed inside the loop. For example, five sets of data are processed in Fig. 4(b). The latency problem is partially overcome by taking advantage of the single-cycle throughout of *load* and *inner-product-add* and filling the delay slots with the unrolled instructions. But many instruction slots are still empty since the LSU and AU are not used simultaneously. Software pipelining can be used to fill these empty slots, where operations from different iterations of the loop are overlapped and executed simultaneously by the LSU and AU as shown in Fig. 4(c). By the LSU loading the data to be used in the next iteration and by the AU executing the *inner-product-add* instruction using the data loaded in the previous iteration, the LSU and AU execution is overlapped, thus increasing the instruction slot utilization. However, to utilize software pipelining, some preprocessing and postprocessing need to be performed. For example, loading in the prologue the data to be used in the first iteration and executing *inner-product-adds* in the epilogue for the data loaded in the last iteration as shown in Fig. 4(c). Thus, the judicious use of loop unrolling and software pipelining results in the increased data processing



(a) Without loop unrolling and software pipelining

(b) After loop unrolling

(c) After loop unrolling and software pipelining

Fig. 4 Example of loop unrolling and software pipelining.

throughput [approximately a factor of five for a 3×3 kernel when Figs. 4(a) and 4(c) are compared].

2.3 *Implementation on the MAP1000*

The MAP1000 is a programmable single-chip media processor with a highly parallel internal architecture optimized for image and video processing.[3] Figure 5 shows its high-level block diagram where the core processor block consists of two clusters, a 16-kbyte four-way set-associative data cache, and a 16-kbyte two-way set-associative instruction cache. It has an on-chip programmable DMA controller called data streamer (DS). Each cluster has 64 32-bit general registers, 16 predicate registers, a pair of 128-bit registers, an Integer Arithmetic Logic Unit (IALU), and an Integer Floating-Point Graphics Arithmetic Logic Unit

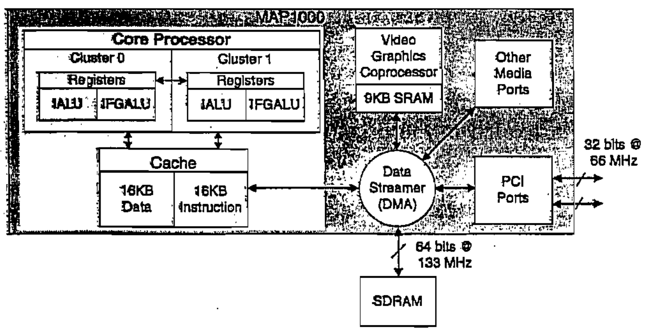


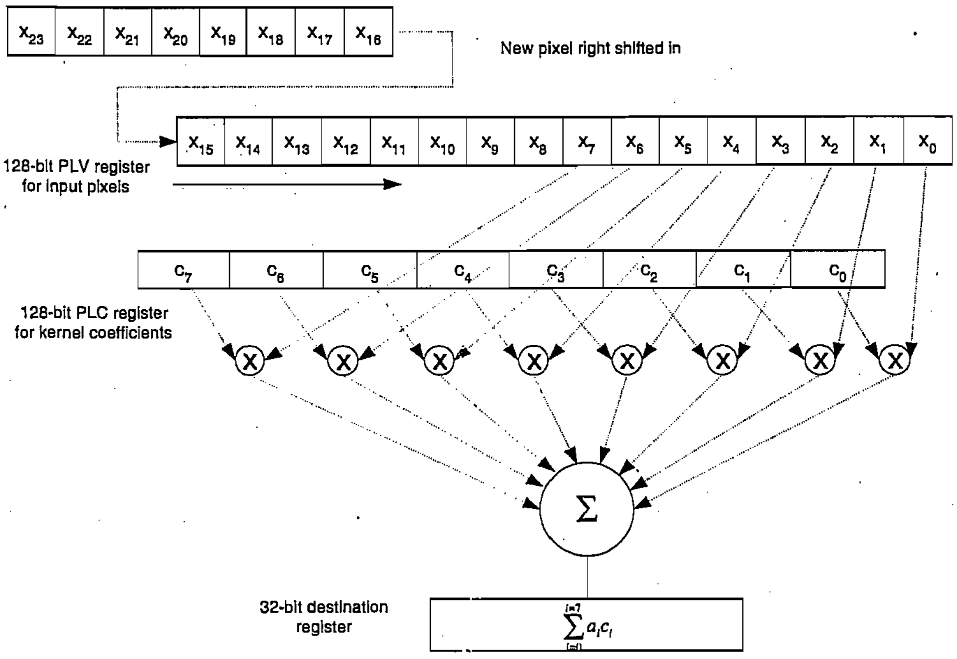
Fig. 5 Block diagram of the MAP1000.

(IFGALU). Two clusters are capable of executing four different operations (e.g., two on IALUs and two on IFGALUs) per clock cycle, providing a high level of computing power at 200 MHz and higher. The IALU can perform either a 32-bit fixed-point arithmetic operation or a 64-bit load/store operation while the IFGALU can perform either a 64-bit partitioned arithmetic operation or a floating-point operation including division and square root operations. This VLIW processor includes a rich set of more than 700 instructions with many multimedia-optimized extensions. Many IFGALU instructions can specify different partitions (each with 8, 16, or 32 bits), providing a performance improvement of 8×, 4×, or 2×, respectively. In addition, the MAP1000 has two PCI ports, a video graphics coprocessor (VGC) and various multimedia input/output ports. The VGC has its own 9 kbytes of on-chip memory and is designed to do bit-serial processing, such as that required in MPEG's Huffman decoding.

To obtain the best performance possible for 2D convolution, we have carefully mapped the algorithm to the MAP1000's multiple processing units, employed loop unrolling, and software pipelining to maximize the utilization of on-chip resources, and efficiently managed the data flow using the on-chip programmable DMA controller to minimize the I/O data transfer overhead.

2.3.1 *Algorithm mapping*

The MAP1000 has an advanced inner-product, srshinprod.pu8.ps16, as shown in Fig. 6. This instruction utilizes two 128-bit registers, i.e., PLV and PLC, to mul-



x_{23} x_{22} x_{21} x_{20} x_{19} x_{18} x_{17} x_{16}
New pixel right shifted in
x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
128-bit PLV register for input pixels
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
128-bit PLC register for kernel coefficients
X
Σ
32-bit destination register
$\sum_{i=0}^{i=7} a_i c_i$

Fig. 6 srshinprod.pu8.ps16 instruction using two 128-bit registers.

tiply eight 16-bit kernel coefficients by eight 8-bit input pixels and sum up the multiplication results. This instruction can also shift a new pixel into the PLV register. x_0 through x_{23} represent sequential input pixels while c_0 through c_7 represent kernel coefficients. The image data are loaded and shifted into the PLV register while the kernel coefficients are kept in the PLC register. After performing an inner-product operation shown in Fig. 6, x_{16} is shifted into the leftmost position of the PLV register while x_0 is shifted out. Next time, the inner-product operation will be performed between $x_1 \sim x_8$ and $c_0 \sim c_7$. This new pixel shifting-in capability eliminates the need of multiple *align* instructions used in the pseudo code in Sec. 2.1. The MAP1000 also has *compress* instructions similar to the ones shown in Fig. 3. Since each of these instructions on the MAP1000 has a multiple-cycle latency (five for *load/store*, three for *compress*, and six for srshinprod.pu8.ps16) with a single-cycle throughput, loop unrolling, and software pipelining have been heavily utilized to overcome the latency problem.

2.3.2 *Data flow management with programmable DMA controller*

Since the on-chip data cache is limited in size (16 kbytes for the MAP1000) and cannot hold an entire image, we usually process a small image block at a time. To keep the processor from waiting for the data to arrive in the on-chip memory, an on-chip programmable DMA controller can be used to manage the data movements concurrently with the processor's computation. This technique is illustrated in Fig. 7 and is commonly known as *double buffering* where four buffers, two for input blocks (*ping_in_buffer* and *pong_in_buffer*), and two for output blocks (*ping_out_buffer* and *pong_out_buffer*), are allocated in the on-chip

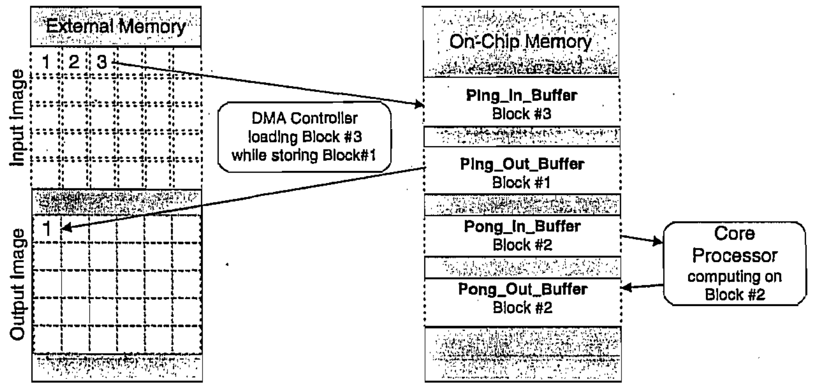


Fig. 7 Double buffing with a programmable DMA controller.

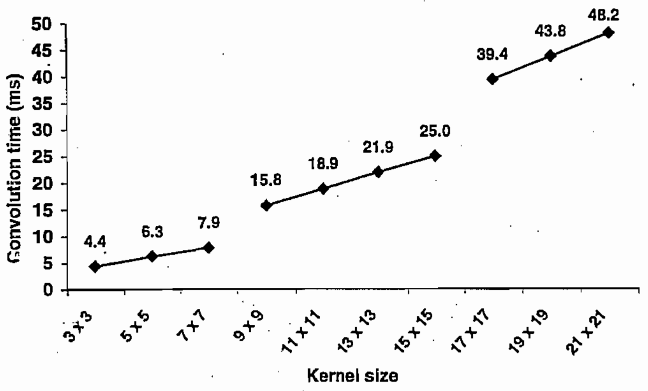


Fig. 8 MAP1000 convolution time vs kernel size on a 512×512×8-bit image.

memory. While the core processor computes on a current image block (e.g., block No. 2) from *pong_in_buffer* and stores the result in *pong_out_buffer*, the DMA controller moves the previously calculated output block (e.g., block No. 1) in *ping_out_buffer* to the external memory and brings the next input block (e.g., block No. 3) from the external memory into *ping_in_buffer*. When the computation and data movements are both completed, the core processor and DMA controller switch buffers with the core processor starting to use the ping buffers and the DMA controller working on the pong buffers.

3 Results and Discussion

In this section, we first present the convolution performance on the MAP1000 and compare it to other currently available convolution solutions. Then, we will discuss the convolution algorithm on other media processors and for specialized kernels. Finally, we will briefly introduce the University of Washington Image Computing Library that we have developed on the MAP1000.

3.1 *Performance*

The 2D convolution times on a 512×512 image with different-size kernels on the 200 MHz MAP1000 are plotted in Fig. 8. These performance numbers include the tight-loop execution time, instruction and data cache miss time, and all the additional time needed for the I/O requests and data transfers.

It is interesting to note that the convolution time on the MAP1000 does not increase quadratically $[O(M^2)]$ as the kernel size increases because of the availability of inner product. Instead, it increases linearly since it only has to process more rows with the increasing kernel height until the kernel width reaches 8 (the number of pixels that can be processed in one inner-product instruction). When the kernel width exceeds 8, there is a jump in the convolution time as shown in Fig. 8. Another jump occurs when the kernel width crosses 16.

3.2 *Comparison with Other Convolution Implementations*

The LSI Logic's L64240[9] and PSDP16488[10] are dedicated hardware convolver chips that have been utilized to perform fast convolution. The systems built with these chips to perform convolution require many additional supporting chips for performing other tasks, such as controlling these convolver chips and providing the data to these chips. Our 8×8 convolution performance of 8.83 ms on the MAP1000 is slower than 6.56 ms on the 40 MHz PDSP16488 and is 1.48 times faster than 13.11 ms on the 20 MHz LSI Logic's L64240. However, as discussed in Sec. 1, these hardwired chips can only perform convolution. Additionally, these hardwired chips cannot easily accommodate a kernel larger than 8×8, and their coefficients are 8 bits.

Programmable imaging systems from commercial vendors, such as Blue Wave Systems and Alacron, offer the flexibility of performing not only convolution, but also other imaging functions, such as FFT, affine warping and binary morphology. The Blue Wave Systems PCI/C6200[11] based on a single 200 MHz Texas Instruments TMS320C6201 reports 7.2 ms for a 3×3 convolution (the execution only, excluding the I/O) for a 512×512 image. This is 1.7 times slower than the total (execution, I/O and other overhead) time on the MAP1000. Similarly, the Alacron's Al-860[12] based on dual 40 MHz i860 processors reports 66.1 ms (again execution only, excluding the I/O time), which is 15 times slower than that with the MAP1000 for the same image and kernel sizes. Systems using field programmable gate arrays (FPGAs) possess an intermediate level of flexibility in implementing different imaging functions. For example, Peterson *et al.*[13] developed a system called *Splash-2* with 16 Xilinx XC4010 FPGAs as processing nodes. This FPGA-based system can perform a 15×15 8-bit convolution on a 512×512 image in 66.7 ms, which is 2.7 times slower than the fully programmable MAP1000.

To compare our performance on the MAP1000 with a general purpose processor, we measured the convolution execution times on the 500 MHz Pentium III utilizing the optimized Intel Image Processing Library[8] version 2.1. The execution time for a 7×7 convolution on a 512×512 8-bit image is 56.5 ms, which is 7.2 times slower than the MAP1000's performance of 7.9 ms.

3.3 *Convolution Algorithm on Other Media Processors*

The pseudo code described in Sec. 2.1 for convolution can be ported to other media processors as well. On the Philips Trimedia TM1000, inner product is available under the name of *ifir16*, which performs two 16-bit multiplications and accumulation, and align is available under the name of *funshift*.[2] Two *ifir16* can be issued each cycle on the TM1000, executing four multiplications and accumulations in a single cycle. Instead of specifying a shift amount for *align*, the TM1000 supports several variations of *funshift* to extract the desired aligned data, e.g., *funshift1* is equivalent to align with one-pixel shift while *funshift2* is equivalent to align with two-pixel shift. On some media processors that do not support *inner product*, such as the TMS320C80, multiply and add instruction can be utilized to perform convolution. An efficient algorithm to implement 2D convolution on the TMS320C80 is discussed elsewhere.[14]

A comparison in convolution performance between the 50 MHz TMS320C80, 100 MHz TM1000 and 200 MHz MAP1000 is given in Fig. 9. On the TMS320C80, the convolution time grows quadratically $[O(M^2)]$, with the ker-

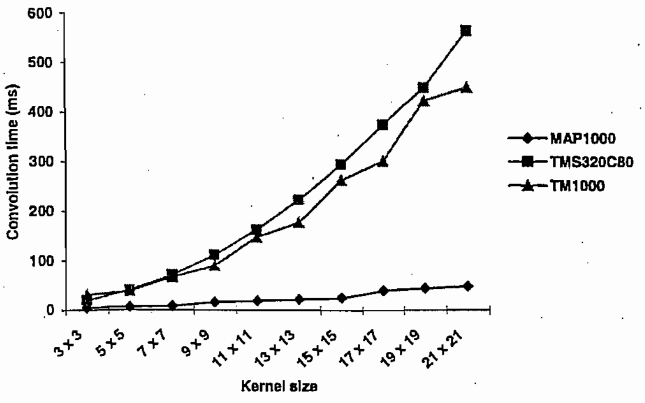


Fig. 9 Comparison of convolution time between the MAP1000, the TM1000, and the TMS320C80.

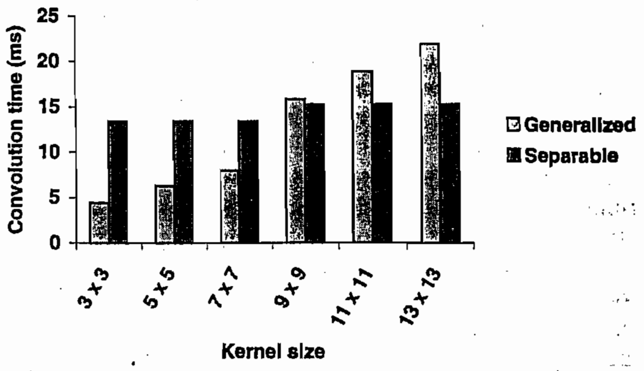


Fig. 10 Generalized vs separable convolution on the MAP1000.

nel size since the number of operations increases quadratically with the kernel size as discussed in Sec. 2. However, on the MAP1000 and TM1000, the convolution time increase is piecewise linear due to the availability of the *inner product* as discussed in Sec. 3.1. The steeper increase in convolution time on the TM1000 compared to the MAP1000 is because the number of multiplications and additions that can be executed on the TM1000 is four times less than that of the MAP1000. Even though the 128-bit registers in the MAP1000 are not used to their full capability for convolution with small-size kernels, e.g., 3×3, they can be fully utilized for convolution with large kernels. For example, the convolution time ratio between the 21×21 and 3×3 kernels is 10.9 on the MAP1000 compared to 29.4 on the TMS320C80, thus the MAP1000 reduces the rate of increase in computation time significantly as the kernel size increases.

3.4 *Convolution Algorithm for Specific Kernels*

There are several methods to further reduce the amount of computation in implementing convolution in a programmable fashion. If a kernel is separable, 2D convolution could be implemented by applying two consecutive 1D kernels (x-directional convolution on the original image followed by y-directional convolution on the x-directional convolved image), thus substantially reducing the number of multiplications.[14] In order to utilize the powerful inner-product instruction, the intermediate image after x-directional convolution is transposed (row-column transformation) so that another x-directional convolution (instead of y-directional convolution) can be performed. Then, one more transposition is performed before storing the final result. The performance of separable convolution along with generalized convolution on the MAP1000 is shown in Fig. 10. Note that separable convolution is not advantageous over generalized convolution on the MAP1000 until the kernel size reaches 9×9. This is unusual since the number of multiplications has been traditionally the most important factor in determining the convolution performance in programmable processors.[15] For an application where the necessary kernel size is less than 9×9, generalized 2D convolution that requires more multiplications would be faster in the MAP1000 than separable convolution, demonstrating the computational power of newer media processors.

Other techniques to reduce the computation include optimization for the specific convolution kernel type, such as the kernels shown in Fig. 11. If all the kernel elements have a value of 1 (i.e., boxcar kernel), then the moving average method can be used to perform convolution.[16] On the MAP1000, we found that due to the availability of *inner product*, the moving average method and separable convolution method have similar performance. For convolution with the 3×3 kernels in Figs. 11(b) and 11(c), we were able to improve the performance by 29% over generalized convolution by eliminating load and multiply/accumulate operations for one entire row of the kernel since they are all zeros. Thus, in general it is possible to improve the convolution performance for a specific kernel by analyzing the kernel, eliminating loads, skipping other operations or using the kernel's separability property. However, the speed improvement we can obtain for these specific kernels is getting smaller in modern media processors due to the increasing number of simultaneous multiplications and additions that can be performed in a single cycle.

3.5 *University of Washington Image Computing Library (UWICL)*

We have developed a library of image computing algorithms for the MAP1000 utilizing programming techniques similar to those used for the convolution algorithm. This library currently has 121 functions, and it includes most low-level imaging routines, such as filtering, morphology, histogram, transforms, and geometric manipulations.[17] Using the UWICL functions as building blocks, more complex imaging applications or algorithms can be quickly developed and integrated on an MAP1000-based system, resulting in shorter time-to-market and lowered R&D costs.

1	1	1
1	1	1
1	1	1

(a)

-1	-1	-1
0	0	0
1	1	1

(b)

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

(c)

Fig. 11 Examples of specific convolution kernels.

4 Conclusion

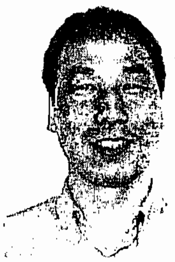
Even though media processors are being developed to meet the growing computation demand, any significant performance improvement for a given function is possible only by careful algorithm design and use of many media processor programming techniques. Otherwise, a generic algorithm may not produce the desired performance gain. By efficiently mapping the convolution algorithm to the underlying MAP1000 architecture, we were able to obtain performance figures that are faster than most existing solutions specifically tailored to perform only convolution. The programmability of modern media processors provides an added advantage of being more flexible than the ASIC based or hardwired approaches. For example, the same processing module can be used not only in supporting convolution, but also in warping, MPEG-2 decoding, and other functions/applications. Thus, we believe that this high performance software-based approach using media processors will provide much more flexibility and improved cost/performance in the future than the existing hardwired solutions for many imaging and video applications.

References

1. K. Guttag, R. J. Gove, and J. R. Van Aken, "A single chip multiprocessor for multimedia: The MVP," *IEEE Comput. Graphics Appl.* **12**, 53–64 (1992).
2. S. Rathnam and G. Slavenburg, "Processing the new world of interactive media. The Trimedia VLIW CPU architecture," *IEEE Signal Process. Mag.* **15**, 108–117 (1998).
3. C. Basoglu, R. J. Gove, K. Kojima, and J. O'Donnell, "A single-chip processor for media applications: The MAP1000," *Int. J. Imaging Syst. Technol.* **10**, 96–106 (1999).
4. J. J. Shieh and C. A. Papachristou, "Fine grain mapping strategy for multiprocessor systems," *IEE Proc. Comput. Digit. Tech.* **138**, 109–120 (1991).
5. A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro.* **16**(4), 42–50 (1996).
6. S. D. Pathak, "Interactive automatic fetal head boundary detection from ultrasound images," Master's thesis, University of Washington, Seattle, WA, 1996.
7. M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN: Conf. Programming Language Des. Implementation*, **23**, pp. 318–328 (1988).
8. Intel's® image processing library V2.1, http://support.intel.com/support/performancetools/libraries/, 1999.
9. LSI, L64240 Multibit Filter (MFIR), LSI Logic Corporation, Milpitas, CA, 1989.
10. Mitel Semiconductor, "PDSP16488 Single chip convolver with integral line delays," http://www.mitelsemi.com/products/htm_view.cgi/PDSP16488/, 1997.
11. Blue Wave Systems, http://www.bluewavesystems.com/, 1998.
12. Alacron, http://www.alacron.com/, 1999.
13. J. B. Peterson and P. M. Athanas, "High-speed 2D convolution with a custom computing machine," *J. VLSI Signal Process.* **12**, 7–19 (1996).
14. R. A. Managuli, C. Basoglu, S. D. Pathak, and Y. Kim, "Fast convolution on a programmable mediaprocessor and application in unsharp masking," *SPIE Med. Imaging* **3335**, 675–683 (1998).
15. J. Kim and Y. Kim, "Efficient 2D convolution algorithm with the single-data multiple kernel approach," *Graph. Mod. Image Process.* **57**, 175–182 (1995).
16. M. J. McDowell, "Box-filtering technique," *Comput. Graph. Image Process.* **17**, 67–70 (1981).
17. Imge Computing Systems Laboratory at the University of Washington, MAP UWICL Consortium, http://icsl.ee.washington.edu/, 1999.

Ravi Managuli earned his BSEE degree in India in 1993 and an MSEE degree from the University of Nevada–Las Vegas in 1996. He is currently a PhD candidate in electrical engineering at the University of Washington, Seattle, Washington. His research interests include designing of signal, image, and video processing algorithms and their efficient implementation for real-time applications on VLIW mediaprocessors. He has experience in programming various digital signal processors and has been working on the image computing libraries for TMS320C80 and MAP1000. Currently he is leading a project in the Image Computing Systems Laboratory of the University of Washington to design a fully programmable ultrasound machine using next generation mediaprocessors.

George York received his BS degree in electrical engineering from the US Air Force Academy in 1986 and MS and PhD degrees in electrical engineering from the University of Washington in Seattle in 1988 and 1999, respectively. As an Air Force officer, he developed embedded computers for missile systems at the Air Force Research Laboratory at Eglin AFB, Florida (1988 to 1992), designed multimedia communication systems at the Korean Agency for Defense Development in Taejon, Korea (1992 to 1994), and taught computer engineering courses in the Electrical Engineering Department at USAFA (1994 to 1996). From 1996 to 1999, he did his PhD research at the University of Washington's Image Computing Systems Lab where he designed architectures and algorithms for fully programmable ultrasound machines, based on mediaprocessor technology. Currently he is employed at the National Security Agency at Ft. Meade, Maryland. His research interests include medical imaging, digital signal processing, mediaprocessors, and real-time computing.



Donglok Kim received his BS degree in electronics engineering and MS degree in control and instrumentation engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively, and his PhD degree in electrical engineering from the University of Washington in 1997. From 1990 to 1997, he was a research assistant at the University of Washington's Image Computing Systems Laboratory. He was involved in various research projects including 3D graphics library, mediaprocessor and system architectures, image computing libraries, and a programmable ultrasound image processor. He has been working as a research assistant professor at the University of Washington in the areas of ultrasound imaging systems, mediaprocessor architectures, MPEG-4 algorithm mapping, and portable image computing libraries for mediaprocessors.

Yongmin Kim received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, and the MS and PhD degrees in electrical and computer engineering from the University of Wisconsin, Madison. He is professor and chair of bioengineering, professor of electrical engineering, and adjunct professor of radiology, computer science and engineering at the University of Washington, Seattle. His research interests are in algorithms and systems for multimedia, image processing, computer graphics, medical imaging, high-performance programmable processor architecture, and modeling and simulation. His group has filed 64 invention disclosures and 38 patents, and 20 commercial licenses have been signed. Dr. Kim received the Early Career

Achievement Award from the IEEE Engineering in Medicine and Biology Society in 1988. He is a member of the editorial board of the *IEEE Transactions on Biomedical Engineering, IEEE Transactions on Information Technology in Biomedicine*, the IEEE Press series in biomedical engineering, and *Annual Reviews of Biomedical Engineering*. He is a member of the IEEE Fellow Committee.