A Reconfigurable Accelerator for 32-Bit Arithmetic

Reiner W. Hartenstein, Rainer Kress, Helmut Reinig

University of Kaiserslautern
Erwin-Schrödinger-Straße, D-67663 Kaiserslautern, Germany
Fax: ++49 631 205 2640, email: abakus@informatik.uni-kl.de

Abstract

This paper introduces the MoM-3 (Map oriented Machine 3) architecture as an accelerator for applications with a large amount of 32-bit arithmetic. The data manipulations in the inner loop of the application can be configured as a complex hardware operator into special field-programmable devices. Several complex address generators are programmed to implement the control structures of the algorithm, so that data is accessed completely under hardware control, without requiring memory cycles for address computations. The MoM-3 supports pipelining on statement and expression level. The compiler takes C as input language and produces the configuration code for the field-programmable devices and the address generators without further user interaction. Acceleration factors in the range of 12 to 62 have been obtained, compared to a high-end Sun SPARC workstation.

1. Introduction

Many computation-intensive algorithms take too much execution time, even on a well-equipped modern workstation. This opens a market for hardware accelerators of all kinds. If the market share is big enough, ASIC-based accelerators are designed. JPEG and MPEG image and video compression can be based on full custom circuits from C-Cube [3] or Zoran [12], for example. But such ASICs are expensive and suffer from a long development time. If a user doesn't require the highest performance possible for a specific algorithm, but a speed-up for several different algorithms, he might be better off with a kind of custom computing machine. A lot of commercial and non-commercial solutions have already been presented, like Splash-2 [1], DEC Perle-1 [2], Virtual Computers [4], and CHS2x4 Custom Computer [10]. All of these are based on commercially available FPGA technology, especially the SRAM-based XILINX devices [11]. All of the currently available FPGAs are configurable at bit-level. Therefore they provide only modest performance and capacity when configuring 32-bit datapaths which are standard for current microcomputers. As a consequence, most custom computing machines do not provide operators which match the word length of standard high level language data types. Especially arithmetic operators of a significant word length consume too much area and suffer from large delays, making it difficult to accelerate computation-intensive algorithms on most custom computing machines.

The MoM-3 design aims at the following goals: 1. programmability from standard high level languages like C, without requiring hardware expertise, 2. acceleration of scientific algorithms that process a large amount of data. Since many of these algorithms do a large amount of arithmetic, the MoM-3 needs a computational device that supports arithmetic operations on standard data types like 8-bit, 16-bit and 32-bit integers. The device must be reconfigurable, because different algorithms shall be accelerated by the same MoM-3 hardware. For lack of commercially available devices that fulfil these requirements, we developed our own, the so-called rDPA [6]. The MoM-3 makes use of another characteristic of scientific algorithms, to achieve further speed-up. Since many of these algorithms are organized in loops, which compute indices to large data arrays, the MoM-3 contains several address generators to compute the appropriate data address sequences completely under hardware control.

In the following section, this paper introduces the hardware structure of the MoM-3. Afterwards, the features of the C compiler for the MoM-3 are outlined. The fourth section explains the way algorithms are run on the MoM-3 by means of an example. The final sections provide a performance comparison and conclude the paper.

2. The MoM-3 Hardware

The MoM-3 architecture is based on the Xputer machine paradigm [7]. MoM is an acronym for Map oriented Machine, because the data memory is organized in

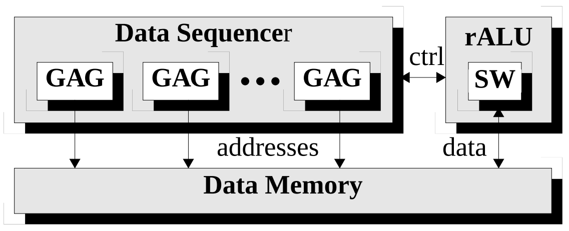

Figure 1. Block Diagram of an Xputer

two dimensions like a map. Instead of the von Neumann-like tight coupling of the instruction set to the data manipulations performed, an Xputer shows only a loose coupling between the sequencing mechanism and the ALU. That's why an Xputer efficiently supports a reconfigurable ALU (rALU). The rALU contains complex operators which produce a number of results from a number of input data (figure 1). All input and output data to the complex operators is stored in so-called scan windows (SW). A scan window is a programming model of a sliding window, which moves across data memory under control of a data address generator. All data in the scan windows can be accessed in parallel by the rALU operator. The rALU operators are activated every time a new set of input data is available in the scan window. This so-called data sequencing mechanism is deterministic, because the input data is addressed by Generic Address Generators (GAG). They compute a deterministic sequence of data addresses from a set of algorithm-dependent parameters. An Xputer Data Sequencer contains several Generic Address Generators running in parallel, to be able to efficiently cope with multiple data sources and destinations for one set of complex operators.

In the MoM-3, the Data Sequencer is distributed across several computational modules (C-Modules), as can be seen in figure 2. Each C-Module consists of a Generic Address Generator (GAG), an rALU subnet and at least two megabytes of local memory. All C-Modules can operate in parallel when each Generic Address Generator accesses data in its local memory only. The MoM-3 includes up to seven C-Modules. Apart from the local memory access, two means of global communication are available. First, the rALU subnets can exchange internal results with their neighbours by use of the rALU interconnect, without disturbing parallel activity. Second, the Generic Address Generators can access data memory and rALU subnets on other C-Modules using the global MoMbus. This can be done only sequentially, of course. The global MoMbus is used by the MoM-3 controller (M3C) as well, to reconfigure the address generators and the rALU whenever necessary. The MoM-3 controller is the coordinating instance of the Data Sequencer, which ensures a well-defined parallel activity of the Generic Address Generators by selecting the appropriate parameter

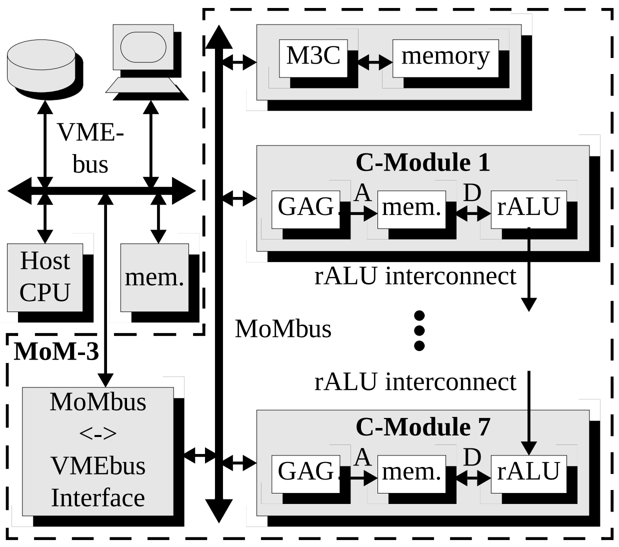

Figure 2. The MoM-3 Hardware

sets for configuration. Via the MoMbus-VMEbus interface, the host CPU has access to all memory areas of the MoM-3 and vice versa. The Generic Address Generators have DMA capability to the host's main memory, reducing the time to download a part of an application for execution on the MoM-3. The host CPU is responsible for all disk I/O, user interaction and memory allocation, so that the MoM-3 completely uses the functionality of the host's operating system.

The printed circuit boards with the MoM-3 controller and the C-Modules reside in their own 19 inch rack with the MoMbus backplane. The only connection to the host computer is through the bus interface. That way, the MoM-3 could be interfaced to a different host with minimal effort. The part of the bus interface that is injected into the host's backplane would have to be redesigned according to the host's bus specifications. All other parts could remain the same.

2.1 The Data Sequencer

The MoM-3 Data Sequencer consists of up to seven Generic Address Generators and the MoM-3 controller. The Generic Address Generators operate in a two-stage pipeline. The first stage computes handle positions for the scan windows (Handle Position Generator). Each Handle Position Generator controls one scan window. A handle position consists of two 16-bit values for the two dimensions of the data memory. The sequence of handle positions describes how the corresponding scan window is moved across the data memory (figure 3). Such a sequence of handle positions is called scan pattern. A Handle Position Generator can produce a scan pattern corresponding to four nested loops at the most. It is programmed by specifying a set of parameters, such like starting position,

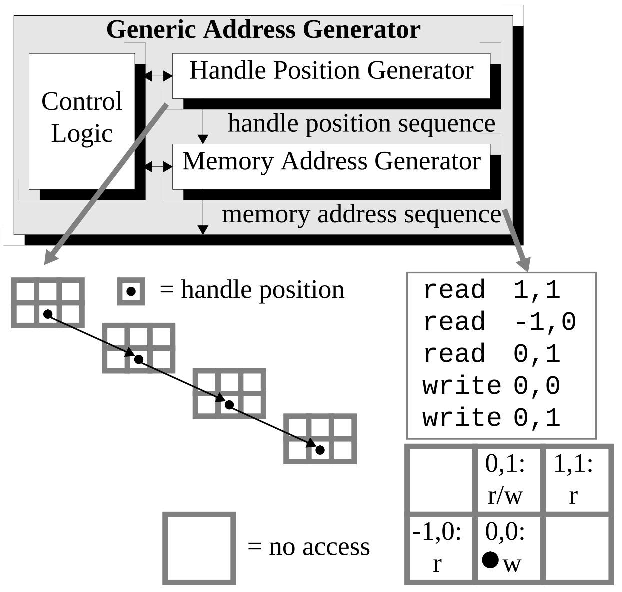

Figure 3. Generic Address Generator

increment value, and end position of a loop, each both for the x and y dimension of the data memory.

The second pipeline stage computes a sequence of offsets to the handle positions, to obtain the effective memory addresses for the data transfers between the scan window / rALU and the data memory. Therefore this stage is called Memory Address Generator. The range of offsets may be –32 to +31 in both dimensions of the data memory. The sequence of offsets may be programmed arbitrarily, but at most 250 references to the data memory can be made from one handle position. The offset sequence may be varied at the beginning and at the end of the loops of the Handle Address Generator. This allows to perform extra memory accesses to fill a pipeline in the rALU at the beginning of a loop, as well as additional write operations to flush a pipeline at the end. The address parts of the two dimensions may combined to a real linear memory address in four ways: one row of two-dimensional memory may consume an address space of 10, 12, 14, or 16 bits. This allows to adjust the “size” of the data memory to the size of the processed data, to reduce wastage of address space. The software environment contains a memory management module, which makes use of the unused area, from the end of a data row to the next appropriate power-of-two boundary. If the algorithm uses another array of data, which fits into the remaining space, such two data arrays are placed in memory, one after the other. After the concatenation of the two address parts to a linear address, a 32-bit base address is added, to obtain the effective memory address. The base address typically is the starting address of the data array referenced by this Generic Address Generator, as determined by the memory management software. The Memory Address Generator itself is a three stage pipeline. The first stage performs the offset calculations and the combination of address parts to a linear address. The second stage adds the 32-bit base address, and the third stage handles the bus protocol for data transfers.

The Generic Address Generators run in parallel. They synchronize their scan patterns through the activations of the rALU. All scan patterns proceed until either they detect an explicit synchronization point in the offset sequence of the Memory Address Generator, or they are blocked in a write operation to memory, waiting for the rALU to compute the desired result.

The hardware controlled generation of long address sequences allows to access data in memory every 120 ns, using 70 ns memory devices and a conventional, non-interleaved memory organization. A further speed-up of memory accesses could be obtained by interleaved memory banks and by the introduction of static RAM caches, like in conventional computers. With caches, it would make sense to have a programmable cache controller in the MoM-3. Since the hardware generated address sequences are deterministic, a compiler could compute a cache update strategy at compile time, which would result in the minimum number of cache misses. This should provide better performance than the probabilistic cache update strategies usually applied.

2.2 Reconfigurable ALU

The computations in each rALU subnet of the MoM-3 are performed by an rDPA array (reconfigurable datapath architecture). The rDPA array consists of eight rDPA chips in a two by four organization. Each rDPA chip contains an array of sixteen (four by four) datapath units (DPU). Each datapath unit can implement any unary or binary operator from C language on integer or fixed-point data types up to 32 bits length. Multiplication and division are performed by means of a microprogram, whereas all other operators can be executed directly. Additionally, operators like multiplication with accumulation are available in the library of configurable operators.

Each datapath unit has two input registers, one at the north and one at the west, and two output registers, one at the south and one at the east (figure 4). All registers can be used for intermediate results throughout the computation. The registers in the DPUs in fact are a distributed implementation of the scan window of the Xputer paradigm. The datapath units can read 32-bit intermediate results from their neighbours to the west and to the north and pass 32-bit results to their neighbours in south and/or east direction. A global I/O bus allows input and output data to be written directly to a DPU without passing them from neighbour to neighbour. The DPUs operate data-driven. The operator is applied each time, new input data is availa-

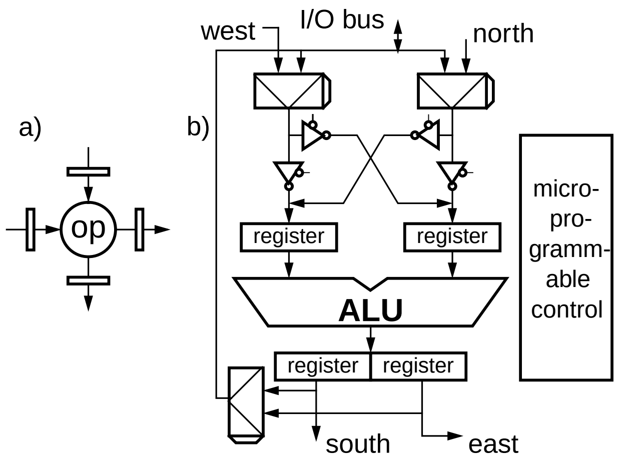

Figure 4. Datapath unit: a) symbol; b) block diagram

ble either from the bus or from a neighbouring DPU. This decentralized control simplifies pipelining of operations, when each operation takes a different time to execute.

The array of DPUs expands across rDPA chip boundaries in a way that is completely transparent to the compiler software. To overcome pinout restrictions, the neighbour to neighbour connections are reduced to serial links with appropriate converters at the chip boundaries (figure 5). Although pipelining drastically reduces the effect of the conversion from 32 bits to 2 bits, this still may turn out a bottle neck in the current rDPA prototype. With state of the art IC technology and packages larger than 144 pins, we could build an rDPA with 66 MHz serial links (instead of 33 MHz) and four bits wide communications channels. This fourfold speed-up would overcome the shortfalls of the current prototype.

In addition to the rDPA array, a MoM-3 rALU subnet (figure 6) contains an rALU controller circuit which interfaces to the MoMbus. It provides a register file as a kind of cache for frequently used data, and controls the data transfers on the global I/O bus. The rDPA, in combination with

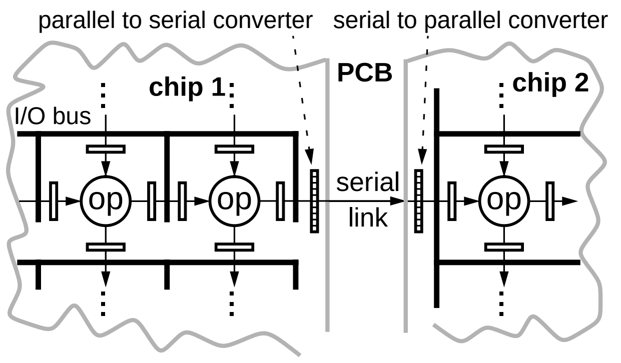

Figure 5. Expansion of nearest neighbour interconnect across chip boundaries

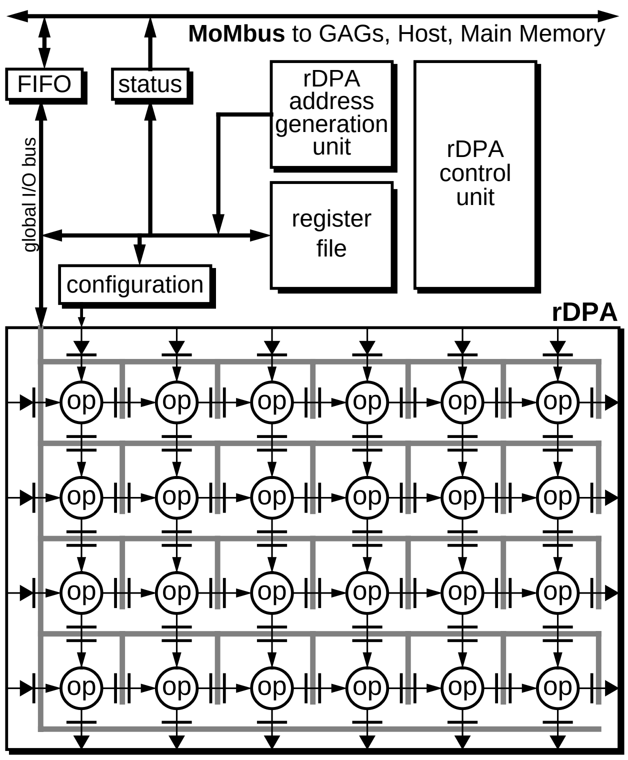

Figure 6. One subnet of the reconfigurable data-driven ALU

the rALU controller, supports pipelining on operator level (e.g. floating point operations are implemented as a pipeline across several datapath units), pipeline chaining [8], and pipelining of loop bodies, as shown in the example in section 4.2. That way, the complex operators of subsequent loop iterations are computed in parallel in a pipeline. Each of the loop iterations is finished to a different degree, depending on its progress in the pipeline.

The whole rDPA array is configured across one serial link of the DPU in the upper left corner of the array. Each configuration word is preceeded by a two-dimensional DPU address in relative format, and the address for the microprogram storage. As long as both DPU address components are different from zero, the configuration word is passed on to a neighbouring DPU, with the appropriate DPU address component decremented by one. If the local interconnect to the east is busy, a configuration word may be routed to the south first (if the DPU address component in vertical direction is different from zero), which results in an automatic load balancing for the slow serial links. The relative address format allows to use the same configuration data for rDPA arrays of arbitrary size. Configuration data words are marked by a special configuration bit set by the MoM-3 controller. That way the rDPA array may be reconfigured partially during runtime, while other areas of the array are still busy with computations. Configuration data is passed along with higher priority than computational data, each time before a datapath unit starts with the computations for the next application of its operator. That way, both configuration and computation are slowed down, if part of the rDPA array is busy, but both are sure to continue without deadlocks.

For additional details on the rDPA see [6]. Although this publication refers to an older version of the rDPA, the main structure still is the same, only the implementation has improved.

3. MoM-3 C Compiler

The C compiler for the MoM-3 takes an almost complete subset of ANSI C as input. Only constructs, which would require a dynamic memory management to be run on the MoM-3 are excluded. These are pointers, operating system calls and recursive functions. Since the host's operating system takes care of memory management and I/O, the software parts written for execution on the MoM-3 do not need such constructs. There are no extensions to C language or compiler directives required to produce configuration code for the MoM-3. The compiler computes the parameter sets for the Generic Address Generators, the configuration code for the rDPA arrays, and the reconfiguration instructions for the MoM-3 controller, without further user interaction. First, the compiler performs a data and control flow analysis. The data structure obtained allows restructurations to perform parallelizations like those done by compilers for supercomputers. These include vectorization, loop unrolling, and parallelizations on expression level. The next step performs a re-ordering of data accesses to obtain access sequences, which can be mapped well to the parameters of the Generic Address Generators. Therefore, the compiler generates a so-called data map, which describes the way the input data has to be distributed in the data memory to obtain optimized hardware generated address sequences. After a final automatic partitioning, data manipulations are translated to a rALU description, and the control structures of the algorithm are transformed to Data Sequencer code. An assembler for the Data Sequencer translates the Data Sequencer code to parameter sets for the Generic Address Generators and a reconfiguration scheme for the MoM-3 controller.

The rALU description is parsed by the ALE-X assembler (arithmetic and logic expression language for Xputers). It generates a mapping of operators to DPUs, merges DPU operators where possible, and computes a conflict-free I/O schedule, which matches the operators' speed, to keep the DPUs as busy as possible. The separation of the rALU code generation from the rest of the compiler allows to use other devices than the rDPA to build a MoM-3 rALU, without having to rewrite the C compiler with all its optimizations. For a new rALU, only an assembler generating rALU configuration code from ALE-X specifications has to be written. This is not too difficult, because ALE-X describes only arithmetic and logic expressions on the scan windows' contents.

A more detailed description of the MoM-3 programming environment can be found in [13]. The most important benefit of the MoM-3 C compiler compared to development environments of other custom computing machines is the fully automatic code generation. The programmer neither has to be a hardware expert, to guide hardware synthesis with appropriate constraints or compiler directives, nor has he to interfere with different stages of the compilation to get reasonable results.

4. Example: two dimensional FIR filter

The operation of the MoM-3 and the way an algorithm is adapted to the non-von Neumann architecture of the MoM-3 is illustrated with a two-dimensional FIR filter. The two dimensional FIR (finite impulse response) filter is one whose impulse response processes only a finite number of nonzero samples. The equation for a general two dimensional FIR filter is

$$y_{nm} = \sum_i \sum_j k_{ij} \cdot x_{n-i,\, m-j} \cdot \qquad (1)$$

4.1 Straightforward Implementation

In this example a two dimensional filtering of second order is shown, that is indices i and j have a range of zero to two. Given the order of a filter and its weights k_{ij}, the most efficient implementation on a von Neumann computer is to unroll the two loops of the filter kernel, to create a large expression computing the new value of y_{nm} (see figure 7). The weights k_{ij} now are constants, so that the compiler can optimize the multiplications by replacing them with a matching sequence of additions and shift operations.

The same source code produces an efficient implementation on the MoM-3 as well. The C compiler vectorizes the expression and performs loop unrolling to the extend of the capacity of the rDPA array. With multiplication and accumulation being a valid operator, the expression to compute y[n][m] takes three by four DPUs. The multiplications and summation of the rows are done in a three by three array of multipliers and multiplier-accumulators. A column of three routers/adders combines the sums of rows to the complete result (figure 8).

Taking the chip boundaries into account (with the costly parallel-to-serial converters) eight of these expressions can be computed in parallel, each in its own rDPA chip. The configuration for the whole rDPA array simply

```
For (n = 0; n < maxn; n++) {
   For (m = 0; m < maxm; m++) {
      y[n][m]  = k00*x[n][m]
               + k01*x[n][m+1]
               + k02*x[n][m+2]
               + k10*x[n+1][m]
               + k11*x[n+1][m+1]
               + k12*x[n+1][m+2]
               + k20*x[n+2][m]
               + k21*x[n+2][m+1]
               + k22*x[n+2][m+2];
   }
}
```

Figure 7. Source code of a 3 * 3 two-dimensional FIR filter

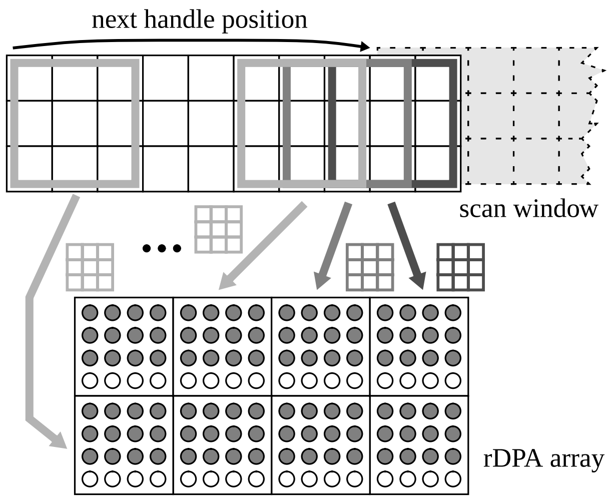

Figure 9. Scan window and rALU configuration of a vectorized 3 * 3 2-D FIR filter

is a concatenation of eight of these operators in a two by four array, leaving always a spare row to adjust to the chip boundaries. The scan window made of the DPU registers now consists of ten words in x direction and three words in y direction, instead of the three by three words of a single filter operator.

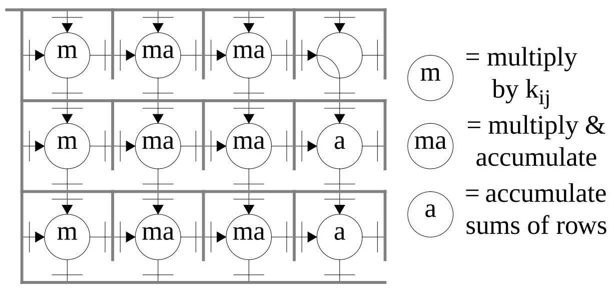

Figure 8. rALU operator for a single 3 * 3 2-D FIR filter operation

During vectorization, the compiler performs a partial unrolling of the inner loop, so that eight consecutive loop iterations are executed in parallel, and the loop index x is incremented by eight. The number of vectorizable loop iterations is taken from the capacity of the rDPA array, which can implement eight 3 * 3 FIR filter operators in parallel. Furthermore, the C compiler knows from the hardware configuration file the number of available C-Modules and splits the input array into a corresponding number of stripes. The required overlap of two can be computed from the data dependency analysis. That way all C-Modules may operate in parallel to speed-up computation by a factor of seven, at the most. The ALE-X assembler detects, that most of the input values to the eight parallel expressions are used several times in different multiplications, and stores these values in the register file for quick access. Furthermore, the two input values in x direction, that overlap with the next loop iteration are read from the register file as long as the pipeline is full throughout the inner loop. The compiler generates a Memory Address Generator configuration that first fills the pipeline by reading ten columns of three input values. Within the inner loop, only eight columns have to be read from memory. The overlapping two columns are taken from the register file.

The only rearrangements to the data are the distribution of the stripes onto the C-Modules. Therefore the data map is quite straightforward. The input values are stored row by row, each strip of rows in a separate C-Module memory. All Generic Address Generators compute the same address sequence (figure 10). Scanning the memory with a three by ten window row by row, at each scan window position ten input values are read from memory and eight values are written as results. All values required at overlapping scan window positions are fetched from the regis-

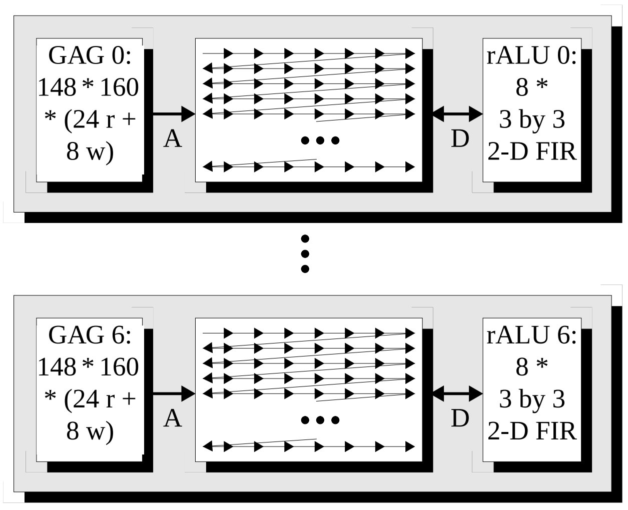
GAG 0: 148 * 160 * (24 r + 8 w)
A
D
rALU 0: 8 * 3 by 3 2-D FIR
GAG 6: 148 * 160 * (24 r + 8 w)
A
D
rALU 6: 8 * 3 by 3 2-D FIR

Figure 10. Compiler generated scan pattern for a 3 * 3 2-D FIR filter

ter file for subsequent accesses. The row by row scan pattern for the scan windows exactly matches the two nested for-loops of the algorithm in figure 7, with the loop index of the outer loop being incremented by eight.

Knowing this structure of the compiler output, a performance estimation is quite simple. Because eight filter operations are performed in parallel, the computation time for the multiplication is not the bottleneck, but the memory and register file access times to provide the large number of input data. For each group of eight result values, 3 * 8 = 24 inputs have to be fetched from memory, each memory access taking 120 ns. Two columns of inputs are reused in the next loop iteration and can be fetched from the register file in 60 ns each. Two of the ten input columns are used in the computation of two results, requiring 2 * 3 = 6 additional register file accesses. Six input columns are used, each in three result computations, totalling in another 6 * 3 * 2 = 36 register file accesses. The total time to compute eight result values and store them in memory is 24 * 120 ns + 8 * 120 ns + 6 * 60 ns + 6 * 60 ns + 36 * 60 ns = 6720 ns. A single row includes 1280 - 2 = 1278 positions. Eight positions are computed in parallel, resulting in 1278 / 8 = 160 iterations of the inner loop. One stripe takes (1024 + 6 * 2) / 7 = 148 rows, which is equal to the number of iterations of the outer loop. Using seven C-Modules in parallel, it takes 148 * 160 * 6720 ns = 0.159 s to filter a 1280 by 1024 pixel image. The filter weights may be chosen arbitrarily, because the execution time is bound by memory I/O speed. On the MoM-3, the result would be computed in the same time, regardless whether a pixel is 8 bits wide or 32 bits, because even 32-bit multiplications would be faster than memory access time in this example. The corresponding performance figures for von Neumann computers in table 1 were based on the same C source. All variables were declared as registers, the ones with the highest reference count first. Using GNU gcc compiler, all optimizations were turned on, including the exact specification of the microprocessor type.

4.2 Manually Optimized Implementation

From the algorithm description in figure 7, the MoM-3 C compiler cannot make use of all optimization possibilities. If the input data is only 8 bits wide (a typical case for most grayscale images), four values could be transferred in a single 32 bit memory access, reducing I/O time by a factor of four. The data map for a packed representation of a 1280 by 1024 pixel image, using eight bits per pixel, can be seen in figure 11.

Instead of using the register file, values which are required in subsequent loop iterations could be passed to the next DPU using the neighbour to neighbour intercon-

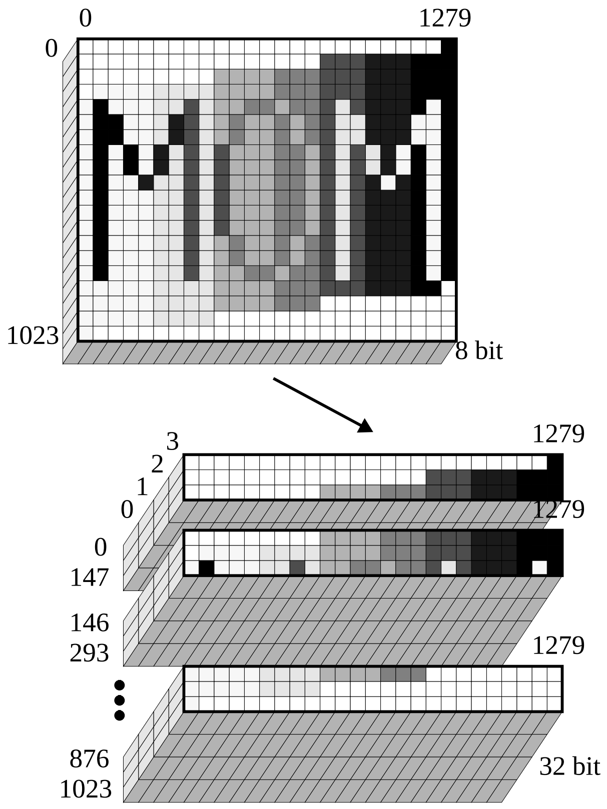

Figure 11. **Data map for an optimized 2-D FIR filter**

nect. Since these transfers can be done in parallel, they are much faster than register file accesses. Furthermore, if the eight FIR filter operators would compute the results for eight rows in parallel, instead of eight results in a single row, the neighbour to neighbour connections could be utilized to a far larger extend. Instead of six overlapping positions in the next loop iteration, twenty positions would overlap, so that fewer inputs would have to be read from memory and more inputs would be passed on from DPU to DPU.

The straightforward compiler implementation doesn't make use of the fact, that the next row of results requires to read again most of the inputs, which have been read during the computation of the previous row of results. Computing eight rows of results in parallel, the reused inputs can be fetched from the register file instead of the memory. The rALU configuration and the scan window corresponding to these manual optimizations of the filtering algorithm can be found in figure 12.

In order to achieve such optimizations, manual changes to the input source are required. The passing of the input values to the next DPU for the following loop iteration has to be specified explicitly. Since the compiler performs a vectorization, proceeding from the inner loops to the outer loops, the parallelization of the row computations has to

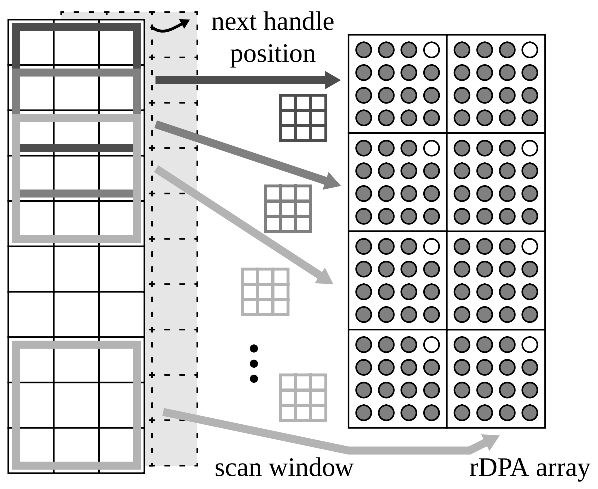

Figure 12. Manually optimized scan window and rALU configuration

be done explicitly. Packed transfers of four 8-bit inputs in one 32-bit memory access cannot be specified solely in the C source. The unpacking of 32 bit values can be done in a single DPU, which accepts a 32 bit value and four times passes an eight bit value to the next DPU for computation. But this has to be done in the ALE-X assembler source. A description of that operation sequence in C would not pass unchanged through the compiler's transformation and optimizations steps, so that the ALE-X assembler would not recognize an unpack operator in the resulting description and merge it to a single DPU.

An rALU operator (figure 13) to compute a single result value consists of three unpacking DPUs ("u"), a three by three array of multiply-accumulate DPUs ("m" and "ma"), and three DPUs to accumulate the sums of rows ("a"). The last of these DPUs performs the packing of four result values to a 32 bit word as well ("ap"). The partial sums of rows are transferred from west to east to the next DPU. The operation is pipelined on expression

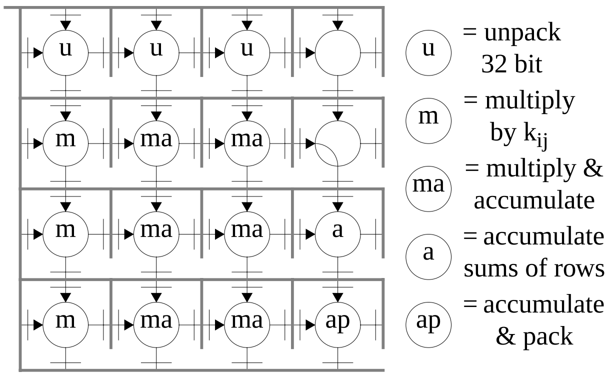

Figure 13. rALU operator for a 3 by 3 filter operation (manually optimized)

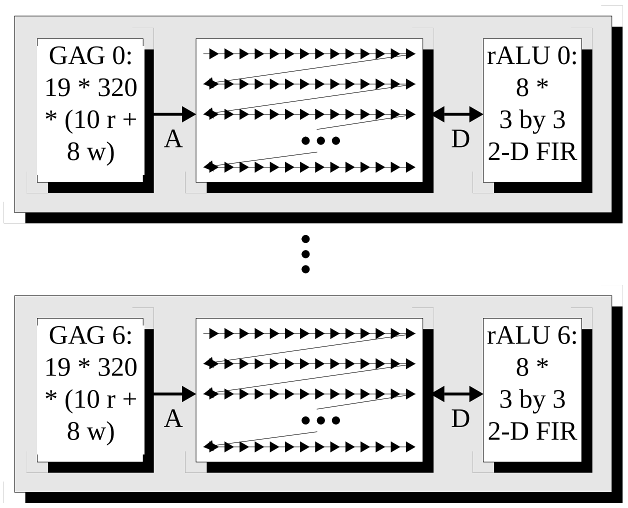
GAG 0: 19 * 320 * (10 r + 8 w)
A
D
rALU 0: 8 * 3 by 3 2-D FIR
GAG 6: 19 * 320 * (10 r + 8 w)
A
D
rALU 6: 8 * 3 by 3 2-D FIR

Figure 14. Scan Pattern for a 2-D FIR Filter

level, because each multiply-accumulate DPU accepts the next input operand (of the subsequent loop iteration) after it has calculated its partial sum and passed the result to the next DPU to the east. That way, the whole rDPA array can be kept busy, with filter operators of consecutive loop iterations finished to different degrees in the pipeline. One filter operator fits into a single rDPA chip, so that the parallel-to-serial converters do not impose a delay on the operation. Eight of these operators fit into the rDPA array of a single C-Module.

The scan window moves one position to the right in the inner loop, but eight positions downwards in the outer loop, because eight rows are processed in parallel (figure 14). That way the overlapping inputs in each row can be transferred in parallel to the next DPU to the south, where they are required as inputs for the following iteration. The overlapping inputs of the adjacent rows of the eight operators are fetched from memory only once and then taken from the register file in half of the time.

One column of eight result values requires the following time for I/O: ten values have to be read from memory, and eight written back. Furthermore, two input values are fetched a second time from the register file and six values are fetched twice from the register file for reuse in another filter operator. The total I/O time is 3000 / 4 = 750 ns. The division by four stems from the fact, that every memory or register access transfers the values for four iterations. One pipeline stage of the multiplication and accumulation takes 420 ns in the average case, because the multiplications with constants are resolved to shift and add operations. The parallel-to-serial converters, taking 600 ns, do not account, because each filter operator is local to one rDPA chip. Although optimized, this implementation is still I/O bound. The optimized algorithm takes 148 / 8 = 19 iterations of the outer loop. The inner loop contains 1280 -2 = 1278 iterations, each producing a column of eight results.

```
For (n = 0; n < maxn; n++) {
   x00=x[n][0];   x01=x[n][1];
   x10=x[n+1][0]; x11=x[n+1][1];
   x20=x[n+2][0]; x21=x[n+2][1];
   For (m = 0; m < maxm; m += 3) {
      x02 = x[n][m+2];
      x12 = x[n+1][m+2];
      x22 = x[n+2][m+2];
      y[n][m] =
         k00*x00+k01*x01+k02*x02
       + k10*x10+k11*x11+k12*x12
       + k20*x20+k21*x21+k22*x22;
      x00 = x[n][m+3];
      x10 = x[n+1][m+3];
      x20 = x[n+2][m+3];
      y[n][m+1] =
         k00*x01+k01*x02+k02*x00
       + k10*x11+k11*x12+k12*x10
       + k20*x21+k21*x22+k22*x20;
      x01 = x[n][m+4];
      x11 = x[n+1][m+4];
      x21 = x[n+2][m+4];
      y[n][m+2] =
         k00*x02+k01*x00+k02*x01
       + k10*x12+k11*x10+k12*x11
       + k20*x22+k21*x20+k22*x21;
   }
}
```

Figure 15. Manually Optimized C Source Code

The total time to filter a 1280 by 1024 8-bit grayscale image is 19 * 1278 * 750 ns = 0,018 s.

For a fair comparison, the C source for the von-Neumann computers is manually optimized as well. Instead of copying the input values to the next position, three iterations of the inner loop are unrolled and a cyclic buffering scheme is applied. This eliminates the copy time for the microprocessors, because they cannot rely on a register to register interconnect to do the copy operations in parallel. The resulting source code can be seen in figure 15. The “x” array is declared as “unsigned character” of course, to allow the compiler to do the same optimizations on multiplications as on the MoM-3. The “xNM” variables and the loop counters are declared as register variables, where the order of declarations takes into account, how often each variable is referenced. In this source code, there is no explicit unrolling of the outer loop like in the optimized MoM-3 source, because a standard microprocessor cannot process multiple FIR filter operators in parallel. Table 1 reveals, that the manual optimizations improve performance on the conventional computers by more than a factor of two.

5. Performance Evaluation

The performance figures for some example algorithms are given in table 1. The first column lists the algorithms. MergeSort performs a sorting of 256k = 262144 32-bit integer numbers. On the MoM-3, initially six C-Modules operate in parallel, but the last merging steps are done in the memory of a single C-Module. The CPU times for the conventional computers are measured only for the time to sort the numbers in memory. The two-dimensional FIR filter algorithms are measured for different kernel sizes. For all implementations, the weights of the kernel are constants compiled into the code, allowing the compilers to replace multiplications with optimized shift and add sequences. Input to the filter is a 1280 by 1024 pixel grayscale image using 8 bits per pixel. Only the time to filter the image in memory is counted, excluding disk I/O operations. The first group of performance figures are measured on the output of the compilers, with all optimizations turned on. The second group of figures are measured on manually optimized code, that takes into account that the same values are multiplied to different weights in succeeding steps of the filter algorithm (see section 4.2). This

Algorithms	68020, 16 MHz [s]	Sparc 10, 50 MHz [s]	MoM-3, 33 MHz [s]
MergeSort[a]	17.85	1.20	0.653
3x3 2-D FIR[b]	365.13	2.65	0.159
5x5 2-D FIR[b]	1088.20	4.73	0.368
7x7 2-D FIR[b]	1784.30	7.96	0.674
3x3 2-D FIR[c]	167.14	0.71	0.018
5x5 2-D FIR[c]	451.38	1.88	0.038
7x7 2-D FIR[c]	743.38	3.60	0.058
JPEG[d]	74.50	1.51	0.128

Table 1. CPU time required (in seconds)

a. sort 262144 32-Bit integers in memory
b. 1280 * 1024 grayscale image, 8 bits per pixel, in memory, compiler optimizations only
c. 1280 * 1024 grayscale image, 8 bits per pixel, in memory, manually optimized
d. IJG cjpeg compression, 704 * 576 RGB image, 24 bits per pixel, excluding disk I/O

allows to reduce memory cycles by storing these values internally. Multiplications and accumulations are done in the same DPU. The additional cost of the addition is weighed out far by the reduction of the required rDPA array size. Since all DPUs fit into a single chip in the 3 * 3 case, the comparably high cost (in execution time) of the serial links can be avoided. The MoM-3 uses all seven C-Modules in parallel on overlapping strips of the input image to speed up the filter operation. The JPEG compression algorithm is based on the IJG program "cjpeg" [9], inserting time measuring code around the compression

Algorithms	MoM-3 vs. 68020, 16 MHz	MoM-3 vs. Sparc 10/51	modern MoM-3 vs. Sparc 10/51[a]
MergeSort	27.3	1.84	3.81[b]
3x3 2-D FIR[c]	2296	16.7	33.3[d]
5x5 2-D FIR[c]	2957	12.9	25.7[d]
7x7 2-D FIR[c]	2647	11.8	23.6[d]
3x3 2-D FIR[e]	9286	39.4	78.9[d]
5x5 2-D FIR[e]	11878	49.5	145[f]
7x7 2-D FIR[e]	12817	62.0	189[f]
JPEG	582	11.8	23.6[d]

Table 2. Speed-up Factors

a. Sparc 10/51 vs. MoM-3 in Sparc's technology
b. critical factor is memory access time: 60 ns vs 120 ns DRAM access, 30 ns vs. 60 ns SRAM (register file) access, in the first stage of the algorithm, the critical factor is the speed of the serial links, which changes to memory access time for new technology, therefore more than a factor of two additional speed-up
c. compiler optimizations only
d. critical factor is solely memory access time: additional speed-up of two
e. manually optimized
f. critical factor is speed of serial links: 4 bit vs 2 bit with larger packages, and 66 MHz vs 33 MHz serial link clock, but only an additional speed-up of approximately three, because with new technology critical factor is memory access time

kernel, after the input image is read. To exclude disk output, output is redirected to "/dev/null". A 704 by 576 RGB colour image using 24 bits per pixel is used as input for the time measurements. The JPEG compression algorithm is adapted to the MoM-3 so that seven C-Modules operate in parallel. A more detailed description of the MoM-3 implementation of this algorithm can be found in [5]. The second column of table 1 gives the CPU times in seconds for the ELTEC host, which runs an MC68020 at 16 MHz. The third column describes the performance of a SPARCstation 10/51 running at 50 MHz. The CPU times of the conventional computers were all based on compilations (GNU gcc) with all optimizations turned on, producing output specially adapted to exactly the kind of processor inside the computer. The last column indicates the time to run the same algorithm on the MoM-3 prototype (33 MHz version).

Table 2 illustrates the speed-up factors obtained compared to the hosting ELTEC workstation and a modern SUN Sparc 10/51. The last column takes into account, that our prototype is built with an inferior technology compared to a modern Sparcstation. The notes at the bottom of the table explain how the extra speed-up factors would be achieved.

The rather minor speed-up for the MergeSort application stems from the fact, that the sorting is not really computation intensive and can only be parallelized and pipelined at the first stages. Most of the execution time is spent in the merging process, where most of the rDPA hardware is idle. The whole algorithm is dominated by memory access time, a reason why the Sparc 10 provides comparably small speed-ups over the MC68020, as well.

6. Conclusions

The MoM-3, a configurable accelerator has been presented, which provides acceleration factors of three orders of magnitude for its outdated host computer, and speed-ups in the range of 12 to 62 when compared to a state of the art workstation. High acceleration factors can still be maintained, when working from a high level input language. The compilation is done from an almost complete subset of standard C language. Good results are achieved, without requiring user interaction or special compiler directives to generate code for field-programmable devices and the controlling hardware of the MoM-3. The custom designed rDPA circuit provides a coarse grain field-programmable hardware, especially suited for 32-bit datapaths for pipelined arithmetic. A new sequencing paradigm supports multiple address generators and a loose coupling between sequencing mechanism and ALU. The address generation under hardware control leaves all memory accesses free for data transfers, providing de facto a higher memory bandwidth from the same memory hardware. The loose coupling of the data sequencing paradigm allows to fully exploit the benefits of reconfigurable computing devices. The combination of hardwired address generators and configurable devices is the key to speed up both data manipulations and data accesses - and still maintain a programming environment familiar to a conventional software developer.

The MoM-3 prototype is currently being built. The address generators, the MoM-3 controller and the rALU control chip have just returned from fabrication and are now under test. All three are integrated circuits based on 1.0 m CMOS standard cells, a technology made available by the EUROCHIP project of the CEC. The rDPA circuits will be submitted to fabrication in a 0.7 m CMOS standard cell process soon. All MoM-3 performance figures were obtained from simulations of the completed circuit designs, which were integrated into a simulation environment, describing the whole MoM-3 at functional level.

References

[1] J. M. Arnold, D. A. Buell, E. G. Davis: Splash-2; 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPPA '92), pp. 316-322, ACM Press, San Diego, CA, June/July 1992

[2] P. Bertin, D. Roncin, J. Vuillemin: Introduction to Programmable Active Memories; DIGITAL PRL Research Report 3, DIGITAL Paris Research Laboratory, June 1989

[3] D. Bursky: Codec compresses images in real time; Electronic Design, vol. 41, no. 20, pp.123-124, Oct. 1993

[4] S. Casselman: Virtual Computing and The Virtual Computer; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'93, IEEE Computer Society Press, Napa, CA, pp. 43-48, April 1993

[5] R. W. Hartenstein, J. Becker, R. Kress, H. Reinig, K. Schmidt: A Reconfigurable Machine for Applications in Image and Video Compression; European Symposium on Advanced Services and Networks / Conference on Compression Techniques and Standards for Image and Video Communications, Amsterdam, March 1995

[6] R. W. Hartenstein, R. Kress, H. Reinig: A Reconfigurable Data-Driven ALU for Xputers; Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'94, Napa, CA, April 1994

[7] R. W. Hartenstein, A. G. Hirschbiel, K. Schmidt, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High-Performance Hardware; Future Generation Systems 7 (1991/92), p. 181-198, Elsevier Science Publishers, North-Holland, 1992

[8] Jaap Hollenberg: The CRAY-2 Computer System; Supercomputer 8/9, pp. 17–22, July/September 1985

[9] T. G. Lane: cjpeg software, release 5; Independent JPEG Group (IJG), Sept. 1994

[10] N.N.: The CHS2x4: The World's First Custom Computer; Algotronix Ltd., 1991

[11] N.N.: The XC4000 Data Book; Xilinx, Inc., 1994

[12] A. Razavi, I. Shenberg, D. Seltz, D. Fronczak: A High Performance JPEG Image Compression Chip Set for Multimedia Applications; Proceedings of the SPIE - The International Society for Optical Engineering, vol.1903, p.165-74, 1993

[13] K. Schmidt: A Restructuring Compiler for Xputers; Ph. D. Thesis, University of Kaiserslautern, 1994