OPAC : a floating-point coprocessor dedicated to compute-bound kernels

André Seznec, Karl Courtel

UNITÉ DE RECHERCHE
INRIA-RENNES

Rapports de Recherche

N° 1555

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

OPAC : A FLOATING-POINT COPROCESSOR DEDICATED TO COMPUTE-BOUND KERNELS

André SEZNEC
Karl COURTEL

Novembre 1991

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

OPAC: a floating-point coprocessor dedicated to compute-bound kernels *

André Seznec, Karl Courtel[†]
IRISA, Campus de Beaulieu
35042 Rennes Cedex
FRANCE
e-mail : seznec@irisa.irisa.fr

October 7, 1991

Publication Interne n°613 - Octobre 1991 - 28 pages - Programme I

OPAC : UN COPROCESSEUR FLOTTANT DEDIE AU CALCUL MATRICIEL

Résumé

Dans d'importants domaines d'applications, les programmeurs ne sont pas des spe'cialistes en calcul parallèle, mais ont tout de même besoin de performances élevées qui ne peuvent être atteintes sans utiliser de parallélisme matériel. Pourtant, dans de nombreuses applications, la majeure partie des calculs peut être réunie dans des appels à des primitives de calculs "compute-bound".

Dans cet article, nous présentons l'architecture du coprocesseur OPAC. OPAC est un prototype d'accélérateur flottant développé fortement pipeliné à l'IRISA entre 1988 et 1991. OPAC a été concu comme la cellule de base d'un coprocesseur dédié a' l'exe'cution des noyaux "compute-bound".

Des performances proches d'une multiplication-accumulation flottante par cycle et par cellule sur des applications telles que résolution de système linéaire, FFTs, corrélations, .. peuvent être atteintes dans un environnement microprocesseur sur un coprocesseur dédié composé de plusieurs cellules de type OPAC.

Abstract

In various application domains, programmers are not specialists of parallel programming, but are demanding for performance that cannot be reached without using parallelism . Nevertheless, in many applications, the main part of the computations may be encapsulated in

[*] This work was partially supported by the French ministry of defense under grant DRET-INRIA No 88.34.191.00.470.75.01 and the CNRS (PRC-ANM and GCIS)

[†] at present, BULL S.A., Les Clayes-sous-Bois

compute-bounds kernels with exhibit high potential parallelism. Achieving high performance on compute-bound primitives at a low hardware cost has became an important challenge.

In this paper, we present the architecture of the OPAC floating-point operator. OPAC has been designed in order to be the basic cell in a multi-cell floating-point coprocessor dedicated to the execution of the most useful compute-bound kernels.

The peak performance of one floating-point multiply-add per cycle per cell obtained on the OPAC prototype may be approached in a microprocessor environment on a multi-cell OPAC floating-point coprocessor on a large set of numerical applications.

Keywords

array processing, floating-point coprocessor, compute-bound kernels, OPAC architecture

1 Introduction

In application domains such as signal and image processing or numerical applications on dense structure, demand for computing power is always increasing : this demand cannot be satisfied without using hardware parallelism.

But application programmers in these domains are not always specialists of parallel programming. Nevertheless, in many applications, the main part of the computations may be encapsulated in calls to primitives working on arrays and even in compute-bounds kernels where the number of operations largely exceeds the number of data and where the potential parallelism is high.

As a compromise, the architect may ask to the programmer to forget the parallel execution of its applications, but to structure its applications in order to use calls for compute-bound kernels primitives as for example those included in the de-facto standard BLAS level 3 [Do88]. The parallel execution of the programs may be then be hidden to the programmer.

When considering such an approach, achieving high performance on array processing at low cost becomes an important challenge. It may be envisaged to associate a monoprocessor host (e.g. a state-of-the-art microprocessor) with a special floating-point coprocessor dedicated to array processsing. Such a floating-point coprocessor must be built with several floating-point operators.

In this paper, we present the OPAC floating-point operator. OPAC may be used as a basic cell in order to built a multi-cell horizontal floating-point coprocessor. A prototype of OPAC has been built at IRISA during the years 1988-91.

In section 2, we list some of the most useful compute-bound kernels. In section 3, we recall the principal types of architectures that have been envisaged to achieve performance on compute-bound kernels at low cost. In section 4, we isolate some minimum characteristics that a floating-point coprocessor must respect in order to achieve nearly peak performance on calls to library of compute-bound kernels : an addressable local memory, asynchronous and independent sequencing ..

Then, in section 5, we present the architecture of the floating-point operator OPAC. OPAC has been designed in order to be used as a basic cell in a floating-point coprocessor dedicated to the execution of libraries of compute-bound kernels : parameters (i.e. size of arrays) for the calls are unknown at compile time.

FIFO queues are used in the OPAC operator for storing intermediate results and reusable operands : these FIFO queues are not used as interface buffers, but as memories which are implicitly written and read with stride one.

The floating-point operator OPAC can achieve near one floating-point multiply-add per cycle on a large set of numerical primitives including the whole library BLAS LEVEL 3, FFTs, correlations with a very limited amount of data exchanges with the host.

In section 6, simulation results show that the peak performance of one floating-point multiply-add per cycle per cell obtained on the OPAC prototype may be approached in a microprocessor environment on a multi-cell OPAC floating-point coprocessor on a large set of *effective* applications such as solving dense linear systems, dense eigenvalue or singular value problems, correlations, convolutions...

2 A set of useful compete-bound primitives

Generally, application programmers are not specialist of parallel processing; then the performance on numerical kernels working on dense structure must be accessible through standard high level languages (Fortran or C) or through numerical libraries which ranges from low level kernels (for instance BLAS LEVEL 3 [Do88]) to sophisticated algorithms (for instance LAPACK [An90]).

Ideally, the programmer may see its application as a sequential program with calls to routines. Low level primitives are used to shield the user from the hardware complexity of the architecture.

Here we list some of the most useful compute-bound primitives.

2.1 Matrix multiplication and the BLAS LEVEL 3 library

As most numerical computers reach their best performance (in terms of Mflops/s) on matrix multiplications, there has been a particular effort to express linear algebra algorithms in such a way that most of the computations is carried in matrix multiplications [Ja86]. In fact, most of the linear algebra algorithms on dense matrices can be rewritten in block algorithms; here we listed some of these applications :

- Direct methods for solving linear systems : Gauss method, Gauss-Jordan method, LU decomposition, Cholesky decomposition, ..

- Orthogonalization algorithms : Gram-Schmidt method, Polar decomposition via an iterative method [Ph87], ..

It is noteworthy that an efficient matrix decomposition may be useful also for operations on banded matrices and even in a few methods for solving sparse linear systems [Ch87][Ol80].

The BLAS LEVEL 3 [Do88] library also contains other useful primitives such as multiplication of a full matrix by a triangular matrix (or its opposite); some other matrix operations can also be interesting such as operations on banded matrices. Most of the subroutines of LINPACK [Do79] and EISPACK have already been rewritten in order to encapsulate the major part of the computations in calls to routines of BLAS LEVEL 3 [An90].

The BLAS 3 library tends to become a de-facto standard for achieving high performance on numerical applications.

2.2 Fast Fourier Transform

Fourier transform is one of the most popular methods in signal processing. It is very efficiently implemented by the FFT algorithm [Co65], nevertheless the FFT algorithm remains compute-bound : 2^n complex data read and written, 2^{n-1} complex coefficients referenced and $5n * 2^n$ floating point operations executed. Moreover, in many cases, the Fourier transform has to be applied to a set of vectors : coefficients may be read one time, then the asymptotic average number of floating point operations per memory access is $5n/4$.

2.3 Correlations

As other useful primitives, one can also enumerate convolutions and derived algorithms (correlations, filtering, ..). On all these algorithms the number of operations is large besides the number of needed data.

3 Architectures for executing compute-bound kernels at low cost

In order to achieve high performance on compute-bound kernels, parallel hardware must be used. But the characteristics of these kernels push to try to limit hardware duplication and particularly to avoid the replication of the memory system : in a general purpose high-performance systems , the memory system represents the major hardware cost.

3.1 Systolic arrays

At the beginning of eighties, the most popular approach in order to obtain performance at a low hardware cost on compute-bound kernels was the systolic concept developed by H.T. Kung at Carnegie-Mellon [Ku82]. It is based on massive and regular parallelism and pipelining : data flow through the array, but are not stored in the cells.

The original systolic concept was based on the following axioms:

- Silicon will become cheaper and cheaper.
- Control is expensive on silicon
- Large number of *slow* processors with very simple control may be statically organized for executing a specific compute-bound algorithm.

For examples of systolic arrays, see [Ku82].

Objectives reasons have lead to give up this approach for numerical applications at the end of the eighties.

- A particular systolic array is adapted to a specific algorithm and a specific size of the problem : in order to accelerate a whole application, several systolic arrays would be needed.

- In most of the algorithms, floating-point arithmetic is needed : a large number of transistors is necessary for implementing floating-point operators. In terms of silicon cost, adding some control functions and local storage supports is not prohibitive.

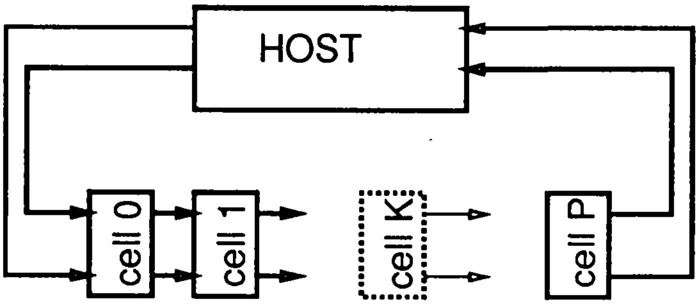
- The silicon has become cheaper and cheaper, but it has also become faster and faster : the throughput of a floating-point operator may attained of one result per cycle with a cycle similar to the host machine. Then the problem of I/0 management on a systolic array has become critical :

  > for a 10* 10 two-dimensional systolic array designed for updating matrices, 40 words of data must enter or exit the array on each cycle. memory accesses are needed per cycle.

3.2 Programmable "systolic" arrays

In the mid-eighties, H.T. Kung [An87] gave up the basic concept of systolic arrays and developed a linear programmable "systolic" array : the Warp. Fig. 1 described the linear organization of the Warp array.

Figure 1: Organization of the Warp array



The basic cell in the Warp array is a complete pipeline processor including :

- floating-point operators
- integer unit
- multiport file register
- local memory

- complete sequencing unit

The Warp array can execute all the compute-bound kernels. The natural way to map a compute-bound kernel on a Warp array of P cells consists in dividing the set of the consecutive operations contributing to a single result in P consecutive subsets of operations :

each cell executes one of the subsets

Mapping compute-bound kernels such as the matrix multiply or a correlation on the Warp is quite easy; but it is quite difficult to map a complete application on the array without intermediate storage of the data in the host memory. The natural way to use the Warp array seems to use it as a coprocessor dedicated to the execution of compute-bound kernels.

3.3 Horizontal coprocessor array versus linear coprocessor array

In most of the useful compute-bound primitives, there is an alternative to the pipeline decomposition of the algorithm : the block decomposition.

This type of algorithm decomposition leads to a different structure of a multi-cell floating-point coprocessor : each of the cells executes all the computations associated with a block and directly communicates with the host (see fig.2).

We refer to this organization of the cells in a coprocessor as a horizontal coprocessor array (versus linear coprocessor array for Warp-like organization).

The number of cells that may be associated with a single host in a horizontal coprocessor array is limited to a few tens by electrical problems.

4 Achieving performance on compute-bound kernels

From now, we consider a floating-point coprocessor built with P basic cells; we also assume that the basic cell is built around a pipelined floating-point operator which is able to deliver a throughput of one floating-point multiply-add per cycle [1].

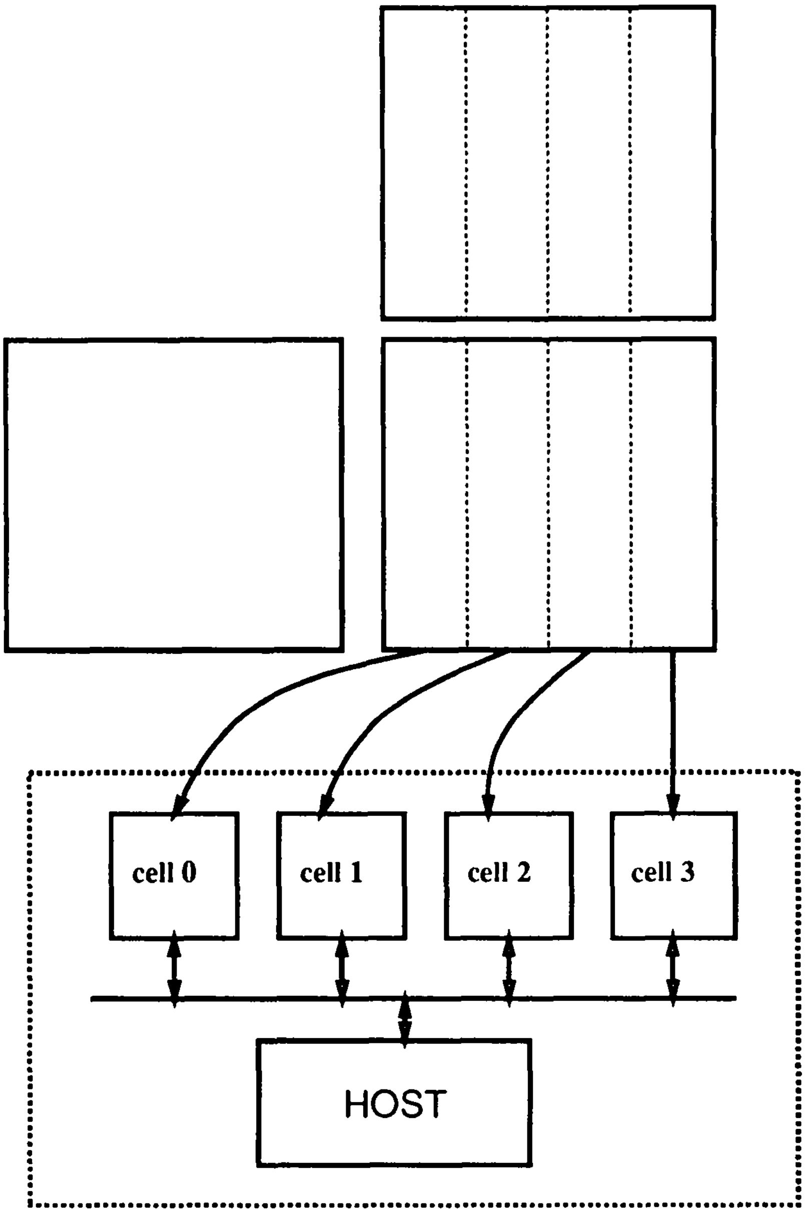
In this section, we try to determine which hardware mechanisms are needed in the basic cell of a multi-cell floating-point coprocessor in order to achieve near optimal performance on calls to libraries of compute-bound kernels in a microprocessor environment.

4.1 Data memory throughput on a microprocessor

On first generation RISC microprocessor (e.g. SPARC version 7, MIPS R3000), it was difficult to obtain an effective data memory throughput higher than one floating-point data per 3 or 4 cycles : one instruction may be issued on each cycle, but this includes loop management, address computation and memory accesses, and the instruction issuing is stopped by cache

[1] This corresponds to the current standard technology

Figure 2: Mapping a matrix multiply on a horizontal coprocessor array



misses, TLB faults, exceptions, .. On the new generation RISC microprocessor (IBM RS6000 for example), the effective memory throughput may be quite close to one floating-point data per cycle when all the data fit in the data cache [Ch90]: up to four instructions may be issued per cycle, preincremented memory accesses have been added in the instruction set.. Nevertheless, when the data do not fit in the cache, the effective memory throughput is quite inferior to the peak throughput.

Let us call τ, the average number of cycles needed by the host for accessing a floating-point data in the global memory. From now, for simplicity, we assume that the behavior of the host memory can be modelized by τ. We will consider the following values of τ :

- $\tau = 4$ for first generation RISC microprocessors
- $\tau = 2$ for current superscalar microprocessors

4.2 Need for local memory in the coprocessor

Let us now consider the matrix updating $A(N,N) = A(N,N) + B(N,N) * C(N,N)$.

If no intermediate result is stored back in the memory and if no reusable operand is read more than one time, then the minimum number of memory accesses needed to execute this matrix update is $4N^2$.

The number of operations in the matrix updating is N^3 : we consider that each cell receives N^3/P multiply-adds to execute.

In order to be able to achieve a performance close to one floating-point multiply per cell on the matrix updating, the time needed for executing the $4N^2$ memory accesses must not be critical i.e. it must not exceed N^3/P cycles i.e. $N \geq 4\tau * P$.

Now, we analyze the size of the local memory needed in each cell :

to avoid unnecessary data exchanges with the host, at least one of the three matrix operands must be stored in the local memory.

Table 4.2a and 4.2b shows the minimum value of N and the minimum size LM (in words) of the local memory in each cell for $\tau = 4$ (first generation RISC microprocessor) and for $\tau = 2$ respectively (current superscalar microprocessor).

Table 4.2a

$\tau = 4$

P	1	2	4	8	16
N	16	32	64	128	256
LM	256	512	1024	2048	4096

Table 4.2b

$\tau = 2$

P	1	2	4	8	16
N	8	16	32	64	128
LM	64	128	256	512	1024

4.3 Need for dynamic addressing on local memory

Let us consider that the local memory in each cell is only statically referenced i.e no address is computed at execution time.

If a location in this memory is used in an application, then at least one instruction must *explicitly* reference this word :

- The code volume of a primitive is directly proportionnal to the size of the local memory used by this primitive.

- Parameters of calls for a primitive are strongly constrained : the different dimensions of all the objects that are intermediately stored in the local memory must be statically defined at code generation.

  This can severely increase the complexity of writing efficient numerical kernels and/or degrade the performance when dimensions are only known at execution time (e.g. calls for library routines) : block decomposition of the algorithms in fixed size blocks induces particular treatment for remaining "small" blocks when the parameters do not fit with multiples of the "fixed" size.

  Then some dynamic addressing is needed for accessing the local memory of the cell.

4.4 Autonomous sequencing on the cells

At a first point, the sequencing of the whole coprocessor must be asynchronous with the host processor :

> As pointed out in 4.1, the sequencing of the host may be stopped by cache misses, page faults, etc,. There is no absolute need for stopping the sequencing on the coprocessor : the cells may work on data which already lie in their local memory. Effects of cache misses may be hidden.

In order to achieve high global performance, the sequencing of the distinct cells in the coprocessor may also be asynchronous. A SIMD control of the cells may be envisaged, but

- When the sequencing on the cells is synchronous, all the cells must be stopped when a data is missing at the entry of only *one* cell.

- The algorithm parameters may not fit with multiple of the cells number : the distint cells may have to execute slightly different sequences of operations.

- Two distinct hardware entities would have to be designed : the basic cell and the SIMD control unit.

Which task granularity ?

Considering the granularity of the "tasks" that can execute the cells, several different types of asynchronous sequencing may be envisaged : from *à la* "decoupled-access execute" [Sm82, Sm87] to *à la* MIMD.

Smith proposed the"Decoupled Access-Execute" architecture [Sm82, Sm87] : on Astronautics ZS-1, a global splitter dispatches "simple" instructions to two decoupled units the Access Processor and the Execute Processor (X-Processor) (more than one instruction may be sent to the same "Processor" on a single clock period) . Instructions are queued at the entry of the two "Processors". The two "Processors" communicate with the memory and with each other through FIFO queues. The execution of an instruction on one of the "Processors" may be conditioned by the presence of data on an entry FIFO queue.

Using a similar model for sequencing a P cells floating-point coprocessor and its host is quite unrealistic :

-One must at least dispatch P+1 instructions per cycle : one instruction for each floating-point cell and one for the host.

-Generating efficient code would be a nightmare : as pointed out previously, the instruction sequences in the distinct cells are sometimes different.

At the opposite, each cell may be a complete processor with sequencer, instruction memory etc, as in the Warp.

And many intermediate solutions may be envisaged.

5 The architecture of OPAC

In this section, we present the architecture of the basic cell we propose for a horizontal floating-point coprocessor array dedicated to the execution of the frequently used compute-bound kernels : the OPAC operator.

OPAC has been designed in order to be directly interfaced with a host computer. Several OPAC operators may be associated with a single host (fig.3).

The sequencing on each OPAC coprocessor and the host are completely asynchronous : FIFO queues are used for buffering operands, results and calls for numerical kernels between the host microprocessor and an OPAC operator.

5.1 Architecture of the computation block

The architecture of the computation block of the OPAC operator has been studied in order to achieve one floating-point multiply-add per cycle on the most useful compute-bound kernels (see section 2) assuming that the data throughput on connection host-OPAC may be quite limited (one word per $P * \tau$ cycles). Studying these primitives has lead us to the architecture presented in fig.4.

Figure 3: Coprocessor OPAC in its environment

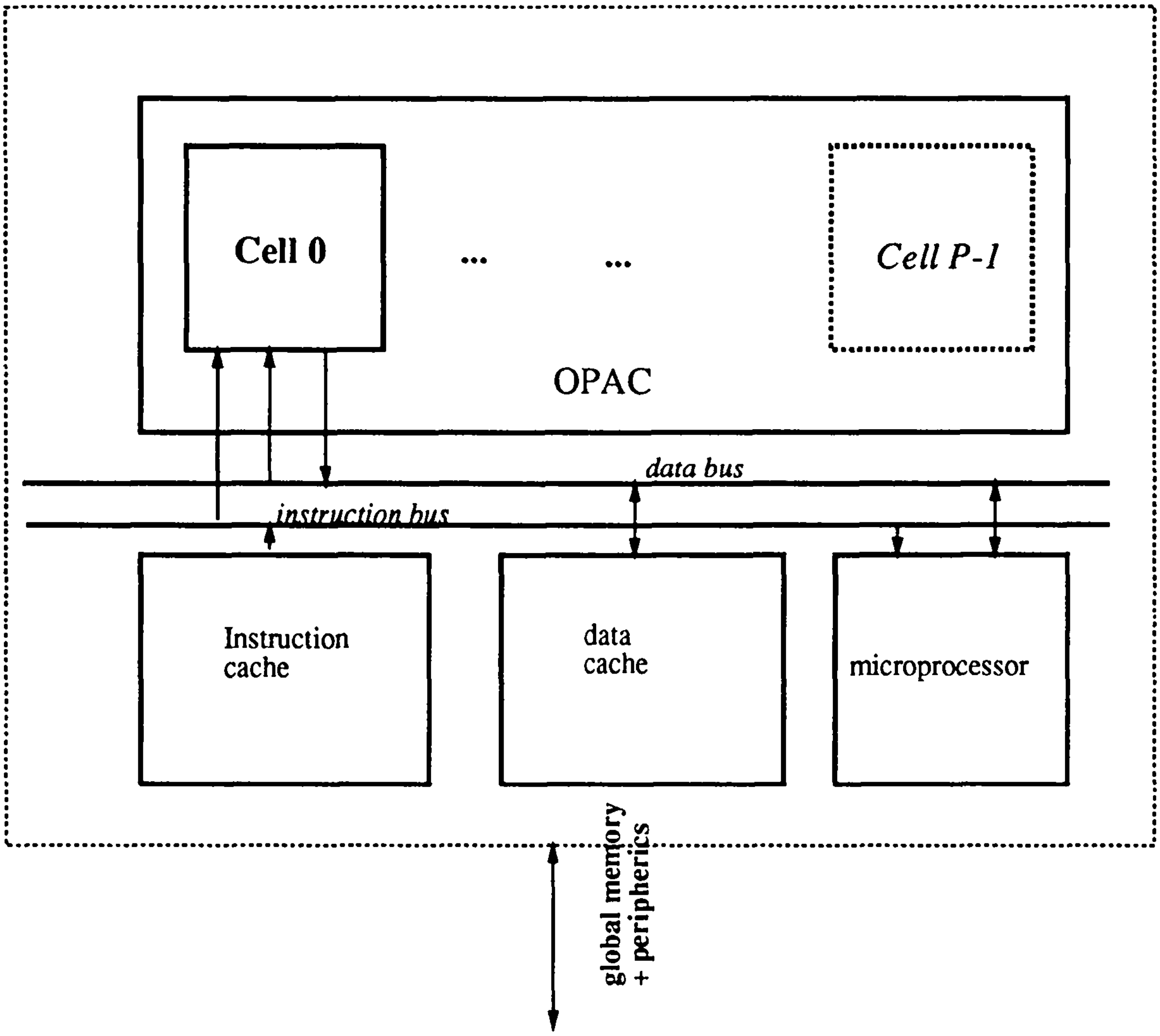
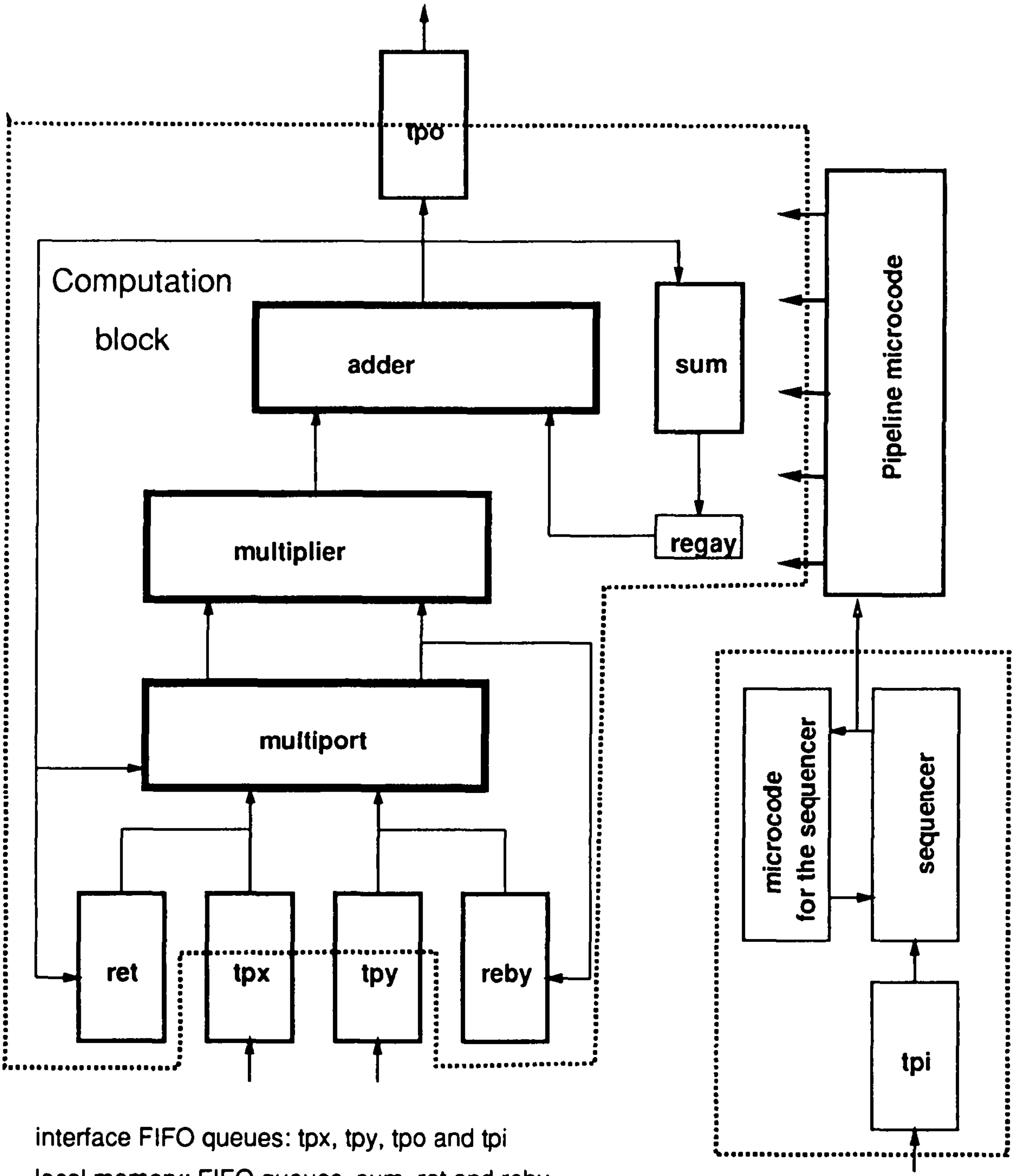


Figure 4: Architecture of the coprocessor OPAC



interface FIFO queues: tpx, tpy, tpo and tpi

local memory: FIFO queues sum, ret and reby

registers : multiport, regay.

operators: floating-point multiplier and adder

A prototype has been built with chips which were off-the-shelf in 1988. To approach our performance objective, the computation block is heavily pipelined, each element in it is supposed to be able to deliver one result per cycle.

Optimized data paths

In the set of compute-bound primitives which have been listed in section 2, in most cases, the result of a multiply is then accumulated with a previously computed intermediate result. So we have chosen to implement a direct path from the output of the multiplier to one of the input of the adder[2].

Let us point out that this simplifies the hardware and software management of the pipeline and that is a key issue for achieving good performance on calls to libraries of compute-bound routines [Se91].

Using FIFO queues as local memory

As pointed out in section 4.2, some local and reasonably large (about one Kwords) physical support is needed to store intermediate results and reusable operands.

We have also shown that static addressing of this memorization support is not a competitive approach, because it does not allow to implement efficient primitives with variable parameters and it leads to very large code size.

FIFO queues *ret*, *reby* and *sum* are used as local memory in OPAC :

- For each of the primitives listed in section 2, we have been able to find an efficient implementation where these FIFO queues are used as only local storage support and where no extra external memory access is performed.
- FIFO queues are dynamically but implicitly addressed :
  1. FIFO queues are used here as dual-port memory.
  2. Code is compact : no extra instruction is needed for address initialization.
  3. The width of the instruction word is reduced : for each FIFO queue, only the READ and WRITE information has to be coded.
- In vector sections, the difference between a vector register and a FIFO queue is very tiny; but sometimes the use of FIFO queues may be easier because it is possible to dissociate consecutive elements of a FIFO queue (for example when solving a triangular system, at each step of the outside loop, the size of the vector is decremented, and the computation on the first element is different from the computations on the other elements).

[2] This approach has become very popular since the project was started (IBM RS6000, Intel i860, ..)

- Large FIFO queue RAMs were available in 1988 (at least 2048 words of 9 bits) with fast access times (25 MHZ) and correct characteristics : a data written at t may be available on the output at $t + 80ns$. An decrease of the period and an increase of the capacity has been observed : 50 Mhz 8 Kwords FIFO queues are now available.
- In terms of hardware, using FIFO queues is a quite cheap solution versus classical local memory and address computations (address generators, data paths for initializing addresses, microcode RAMs for controlling these, etc.).
- In [Se91], we explain how the use of FIFO queues facilitates an hardware management of the pipeline, and then facilitates code generation and decreases the code volume (in number of instructions).

The locations of the three FIFO queues *sum*, *ret* and *reby* are justified by the study of the algorithms listed in section 2. FIFO queues *sum* and *ret* have been introduced to store intermediate results flowing out from the output of the adder to the second entry of the adder and the entries of the multiplier respectively. The FIFO queue *reby* has been introduced to store vectors (or matrices) of data which are used several times as an operand for a multiply (see fig.5).

Task granularity

In 4.4, we have pointed out that the granularity of the tasks executed on the cell cannot be single instruction, then some independent sequencing mechanism is needed in the cell.

We have envisaged three options :

1. Complete sequencing facilities : loop management, data memory for the sequencer, stacks.. i.e a whole program may be sequenced as in the Warp cell.
2. Only vector instructions : Compute-bound primitives may generally be decomposed in vector instructions
3. Special hardware mechanisms for sequencing compute-bound kernels

For OPAC, having complete sequencing facilities is not necessary : as OPAC is dedicated to the execution of the compute-intensive kernels of complete applications, many sequencing decisions will be taken by the host (convergence tests, ..) or replicated by the host (e.g. the number of data read on an interface FIFO queue by the operator must be exactly the same as the number of data written on it by the host).

On the other hand, being able to execute only vector instructions will lead to the sending of many instructions and parameters by the host to the different cells in the floating-point coprocessor and may saturate the host throughput. Moreover classical vector instructions may not be sufficient (for instance for performing the perfect shuffle).

As a compromise, we have chosen the compute-bound kernel as the task granularity that can be managed by the OPAC sequencer. The sequencer receives in FIFO tpI the address of the primitive in the microcode and parameters (sizes of arrays generally); a complete primitive may be managed by the sequencer : -Several loop levels may be managed
-Linear code and several loops may be chained consecutively
-Very limited manipulations on parameters are authorized : incrementation, decrementation (for solving triangular systems), multiply or divide by 2 (for FFTs)

An example of the sequencing of a matrix updating on a single OPAC operator associated with an host is illustrated in fig.5

5.2 Pipeline management : a critical issue for performance on OPAC

Let us consider the matrix updating on OPAC. In order to limit the demand on data from and to the host, intermediate results and reusable are stored in the internal FIFO queues in OPAC as shown in the algorithm illustrated in fig.5.

In this implementation, the whole updated matrix A M*N is stored in the FIFO queue sum in OPAC. If M*N exceeds the size of the FIFO queue sum then a block matrix updating algorithm is used ; sizes of blocks are chosen such that a block of A fits in the FIFO queue *sum*.

Analysis shows that using square blocks will limit data exchanges between the host and OPAC. On the current OPAC prototype, the size of the FIFO queues is 2048 words : square root = 45; in a VLSI implementation of OPAC, this size would be probably limited to 512 words (or less) : square root = 22.

For most of the compute-bound kernels we want to implement on the OPAC floating-point coprocessor, the same situation arises : due to the finite size of the FIFO queue, we need to implement block versions of the kernel and most of the computations will lie in the inner most loops with quite limited iteration numbers (less than 50 for our prototype, on the order of 20 in a VLSI implementation).

Then in order to achieve effective performance close to one floating-point multiply-add per cell on the OPAC coprocessor, we need to reach this order of performance on the basic cell when sequencing loops with a very limited iteration numbers.

Special features for the control and the sequencing have been implemented in the OPAC prototype; they allow to reach near asymptotic performance on loops with very limited iteration numbers. These mechanisms are described in [Se91].

5.3 Synthesis

The structure of the computation block of the OPAC operator allows to execute the primitives listed in section 2 with a performance close to one multiply-add per cycle without extra memory

Figure 5: Sequencing the matrix update A= A+B*C on oa single cell OPAC coprocessor and its host

Sequencing on OPAC :

```
1) Initialization :

 Load of A in the FIFO queue sum
from FIFO queue tpx

2) Computing :

For k=1 to K do
Load of vector B(. ,k) in  FIFO queue reby
                           from FIFO queue tpy

   For n=1 to N do
     For m=1 to M do
        $A(m,n)= A(m,n)+ B(m,k)*C(k,n)$
     Endfor
   Endfor
% A(m,n) is read and written on FIFO queue sum
% B(m,k) is read and written on FIFO queue reby
% Constant C(k,n) is read on FIFO queue tpx

  Reset of FIFO queue reby

Endfor

3) Sending results to the host

FIFO queue sum is emptied in FIFO  queue out
```

Sequencing on the host:

```
0) sending the call for OPAC
     and parameters M,N,K

1) Initialization:

Store matrix A in FIFO queue tpx

2) Comnuting:

For k=1 to K do

Store  B(.,k) in FIFO queue tpy

Store C(k,.) in FIFO queue tpx

Endfor

3) Store the result matrix
```

access when the sizes of reusable operand arrays and intermediate results arrays allow to store them in the internal FIFO queues.

We claim that the architecture of OPAC cell is a good approximation of the "minimum" architecture that allows to reach this level of performance with a very limited demand on I/O.

For larger parameters, the whole set of data cannot be intermediately stored in an internal FIFO queue of a single OPAC operator.

Block algorithms have to be implemented, and the primitives have to be split into calls to basic kernels working on smaller sets of data; the computations on the different blocks are dispatched among the P cells.

6 Performance

A single OPAC operator prototype has been implemented to prove the feasibility of the different proposed hardware mechanisms, particularly the controlling and sequencing mechanisms described in [Se91].

A functional simulator of a P cells OPAC coprocessor has been implemented. This simulator respects all the timings of the prototype.

We analyze here the implementation of two compute-bound kernels :

1. The matrix updating C(N,M) = C(N,M) + A(N,K)*B(K,M)

2. the two-dimensional 5*5 convolution on a 1024*1024 matrix

and a more complex application : the LU factorization.

We try to determine the influence on performance of different parameters :

1. The number P of cells in the coprocessor

2. The host memory throughput : modelized by τ the inter-access delay

3. The size Tf of the internal FIFO queues in OPAC :
   -the prototype has been realized with 2048 words FIFO queues
   -the integration of a complete OPAC operator on a single chip seems feasible; using 512 words of 32 bits FIFO queues, the complexity of a single OPAC operator has been evaluated to a million of transistors

6.1 The matrix updating

The implementation of the matrix updating that have been tested has been previously described by fig.2 (for the block decomposition among the cells) and fig.5.

When the whole result matrix is too large for being intermediately stored in the FIFO queues sum of the OPAC cells, a new level of block decomposition must be added.

We only report here results for one square block of maximum size : i.e. given Tf the size of the internal FIFO queues in an OPAC operator, P the number of cells in the coprocessor, we consider the operation A(N,N)=A(N,N)+B(N,K)*C(K,N) with N being the greatest integer verifying : N^2 is a multiple of P and $N^2 \leq Tf$.

Results from simulation are given for P=1,4 and 16, Tf=512 and 2048 words and K= 40, 100, 300 and 1000 in table 6.1 : the computing time that has been considered here is the time spent *on the host* between the sending of the first element to the coprocessor and the receipt of the last result.

The results have been normalized in multiply-adds per cycle.

Table 6.1

K	40	100	300	1000	Asymptotic
Tf= 512, $\tau = 2$					
P=1, N= 22	0.879	0.930	0.955	0.964	0.968
P=4, N=44	2.779	3.345	3.679	3.812	3.872
P=16, N=88	5.849	9.047	11.95	13.46	15.49
Tf= 512, $\tau = 4$					
P=1, N=22	0.806	0.896	0.942	0.960	0.968
P=4, N=44	2.168	2.946	3.504	3.754	3.872
P=16, N=88	3.427	5.839	8.497	10.10	11.00
Tf=2048, $\tau = 2$					
P=1, N=44	0.901	0.953	0.978	0.987	0.992
P=4, N=88	2.834	3.420	3.766	3.904	3.971
P=16, N=176	6.121	9.694	13.09	14.91	15.87
Tf= 2048, $\tau = 4$					
P=1, N=44	0.825	0.917	0.965	0.983	0.992
P=4, N=88	2.205	3.006	3.585	3.844	3.971
P=16, N=176	3.792	6.979	11.13	14.07	15.87

Let us notice that, except for the case $\tau = 4$,$Tf = 512$ and $P = 16$, the asymptotic performance is very close to one floating-point multiply-add per cycle per cell.

For $\tau = 4$, $Tf = 512$ and $P = 16$, the performance is limited by the data bandwidth of the host : 704 = 4*(88+88) cycles are needed to send a column of B and a row of C to the coprocessor when it generates only (88*88)/16 = 484 multiply-adds on each cell.

6.2 The two-dimensional 5*5 convolution

Given a matrix A(N,M), one has to compute the matrix B defined by:

$$B(n,m) = \sum^{i=0,p-1} \sum^{j=0,q-1} w(i,j) * A(n-i,m-j) \quad .$$

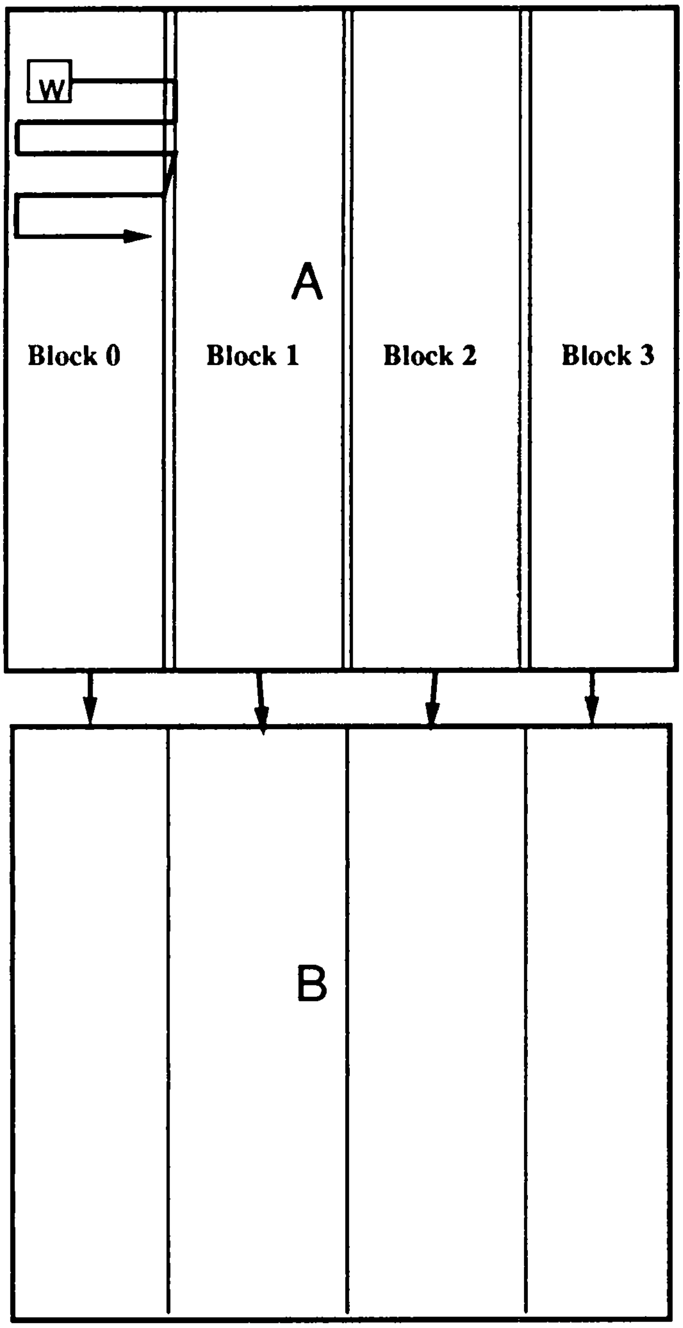
We are able to implement this algorithm on a single OPAC cell without requiring any extra global memory access from the host memory -i.e. no reusable operand is read more than one time, no intermediate result is stored back- at the condition that p rows of B + q words may be stored in the FIFO queue *sum*.

When it is not the case, we may use block decomposition of the matrix illustrated in figure 6. Blocks are sequentially computed, frontier rows of A must be emitted twice and some redundant computations are executed [3].

The same decomposition may be used to execute the algorithm on a multi-cell coprocessor.

[3] When working on a row of N elements, N+(p-1) operations are executed for only N useful

Figure 6: Block decomposition of the two-dimensional convolution



The 5*5 two-dimensional convolution requires 1 read and 1 write on the host memory per 25 multiply-adds.

In table 6.2, we give the performance measured by simulations for floating-point coprocessors consisting in 1, 4 and 16 OPAC operators, performance is normalized in *useful* multiply-adds per cycle.

Table 6.2

Tf	512	512	2048	2048
τ	2	4	2	4
P=1	0.925	0.925	0.980	0.980
P=4	3.700	2.941	3.919	3.07
P=16	5.882	2.941	5.882	2.941

Performance for the 16-cell coprocessor is clearly limited by the memory throughput : for the two considered sizes of FIFO queue, each block consists in 72 columns (the frontiers consists in 4 columns on each side) which must be emitted to two different cells : computing a row on matrix B generates 16*72 reads plus 1024 writes on the host memory (i.e. 2176 memory accesses) and only 64*25 =1600 useful multiply-adds per cell.

For $\tau = 4$, (resp. 2), it limits the global performance to 2.95 (resp. 5.9) multiply-adds per cycle. These limits are approximatively reached.

For the 4-cell coprocessor, for $\tau = 4$, the performance is also limited by the memory throughput, but when FIFO queue size is 2048 words less memory bandwidth is lost for emitting frontiers.

In the other cases, performance is limited by the throughput of the floating-point operators and by the redundant computations which are executed, but the size of the internal FIFO queues in the OPAC cell is not quite a limiting factor.

6.3 LU factorization

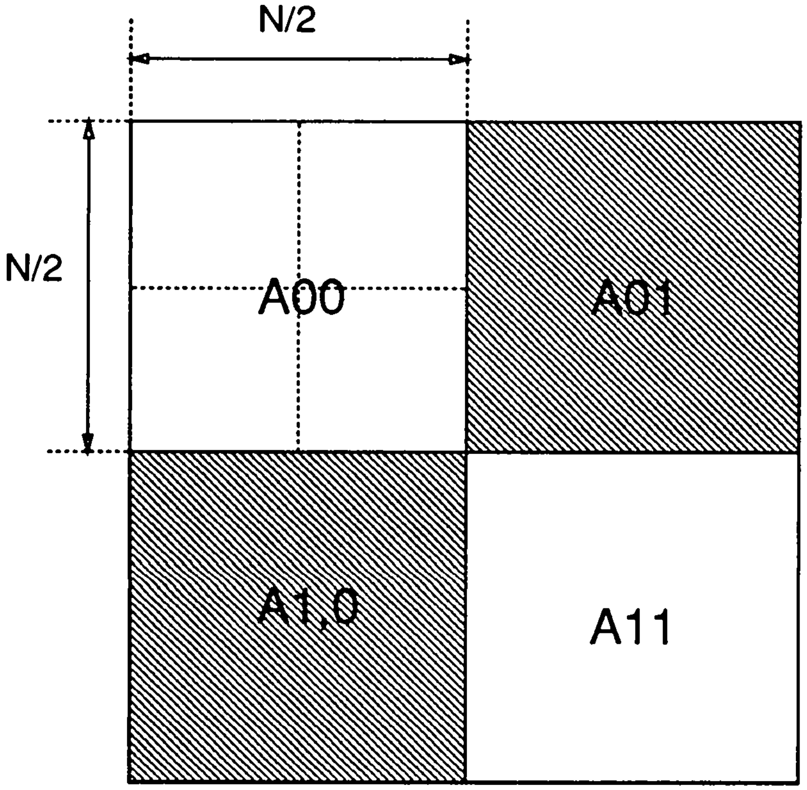
The LU factorization algorithm on an arbitrary size of matrix illustrates the way OPAC may be used on large matrix algorithms.

The LU factorization of a matrix N*N can be performed on a single OPAC operator without any extra memory access when the whole matrix can be stored in an internal FIFO queue of the coprocessor i.e $N^2 \leq 2048$ or $N \leq 45$ for our prototype or $N^2 \leq 512$ for a VLSI implementation of OPAC. Nevertheless larger LU factorization can be implemented using a block algorithm. Figure 7 illustrates one of the possible algorithm.

The different steps are :

1. LU factorization of the N/2 *N/2 submatrix A0,0.
2. update of the $(N - N/2) * N/2$ submatrix A1,0 (resolving (N-N/2) triangular systems $Tx = y$ with the same matrix T).

Figure 7: Recursive LU factorization algorithm



3. update of the $N/2*(N-N/2)$ submatrix A0,1 (multiplication of a triangular matrix by a full matrix)

4. update of the $(N-N/2)*(N-N/2)$ submatrix A1,1 by $A1,1 = A1,1 - A1,0 * A0,1$

5. LU factorization of the submatrix A1,1

If A0,0 and A1,1 are still too large to be processed by a single operator, steps 1 and 5 may be also decomposed in the same way.

Steps 2, 3 and 4 are direct calls to primitives of BLAS level 3 [Do88].

In table 6.3, simulation results are given for matrices N*N for N=44 to N= 704. All results are normalized in multiply-adds per cycle. The implementation has not been specifically optimized.

Table 6.3

N	44	88	176	352	704
Tf= 512, $\tau = 2$					
P=1	0.48	0.66	0.85	0.95	0.96
P=4	0.89	1.67	2.62	3.37	3.60
P=16	1.03	2.31	4.41	7.27	8.89
Tf= 512, $\tau = 4$					
P=1	0.44	0.62	0.81	0.93	0.94
P=4	0.74	1.33	2.20	3.14	3.40
P=16	0.74	1.38	2.50	3.89	4.63
Tf=2048, $\tau = 2$					
P=1	0.57	0.65	0.81	0.94	0.94
P=4	0.57	1.33	2.32	3.21	3.45
P=16	0.57	1.68	3.96	7.44	9.71
Tf= 2048, $\tau = 4$					
P=1	0.53	0.62	0.77	0.91	0.91
P=4	0.53	1.18	2.03	2.87	3.19
P=16	0.53	1.27	2.59	4.72	6.10

6.4 Synthesis

The results of simulations given in this section indicates that a 4-cell OPAC coprocessor associated with a standard microprocessor will reach near asymptotic performance on many algorithms and for medium size of the parameter calls : more than 3 multiply-adds per cycle on the N*N LU factorization for N=352 or for the 44*100 by 100*44 matrix updating. Influence of increasing FIFO queue size is quite marginal.

For a 16-cell OPAC coprocessor, performance per cell is worse : large size of the parameters are needed to approach the asymptotic performance, this is due to higher demand on data and a very important start-up time. The influence of the internal FIFO queue size becomes important : host memory data throughput becomes the bottleneck for performance when the $Tf = 512$ and $\tau = 4$.

Let us notice that the results presented in this section are also valid with little change for a floating-point coprocessor built with a different basic cell.

7 Conclusion

Current state-of-the-art microprocessors can approach performance of one floating-point multiply add per cycle on compute-bound primitives. Demand for computing power in array processing in many application domains are higher.

Asking to the application programmer -which may not be an expert in parallel programming, but in its application domain- to encapsulate the computations in calls to standard library of low level routines seems realistic. Then compute-bound kernels may be executed in parallel by a specialized floating-point coprocessor consisting in several floating-point cells : the programmer does not have to worry about this parallel execution.

The OPAC operator has been designed as the basic cell of such a specialized floating-point coprocessor.

When defining the OPAC operator, we have tried to design the "minimum" -in terms of integration complexity- architecture which is able to achieve one floating multiply-add per cycle and per cell on calls to the most useful compute-bound kernels in a microprocessor environment with the following constraints :

1. Data exchange bandwidth with the host is very limited
2. Compute-bound kernels are library routines : parameters (and particularly array sizes) are unknown at compile time

We have pointed out that some dynamically addressable multiport local memory is needed (see section 4.2). In the OPAC architecture, FIFO queues as dynamically but implicitly addressable local memory :

- The most useful compute-bound kernels may be mapped on the cell without needing any non-intrinsic exchange of data with the host.
- No complex address generators are needed : no data path for initializing them, no wide instruction parcels for control, etc ..

Due to special features in the control of the pipeline and in the sequencing which are described in [Se91], performance on the OPAC cell is very close to one floating-point multiply-add per cycle.

Validity of our approach has been checked in section 6 : reaching effective performance of 10 or even more floating multiply-add per cycle is possible in a microprocessor environment when using a multi-cell OPAC coprocessor.

A prototype of the OPAC operator has been realized with off-the-shelf chips.

Integration of a complete OPAC-like operator on a single chip is envisaged with an industrial partner : it would require approximatively one million transistor when considering FIFO queues of 512 32 bits words.

References

[An90] E.Anderson & al “LAPACK : A portable linear algebra library for high-performance computer” Proceedings of Supercomputing ’90, New-York, Nov. 1990

[An87] M. Annaratone & al “The Warp Computer: Architecture, Implementation, and Performance”, IEEE Transactions On Computers, Dec. 1987

[Ch90] D.Chen “Hierarchical blocking and data flow analysis for numerical linear algebra” Proceedings of Supercomputing ’90, New-York, Nov. 1990

[Ch87] A.T.Chronopoulos, C.W.Gear ”Implementation of s-step methods on parallel vector machines” Illinois University, 1987

[Co65] J.W. Cooley, J.W.Tukey ”An algorithm for the Machine Calculation of Complex Fourier Series” Mathematics of Computation, April 1965

[Co91] K.Courtel, “Etude et développement d’un coprocesseur de calcul matriciel:OPAC” Ph.D. Thesis, Université de Rennes I, Jan. 1991

[Do79] J.Dongarra, J.Bunch, C.Moler, G.W.Stewart LINPACK Users’ Guide, SIAM, 1979.

[Do84] J.Dongarra, FG. Gustavson, A.Karp ”Implementing linear algebra algorithms for dense matrices on a vector pipeline machine” SIAM Review 26.1 (1984) pp. 91-112.

[Do88] J.Dongara, J.DuCroz, I.Duff, S.Hammarling ”A set of level 3 Basic Linear Algebra Subprograms” Argonne Technical Report May 1988.

[Ja86] W.Jalby, U.Meier, ”Optimizing matrix operations on a parallel multiprocessor with a memory hierarchy” Proceedings ICPP 1986

[Ku82] H.T. Kung “Why systolic architecture ?” IEEE Computer, Dec. 1982

[Ol80] D.P.Oleary, ”The block conjugate gradient algorithm and related methods” Linear algebra and its application, 29, pp293-322, 1980

[Ph87] B.Philippe ”An algorithm to improve nearly orthonormal set of vectors on a vector processor”, SIAM Journal on Algebraic and Discrete Methods, Vol.8, No.3, 1987

[Se91] A.Seznec, K.Courtel "Controlling and sequencing on an heavily pipelined floating-point operator" INRIA report, Sept. 1991

[Sm82] J. E. Smith, "Decoupled Access/Execute Computer Architectures", Proceedings of 9th Annual International Symposium on Computer Architecture, pp 112-119, April 1982

[Sm87] J. E. Smith, "The ZS-1 Central Processor", Proceedings of 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), Oct 1987. pp 199-204

PI 604 A GENERAL METHOD TO DEFINE QUORUMS
Mitchell L. NEILSEN, Masaaki MIZUNO, Michel RAYNAL
Septembre 1991, 20 pages.

PI 605 OBTENTION DES EQUATIONS DYNAMIQUES D'UN SYSTEME PHYSIQUE A PARTIR DE SON MODELE BOND GRAPH
Bénédicte EDIBE
Septembre 1991, 26 pages.

PI 606 CONSTRUCTIVE PROBABILITY AND THE SIGNalea LANGUAGE : BUILDING AND HANDLING RANDOM PROCESSES VIA PROGRAMMING
Albert BENVENISTE
Septembre 1991, 60 pages.

PI 607 ABOUT LOGICAL CLOCKS FOR DISTRIBUTED SYSTEMS
Michel RAYNAL
Octobre 1991, 16 pages.

PI 608 UNE NOUVELLE APPROCHE REALISTE DE SIMULATION D'ECLAIRAGE DANS UN ENVIRONNEMENT DIFFUS
Eric LANGUENOU, Kadi BOUATOUCH, Pierre TELLIER
Octobre 1991, 70 pages.

PI 609 INTEGRATION D'UN CORRECTEUR ORTHOGRAPHIQUE DANS L'EDITEUR STRUCTURE GRIF
Patrice FRISON, Eric PICHERAL, Hélène RICHY
Octobre 1991, 22 pages.

PI 610 SYNCHRONIZATION AND CONCURRENCY MEASURES FOR DISTRIBUTED COMPUTATIONS
Michel RAYNAL
Octobre 1991, 20 pages.

PI 611 MALI v06 - TUTORIAL AND REFERENCE MANUAL
Olivier RIDOUX
Octobre 1991, 86 pages.

PI 612 SENSITIVITY COMPUTATION IN NETWORK RELIABILITY ANALYSIS
Gerardo RUBINO
Octobre 1991, 38 pages.

PI 613 OPAC : A FLOATING-POINT COPROCESSOR DEDICATED TO COMPUTE-BOUND KERNELS
André SEZNEC
Karl COURTEL
Octobre 1991, 28 pages.

PI 614 CONTROLLING AND SEQUENCING AN HEAVILY PIPELINED FLOATING-POINT OPERATOR
André SEZNEC
Karl COURTEL
Octobre 1991, 28 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399