Architecture Exploration for Efficient Data Transfer and Storage in Data-Parallel Applications

Rosilde Corvino, Abdoulaye Gamatié, and Pierre Boulet

LIFL - UMR CNRS/USTL 8022 Inria Lille - Nord Europe Parc Scientique de la Haute Borne, 40 avenue Halley Bat.A, Park Plaza 59650, Villeneuve d'Ascq, France
{rosilde.corvino,abdoulaye.gamatie,pierre.boulet}@inria.fr

Abstract. Due to the complexity of modern data parallel applications such as image processing applications, automatic approach to infer suitable and efficient hardware realizations are more and more required. Typically, the optimization of data transfer and storage micro-architecture has a key role for the data parallelism. In this paper, we propose a comprehensive method to explore the mapping of a high-level representation of an application into a customizable hardware accelerator. The high-level representation is in a language called Array-OL. The customizable architecture uses FIFO queues and double buffering mechanism to mask the latency of data transfers and external memory access. The mapping of a high-level representation onto the given architecture is performed by applying a set of loop transformations in Array-OL. A method based on integer partition is used to reduce the space of explored solutions.

Keywords: design space exploration, data parallel applications, image processing, Array-OL, hardware architecture, data management.

1 Introduction

Historically, the parallel computing has been used to model and solve complex problems related to the real world: meteorology, physics phenomena and mathematics. Today, a large part of commercial applications uses the parallel computing to process large sets of data in sophisticated ways, e.g., web searching, financial modeling and medical imaging. A major problem in designing accelerators for data parallel applications is the design of an efficient data transfer and storage micro-architecture. In fact, the applicability of the data parallelism itself depends on the possibility to easily distribute data on parallel computing resources. Furthermore, the data transfer and storage micro-architecture affects the three most important optimization criteria of a hardware accelerator design: power consumption, area occupancy and temporal performance [1].

Related Works. The problem of finding an efficient data transfer and storage micro-architecture has been largely studied in the literature and is related to several research domains. We focus on three of them: memory hierarchy for data

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 101–116, 2010.

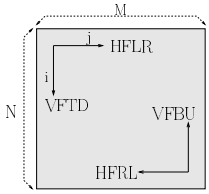
parallelism, mapping of a high level specifications onto a parallel architecture and design space exploration.

An efficient memory hierarchy exploration includes an estimation of the storage requirement and the choice of a hierarchy structure, with the corresponding data mapping. Balasa *et al.* [2], survey the works on storage requirement estimation and conclude that they are limited to the case of a single memory. The works exploring the hierarchy structure [3,4] are usually limited to the usage of cache memories. The caches are power and time consuming memories because of the *fetch-on-miss* and *cache prediction* mechanisms. We argue that an internal memory with a pre-calculated data fetching is more efficient for application specific circuits. In addition, Chen *et al.* [5] show that local memories with a "pre-execution pre-fetching" are a suitable choice to hide the I/O latency.

Many studies exist on methods to map a high-level specification of an application onto a parallel architecture. These techniques are usually loop transformations [6], which enhance data locality and allow parallelism. Data-parallel applications are usually mapped onto systolic architectures [7], i.e., nets of interconnected processors, in which data are streamed through in a rhythmic way. Amar et al [8] propose to map a high-level data parallel specification onto a Kahn Process Network (KPN), which is a network of processing nodes interconnected by FIFO queues. We argue that KPNs are not suitable to take into account the transfer of multidimensional data. The FIFO queues can be used only when two communicating processors produce and consume data in the same order. In the other cases, a local memory is necessary. When a local memory is used, the risk of conflicts on a memory location imposes that the processor producing and the processor consuming the data execute at different times. This creates a bottleneck in the pipeline execution, which can be avoided by using a double buffering mechanism [9]. Such a mechanism consists in using two buffers that can be accessed at the same time without conflict: the first buffer is written while the second one is read and vice-versa.

The complexity of data parallel architectures is so high that only an Electronic Design Automation (EDA) tool can efficiently undertake their design. One of the biggest challenge of the EDA is to perform a Design Space Exploration (DSE) able to find in a short time an optimal hardware realization of a target application. We can distinguish two kinds of approaches to perform an exploration: *exact* and *approximated* approaches. Ascia *et al.* [10] briefly survey these approaches and propose to mix them in order to reduce the exploration time and approximate the optimal solution more precisely.

Our contribution. In this paper we present an exact DSE approach to map data parallel applications onto a specific hardware accelerator. The analysis starts from a high level specification defined in a language called Array-OL [11]. The proposed method combines several optimizations from the three research domains previously presented: the memory hierarchy for data parallelism, the mapping of a high level specifications onto a parallel architecture and DSE. The method is based on a customizable architecture including parallel processors and distributed local memories used to hide the data transfer latency. The



$$HFLR: \; p_1(i,j) = a\,p_1(i,j-1) + (1-a)p_0(i,j)$$
$$VFTD: \; p_2(i,j) = a\,p_2(i-1,j) + (1-a)p_1(i,j)$$
$$HFRL: \; p_3(i,M-j) = a\,p_3(i,M-j-1) + (1-a)p_2(i,M-j)$$
$$VFBU: \; p_4(N-i,j) = a\,p_4(N-i-1,j) + (1-a)p_3(N-i,j)$$

Fig. 1. Specification of a Low Pass Spatial Filter: HFLR, VFTD, HFRL and VFBU respectively stand for Horizontal Filter Left Right, Vertical Filter Top Down, HF Right Left and VF Bottom Up. $p_t(i,j)$ is a pixel of coordinates (i,j) at the time t. M and N are the horizontal and vertical sizes of the filtered image. a is the filtering factor.

target application is transformed in order to enhance data parallelism through data partitioning. The blocks of data are rhythmically streamed into the architecture. Several parallelism levels are possible: inter-task parallelism thanks to a systolic processing of blocks of data; parallelism between the data-access and the computation thanks to the double buffering mechanism; data parallelism in a single task thanks to the pipelining or the instantiation of parallel hardware resources. The parallelism level and the size of transferred blocks of data are chosen in order to hide the latency of data transfers.

To illustrate our approach we will refer to an example of application called low-pass spatial filter (LPSF). LPSF filter is used to eliminate the high spatial frequency in the retina model [12,13]. It is composed of four inter-dependent filters as shown in **Fig. 1**. Each filter performs a low pass filtering according to a given direction. For example, HFLR computes the value of the pixel of coordinates (i,j) at instant $t=1$, by computing the pondered sum of the pixel $(i,j-1)$ at instant $t=1$ and the pixel (i,j) at a previous instant $t=0$.

In Section 2 of this paper, we describe the Array-OL formalism and the associated specification transformations. Then in Section 3, we propose an implementation architecture for data parallel applications and we define a corresponding mapping model. In Section 4, we define a method to systematically apply a set of Array-OL transformations in order to optimize the memory hierarchy and the communication sub-system for a target application. We illustrate the method with the example of the low pass spatial filter.

2 High-Level Design of Data Parallel Applications with Array-OL

Array-OL (Array-Oriented Language) is a formalism able to represent data intensive applications as a pipeline of tasks applied on multidimensional data arrays [11].

The target applications have affine array references. An Array-OL representation of the LPSF is given as in **Fig. 2**. Each filter has an inter-repetition dependency, i.e., the result of a task repetition instance is used to compute the next repetition instance. In an Array-OL representation we distinguish three

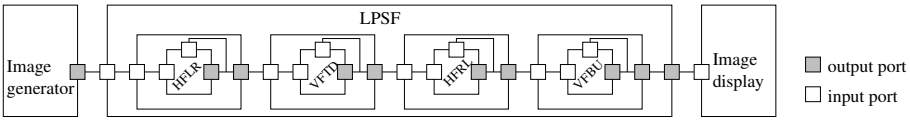


Fig. 2. Sketch of an Array-OL specification for the LPSF filter

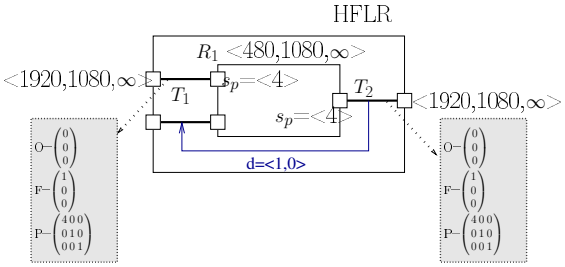


Fig. 3. Example of an Array-OL specification for a HFLR filter. The filter executes 480×1080 repetitions (R_1) per frame of 1920×1080 pixels. The task repetitions are executed on an infinite flow of frames. Temporal and spatial dimensions are all array dimensions without any distinction with each other. But usually the infinite dimension is dedicated to the time. The tilers (T_1 and T_2) describe the shape, the size and the parsing order of the patterns that partition the processed frames. The pattern s_p, that tiles the input data and is described by T_1, is monodimensional. It contains 4 pixels and lies along the horizontal-spatial dimension. The vector d is an inter-repetition dependency vector.

kinds of tasks: *elementary*, *compound* and *repetitive* tasks. An elementary task, e.g. image generator in **Fig. 2**, is an atomic black box taken from a library and it cannot be decomposed in simpler tasks. A compound task, e.g. LPSF filter in **Fig. 2**, can be decomposed in simpler interconnected task hierarchy. A repetitive task, e.g. HFLR filter in **Fig. 2**, specifies how a task is repeated on different subsets of data: data parallelism. In an Array-OL representation, no information is given on the hardware realization of tasks. **Fig. 3** shows a detailed Array-OL representation for the HFLR filter.

The information on the array tiling is given by: the origin vector O, the fitting matrix F and the paving matrix P. The origin vector specifies the coordinates of the first datum to be processed. The fitting matrix F says how to parse each data pattern and the paving matrix P says how the chosen pattern covers the data array. The size of the pattern is denoted by s_p. Array-OL is a single data assignment and a deterministic language, i.e., each datum can be written only once and any possible scheduling, which respects the data dependencies, produces the same result. For this reason, an application described in Array-OL can be statically scheduled. A crucial problem for an Array-OL description is to define the optimal granularity of the repetitions, i.e., finding an optimal data paving in order to optimize the target implementation architecture.

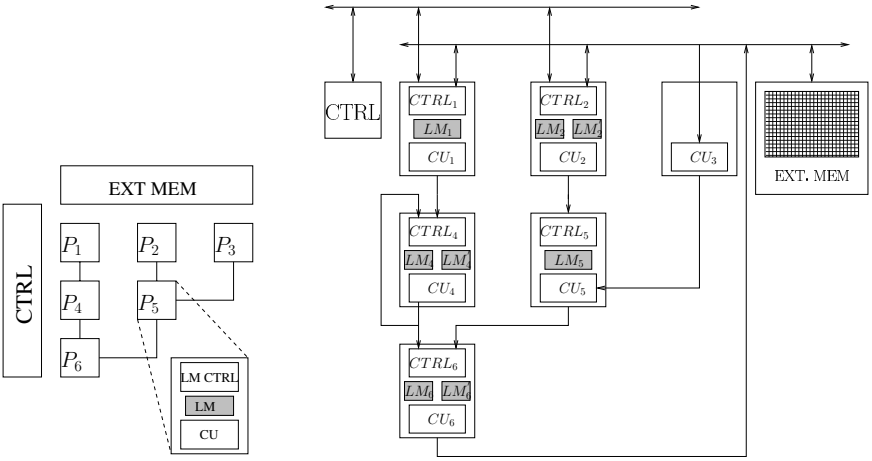
Glitia et al [14] describe several Array-OL transformations. In our work, we use the *fusion*, the *change paving* and the *tiling*. These transformations are respectively equivalent to the loop fusion [15], loop unrolling [16] and loop tiling [17,18]. One of the biggest problem for a competitive exploration is to define an optimal composition of these transformations. In fact, by composing them we can optimize the following criteria: the calculation overhead, the hierarchy complexity, the size of the intermediate array and the parallelism level. We propose to compose them in order to find a efficient architecture to mask the data access and transfer latencies.

3 A Target Customizable Architecture

3.1 Overview

We propose a synchronous architectural model (**Fig. 4**) to realize data parallel applications. This model can be customized with respect to the data transfer, the data storage and the computing process to be accomplished. All the data transfers are managed by FIFO queues so that the task execution depends on the data flow, i.e. if there are data in the queues the tasks can be executed otherwise they stall. The manipulated data can also be stored into local memories, but the access to these memories is managed by a local controller and FIFO queues. At a higher abstraction level (**Fig. 4(a)**), the architecture is a mesh of communicating processors P_k with an external memory (EXT. MEM) interface and a global controller (CTRL). The global controller starts the communication with the external memory by injecting data into the mesh of processors. It also synchronizes the external memory accesses. As shown in **Fig. 4(b)**, each single processor has essentially three customizable parts: a local memory controller ($CTRL_k$), a set of buffers (Local Memory LM_k) and a computing unit (CU_k). The behavior of such a processor has been tested at the RTL level by Corvino et al. in [19].

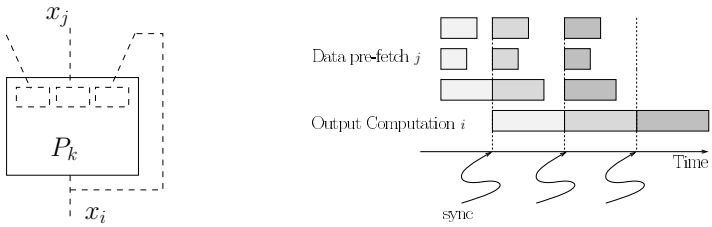
Two processors may communicate through a stand-alone FIFO queue or through a local memory. In the former case, the local memory controller and the buffers are not instantiated. In the latter case, the buffers and the controller are instantiated. Each buffer is a single port memory because the realization of multiport memories is not mature yet and it is subject to many technological problems [20]. For this reason, a processor receiving two data flows includes two distinct buffers, one for each communication flow. Furthermore, as a buffer receives a single data flow, all data transfers can be parallel without any conflict. Finally, each local buffer has a double buffering mechanism which allows to parallelize the data access and the computation. For a given application, the data to be processed are stored into the external memory. They are streamed per groups of parallel blocks into the architecture. The size of data blocks is chosen, on the one hand, in order to mask the data access latency thanks to the double buffering, and on the other hand, in order to respect the constraints on the external memory bandwidth.



(a) Abstract model.

(b) Detailed model: customizable sub-system for data transfer and storage.

Fig. 4. Target customizable architecture



(a) Sketch of a processor with its input and output data flows.

(b) Example of the timeline for sequential groups of iterations on a single processor. The processor has multiple input flows (j) and a single output flow (i).

Fig. 5. The double buffering mechanism allows to mask the time to fetch with the time to compute

3.2 The Data Transfer and Storage Model

As shown in **Fig. 5(a)**, each processor of the proposed architecture can have several input data blocks x_j but a single output data block x_i.

Thanks to the double buffering mechanism, each processor fetches the data needed to compute the next group of iterations while it is computing the current group of iterations. One of the aim of the proposed method is to chose a data block size that allows to mask the time to fetch with the time to compute. As a result, such a block size must respect the following mapping rule:

$$\max_j \{t_{fetch}(x_j)\} \leq t_{com}(x_i). \tag{1}$$

As illustrated in **Fig. 5(b)**, the execution timeline have two parallel contributions: the time to pre-fetch and the time to compute. The duration of the longest of these contributions synchronizes the beginning of a new set of computations in the pipeline. The time to pre-fetch a set of data depends on the communication type, i.e., when the processor receives data from the external memory or from another processor. We propose a model for each of these transfer types.

Data transfer with external memory. We make the following hypotheses: *The external memory is accessed in a burst mode [21], i.e. there is a latency before accessing the first datum of a burst, than a new datum of the burst can be accessed at each processor cycle. A whole data partition is contained in a memory burst and the access to a whole data partition is an atomic operation. It is possible to store m data per word of the external memory (for example we can have 4 pixels of 8 bits per each memory word of 32 bits).*

Under the above hypotheses, we propose the following definition:

Definition 1. *Given L and m denoting respectively the latency before accessing a burst and the number of data per external memory word, the time to fetch a set of data x_j from an external memory is:*

$$t_{fetch}(x_j) = L(j) + \frac{x_j}{m} \qquad \text{with } x_j, m \in \mathbf{N}^*.$$

The latency $L(j)$ has three contributions: the latency due to the other data transfers between the target architecture and the external memory, the latency due to the burst address decode and the latency due to other data transfers which are external to the target architecture (these two last contributions are indicated as L_m). In the worst case, this latency is:

$$L(j) = L_m + \sum_{z \neq j} \{L_m + \frac{x_z}{m}\} \tag{2}$$

with $x_z \in X^{Mem}$ and X^{Mem} being respectively the set of all the data transfers (in input and output) between the given architecture and the external memory. Hence, we have:

$$t_{fetch}(x_j) = N_k L_m + \frac{1}{m} Vect(1) X^{Mem}. \tag{3}$$

where N_k is the number of data blocks exchanged between the target architecture and the external memory. The expression $Vect(1)$ indicates a line vector of coordinates 1.

Transfer between two processors and the computation time. In the proposed architecture, the data computed by a processor are immediately fetched into a second processor, in a dedicated buffer. Thus, the time to fetch a set of data x_j into a processor P_k corresponds to the time to compute a set of data x_j by a processor P_l. To define the time to compute we use the simplification hypothesis proposed by Schreiber et al in [22]:

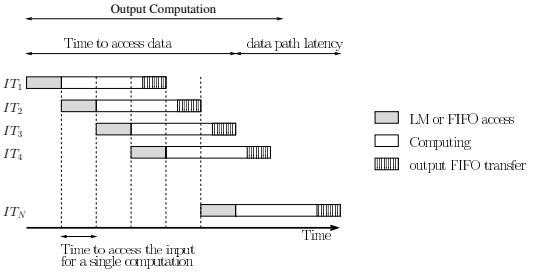


Fig. 6. Example of a pipeline of iterations (IT) during which two sub-tasks are realized: the access to a Local Memory (LM_j) and the output computing. If the number of iteration N is very high, the datapath latency can be neglected with respect to the time spent to access data.

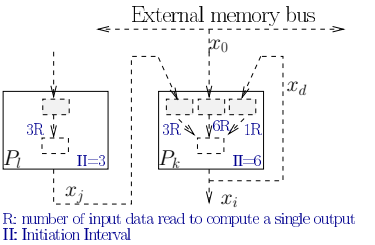


Fig. 7. Example of a processor receiving three input data blocks and producing a single output data block

The iterations of a task can be pipelined on the same hardware, but in order to avoid the conflicts on the streams or memory accesses, a minimum interval of time (called Initiation Interval - II) has to pass between the beginning of two successive iterations. The time to access data increases with the increasing of the number of iterations while the datapath latency remains constant (cf. ***Fig. 6****). If the number of pipelined iterations is sufficiently high, the datapath latency can be neglected with respect to the time spent to access data.*

This simplification leads to a transaction-based model, without precision on the execution time of the functional core. Under the given hypothesis, we define:

Definition 2. *Let c_{x_l} be the Initiation Interval of a processor P_l. The time to compute a set of data x_j is: $t_{com}(x_j) = c_{x_l} x_j$.*

Example 1. *An application of inequality (1). Let P_k be a processor, as presented in* ***Fig. 7****. It has three input flows: one from the external memory, one from a processor P_l and another due to an inter-repetition. To avoid a deadlock, all the data needed by an inter-repetition are stored in internal memories without any latency. From definitions 1 and 2, the time to fetch is $t_{fetch} = max\{L + \frac{x_0}{m}, 3x_j\}$ and the time to compute is $t_{com} = 6x_i$. Thanks to the access affinity we know that, $x_0 = 6x_i$ and $x_j = 3x_i$. By applying the inequality (1), we have that either $L + \frac{6x_i}{m} \leq 6x_i$, possible only if $m > 1$ or $9x_i > 6x_i$, which is impossible. In this last case, the usage of data parallelism can mask the time to fetch.*

Scaling the parallelism level. In the proposed architecture there are two levels of parallelism: 1) the parallelism between data access and computation 2) the parallelism due to the pipeline of iterations on the same processor. It is possible to further scale the parallelism level, by computing in parallel independent groups of data. We indicate with N_{cu_k} the number of parallel computation units of a processor P_k, which is able to compute N_{cu_k} blocks of data in parallel. To apply the data parallelism, we distinguish between the processors communicating with the external memory and the processors communicating with each other. *For*

the processors communicating with the external memory the number of parallel computation units N_{cu_k} is limited by the memory bandwidth. When a processor writes a data flow into the external memory, its number of parallel computation units N_{cu_k} is limited to m the number of data per memory word:

$$N_{cu_k} < m. \tag{4}$$

When a processor receives an input data flow from the external memory, all the data are sequentially streamed on a unique input FIFO queue. It is possible to duplicate the internal memories and the computation units, but the time to fetch the input data is multiplied by the number of parallel computation units N_{cu_k}, while the computation time will be divided by a factor N_{cu_k}:

$$N_{cu_k}\left(L + \frac{x_j}{m}\right) < \frac{c_{x_k} x_i}{N_{cu_k}}. \tag{5}$$

For the processor communicating with each other the parallelism level reduces both the time to fetch and the time to compute. The inequality (1)becomes $\frac{C_{x_l} x_j}{N_{cu_l}} \leq \frac{C_{x_k} x_i}{N_{cu_k}}$. Thanks to the affinity of the array accesses, i.e. $x_j = k_{ij} x_i$ with $k_{ij} \in \mathbf{N}^*$, we can infer a constraint on the parallelism level of two communicating processors:

$$\frac{N_{cu_k}}{C_{x_k}} \leq \frac{N_{cu_l}}{C_{x_l} k_{ij}}. \tag{6}$$

Generalization of the transfer and storage model. We generalize the mapping rule of inequality (1)to the whole processor network, by taking into account the processors interconnections. For that, we consider the following definitions presented in a progressive way.

Definition 3. *Given a network of processors, a column-vector X of possible data block sizes produced or consumed by the processors in the network is: $X = (x_0, \ldots, x_{n-1})$ where x_i is a data block produced or consumed by P_k, $\forall k \in [0, n-1]$.*

The vector X is is associated with two vectors X_{out} and X_{in}. The coordinates of X_{in} (riespectively X_{out}) are either 0 or those of X at the same position and corresponding to the input (respectively output) data blocks in the network. A coordinate x_j of X_{in} equals $k_{ij} x_i$, where $x_i \in X_{out}$ and $k_{ij} \in \mathbf{N}^*$. The data blocks of size x_j can be received either from the external or from another processor of the network.

In the following definitions, we give the relation between X_{in} and X_{out} and we distinguish the case when the input data blocks are received from the external memory.

Definition 4. *Let X_{in}^{Mem} be a vector of possible sizes for the data blocks read from the external memory. The matrices K_δ and K_δ^{Mem} giving the mapping between the sizes of input and output data blocks are:*

$X_{in} = K_\delta X_{out}$ *for all the input communications*
$X_{in}^{Mem} = K_\delta^{Mem} X_{out}$ *for the communications from the external memory.*

where X_{out} and X_{in} are vectors of possible sizes for respectively output and input data blocks.

An element δ_{ij} of K_δ (or K_δ^{Mem}) is defined as follows:

$$\delta_{ij} = \begin{cases} k_{ij} \text{ if } x_j = k_{ij} x_i \text{ and } x_j \in X_{in} (\text{ or respectively } x_j \in X_{in}^{Mem}) \\ 0 \text{ otherwise.} \end{cases}$$

Definition 5. *Let X_{out}^{Mem} and X_{out}^{Com} be two vectors of possible sizes of data blocks respectively written into the external memory and exchanged by the processors. We define two matrices I_δ^{Mem} and I_δ^{Com} so that:*

$$X_{out}^{Mem} = I_\delta^{Mem} X_{out}$$

$$X_{out}^{Com} = I_\delta^{Com} K_\delta X_{out}$$

where K_δ gives the mapping between the sizes of input and output data blocks for all the input communications. An element δ_{ij} of I_δ^{Mem} (or I_δ^{Com}) is:

$$\delta_{ij} = \begin{cases} 1 \text{ if } x_j \in X_{out}^{Mem} (\text{ or respectively } x_j \in X_{in} \setminus X_{in}^{Mem}) \\ 0 \text{ otherwise.} \end{cases}$$

Definition 6. *Given a processor P_k, a column vector C_x giving the Initiation Interval per processor, is $C_x = \{c_{x_k} : c_{x_k}$ is the Initiation Interval of a processor $P_k\}$.*

Definition 7. *Given a processor P_k, a column vector N_{cu} giving the number of parallel computation units per processor, is $N_{cu} = \{N_{cu_k} : N_{cu_k}$ is the number of $parallelcomputationunitsinP_k\}$.*

From the above definitions, we infer the mapping criteria to mask the data access latency for the input, output and internal communications of the target architecture.

Mapping Criterion 1. *Input communication from the external memory. Let be $X^{Mem} = X_{in}^{Mem} + X_{out}^{Mem} = (K_\delta^{Mem} + I_\delta^{Mem}) X_{out}$ and $Diag(N_{cu})$ a diagonal matrix whose diagonal elements are the coordinates of vector N_{cu}. It is possible to write the equation 3 as follows: $t_{fetch}(X^{Mem}) = Diag(N_{cu}) Vect(1)^T (N_k L_m + \frac{Vect(1)(K_\delta^{Mem} + I_\delta^{Mem}) X_{out}}{m})$ and **Definition 2.** as follows: $t_{com}(X_{in}^{Mem}) = \frac{Diag(C_x) K_\delta^{Mem} X_{out}}{Diag(Ncu)}$. The substitution of $t_{fetch}(X^{Mem})$ and $t_{com}(X_{in}^{Mem})$ in inequality (1), gives:*

$$\left(\frac{Diag(C_x) K_\delta^{Mem}}{Diag(N_{cu}) Diag(N_{cu})} - \frac{Vect(1)^T Vect(1)(K_\delta^{Mem} + I_\delta^{Mem})}{m} \right) X_{out} \leq Vect(1)^T N_k L_m.$$

Mapping Criterion 2. *Output communication to the external memory. Let* $N_{cu}^T I_\delta^{Mem}$ *be the number of parallel computing units of the processors communicating with the external memory. From inequality 4, we infer:*

$$Diag(N_{cu})I_\delta^{Mem} \leq m.$$

Mapping Criterion 3. *Given* X_{out}^{Com} *of definition 5. Let* I_i *and* I_j *be two squared matrices so that the left multiplication* $I_j I_\delta^{Com}$ *selects the* j^{th} *line of* I_δ^{Com}*. From inequality 6, we infer:*

$$\forall i, j \quad I_j Diag(C_x) I_\delta^{Com} K_\delta Diag(N_{cu}^{-1}) \leq I_i Diag(C_x) Diag(N_{cu}^{-1}).$$

The above mapping criteria form a system of inequalities whose variables are the parallelism level N_{cu} and the size of the transferred data blocks X_{out}. Solving the system of inequalities means to find N_{cu} and X_{out} that mask the time to access data and respect the external memory bandwidth.

4 The Design Space Exploration Approach

We propose a Design Space Exploration (DSE) which is aimed to efficiently map an application described in Array-OL onto an architecture as proposed in **Fig. 4**. The DSE chooses a task fusion that improves the execution time and uses the minimal inter-task patterns. Then it changes the data paving in order to mask the latency of the data accesses. The space of the exploration is a set of solutions with a given parallelism, a fusion configuration and data block sizes that meet the constraints on the external memory bandwidth. The results of the exploration is a set of solutions which are optimal (also termed *pareto*) with respect to two optimization criteria: the architecture latency and the internal memory amount.

The optimization criteria. A processor P_i contains a buffer of size LM_j per each input flow:

$$LM_j(P_i) = 2N_{cu_i} x_j.$$

The factor 2 is due to the double buffering. The total amount of used internal memory is:

$$IM = \sum_i \sum_j LM_j(P_i).$$

We define the architecture latency as the latency to compute a single output image. As in our model, the times to access data are always masked, we can approximate the architecture latency with the time necessary to execute all the output transfers towards the external memory:

$$AL = Image_{size} \triangle (I_\delta^{Mem}).$$

where $\triangle$ denotes the determinant. $Image_{size}$ can be inferred from the Array-OL specification.

The Array-OL model on the target architecture. An Array-OL model can directly be mapped onto an architecture like that presented in **Fig. 4**, by using the following rules:

1. *The analysis starts from a canonical Array-OL specification, which is defined to be equivalent to a perfectly loop-nested code [23]: it cannot contain repetition around the composition of repetitive task.*
2. *The hierarchy of the Array-OL model is analyzed from the highest to the lowest level.*
3. *At the highest level, we infer the the parameters of the mapping, i.e.* C_x, I_δ^{Mem}, I_δ^{com}, K_δ *and* K_δ^{Mem}. *Given a task* $task_i$, *let* ${s_p}^{in}(task_i)$ *and* ${s_p}^{in}(j)$ *be respectively the size of the output pattern and input patterns (j). The elements of* K_δ *and* K_δ^{Mem} *are computed as:*

$$\delta_{i,j} = \frac{\triangle(Diag({s_p}^{in}(j))}{\triangle(Diag({s_p}^{out}(task_i))}.$$

   An element c_{x_i} *of* C_x *is* $c_{x_i} = \max_j\{\delta_{i,j}\}$. *The values of* I_δ^{Mem} *and* I_δ^{com} *elements depend on the inter-task links.*
4. *At each level we distinguish among an elementary, a composite or a repetitive task.*
   - *If the task is elementary or a composition of elementary tasks, a set of library elements is instantiated to realize it.*
   - *When a task is a repetition we distinguish two cases: if it is a repetition of a single task we instantiate a single processor; if it is a repetition of a compound task we instantiate a set of processors in a SIMD, MIMD or pipelined configuration. The repetitions of the same task is iterated on the same hardware or executed in parallel according to mapping criteria.*
5. *Each instantiated processor contains at least a datapath (of library elements) and may contain some local buffers and a local memory controller.*

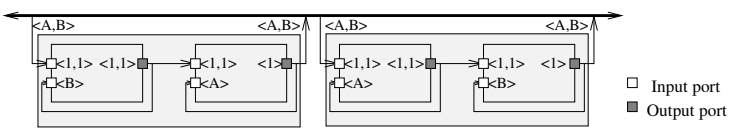


Fig. 8. Example of a possible canonical Array-OL specification for a LPSF filter

Example 2. *Mapping constraints for a LPSF filter in an Array-OL specification.* ***Fig. 8*** *shows a possible canonical specification for a LPSF filter.*

The specification contains 4 elementary tasks, thus we instantiate 4 processors (P_1, P_2, P_3, P_4). *The user's specified parameters are:* $m = 4$ *and* $L_m = 30$. *From the Array-OL specification analysis, we infer the following parameters of the mapping criteria:*

$C_x = (1, 1, 1, 1)$;

$$I_\delta^{Mem} = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&1 \end{pmatrix}; I_\delta^{com} = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&0 \end{pmatrix}; K_\delta = \begin{pmatrix} 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \\ 0&0&0&0&0&0 \end{pmatrix}; K_\delta^{Mem} = \begin{pmatrix} 0&1&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix}$$

A 200 lines Python code processes these parameters and obtains the following results by solving the inequality system of the mapping criteria: $N_{cu} \leq (4,4,4,4)$*;* $X_{out} \geq (480,480,480,480)$*. These constraints are used to chnage the data paving in the Array-OL specification. The Python program also computes the optimization criteria to enable the successive design space exploration:* $IM = 15K(data)$ $AL = 4M(cycles)$*.*

The DSE flow. By starting from a canonical form, all the possible task fusions are explored (according to a method explained in the next paragraph). The obtained Array-OL model is mapped onto an architecture as presented in **Fig. 4** and the repetition granularity of the merged tasks is changed in order to mask the external memory latency. The granularity is changed through a change paving and in order to have $\triangle(Diag(s_p^{out}(task_i))) \geq x_i$. Finally, the obtained solutions are evaluated against the amount of internal memory used and the architecture latency. The *pareto* solutions are chosen.

Reducing the space of possible fusions. To reduce the exploration space of the possible fusions, we propose to adapt the method used by Talal at al. ([24]) to generate optimal coalition structures. Given a number of tasks n, we can map the space of possible fusions onto an integer partition of n. An integer partition is a set of positive integer vectors whose components add up to n. For example, the integer partition of $n = 4$ is $[1,1,1,1], [2,1,1], [2,2], [3,1], [4]$. Each vector of the integer partition can be a mapping for more possible fusions as shown in **Fig. 9**. Let a macrotask be the result of a fusion, as proposed by Talal *et al.*, we reduce the number of sub-spaces by merging the sub-spaces whose solutions contain the same number of macrotasks. For the example of **Fig. 9**, the sub-spaces mapped on the integer partitions [3,1] and [2,2] are merged. In this way the number of sub-spaces is limited to n. This mapping reduces the number of comparisons between the possible solutions. In fact we search for the *pareto* solutions of each sub-space and we compare them to each other in order to find the *pareto* solutions of the whole space. For the example of **Fig. 9**, we perform 32 comparisons instead of 56. The *pareto* solutions are denoted by $Psol_i$ in **Fig. 9**. Among these solutions a user can choose the most adapted to his or her objectives. In our case, we have chosen the solution $Psol_1$, which has the most advantageous tradeoff between the used internal memory and the architecture latency. The mapping constraints for this solution are given in **Example** 2. Blocks of data parsing the input image from the top left to the bottom right corner are pipelined on the HFLR and VFTD filters. It is possible to process up to 4 parallel blocks of data per filter. Each block has to contain at least 60

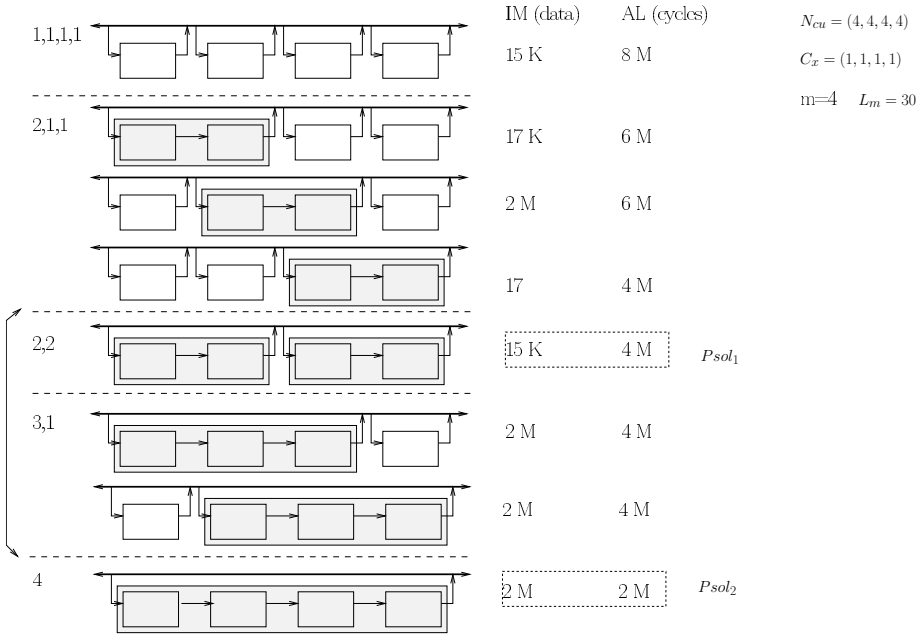


Fig. 9. Example of an exploration for a LPSF. Solutions merging VFDT and HFRL have to store a whole image, thus they use 2M of internal memory

data to mask the time to fetch. The execution of HFRL and VFBU filters is similar except that it starts from the bottom right and stops to the top left corner.

5 Conclusion

We presented a method to explore the space of possible data transfer and storage micro-architectures for data parallel application described in Array-OL. This method starts from a canonical Array-OL representation and apply a set of transformations in order to infer an Application Specific architecture that masks the times to transfer data with the time to perform the computations. We propose a customizable model of the target architecture including FIFO queues and double buffering mechanism. The mapping of a given image processing application onto the proposed architecture is performed through a flow of Array-OL transformations aimed to improve the parallelism level and to reduce the size of the used internal memories. We also use a method based on an integer partition to reduce the space of explored transformations. This method is aimed to be integrated into Gaspard, an Array-OL framework able to map Array-OL models onto different kind of target architectures [25]. An industry-size case study is currently in progress.

References

1. Catthoor, F., et al.: Data Access and Storage Management for Embedded Programmable Processors. Kluwer Academic Publishers, Dordrecht (2002)
2. Balasa, F., Kjeldsberg, P., Vandecappelle, A., Palkovic, M., Hu, Q., Zhu, H., Catthoor, F.: Storage Estimation and Design Space Exploration Methodologies for the Memory Management of Signal Processing Applications. Journal of Signal Processing Systems 53(1), 51–71 (2008)
3. Hiser, J.D., Davidson, J.W., Whalley, D.B.: Fast, Accurate Design Space Exploration of Embedded Systems Memory Configurations. In: SAC 2007: Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 699–706. ACM, New York (2007)
4. Hu, Q., Kjeldsberg, P.G., Vandecappelle, A., Palkovic, M., Catthoor, F.: Incremental hierarchical memory size estimation for steering of loop transformations. ACM Transactions on Design Automation of Electronic Systems 12(4), 50 (2007)
5. Chen, Y., Byna, S., Sun, X.-H., Thakur, R., Gropp, W.: Hiding I/O latency with pre-execution prefetching for parallel applications. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–10 (2008)
6. Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G.: Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation of Electronic Systems 6(2), 149–206 (2001)
7. Kung, H.T.: Why systolic architectures. Computer 15(1), 37–46 (1982)
8. Amar, A., Boulet, P., Dumont, P.: Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model. In: ISPAN 2005: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks, pp. 496–503 (2005)
9. Kim, D., Managuli, R., Kim, Y.: Data cache and direct memory access in programming mediaprocessors. IEEE Micro 21(4), 33–42 (2001)
10. Ascia, G., Catania, V., Di Nuovo, A.G., Palesi, M., Patti, D.: Efficient design space exploration for application specific systems-on-a-chip. Journal of Systems Architecture 53(10), 733–750 (2007)
11. Glitia, C., Dumont, P., Boulet, P.: Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. In: Multidimensional Systems and Signal Processing. Springer, Netherlands (2010)
12. de Lavarene, B.C., Alleysson, D., Durette, B., Herault, J.: Efficient demosaicing through recursive filtering. In: IEEE International Conference on Image Processing (ICIP 2007), vol. 2 (October 2007)
13. Hérault, J., Durette, B.: Modeling visual perception for image processing. In: Sandoval, F., Prieto, A.G., Cabestany, J., Graña, M. (eds.) IWANN 2007. LNCS, vol. 4507, pp. 662–675. Springer, Heidelberg (2007)
14. Glitia, C., Boulet, P.: High level loop transformations for systematic signal processing embedded applications. In: Bereković, M., Dimopoulos, N., Wong, S. (eds.) SAMOS 2008. LNCS, vol. 5114, pp. 187–196. Springer, Heidelberg (2008)
15. Maximizing loop parallelism and improving data locality via loop fusion and distribution, pp. 301–320. Springer, Heidelberg (2006)
16. Hannig, F., Dutta, H., Teich, J.: Parallelization approaches for hardware accelerators – loop unrolling versus loop partitioning. In: Architecture of Computing Systems – ARCS 2009, pp. 16–27 (2009)
17. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers, Dordrecht (2000)

18. Panda, P.R., Nakamura, H., Dutt, N.D., Nicolau, A.: Augmenting loop tiling with data alignment for improved cache performance. IEEE Transactions on Computers 48, 142–149 (1999)
19. Rosilde, C.: Design Space Exploration for data-dominated image applications with non-affine array references. PhD thesis (2009)
20. Liu, L., Nagaraj, P., Upadhyaya, S., Sridhar, R.: Defect analysis and defect tolerant design of multi-port srams. J. Electron. Test. 24(1-3), 165–179 (2008)
21. Imondi, G.C., Zenzo, M., Fazio, M.A.: Pipelined Burst Memory Access, US patent (August 2008)
22. Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, B., Cronquist, D., Sivaraman, M.: Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. The Journal of VLSI Signal Processing 31(2), 127–142 (2002)
23. Ahmed, N., Mateev, N., Pingali, K.: Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. International Journal of Parallel Programming 29(5), 493–544 (2001)
24. Rahwan, T., Ramchurn, S., Jennings, N., Giovannucci, A.: An anytime algorithm for optimal coalition structure generation. Journal of Artificial Intelligence Research (JAIR) 34, 521–567 (2009)
25. Gamatié, A., Le Beux, S., Piel, É., Atitallah, R.B., Etien, A., Marquet, P., Dekeyser, J.-L.: A model driven design framework for massively parallel embedded systems. In: ACM Transactions on Embedded Computing Systems (TECS) ©. ACM, New York (to appear 2010), `http://hal.inria.fr/inria-00311115/`