Sparse computations with PEI

Frédérique Voisin, Guy-René Perrin

ICPS, Université Louis Pasteur, Strasbourg
Pôle API, Boulevard Sébastien Brant
F-67400 Illkirch

1 Introduction

A wide range of research work on the static analysis of programs forms the foundation of parallelization techniques which improve the efficiency of codes: loop nest rewriting, directives to the compiler to distribute data or operations, etc. These techniques are based on geometric transformations either of the iteration space or of the index domains of arrays.

This geometric approach entails an abstract manipulation of array indices, to define and transform the data dependences in the program. This requires to be able to express, compute and modify the placement of the data and operations in an abstract discrete reference domain. Then, the programming activity may refer to a very small set of primitive issues to construct, transform or compile programs. PEI is a program notation and includes such issues.

In this paper we focus on the way PEI deals with sparse computations. For any matrix operation defined by a program, a sparse computation is a transformed code which adapts computations efficiently in order to respect an optimal storage of sparse matrices [2]. Such a storage differs from the natural memory access since the location of any non-zero element of the matrix has to be determined. Due to the reference geometrical domain in PEI we show how a dense program can be transformed to meet some optimal memory storage.

2 Definition of the formalism PEI

2.1 An introduction to PEI Programming

The theory PEI was defined [7, 8] in order to provide a mathematical notation to describe and reason on programs and their implementation. A program can be specified as a relation between some *multisets* of value items, roughly speaking its *inputs* and *outputs*. Of course, programming may imply to put these items in a convenient organized directory, depending on the problem terms. In scientific computations for example, items such as arrays are functions on indices: the index set, that is the reference domain, is a part of some Z^n. In PEI such a multiset of value items mapped on a discrete reference domain is called a *data field*.

For example the multiset of integral items $\{1, -2, 3, 1\}$ can be expressed as a data field, say `A`, each element of which is recognized by an index in Z (e.g. from 0 to 3). Of course this multiset may be expressed as an other data field, say `M`, which places the items on points $(i, j) \in Z^2$ such that $0 \leq i, j < 2$. These two data fields `A` and `M` are considered equivalent in PEI since they express the same multiset. More formally, there exists a bijection from the first arrangement onto the second one, e.g. $\sigma(i) = (i \ mod\ 2, i \ div\ 2)$. We note this by the equation:

$$\texttt{M} = \texttt{align} :: \texttt{A}$$

where $\qquad \texttt{align} = \lambda i \mid (0 \leq i \leq 3) \,.\, (i \ mod\ 2, i \ div\ 2)$

Any PEI program is composed of such unoriented equations. Here is an example of PEI program:

Example 1. For any i in $[1..n]$ compute $x_i = \sum_{j=1}^{j=n} a_{i,j} \times v_j$.

Any x_i can be defined by a recurrence on the domain $\{j \mid 1 \leq j \leq n\}$ of Z by:

$$\begin{cases} s_{i,1} = a_{i,1} \times v_1 & & (1) \\ s_{i,j} = s_{i,j-1} + (a_{i,j} \times v_j) & 1 < j \leq n & (2) \\ x_i \ = s_{i,n} & & (3) \end{cases}$$

where the $s_{i,j}$ are intermediate results. The recurrence equation (2) emphasizes uniform dependencies $(0,1)$ for the calculation of $s_{i,j}$.

In PEI, this definition can be expressed in the following way. Matrix A is expressed as a data field `A` mapped on the domain $\{(i,j) \mid 1 \leq i,j \leq n\}$. Vectors v and x are expressed as data fields `V` and `X` mapped on $\{i \mid 1 \leq i \leq n\}$: this means that for any i, the values v_i and x_i are located at point i. Of course, any program which lies on another mapping for A, V or X defines an equivalent program, provided its operations result in an equivalent data field.

```
MatVec: (A,V) ↦ X
{
matrix :: A = A
align :: B = V
P = prod ▷ (A /&/(B ◁ spread))
S = add ▷ (P /&/(S ◁ shift))
X = project :: (S ◁ last)
}
matrix  = λ(i,j) |(1≤i≤n, 1≤j≤n) . (i,j)
align   = λ(i,j) |(i=1, 1≤j≤n) . j
project = λ(i,j) |(1≤i≤n, j=n) . i
shift   = λ(i,j) |(1≤i≤n, 1<j≤n) . (i,j−1)
last    = λ(i,j) |(1≤i≤n, j=n) . (i,j)
spread  = λ(i,j) |(1≤i≤n, 1≤j≤n) . (1,j)
add     = id # λ(a·b) . a+b
prod    = λ(a·b) . a×b
```

The recurrence defining the variable s suggests to map the data field `S` in Z^2. Since the v_j are used by the $s_{i,j}$, for any i, the values of `V` are mapped in Z^2 too, by a kind of data *Strip-Mining* on data field `B`. Its values are then broadcasted to *localize* the right values onto the right locations in order to compute the *recurrence* steps:

- the first equation defines the mapping of the input data field `A`. Function `matrix`, defined on Z^2, applies a change of basis (notation `::`) which is the identity on the domain $[1..n] \times [1..n]$. It means that the values are placed onto a square $[1..n] \times [1..n]$,
- the second equation implicitely defines `B`. The function `align`, defined on Z^2, applies a change of basis which expresses that the projection of its argument `B`, supposed to be a row, on a one-dimensional domain, is equal to `V`. So, `V` is *aligned* on a two-dimensional domain. Notice that functions, such as `align`, define the context of the program: they are expressed as λ-expressions, in which the separator "|" allows to define the domain of the function, and the "." begins the function body,
- in the third equation, the operation `/&/` builds the pairs of value items of `A` and (`B` $\triangleleft$ `spread`) which are placed at the same location: the last expression applies a *geometrical operation* (notation $\triangleleft$) which broadcasts the values of the "first row" of data field `B` in the direction $(1,0)$ in Z^2,
- the data field `S` results from the application of a so-called *functional operation* (notation $\triangleright$) on two data fields. The first one results of a geometrical operation on `S` which expresses the dependency $(0,1)$ in Z^2,
- the last equation defines the data field `X`, by a change of basis: it projects the part of `S` mapped on $\{(i,n) \mid 1 \leq i \leq n\}$ onto Z. So, the result `X` is one-dimensional.

2.2 Syntactic issues

The previous example allows to understand that data fields are main features in PEI. Besides data fields, partial functions define operations on these objects. In the following `A`, `B`, `X`, etc. denote data fields, whereas `f`, `g`, etc. denote functions. The PEI notation for partial functions is derived from the lambda-calculus: any function `f` of domain $dom(\mathtt{f}) = \{x \mid P(x)\}$ is denoted as $\lambda x \mid P(x) .\mathtt{f}(x)$. Moreover a function `f` defined on disjunctive sub-domains is denoted as a partition $\mathtt{f}_1 \# \mathtt{f}_2$ of functions.

Expressions are defined by applying operations on data fields. PEI defines one internal operation on the data field set, called *superimposition*: it is denoted as `/&/` and builds the sequences of values of its arguments[1]. Three external operations are defined and associate a data field with a function:

- the *functional operation*, denoted as $\triangleright$, applies a function `f` (which may be a partial function) on value items of a data field `X` (notation $\mathtt{f} \triangleright \mathtt{X}$),
- the *change of basis* (denoted as `::`). Let `h` be a bijection, `h::X` defines a data field that maps the value items of `X` onto an other discrete reference domain,
- *the geometrical operation*, denoted as $\triangleleft$, moves the value items of a data field on its domain: the data field $\mathtt{X} \triangleleft \mathtt{g}$ is such that the item mapped on some index z, $z \in dom(\mathtt{g})$, "comes from" `X` at index $\mathtt{g}(z)$.

2.3 Semantics and Program transformations

The semantics of PEI is founded on the notion of discrete domain associated with a multiset. We call *drawing* of a multiset M of values in V, a partial function v from some Z^n in V whose image is M.

Such a drawing should define a natural functional interpretation $[\![\mathtt{X}]\!] = v$ of a data field `X`. As we have told it at the beginning of this section, any other drawing can be deduced by applying any bijection, say h. As soon as its domain $dom(h)$ contains $dom(v)$, its interpretation should be also equal to $v \circ h^{-1}$. This is not sound and we have to consider a data field `X` as the abstraction of any drawing of a given multiset indeed. Here is the definition: a *data field* `X` is a pair $(v : \sigma)$, composed of a drawing v of a multiset M and of a bijection σ such that $dom(v) \subset dom(\sigma)$.

This definition induces a sound interpretation of a data field $\mathtt{X} = (v : \sigma)$ as the function $[\![\mathtt{X}]\!] = v \circ \sigma^{-1}$. It founds the semantics of operations on data fields.

The semantics allows to define transformations of programs in PEI, as soon as a data field is substituted for any equivalent one in an expression, or by applying some *algebraic law* on the operations. Here is a few examples of such laws (a detailed list is given in [9]). Assuming right and left expressions are valid expressions:

$$
\begin{array}{ll}
\mathtt{f1} \triangleright (\mathtt{f2} \triangleright \mathtt{X}) & = (\mathtt{f1} \circ \mathtt{f2}) \triangleright \mathtt{X} \\
\mathtt{X} \triangleleft (\mathtt{g1} \circ \mathtt{g2}) & = (\mathtt{X} \triangleleft \mathtt{g1}) \triangleleft \mathtt{g2} \\
(\mathtt{X1/\&/X2}) \triangleleft \mathtt{g} & = (\mathtt{X1} \triangleleft \mathtt{g})\mathtt{/\&/}(\mathtt{X2} \triangleleft \mathtt{g}) \\
\mathtt{h::(X1/\&/X2)} & = \mathtt{(h::X1)/\&/(h::X2)} \\
\mathtt{h::}(\mathtt{f} \triangleright \mathtt{X}) & = \mathtt{f} \triangleright (\mathtt{h::X}) \\
\mathtt{h::}(\mathtt{X} \triangleleft \mathtt{g}) & = (\mathtt{h::X}) \triangleleft \mathtt{h} \circ \mathtt{g} \circ \mathtt{h}^{-1}
\end{array}
$$

3 PEI and sparse computations

PEI can deal with sparse computations by adapting a code in order to respect an optimal storage of sparse data structures. Such a storage differs from the natural memory access since the location of any non-zero element of the matrix has to be determined. In terms of PEI features this means that the optimal storage and the dense array are two equivalent data fields: the change of basis from one data field onto the other one defines the way the sparse matrix is stored. So, transform the code consists in applying this change of basis in the dense program.

Let us consider again the example of a matrix-vector product, assuming the matrix is composed of a few bands parallel to the diagonal (see fig. 1 for example, where non-zero elements are located at indices such that either $i = j + 2$, or $i = j$, or $i = j - 2$).

One of the most popular storage (used in LAPACK [1] for example) for sparse matrices is Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC). This storage requires three one-dimension arrays: A stores the non-zero elements in the same order as they appear in the matrix, by row for CSR (by column for CSC), JA (resp. IA) stores the original column (resp. row) of an element, and IA (resp. JA) points to the beginning of each line in A and JA (resp. IA). A modified version of this storage is MSR (Modified Sparse Row): the main diagonal is stored at the beginning of the array A, the rest of the matrix is stored using CSR (or CSC, for MSC), except that IA is replaced by the first positions of JA (those corresponding to the diagonal elements in A, as shown in fig. 1).

---

[1] We use two operators on sequences: an associative constructor denoted as "," and the function "`id`" which is the identity on sequences of one element.

3.1 Program transformation in Pei

The non-zero value items of the matrix and the vector value items must belong to the domain of the change of basis to be applied in Pei. Therefore, a first transformation step consists for example in aligning the vector with the main diagonal: MSR storage is then well-adapted. This alignment can be defined by rewriting `spread` as:

$$\texttt{spread} = \texttt{pivot} \circ \texttt{diagonal}$$

where

$$\begin{aligned} \texttt{pivot} &= \lambda(i,j)\ |\, (1{\le}i{\le}n, j{=}i)\,.\,(1,j) \\ \texttt{diagonal} &= \lambda(i,j)\ |\, (1{\le}i{\le}n, 1{\le}j{\le}n)\,.\,(j,j) \end{aligned}$$

we obtain the new program in Pei:

```
{
matrix :: A = A
align :: B = V
P = prod ▷ ( A /&/( (B ◁ pivot) ◁ diagonal ) )
...
}
```

The change of basis itself (figure 1) is defined by the following function:

$$\begin{aligned} \texttt{gather} = \ & \lambda(i,j)\ |\, (1{\le}i{\le}2, 3{\le}j{\le}4, j{=}i{+}2)\,.\,(n+i+1) \\ \# \ & \lambda(i,j)\ |\, (3{\le}i{\le}n-2, 5{\le}j{\le}n, j{=}i{+}2)\,.\,(n+2i-1) \\ \# \ & \lambda(i,j)\ |\, (1{\le}i{\le}n, 1{\le}j{\le}n, i{=}j)\,.\,(i) \\ \# \ & \lambda(i,j)\ |\, (3{\le}i{\le}n-2, 1{\le}j{\le}n-4, j{=}i-2)\,.\,(n+2i-2) \\ \# \ & \lambda(i,j)\ |\, (n-1{\le}i{\le}n, n-3{\le}j{\le}n-2, j{=}i-2)\,. \end{aligned}$$

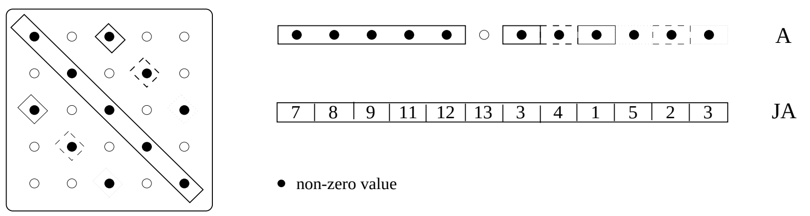

Fig. 1. Sparse matrix in full representation and MSR storage

Its inverse is :

$$\begin{aligned} \texttt{gather}^{-1} = \ & \lambda(i)\ |\, (1{\le}i{\le}n)\,.\,(i,i)) \\ \# \ & \lambda(i)\ |\, (n+2{\le}i{\le}n+3)\,.\,(i-n-1, i-n+1) \\ \# \ & \lambda(i)\ |\, (n+4{\le}i{\le}3n-5), (i-n) \bmod 2 = 0)\,.\,((i-n-1)/2+2), (i-n-1)/2) \\ \# \ & \lambda(i)\ |\, (n+4{\le}i{\le}3n-5), (i-n) \bmod 2 = 1)\,.\,((i-n-1)/2+1), (i-n-1)/2+3) \\ \# \ & \lambda(i)\ |\, (3n-4{\le}i{\le}3n-4)\,.\,(i-2n+3, i-2n+1) \end{aligned}$$

Applying the change of basis to `P` leads to this definition:

$$\begin{aligned} \texttt{gather::P} = \ & \texttt{prod} \rhd (\texttt{gather::A /\&/ gather::}((\texttt{B} \lhd \texttt{pivot}) \lhd \texttt{diagonal})) \\ & \texttt{prod} \rhd (\texttt{gather::A /\&/}((\texttt{gather::}(\texttt{B} \lhd \texttt{pivot})) \lhd \texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1})) \end{aligned}$$

Let

$$\begin{aligned} \texttt{P}' &= \texttt{gather::P} \\ \texttt{A}' &= \texttt{gather::A} \\ \texttt{T}' &= \texttt{gather::}(\texttt{B} \lhd \texttt{pivot}) \end{aligned}$$

We can write:

$$\texttt{P'} = \texttt{prod} \ \triangleright \ (\ \texttt{A'}\ \texttt{/\&/}\ (\texttt{T'} \ \triangleleft \ \texttt{spread})\ )$$

where

$$\begin{aligned}
\texttt{spread} = \texttt{gather} \circ \texttt{diagonal} \circ \texttt{gather}^{-1} & \\
= \ & \lambda(i)\ |(1 \le i \le n)\,.\,(i) \\
\# \ & \lambda(i)\ |(n+2 \le i \le n+3)\,.\,(i-n+1) \\
\# \ & \lambda(i)\ |(n+4 \le i \le 3n-5, (i-n) \bmod 2 = 0)\,.\,((i-n-1)/2) \\
\# \ & \lambda(i)\ |(n+4 \le i \le 3n-5, i-n) \bmod 2 = 1)\,.\,((i-n-1)/2+3) \\
\# \ & \lambda(i)\ |(3n-5 \le i \le 3n-4)\,.\,(i-2n+1)
\end{aligned}$$

In order to apply the change of basis `gather` in the equations, the definition of `S` is constrained on the domain of `gather` by geometrical operation `sparse` and folding:

$$\begin{aligned}
\texttt{sparse} = \ & \lambda(i,j)\ |(3 \le i \le n, 1 \le j \le n, i{=}j{+}2) \\
\# \ & \lambda(i,j)\ |(1 \le i \le n, 1 \le j \le n, i{=}j) \\
\# \ & \lambda(i,j)\ |(1 \le i \le n{-}2, 1 \le j \le n, i{=}j{-}2)
\end{aligned}$$

$$\begin{aligned}
\texttt{S} \triangleleft \texttt{sparse} &= \texttt{add} \triangleright (\texttt{P} \triangleleft \texttt{sparse}\ \texttt{/\&/}\ (\texttt{S} \triangleleft \texttt{shift} \circ \texttt{shift}) \triangleleft \texttt{sparse}) \\
&= \texttt{add} \triangleright (\texttt{P} \triangleleft \texttt{sparse}\ \texttt{/\&/}\ (\texttt{S} \triangleleft \texttt{sparse}) \triangleleft \texttt{sparse_shift})
\end{aligned}$$

where

$$\begin{aligned}
\texttt{sparse_shift} = \texttt{shift} \circ \texttt{shift} \circ \texttt{sparse} & \\
= \ & \lambda(i,j)\ |(3 \le i \le n, 3 \le j \le n, i{=}j)\,.\,(i, j{-}2) \\
\# \ & \lambda(i,j)\ |(1 \le i \le n{-}2, 3 \le j \le n, i{=}j{-}2)\,.\,(i, j{-}2)
\end{aligned}$$

Applying the change of basis `gather` to the summation results in the geometrical operation:

$$\begin{aligned}
\texttt{shift} = \texttt{gather} \circ \texttt{sparse_shift} \circ \texttt{gather}^{-1} & \\
= \ & \lambda(i)\ |(n+2 \le i \le n+3)\,.\,(i-n-1) \\
\# \ & \lambda(i)\ |(n+4 \le i \le 3n-5, (i-n) \bmod 2 = 1)\,.\,((i-n-1)/2+1) \\
\# \ & \lambda(i)\ |(3 \le i \le n-2)\,.\,(n+2i-2) \\
\# \ & \lambda(i)\ |(n-1 \le i \le n)\,.\,(2n+i-3)
\end{aligned}$$

and

$$\begin{aligned}
\texttt{last} = \texttt{gather} \circ \texttt{sparse_last} \circ \texttt{gather}^{-1} & \\
= \ & \lambda(i)\ |(n+1 \le i \le n+2) \\
\# \ & \lambda(i)\ |(n+3 \le i \le 3n-5, (i-n) \bmod 2 = 1) \\
\# \ & \lambda(i)\ |(n-1 \le i \le n)
\end{aligned}$$

where

$$\begin{aligned}
\texttt{sparse_last} = \ & \lambda(i,j)\ |(1 \le i \le n{-}2, j{=}i{+}2) \\
\# \ & \lambda(i,j)\ |(n{-}1 \le i \le n, j{=}i)
\end{aligned}$$

This completes the program transformations in PEI, and defines a sparse solution. It can be implemented in terms of classical arrays in MSR storage.

3.2 Concrete implementations

PEI expression leads easily the definition of the arrays in MSR storage. For example, let $\texttt{second} = \lambda(i,j).(j)$, we clearly have :

$$JA(i) = \texttt{second} \circ \texttt{gather}^{-1}(i) \ \text{ for } \ i > n+1$$

Note that PEI does not require to implement the first $n+1$ elements of JA because references to rows are unusefull since the vector is explicitely aligned with the computation points by the change of basis.

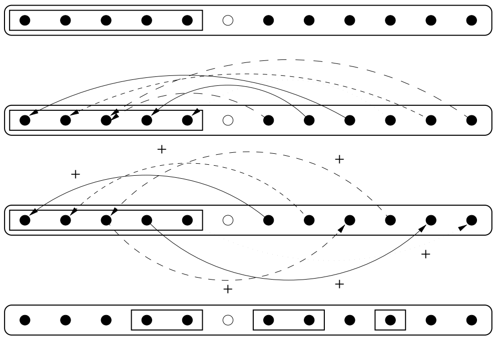

Fig. 2. Transformed functions spread, shift and last.

4 Conclusion

The presented technique shows the interest of using PEI for program transformation to address sparse computations. It can be used in order to build a sparse computation library in the data parallel programming model, for example expressed in HPF. The definition of the library requires the definition of a PEI to HPF translator. Such a software tool is under development [4].

In this paper, we have presented an example of CSR storage. Of course, other sparse matrix strorages (Sparse General Pattern [5], Compress Diagonal Storage or other storages described in [6]) can be considered by applying convenient change of basis in PEI. The example considered here is also based on a tridiagonal matrix which is a special case of sparse matrix. Unfortunately sparse matrices do not always have this special shape and the main difficulty is then to define the functions. It can be achieved by reading the matrices for example from a Harwell-Boeing formatted file [3].

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.
2. A. Bik and H. Wijshoff. Advanced compiler optimizations for sparse computations. *J. of Parallel and Distributed Computing*, 31:14–24, 1995.
3. I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for Harwell-Boeing sparse matrix test problems collection. Tech. Report RAL-92-086, Computing and Information Systems Department, Rutherford Appleton Laboratory, Didcot, UK, 1992.
4. Stéphane Genaud. On deriving HPF code from PEI programs. Technical Report RR97-04, ICPS, September 1997.
5. Serge Petiton and Nahid Emad. *The data parallel programming model: foundations, HPF realization, and scientific applications*, chapter A data parallel scientific computing introduction, pages 45–64. Springer-Verlag Inc., 1996.
6. Yousef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.
7. E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.
8. E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, 1993.
9. E. Violard and G.-R. Perrin. Reduction in PEI. *CONPAR'94, LNCS 854*, pages 112–123, 1994.