DSL* stream programming on multicore architectures

Pablo de Oliveira Castro[1], Stéphane Louise[1], and Denis Barthou[2]

[1] CEA, LIST
[2] University of Bordeaux - Labri / INRIA

1 Introduction

The advent of multicore processors raises new programmability challenges. Complex applications are hard to write using threads, since they do not guarantee a deterministic execution, and are difficult to optimize because the programmer must carefully tune the application by hand.

Stream languages are a powerful alternative to program multicore processors for two main reasons: (i) they offer a deterministic execution based on a sound mathematical formalism (Synchronous Data Flow [22]), (ii) the expression of the parallelism is implicitly described by the stream structure, which leverages compiler optimizations that can harness the multicore performance without having to tune the application by hand.

The stream programming model emphasizes the exchange of data between filters. To properly express and optimize stream programs it is crucial to capture the data access patterns in the stream model. We can distinguish two families of stream programming languages:

- languages in which the data access patterns are explicitly described by the programmer through a set of reorganization primitives
- languages in which the data access patterns are implicitly declared through a set of dependencies between tasks.

We present in the following a brief overview of related works concerning these language families and then expose the principle of a two-level approach combining the advantages and expressivity of both types of languages.

1.1 Explicit manipulations of streams

StreamIt StreamIt[4] is both a streaming language and a compiler for RAW and SMP architectures. StreamIt revolves around the notion of filters. A filter takes a stream of input elements, performs a computation and produces the result of the computation on an output stream, thus capturing the producer-consumer pattern often used in signal applications.

* Domain Specific Language

```
float->float pipeline MatrixMultiply (int x0, int y0, int x1, int y1) {
  add splitjoin {
    split roundrobin(x0 * y0, x1 * y1);
    add DuplicateRows(x1, x0);
    add pipeline {
      add Transpose(x1, y1);
      add DuplicateRows(y0, x1*y1);
    }
    join roundrobin;
  }
  add MultiplyAccParallel(x0, x0);
}
float->float splitjoin Transpose(int x, int y) {
 split roundrobin;
 for (int i = 0; i < x; i++) add Identity<float>();
 join roundrobin(y);
}
float->float splitjoin DuplicateRows(int t, int l) {
  split duplicate;
  for (int i = 0; i < t; i++) add Identity<float>();
  join roundrobin(l);
}
float->float splitjoin MultiplyAccParallel(int x, int n) {
  // Omitted ... realises the dot product of
  // the rows of A and the columns of B in parallel.
}
```

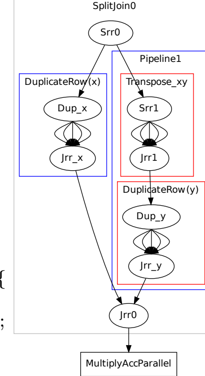


Fig. 1. Excerpt of a StreamIt program for matrix multiplication.

Filters are assembled in a flow graph by using a set of connectors: pipes form chains of consumers and producers, split-joins allow to dispatch the elements inside a stream to a group of filters (parallelizing the computation) and reassemble the results, feedback loops allow to introduce cycles in the flow graph. Using these connectors constraints the structure of the StreamIt graphs to a series-parallel hierarchical organization. This is a conscious design choice of the StreamIt designers[27] since it simplifies the textual description of the graph. The use of these connectors is demonstrated on fig. 1 where a program implementing the Matrix Multiplication in StreamIt is provided.

StreamIt adapts the granularity and communications patterns of programs through graph transformations [17], belonging to one of these three types: fusion transformations cluster adjacent filters, coarsening their granularity; fission transformations parallelize stateless filters decreasing their granularity; reordering transformations operate on splits and joins to facilitate fission and fusion transformations. Complementary transformations have also been proposed. For example optimizing transformations proposed in [1] take advantage of algebraic simplifications between consecutive linear filters. On cache architectures, fusion transformations proposed in [26] optimize filters to instruction and data cache sizes.

Brook Brook is a stream programming language that targets different architectures: Merrimac, Imagine, but also graphic accelerators. The Brook syntax is inspired by the C language and implements many extensions for stream manipulation. Streams are typed and possess an arbitrary high number of dimensions.

To the best of our knowledge, current Brook compilers are limited to primitive types on streams (no composite or arrays types).

Filters in Brook are normal C functions but preceded with the keyword `kernel` which indicates they accept streams as parameters. Side effects between filters must be strictly confined to stream communications. The access rights of kernels to stream parameters can be specified as write only, read only or random access, which allows the compiler to optimize memory handling.

To express data reorganizations, Brook introduces a set of functions that reorder the elements within a stream:

- `streamStencil` which extracts blocks of data inside a stream by moving a *stencil* inside its shape.
- `streamStride` allows to select the elements in a stream that are separated with a given stride factor.
- `streamRepeat` allows to duplicate elements in a stream.
- `streamMerge` which combines elements from multiple streams.

In [23] an optimization method is proposed to leverage the affine partitioning framework. To do this it translates the above data reorganizations functions into a set of dependences that can be optimized in the polyhedral model. The dependence equations are not necessarily affine, but according to the authors in many cases they can be reduced to a set of equivalent affine equations.

1.2 Expressing streams through dependencies

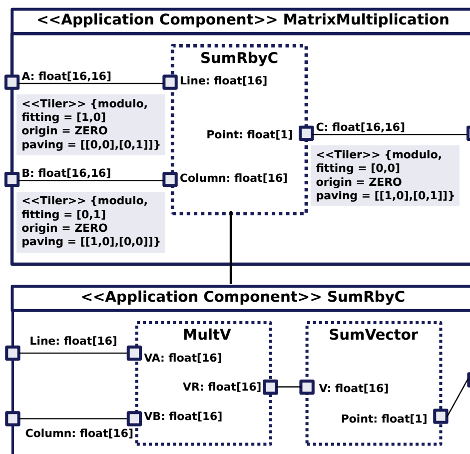


Fig. 2. Matrix multiplication in Array-OL as viewed in the IDE Gaspard2[8].

Array-OL Array-OL[16] is a language that specifically targets signal applications. Data is represented using multidimensional arrays which can have one infinite dimension (for example to represent time). Arrays are toroidal avoiding border effects in many applications.

In Array-OL, programs are composed of filters that can exchange data arrays through streams. The program description is done at two levels:

- *the global level* describes connexions between filters using an oriented acyclic graph. A filter can have multiple input and output streams. The absence of cycle forbids feedback loops but simplifies scheduling.
- *the local level* describes dependences between filter inputs and outputs. Each input has an associated *tiler* describing the order in which the filter consumes its elements. A tiler is composed of an origin point, a shape, a paving matrix and a fitting matrix. Each time the filter is executed, it consumes a stencil of elements inside the input arrays, determined by the tilers *shape*. The stencil is then translated according to the origin point, paving and fitting matrix. Just like a tiler determines the dependences for each input stream, each output stream also possesses a tiler describing the order of the elements produced by the filter.

Array-OL programs can be developed in a visual IDE called Gaspard[12] which eases the visualization of the local and global model. Fig. 2 shows a matrix multiplication program as seen in Gaspard.

Array-OL programs can be transformed into a Kahn Process Network[3] which enables a concurrent execution of the tasks. Recent works on Array-OL compilation propose a set of optimizations that fusions Array-OL filters to coarsen the grain of parallelism and factor producer-consumer dependencies to increase reuse in pipelines[15][13]. But to the best of our knowledge, there is no automatic framework to decide when these transformations should be applied.

Block Parallel Block Parallel[6] also targets signal applications. The author argues that the multidimensional formulations proposed, for example, by Array-OL are difficult to optimize since each new dimension increases the number of possible data traversals. He pushes for a compromise between expressivity and ease of compilation, by allowing only data shapes of one or two dimensions and restricting the input programs to acyclic graphs. He combines the input and output filter dependencies proposed by Array-OL with the splitter and joiners proposed by StreamIt (used to introduce data parallelism in the application). The author proposes a set of optimizations to increase reuse in the filters and optimize the order of access. Yet these transformations are very limited since they only work on the programs that can be expressed using Block Parallel filter dependencies. For example, Matrix multiplication of Fast Fourier Transformation are out of the scope of Block Parallel optimizations.

1.3 A two-levels approach

Brook and StreamIt propose a low-level language to manipulate streams: StreamIt uses joiners and splitters that route and copy data through the graph while

Brook manipulates the streams using primitives that reorder and select elements on streams. StreamIt and Brook propose efficient optimizations. StreamIt uses fusion, fission and reordering transformations to optimize the throughput and Brook leverages the optimizations offered by affine partitioning[23]. Array-OL or Block Parallel on the other hand propose a high-level description of data dependences[16][6]. Nevertheless the high-level description comes at a price: optimizations in theses languages are harder to implement, in particular optimization regarding the routing of data through the application. As pointed in [14][15], the formalism underlying Array-OL dependences (ODT) makes difficult to express some transformations: since the result of the optimizations must be a valid ODT Array-OL dependences set, the palette of available transformation is limited.

Instead of using a single language to both describe and optimize the application, we propose a two-level language approach. A high-level typed DSL, called SLICES, is used to describe the data dependencies. SLICES is then converted to an intermediary stream language, SJD, which can be efficiently optimized with a set of semantically preserving stream graph transformations. The use of different levels of abstraction allows a clean separation of concerns and a modular compilation chain. The expressivity problematic is addressed by a domain specific high-level typed language which can grow more complex to accommodate the users' demands. The optimization problematic is addressed by a simple and restricted language easier to optimize.

Recent works have also considered intermediary stream representation to capture the parallelism and flow of data information. Erbium[25] proposes a data flow intermediary representation enabling mainstream compilers to better optimize stream applications. Fastflow [2] is a parallel programming runtime based on skeletons that also advocates a multi-layered approach. The high-level layer is a library of very general parallel patterns (Farm, Pipeline, etc.) that are build upon the simple but efficient lock-free queues of the lower layers.

2 A high-level DSL: SLICES

The high-level *domain specific language* SLICES enables to model the multidimensional data dependencies of filters in signal applications. For this, the domain of each filter is described as a combination of multidimensional slicings over the input streams. The language is built around five concepts: shapes, grids, blocks, iterators and zippers. We are going to present the language through a practical example: the data dependencies of a matrix multiplication filter. To multiply two matrices, as in $C_{y0,x1} = A_{y0,x0} \times B_{y1,x1}$, we must extract the lines of A and pair them with the columns of B, before processing them through a dot-product filter.

Lines *1-2* of the program in fig. 3 instantiate a datafilter embedding SLICES code. The filter has two stream inputs with `float` values containing the elements of each matrix and is parametrized with the matrix dimensions.

Shapes SLICES allows us to restructures input streams into multidimensional views using *shape* types. In line *3*, we cast the raw input of the first matrix to a

```
1  (float, float) -> float datafilter
2  PairRowsAndCols (int x0, int y0, int x1, int y1) {
3     shape  [x0,y0] A    =     input(0)
4     shape[x1,y1] B    =     input(1)
5     for l  in A[0:1:,::] x (0:x0-1,0:0):
6       for c  in B[::,0:1:] x (0:0,0:y1-1):
7         push l
8         push c
9  }
```

Fig. 3. SLICES program that captures a matrix multiplication communication pattern. First the raw stream inputs are casted to `shape` to view them as matrices. Then the rows of A are paired with the columns of B in the iterator loop, and pushed to the output. In the graphical examples we have chosen `x0 = 5 and y0 = 4`.

type `shape[x0,x1]`. This produces a view `A`, where `input(0)` is seen as a stream of matrices. In practice we can use an arbitrary number of dimensions in a *shape* type.

Blocks Blocks are used to select a set of elements inside a *shape* view. A block is defined by a d-dimensional box parametrized by its min and max coordinates on each dimension: $(a_1{:}b_1, \ldots, a_d{:}b_d)$ with $a_i, b_i \in \mathbb{Z}$. In our example we want to extract from view `A` each horizontal line. To achieve this we define in line *5* the block `(0:x0-1, 0:0)`.

Grids To select *each and every* horizontal line from A, we must apply the previous block to each row. To define a set of anchor points where a block is applied, SLICES provide the grid constructor. A grid is defined by three parameters for each dimension i: the lower bound of the grid l_i, the upper bound of the grid h_i, and the stride δ_i. This triplet describes for each dimension i, the set of points $G_i = \{\delta_i.k.\boldsymbol{e_i} : \forall k \in [|\frac{l_i}{\delta_i}; \frac{h_i}{\delta_i}|]\}$. The elements of a grid are constructed by computing the Cartesian product of the G_i in lexicographical order. The grid operator uses a standard slicing notation where l_i, h_i, δ_i are separated by colons and each dimension is separated by commas. For example `V[0:10:2]` would describe the points in V that are between position 5 and 10 with a stride of 2. Out of simplicity, it is possible to omit one or more values of the triplet; missing values are replaced by sensible default values (0 in place of l_i, the size of the dimension in place of h_i, 1 in place of δ_i). For instance, the previous example could be rewritten `B[:10:2]`.

A block can be applied upon a grid with the *grid* $\times$ *block* operator. This returns the set of points produced by centering the block around each point of the grid. For example, to extract the rows of A we must apply the previous block to every point in the first column of A in line *5*. Indeed the grid `A[0:1:, 0:y0:1]` defines the first column as a set of anchor points and is combined with the `(0:x0-1,0:0)` block.

When applied to a grid, successive blocks may overlap which is convinient to write filters working on sliding windows of data (eg. FIR or Gauss filter). Blocks may also partially fall outside of the view *shape* to handle border effects.

Iterators `shape`, `grid` and `block` return instances of the *iterator* type that we can interleave using nested "**for** `v` **in** *iterator*" loops. A loop iterates over the elements of the given iterator, binding each returned set of points to the variable `v`. In lines *5-6* we iterate over the rows of A and the columns of B, and produce each pair to the output using the `push` keyword.

For a complete presentation of the language and of its underlying type system please refer to [10]. SLICES is able to capture frequently used data reorganization patterns in signal applications: [9] presents the design with SLICES of a Sobel filter, a Gauss filter, a Hough filter and the odd-even mixing stage of a fast Fourier butterfly transformation.

3 Intermediary representation SJD

The intermediary representation must provide a framework for the efficient optimization of applications. To accomplish this objective, two requirements must be satisfied: first, the representation must be simple enough to enable a well-understood set of optimizations ; second, the representation should capture all the possible static data reorganizations (we cannot optimize what we cannot model). A high-level multidimensional representation like Array-OL and Block Parallel does not satisfy the first requirement, since the optimization complexity grows with the number of dimensions [6]. A simple graph language like StreamIt is much easier to optimize. Nevertheless, by design StreamIt imposes a hierarchical series-parallel structure on the application graphs that cannot model all the possible static data reorganization. As a simple example [27] shows that StreamIt can never alter the position of the first element of a stream. Therefore, in StreamIt to reverse the order of a vector of elements we cannot use splitters and joiners and must hide the communication pattern inside a filter. Another limitation of the hierarchical graph restriction is that it cannot capture all the optimizing transformations we propose (UnrollRemove or BreakJS in section 4.2 cannot be expressed with a series-parallel graph). To build our intermediary representation, we have removed the hierarchical restriction from StreamIt graphs that hampers the expressivity of the language.

Source (**I**) and **Sink** (**O**) nodes model respectively the program inputs and outputs. The source produces a stream of inputs elements, while the sink consumes all the elements it receives. A source producing always the same element is a *constant* source (**C**). If the elements in a sink are never observed, it is a *trash* sink (**T**).

Functions in the imperative programming paradigm are replaced by **filter** nodes ($\mathbf{F}(\mathbf{c_1}, \mathbf{p_1})$). Each filter has one input and one output, and an associated pure function f (*i.e.* with no internal state). Each time there are at least c_1 elements on the input, the filter is fired: the function f consumes the c_1 input elements and produces p_1 elements on the output.

Another category of nodes dispatch and combine streams of data from multiple filters, routing data streams through the program and reorganizing the order of elements within a stream.

Join $\mathbf{J}(\mathbf{c_1} \ldots \mathbf{c_n})$: A Join node has n inputs and one output. Each time it is fired, it consumes c_i elements on every i^{th} input and concatenates the consumed elements on its output.

Split $\mathbf{S}(\mathbf{p_1} \ldots \mathbf{p_m})$: A split node has m outputs and one input. A split consumes $\sum_i p_i$ elements on its input and dispatches them on the outputs (the first p_1 elements are pushed to the first output, then p_2 elements are pushed to the second, etc.).

Dup $\mathbf{D}(\mathbf{m})$ has one input and m outputs. Each time this node is fired, it takes one element on the input and writes it to every output, duplicating its input m times.

By breaking the hierarchical constraint of StreamIt and introducing trash nodes, SJD is able to capture all the finite static data reorganizations in the

application: for example, a vector can be easily reversed without using filters. In [9] we prove the following result.

Theorem 1 (Expressivity). *SJD graphs without filters exactly capture the reorganizations* $[i_{\phi(1)}, \ldots, i_{\phi(m)}]$ *where* $[i_1, \ldots, i_n]$ *are the elements being reorganized and* ϕ *is an application from* $[1, \ldots, m]$ *to* $[1, \ldots, n]$.

In other words, SJD graphs enables any finite permutation, reordering, duplication or pruning of elements. Fig. 4 demonstrates those features on a simple example.

Like StreamIt, our intermediate representation is built upon the a synchronous data flow (SDF) computation model[22] where nodes are actors that are fired periodically and edges represent communication channels. We can schedule an SDF graph in bounded memory if it has no deadlocks and is consistent. A *consistent SDF graph* admits a repetition vector $\boldsymbol{q_G} = [q_1, q_2, \ldots, q_{N_G}]$ where q_N is the repetition number of node N. A schedule where each actor N is fired q_N times is called a *steady-state* schedule. Such a schedule is rate matched: for every pair of actors (U, V) connected by an edge e, the number of elements produced by U on e is equal to the number of elements consumed by V on e during a steady-state execution (data dependencies are satisfied). The number of elements exchanged in a steady-state through edge e is noted $\beta(e)$.

3.1 Compiling SLICES to SJD

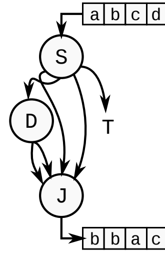


Fig. 4. Example of data reorganizations enabled by SJD. (The split and join consumptions and productions are always 1 in this graph.)

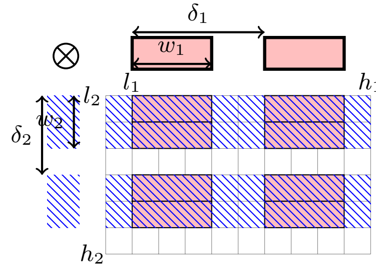


Fig. 5. Multidimensional grids and blocks extraction.

To be able to optimize programs written using SLICES, we must compile SLICES programs to the intermediate representation SJD. A detailed description of the compilation process is outside the scope of this chapter, please see [10]; however, the main steps are:

1. Each SLICES datafilter is parsed and type checked. For every SLICES program that type checks the compiler is able to generate a correct reentrant SJD graph without dead-locks.

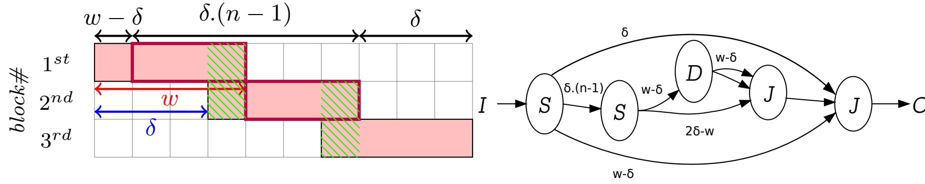


Fig. 6. Compiling 1D blocks that partially overlap ($1 < \frac{w}{\delta} < 2$).

2. Multidimensional grids and blocks, are by construction cartesian products of their 1D counterparts. For instance the following grid and block 2D expression,

$$[l_1 : h_1 : \delta_1, l_2 : h_2 : \delta_2] \times (a_1 : b_1, a_2 : b_2)$$

   can be decomposed as shown in fig. 5 into,

$$([l_1 : h_1 : \delta_1] \times (a_1 : b_1)) \otimes ([l_2 : h_2 : \delta_2] \times (a_2 : b_2))$$

3. We compile each 1D constituent to an equivalent SJD graph using a set of simple patterns. As an example, in the case of partial overlapping blocks ($1 < \frac{w}{\delta} < 2$) the SJD graph produced is given in fig. 6.
4. Our compiler analyzes the nested `for` loops, duplicates and reorders (inserting appropriate Dup and Join nodes in the final graph) according to the iterators length and the nesting depth of `push` instructions.

We prove in [9], that the number of nodes in the SJD graph produced by this compilation process is $\mathcal{O}(p.d.w)$, where p is the number of `push` instructions, d is the maximum number of dimensions used and w is the largest width in any dimension of the extracted blocks. Thus the complexity of the generated graphs is independent of the size of the input shapes. This means that working on large sets of data will not increase the number of nodes in the intermediate representation.

When we compile the SLICES program from matrix multiplication of fig. 3 our compiler generates the SJD graph in fig. 7. The matrix B is transposed using the first $S-J$ pair, then the rows of A and columns of B are duplicated with the $D-J$ pairs and paired together with the final J node, before been sent through the DotProduct filter.

4 Optimizing the intermediate representation

The way we optimize the intermediate representation is through a set of transformations of the program. These transformations alter the communication patterns and the degree of parallelism in the SJD representation of the original program while preserving its semantics.

We follow the formulation given in [5]: a transformation T applied on a graph G, generating a graph G' is denoted $G \xrightarrow{T} G'$. It is defined by a matching

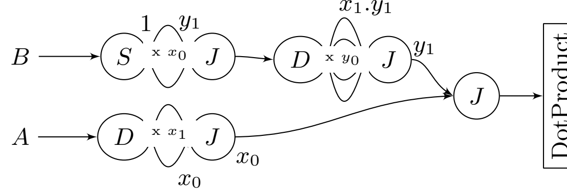


Fig. 7. Intermediate SJD representation equivalent to the SLICES matrix multiplication program (fig. 3). The intermediate representation was automatically generated by our SLICES to SJD compiler.

subgraph $L \subseteq G$ and a replacement graph $R \subseteq G'$. It operates by deleting the match subgraph L from G and replacing it by the replacement subgraph R. The part of graph that remains untouched ($G \backslash L$) is called the *context* of the transformation.

4.1 Soundness of transformations

An optimizing transformation can only be applied if it preserves the semantics of the original program, preserves consistency and does not introduce dead-locks.

In [9] we prove the following sufficient condition where $L(I)$ are the output traces of subgraph L for given input traces I and $\overline{L(I)}$ is the length of the output traces. For simplicity we consider that L and R have only one input and output, but the sufficient condition stands for multiple input/output subgraphs.

Lemma 1 (Local correction). *If a transformation $G \xrightarrow{T} G'$ satisfies*

$$\begin{array}{rcr} \forall I & \Rightarrow & L(I) \textit{ is a prefix of } R(I) \\ \exists b \in \mathbb{N}, \forall I & \Rightarrow & \overline{R(I)} - \overline{L(I)} \leq b \\ L \textit{ is consistent} & \Rightarrow & R \textit{ is consistent} \end{array}$$

then the transformation T is correct.

This lemma establishes the correction of a transformation independly of the context. A transformation that verifies the lemma 1 can be applied to any input SDF program. In particular such a transformation is legal inside a feedback loop in the SJD graph without introducing dead-locks or breaking consistency.

4.2 Transformations

Using the previous lemma we have constructed a set of *correct* transformations on SJD graphs. In fig. 8 a subset of these transformations is presented. The transformations split or reorder the streams of data and modify the expression of concurrency, they can be separated in three groups according to their effect.

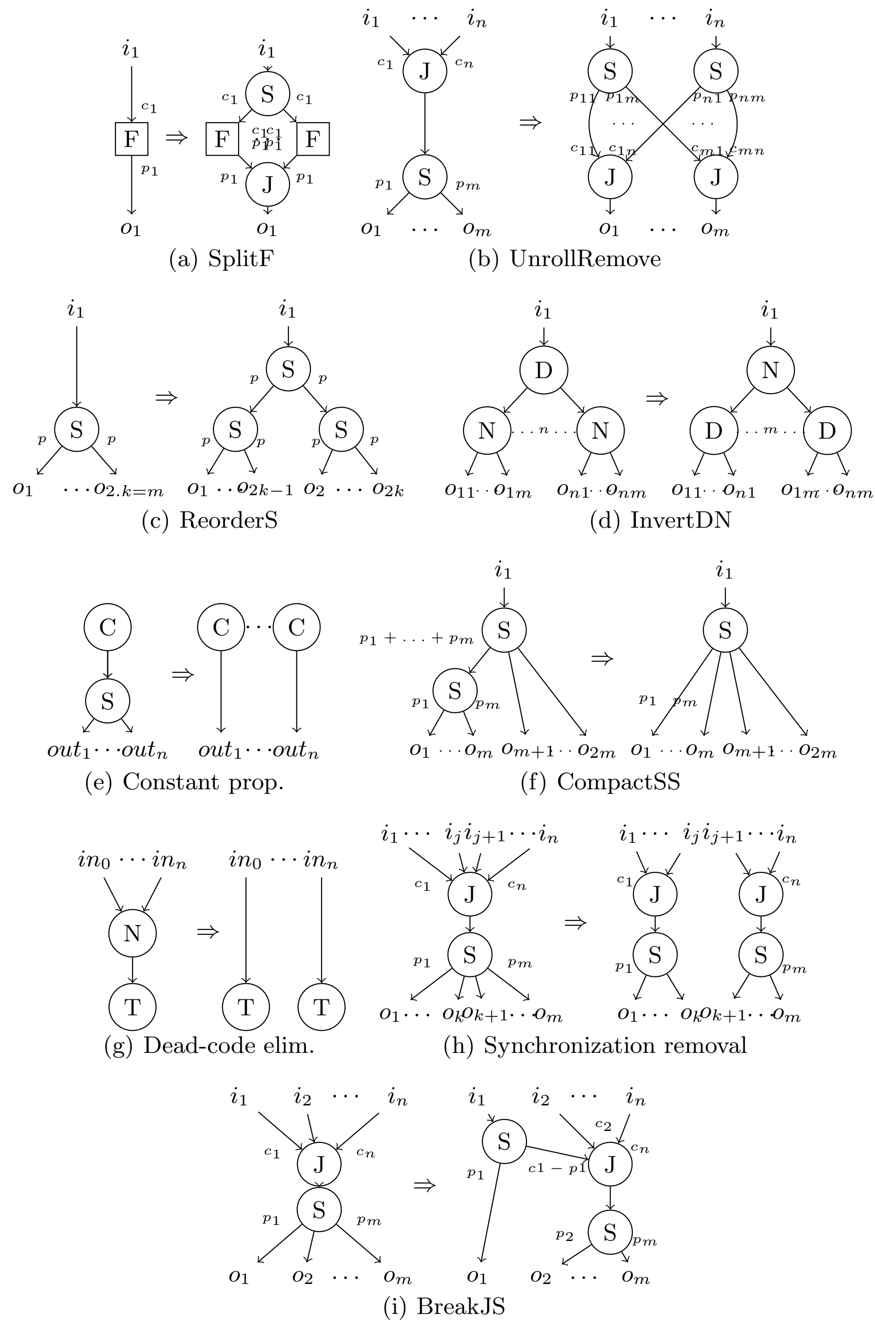


Fig. 8. Set of transformations considered. Each transformation is defined by a graph rewriting rule. Node N is a wildcard for any arity compatible node.

Node removal These transformations rewrite communication structures that use less nodes, for example by removing nodes whose composed effect is the identity.
RemoveJS / RemoveSJ / RemoveD These transformations (not shown in the figure) are very simple and remove nodes whose composed effect is the identity: a Split and a Join of identical consumption and productions, a single branch Dup, a single branch Split, etc ...
CompactSS/CompactDD/CompactJJ (fig. 8(f)) CompactSS (resp. JJ, DD) fuses together a hierarchy of Split (resp. Join, Dup) nodes.

Synchronization removal These transformations remove synchronization points inside a communication pattern, usually by decomposing it into its smaller constituents.
Constant propagation(fig. 8(e)) when a constant source is split we can eliminate the Split duplicating the constant source.
Dead code elimination(fig. 8(g)) eliminate nodes whose outputs are never observed.
BreakJS(fig. 8(i)) breaks Join-Split junctions into smaller constituents, it often triggers **Synchronization Removal**(fig. 8(h)) which tries to find two matching groups in the productions/consumptions of the junction. This allows to break a Join-Split junction into two smaller junctions.

Restructuring These transformations restructure communication patterns. They find alternative implementations which may be more efficient in some targets and sometimes trigger some of the previous transformations.
SplitF(fig. 8(a)) This transformation splits a filter on its input. SplitF introduces split-join parallelism in the programs. Because filters are pure: we can compute each input block on a different filter concurrently.
InvertDN(fig. 8(d)) This transformation inverts a duplicate node and its children, if they are identical. This transformation eliminates redundant computations in a program.
UnrollRemove(fig. 8(b)) This transformation inverts the order between Join and Split nodes. The transformation is admissible in two cases:
1- Each p_j is a multiple of $C = \sum_i c_i$, the transformation is admissible choosing $p_{ij} = c_i.p_j/C$, $c_{ji} = c_i$.
2- Each c_i is a multiple of $P = \sum_j p_j$, the transformation is admissible choosing $p_{ij} = p_j$, $c_{ji} = p_j.c_i/P$.
ReorderS/ReorderJ(fig. 8(c)) ReorderS (resp. ReorderJ) creates a hierarchy of Split (resp. Join) nodes. In the following we will only discuss SplitS. The transformation is parametric in the Split arity f. This arity must divide the number of outputs, $m = k.f$. In the figure, we have chosen $f = 2$. As shown in figure 8(c), the transformation works by rewriting the original Split using two separate stages: odd and even elements are separated then odd (resp. even) elements are redirected to the correct outputs. We have omitted some more complex transformations for simplicity sake. For an in-depth description of the transformation set, please see [9][11].

5 Reducing inter-core communication cost

The previous set of transformations change the degree of parallelism and the communications patterns of the original program. In this section we will demonstrate how they can be used to reduce the inter-core communication cost in a parallel program.

5.1 Measuring inter-core communication cost

To execute a SJD program on a multicore target, we partition the nodes in the SJD graph among the available cores. For a given partitioning $\mathcal{P}$ of G, we define $inter(G, \mathcal{P})$ as the set of edges that connect nodes in different partitions.

The Hockney[19] model distinguish two cost factors in a point-to-point communication: (i) a fixed cost equal to the latency c_0, (ii) a variable cost that increases with the number of streamed elements and depends on the bandwidth bw. The communication cost during a steady-state schedule execution is noted $c_e = c_0 + \frac{\beta(e).s(e)}{bw}$ where $\beta(e)$ is the number of elements exchanged during a steady-state and $s(e)$ is the size in bytes of each element. The inter-core communication cost is computed by aggregating the costs of all the edges that link different cores, $C(G, \mathcal{P}) = \sum_{e \in inter(G,\mathcal{P})} c_e$.

5.2 Exploring the optimization space

We can improve the inter-core communication cost by optimizing two factors: partitioning of the SJD nodes among the processors and the communications patterns between filters.

To partition the SJD nodes among the processors, we solve the following optimization problem: (i), reduce the inter-core communication cost C, (ii) under the constraint that the *work imbalance* among the cores is less than a small threshold (5% in our setup). The work imbalance is the difference of load between the the core which is busiest and the core with the less work. To solve this problem we use the graph partitioner METIS[20]. The threshold makes the balancing constraint a bit more flexible, opening opportunities for the partitioner to improve the communication cost.

To optimize the communication patterns between filters we use the set of transformations presented in sec. 4.2. Given a SJD graph G_0, a derivation is a chain of transformations that can be successively applied to G_0: $G_0 \xrightarrow{T_0} G_1 \ldots \xrightarrow{T_n} \ldots$. Each derivation produces a new variant, semantically equivalent to G_0 but with different communication patterns. Given an initial graph G_0, the number of derivations that exist is very large, how should we pick one ? In [9] we prove that for any given graph there are no infinite derivations, that is to say the optimization space is bounded. To choose which transformations to apply we use the BEAMSEARCH[24] search heuristic which is tuned with a constant parameter $beamsize \in \mathbb{N}$. The algorithm explores the optimization space recursively by applying all the possible transformations to the initial graph G_0,

sorting the produced first-generation variants by their inter-core communication costs and discarding all but the first *beamsize* ones. The algorithm is then applied recursively on the selected best first generation variants. BEAMSEARCH is guaranteed to terminate since the optimization space is finite. These two passes: partitioning and communication optimization are interleaved in an iterative process, depicted in fig. 9, similar to [7].

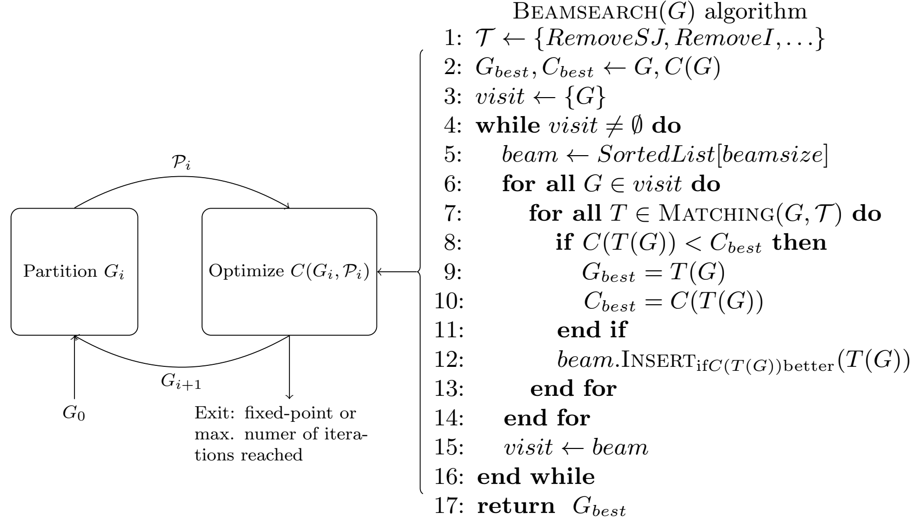


Fig. 9. Reducing the inter-core communication through an iterative process.

6 Evaluation

We have evaluated the inter-core communication reduction technique on a two sets of signal application benchmarks: a first set of SLICES programs, Matrix multiplication, Gauss Filter, Sobel Filter which are first compiled to the SJD intermediate representation and a second set of programs from the StreamIt benchmarks [18], Bitonic sort, FFT, DES, DCT which are directly translated to the SJD representation.

The target architecture is quadcore SMP Nehalem (Xeon© W3520 at 2.67GHz) with 256KB of L2 cache and a shared 8MB L3 cache. The communications between cores happen through the L3 cache with a very low latency (here we suppose that $c_0 = 0$).

We have measured the communication cost for two versions of the programs: the *original* one is mapped with METIS to reduce the communication cost but graph transformations are *not* applied to it; the *optimized* one is mapped with METIS *and* optimized using the graph transformation set.

Table 1 summarizes the inter-core communication reductions achieved by our optimization framework. The mean percentage of reduction among all the programs is 49.9% and the mean time spend optimizing the programs is 3.4 minutes.

The DES encryption program shows no gains at all: the program admits very few graph transformations that have no impact on the global layout of communications.

The gains in MM-COARSE can be attributed to several transformations of the flow graph that can be seen in fig. 10. After optimization, the synchronization bottlenecks (nodes $J8$ and $S15$) have been removed. The transposition of matrix B has been decomposed in blocks and distributed among the four cores. Finally, duplications have are made locally which reduces the volume of inter-core communications.

The GAUSS filter is a bi-dimensional sliding-window filter that extract overlapping 3×3 windows of data from the input image. Our transformations are able to break the sliding-window extraction among the different cores and reorganize the Split, Join and Dup nodes to increase the horizontal reuse of data among filters, reducing inter-core communication.

The HOUGH filter computes the Hough transformation in a tight loop. Our optimization framework breaks this loop in three smaller loops that are distributed among the cores, making the state in the loop local to each processor.

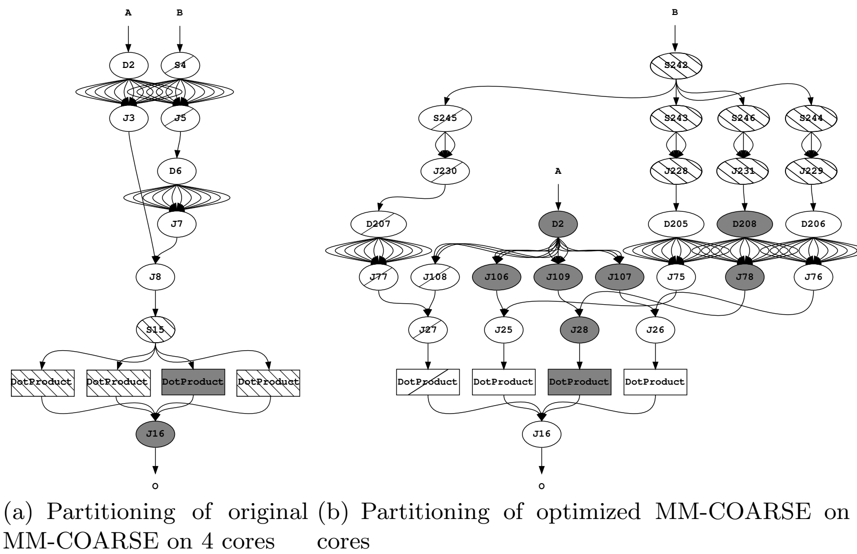
The FFT and DCT filters possess many synchronizations points that are removed by our transformation process allowing a better partitioning among the cores.

	MM-COARSE	GAUSS	BITONIC	HOUGH	FFT	DES	DCT
original (B)	18864	10563200	384	48480000	384	192	3072
optimized (B)	6624	7340000	256	401624	192	192	1956
C reduction (%)	64.9	30.5	33.4	99.2	50	0	36.3
opt. cost (s)	5	18	925	411	10	66	10

Table 1. Inter-core communication cost reduction. "original" and "optimized" represent the inter-communication volume per-steady state for the original program and the optimized program in Bytes. "C reduction" is the reduction percentage of inter-core communications computed as $\frac{C_{original} - C_{optimized}}{C_{original}}.100$. "opt. cost" is the time spend optimizing the SJD representation in seconds.

6.1 Impact on the execution time

We have implemented[9] a complete backend that compiles the intermediate SJD representation to C code running on a SMP architecture. The compilation process can be broken in a series of steps: partitioning, scheduling, task fusion, communication fusion and code generation.



(a) Partitioning of original MM-COARSE on 4 cores (b) Partitioning of optimized MM-COARSE on 4 cores

Fig. 10. Optimizations applied to MM-COARSE. In the original program the inter-core communications cost is high since the program presents many synchronizations points: for example at runtine $J8$ and $S15$ quickly become communication bottlenecks. After optimization: $J8$ and $S15$ have been split, the transposition of matrix B has been divided in blocks and distributed among the four cores; duplicate nodes (nodes $D205$, $D208$, etc.) have also been distributed among the cores, since the consumers of the duplicate streams are all in the same core, duplications can be compiled to multiple reads to the same buffer and are not added to the inter-core traffic.

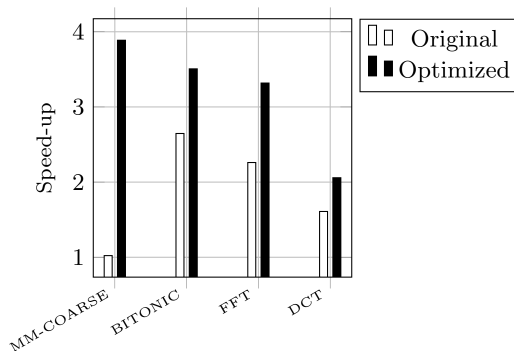


Fig. 11. Impact of the communication reductions on the performance: Speedups of the original program and the optimized program on a SMP Nehalem quadcore. The executions times are normalized to the reference (StreamIt original program execution time on a single core).

Our compiler takes into account two types of parallelism: (i) task or data parallelism which is explicit in the SJD graph and (ii) pipeline parallelism which allows to overlap successive executions of consumers and producers in a Stream Graph Modulo scheduling[21]. In the Stream Graph Modulo scheduling, a node execution can be overlapped with its communications, hiding the cost of the cheaper operation. In this context, the performance of a program is determined by the maximum between: (i) the time needed to complete a schedule tick of a node execution, (ii) the time needed to copy the productions of the node to the consumer.

Reducing the inter-core communication cost should therefore have an impact on performance on communication-bound programs. To verify this hypothesis we have selected among the previous benchmarks the programs for which we had a reference implementation (StreamIt) and for which our inter-core communication reduction was successful.

Our baseline is the execution time of the StreamIt version of the program compiled on **a single core** with the command `strc -O3`. The speed-ups presented are normalized by the StreamIt single-core performance. Then we have measured the execution time of the SJD original program and the SJD optimized program using our backend on **four cores**. Fig. 11 presents the results obtained. The mean speedup without applying optimizations is $\times 1.85$ while the mean speedup with inter-core communications reductions is $\times 3.2$.

7 Conclusion

The challenge for stream programming on multicore architectures is to describe stream manipulation, dependent on the application, and adapt this stream to complex and changing multicore architectures. In particular, the program has to adapt to the parallelism of the target architecture, to the bandwidth limitation and limited cache (or buffer) sizes.

Stream transformations and optimizations are the key to this adaptation, and both parallelism and communication metrics can be evaluated on a flow graph describing the stream. Being able to explore different stream formulation according to the metrics to be optimized is essential to obtain high performance stream programs. So far, research efforts in stream specific languages have focused on two language categories: languages such as Array-OL and Block-Parallel describe streams through dependences between filters, languages such as StreamIt and Brook explicitly manipulate stream operators. While it is more natural to the developer to describe its program as a set of filters communicating through dependences, stream optimizations are hampered by the strong constraints imposed by the underlying dependency model. For languages explicitly manipulating stream objects, the range of possible optimizations is larger but is suffers from the difficulty to describe complex flows.

We have presented in this chapter a novel approach for stream programming. Based on the fact that the description of the stream and its optimization are seperate concerns, we proposed an approach based on two domain specific lan-

guages, one for each concern. This approach retains both the expressivity of high level languages such as Array-OL and Block Parallel and the rich optimization framework, similar to StreamIT and Brook.

SLICES manages to retain a high-level multidimensionnal expression of programs while enabling an efficient compilation to the intermediary language. The SJD intermediary language extends the expressivity of StreamIt by allowing non hierarchical graphs, extending the range of possible optimizations. We introduce a formal framework for building correct transformations of SJD programs and an iterative exploration algorithm to optimize a program according to a metric. This method achieves a mean 49.9% reduction of the inter-core communication cost among a set of significant benchmarks. We expect our results to be even more relevant as the number of cores increases, but this will be shown as future work. A limited exploration of the space of solutions seems to be difficult to overcome so far: the metrics, such as inter-core communication or memory consumption are non linear metrics.

References

1. S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 126–136. ACM, 2005.
2. M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating sequential programs using FastFlow and self-offloading. Technical Report TR-10-03, arXiv:1002.4668, Universita di Pisa, 2010.
3. A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *Proc. Parallel Architectures, Algorithms and Networks, 2005.*, page 6, 2005.
4. S. Amarasinghe, M. Gordon, M. Karczmarek, J. Lin, D. Maze, R.M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2):261–278, 2005.
5. Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph Transformation for Specification and Programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.
6. D. Black-Schaffer. *Block Parallel Programming for Real-Time Applications on Multi-core Processors.* PhD thesis, Stanford University, 2008.
7. Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–66, New York, NY, USA, 2009. ACM.
8. A. Charfi, A. Gamatié, A. Honoré, J.L. Dekeyser, and M. Abid. Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08), Mulhouse, France*, 2008.
9. P. de Oliveira Castro. *Expression et optimisation des réorganisations de données dans du parallélisme de flots.* PhD thesis, Université de Versailles Saint Quentin en Yvelines, 2010.

10. P. de Oliveira Castro, S. Louise, and D. Barthou. A Multidimensional Array Slicing DSL for Stream Programming. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 913–918. IEEE, 2010.
11. P. de Oliveira Castro, S. Louise, and D. Barthou. Reducing memory requirements of stream programs by graph transformations. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 171–180. IEEE, 2010.
12. F. Devin, P. Boulet, J.L. Dekeyser, and P. Marquet. GASPARD: a visual parallel programming environment. In *Proceedings of International Conference on Parallel Computing in Electrical Engineering, 2002. PARELEC'02.*, pages 145–150, 2002.
13. P. Dumont. *Spécification Multidimensionnelle pour le traitement du signal systématique*. PhD thesis, Université des sciences et technologies de Lille, 2005.
14. C. Glitia. *Optimisation des applications de traitement systématique intensives sur system-on-chip*. PhD thesis, Université des Sciences et Technologies de Lille, 2009.
15. C. Glitia and P. Boulet. High level loop transformations for systematic signal processing embedded applications. In *Proc. of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 187–196. Springer-Verlag, 2008.
16. C. Glitia, P. Dumont, and P. Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Syst. Signal Process.*, 21:105–131, June 2010.
17. Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. "A Stream Compiler for Communication-Exposed Architectures". In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 291–303. ACM, 2002.
18. StreamIt Group. Streamit benchmarks. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.
19. Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3):389–398, 1994.
20. George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
21. Manjunath Kudlur and Scott Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Proc. of the SIGPLAN conf. on Programming Language Design and Implementation*, pages 114–124. ACM, 2008.
22. EA Lee and DG Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
23. Shih-wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Int. Symp. on Code Generation and Optimization*, Washington, DC, USA, 2006. IEEE.
24. Bruce T. Lowerre. *The Harpy Speech Recognition System.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1976.
25. C. Miranda, P. Dumont, A. Cohen, M. Duranton, and A. Pop. ERBIUM: a deterministic, concurrent intermediate representation for portable and scalable performance. In *Conf. Computing Frontiers*, pages 119–120, 2010.
26. J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7):126, 2005.
27. W. Thies. *Language and Compiler Support for Stream Programs.* PhD thesis, Massachussetts Institute of Technology, 2009.