Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization

Nicolas Vasilache[1], Benoit Meister[1], Muthu Baskaran[1], and Richard Lethin[1]

[1]{vasilache,meister,baskaran,lethin}@reservoir.com
[1]Reservoir Labs Inc., New-York, NY, USA

ABSTRACT

We describe a novel loop nest scheduling strategy implemented in the R-Stream compiler[1] : the first scheduling formulation to jointly optimize a trade-off between parallelism, locality, contiguity of array accesses and data layout permutations in a single complete formulation. Our search space contains the maximal amount of vectorization in the program and automatically finds opportunities for automatic multi-level vectorization and simd-ization. Using our model of memory layout, we demonstrate that the amount of contiguous accesses, vectorization and simd-ization can be increased modulo data layout permutations automatically exposed by our technique. This additional degree of freedom opens new opportunities for the scheduler that were previously out of reach. But perhaps the most significant aspect of this work is to encompass an ever increasing number of traditional optimization phases into a single pass. Our approach offers a good solution to the fundamental problem of phase ordering of high-level loop transformations.

1. INTRODUCTION

The problem of optimizing loop nests has been extensively studied in the last five decades [13]. So-called "traditional compilers" perform these optimizations on a tree-based representation of the program. Unfortunately, this representation suffers from crippling limitations such as code explosion arising from the application of successive transformations and the dreaded phase ordering problem. The polyhedral model [2, 8, 10] is a mathematical representation that greatly reduces the code explosion and phase ordering issues related to sequences of loop transformations [9]. A notable property of this model is that a single solution to an optimization problem is a combination of many loop transformations in the tree-based representation. Such a solution represents a whole class of (infinitely many) sequences of equivalent program transformations. Intuitively, the reader may use the simple analogy of a line in a plane: a line contains an infinite number of points (the transformations in the tree-based representation) but it can be described by a single equality (the représentant of this class of transformations in the polyhedral representation). Despite all the recent progress in optimizing the schedule of loop nests at a high level [5, 6, 17], a general formulation for discovering vectorization is still missing.

This paper is decomposed as follows. First we remind the reader of polyhedral concepts and restate one of the results of Vasilache's thesis: the construction of a *convex linear space containing all legal affine schedules of loop nests* [19]. In the following section, we build on this space to incrementally introduce constraints expressing contiguity of a single array reference along an arbitrary memory dimension and an arbitrary schedule dimension. We then mix constraints on multiple arrays and parallelism to derive vectorization constraints. These constraints allow the extraction of vector loops at various levels in the loop nest hierarchy. We proceed by adding data layout permutations to our formulation and discuss the benefits this additional degree of freedom entails. Lastly we discuss the quality of the schedules and compilation time implications of our successive formulations on a set of 400 kernel benchmarks.

[1]The technologies described in this paper are Patent Pending, Reservoir Labs, Inc. This work has been supported by funding from DARPA under the UHPC program, contract HR0011-10-3-0007, subcontract from Intel Corporation.

IMPACT 2012
Second International Workshop on Polyhedral Compilation Techniques
Jan 23, 2012, Paris, France
In conjunction with HiPEAC 2012.

http://impact.gforge.inria.fr/impact2012

2. THE CONVEX AFFINE SPACE OF ALL LEGAL SCHEDULES

The polyhedral model is a mathematical abstraction to represent and reason about programs in a compact representation. R-Stream operates on a generalized dependence graph (GDG)-based IR containing, among other, the following information.

2.1 GDG-based Intermediate Representation

Statements : A statement S is a set of operations grouped together in our internal representation. Statements are the nodes of the GDG. A statement in the model often corresponds to a statement in the original program. Depending on the level of abstraction, a statement can be arbitrarily simple (i.e. micro-code) or arbitrarily complex (i.e. external precompiled object).

Iteration Domains : An iteration domain $\mathcal{D}^S$ is an ordered set of iterations associated to each statement S. It describes the loop iterations in the original program which control the execution of S. To model multiple levels of nested loops, iteration domains are multi-dimensional sets. We denote the order between 2 iterations i_1 and i_2 of S by $i_1 \ll i_2$ [2] if $S(i_1)$ occurs before $S(i_2)$ in the program.

Memory References : A memory reference F is a function that maps domain iterations to locations in the memory space. The image of $\mathcal{D}^S$ by F represents the set of memory locations read or written by S through memory reference F. If F is injective, distinct memory locations are touched; otherwise, memory reuse exists within the program. Each statement can access multiple memory references in read and/or write mode.

Schedules : A scheduling function Θ^S is a function that maps the iterations of S to time. It is a partial order that represents the relative execution order of each iteration of S relative to all other iterations of any statement in the program. We use well-established terminology [9] where Θ^S is decomposed in (1) a linear part α that encompasses unimodular and non-unimodular loop transformations, (2) a parametric constant part Γ for expressing multi-dimensional shifts and (3) a constant β vector that encodes all possible imperfectly nested loop structures. If the α function is injective, the output program is sequential; otherwise parallel iterations exist. In particular, the order $\ll$ extends to time after scheduling is applied.

Dependences : A dependence $\{T \to S\}$ is a relation between the set of iterations of S and T. It conveys the information that some iteration $i^T \in \mathcal{D}^T$ "depends on" $i^S \in \mathcal{D}^S$ (i.e. they access the same memory location by application of a memory reference) and that $i^S \ll i^T$ in the original program. We write the set relation $\{(i^T, i^S) \in \{T \to S\}\}$ to refer to the specific iterations of T and S that take part in the dependence. With this notation, in the case of a read-after-write dependence, T would be a read and S would be a write.

2.2 Multi-dimensional Formulation

Dependences in the program are captured by dependence edges in the GDG. Each edge is decorated by a dependence polyhedron. Feautrier [8] showed that a necessary and sufficient condition for a transformed program to be legal is that the order of all iterations in a dependence be preserved: $\forall (i^T, i^S) \in \{T \to S\}\ \Theta^S(i^S) \ll \Theta^T(i^T)$. The affine form of Farkas' lemma, allows the linearization of such constraints into a linear programming problem:

LEMMA 2.1. *Let $\mathcal{D}$ be a nonempty polyhedron defined by $A\vec{x} + \vec{b} \geq \vec{0}$. Any affine function $f(\vec{x})$ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination: $f(\vec{x}) = \lambda_0 + \vec{\lambda}^t(A\vec{x} + \vec{b})$, with $\lambda_0 \geq 0$ and $\vec{\lambda}^t \geq \vec{0}$. λ_0 and $\vec{\lambda}^t$ are called Farkas multipliers.*

This relationship is based on lexicographic ordering. Consider a loop nest with induction variables $(i_1 \dots i_d)$. Any dependence relation must be (1) strongly satisfied at some loop depth $k \in [1, d]$ (i.e. $\Theta_k^T - \Theta_k^S \geq 1$), (2) weakly satisfied until that depth (i.e. $1 \leq l < k \Rightarrow \Theta_l^T - \Theta_l^S \geq 0$) and (3) not influence correctness after is has been strongly satisfied (i.e. $k < l \leq d \Rightarrow \Theta_l^T - \Theta_l^S \geq -\mathcal{N}_\infty(\vec{n} + 1)$, where $\mathcal{N}_\infty$ is a large enough constant [3] and $\vec{n}$ represents the global parameters in the program). This definition of satisfaction encompasses traditional correctness criteria of loop-carried and loop-independent dependences [13].

Each dependence must be strongly satisfied at some depth k. We encode the satisfaction of a dependence $\{T \to S\}$ at a depth k by a decision variable $\delta_k^{\{T \to S\}} \in \{0, 1\}$. Following a simplification of Vasilache's formulation proposed by Feautrier, we write: $\forall \{T \to S\}, \sum_{k=1}^{min(d^S, d^T)} \delta_k^{\{T \to S\}} = 1$. The satisfaction constraints become:

$$\forall\ \Delta = \{T \to S\},\ \forall\ k \in [1, min(d^S, d^T)],\ \forall\ (i^T, i^S) \in \Delta :$$
$$\begin{cases} \delta_k^\Delta \in \{0,1\} \\ \sum_{l=1}^{min(d^S,d^T)} \delta_l^\Delta = 1 \\ \Theta_k^T(i^T) - \Theta_k^S(i^S) \geq \\ \qquad -\mathcal{N}_\infty \left(\sum_{l=1}^{l \leq k-1} \delta_l^\Delta\right).(\vec{n} + 1) + \delta_k^\Delta \end{cases}$$

Figure 1: Convex space of all legal schedules.

For a more in-depth description, the reader should refer to previous work [19, 17].

2.3 The Need for Invertible Schedules

In theory, the α portion of any schedule can be any legal point in the Farkas cone generated by applying Farkas' lemma to the previous convex polyhedron. When α is not invertible, it is always possible to complete it into a full-rank schedule to iterate over all the points in the domain. This completion can be done at code generation time and consists in adding innermost parallel loops to each statement, with a heuristic for fusion [2]. Although this approach does not reduce the number of parallel dimension, it may degrade the parallelism granularity. Also, it does not exploit degrees of freedom in the program to improve locality, contiguity of memory accesses or vectorization. The problem is that singular schedules give the impression a loop dimension is parallel with the caveat that it may only contain a single point !

Consider the example of a single statement $S(i, j, k)$, suppose the objective function is to maximize the 2 outermost degrees of parallelism; the returned schedule / parallelism markers may be $(i, i, j)/(doall, doall, seq)$. After completion, the schedule will really be $(i, i, j, k)/(doall, doall, seq, doall)$, which is exactly the same as $(doall, seq, doall)$. In other words, a scheduler may wrongly think it has found 2 outermost levels of parallelism when it may have missed the $(k - i, j + i, i + j + k)/(doall, doall, seq)$ solution.

R-Stream uses the multi-dimensional formulation to iteratively search for invertible schedules. At each step, it computes full Θ schedules. These schedules may not be invertible as a result of a single resolution of the ILP, so we incrementally add linear independence constraints while still searching for a full Θ schedule. Since a singular schedule can always be completed at the innermost level, this approach will always succeed if it is performed top-down. Linear independence constraints are added using the expression of orthogonal subspaces [6] with additional enhancements. In particular, we also model loop reversals and their combinations unlike previous work [6, 17].

The contiguity constraints that we detail in Section 3.1 require a multi-dimensional formulation and the search of full

[2]This relationship is based on lexicographic ordering: the equation $\vec{x} \ll \vec{0}$ is satisfied for $\vec{x} = (0, 0, 1, -1)$ but not for $\vec{x} = (-1, 0, 2)$.

[3]For $\mathcal{N}_\infty$ to be bounded a limit is set on the absolute values of the schedule coefficients; typically 16.

Θ functions at each step. The following search strategies are currently implemented in R-Stream : (1) encoding the set of all possible combinations of permutations and reversals; this uses simple linear constraints and does not require an iterative search, (2) top-down iterative search; this is guaranteed to succeed but metrics computed at a given step may be degraded at the next step because the linear independence constraints make them unfeasible.

3. SCHEDULING FOR VECTORIZATION

In this section, we describe some features of the scheduling algorithm implemented in R-Stream. We formulate trade-offs between amount of parallelism, amount of locality and amount of contiguous of memory accesses in a joint problem. This contiguity metric optimizes spatial reuse and is targeted at memory hierarchies where accessing a contiguous set of memory references is *crucial* to obtaining high performance. Such hardware features include hardware prefetch streams, simd and vector operations, and coalescing hardware in GPUs.

3.1 Contiguity

We are interested in a characterization of the linear part of an r-dimensional access function F to a memory reference $\mathcal{R}$[4]. For the remainder of this subsection, F represents the linear part of the memory reference. We define the notion of contiguity exhibited by a schedule relative to a memory reference as follows.

3.1.1 Innermost Reuse Along a Memory Dimension

Definition Let F be a memory reference with r rows accessed by a statement S and α an invertible affine scheduling function for S. We say α exhibits contiguity along memory dimension r and along some (unspecified) schedule dimension, for reference F *iff*: $F \cdot \alpha^{-1} = \begin{bmatrix} & M & \\ m_{r,1} & \dots & m_{r,l} \end{bmatrix}^t$, for some matrix M and some tuple $(m_{r,1}, \ \dots, \ m_{r,l})$ where $card\{i \mid m_{r,i} \neq 0\} \leq 1$.

If additionally, $card\{i \mid m_{r,i} \neq 0\} = 0$, the reuse is both spatial and temporal. The idea is then to include this additional metric in our ILP. It is based on the realization that in the transformed program, the last column of $F \cdot \alpha^{-1}$ represents the innermost dimension of the array access *after code generation* [2]. In the remainder of this paper, given a matrix U, we use the notation $U_{\overline{k}}$ for the sub-matrix obtained by *removing* row k. We suppose the number of loops enclosing statement S is d. In this context, the invertible affine schedule α has d rows. We are interested in a characterization of $\alpha_{\overline{d}}$.

THEOREM 3.1. *Let α an invertible affine schedule, α has contiguity along some memory dimension $k \leq r$ and along schedule dimension d on F iff $Ker\ \alpha_{\overline{d}} \subseteq Ker\ F_{\overline{k}}$.*

PROOF. **If**: Let $\alpha = \begin{bmatrix} \alpha_{\overline{d}} \\ \tau \end{bmatrix}$ and $\alpha^{-1} = \begin{bmatrix} T & t \end{bmatrix}$. Since $\alpha \cdot \alpha^{-1} = \mathbf{I}$, we have $\alpha_{\overline{d}} \cdot t = 0$ (i.e. $t \in Ker\ \alpha_{\overline{d}}$). If $Ker\ \alpha_{\overline{d}} \subseteq Ker\ F_{\overline{k}}$, then $t \in Ker\ F_{\overline{k}}$ (i.e. $F_{\overline{k}} \cdot t = 0$). It follows, $F \cdot \begin{bmatrix} T & t \end{bmatrix} = F \cdot \alpha^{-1} = \begin{bmatrix} & M_1 & & \\ 0 & \cdots\ 0 & m_{k,d} \\ & M_2 & & \end{bmatrix}^t$ for some M_1, M_2 and $m_{k,d}$.

Only if: Let α be an invertible schedule that exhibits contiguity for F along memory dimension $k \leq r$. Let $\alpha^{-1} = \begin{bmatrix} T & t \end{bmatrix}$. From $\alpha \cdot \alpha^{-1} = \mathbf{I}$, we get $\alpha_{\overline{d}} \cdot t = 0$. From the definition of contiguity, we get $F_{\overline{k}} \cdot t = 0$. These imply $Im\ t \subseteq Ker\ \alpha_{\overline{d}}$ and $Im\ t \subseteq Ker\ F_{\overline{k}}$. Furthermore, since α is of full rank, $\dim(Ker\ \alpha_{\overline{d}}) = 1$. Thus $Im\ t = Ker\ \alpha_{\overline{d}}$ and $Ker\ \alpha_{\overline{d}} \subseteq Ker\ F_{\overline{k}}$. $\square$

COROLLARY 3.1. *Let α an invertible affine schedule. α exhibits contiguity for F along memory dimension $k \leq r$ and schedule dimension d iff $Im\ F_{\overline{k}} \subseteq Im \alpha_{\overline{d}}$.*

The proof is trivial: $Ker\ A \subseteq Ker\ B \Leftrightarrow\ Im\ B \subseteq Im\ A$. A simpler version of this characterization of contiguity had already been discussed by Bastoul [3] in the case of the innermost memory dimension. We go a step further by generalizing it to any memory dimension and by integrating this characterization in our affine scheduler. The previous demonstration generalizes to any row of α and reuse across any schedule dimension $d' < d$ is obtained in the same way as for d.

3.1.2 Simplified Objective Function

For the purpose of readability of this paper, we remove discussions on the benefit of loop fusion/distribution. Still the search space contains all the legal fusion/distribution structures but we do not use them in the objective function.

The inclusion in our integer linear programming problem is done by constructing the following objective function which trades off parallelism for innermost contiguity on the innermost memory dimension. Throughout the formulas of this paper, we use the letter $\mathcal{G}$ to denote the GDG. For each statement in the program:

- w_k is the benefit of dimension k of α_k executing in parallel,
- Δ_k is a boolean variable determining whether dimension k of α_k has doall semantics ($\Delta_k = 1$ iff k is run in parallel),

$$\max \sum_{S \in\ \mathcal{G}} \sum_{k \in [1, d^S]} w_k \Delta_k + Benefit_{cont}$$

Linking the Δ_k to the dependence satisfaction variables using the Farkas lemma is an interesting topic [14]. The selection of the costs for the various w_k is also important. Usually, the coarser the parallelism, the higher the value. We do not detail these points further in this paper.

3.1.3 Constraints for a Single Reference

In this section, we focus on constraints for a single statement S of dimension d accessing a memory reference $\mathcal{R}$ with access function F. The coefficients of the matrix F are known; we are searching for the coefficients of α. We describe contiguity along the innermost schedule dimension d of S and along the innermost array subscript r. For each tuple F, d and r, we create a contiguity decision variable $c^F_{r,d}$ to encode whether contiguity is achieved along memory dimension r and schedule dimension d. α_{sf} denotes the

[4]Suppose a statement accesses A$[2 \cdot i + 3 \cdot j - N + 1][k + 2]$ of memory dimension 2, where N is a constant global parameter. The linear part is $(2 \cdot i + 3 \cdot j, k)$. The linear part of an access is subject to traditional unimodular transformations [1] but independent of loop shifting, loop peeling, fusion or fission.

schedule computed 'so far', while α denotes the total schedule out of which some of the first rows are fixed. $\alpha_{\overline{d}}$ denotes α stripped of its last row d.

Linking the benefits of contiguity into the formulation is done using Corollary 3.1. This is not standard practice in polyhedral schedulers so we give additional details on how to proceed. Note that in general the relationship $Im\ F \subseteq Im\ \alpha$ is not linear so we can only bias the search towards the desired space.

We start by pruning the following trivial cases:

1. if $d = 1$, contiguity is realized iff the access does not have a component along d on another row than the contiguity row,

2. if $Im\ F_{\overline{r}} \subseteq Im\ \alpha_{sf}$ contiguity is already achieved,

3. if $n = dim\ (F_{\overline{r}} \cap \alpha_{sf}) - dim\ (\alpha_{sf})$ is greater than the number of remaining dimensions to schedule, contiguity is trivially impossible.

The remaining cases are as follows.
Case 1 : if $n = dim\ (F_{\overline{r}} \cap \alpha_{sf}) - dim\ (\alpha_{sf})$ is equal to the number of dimensions remaining to schedule, the condition becomes $Im\ F_{\overline{r}} = Im\ \alpha$ which can be written as a set of linear constraints. To this effect, we compute a basis of $F_{\overline{r}}$. Without loss of generality, we also denote this basis by $F_{\overline{r}}$. Let μ denote the line of α currently computed. Applying Corollary 3.1 on $\alpha_{\overline{d}}$, we obtain the constraints:

$$c^F_{r,d} \in \{0,1\} \tag{1}$$
$$\mu - F_{\overline{r}} \cdot \lambda + \mathcal{N}_\infty \cdot (1 - c^F_{r,d}) \geq 0 \tag{2}$$
$$-\mu + F_{\overline{r}} \cdot \lambda + \mathcal{N}_\infty \cdot (1 - c^F_{r,d}) \geq 0 \tag{3}$$

where λ are unconstrained variables that are added to the unknown set and serve the purpose of expressing the linear dependence of μ on R_k; $\mathcal{N}_\infty$ are well-chosen large constants. The reasoning goes as follows. If $c^F_{r,d} = 1$ then the constraints become $\mu = F_{\overline{r}} \cdot \lambda$. If $c^F_{r,d} = 0$, $\mathcal{N}_\infty$ nullifies the effects of these constraints by allowing the trivial solution where all lambda are 0. It is important to realize the λ and $c^F_{r,d}$ variables influence each other in the search but that $\mathcal{N}_\infty$ only depends on the maximal value of μ and *can be statically chosen without any assumption on* λ.
Case 2 : if $n = dim\ (F_{\overline{r}} \cap \alpha_{sf}) - dim\ (\alpha_{sf})$ is smaller than the number of remaining dimensions to schedule, then we have slackness and we bias the remaining dimensions to schedule towards $Ker(F_{\overline{r}} \cap \alpha_{sf})$-orthogonal. In theory this slackness makes the problem easier to solve by giving us more options to extract contiguity. In practice, we need to select n out of the remaining α dimensions. To this effect, we compute a basis of $Ker(F_{\overline{r}} \cap \alpha_{sf})$-orthogonal and we link the contiguity decision variable to the number of dimensions n' along which the current dimension of α has non-zero dot product. If $n' < n$, the contiguity decision variable must be 0.

3.1.4 *Discussion*

The benefit of contiguity then becomes:

$$\sum_{\substack{S \in \mathcal{G} \\ d = d^S}} \sum_{\substack{R \in S \\ r\ =\ dim(F)}} \rho^F_{r,d} . c^F_{r,d}$$

where $\rho^F_{r,d}$ are well-chosen cost coefficients. These coefficients depend on properties of memory accesses (latency, bandwidth, volume of data) and whether we give more precedence to parallelism, contiguity or other constraints. A thorough investigation of the space of those coefficients is a very interesting research topic. The reader may note that our constraints precisely pinpoint the arrays having innermost contiguous accesses[5]. Note that it is straightforward to plug in an upper bound on the number of non-contiguous accesses and minimize this bound thus encompassing an existing formulation on memory streaming prefetches [5].

```
for (i=1; i<=N; i++) {
  for (j=1; j<=N; j++) {
    for (k=1; k<=N; k++) {
      A[i-k][k]=A[i-k][k]+1;
}}}
```

```
for (i=1; i<=N; i++) {
  for (j=-N+1; j<=N-1; j++) {
    for (k=max(-j+1,1);
          k<=min(-j+N, N); k++) {
      A[-j][j+k]=A[-j][j+k]+1;
}}}
```

```
for (i=2; i <= 1+N; i++) {
  for (j=2; j <= 1+M; j++) {
    for (k=1; k <= L; k++) {
      A[k][i][j]=A[k][i][-1+j]+
        A[k][-1+i][j];
      B[1+k][i][j]=A[k][i][j]+
        B[k][i][j];
}}}
```

```
for (i=2; i <= N+1; i++) {
  for (j=2; j <= M+1; j++) {
    for (k=1; k <= L; k++) {
      A[j][i][k]=A[j][i][k-1]+
        A[j][i-1][k];
}}}
for (i=2; i <= M+1; i++) {
  for (j=2; j <= N+1; j++) {
    for (k=1; k <= L; k++) {
      B[i+1][j][k]=A[i][j][k]+
        B[i][j][k];
}}}
```

Figure 2: Contiguity Optimization

This is a good point to introduce two examples to illustrate contiguity. To better understand individual contributions, we completely disable the search for parallelization, permutability and fusion in this first example. The ILP objectives we use are, in their order of importance (1) maximize the contiguity metric, (2) minimize the absolute value of schedule coefficients and (3) bias coefficients towards positive coefficients whenever possible. In particular, we only try to optimize contiguity along the innermost level of schedule and the innermost level of memory. Figure 2 shows the original kernels we consider on the left and the result of optimizing communications on the right.

The top kernel illustrates the optimization of a single array reference. The schedule found is $(j, -i + k, i)$ and the access to A is contiguous along k. We additionally experiment with forcing all coefficients to be positive. In that case, the returned schedule is (j, i, k) and the transformed access function is $A[j - k][k]$. This is an example where restricting schedules to the positive quadrant misses contiguity [6, 17].

The bottom kernel illustrates the tendency for the scheduler to fission loops in the absence of any fusion optimization objective. The schedule found is (i, k, j) for the first statement and (k, i, j) for the second.

3.2 Vectorization

With the availability of contiguity constraints, vectorization can easily be encoded and optimized. The high-level scheduling constraints and properties are the same except for memory alignment.

[5]...the *crucial* objective as mentioned at the beginning of this section.

3.2.1 *Vectorization Constraints*

We already mentioned that the proof of Section 3.1.1 holds for any schedule dimension; the constraints are not difficult to derive. When transposed to vectorization, this simple generalization allows to search for multi-level vectorization as follows.

First we introduce for each statement S and each loop dimension $1 \leq l \leq d^S$ a decision variable to encode whether or not it is vectorizable: Σ_l^S. If a statement is vectorizable along k then it must also be parallel:

$$0 \leq \Sigma_l^S \leq \Delta_l^S$$

We tie the Δ variables together using the β coefficients: if 2 statements are nested under the same outer l loops, their $\Delta_1, ..., \Delta_l$ will be equal. As a consequence, the Σ_l^S for statements in a same loop will also be equal.

The next step is to link together the contiguity constraints of all references within a statement. This is done schedule dimension by schedule dimension. For each tuple $(S,\ 1 \leq l \leq d^S,\ F \in S,\ r = dim(F))$, we write:

$$0 \leq \Sigma_l^S \leq c_{r,l}^F$$

Integrating vectorization along any loop level in our cost function is straightforward:

$$\sum_{\substack{S \in \mathcal{G} \\ l \in [1, d^S]}} \sigma_l^S . \Sigma_l^S$$

where again the σ_l^S are well-chosen costs to model which schedule dimensions of which statements are the most important to vectorize.

3.2.2 *Discussion*

We now replace the contiguity portion of the ILP objective by the vectorization benefit. The reader may verify that the second example of Figure 2 also exhibits innermost parallelism in addition to contiguity. Therefore both its statements are immediately simd-izable and the schedule is the same.

```
for (i=2; i<=1+N; i++) {
  for (j=2; j<=1+M; j++) {
    for (k=1; k<=L; k++) {
      A[i][j][k]=A[i][j-1][k+1]+
        A[i-1][j][k+1];

}}}
```

```
doall (i=5; i<=N+M+L+2; i++) {
 for (j=max(2, i-N-L-1);
      j<=min(M+1, i-3); j++) {
   for (k=max(2, i-j-L);
     k<=min(i-j-1, N+1); k++) {
     A[k][j][i-j-k]=
       A[k][j-1][i-j-k+1]+
       A[k-1][j][i-j-k+1];
}}}
```

Figure 3: Outer Vectorization Optimization

Another example is shown in Figure 3: the kernel on the left becomes outermost vectorizable along i after scheduling with $(i + j + k, j, k)$.

The benefits offered by multi-level vectorization are multiple. First, if the low-level compiler is powerful enough to simd-ize non-innermost, imperfectly nested loops, R-Stream will expose those loops automatically. Secondly, since the vectorizable loops have doall semantics, they can always be strip-mined and sunken to the innermost level. This creates opportunities to exploit multiple threads at the original loop level combined with simd parallelism at the innermost level.

Current limitations of this formulation are twofold. First, the vectorization constraints allow the extraction of parallelism and contiguity along a single schedule dimension at a time, per statement. This dimension may still be different for each statement. This combines well with additional parallelism and locality extraction but not yet with a second level of vectorization. Secondly, the stride along the innermost dimension is not guaranteed to be minimal and packing/unpacking instructions may be necessary for simdization even if a better solution would exist. Last, alignment constraints do not fold directly in this formulation, particularly when the transformed access on the innermost memory dimension is a function of multiple induction variables. This is the case in Figure 3 where the alignment changes at every single iteration of the i and j loop. Recent work by Henretty et al. [11] can be applied to change the layout and is applicable in our case. Addressing these issues will be the subject of future research.

3.3 Joint Vectorization And Data Layout

In this section, we take advantage of the data layout transformation phase in R-Stream to devise even more powerful scheduling algorithms. We show how to add constraints that allow contiguity along *any memory dimension*, and not just the innermost. This makes the problem *less constrained* and the scheduler more flexible thanks to an additional degree of freedom. The output of our optimization problem is a schedule with parallelism indicators augmented with a set of data layout permutations that should be applied to realize the full vectorization. This generalization requires the introduction of new variables:

- for each tuple determined by a statement S, an array A accessed by that statement and an admissible schedule dimension $l \in [1, d^S]$ of reuse, we insert a new decision variable $p_l^{S,A}$. It encodes whether all the references to the given array within the statement are contiguous along schedule dimension l and some unspecified memory dimension,

- for each tuple determined by a statement S, an array A accessed by that statement, an admissible schedule dimension $l \in [1, d^S]$ of reuse and and admissible memory dimension of reuse r, we insert a new decision variable $q_{l,r}^{S,A}$. It encodes whether all references to A within the statement are contiguous along a fixed schedule dimension and a fixed memory dimension.

These clearly represent a lot of variables and special care must be taken with respect to the scalability of the ILP. On the other hand these new variables are auxiliary variables that do not appear in the objective function. We discuss scalability issues in the next section. The constraints are not difficult to write but they involve potentially many variables. First the simd decision variables Σ_l^S are linked to the $p_l^{S,A}$, then the $q_{l,r}^{S,A}$ are linked to the $p_l^{S,A}$:

$$\forall S \in \mathcal{G},\ \forall l \in [1, d^S] \qquad \mathcal{K}_1^S \cdot \Sigma_l^S \leq \sum_{A \in S} p_l^{S,A}$$

$$\forall S \in \mathcal{G},\ \forall l \in [1, d^S],\ \forall A \in S \qquad p_l^{S,A} \leq \sum_{r=1}^{dim\ A} q_{l,r}^{S,A}$$

$$\forall S \in \mathcal{G},\ \forall l \in [1, d^S],\ \forall A \in S,$$
$$F\ accesses\ A \qquad \mathcal{K}_3^{S,A} \cdot q_{l,r}^{S,A} \leq \sum_{F\ acc.\ A} c_{l,r}^{F}$$

$\mathcal{K}_1^S$ is the number of distinct arrays accessed by S. $\mathcal{K}_3^{S,A}$ is the number of distinct references to array A within S.

Figure 4 shows an example of this additional benefit of our formulation. In this case, we only look for vectorization at the innermost schedule level. We allow contiguity along any array dimension to illustrate data layout permutation constraints. We do not show the original kernel and leave it as an exercise for the reader to derive it from the transformed variants.

The code on the left is the result of looking for maximal innermost vectorization without data layout permutation. In this case the computed schedules are (i, j, k) and (j, k, i) respectively.

The code on the right is obtained after turning on data layout permutations in our formulation. Now contiguity is allowed along any memory dimension as long as it is the same dimension in all the accesses to the same array within the statement. In this case the schedules computed are (j, k, i) and (k, i, j). The second statement becomes vectorizable modulo the permutation of rows 2 and 3 of both arrays A and B.

The benefits of this formulation are clear: additional opportunities for vectorization appear that are not available without this new degree of freedom.

```
for (i=2; i<=N+1; i++) {           for (i=2; i<=M+1; i++) {
  for (j=2; j<=M+1; j++) {           for (j=2; j<=N+1; j++) {
    doall (k=1; k<=L; k++) {           doall (k=1; k<=L; k++) {
      A[i][j][k]=A[i][j-1][k]+           A[j][i][k]=A[j][i-1][k]+
        A[i-i][j][k];                      A[j-1][i][k];
}}}                                }}}
for (i=2; i<=M+1; i++) {           for (i=1; i<=L; i++) {
  for (j=1; j<=L; j++) {             for (j=2; j<=N+1; j++) {
    for (k=2; k<=N+1; k++) {           doall (k=2; k<=M+1; k++) {
      B[k][i][j+1]=A[k][i][j]+           B[j][k][i+1]=A[j][k][i]+
        B[k][i][j];                        B[j][k][i];
}}}                                }}}
```

Figure 4: Joint Vectorization and Data Layout

At this point, the expression of data layout permutations is at the granularity of a statement and the simd-ization we find is likely to require the insertion of layout transformation between statements. The kernel on the right is an example of this property: A is written in the first statement and it is read in the second. Because of these dependences, it is not possible to have a single layout transformation before the first statement. It is easy to extend our formulation to increase the granularity of the layout to a set of statements; for instance to all the statements that would end up in the same tile after one pass of coarse-grained scheduling. The prospects for future extensions and experiments are exciting.

4. SCALABILITY EXPERIMENTS

The experiments we conduct focus on statistical results and on the scalability of various scheduling strategies based on our formulation. In Figure 5 we describe the amount of contiguity, vectorization and the compilation times incurred with different strategies.

Default : the base strategy that optimizes in this order for (1) maximal vectorization along the innermost schedule and memory dimensions, (2) maximal outermost parallelism, (3) maximal contiguity for independent arrays, (4) minimal absolute value of schedule coefficient and (5) bias coefficients towards positive values.

NoObj is the base strategy with no objective function. It just picks an integer point in the search space.

Identity additionally forces the identity schedule for all statements and uses the cost function of the *Default* strategy. Beta coefficients are still free so it will only look for the best fusion/distribution structure that maximizes the objective.

Permutations is a little less constrained than *Identity*; it forces the schedules to be combinations of loop interchanges and loop reversals. In this particular case, a single scheduling pass can be performed and we avoid the cost of incrementally searching for invertible schedules.

OuterSimd allows the *Default* strategy to look for the best outermost simd-ization.

Layout allows the *Default* strategy to look for the simd-ization modulo data layout permutations (independently on each statement).

OuterSimdLayout is the conjunction of *OuterSimd* and *Layout.*

AS : the coarse-grained multi-dimensional Affine Scheduling strategy implemented in R-Stream trades off parallelism and locality. It does not contain the formulations of this paper and is shown for reference.

The strategies *NoObj*, *Identity*, *Permutations* are internally used as debugging and as the base cases to incrementally check the overhead of our formulation. Our testbed consist in close to 400 kernel benchmarks whose complexity vary from the simple examples presented in this paper to bigger kernels such as computation intensive radar codes with a few dozen statements. To stress test the formulation, we schedule the whole kernel and do not limit the optimization to well-chosen subgroups of statements (i.e. statements that would belong to the same tiles). Given the complexity of the constraint sets we generate, we implement various independent checks to ensure (1) the schedules and their parallelism are correct with respect to the input dependences, (2) the contiguity decision variables are true if and only if the accesses after code generation are indeed contiguous and (3) simd variables are true if and only if the statement is simdizable (modulo data layout permutations when applicable).

We run the following steps in this order. First we parse the input program and convert it to a GDG, this has negligible overhead. Then we perform exact dependence analysis, scheduling using our multi-dimensional formulation with the different objectives described and, lastly, polyhedral scanning. We report the total time spent in each phase for the whole set of 400 kernel benchmarks and provide additional statistical information to quantify the quality and properties of the schedules we find. R-Stream is run on a Xeon E5520 and we use version 3.1 of the Gurobi ILP solver. The timeout in the solver is set to 15 seconds. For the sake of il-

Strategy	Num contiguous	Num simd	Simd depth	DepAnal (s)	Scheduling (s)	CodeGen(s)	Num T/O
NoObj	-	-	-	16.5	190	14.2	2
Identity	179	23	27	16.7	290	16.7	2
Permutations	673	75	119	17.2	78	17.2	2
Default	1637	386	586	16.5	467	16.6	2
OuterSimd	2891	348	418	16.5	575	16.5	2
Layout	2107	483	772	16.6	466	16.6	2
OuterSimd + Layout	6999	368	244	15.8	796	15.8	3
AS	-	-	-	17.1	90	4	0

Figure 5: Statistical Aggregate Performance of the Scheduler on 400 Examples

lustration of the current performance of our unoptimized formulation we report those timeouts. It can safely be assumed that our formulation is currently less scalable than previous techniques by virtue of a higher number of integer variables and constraints. Scalability will be an important subject of future work where we will also report additional statistical data on the composition of our kernel benchmarks.

The table in Figure 5 displays the aggregated results of running our set of 400 kernel benchmarks. The first column returns the total number of loops that exhibit contiguous memory accesses. The first 4 rows in the table only track contiguity along the innermost schedule and memory dimensions. The number of contiguity dimensions gradually augment as we relax the constraints on the admissible schedules. In the 5^{th}, 6^{th} and 7^{th} row, the scheduler allows contiguity along any schedule dimensions, any memory dimension and any schedule/memory dimension respectively. Comparing rows for *OuterSimd* and *Layout* show there are more opportunities for contiguity in our examples by varying the schedule dimension than the memory dimensions. *OuterSimdLayout* finds significantly more contiguity.

The analysis of the simd columns is interesting. The numbers unsurprisingly augment as we relax the strategies but the numbers suddenly drop in the *OuterSimdLayout* row. This actually reflects a drastic change in the "quality" of the vectorization found. There are quite less simd dimensions found but their depth is also much smaller: vectorization opportunities are found at a substantially coarser grain. It is interesting to note that considering permutations and reversals only seem to offer an interesting tradeoff between compilation time and quality of the vectorization. Lastly, the coarse-grained affine scheduler is rather scalable and succeeds on all the kernels. This hints at a future strategy where we will (1) use *AS* to schedule for coarse-grained parallelism and locality, (2) perform tiling and (3) use our new scheduler for fine-grained and simd parallelism and (4) adjust with potential data layout transformations.

5. RELATED WORK

Vectorization and simd-ization are important topics attacked by industrial compilers [4, 7, 21]. Still the problem of finding good loop transformations to maximize the amount of simd-izable loops does not have a good general solution to this date. Previous work has tackled the efficient generation of simd instructions for already vectorizable loops and has focused on important low level issues such as packing and unpacking data, adapting to various kind of misalignments and devising cost models to decide when a loop should be vectorized or not. Our work is complementary and attacks the difficult problem of loop nest transformations to find proper preconditionings that enables further optimizations to proceed. Since our solution finds multi-level vectorization, it is also a good solution for GPUs which "should be viewed as multi-threaded multi-core vector units" [20]. Our work bridges the gap between the input language and the required performance: it is less critical to program using vector intrinsics or CUDA if a good vectorizer is available.

Another interesting contribution shows the benefits of generating outer-loop vectorization and emphasizes on the fact that outer-loop vectorization provides new reuse opportunities, including efficient handling of alignment [16]. Our work finds these opportunities for outer loop vectorization even when they are deeply hidden in the loop structures and require a complex mix of affine transformations and data layout permutations.

Recent work showed the importance of vectorizing on a high-level representation of the program [12] but in the special case of streaming programing. Our solution also deals with a high-level representation of the program but is not limited to programs expressible in a streaming paradigm. Since the authors mention that "traditional vectorization techniques can also be a viable approach to perform simdization on streaming applications," our work may also generalize theirs.

Within the polyhedral framework, it is known that useful high-level properties of programs can be optimized such as maximal fine-grained parallelism using Feautrier's algorithm [8], maximal coarse-grained parallelism of a fixed loop nest using Lim and Lam's algorithm [15] or maximal parallelism given a (maximal) fusion/distribution structure using Bondhugula et al.'s algorithm [6]. More recent work explored ways to integrate in a single formulation constraints for hardware prefetch stream buffer utilization, locality and parallelism [5]. Recently, a new characterization of the exact convex space of all legal multi-dimensional schedules has been devised [19] and schedulers are starting to appear that use this formulation [17], though not as powerful as the R-Stream implementation described here. Our solution exploits this structured space and extends on previous work by enabling joint optimization of multiple search criteria. These criteria are: parallelism, locality, vectorization and data layout permutations. Our formulation unifies previous work and additionally enables vectorization and data layout transformations in a single complete formulation. Our work is also related to a recent contribution by Trifunovic et al.[18]. They devise a polyhedral loop scheduling strategy and a cost model to drive auto-vectorization. Their scheduling strategy only looks for loop permutations and is weaker than the *Permutations* strategy which also considers loop reversals. Our work introduces a much more general scheduling formulation where arbitrary affine transformations and fusion/distributions are found. Our work formulation also includes data layout transformations. On the other hand, our work does not detail our cost model, this is left for future work.

6. CONCLUSION AND FUTURE WORK

In this paper, we focused on thoroughly describing a new formulation implemented in the R-Stream compiler. We will present detailed performance evaluation using advanced multi-level scheduling strategies in a subsequent paper. We will be interested in exploring the transformation space across benchmarks and across architectures, on both cache-based and explicitly managed memory architectures.

The opportunities opened by our algorithm are significant. In particular, we showed that the formulation of contiguity and vectorization is inherently multidimensional. Therefore, any search procedure that is not based on the convex space of all legal schedules will miss vectorization opportunities. It will be interesting to compare heuristics in advanced compilers and determine how much vectorization is lost.

A very promising aspect of these joint scheduling and data layout transformations is they fit naturally in the current mapping flow of R-Stream. At a high-level glance, R-Stream first performs coarse-grained scheduling and tiling. Tile sizes are selected such that the footprint of the accessed arrays fit into a given memory (scratchpad, cache, ...). Then R-Stream performs fine-grained scheduling to improve the contiguity and simd properties of the code. At this point, R-Stream introduces communications and temporary arrays to store data closer to the processing elements. This feature is available for cache-less memory architecture and as an option on cache based machines by using virtual scratchpads. This is where R-Stream performs data layout transformations that could be very useful for memory alignment [11].

Iterative optimization and machine learning based on our formulation are very likely to have a drastic impact on the generated code. In this context it will be interesting to turn optimizations on or off and explore the values of the different cost coefficients we mentioned.

Lastly, our formulation has interesting implications on power consumption since contiguity and locality are key optimizations to improve the behavior of memory traffic.

7. ACKNOWLEDGMENTS

We are grateful Allen Leung for his successful linking of Bastoul's characterization to the constraints (1)-(3) in the special case when $k = l$ on a single dimensional formulation. We also thank Albert Hartono for preliminary work on reuse constraints on a single dimensional formulation.

8. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures : a dependence-based approach.* Morgan Kaufmann, 2002.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04*, pages 7–16, Juan-les-Pins, Sept. 2004.

[3] C. Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, University Paris 6, Pierre et Marie Curie, France, Dec. 2004.

[4] A. J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance.* Intel Press, 2004.

[5] U. Bondhugula, O. Günlük, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *PACT*, pages 343–352, June 2010.

[6] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, Tucson, Arizona, June 2008.

[7] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 82–93, New York, NY, USA, June 2004. ACM.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, Dec. 1992.

[9] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[10] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *PPoPP'07*, pages 237–248, New York, NY, USA, June 2007. ACM Press.

[11] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short SIMD architectures. In *CC'11*, Saarbrücken, Germany, Mar. 2011. Springer Verlag.

[12] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. M. Rabbah, T. N. Mudge, and S. A. Mahlke. Macross: macro-simdization of streaming applications. In *ASPLOS*, pages 285–296, Mar. 2010.

[13] K. Kennedy and J. R. A. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[14] A. K. Leung, N. T. Vasilache, B. Meister, and R. A. Lethin. Methods and apparatus for joint parallelism and locality optimization in source code compilation, Sept. 2009.

[15] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *PoPL'97*, pages 201–214, Paris, France, Jan. 1997.

[16] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT 08*, pages 2–11, New York, NY, USA, Sept. 2008. ACM.

[17] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*

(POPL'11), Austin, TX, Jan. 2011.

[18] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, pages 327–337, Sept. 2009.

[19] N. Vasilache. *Scalable Program Optimization Techniques In the Polyhedral Model*. PhD thesis, University of Paris-Sud, Sept. 2007.

[20] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08*, pages 1–11, Piscataway, NJ, USA, Nov. 2008. IEEE Press.

[21] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *CGO '05*, pages 153–164, Washington, DC, USA, Mar. 2005. IEEE Computer Society.