Memory Reuse Optimizations in the R-Stream Compiler

Nicolas Vasilache, Muthu Baskaran, Benoit Meister, Richard Lethin
Reservoir Labs Inc.
632 Broadway
New-York, NY, USA
vasilache,baskaran,meister,lethin@reservoir.com

ABSTRACT
We propose a new set of automated techniques to optimize memory reuse in programs with explicitly managed memory. Our techniques are inspired by hand-tuned seismic kernels on GPUs. The solutions we develop reduce the cost of transferring data across multiple memories with different bandwidth, latency and addressability properties. They result in reduction of communication volumes from main memory and faster execution speeds, comparable to hand-tuned implementations, for out-of-place stencils. We discuss various steps of our source-to-source compiler infrastructure and focus on specific optimizations which comprise: flexible generation of different granularities of communications with respect to computations, reduction of redundant transfers, reuse of data across processing elements using a globally addressable local memory and reuse of data within the same processing elements using a local private memory. The models of memory we consider in our techniques support the GPU model with device, shared and register memories. The techniques we derive are generally applicable and their formulation within our compiler can be extended to other types of architectures.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors – Code Generation, Compilers, Optimization

General Terms
Algorithms, Design, Performance

Keywords
Parallelization, Compiler Optimziation, Polyhedral Model, GPGPU, CUDA, Automatic Translation

1. INTRODUCTION
With the availability of multi-node, multi-socket architectures and accelerators, the programming trends evolve towards more explicitly managed memories. Addressing remote data is increasingly more expensive than addressing local data. Even when a partitioned global address space or a machine-wide memory coherence mechanism is available, it is likely that the programmer and compiler will have to manage data localization – at least for performance reasons. R-Stream is a high-level compiler that aims at solving this problem [20]. It provides an automatic source-to-source mapping pathway from a high-level textbook-style code expressed in ANSI C to target-specific source code. It also provides a programmable IR interface and many options for user-in-the-loop interactions. Although R-Stream is architecture agnostic and has been ported to different targets, for the purpose of demonstration, we build on our capabilities to generate CUDA code that we have previously described [16] and augment them with various communication optimization strategies. The problem of finding a good mapping of computations to processing elements is an important topic that our compiler handles. In R-Stream, for a single GPU, this problem comprises the following automatic strategies organized as compilation phases:

1. exact dependence analysis and coarse-grained parallelization based on a state-of-the-art scheduling algorithms [17, 29].

2. creation of groups of statements that determine the tile shape as well as the allocation and lifespan of local arrays (see section 2.1),

3. loop tiling with sizes based on counting integer points in parametric polyhedra [21], which guarantees that the memory footprint of a task fits within a fixed limit,

4. management of temporary storage and generation of indexing functions (see section 2.1),

5. communication generation from device memory to shared memory for non-streaming arrays (see section 2.2),

6. implicit placement of code along blockIdx and threadIdx dimensions (see section 3.1.1),

7. explicit embedding of blockIdx and threadIdx dimensions into the code,

8. privatization of shared memory locations in registers,

9. synchronization generation.
The description of the last three steps can be found in a previous contribution [16]. In this paper, we focus on source-to-source optimizations targeted at improving the performance of communications. For each of the following three optimizations in our contribution, we describe the model of memory addressability targeted and we detail the properties of the data transfers we optimize: (1) varying communication granularity (a new Step 5b), (2) removing redundant communications (a new Step 6b), and (3) an algorithm to improve memory reuse, applied as 2 new steps (Step 6c and 7b) during the optimization process. Our techniques are fully implemented in our automatic compiler toolchain, within the polyhedral model [9], and take into account placement of communication and computation code across multiple processing elements.

2. R-Stream MAPPING OVERVIEW

This section succinctly describes Steps 2, 4 and 5 of the mapping process. The detailed description and the optimization problems we construct are non-trivial and are left for future publication. Here we only describe them by the way of examples, to properly introduce our contribution.

2.1 Local Memory Management

R-Stream supports targets with explicitly managed memories by allocating regions in local memory to store temporary data. When data is migrated from a memory to another, opportunities arise to restructure the data layout at a reduced relative cost. Such reorderings help reduce storage utilization and can enable further optimizations (e.g. simdization). Our solution is based on the work of Schreiber [28] and generalizes it by (1) accepting arbitrary parametric iteration domains, (2) supporting strided domains via lattice extraction and non-convex footprints, (3) utilizing index-set splitting in novel ways. Most notably, our solution differs from previous contributions based on modular reindexings [15, 23] which have been generalized by Darte et al. [6]. The main insight that motivates our current avoidance of modular reindexings is they overconstrain schedule and distribution opportunities within the tiles. Modular indexing introduces memory-based dependencies in order to reduce the memory consumption as well as moduli in indexing functions. As a result, this approach destroys opportunities for further optimizations within the tile. While we believe it is possible to reconcile both techniques, this is left for future work and is not the topic of this contribution.

Our formulation is as follows: given a set of loop nests with parametric affine array references \(A_1[f_1(x)], \ldots, A_n[f_n(x)]\), the result of local memory compaction is a mapping of these global references to new local references: \(A_1[f_1(x)] \mapsto A'_1[g_1(x)]\) where \(A'_1\) are new arrays to be allocated in the local memory of the remote processor. To perform local memory management, we need to define the notion of a data footprint. For each array \(A\), we are given a set of parametric affine references of the form: \(A[f_k(x,y)]\), \(x \in D_k(y)\) where \(f_k(x,y)\) is a parametric affine function and \(D_k(y)\) is a parametric Z-Domain [25] that represents the iteration domain of the statement accessing reference \(A\). For each array referenced in a statement group (see section 2.1.1), we define the parametric data footprint of a tile \(R\) as the projection of the domain image by the access functions \(f_k\):

\[
R(y) = \bigcup_k \{ f_k(x,y) \mid x \in D_k(y) \}
\]

Anticipating on the next section, \(x\) represent varying intratile dimensions and \(y\) comprise fixed intertile dimensions and global parameters (see section 2.1.1). We look for affine access functions \(g_i\) (i.e. non-modular). Our algorithm contains the following three basic steps: (1) partition the set of references so that related references are grouped and allocated together; (2) for each group of references decided in Step (1), perform algebraic simplification via Hermite decomposition and; (3) for each group of references, perform geometric rearrangement via unimodular reindexing.

2.1.1 Tiling and Statement Grouping

Tiling (a.k.a blocking) [14] is the process by which coarse-grained tasks are formed once permutable loops have been exhibited [1] by our affine scheduling algorithm. In our work, a tile correspond to an atomic unit of execution. Loops iterating over (resp. within) tasks are called intertile or ITD (resp. intratile or itd) loops. Statement grouping is a heuristic to partition statements that can be tiled together to fit within a constrained memory space [21]. Such a group forms an atomic unit of memory allocation. Our algorithm allocates different arrays independently. Nonoverlapping references to the same array are placed into distinct allocation groups. For instance, in the following pseudo-code—which represents a 16x16 tile—the footprints are disjoint and three local arrays \(A_2, A_3\) and \(A_4\) are generated.

```plaintext
float A[256][256];
doall (i=128*j+16*p; l <= min(-i+254,128*j+16*p+15); l++)
doall (m = 16*k; n <= min(-i+254, 16*k+15); n++)
A[i+i+1][j+i+1] = A[i-i-1][j-i-1];
// will be transformed to:
```

References to the same array are generally allocated together if they overlap; unless the footprint union is larger than the sum of the footprints. In the example below, the footprint of the union is a cross whose bounding box takes significantly more space than 2, one-dimensional arrays.

```plaintext
double A[100][100];
for (j = 0; j < 100; j++) {
    .. = A[j][j] * A[j][j];
}
```

A heuristic based on the above principles partitions memory references in different groups. These partitions influence the chosen tile sizes and vice-versa. The description of this work is outside the scope of this paper.

2.1.2 Hermite Normal Form Decomposition

The purpose of decomposition in Hermite Normal Form (HNF) is to reduce the dimension of the reference \(g\) to the actual geometric dimension of the data footprint. In addition, if the access pattern contains strides, they are removed in the resulting local references. Given an affine function \(f(x,y)\) on loop indices \(x\) and parameters \(y\), we first decompose it into the sum of \(g(x) + h(y)\), where \(g(x)\) is a linear function on \(x\) and \(h(y)\) is an affine function on \(y\). Function \(g(x)\) can be decomposed into \(g(x) = HU\), where \(H = [H']^T\) is the Hermite normal form of \(g\), and \(U\) is a unimodular matrix. Let \(U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}\) where \(HU = H'U_1\). We can then generate the following mapping from global to local indices:
\( f(x, y) \rightarrow U_{1, x} \). For example, suppose we are given the following loop nest:

```c
for (i=0; i<9; i=+2)
    // Kernel starts here.
    for (j=max(-1, -9+i); j<min(4, 3+i); j=+2)
        // Kernel starts here.
            for (k=max(1, 1, 1-3); k<min(6+j-1, 14+i, 9); k=+4)
                for (l=max(-i+j+k, 1); l<min(4, 1-i+j+k); l=+4)
                    \( A[l][2*l] = \ldots \)
                        // Kernel ends here.
```

\[ H = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 \end{bmatrix} \cdot (i \ j \ k \ l)^T \]

and the following compact global to local mapping is obtained: \( [k, 2^i] \rightarrow [k-i-1, l-j+1] \). This method generalizes to non-full rank mappings and is complemented by unimodular reindexing.

\subsection*{2.1.3 Unimodular reindexing}

The final step reorients a skewed data footprint so that the resulting access pattern is more rectangular. The problem can be approximated by a maximization problem. Given an array \( A \) in \( d \) dimensions, a set of parameters \( y = [y_1, \ldots, y_d] \) and a parametric data footprint set \( D(y) \) for a group of references to array \( A \), we want to find: the dimensions of the local array \( S \); \( S \in \mathbb{Z}^d \) as a bounding-box, an affine function \( L : Z^d \rightarrow Z^d \), which represents the lower bound of the local array as a function of the parameters \( y \), and a non-singular matrix \( U : \mathbb{Z}^{d \times d} \), which represents the skewing on the original footprint:

\[
\min \prod S_i \left\{ \begin{array}{l}
L(y) \leq Ux < L(y) + S, \forall x \in D(y) \\
S : Z^d \\
L : Z^d \rightarrow Z^d \\
U \text{ non-singular}
\end{array} \right. 
\]

This optimization problem is non linear since it involves finding a non-singular matrix and a non-linear cost function. However, with a few clever formulation tricks, we can create multiple small integer linear programming problems incremented with dimension-by-dimension constraints. The description of these techniques and multiple other considerations are left for future publication.

\subsection*{2.2 Communication Generation}

In Step 5, explicit memory copies are automatically inserted between memory regions for data footprints derived at the granularity of a tile. Optimizations such as bulk DMA generation and pipelining communications with computations can be obtained and have been described previously [19]. Figure 1 illustrates the result of such a communication generation step on a 1024x1024 matrix multiplication kernel. In this context, statement \( S_1 \) is the 2-D initialization of array \( C_1 \) to 0 and is tiled by 128x128 (since \( C_1 \) is only written to, it is not brought into local memory, just written directly to main memory). Statement \( S_2 \) is the 3-D computation which requires communications from main memory for \( A, B \) and \( C \) but only write-back of \( C, S_2 \) is tiled by 128x128x128. Notice the seemingly unnecessary copies to \( C_1 \) denoted by copy statements \( C_1 \) and \( C_2 \). Despite \( S_1 \) and \( S_2 \) belonging to the same statement group, redundant copies are introduced precisely because the generation of communication is performed at the granularity of a tile. Even though memory promotion and array allocation are performed at the granularity of a statement group, communications at the level of a tile are necessary because a tile is the atomic unit of execution and code distribution across multiple processing elements may happen.

```
1 doall (i = 0; i <= 7; i++) {
    doall (j = 0; j <= 7; j++) {
        doall (k = 0; k <= 127; k++) {
            doall (l = 0; l <= 127; l++) {
                C[128*j+k][128*i+l] = C_1[l][1];
            }
        }
    }
}
2 doall (m = 0; m <= 127; m++) {
    doall (n = 0; n <= 127; n++) {
        C_1[l][m] = C_1[l][m];
    }
}
3 doall (i = 0; i <= 7; i++) {
    doall (j = 0; j <= 7; j++) {
        doall (k = 0; k <= 127; k++) {
            doall (l = 0; l <= 127; l++) {
                C_1[l][m] = C_1[l][m];
            }
        }
    }
}
```

\section*{Figure 1: Redundant matmul copies (\( C_1 \) and \( C_2 \))}

Looking at Figure 1, suppose \( S_1 \) were entirely mapped to a thread 0 and \( S_2 \) were entirely mapped to a thread 1. In the absence of extra copies \( C_1 \) and \( C_2 \) to/from main memory the code as written would be incorrect. If both threads could address each other’s local memory, it would be possible to rewrite the code without communications through main memory. But if the \( C_1 \) arrays have threadprivate semantics or are stored in private registers on a GPU, communications must occur.

**Automatically optimizing communications on a parallel architecture requires a model of memory addressability as well as a model of code distribution.**

This is a fundamental difference with a recent contribution by Plesco et al. which automatically offloads computations to accelerators given a predefined schedule and tiling specification [5]. In their work, remote accesses and communications are optimized thanks to an integer linear programming approach that computes communications with reduced lifetimes to begin with. As a consequence, they rightfully claim inter-tile reuse benefits. However, there are multiple caveats to this approach: (1) first, a tradeoff exists: the innermost parallel ITD is sacrificed to obtain reuse. In a CUDA mapping strategy, this would mean either not performing optimizations or losing a parallel blockIdx level which may be unacceptable (for instance on the Heat 2-D stencil in Figure 4), (2) second, the reuse they obtain is only achievable if values from main memory are stored at a fixed location in the accelerator memory. In other words, they do not exploit movement of data within the accelerator memory, which is a purpose of our contribution, (3) their allocation scheme is based on modular reindexings. As discussed in section 2.1, this has adverse implications on the quality of the code within the tile and may result in losing parallelism due to the introduction of false dependences.

In contrast, our optimizations take into account a specification of schedule, tiling and placement on a processor grid. They are not limited to the innermost ITD but only work with optimized communications.
are more generally related to loop carried dependences [1] and their extension to processor carried dependences. Additionally, our approach handles more complex reuse patterns where the location of values in the accelerator memory may vary (see sections 3.4 and 3.5) and does not impose any new dependence on the code within the tiles.

3. COMMUNICATION OPTIMIZATIONS

We now discuss the technical contributions of this paper. For the purpose of illustration, we will use code examples with static constant bounds. R-Stream handles general, parametric code conformant to SCoP specifications [3] but the complexity of the generated CUDA code would obscure the presentation. Additional limitations of automatic compilation to CUDA are discussed in section 3.7.

3.1 Prerequisites

3.1.1 Model of Code Distribution

R-Stream features a code distribution mechanism that associates affine functions of loop induction variables to processor dimensions. These affine functions are called placement functions and depend on a high-level model of the machine hierarchy. Given a regular p-dimensional grid of processing elements, each statement Si in the program (including communication statements introduced in section 2.2) is associated to a p-dimensional modular affine placement function $\Pi_S \mod m_i$ over the loop iterators of Si. When statement Si has d enclosing loops (i.e., the iteration domain $D_S$ is of dimension d), $\Pi \in Z^{d \times d}$ and $m \in N^d$. The semantics follow a block cyclic distribution of loop iterations, under transformation by $\Pi_S$, to physical processors, of multiplicity $m_i$, $i \in [1,d]$ along each dimension.

Consider a decomposition of the transformed iteration domain $\Pi_S \odot D_S$ as a Cartesian product:

$$\bigotimes_{x \leq p} [l_i, u_i] = \Pi_S \odot D_S$$

For each such dimension $[l_i, u_i]$, we form its regular subdivision in $m_i$ chunks, each of size at most $\frac{u_i - l_i}{m_i}$. Given an integer $j \in [l_i, u_i]$, we denote by $k_i = \text{subd}(j, l_i, u_i, m_i)$ the $\lfloor \frac{u_i - l_i}{m_i} \rfloor$-length subdivision to which it belongs. In other words, $k_i = \text{subd}(j, l_i, u_i, m_i)$ is the unique integer such that:

$$j \in \left[ l_i + k_i \cdot \left( \frac{u_i - l_i}{m_i} \right), l_i + (k_i + 1) \cdot \left( \frac{u_i - l_i}{m_i} \right) \right]$$

We can now write the expression determining on which processor $P(i)$ in the processor grid, an iteration $i$ of $S$ is executed:

$$\forall i \in D_S, \text{ let } \Pi \odot D_S = \bigotimes_{x \leq p} [l_x, u_x], \text{ } P(i) = \bigotimes_{x \leq p} \text{subd}(i_x, l_x, u_x, m_x)$$

innermost ITD by a constant amount and make it the outermost ITD thus turning one problem into the other.

Each $l_i$ and $u_i$ for $i > 1$ is an arbitrarily complex affine expression involving only dimensions $[1, i - 1]$. These expressions are implicit in our representation. To obtain the expression explicitly, one would use an iterative projection and simplification algorithm such as is implemented in CLooG [2].

The decomposition of domain iterations onto processors uses a modified stripmine and interchange transformation [11]. There are essentially two ways of specifying placement information. Implicit Virtual: in this mode, R-Stream maintains for each statement $S$, a processor placement matrix $\Pi_S$ and a multiplicity vector $m_S$ along each dimension. In addition, an abstract processor vector describes for each dimension of the processor grid, an abstract description of the processing elements; this is useful for determining the memory models available for optimizations. Most optimizations in this paper occur in the implicit virtual placement mode and do not account for the multiplicity values $m_i$.

Explicit Physical: in this mode, R-Stream embeds the information contained in the placement matrix as well as the multiplicities directly into the code by using placement parameters (Step 8). These parameters’ behavior resembles traditional “global parameters” used in previous work [11] However the interpretation of the parameters is different. Whereas the semantic meaning of a “traditional” global parameter $1 \leq N \leq 100$ is that $N$ takes a unique arbitrary value between 1 and 100, the semantic meaning of a placement parameter $0 \leq P_N \leq 15$ is that $P_N$ takes all possible values in the range $[0, 15]$. Once explicit parameters are embedded, all dependences necessary for communication optimization become processor-specific. In particular, it becomes impossible to represent dependences across multiple processors and all inter-processor optimizations must have already been performed. In R-Stream, explicit physical placement is a prerequisite for privatization (Step 8). As a consequence, once privatization is performed, optimization of communications across processors is unavailable. Figure 10 illustrates a simple (16 x 40)-loop which performs copies from device memory ($U2\_S$) to shared memory ($U2\_F$). The placement functions on $[th.x, th.y]$ are

$$\Pi_S([th.x, th.y]) = (j, k) \mod [8, 32]$$

The optimizations of section 3.5 occur in this explicit physical placement mode.

Non-trivial considerations are involved in the determination of the placement function legality and profitability. The interested reader may refer to previous work by Feautrier [10]. In our framework, we impose a well-formedness constraint on placement functions. Statements (imperfectly)-nested under the same loops must have conformable processor placement functions. For the remainder of the paper, we take the convention that statements have the same placement function along the loop dimensions under which they are commonly nested. For example in Figure 1, assuming a 2-D processor grid, the placement functions $\Pi_{C_{ij}} = (i + j, l)$ and $\Pi_{C_{lm}} = (i + j + m)$ are valid. Placement functions $\Pi_{C_{ij}} = (i, l)$ and $\Pi_{C_{ij}} = (j, m)$ however are not: since both statements are nested under $i$ and $j$, they must have the same placement functions components on both $i$ and $j$.

3.1.2 Models of Memory

In order to optimize communications, R-Stream uses a high-level model of memory. The original data is assumed to reside on a first “remote” memory $M_1$ and is moved to a second “local” memory $M_2$. In general, $M_2$ is considered closer to the processing elements than $M_1$ (i.e., it has higher bandwidth and lower latency) and also has smaller capacity. Communications are generated from a target processors-
centric” view, i.e., each processor “pulls” (i.e. “receives”) the data it needs before performing computations and “pushing” (i.e. “sending”) the modified data back to remote memory. An implicit notion of temporary ownership of data underlies this model. Correctness is guaranteed by the validity of the schedules computed in Step 1. Non-trivial implications can occur when targeting hardware mechanisms for bulk-communications and require additional synchronizations for correctness. This is not the case for GPUs for which this issue translates into a performance penalty. Lastly, each memory may be associated exclusively to a processing element or shared by multiple elements. Depending on this association, a processing element may or may not address the memory read/written by another. Implications arise on the type of transfers and available optimizations. Non-exhaustive examples of models of memory supported by R-Stream include: OpenMP + threadprivate buffers: in this context, \( M_1 \) is the DRAM and \( M_2 \) is a threadprivate region of memory that fits into some level of cache (usually L2). In this context, copying data on \( M_2 \) is not mandatory and may help for cache conflict issues, especially if coupled with a cache-conscious memory allocation scheme [18].

OpenMP + multiple accelerators: \( M_1 \) is the DRAM and \( M_2 \) is the device memory on each GPU. Transfers are mandatory from the host CPU to each GPU for correctness.

Single GPU + shared memory: \( M_1 \) (resp. \( M_2 \)) represents the device (resp. shared) memory. Transfers improve performance of non-coalesced global accesses and code with data reuse.

Single GPU + registers: \( M_1 \) represents the shared memory, \( M_2 \) represents the private memory (registers). Explicitly reusing registers usually improves performance.

The example discussed in Figure 11 is a good illustration to CUDA of non-trivial optimizations that combine different memory models.

3.2 Hoisting and Removing Redundant Communications

Communication hoisting and removal factors out loop invariant transfers that result from creating communications at the granularity of a tile (see section 2.2). For example, the transfer of an on the leftmost kernel is invariant in loop \( j \), as long as \( j \) is not used in the placement of \( "A_1 = A[j]" \) (and by extension of \( S \), because of the rule for conformance placements, see section 3.1.1). The candidate loop above which to hoist in our example is \( j \). Its level is denoted by \( m \).

```c
for i,j,k
    for l,m
        A_1[l+1][j] = A[l]
        for i,j,k
            for i,j,k
            for i,j,k
        for i,j,k
    for i,j,k
for i,j,k
    for l,m
        A_1[l+1][j] = A[l]
        for i,j,k
            for i,j,k
            for i,j,k
        for i,j,k
```

In the pseudo-codes above, \( A \) is considered to be the remote array and \( A_t \) the local array. They illustrate cases where the footprint varies: (1) an anti-dependence \( \{ C_1 \rightarrow S_1 \} \) may modify values of the remote array read into the local array and thus prevent the hoisting of \( C_1 \); (2) in a symmetrical fashion, a true dependence \( \{ S_2 \rightarrow C_2 \} \) may modify the local array and \( C_2 \) may not be hoisted; (3) lastly, the conjunction of a true and output dependence \( \{ C_1 \rightarrow C_2 \} \) may shift data values in the local memory; even in the absence of any dependence based on non-copy statements. In our example, data read from \( A[l] \) is copied to \( A_1[l+1] \) then back into \( A[l+1] \). We use dependence analysis based on the polyhedral model [8] to test these cases. Our algorithm uses a simple pruning that incrementally tests for dependences, with equalities enforced on the placement dimensions. We devise a straightforward iterative algorithm that iterates over the copy statements. The algorithm incrementally tests each statement for the conditions previously described and performs hoisting when allowed until a fixed point is reached. Applying this algorithm on Figure 1 yields the code in Figure 2.

```c
for i,j,k
    doall (i = 0; i <= 7; i++) {
        doall (j = 0; j <= 7; j++) {
            for k = 0; k <= 127; k++) {
                // Statement 1
                C_128[j+k][128*i+l] = C[l][k][i];
                // Send 1
            }
            doall (m = 0; m <= 127; m++) {
                // Receive 2
                B[l][m] = B_128[k][128*i+1];
            }
            // Receive 3
        }
        // Statement 2
        S_2[0<=l<=127, 0<=m<=127, 0<=n<=127];
        // Send 2
    }
}
```

Figure 2: Matmul with hoisted copies

Note that we do not address the problem of preconditioning for minimizing communications. If we applied hoisting to the program in Figure 3, only the copies from/to the C array would be hoisted. Additional opportunities for hoisting
A and B exist but they depend on different itd schedules: if the n (resp. l) loop was interchanged under n the copy to B (resp. A) would become hoistable. This scheduling concern is out of the scope of this paper.

Since local memory lifespan has the granularity of state-groups and actual communications have the granularity of memory channels, redundant copies may be introduced under proper placement functions. It is easy to construct placement functions under which the elimination is illegal. R-Stream ensures the legality of the removal but does not otherwise precondition schedules and placements for better redundant copy elimination.

3.3 Refining Communications Granularity

By default the local memory management and communications are determined for each statement assuming the granularity of communications is exactly determined by the number of ITD. We extend this behavior by allowing R-Stream to sink the communications at finer levels within the computations. This mechanism can be controlled by a command line arguments with a memory sinking parameter. Eventually, the combination of ITD and this parameter yield for each communication its memory channel dimension. We thus extend the properties of communications described in section 2.2: the generation of communication is now performed at the level of a memory channel. When the memory channel is strictly greater than the last ITD, this results in the generation of finer-grained communications inter-leaved more closely with the computations. In turn, this shifts the balance between computations and communications. In the particular case of GPUs, the overlap of computations and communications is done by the (over-subscribed) hardware. Simply varying the granularity of the communications is then enough to generate codes with different communication-to-computation ratio and one does not need to devise complex software multi-buffering schemes.

This refining of communication granularities enables further memory reuse strategies described in sections 3.4 and 3.5.

The example in Figure 3 illustrates the effects of setting the sinking parameter to the maximal value (in this example 3), assuming redundant copies have previously been eliminated under proper placement functions.

3.4 Reuse of Addressable Remote Memory

The first kind of memory reuse we optimize assumes the following model of memory and communication: the data originally lives in a remote, globally addressable, memory and it is copied in a closer, partially addressable local memory. This model of memory corresponds to device and shared memories on GPUs. The device memory is globally addressable by all threads and all threadgroups. The shared memory is partially addressable: all threads within a single threadgroup address the same region in shared memory but cannot communicate across threadgroups directly. Each concurrent thread “gets” data from the remote memory and “puts” it into the local memory. Later, data needed for a computation may already be present in the local memory. At that time, each thread may get data from the local memory rather than from the remote memory.

![Figure 3: Matmul with innermost copies](image)

Our contribution considers reuse distances of length \( \leq 1 \). Greater reuse distances require the introduction of extra local storage that we do not yet handle. Simply put, our approach consists in exploiting a subset of the Read-After-Read (RAR) reuse dependences of the program to replace distant communications by shorter ones.

We start with a motivating example: a single time iteration of a Heat 2-D stencil illustrated in Figure 4. The code on the left represents the input code, the code on the right is the code after tiling by \( [128, 16] \) and communication generation with a memory channel sunken to 3 (instead of 2, the number of itd). Communication hoisting does not change the code because all remote footprints depend on k. There are no redundant copies as defined in section 3.2: even though there is an overlap on loop l for the data values copied to \( A_1 \) and \( A_2 \), their addresses in the local memories are different. Direct redundant communication elimination techniques such as proposed in section 3.2 or by Plesco et al. [5] cannot take advantage of the available reuse.

We consider opportunities for reuse which arise on (1)
Figure 5: Optimized Copy into A_\perp 3

As shown in Figure 5, the optimized version, 15 out of 16 writes into A_\perp 3 are now read directly from the local memory. However, computing the reuse domains may yield much more complex code since they are computed by arbitrarily complex parametric ZDomain projections and differences. We implemented a heuristic that only allows a 4-fold split of communications to reduce prohibitive control flow overhead (see a discussion of limitations in section 3.7).

Following the same logic, a reuse opportunity exists when exploiting loop-carried reuse of distance 1 on the following reuse iteration set:

$$RS = \{ A[k + 1][16 \cdot j + l] \rightarrow A[k' + 2][16 \cdot j' + l' + 2], \quad k = k' + 1 \land \Pi C_4 = \Pi C_2 \land \Delta_{RAR}^{C_1 \rightarrow C_2}\}$$

where $\Delta_{RAR}^{C_1 \rightarrow C_2}$ is the subset of $D_{C_1} \times D_{C_2}$ on which the RAR dependence is defined.

We now discuss our algorithm for communication optimization shown in Figure 6. This algorithm optimizes statement groups independently. It starts by selecting a maximal consecutive set of copy statements (in lexicographic order) that read data from the remote memory and write into the local memory. These copies are parallel and free of self-dependencies by construction. The copies are sorted according to their footprint in the remote memory. Sorting is performed by polyhedral comparisons in the same spirit as the sorting step during polyhedral code generation [2, 24]. Sorting orders RAR dependences and improves the propagation of dataflow values.

An illustration is given in Figure 7: without sorting, the copy $a_3[i] = a_1[i]$ is illegal without additional temporary storage (left and center kernels). The code on the right shows better loop-independent reuse with sorting and without the need for extra storage. This property also extends to loop-carried reuse.

Figure 6: Memory Reuse Optimization Algorithm

```plaintext
for(i=0;i<N;i++) {   for(i=0;i<N;i++) {   for(i=0;i<N;i++) {
  a[i] = A[i]      if (i == 0) {   if (i == 0) {   if (i == 0) {
    ...    ...    ...
  }                     }                     }
}                           }                           }
```

Figure 7: Benefits of Sorting by Footprint (right)
ponents must be used during constraints building [11].

Continuing to Step 6, our algorithm further constrains the RAR dependence by memory addressability constraints. These constraints force the equality of the placement dimensions of S and T for processing elements that do not see the same local memory. For instance, in CUDA, threadblocks do not use the same portion of the shared memory whereas threads within a same threadblock do. Addressability constraints are specific to a particular memory and a particular processing element in the machine model hierarchy. On a single GPU, for threadblocks, we generate equalities between the placement functions of the statements if the memory considered is not the global memory. For threads, equalities are generated only if the considered memory is the private memory. If placements are incompatible between copies, the candidate dependence is pruned.

At this point, if the dependence is empty, the original domain of S is added along with S to the unoptimized set. The loop in Step 3, then proceeds. If the dependence is not empty then an optimized transfer will be generated. Steps 11, and 12, will be discussed section.

Steps 13-15, form the domains of the optimized and unoptimized transfers using ZDomain differences. The unoptimized copies are the complement of the optimized copies in the original domain of statement S. Statement T provides the optimized variable and access function.

Steps 16-20, remove all copies and recreate new statements. These new statements are deep clones of the original copies where only the domain and the optimized read are updated. The insertion Step 21, uses a tree structure to represent the beta coordinates (which correspond to a specification of loop nesting in the polyhedral representation [11]) and automatically updates the new beta vector for the inserted statements.

3.5 Reuse of Non-Addressable Memory

The second kind of reuse we optimize assumes the following model of memory: the data originally lives in a remote, globally addressable, memory and is copied in a closer, private local memory. Each concurrent thread “pulls” data from remote memory and stores it into its private memory. This model suits the private memory on GPUs from the point of view of threads in a threadblock and also the shared memory from the point of view of different values of threadIdx.x. Later, data needed for a computation may already be present in the private memory of a thread. Two cases occur depending on which thread requires the data.

First, a thread T1 reads some data at iteration i1. Later in the program execution, at iteration i2 ∈ {i1, i1 + 1}, T1 reads the same data. In that case, our algorithm handles the optimization in the same way as we previously described. In fact, the example of Figure 5 also illustrates the case when (1) A⊙ is declared “private” in CUDA and (2) loop i is not used for placement. In this case, our optimization reduces to register reuse thanks to register rotation (see Figure 7 and assume a[i], a[i] and a[i] are privatized to registers).

Alternatively, a thread T1 may read some data at iteration i1. Later in program execution, at iterations in {i1, i1 + 1}, another thread T2 may read the same data. This case triggers Step 12 of our algorithm.

To illustrate this latter case, we use an example kernel that represents a 256x256 2-D stencil computing the solution to the Discretized Wave Equation (DWE). Figure 8 shows the code produced by R-Stream for CUDA where one placement dimension is used for threadblocks with multiplicity 16 (bl.x) and another one for threads with multiplicity 16 (th.x). Explicit placement is applied (see section 3.1.1). Reuse optimization is turned off for the shared memory level and U2l ⊙ j . . . U2l ⊙ 0 are stored in private memory during the privatization step. U2 resides in device memory which is globally addressable by all threads and threadblocks. U2 ⊙ j resides in shared memory which is addressable by all threads within a threadblock.

for (i = 0; i <= 255; i++) {
... U2_l_1 = U2[3 + i][4 + 16 * bl.x + th.x];
U2_l_2 = U2[5 + i][4 + 16 * bl.x + th.x];
... doall (j = 0; j <= (- th.x + 23) / 16; j++) {
    U2_l[16*j+th.x] = U2[4+i][16*j+16*bl.x+th.x];
    _syncthreads();
}
... _syncthreads();
}

Figure 8: 2-D Discretized Wave Equation

Reuse opportunities exist between C1 and C2 through a loop-independent RAR dependence on the following reuse iteration set, across different values of threadIdx.x:

\[ RS = \{ \begin{array}{c}
    U2[4+i][4+16 * bl.x + th.x] \\
    U2[4+i][16*j+16 * bl.x + th.x + Y],
\end{array} \]

\[ i = i' \land j = 0 \land 0 \leq bl.x \leq 15 \land 0 \leq th.x \leq 15 \land 0 \leq Y \leq 15 \Delta C_{RAR} C_1 \}

Notice the constraint \( \Pi C_2 = \Pi C_1 \) does not appear in RS anymore since placement information has been explicitly embedded in the form of processor parameters. The RAR dependence in this case exists across threads, i.e. data residing in a thread’s private memory can be potentially reused by another thread through a store to a memory that is addressable by both threads. To exploit the reuse, the store to the partially addressable memory must be performed by the thread owning the private memory that holds the correct data value. Hence the optimized access to the globally addressable memory (access to U2 ⊙ j in this case) must be shifted by a proper amount. We explained in section 3.1.1 that processor parameters resemble traditional global parameters [11] but have a different semantic: they take all possible values in the range. However, once placement parameters are explicitly embedded in the code, all statements see the same global parameter “th.x” and it becomes impossible to distinguish between different processors in the reuse relation. We thus introduce a new temporary parameter \( Y \). If a static constant value is imposed on \( Y \) by the reuse then we exploit it (e.g. in this case val(Y) = 4). Otherwise we ignore the reuse relation, as exploiting reuse in such a case would result in an excessively complex code with very high control overhead.

Reuse is available for the portion of the source domain determined by reuse validity constraints on the processor parameters. Such validity constraints are obtained by substituting the expression “th.x+val(Y)” for “th.x” in the portion of the reuse relation corresponding to the source statement (i.e. the projection of the reuse relation on the source
for (i = 0; i <= 255; i++) {
    ...
    U2_l_1 = U2[3 + i][4 + 16 * bl.x + th.x];
    U2_l_9 = U2[4 + i][4 + 16 * bl.x + th.x]; // C1
    U2_l_2 = U2[5 + i][4 + 16 * bl.x + th.x];
    ...
    if (th.x <= 3) {
        U2_l[4 + th.x] = U2_l_9; // C2
        U2_l[th.x] = U2[4 + i][16 * bl.x + th.x]; // C3
    }
    if (th.x >= 4) {
        U2_l[4 + th.x] = U2_l_9; // C4
    }
    doall (j = max((36-th.x)/16, 1); j<=(23-th.x)/16; j++) {
        U2_l[16*j+th.x] = U2[4+i][16*j+16*bl.x+th.x]; // C5
    }
    __syncthreads();
    U1[4 + i][4 + 16 * bl.x + th.x] = U2_l[....] ...
    __syncthreads();
}

Figure 9: 2-D DWE Optimized Private-to-Shared iteration domain). Reuse validity constraints cut the iteration space of the source statement $C_2$ into a portion that exhibits reuse and a portion that does not. These portions result in index-set splittings (obtained by a set difference operation) that are embodied by statements $C_2$ through $C_5$ in Figure 9. This optimization can be viewed as a rotation of the code performing the copy into $U2_J$ across threads until the data is properly aligned.

Examining code in Figures 8 and 9 illustrates an interesting property of processor parameters. Statement $C_2$ of Figure 8 has 2 iterations when executed by processors with $th.x \leq 7$ and only 1 iteration when executed by processors with $th.x \geq 8$. After reuse exploitation, the following distribution occurs (1) processors with $th.x \leq 3$ execute 1 iteration of $C_2$ (optimized) and 1 iteration of $C_3$ (unoptimized), (2) processors with $4 \leq th.x \leq 7$ execute 1 iteration of $C_4$ (optimized) and 1 iteration of $C_5$ (unoptimized), (3) processors with $th.x \geq 8$ execute only 1 iteration of $C_4$ (optimized). Eventually, 16 copy iterations out of the original 24 are optimized. The reader may care to verify that both versions execute exactly 24 copy operations.

3.6 Other Transformations

Lastly, R-Stream performs two simple loop transformations aiming at enhancing the control-flow behavior of the copy statements.

3.6.1 Hoisting/Peeling

This transformation involves hoisting data transfer code above loops whenever possible. Typically, when the first (or last) iterations of a loop behave differently, we can generate guards within the loop for these statements (such as if $(i<2)\ldots$), or we can peel the proper number of iterations. In practice, this behavior is triggered by simply hoisting the communication above the reuse dimension. At code generation, during polyhedral separation [24], the hoisting results in a peeling of the first and/or last iteration of the loop. We have experienced quite substantial performance variations (over 30%) depending on GPUs and nvcc versions.

3.6.2 Loop Interchange for Inner Loop Unrolling

The benefits of this transformation are also related to the capabilities of the back-end compiler. When targeting CUDA, it is usually detrimental to have innermost loops that depend on the threadIdx.x parameter. The main reasons are related to the difficulty of unrolling loops with trip counts that vary with the processor id as well as the issue of thread divergence. In the case of copies, innermost loops can be freely interchanged and it is easy to do so in the polyhedral model, whatever the imperfectly nested structure of these copies may be. The following code illustrates this phenomenon. It is clear that the innermost loop is easier to unroll in the rightmost version. Unrolling may be done directly in R-Stream or triggered by a pragma generated by R-Stream. Also, note that when using 32 threads on the threadIdx.x dimension, the $j$ loop executes: (1) 2 iterations if $0 \leq th.x \leq 24$, (2) 3 iterations if $25 \leq th.x \leq 31$, thus adding to the complexity of performing a simple loop unrolling. Lastly, thread divergence forces synchronizations in the $j$ loop. The left version has twice as many synchronizations than the version on the right.

```c
int ix = 32 * bl.x + th.x;
int iy = 8 * bl.y + th.y; // After Interchange
for (k=0;k<=1;k++) { for (j=0;j<=(th.x+39>>5);j++) {
    for (j=0;j<(th.x<<2));j++) {
        U2_l[8*k+th.y][32*j+th.x] = U2[16*k+th.y][32*j+th.x] =
        U2_3[4+i][8*k+16*bl.x+th.x];
    }
}
```

Figure 10: Effects of interchanging $j$ and $k$

3.7 Control Flow Overhead

Current limitations to proper communication optimization are related to explosions in the code complexity which have serious adverse effects, in particular on CUDA. For instance, the optimization described in section 3.2 should be seen as a special case of the work of Plesco et al. [5]. It only removes redundant communications in the particular case where hoisting is sufficient. This behavior can be extended to handle more advanced reuse patterns but would then rely on index-set splitting transformations to properly break down the different cases. Complex control-flow is then emitted, especially when combined with explicit placement parameters introduction which result in adverse performance results. One peculiarity of the CUDA programming model is related to the handling of parameters encoding the various threads of execution. Consider a single occurrence of a statement $S$ whose control is determined by a thread parameter (for instance $S(threadIdx.x)$ with $0 \leq threadIdx.x \leq 31$), the semantics of the threadIdx.x parameter is that 32 occurrences of $S$ execute, one for each thread. Low-level CUDA constraints require that the unit of atomic execution is a half-warp (i.e. a block of 16 contiguous threads threadIdx.x aligned on 0 modulo 16). Programs may be predicated by their threadIdx.x but may result in sequential code. In the simple example below, the first half-warp code for $0 \leq threadIdx.x \leq 3$ is executed first, then $4 \leq threadIdx.x \leq 15$. In parallel, the second half-warp code for $16 \leq threadIdx.x \leq 31$ may

3The index-set splitting is necessary for case disjunction and is conceptually similar to the different branches of the QUAST representation used by Plesco et al.
be executed since the second half-warp is not “cut” by the control flow. This execution property is called thread divergence. When thread divergence occurs, the performance of the automatically generated code is greatly degraded.

```c
if (threadIdx.x < 4) {
  S(i, threadIdx.x)
} else {
  S(i+1, threadIdx.x)
}
```

float __shared__ U2_1[16][40];
float U2_1_1, U2_1_2, U2_1_3, U2_1_4, U2_1_5, U2_1_6, U2_1_7,
U2_1_8, U2_1_9;
// Hoisted prologue for register reuse
U2_1_5 = U2_3[0][4 + iy][4 + ix];
U2_1_4 = U2_3[1][4 + iy][4 + ix];
U2_1_3 = U2_3[2][4 + iy][4 + ix];
U2_1_2 = U2_3[3][4 + iy][4 + ix];
U2_1_1 = U2_3[4][4 + iy][4 + ix];
U2_1_0 = U2_3[5][4 + iy][4 + ix];
U2_1_9 = U2_3[6][4 + iy][4 + ix];
U2_1_8 = U2_3[7][4 + iy][4 + ix];
U2_1_7 = U2_3[8][4 + iy][4 + ix];
for(i = 0; i < 255; i++) {
  // Register reuse
  if (i > 1) {
    U2_1_5 = U2_1_4;
    U2_1_4 = U2_1_3;
    U2_1_3 = U2_1_2;
    U2_1_2 = U2_1_1;
    U2_1_1 = U2_1_0;
    U2_1_0 = U2_1_9;
    U2_1_9 = U2_1_8;
    U2_1_8 = U2_1_7;
    U2_1_7 = U2_3[8 + i][4 + iy][4 + ix];
  } else {
    U2_1_5 = U2_3[8][4 + iy][4 + ix];
  }
  // Complex loop structure, non-convex transfer
  for(j ...) {
    for(k ...) {
      U2_1[8*ky+th.y,32*j+th.x] = U2[4+i,8*ky+8*bl.y+th.y,32*j+32*bl.x+th.x];
    }
  }
  // Shared memory opt – Filling from register
  U2[1[threadIdx.y + 4] = [threadIdx.x + 4] = U2_1_9;
  __syncthreads();
  // Compute
  U1[4 + 1][4 + 16 * bl.x + th.x] = ...
  __syncthreads();
```

Figure 11: Register and Shared reuse 3-D DWE

In addition to thread divergence, another limitation (from the point of view of automatic CUDA generation using the polyhedral model) arises from the fact that synchronization statements must not diverge. This has serious implications on the polyhedral code generation step [2]: statements belonging to loops where synchronizations are present must not be broken by polyhedral separation [24]. This combines poorly with the need to generate non-convex transfers and other optimizations to simplify the output code. We believe the difficulty of handling non-convex transfers is precisely the reason a specialized communication library approach [4] performs better than the hand-tuned solution which inspired our work [22]. A more careful study of harnessing the complexity of the automatically generated CUDA is a topic of future research. It is likely, constraints should be imposed on the form of schedules and placement functions. Another possible solution could be to target high-level memory primitives for GPUs [4].

4. EXPERIMENTS

We present experimental results on two different GPU chips, (1) GTX 285 and (2) GTX 480. The GTX 285 has 30 multiprocessors each with 8 streaming processors and 1GB total device memory. Each multiprocessor has a 16KB user-managed on-chip cache (shared memory) and 16K registers. The GTX 480 has 15 multiprocessors each with 32 streaming processors and 1.5GB total device memory. Each multiprocessor has 64KB on-chip cache which the user can configure either as a 16KB user-managed cache and a 48KB hardware-managed cache or vice-versa.

In a first experiment, we generate different code versions with the various optimizations we described. Measured execution times (in milliseconds) are compared to two baselines in Table 1: (1) a “simple mapping”, without the optimizations described in this paper (i.e. it always loads the data from global memory into shared memory before the execution of each tile); and (2) a hand-optimized version following the guidelines of [22], where the authors perform optimal tile size selection, register rotation and shared memory optimization by hand. Note that the “simple mapping” is by no means naive, it corresponds to code with a well-performing tile size and is generated automatically.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Performance(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GTX285</td>
</tr>
<tr>
<td>Simple mapping</td>
<td>12.91</td>
</tr>
<tr>
<td>Register reuse</td>
<td>- no loop interchange + no hoisting</td>
</tr>
<tr>
<td></td>
<td>- no loop interchange + hoisting</td>
</tr>
<tr>
<td></td>
<td>- loop interchange + no hoisting</td>
</tr>
<tr>
<td></td>
<td>- loop interchange + hoisting</td>
</tr>
<tr>
<td>Shared mem opt</td>
<td>9.82</td>
</tr>
<tr>
<td>Manual [22]</td>
<td>4.58</td>
</tr>
</tbody>
</table>

Table 1: Execution times for DWE,3-D (ms)

In a second experiment, we evaluate the best combination of our techniques on three different out-of-place stencil kernels. (a) a Discrete Wave Equation kernel (a 3-D order-8, 27 points stencil, 1 input array and 1 output array). (b) a 3-D finite-difference (FD) kernel (a 3-D order-2, 7-points stencil, 1 input array and 3 output arrays), and (c) a Heat-3D kernel (a 3-D order 2, 7 points stencil, 1 input array and 1 output array) modeling the solution to the heat equation. All performance measurements are run in single precision on problems of size 256 × 256 × 256. Table 2 presents performance results, as previously, the “simple” version uses well-performing tile sizes; the “best” mapping corresponds to the best combination of reuse, interchange and hoisting transformation.

<table>
<thead>
<tr>
<th>Stencil</th>
<th>GPU</th>
<th>DWE</th>
<th>FD</th>
<th>Heat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>285</td>
<td>480</td>
<td>285</td>
<td>480</td>
</tr>
<tr>
<td>Simple mapping</td>
<td>12.91</td>
<td>7.79</td>
<td>6.04</td>
<td>4.7</td>
</tr>
<tr>
<td>Best mapping</td>
<td>5.99</td>
<td>4.68</td>
<td>4.74</td>
<td>3.50</td>
</tr>
<tr>
<td>Manual [22]</td>
<td>4.58</td>
<td>4.20</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Execution times of 3-D stencil kernels (ms)
We attribute the difference in performance between the hand-tuned version and our best generated version to the lack of shared memory reuse optimization in our best performing variant. The impact of this transformation is negative due to the non-convexity of the optimized transfer code and the thread divergence issues arising from the control-flow complexity. On GTX 480, the presence of a cache alleviates the importance of the shared memory optimization but also reduces the overall impact of the transformations. Still, the best DWE version generated automatically by R-Stream is within 23% of the hand-tuned DWE code on GTX 285 and within 10.2% on GTX 480. Lastly, the relative benefits of the optimizations we identified increase with the order of the stencil which make them promising for the study of larger stencils.

5. RELATED WORK

Our algorithm for memory reuse optimization automates previous and tuning strategies for stencil applications on CUDA [22]. We have created a general framework based on dependence analysis across iterations and processors and memory addressability that can be applied to other architectures. In the process, we have also added capabilities to R-Stream that can be used in iterative compilation and auto-tuning such as annotations for selecting which arrays are promoted to local memory as well as the granularity of communications. In the context of CUDA, our work could be related to previous contributions [27, 30] although the base assumptions are very different. Both contributions assume an embarrassingly parallel input: (1) “applications that have been ported to CUDA” [27] and (2) “The input to our compiler is a naive GPU kernel function, which is functionally correct” [30]. The first contribution also seems to be mostly manual. Both these contributions ignore the 40-year old problem of auto-parallelization that we attack. Our input is a sequential specification. We use the full power of the polyhedral model to optimize quantities such as amount of parallelism, amount of reuse, number of memory transfers. Our work builds on a long tradition dating back to work in the systolic array community [26] and the first algorithm to provably extract the maximal amount of fine-grained parallelism in affine control programs [3]. The solution we chose for the problem of local memory management is an extension of Schreiber’s work [28]; a discussion on modular mappings is provided in section 2.1. Other compilers capable of generating CUDA include the HMPP compiler [7] and The PGI Accelerator Compiler [12]. We are not aware of advanced communication optimizations being implemented in their frameworks at this time.

Holewinski et al. [13] propose a solution that trades additional communications for reduced communications and control-flow regularity for stencils on GPUs. Their approach is based on overlapped tiling and handles tiling across multiple time iterations. They demonstrate impressive results. However, they mention, “3-D stencils are not handled well by overlapped tiling on GPU architectures due to the amount of redundancy computation that is needed for even small time tile sizes”. Our approach is complementary and reduces communications without introducing additional computations. Its expected benefits increase with the size and volume of the considered stencils.

Specific distinctions in the comparison with the closest related work of Plesco et al. [5] have been discussed in the relevant sections. We should also add that the control flow overhead problem introduced by modular expressions would likely be worse in the context of CUDA.

6. CONCLUSION AND FUTURE WORK

In this paper, we start from R-Stream, a full-fledged polyhedral source-to-source compiler implementation which performs dependence analysis, state-of-the-art affine scheduling, imperfect loop nest tiling, distribution of computations across a processor hierarchy and scratchpad memory management. We then augment R-Stream with capabilities to generate finer-grained communications and explore communication optimizations based on various models of memory addressability that are generally applicable. All these steps are fully automated and applied within the polyhedral representation. We implemented multiple tuning options to throttle the granularity of communications and the aggressiveness of memory reuse exploitation. In the future it will be interesting to use these parameters in an iterative framework and devise cost models that try to predict the tradeoff between memory reuse benefits and increase in code size complexity. It will be particularly interesting to investigate programs that are not parallel unless skewing transformations are applied. These programs tend to quickly result in complex codes when compiled for CUDA.

7. REFERENCES


