Productivity via Automatic Code Generation for PGAS Platforms with the R-Stream Compiler

Benoît Meister, Allen Leung,
Nicolas Vasilache, David Wohlford, Cédric Bastoul and Richard Lethin
{meister, leunga, vasilache, wohlford, bastoul, lethin}@reservoir.com

Reservoir Labs, Inc.

Abstract. Emerging computing architectures present concurrent, heterogeneous, and hierarchical organizations. Explicit management of distributed memories, bulk communications, and the careful scheduling of data and computation for locality of reference appear to be necessary to achieve high efficiencies relative to the peak performance. In some cases, the architectures present mixed execution models. We present the design of a software mapping tool, the R-Stream® High-Level Compiler, which permits a simplified programming model in terms of abstracted, programmer-friendly expressions of algorithms by providing an automatic procedure for producing a mapping that conforms to the requirements for the emerging architectures. This mapping procedure includes phases that can produce a variety of communication models provided by PGAS (specifically, asynchronous bulk communications) covered by a variety of automatically generated synchronization methods, and includes provisions for addressing the challenges of employing accelerators such as GPUs. The approach of targeting Asynchronous PGAS from a high level description of the algorithm simplifies mapping challenges, by simplifying the challenge of "raising" algorithm descriptions that are obscured by programmer-provided physical mapping details.

1 Programming Model

The next step up in absolute computational performance and in computational efficiency for envisioned next-generation architectures [KBC+08] will entail programming problems that are much thornier than just the traditional challenges of parallelization or vectorization. Locality of reference, to avoid burning power for moving data unnecessarily has always been important, but it will become much more important. Architectures will be composed of many levels of hierarchy and mixed computing models (e.g., clusters of multi-core). Computational accelerators (e.g., GPGPU, ClearSpeed, FPGA) will provide application class specific performance boosts, but will further complicate the programming model. Some researchers are providing evidence that the best processor utilization can be achieved by explicit programmer management of architectural features, vs. dynamic reactive hardware management[DKW+09]. PGAS languages reflect this trend by allowing the programmer to indicate a rich set of options for organizing the placement of data with varying degrees of explicitness (e.g., in terms of

specific processors in UPC [CBD+03] or in terms of a less concrete "locale" in Chapel [CCZ07]).

This has led to research into compilers and tools for taking programs expressed in a PGAS language and transforming them to incorporate the important aspects of explicit control (e.g., bulk communications, overlapped communications/computation) needed for good performance. The challenge in doing this is that when an algorithm has been rendered to a PGAS language, aspects of the original intent of the programmer in terms of the logical constructs of the algorithm e.g., the high-level arrays and the fundamental algorithm, are not necessarily present. Furthermore, reasoning (automatically or manually) about concurrency is an intractable problem in both a formal and practical sense. This reduces the scope of optimization of a section of a program written in a PGAS language.

One approach to addressing this dilemma would be to refine the existing PGAS programming models for these explicitly managed hardware mechanisms, extending the programming languages with new keywords, semantics or defining particular idioms in the existing languages to indicate the more detailed mapping. This does allow programmers to "get their hands on the knobs" of the machine. But it reduces portability and productivity. It also further obscures the logical intent.

An alternative, which we are investigating, is to allow the programmer to express their algorithm at a high level, as abstracted as possible from the final machine and execution model. A compiler takes this abstract description and generates the mapping. This does mean, for every explicitly managed mechanism of machine execution that is removed from the programming model, that the compiler's job grows. However, it also makes the job of the compiler simpler. First of all, there is less work to do in trying to "raise" the program to semantics closer to the original programmer intent. The compiler starts with something that it can map readily. Second, the input program is smaller - details expressing physical control of the machine are not present in the original program. With extraneous constraints removed, the space of target mappings that the compiler can choose from is expanded. But while the space is greater, it is also simpler - the description is compact, and in forms that are tractable as analytical mathematical optimizations. In this approach, the PGAS language is the target language for the compiler.

The high-level compiler, R-Stream, that we have built, illustrates the merit of this approach. Accepting the algorithms at a high-level, in this case, as sequential loop nests in C[1], facilitates the abstraction of the program in a manner such that the objectives of PGAS mapping are more naturally formalized and implemented. Section 2 describes the R-Stream compiler. In Section 3, we show how the approach facilitates the mapping of the algorithm to the PGAS abstrac-

[1] That our compiler accepts and emits the mapped algorithms in C is a practical choice. The mapping algorithms are orthogonal to the C implementation; it might well be easier to accept the algorithms in an even higher-level language, e.g., a matrix language such as Flame [GGHvdG01].

tion. We illustrate the relevance of R-Stream by focusing on an algorithm for rearranging computation and explicit asynchronous communication operations which implements multi-buffering automatically in R-Stream . We end by giving our conclusions in Section 4.

2 An overview of polyhedron-based parallelization techniques

R-Stream accepts a domain-specific class of programs, which consists of "regular" loop nests that work on multi-dimensional arrays of data. This is good for HPC and HPEC linear algebra and signal processing applications.

R-Stream uses a particular mathematical model to represent programs, presented in Section 2.1. The model offers a way to manipulate a representation of those programs under the form of multi-dimensional polyhedra whose definition depend on symbolic constants. The polyhedra reflect the iteration space of the loops, the coordinates of the array elements they access and the order in which they are accessed. The symbolic constants represent program expressions that are constant during the execution of the loop. We call those symbolic constants "parameters." Polyhedron-based optimization differs from "classic" loop optimization [KA02] in offering a means for finding and using perfect dependence information and for subsuming many of the classic transformations into single "uber" steps, (e.g., "parallelization") framed as mathematical analytic optimization steps. Also, by supporting imperfect loop nests, the polyhedral model allows much larger scopes of loops, i.e., interesting kernels, to be considered jointly in optimizations.

The R-Stream compiler, presented in Section 2.2, leverages this model and some of the compiler theory literature for polyhedral optimization, to proceed to automatically parallelizing sequential C programs to different multi-core target machines. R-Stream is unique, we believe, in offering (1) a robust implementation of the polyhedral optimization algorithms in the literature, with improvements to make them practicable (e.g., scalable to interesting kernels) (2) a foundation of a state of the art, type-preserving scalar C infrastructure (3) a "pure" polyhedral approach, whereas all significant parallel mapping steps are performed within the polyhedral representation, (vs. e.g., intermediate re-renderings and raisings from syntactic or scalar representations.) and (4) several new algorithms, specifically associated with emerging architectures. While many of the new algorithms would possibly be of interest to this workshop audience, this paper focuses on one of these new algorithms, in particular, the re-scheduling of computations and explicit communications to automatically put a multi-buffering scheme in place (in Section 3).

2.1 The polyhedral model of loop nests

The requirements of the polyhedral model shape the set of input programs that R-Stream accepts. They must be close to "regular" static control program form, according to this strict definition:

– the loop counters must be integer, and they must be incremented by constant steps
– the loop counters must be bounded, and the bounds of a loop counter must be an affine expression of the value of the outer loop counters and of the parameters.
– the statements enclosed within the loops access multi-dimensional arrays[2] through multi-dimensional affine functions of the loop counters and the parameters.

When considering the loop counters as variables in a vector space, the set of iterations of the loop nest is defined as a parametric polyhedron. An example of such a loop nest is illustrated in Fig. 1. The considered iteration domain can be described by the following polyhedron:

$$P_1(n) = \{(i,j) \in \mathbb{Z}^2 \mid \exists k \in \mathbb{Z}, i = 2k, 5 \leq i \leq n, 0 \leq j \leq i\}$$

Similarly, the set of points of array A that are accessed through the illustrated reference is defined as:

$$D_1(n) = \{x, y \in \mathbb{N}^2 \mid x = i + 2j + n, y = i + 3, (i,j) \in P(n)\}$$

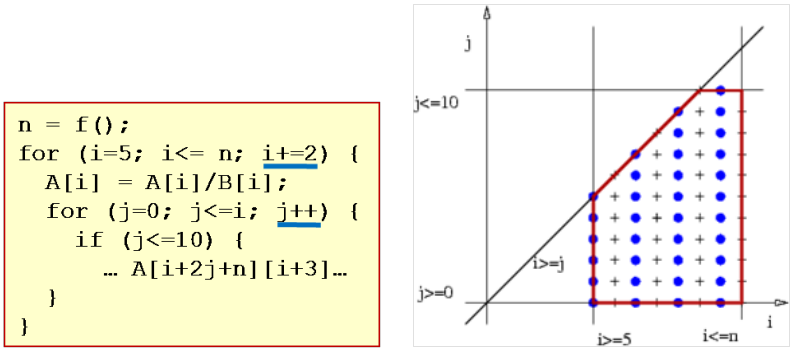

Fig. 1. Polyhedral representation of a loop's iteration domain

R-Stream extends this strict model to a larger application domain of less regular programs by using approximate – but conservative – models of the loops' execution. For instance, while only conditionals whose predicates are affine constraints on the loop counters are supported in the strict model, R-Stream supports the presence of arbitrary and data-dependent conditionals. We call our larger (and increasing) set of input programs the "extended static control program" form.

[2] which include scalars and one-dimensional arrays

2.2 The R-Stream compiler

The basis of our mapping tool is Reservoir Labs' proprietary compiler, R-Stream. DARPA funded development of R-Stream by Reservoir between 2003 and 2007 in the Polymorphous Computing Architectures (PCA) Program [LLM+08].

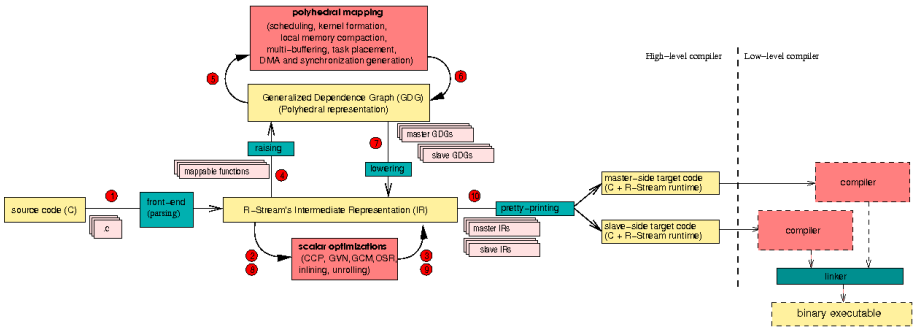

Fig. 2. R-Stream compiler flow

The flow for the R-Stream high-level compiler is shown in Fig. 2. An EDG-based front end reads the input C program (1), and translates it into a static single assignment intermediate form, where it is subject to scalar optimizations (2) and (3). Care is taken to translate the C types into a sound type system, and to preserve them through the optimizations, for later syntax reconstruction in terms of the original source's types.

The part of the program to be mapped is then "raised" (4) into the geometric form based on parametric multidimensional polyhedra for iteration spaces, array access functions, and dependencies.

In this form, the program is represented as a "Generalized Dependence Graph" over statements, where the nodes represent the program's statements and are decorated by the iteration spaces and array access functions (polyhedra), and the inter-statement dependences are represented as labeling of the edges.

Kernels that meet certain simple criteria, "mappable functions" that can be incorporated into the model, are raised. No new syntactic keywords or forms in the C program are involved in this criteria for raising; R-Stream raises various forms of loops, pointer-based and array based memory references, based on their semantic compatibility with the model. We note here that the type of C that R-Stream optimizes best is "textbook" sequential C, e.g., like a Givens QR decomposition in a linear algebra text. If the programmer "optimizes" their C (e.g. with user-defined memory management or by linearizing multidimensional arrays), those optimizations typically interfere with R-Stream's optimizations.[3]

[3] So then we need to bring more ammo to the raising problem, e.g., we have in progress the development of raising modules that de-linearize array accesses. We are op-

These mappable functions are then subjected to an optimization procedure which (for example) identifies parallelism, tiles the iteration spaces of the statements into groups called "tasks," places the tasks to processing elements (PEs, which correspond to the cores of a multicore processor or the nodes of a cluster), sets up improvements to locality, generates DMA, adds synchronization and organizes transfers of control flow between a master execution thread and the mapped client threads.

Each of these mapping steps both reads (5) and writes (6) to the GDG. A diagram of this mapping process is shown in 3.

After this process, the GDG is lowered (7) back to the scalar IR, through a process of "polyhedral scanning" that generates loops corresponding to the transformed GDG. This scalar IR of the mapped program is then subject to further scalar optimization (8) and (9), for example to clean up synthesized array access functions. The resulting scalar IR is then pretty-printed (10) via syntax reconstruction to generate the resulting code in C form, but mapped in the sense that it is parallelized, scheduled, and has explicit communication and memory allocation.

Code that was not mappable is emitted as part of the master execution thread, thus the R-Stream compiler partitions across the units of a heterogeneous processor target. From this point, the code is compiled by a "low-level compiler" (LLC), which performs relatively conventional steps of scalar code generation for the host and PEs.

R-Stream supports generating code mapped for STI Cell and SMP with OpenMP directives. There are projects in progress targeting R-Stream to ClearSpeed, GPGPU (emitting optimized code in CUDA form), FPGA-accelerated targets, and Tilera.

This mapping process is driven by a declarative machine model for the architectures, using an XML syntax for describing hierarchical heterogeneous accelerated architectures. The contents of the machine model provide the overall architecture topology as well as performance and capacity parameters, a formalization and implementation of a Kuck diagram [Kuc96]. For example, it is possible to describe a shared memory parallel computer with many FPGA accelerators, such as SGI Altix 4700 with RASC FPGA boards. It also can describe the complex execution model for GPGPU, in terms of the complex distributed memories and parallel execution engines [Gro09]; it can also describe (and we have plans to map to) machines with multiple hosts and multiple GPGPU accelerators. While this language (and a graphical machine model browser for it) are human-understandable, they are also closely tied to the mapping algorithms, providing parameters for different optimizations and driving the mapping tac-

---

timistic that it would be possible to raise some PGAS languages. For example, Chapel with its ZPL background, is array-oriented, so the array constructs should be retrievable from the language. Chapel's language strategy of locales make the physical considerations orthogonal to the logical algorithm. It would be interesting to study what Chapel's locale information means in terms of the analytical locality optimizations in R-Stream.

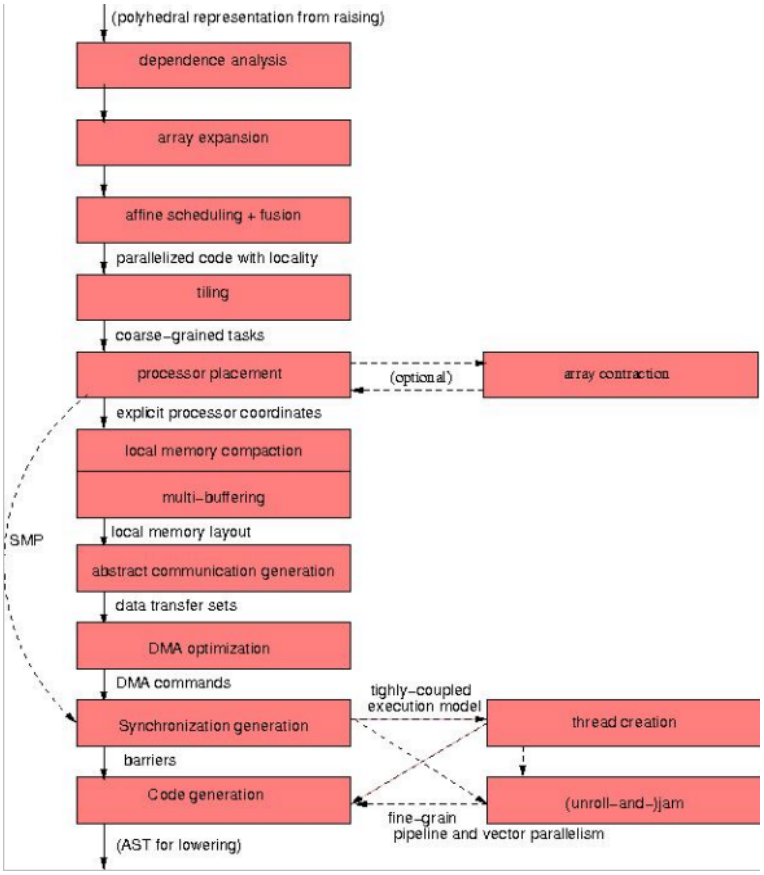

Fig. 3. The R-Stream mapper flow

tics. For hierarchical targets, the mapper can perform a recursive partitioning of the source kernel that is refined at lower levels for the child PEs. The final code is emitted for the target in terms of the task invocations, explicit communications, and synchronizations among parent and child PEs and among the PEs at any level.

For some targets, R-Stream uses existing APIs, e.g., OpenMP pragmas for SMP, C^n for ClearSpeed [cle07], or a proprietary assembly API for the Mitrion Virtual Processor for FPGA; in other cases, for example Cell, R-Stream targets a more abstracted API for explicit operation on the machine that overlays DaCS/ALF, and which is portable to other targets. It is easy to change the syntax from one target to another. For example, it would be easy to make R-Stream target the PGAS language UPC or (perhaps better) the underlying execution abstraction layer, GASNet. What matters more than the syntax is the particular target execution model; R-Stream can target a variety, e.g., with variations in where DMA is dispatched, in how tightly host processors are coupled with their accelerators, or in even whether there *is* a host at all.

A full accounting of the capabilities of the individual mapping phases (Fig. 3) is beyond the scope of this workshop paper and will be submitted for publication elsewhere. As mentioned earlier, Section 3 focuses on the multi-buffering algorithms for overlapping communications and computation, which is a focus of this workshop. Briefly, we give some description of the other phases. For the affine partitioning step (or parallelization) we provide implementations of some algorithms described in the literature [Fea92,LL97,BBK$^+$07], though by default R-Stream uses new parallelization steps that jointly balance locality considerations. Tiling is implemented as a guided search that hierarchically groups and tiles iterations that respect scratchpad memory capacities into tasks, and that carves boundaries according to considerations such as SIMD alignment. Placement distributes tasks across the physical processor space. For SMP architectures, steps of communication and synchronization generation are then skipped (since communication is implicit in memory accesses and, at least for OpenMP, synchronization comes for free at the exit of parallel constructs). For distributed memory machines, though, the local memory compaction stage begins the formation of abstract communication operators in the GDG, and later steps progressively form them into concrete operators such as DMA calls. Various architecture, thread generation and synchonization generation phase sequences further refine the schedule depending on the degree of coupling of the host to accelerators. Code generation is done via an improved polyhedral scanning algorithm. Data layout optimization is via optimized "reshapings" using the synthesized DMA operations as data is moved from one place (e.g., main memory) to the scratchpad and results are written out, and via the use of periodic polynomial access functions [CM00].

R-Stream is several hundred thousand lines of Java, which we find improves our productivity in implementing new representations and optimizations. Mathematical optimization problems are solved with our proprietary libraries for manipulating Z-Domains that include new algorithms for rapid exact and approx-

imate counting [MV08] that are used, for instance, to estimate bandwidth and data footprint, and which are significantly more robust than and competitive with the open source versions which they subsume.

3 Hiding distant memory access through automatic multi-buffering

Multi-buffering is a well-known high-level software pipelining technique whose objective is to hide the latency of asynchronous communications by starting the fetching of the input data of future tasks, and by postponing the check for termination of the sending of output data on full-duplex asynchronous communication channels.

In this section, we contribute an algorithm for automatically modifying parallel loop-intensive codes based on bulk one-sided communications. The algorithm is relevant to distributed-memory systems which have access to a common memory, such as in the PGAS programming model.

We give conditions for the presented algorithm to be applicable, and explain why R-Stream's mapping components expose opportunities for optimizing multi-buffering.

3.1 Algorithm

Our algorithm is applicable to codes that contains loops, possibly complex (hierarchy of imperfectly nested loops) and irregular, which iterate on a number of computational entities, which we call "tasks", and for which communication commands associated to these tasks are present. Those communications include all the possible data transfers that can be done asynchronously, whether they are transfers from and to remote memory or local copies.

Without loss of generality, we will assume that the incoming communications are already expressed as asynchronous communications, i.e., as a pair of operations: (`recv`, `recv_wait`), where the "recv" operation initiates the incoming transfer and the "recv_wait" waits for the transfer to complete. Similarly, outgoing transfers are assumed to be written as a pair (`send`, `send_wait`). One frequent optimization, supported here, is to have one "wait" operation to wait for the completion of several incoming and outgoing transfers.

Putting a multi-buffering execution scheme in place requires duplicating the data structures whose transfers are optimized and re-scheduling the transfers. We will call a "buffer" any of the B locations to hold a copy of a data set. The determination of the desired number B of buffers, and the number a of tasks executed between a `recv` and the execution of the task that needs the data, is out of the scope of this report.

Let us call $l = B-a-1$ the number of tasks executed between the termination of a task that uses a given buffer and the execution of the `send_wait` operation for that buffer. At a high level, our algorithm proceeds as follows:

1. Select a number of "pipeline-prone" loops, in which multi-buffering will be implemented. Let D be the data set transferred within those loops.
2. Select an "appropriate" enclosing loop to iterate over B buffers. Let us call that loop the "buffer loop".
3. Sink the inner resulting loop down to the pipeline-prone loop level(s).
4. Produce B copies of the data sets.
5. Re-schedule the iterations of the `recv` operations by shifting them by $-a$ iterations along the buffer loop.
6. Re-schedule the iterations of the `send_wait` operations by shifting them by $B - a - 1$ along the buffer loop.
7. Insert one "buffer rotation" operation in every buffer loop involving the data set.

Multi-buffering has a cost in terms of control flow overhead and memory footprint. Hence in step 1, only loops whose bodies have the largest number of occurrences are selected. We call pipeline-prone loop levels the set of loops along which a set of coarse-grained communications and computations are repeated. They are the loop levels along which it is possible to perform multi-buffering. To define them more precisely, they are "the level immediately outer to the outermost loops or statements that enclose either only communications or only computations."

The buffer loop is chosen so as to respect the following constraints: if it encloses communications for one data set, it has to enclose all the communications for that data set. Three types of loops are valid as buffer loops, as soon as their trip count is at least as large as B:

- loops that do not carry dependences (also called "doall" loops),
- loops whose read-after-write dependences on D have a maximum non-negative dependence distance of l and do not enclose inter-processor synchronizations. This latter class of loops relies on the ability for the system (whether it is the compiler or the runtime) to detect this dependence and turn the multiple accesses to the same data, which are materializing the dependence, to be optimized to use the locally-defined data rather than to re-load (a stale version of) it. The fact that the loop does not contain any inter-PE synchronization ensures that the dependences carried by the loop is intra-PE, and that this optimization is correct.
- loops whose dependence distances are all greater than B. We believe that this case is very rare, unless artificially produced by loop restructuring, and hence we will ignore it.

An additional constraint for the latter two types of loops is that it must be legal to sink those loops down to the pipeline-prone levels.

Notice that choosing a non-doall buffer loop enforces a strict order of execution of the multi-buffered loop. Using a doall loop as the buffer loop makes it possible to dynamically schedule the execution of the loop iterations, based on the arrival of their input data, in the flavor of what GPGPUs do. But the program may not have enough doall parallelism available, and the use of the

second type of loops significantly increases the opportunities for overlapping asynchronous communications with computations.

3.2 Trade-offs

When determining the multi-buffering characteristics of the program, a number of sometimes conflicting constraints appear. Ideally, a should be large enough for the execution of a iterations of the buffer loop to cover the average communication latency of the input data. The greater l, the more loops may have a dependence distance of less than l and become buffer loop candidates. Increasing a or l increases B, which brings up two issues. When the amount of storage for temporary data is limited, the size of the buffers is limited as well, which in turn decreases the amount of computations that can be done on each buffer and the granularity of communication. Also, the trip count of buffer loops must be at least B, hence the number of candidate buffer loops decreases when B increases.

Typically, the user has no handle on communication latencies, nor (arguably) on the amount of temporary data storage available. Hence, in order to keep B reasonably low while keeping a good coverage of the communication latency, the user's only option is to restructure the code in order to keep l low, by exposing either doall loops or loops with short maximum dependence distance.

In addition to forming parallel tasks and their related bulk communications, R-Stream produces loops whose carried dependences have a short maximum distance, which maximizes the number of candidate buffer loops in the resulting program. Additionally, the depth down to which a loop is sinkable is implicitly encoded in the polyhedral mapper's representation of the code. We have leveraged those properties and implemented automatic multi-buffering components in R-Stream.

3.3 Experiment

The issue of PGAS-based PEs storing data from other PEs in a coarse-grained manner poses identical problems to a compiler as it does for distributed systems which have access to a common memory, with the main difference that in the PGAS paradigm, the common memory is purely distributed across the PEs.

We have chosen to illustrate the multi-buffering algorithm on such a setting offered by the Cell processor, for a 1024×1024 matrix multiply code with $B = 2, a = 1, l = 0$, in Figure 5 (with minor cosmetic modifications). The original algorithm , on the left side, is parallelized by R-Stream across 8 PEs. The "PROC0" parameter to the function represents the identifier of the running PE.

The algorithm chooses loop k as pipeline-prone loop, which contains only communication statements (128 of them) and pure-computation loops. Communications on C are deemed negligible as compared to communications on both A and B. The algorithm selects k, which does not carry any read-after-write dependence on A or B. Its maximum read-after-write dependence distance of zero, which is less than or equal to l, makes it eligible. Its trip-count of 64 also makes it the only eligible loop worthwhile for multi-buffering, since 2 (for the i

loop) and 4 (for the j loop) iterations are not enough to amortize the initiation cost.

Shifting the iterations of the `dma_get` operations is trivial in R-Stream's polyhedral mapper, since it consists in a simple translation of their iteration domain along k. In R-Stream, all the code transformation, the generation of communications, synchronizations and transfer of control operations, and their subsequent optimizations happen in a polyhedral representation. All the burden of producing C code whose execution corresponds to the polyhedral program definition is left to a final phase, sometimes called "polyhedral code generation" or "polyhedral scanning" [QRW00,VBC06].

The performance measurements over five runs for three versions of the code are reported on Figure 4: the original as provided by R-Stream, the multi-buffered version, and the multi-buffered version in which the DMA operations were removed.

Version	**Exec. time (s)**	**GFLOP/s**
original	0.0299 ± 0.0003	71.9 ± 0.8
multi-buffered	0.0217 ± 0.0006	98.8 ± 2.8
multi-buffered, no DMA	0.0216 ± 0.0005	99.4 ± 2.3

Fig. 4. Performance measurements over five runs

The overlapping performance numbers of the multi-buffered version and the version without DMAs confirm that the 27 GFLOP/s performance improvement obtained with multi-buffering is due to communication-computation overlap (as opposed to tile sizes or DMA shape, for instance). Their slight difference is most likely explained by the fact that transfers of array C were not multi-buffered. No additional processor-specific optimization (SIMDization, register tiling) was performed in this experiment, which explains why these numbers are still 45% of the peak performance of the Cell chip.

4 Conclusion

We have demonstrated how explicit communication management can be optimized to overlap communication and computation in a PGAS execution model through a multi-buffering algorithm. The technique is enabled by the programming strategy of separating the logical expression of the algorithm (i.e., the programming model) from the physical expression of the algorithm (i.e., the execution model). This strategy facilitates the implementation of automatic parallelization and communications management in a high-level compiler, by removing the problem of reasoning about the physical details at the input and undoing the original programmer's (probably incorrect) physical bindings. There is a virtuous circle: removing physical details from the programming model makes it easier to automate their production for the execution model.

```
static void _matmult_PE(float (* A)[1024], float (* B)[1024],
                        float (* C)[1024], int PROC0) {
   float C_l[64][512] __attribute__((aligned(128)));
   float A_l[64][16] __attribute__((aligned(128)));
   float B_l[16][512] __attribute__((aligned(128)));
   int i;
   for (i = 0; i <= 1; i++) {
      int j;
      for (j = 0; j <= 1; j++) {
         int k;
         int k_1;
         for (k = 0; k <= 63; k++) {
            int i1;
            for (i1 = 0; i1 <= 511; i1++) {
               C_l[k][i1] = 0.0f;
         }  }
         for (k_1 = 0; k_1 <= 63; k_1++) {
            int i1;
            int i1_1;
            int i1_2;
            int i1_3;
            CELL_dma_get(
                  &B[16 * k_1][512 * j],
                  &B_l[0][0],
                  512*4,  // bytes
                  1024*4, // src stride
                  512*4,  // target stride
                  16,     // count
                  0);     // tag (for the wait)
            CELL_dma_get(
             &A[512 * i + 64 * PROC0][16 * k_1],
                  &A_l[0][0],
                  16*4,   // bytes
                  1024*4, // src stride
                  16*4,   // target stride
                  64,     // count
                  0);     // tag (for the wait)
            CELL_dma_wait(0);
            for (i1_2 = 0; i1_2 <= 63; i1_2++) {
               int j1;
               for (j1 = 0; j1 <= 511; j1++) {
                  int k1;
                  for (k1 = 0; k1 <= 15; k1++) {
                     C_l[i1_2][j1] = C_l[i1_2][j1] +
                        A_l[i1_2][k1] * B_l[k1][j1];
            }  }  }
            CELL_dma_put(
                  &C_l[0][0],
                  &C[512 * i + 64 * PROC0][512 * j],
                  512*4,  // bytes
                  512*4,  // src stride
                  1024*4, // target stride
                  64,     // count
                  1);     // tag (for the wait)
            CELL_dma_wait(1);
}  }  }  }
```

```
static void _matmult_PE(float (* A)[1024], float (* B)[1024],
                        float (* C)[1024], int PROC0) {

   float C_l[64][256] __attribute__((aligned(128)));
   float A_l_buf[2][64][16] __attribute__((aligned(128)));
   float B_l_buf[2][16][256] __attribute__((aligned(128)));

   float (*A_l)[16], (*A_l_1)[16];
   float (*B_l)[256], (*B_l_1)[256];
   A_l   = A_l_buf[0]; A_l_1 = A_l_buf[1];
   B_l   = B_l_buf[0]; B_l_1 = B_l_buf[1];

   int i;
   for (i = 0; i < 2; i++) {
      int j;
      for (j = 0; j < 4; j++) {
         int k;
         for (k = 0; k < 64; k++) {
            int i1;
            for (i1 = 0; i1 < 256; i1++) {
               C_l[k][i1] = 0.0f;
         }  }
         for (k = -1; k < 64; k++) {
            int i1;
            if (k >= 0) {
               float (* _t1)[16];
               float (* _t2)[256];
               _t1 = A_l; A_l = A_l_1; A_l_1 = _t1; // buffer
               _t2 = B_l; B_l = B_l_1; B_l_1 = _t2; // rotation
               CELL_dma_wait(0);
            }
            if (k < 63) {
               CELL_dma_get(
                     &B[16 * (k + 1)][256 * j],
                     &B_l_1[0][0],
                     256*4,  // bytes
                     1024*4,  // src stride
                     256*4,  // target stride
                     16,    // count
                     0);

               CELL_dma_get(
                     &A[512 * i + 64 * PROC0][16 * (k + 1)],
                     &A_l_1[0][0],
                     16*4,  // bytes
                     1024*4,  // src stride
                     16*4,  // target stride
                     64,    // count
                     0);
            }
            if (k >= 0) {
               for (i1 = 0; i1 < 64; i1++) {
                  int j1;
                  for (j1 = 0; j1 < 256; j1++) {
                     int k1;
                     for (k1 = 0; k1 < 16; k1++) {
                        C_l[i1][j1] = C_l[i1][j1] +
                           A_l[i1][k1] * B_l[k1][j1];
            }  }  }  }  }
            CELL_dma_put(
                  &C_l[0][0],
                  &C[512 * i + 64 * PROC0][256 * j],
                  256*4,   // bytes
                  256*4,   // src stride
                  1024*4,  // target stride
                  64,      // count
                  1);

            CELL_dma_wait(1);
} }  }
```

Fig. 5. Multi-buffered matrix multiply

5 Acknowledgements

We are grateful for the support from programs managed by DARPA, AFRL, NRL, DOE, and other agencies, contracts F30602-03-C-0033, DE-FG02-08ER85149, FA8650-08-M-1428, W9113M-08-C-0146, W31P4Q-08-C-0319 and FA8750-08-C-0229.

References

[BBK+07] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformation for communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[CBD+03] W. Chen, D. Bonachea, J. Duell, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *17th Annual International Conference on Supercomputing (ICS)*, 2003.

[CCZ07] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programming and the Chapel language. *Journal of High Performance Computing Applications*, August 2007.

[cle07] *ClearSpeed Software Development Kit Introductory Programming Manual*, July 2007.

[CM00] Ph. Clauss and B. Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. *ACM SIGARCH, Computer Architecture News*, 28(1), 2000.

[DKW+09] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[GGHvdG01] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[Gro09] Khronos OpenCL Working Group. The openCL specification (version 1.0), 2009.

[KA02] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[KBC+08] Peter M. Kogge, Shekhar Borkar, William W. Carlson, William J. Dally, Monty Denneau, Paul D. Franzon, Stephen W. Keckler, Dean Klein, Robert F. Lucas, Steve Scott, Allan E. Snavely, Thomas L. Sterling, R. Stanley Williams, Katherine A. Yelick, William Harrod, Daniel P. Campbell, Kerry L. Hill, Jon C. Hiller, Sherman Karp, Mark A. Richards, and Alfred J. Scarpelli. Exascale Study Group: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[Kuc96] David J. Kuck. *High Performance Computing.* Oxford University Press, 1996.

[LL97] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214, Paris, France, 1997.

[LLM+08] Richard Lethin, Allen Leung, Benoît Meister, Peter Szilagyi, Nicolas Vasilache, and David Wohlford. Final report on the the R-Stream 3.0 compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008.

[MV08] Benoit Meister and Sven Verdoolaege. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems*, Apr 2008.

[QRW00] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[VBC06] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, lncs, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.