Extended Static Control Programs as a Programming Model for Accelerators

A Case Study: Targetting ClearSpeed CSX700 with the R-Stream Compiler

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

Abstract

Classical compiler technologies are failing to bridge the gap between high performance and productivity in the multicore and massively parallel architecture era. This results in two complementary trends. Firstly, the rise of many new languages to allow the user to conveniently express parallelism. However, those languages may be target-specific like, e.g., CUDA or C^n, and they leave to the programmer the complex task of extracting parallelism even if expressing it may be more convenient using, e.g., Chapel or X10. Secondly, new high-level abstractions like SPIRAL allow performance portability at the price of versatility. In this paper we discuss an alternative way. We consider a restricted class of classical programs known as Extended Static Control Programs as a programming model. We show that using simple rules while writing sequential programs with usual languages like C, our R-Stream® High-Level Compiler can achieve complex mappings to parallel architectures in a fully automatic way. We present our effort to target ClearSpeed CSX700 architecture with specific contributions to avoid costly control overhead.

1 Introduction

Accelerators aim at providing new hardware features to perform some specific functions much faster or at a much lower energy cost than possible for general purpose processors. Main examples of hardware accelerators are *graphics accelerators* dedicated to manipulating and displaying computer graphics, *floating-point accelerators* designed to issue at a high rate operations on floating point numbers, and FPGAs that can be configured to accelerate a given processing. While those accelerators may provide significant additional computing power, they also require a significant additional work for engineers to learn and use their specific application programming interfaces (APIs) or languages. In this paper, we present a compiler technology that provides high productivity, allowing the programmers to focus on their high-level problems rather than understanding deep language or architecture features. This productivity is achieved provided the programmer writes the most computation intensive parts of its application according to the *Static Control Program* paradigm.

To achieve a reasonnable amount of the peak performance on a given architecture, a programmer may choose (1) to learn the details and specific language layers of the target architecture, or (2) to rely on an optimizing compiler, or (3) to use high level abstractions and tools like SPIRAL [8] to automatically generate a target code. The drawbacks of the first approach are well known: high learning curve, slow and error-prone programming and lack of portability. The second solution raises the problem of the availability of a good enough optimizing compiler, usually several years after a given architecture is available. The third solution requires to learn a new abstraction model and is typically domain-specific (as SPIRAL for DSP algorithms).

We investigate an alternative approach based on the Polyhedral Model, which spans to both optimizing compilers and high-level abstractions. This model is a well known high-level abstraction that allowed many advances in automatic program optimization and parallelization [5, 1, 7]. We present the R-Stream High-Level Compiler, that builds over this model to allow the programmers to write their code directly in a well known language (R-Stream supports C for pragmatic, user convenience, reasons but there is no pratical limitations to the input language). We rely on the compiler infrastructure *to automatically raise the abstraction level* and *to achieve efficient mappings to supported target architectures*. We show that to benefit from the full power of this compiler, the programmers have to use a subset of imperative languages called Static Control Programs as much as possible, i.e., relying on regular loops and array access constructs.

In this paper, we describe our effort to target the ClearSpeed CSX700 floating point accelerator [3] directly from

sequential loop nests in C. This architecture shows the typical challenges that arise when mapping for other accelerators, e.g., GPGPUs. Those massively parallel architectures expose many levels of mapping, deep memory hierarchy with explicit data transfers, and multiple API/language layers. We demonstrate that it is possible to build over our polyhedral framework to provide an optimizing compiler infrastructure for such architecture and to offer high productivity to programmers. We show that managing control overhead is a priority in this architecture and present contributions to minimize it.

This document is organized as follows. In Section 2, we present the Extended Static Control Programs, in Section 3, we present the main characteristics of the CSX700 architecture and its programming model. Section 4 describes the different elements used by the R-Stream compiler to map a sequential C code to those architecture and programming model. In Section 5 we discuss solutions to minimize control overhead for this target before concluding in Section 6.

2 Extended Static Control Programs

Static Control Programs are a sub-class of imperative programs that includes imperfectly nested loops such that:

- the loop counters are integers, and they are incremented by constant steps,
- the loops are bounded, and each bound must be an affine expression of the value of the outer loop counters and of the parameters (constant values that are unknown at compile time, e.g., array sizes),
- the statements enclosed within the loops access multi-dimensional arrays (this includes scalars and one-dimensional arrays) 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. We use this property to automatically raise such programs to the Polyhedral Model [5]. R-Stream focuses on program parts that respect this definition as well as extensions to data-dependent control flow (data-dependent conditionals, useful for, e.g., saturation arithmetics or pivoting) [7]. However, relying on irregular extensions may reduce the amount of extracted parallelism.

3 CSX700 Architecture and Programming

The ClearSpeed CSX700 processor is a floating-point accelerator designed to enhance the global computing power of a system at a very low energy cost (its energy efficiency for full precision is 3.84 Gflops per Watt, it may be compared to the 1.11 of IBM PowerXCell 8i or the 0.42 of the nVIDIA Tesla C1060). Its very low power dissipation footprint makes it suitable for high-performance embedded systems as well as power-efficient supercomputers. However, this processor shows specific architecture and programming models a programmer or a high-level compiler must address to generate efficient mappings.

3.1 A Three-Levels Mapping Architecture

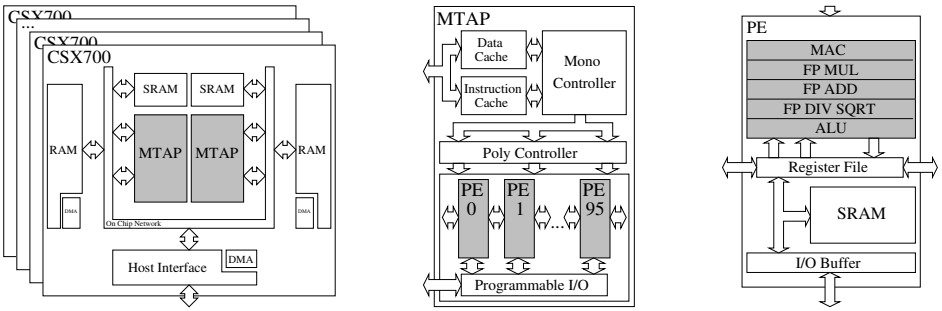
CSX's power-efficient floating-point acceleration results from a massively parallel architecture. In this way, even using a low frequency (hence saving energy), the processor is able to issue a large number of operations every clock cycle. The CSX700 processor is designed to take advantage of multiple levels of parallelism, each corresponding to a specific *mapping* level for the compiler. We can describe the CSX700 accelerator as a three-levels mapping architecture:

1. **System Level.** Because the CSX700 is an accelerator, it has been designed in such a way it is possible to add many accelerators to a host system to improve its performance. Furthermore, the CSX700 chipset is a two core processor. Each core has a 1GB memory. This is depicted in Figure 1(a). Hence a first level of mapping is to provide tasks to all the accelerator cores. We refer to it as the *System Mapping Level*.

2. **Processor Level.** Each CSX700 core, related as either *mono* core or *MTAP* core (for Multi-Threaded Array Processor), is a SIMD engine driving 96 Processing Elements (PEs) called *poly* processors. The poly processors share the same control flow (including predication that may disable some of them temporarily), managed by the mono core. Each PE has a 6KB local memory. Figure 1(b) depicts the MTAP core architecture. Thus, keeping the 96 poly processors busy with different data with a mono control flow is the second level of mapping. We refer to it as the *Processor Mapping Level*.

3. **Pipeline Level.** Each poly processor may issue two double floating-point operations every cycle thanks to its multiply-accumulate functional unit. The poly processor architecture is summarized in Figure 1(c). However, to achieve the best use of its pipeline, we must use *short-vector operations* that pack 4 operations together, hence, we need to benefit from an additional level of parallelism to achieve the vectorization, known in the litterature as *SIMDization*. We refer to SIMDization as the *Pipeline Mapping Level*.

In this document, we only address the processor and pipeline levels. We consider the program data have been



(a) System Level (b) Processor Level (c) Pipeline Level

Figure 1. The Three Levels of Mapping of the CSX700 Architecture

migrated to the device memory of a given mono core. System level mapping, in which the program's data is in the main memory of a system and has to be explicitly transferred to and from the accelerator, is a generalization of the work discussed in this paper.

3.2 Programming Model

Each mapping level described in Section 3.1 requires a specific language or language extension at the output of R-Stream. The system level is programmed in C with an API called CSAPI for managing data transfers between host and mono memories and for managing computing tasks to be run on mono-cores [4]. The processor level is programmed using a new language called C^n [2]. The pipeline level relies on vector types and ultimately on ClearSpeed assembly for the best performance [3].

The C^n language is a programming language based on C that expresses parallelism in an implicit way, depending on the type of the operands. C^n introduces two new type attributes: `mono` and `poly`. Variables with the type attribute `mono` are declared in the mono memory space while variables with the type attibute `poly` are declared in the poly memory (there exists one version of this variable for each poly processor). When an instruction has to be executed, if all its operands are `mono` then the instruction is executed on the mono processor. If one of the operands is `poly`, then the operation is run on all the poly processor (`mono` variables are broadcasted to poly processors if necessary). Hence both mono and poly codes are intimately mixed in a single source code.

4 CSX700 Mapping in R-Stream

R-Stream is Reservoir Lab's proprietary compiler. DARPA funded development of R-Stream by Reservoir between 2003 and 2007 in the Polymorphous Computing Architectures (PCA) Program [7]. It differs from most existing compilers for three main reasons:

1. Its input includes a **Hierarchical Machine Model** as well as an input code. This machine model describes, thanks to a set of XML files, the critical features of the target architecture (e.g., number of PEs, sizes and types of the memories, SIMD widths, relative communication layer speeds etc.), in such a way that the target architecture is described in a precise enough way for generating an efficient mapping as well as providing an easy way for the compiler to target new architectures.

2. It is a **High-Level Compiler**, that does not output an object code directly but another source code with high-level constructs (OpenMP pragmas, threads, C^n parallelism, DMA calls, vector operations etc.) to be compiled by a low-level compiler, ensuring portability and benefiting from low-level optimizations of the target compilers.

3. It is based on the **Polyhedral Model**, an algebraic representation that allows deep and complex code restructuring while ensuring the original program semantics is not altered. The polyhedral representation is modified to map the target architecture along several *phases* that may be shared by every targets or dedicated to a given subset of the targets.

A high-level view of the R-Stream compiler when targetting CSX700 architecture is shown in Figure 2. The input code is first parsed then translated (if possible, see Section 2) in the internal *polyhedral* representation, then it is modified along several phases in the *Polyhedral Mapper* before being translated back to a C^n source code that will be compiled by the low-level ClearSpeed `cscn` compiler (`cscn` has no automatic parallelization facility).

To add the support of the CSX700 architecture in R-Stream, we need a Machine Model that describes this architecture. Next, we need to be able to output efficient

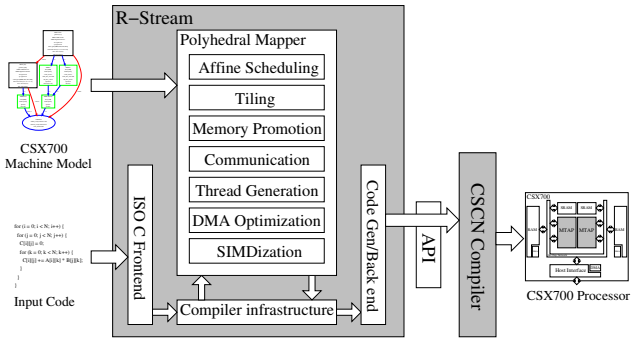


Figure 2. R-Stream for CSX700 Mapping

high-level C^n codes thanks to existing (and possibly new dedicated) compiler phases and an extended code generation scheme. Our strategy to map sequential C programs to a mono core of the CSX700 processor is decomposed in the following phases:

1. Expose parallelism via *affine scheduling*. The first step of the mapping process is to expose all the available parallelism in the program, including both coarse-grained and fine-grained parallelism. Generally speaking, our strategy consists into transforming coarse-grained parallelism into C^n implicit threads, and fine-grained parallelism into SIMD parallelism within a thread block. We use a unified formulation of parallelism and fusion/distribution profitability to derive the optimal tradeoff between parallelism and locality. This formulation is a generalization of the work of Bondhugula et al. [1] and is out of the scope of this paper.

2. Use *tiling* (aka *blocking*)to divide the statement instances into blocks, such that each block fits within the constraints of a thread block (in particular, the data footprint of one block must not exceed the poly memory size) and such that the tile size balances the amount of computation and communication (between tiles).

3. Promote variables in mono memory to poly memory via *memory promotion* then generate communications to copy variables from mono memory to poly memory and conversely via *communication generation*.

4. Transform the explicit parallelism to implicit C^n-like parallelism via *thread generation*. To match the C^n programming model, the parallel loop that distributes tasks to the poly processor must be implicit as explained in Section 3.2.

5. Optimize data transfers by grouping memory communications via *DMA optimization*. We use a generalization of Schreiber et al. work [9] to the ensure only useful data are transferred.

6. Achieve pipeline-level mapping by extracting high-level vector operations via *SIMDization*. R-Stream investigates two complementary strategies for SIMDization. (1) *Short Vector SIMDization* aims at promoting accessed variables to a vector type that is supported by the architecture. This strategy provides the largest flexibility since any statement that satisfies the SIMD constraints may be promoted this way. However the provided performance highly depends on the low-level compiler. (2) *Long Vector SIMDization* aims at relying on a library function to achieve a long vector operation. This strategy offers the best performance as the library function may be written directly in efficient assembly code. However it is only possible to target vector statements in the restricted scope of the long-vector library.

The result of the whole mapping process is shown on a simple yet meaningful matrix-multiply kernel shown in Figure 3(a). An example target code without multi-buffering to hide memory latency is presented in Figure 3(b). This example shows that even for such simple program, complex and error-prone transformations have to be achieved to benefit from the architecture. First, we can notice the local arrays declared in the poly domain. Our Polyhedral Mapper computed iteration space tiles in such a way that the arrays can fit together in the poly memory. The statement `_t1 = get_penum()` stands for the implicit C^n parallel loop on the 96 PEs. Memory copy functions achieve asynchronous transfers and are part of the R-Stream runtime. Their first parameter is a *tag* used by the waiting functions to ensure the memory transfers have been processed before starting computations. Lastly, a long-vector instruction is used in the innermost loop to benefit from the pipeline mapping level.

The target code in Figure 3(b), while not including communication optimization runs at 1.86Gflops (or 2.65Gflops when using a simpler runtime, not suitable for further communication optimizations). While this may be a significant acceleration for the host system at a very low energy cost, this is still one order of magnitude from the best performance of the processor (48Gflops for one core). However, there exists significant rooms for performance improvement. First of all the communication generation scheme we discuss in this paper is still synchronous while a double or triple buffering may hide communication latency. We simulated a perfect communication scheme by removing all memory transfers statements and achieved 12.02Gflops. Second, some tuning about, e.g., the tile size should be done to better exploit the Pipeline Mapping Level. Using perfect communications and a tile size of $2 \times 2 \times 256$ reaches 16.49Gflops, hence we target 30% of the peak performance and 50% of the proprietary BLAS library performance (34Gflops).

```
#define N 1024
float A[N][N], B[N][N], C[N][N];
void kernel() {
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0;
      for (k = 0; k < N; k++)
        C[i][j] = C[i][j] + A[i][k] * B[j][k];
    }
  }
}
```

(a) Input Matrix-Multiply Kernel (B is transposed)

```
float      A[1024][1024], B[1024][1024], C[1024][1024];
float poly A_l[4][64], B_l[8][64], C_l[4][8];

void kernel(void) {
  int poly _t1, _t2;
  int i;
  dcache_flush();
  for (_t1 = get_penum(), _t2 = (__polymins_32(2, (-_t1 + 255) / 96)), i = 0; i <= _t2; i++) {
    int j;
    for (j = 0; j <= 127; j++) {
      int k, k_1;
      for (k = 0; k <= 3; k++) {
        int i1;
        for (i1 = 0; i1 <= 7; i1++)
          C_l[k][i1] = 0.0f;
      }
      for (k_1 = 0; k_1 <= 15; k_1++) {
        int i1, i1_1, i1_2, i1_3;
        for (i1 = 0; i1 <= 7; i1++)
          rstream_csx_memcpym2p(0, B_l[i1] + 0, B[8*j + i1] + 64*k_1, 64*4);
        for (i1_1 = 0; i1_1 <= 3; i1_1++)
          rstream_csx_memcpym2p(0, A_l[i1_1] + 0, A[384*i + (i1_1 + 4*_t1)] + 64*k_1, 64*4);
        rstream_csx_wait(0);
        for (i1_2 = 0; i1_2 <= 3; i1_2++) {
          int j1;
          for (j1 = 0; j1 <= 7; j1++)
            rstream_csx_vector_fmulacc_reduce(C_l[i1_2] + j1, A_l[i1_2] + 0, B_l[j1] + 0, 64);
        }
        for (i1_3 = 0; i1_3 <= 3; i1_3++)
          rstream_csx_memcpyp2m(1, C[384*i + (i1_3 + 4*_t1)] + 8*j, C_l[i1_3] + 0, 8*4);
        rstream_csx_wait(1);
      }
    }
  }
}
```

(b) Example of R-Stream Output for Matrix-Multiply Kernel (no multi-buffering)

Figure 3. Example of CSX700 Mapping of a Matrix Multiply Kernel

5 Control Overhead Minimization

The SIMD architecture model of the CSX700 has a significant impact on control overhead management. Automatic parallelization quite often generates a costly control overhead (complex `if` conditions or loop bounds) because sophisticated transformations have to be done to extract or to set the granularity of the parallelism. The tiling transformation we use is a good example: it generates from a loop nest with a given depth a new, deeper, loop nest with more complex bounds to ensure the global number of iterations does not change. Usually, unless positive cache effects occur, this reduces performance when the parallel code runs on a one-core target. This overhead disappears when targetting multi-core or multi-processor because the control is shared and the benefit of the parallelization hides its cost.

This is not true for SIMD engines like the CSX700. For each MTAP core, the whole control flow is computed by the mono processor only. Hence the control overhead is not shared between the various PEs as it would be in MIMD architectures like, e.g., IBM Cell. It follows that it is necessary to limit the global control overhead. This statement is even stronger when we rely on *predication*. To provide a maximum flexibility despite the single control flow for each MTAP core, the CSX700 architecture supports predication. This means some PEs may be deactivated and may temporarily not participate in the computation or the data transfers while others do. Each branch whose condition in-

volves a poly variable implies predicated statements. For instance a condition such as `if (get_penum() != 95)` disables the 95^{th} PE for all the instructions guarded by this condition. However this is quite costly. We measured that the penalty to use a poly `if/else` conditional is typically a factor 2 when both branches are balanced and slightly more costly than a classical condition when there is no `else` branch (we assume simple conditions). Relying on loops with a poly counter or a poly bound (referred as *poly loops*) is even more critical since it can impact performance up to a factor 4 if the loop is at the innermost level. We expose three solutions to avoid these situations, namely SIMD width reduction, domain simplification and full-tile extraction & normalization.

```
doall (l = 384*i+4*PROC0; l <= 384*i+4*PROC0+3; l++)
  doall (m = 8*j; m <= 8*j+7; m++)
    reduction_for (n = 64*k; n <= 64*k+63; n++)
      C_l[-384*i+l-4*PROC0][-8*j+m] +=
        A_l[-384*i+l-4*PROC0][-64*k+n] *
        B_l[-8*j+m][-64*k+n];
```

(a) Internal Kernel Without Simplification

```
doall (l = 0; l <= 3; l++)
  doall (m = 0; m <= 7; m++)
    reduction_for (n = 0; n <= 63; n++)
      C_l[l,m] += A_l[l,n] * B_l[m,n];
```

(b) Simplified Internal Kernel (no SIMDization)

Figure 4. Example of Transformation Simplification For Matrix-Multiply Kernel

5.1 SIMD Width Reduction

Our mapping strategy described in Section 4 aims at distributing the workload to all the PEs. Typically, predication is necessary when all the PEs do not achieve the same amount of work, i.e., when it is not possible to distribute the workload in exactly the same way between the processing elements. When the challenge comes from the amount of workload only, we can artificially reduce the SIMD width of the CSX700 cores in such a way that the workload may be perfectly distributed. This amounts simply to adding a global outermost poly condition that has nearly no impact on the processing time (no `else` branch). Hence in the case of the CSX700, reducing the number of working PEs may improve overall performance.

5.2 Transformation Simplification

Once a transformation has been computed to optimize a program, it is possible to find another, equivalent transformation, such that the relative order between the various iterations is not modified [10]. R-Stream uses this property to iteratively compute an equivalent transformations (based on compositions of shifting and skewing transformations) in order to simplify the subscript functions of the array accessed inside a loop. This phase relies on a cost model to find the best simplification. For ClearSpeed, because of the performance penalty, we extended the cost model to avoid expressions involving the PE number. The matrix-multiply example described in Section 4 already used this optimization. Without this optimization, the internal kernel would have been the one in Figure 4(a), where `PROC0` stands for the PE number. Thanks to the simplification, it can be transformed to the one in Figure 4(b).

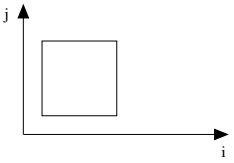
5.3 Full-Tile Extraction & Normalization

It may happen that the workload cannot be distributed in a perfect way between the PEs because it is not possible to build tiles all of the same shape. This may happen because either the original iteration domain is complex or it has to be transformed to extract parallelism. In such a case, a given PE may have to execute a tile of a different shape than other PEs. This phenomenon is illustrated in Figure 5. The figure shows a very simple polynomial multiplication kernel with a rectangular iteration domain (Figure 5(a)) that requires a transformation of its iteration space called *skewing* to expose parallelism (Figure 5(b)). Then we use the extracted parallelism to create blocks of workload to be distributed across the PEs thanks to the tiling transformation. The tiled code is shown in Figure 5(c) (for clarity reasons we keep the outer loop on the PEs visible and we do not show explicit memory transfers). The outermost loop on the iterator `i` is the parallel loop on the 96 PEs. Hence `i` is a poly variable and all expressions using `i` are poly expressions. It follows that due to a snowball effect, all loops in this tiled code are inefficient poly loops.

The reason for this situation is that the same code is used to perform the workload of the full tiles (that have a fixed size determined by the compiler) as well as the partial tiles. Because the compiler knows which loops iterate over the tiles and which loops actually achieve the workload inside a tile, it is possible to separate the processing of the full tiles and the partial tiles.

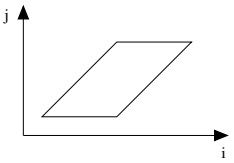
This separation is done at the code generation step [11]. For each statement, we start by determining a set of conditions (a lower bound and an upper bound) on each intra-tile dimension that ensures a constant number of iterations are spanned by the given intra-tile dimension. We perform this extraction for each intra-tile dimension. We subsequently perform projections to derive the necessary conditions exclusively on the inter-tile dimensions to derive a full tile. These necessary conditions are attached to each statement as a predicate at the start of polyhedral scanning. When the first inter-tile dimension is reached during scanning, the predicates for all statements are combined and sepa-

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    C[i+j] += A[i] * B[j];
  }
}
```



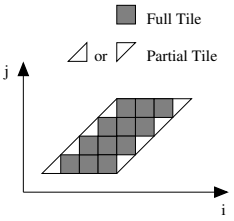
(a) Original Kernel with Rectangular Iteration Domain

```
doall (i = 0; i <= 2*n-2; i++) {
  for (j = max(0, i-n+1); j <= min(n-1, i); j++) {
    C[i] += A[j] * B[i - j];
  }
}
```



(b) Extraction of a Parallel Loop (`doall`) by Iteration Domain Skewing

```
doall (i = 0; i <= 95; i++) {
  doall (j = (95 - i) / 96;
         j <= floorDiv(-116 * i + n - 1, 11136); j++) {
    for (k = max((232*i + 22272*j - n + 1) / 152, 0);
         k <= min(floorDiv(n - 1, 152),
                  (29*i + 2784*j + 2821) / 19 - 147); k++) {
      doall (l = max(232*i + 22272*j, 152*k);
             l <= min(2*n + -2, 152*k + n + 150,
                      232*i + 22272*j + 231); l++) {
        for (m = max(152*k, l - n + 1);
             m <= min(n - 1, 152*k + 151, l); m++) {
          C[l] += A[m] * B[l - m];
        }
      }
    }
  }
}
```



(c) Tiling of 232x152 With an Outer Loop On the PEs Number

```
doall (i = 0; i <= 95; i++) {
  doall (j = (95 - i) / 96; j <= floorDiv(-52*i + n + -1, 4992); j++) {
    for (k = max(0, (104*i + 9984*j - n + 1) / 152) ;
         k <= min(floorDiv(n - 1, 152), (13*i + 1248*j + 1285) / 19 - 67); k++) {
      if (-104*i -9984*j + 152*k + n >= 104) {
        if (104*i + 9984*j - n >= -1 && -152*k + n >= 152) {
          doall (l = 0; l <= 103; l++) {
            for (m = 0; m <= 151; m++) {
              C[l - 104*i - 9984*j] += A[m - 152 * k] * B[l - 104*i - 9984*j - m - 152 * k];
            }
          }
        }
        if (-104*i -9984*j + n >= 2 && 13*i + 1248*j -19*k >= 19) {
          doall (l = 0; l <= 103; l++) {
            for (m = 0; m <= 151; m++) {
              C[l - 104*i - 9984*j] += A[m - 152*k] * B[l - 104*i - 9984*j - m - 152*k];
            }
          }
        }
      }
      if (!(-152*k + n >= 152 && -104*i -9984*j + 152*k + n >= 104 &&
            104*i + 9984*j - n >= -1 || 13*i + 1248*j - 19*k >= 19 &&
           -104*i - 9984*j + n >= 2 && -104*i -9984*j + 152*k + n >= 104)) {
        doall (l = max(152*k, 104*i + 9984*j);
               l <= min(104*i + 9984*j + 103, 2*n - 2, 152*k + n + 150); l++) {
          for (m = max(l - n + 1, 152*k); m <= min(n - 1, 152*k + 151, l); m++) {
            C[l] += A[m] * B[l - m];
          }
        }
      }
    }
  }
}
```

(d) Code After Full Tile Extraction & Normalization

Figure 5. Full Tile Extraction & Normalization for Polynomial Multiplication

rated. The combined predicates are intersected into each statement's domain and a clone of the statement is created with the negated predicate. Polyhedral domain simplification occurs to cleanup redundancies. After this transformation, the trip count of every intra-tile loop is constant but their bounds may still depend on the PE number to reflect their position in the original iteration space. A last step of normalization to 0 moves the offset to the statement expressions and allows the intra-tile loops to be non-poly.

Applying this code generation scheme to our example leads to the code in Figure 5(d). In this code, there are two parts. The first one corresponds to the full tiles where the inner loops (intra-tile) have constant bounds and are non poly. The second part is devoted to scanning the partial tiles and have poly intra-tile loops. Unfortunately because of the normalization, the array subscripts in the statements are more complex. However (without other positive memory transfer effects), the final code in Figure 5(d) is 30% faster than the one in Figure 5(c). The construction of full-tiles has already been used, for instance when considering parametric tiling [6], however, to the best of our knowledge, this is the first time a solution is proposed as a code generation step in the Polyhedral Model and applied to the control overhead minimization problem.

6 Conclusion

The learning curve to achieve good performance using accelerators may be high because of the need to deeply understand their architectures and their specific languages. ClearSpeed accelerators are no exceptions as in addition to the architecture properties, three abstraction levels have to be learned: CSXAPI, C^n and vector intrinsics. Ultimately, assembly may be necessary to achieve the best performance. To bridge the gap between the power-efficient, high performance potential of the CSX700 architecture and the user needs, the availability of an optimizing compiler is necessary.

R-Stream is a multi-platform high-level compiler that, thanks to machine models and possibly some adaptations of a few phases, may be extended to target new architectures in a much faster way than designing a new target-specific optimizing compiler from scratch. In this preliminary work to target ClearSpeed CSX700, in addition to the necessary new back end to support C^n, only two phases have been modified (Memory Promotion and Thread Generation) and one created (SIMDization) in the Polyhedral Mapper. The current state of the implementation does not perform as well as the libraries, but it demonstrates the feasibility of automating the difficult transformations from sequential C source to C^n and ClearSpeed vector intrinsics, potentially achieving a reasonable percentage of peak performance and providing maximum flexibility to programmers. This productivity is granted to the users as long as they conform as much as possible to the Static Control Program paradigm, hence with no need for learning at new architecture nor new language.

Ongoing work on supporting ClearSpeed CSX700 focuses on communications, in particular to support multi-buffering and the swazzle path that allows near-neighbor communications for PEs.

References

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

[2] ClearSpeed Inc. *Cn Standard Libraries Reference Manual, Document 06-RM-1139*. 2008.

[3] ClearSpeed Inc. *CSX700 Floating Point Processor Datasheet, Document 06-PD-1425*. 2008.

[4] ClearSpeed Inc. *Software Development Kit Reference Manual, Document 06-RM-1600*. 2008.

[5] P. Feautrier. Some efficient solutions to the affine scheduling problem. part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[6] A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the ACM International Conference on Supercomputing (ICS'09)*, pages 147–157, Yorktown Heights, New York, June 2009.

[7] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. 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.

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[9] R. Schreiber and D. C. Cronquist. Near-optimal allocation of local memory arrays. Technical Report HPL-2004-24, Hewlett-Packard Laboratories, Feb. 2004.

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

[11] 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.