Automatic Generation of FPGA-Specific Pipelined Accelerators

Christophe Alias, Bogdan Pasca, and Alexandru Plesco

LIP (ENSL-CNRS-Inria-UCBL), École Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
Firstname.Lastname@ens-lyon.fr

Abstract. Recent increase in the complexity of the circuits has brought high-level synthesis tools as a must in the digital circuit design. However, these tools come with several limitations, and one of them is the efficient use of pipelined arithmetic operators. This paper explains how to generate efficient hardware with floating-point pipelined operators for regular codes with perfect loop nests. The part to be mapped to the operator is identified, then the program is scheduled so that each intermediate result is produced exactly at the time it is needed by the operator, avoiding pipeline stalling and temporary buffers. Finally, we show how to generate the VHDL code for the control unit and how to link it with specialized pipelined floating-point operators generated using the open-source FloPoCo tool. The method has been implemented in the Bee research compiler and experimental results on DSP kernels show promising results with a minimum of 94% efficient utilization of the pipelined operators for a complex kernel.

1 Introduction

Application development tends to pack more features per product. In order to cope with competition, added features usually employ complex algorithms, making full use of existing processing power. When application performance is poor, one may envision accelerating the whole application or a computationally demanding kernel using the following solutions: (1) multi-core general purpose processor (GPP): may not accelerate non-standard computations (exponential, logarithm, square-root) (2) application-specific integrated circuit (ASIC): the price tag is often too big, (3) Field Programmable Gate Array (FPGA): provide a balance between the performance of ASIC and the costs of GPP.

FPGAs have a potential speedup over microprocessor systems that can go beyond two orders of magnitude, depending on the application. Usually, such accelerations are believed to be obtained only using low-level languages as VHDL or Verilog, exploiting the specificity of the deployment FPGA. However, designing entire systems using these languages is tedious and error-prone.

In order to address the productivity issue, much research has focused on high-level synthesis (HLS) tools [25, 2, 10, 1, 7], which input the system description in higher level language, such as `C` programming language (C). Unfortunately, so far none of these tools come close to the speedups obtained by manual design. Moreover, these tools have important data type limitations.

In order to take advantage of the hardware carry-chains (for performing fast additions) and of the Digital Signal Processing (DSP) blocks (for performing fast multiplications) available in modern FPGAs, most HLS tools use fixed-point data types for which the operations are implemented using integer arithmetic. Adapting the fixed-point format of the computations along the datapath is possible, but requires as much expertise as expressing the computational kernel using VHDL or Verilog for a usually lower performance kernel. Keeping the same fixed-point format for all computations is also possible, but in this case either the design will overflow/underflow if the format is too small, either will largely overestimate the optimal circuit size when choosing a large-enough format.

For applications manipulating data having a wide dynamic range, HLS tools supporting standard floating-point precisions [10], or even custom precisions can be used [1]. Floating-point operators are more complex than their fixed-point counterparts. Their pipeline depth may count tens of cycles for the same frequency for which the equivalent fixed-point operator require just one cycle. Current HLS tools make use the pipelined FP operators cores in a similar fashion as for combinatorial operators, but employing stalling whenever feedback loops exists. This severely affects performance.

In this paper, we describe an automatic approach for synthesizing a specific but wide class of applications into fast FPGA designs. This approach accounts for the pipeline depth of the operator and uses state of the art code transformation techniques for scheduling computations in order to avoid pipeline stalling. We present here two classic examples: matrix multiplication and the Jacobi 1D relaxation for which we describe the computational kernels, code transformations and provide synthesis results. For these applications, simulation results show that our scheduling is within 5% of the best theoretical pipeline utilization.

The rest of this paper is organized as follows. Section 2 presents related approaches and their limitations. Section 3 presents FloPoCo, the tool used to generate efficient floating-point pipelined operators. Then, Section 4 shows how to compile a kernel written in C into efficient hardware with pipelined operators. For this, Subsection 4.2 studies two important running examples. Then, Subsections 4.3 and 4.4 provide a formal description of our method. Section 5 provides experimental results on the running examples. Finally, Section 6 concludes and presents research perspectives.

2 Related Work

In the last years, important advances have been made in the generation of computational accelerators from higher-level of abstraction languages. Many of these languages are limited to C-like subsets with additional extensions. The more restrictive the subset is, the more limited is the number of applications.

For example, Spark [22] can only synthesize integer datatypes, and is thus restricted to a very narrow application class.

Tools like Gaut [25], Impulse-C [2], Synphony [7] require the user to convert the floating-foint (FP) specification into a user-defined fixed-point format. Other, like Mentor Graphics' CatapultC [5], claim that this conversion is done automatically. Either way, without additional knowledge on the ranges of processed data, the determined fixed-point formats are just estimations. Spikes in the input data can cause overflows which invalidate large volumes of computations.

In order to workaround the known weaknesses of fixed-point arithmetic, AutoPilot [10] and Cynthesizer [1] (in SystemC) can synthesize FP datatypes by instantiating FP cores within the hardware accelerator. AutoPilot can instantiate IEEE-754 Single Precision (SP) and Double Precision (DP) standard FP operators. Cynthesizer can instantiate custom precision FP cores, parametrized by exponent and fraction width. Moreover, the user has control over the number of pipeline stages of the operators, having an indirect knob on the design frequency. Using these pipelined operators requires careful scheduling techniques in order to (1) ensure correct computations (2) prevent stalling the pipeline for some data dependencies. For algorithms with no data dependencies between iterations, it is indeed possible to schedule one operation per cycle, and after an initial pipeline latency, the arithmetic operators will output one result every cycle. For other algorithms, these tools manage to ensure (1) at the expense of (2). For example, in the case of algorithms having inter-iteration dependencies, the scheduler will stall successive iterations for a number of cycles equal to the pipeline latency of the operator. As said before, complex computational functions, especially FP, can have tens and even hundreds of pipeline stages, therefore significantly reducing circuit performance.

In order to address the inefficiencies of these tools regarding synthesis of pipelined (fixed or FP) circuits, we present an automation tool chain implemented in the Bee research compiler [8], and which uses FloPoCo [17], an open-source tool for FPGA-specific arithmetic-core generation, and advanced code transformation techniques for finding scheduling which eliminates pipeline stalling, therefore maximizing throughput.

3 FloPoCo - a tool for generating computational kernels

Two of the main factors defining the quality of an arithmetic operator on FPGAs are its *frequency* and its *size*. The frequency is determined by the length of the *critical path* – largest combinatorial delay between two register levels. Faster circuits can be obtained by iteratively inserting register levels in order to reduce the critical path delay. Consequently, there is a strong connection between the circuit frequency and its size.

Unlike other core generators [3, 4], FloPoCo takes the target frequency f as a parameter. As a consequence, complex designs can easily be assembled from subcomponents generated for frequency f. In addition, the FloPoCo operators are also optimized for several target FPGAs (most chips from Altera and Xilinx), making it easy to retarget even complex designs to new FPGAs.

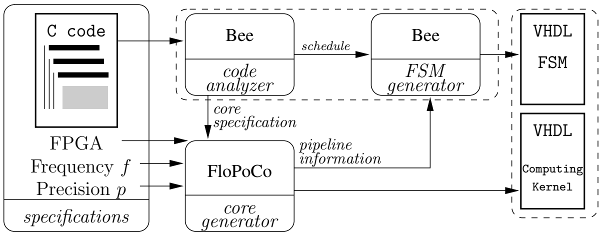


Fig. 1. Automation flow

However, FloPoCo is more than a generator of frequency-optimized standard FP operators. It also provides:

- operators allowing *custom precisions*. In a microprocessor, if one needs a precision of 10bits for some computation it makes sense using single-precision (8-bit exponent, 23-bit fraction) for this computation. In an FPGA one should use custom operators (10-bit fraction), yielding smaller operators and therefore being able to pack more in parallel.
- *specialized operators*, as: squarers, faithful multipliers[1], FPGA-specific FP accumulators [19].
- *elementary functions*, as: square-root [16], logarithm [15], exponential [18] which are implemented in software in processors (and are thus slow).
- dedicated architectures for *coarser operators* which have to be implemented in software in processors, for example $X^2 + Y^2 + Z^2$, and others. [17].

Part of the recipe for obtaining good FPGA accelerations for complex applications is: (a) use FPGA-specific operators, for example those provided by FloPoCo (b) exploit the application parallelism by instantiating several computational kernels working in parallel (c) generate an application-specific finite state machine (FSM) which keeps the computational kernels as busy as possible.

In the following sections we present an automatic approach for generating computational-kernel specific FSMs. Figure 1 presents the automation datapath.

4 Efficient Hardware Generation

This section presents the main contribution of this paper. Given an input program written in C and a pipelined FloPoCo operator, we show how to generate an equivalent hardware accelerator using cleverly the operator. This process is divided into two steps. First, we schedule the program so the pipelined operator is fed at every cycle, avoiding stalls. Then, we generate the VHDL code to control the operator with respect to the schedule. Section 4.1 defines the required terminology, then Section 4.2 explains our method on two important examples. Finally, Sections 4.3 and 4.4 present the two steps of our method.

[1] have and error of 1ulp, while standard multipliers have 0.5ulp, but consume much less resources

4.1 Background

Iteration domains. A *perfect loop nest* is an imbrication of `for` loops where each level contains either a single `for` loop or a single assignment S. A typical example is the matrix multiply kernel given in figure 2(a). Writing $i_1, ..., i_n$ the loop counters, the vector $\boldsymbol{i} = (i_1, ..., i_n)$ is called an *iteration vector*. The set of iteration vectors $\boldsymbol{i}$ reached during an execution of the kernel is called an *iteration domain* (see figure 2(b)). The execution instance of S at the iteration $\boldsymbol{i}$ is called an *operation* and is denoted by the couple $(S, \boldsymbol{i})$. We will assume a single assignment in the loop nest, so we can forget S and say "iteration" for "operation". The ability to produce program analysis at the *operation level* rather than at *assignment level* is a key point of our automation method. We assume loop bounds and array indices to be an *affine expression* of the surrounding loop counters. Under these restrictions, the iteration domain $\mathcal{I}$ is an invariant polytope. This property makes possible to design a program analysis by means of integer linear programming (ILP) techniques.

Dependence vectors. A data dependence is *uniform* if it occurs from the iteration $\boldsymbol{i}$ to the iteration $\boldsymbol{i} + \boldsymbol{d}$ for every valid iterations $\boldsymbol{i}$ and $\boldsymbol{i} + \boldsymbol{d}$. In this case, we can represent the data dependence with the vector $\boldsymbol{d}$ that we call a *dependence vector*. When array indices are themselves uniform (*e.g. a[i-1]*) all the dependencies are uniform. In the following, we will restrict to this case and we will denote by $\mathcal{D} = \{\boldsymbol{d}_1, \ldots \boldsymbol{d}_p\}$ the set of dependence vectors. Many numerical kernels fit or can be restructured to fit in this model [11]. This particularly includes stencil operations which are widely used in signal processing.

Schedules and hyperplanes. A *schedule* is a function θ which maps each point of $\mathcal{I}$ to its execution date. Usually, it is convenient to represent execution dates by integral vectors ordered by the lexicographic order: $\theta : \mathcal{I} \to (\mathbb{N}^q, \ll)$. We consider *linear schedules* $\theta(\boldsymbol{i}) = U\boldsymbol{i}$ where U is an integral matrix. If there is a dependence from an iteration $\boldsymbol{i}$ to an iteration $\boldsymbol{j}$, then $\boldsymbol{i}$ must be executed before $\boldsymbol{j}$: $\theta(\boldsymbol{i}) \ll \theta(\boldsymbol{j})$. With uniform dependencies, this gives $U\boldsymbol{d} \gg 0$ for each dependence vector $\boldsymbol{d} \in \mathcal{D}$. Each line $\boldsymbol{\phi}$ of U can be seen as the normal vector to an affine hyperplane $H_{\boldsymbol{\phi}}$, the iteration domain being scanned by translating the hyperplanes $H_{\boldsymbol{\phi}}$ in the lexicographic ordering. An hyperplane $H_{\boldsymbol{\phi}}$ *satisfies* a dependence vector $\boldsymbol{d}$ if by translating $H_{\boldsymbol{\phi}}$ in the direction of $\boldsymbol{\phi}$, the source $\boldsymbol{i}$ is touched before the target $\boldsymbol{i} + \boldsymbol{d}$ for each $\boldsymbol{i}$, that is if $\boldsymbol{\phi}.\boldsymbol{d} > 0$. We say that $H_{\boldsymbol{\phi}}$ *preserves* the dependence $\boldsymbol{d}$ if $\boldsymbol{\phi}.\boldsymbol{d} \geq 0$ for each dependence vector $\boldsymbol{d}$. In that case, the source and the target can be touched at the same iteration. $\boldsymbol{d}$ must then be solved by a subsequent hyperplane. We can always find an hyperplane $H_{\boldsymbol{\tau}}$ *satisfying* all the dependencies. Any translation of $H_{\boldsymbol{\tau}}$ touches in $\mathcal{I}$ a subset of iterations which can be executed in parallel. In the literature, $H_{\boldsymbol{\tau}}$ is usually refereed as a *parallel hyperplane*.

Loop tiling. With loop tiling [23, 27], the iteration domain of a loop nest is partitioned into parallelogram tiles, which are executed atomically. A first tile is executed, then another tile, and so on. For a loop nest of depth n, this requires to generate a loop nest of depth $2n$, the first n *inter-tile* loops describing the different tiles and the next n *intra-tile* loops scanning the current tile. A *tile* *slice* is the 2D set of iterations described by the last two intra-tile loops for a given value of outer loops. See figure 2 for an illustration on the matrix multiply example. We can specify a loop tiling for a perfect loop nest of depth n with a collection of affine hyperplanes $(H_1, \dots, H_n)$. The vector $\boldsymbol{\phi}_k$ is the normal to the hyperplane H_k and the vectors $\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_n$ are supposed to be linearly independent. Then, the iteration domain of the loop nest can be tiled with regular translations of the hyperplanes keeping the same distance ℓ_k between two translation of the same hyperplane H_k. The iterations executed in a tile follow the hyperplanes in the lexicographic order, it can be view as "tiling of the tile" with $\ell_k = 1$ for each k. A tiling $\mathcal{H} = (H_1, \dots, H_n)$ is *valid* if each normal vector $\boldsymbol{\phi}_k$ preserves all the dependencies: $\boldsymbol{\phi}_k.\boldsymbol{d} \geq 0$ for each dependence vector $\boldsymbol{d}$. As the hyperplanes H_k are linearly independent, all the dependencies will be satisfied. The tiling $\mathcal{H}$ can be represented by a matrix $U_\mathcal{H}$ whose lines are $\boldsymbol{\phi}_1, \dots \boldsymbol{\phi}_n$. As the intra-tile execution order must follow the direction of the tiling hyperplanes, $U_\mathcal{H}$ also specifies the execution order for each tile.

Dependence distance. The *distance* of a dependence $\boldsymbol{d}$ at the iteration $\boldsymbol{i}$ is the number of iterations executed between the source iteration $\boldsymbol{i}$ and the target iteration $\boldsymbol{i}+\boldsymbol{d}$. Dependence distances are sometimes called *reuse distances* because both source and target access the same memory element. It is easy to see that *in a full tile*, the distance for a given dependence $\boldsymbol{d}$ does not depend on the source iteration $\boldsymbol{i}$ (see figure 3(b)). Thus, we can write it $\Delta(\boldsymbol{d})$. However, the program schedule can strongly impact the dependence distance. There is a strong connection between dependence distance and pipeline depth, as we will see in the next section.

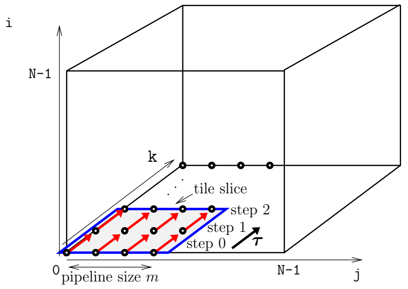
4.2 Motivating examples

In this section we illustrate the feasibility of our approach on two examples. The first example is the matrix-matrix multiplication, that has one uniform data dependency that propagates along one axis. The second example is the Jacobi 1D algorithm. It is more complicated because it has three uniform data dependencies with different distances.

Matrix-matrix multiplication. The original code is given in Figure 2(a). The iteration domain is the set integral points lying into a cube of size `N`, as shown in Figure 2(b). Each point of the iteration domain represents an execution of the assignment `S` with the corresponding values for the loop counters `i`, `j` and `k`. Essentially, the computation boils down to apply sequentially a multiply and accumulate operation $(x, y, z) \mapsto x + (y * z)$ that we want to implement with a specialized FloPoCo operator (Fig. 4(a)). It consists of a pipelined multiplier with ℓ pipeline stages that multiplies the elements of matrices `a` and `b`. In order to eliminate the step initializing `c`, the constant value is propagated inside loop `k`. In other words, for $k = 0$ the multiplication result is added with a constant value 0 (when the delayed control signal `S` is 0). For $k > 0$, the multiplication result is accumulated with the current sum, available *via* the feedback loop (when the

```
typedef float fl;
void mmm(fl* a, fl* b, fl* c, int N) {
  int i, j, k;
  for (i = 0; j < N; j++)
    for (j = 0; i < N; i++){
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j]; //S
    }
}
```

(a)



(b)

Fig. 2. Matrix-matrix multiplication: (a) C code, (b) iteration domain with tiling

delayed control signal `S` is 1). This result will be available m cycles later (m is the adder pipeline depth), for the next accumulation.

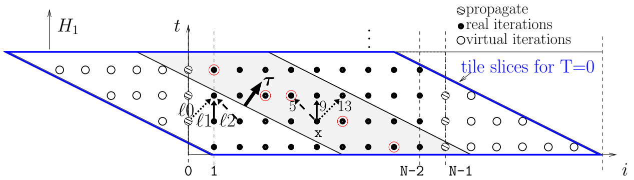
There is a unique data dependency carried by the loop k, which can be expressed as a vector $\boldsymbol{d} = (i_d, j_d, k_d) = (0, 0, 1)$ (Fig. 2(b)). The sequential execution of the original code would not exploit at all the pipeline, causing a stall of $m - 1$ cycles for each iteration of the loop `k` due to operator pipelining. Indeed, the iteration $(0, 0, 0)$ would be executed, then wait $m - 1$ cycles for the result to be available, then the iteration $(0, 0, 1)$ would be executed, and so on.

Now, let us consider the parallel hyperplane $H_{\boldsymbol{\tau}}$ with $\boldsymbol{\tau} = (0, 0, 1)$, which satisfies the data dependency $\boldsymbol{d}$. Each iteration on this hyperplane can be executed in parallel, independently, so it is possible to insert in the arithmetic operator pipeline one computation every cycle. At iteration $(0, 0, 0)$, the operator can be fed with the inputs x = c[0][0]=0, y = a[0][0], z = b[0][0]. Then, at iteration $(0, 1, 0)$, x = c[0][1]=0, y = a[0][0], z = b[0][1], and so on. In this case, the dependence distance would be $N - 1$, which means that the data computed by each iteration is needed $N - 1$ cycles later. This is normally much larger than the pipeline latency m of the adder and would require additional temporary storage. To avoid this, we have to transform the program in such a way that: between the definition of a variable at iteration $\boldsymbol{i}$ and its use at iteration $\boldsymbol{i} + \boldsymbol{d}$ there are exactly m cycles, i.e. $\Delta(\boldsymbol{d}) = m$.

The method consists on applying tiling techniques to reduce the dependence distance (Fig. 2(b)). First, as previously presented, we find a parallel hyperplane $H_{\boldsymbol{\tau}}$ (here $\boldsymbol{\tau} = (0, 0, 1)$). Then, we complete it into a valid tiling by choosing two hyperplanes H_1 and H_2 (here, the normal vectors are $(1, 0, 0)$ and $(0, 1, 0)$), $\mathcal{H} = (H_1, H_2, H_{\boldsymbol{\tau}})$. Basically, on this example, the tile width along H_2 is exactly $\Delta(\boldsymbol{d})$. Thus, it suffices to set it to the pipeline depth m. This ensures that the result is scheduled to be used exactly at the cycle it gets out of the operator pipeline. Thus, the result can be used immediately with the feedback connection, without any temporary buffering. In a way, the pipeline registers of the arithmetic operator are used as a temporary buffer.

```
typedef float fl;
void jacobi1d(fl a[T][N]){
 fl b[T][N];
 int i,t;
 for (t = 0; t < T; t++){
  for (i = 1; i < N−1; i++)
    a[t][i] = (a[t−1][i−1] + a[t−1][i] + a[t−1][i+1])/3;
}}
```

(a)

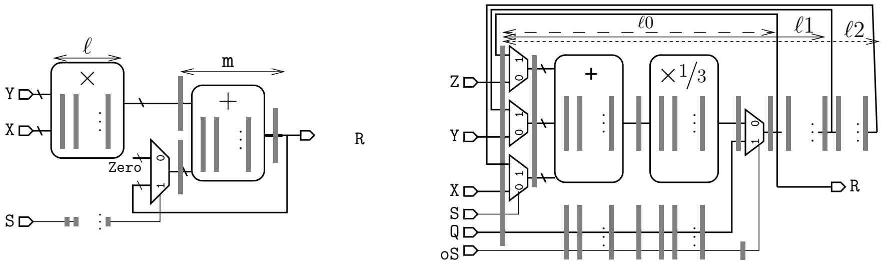


(b)

Fig. 3. Jacobi 1D: (a) source code, (b) iteration domain with tiling

Jacobi 1D. The kernel is given in Figure 3(a)). This is a standard stencil computation with two nested loops. This example is more complex because the set of dependence vectors $\mathcal{D}$ contain several dependencies $\mathcal{D} = \{\boldsymbol{d_1} = (-1,1), \boldsymbol{d_2} = (0,1), \boldsymbol{d_3} = (1,1)\}$ (Fig. 3(b)). We apply the same tiling method as in the previous example. First, we choose a valid parallel hyperplane $H_{\boldsymbol{\tau}}$, with $\boldsymbol{\tau} = (t_{\boldsymbol{\tau}}, i_{\boldsymbol{\tau}}) = (2,1)$. $H_{\boldsymbol{\tau}}$ satisfies all the data dependencies of $\mathcal{D}$. Then, we complete $H_{\boldsymbol{\tau}}$ with a valid tiling hyperplane H_1. Here, H_1 can be chosen with the normal vector $(1,0)$. The final tiled loop nest will have four loops: two inter-tile loops T and I iterating over the tiles, and two intra-tile loops tt and ii iterating into the current tile of coordinate (T,I). Therefore, any iteration vector can be expressed as (T,I,tt,ii). Figure 3(b) shows the consecutive tile slices with T=0. The resulting schedule is valid because it respects the data dependencies of $\mathcal{D}$. The data produced at iteration $\boldsymbol{i}$ must be available 5 iterations later *via* the dependence $\boldsymbol{d}_1$, 9 iterations later *via* dependency $\boldsymbol{d}_2$ and 13 iterations later *via* the dependence $\boldsymbol{d}_3$. Notice that the dependence distances are the same for any point of the iteration domain, as the dependencies are uniform. In hardware, this translates to add delay shift registers at the operator output and connect this output to the operator input *via* feedback lines, after data dependency distances levels ℓ_0, ℓ_1 and ℓ_2 (see Fig. 3(b)). Once again, the intermediate values are kept in the pipeline, no additional storage is needed on a slice.

As the tiling hyperplanes are not parallel to the original axis, some tiles in the borders are not full parallelograms (see left and right triangle from Fig. 3(b)). Inside these tiles, the dependence vectors are not longer constant. To overcome this issue, we extend the iteration domain with virtual iteration points where the pipelined operator will compute dummy data. This data is discarded at the border between the real and extended iteration domains (propagate iterations, when $i = 0$ and $i = N-1$). For the border cases, the correctly delayed data is fed via line `Q` (`oS`=1).



(a) Matrix-Matrix Multiply

(b) Jacobi 1D

Fig. 4. Computational kernels generated using FloPoCo

The two next sections formalize the ideas presented intuitively on motivating examples and presents an algorithm in two steps to translate a loop kernel written in C into an hardware accelerator using pipelined operators efficiently. Section 4.3 explains how to get the tiling. Then, section 4.4 explains how to generate the control FSM respecting the schedule induced by the loop tiling.

4.3 Step 1: Scheduling the Kernel

The key idea is to tile the program in such a way that each dependence distance can be customized by playing on the tile size. Then, it is always possible to set the minimum dependence distance to the pipelined depth of the FloPoCo operator, and to handle the remaining dependencies with additional (pipeline) registers in the way described for the Jacobi 1D example.

The idea presented on the motivating examples is to force the last intra-tile inner loop L_{par} to be parallel. This way, for a fixed value of the outer loop counters, there will be no dependence among iterations of L_{par}. The dependencies will all be carried by the outer-loop, and then, the dependence distances will be fully customizable by playing with the tile size associated to the loop enclosing immediately L_{par}, L_{it}.

This amounts to find a parallel hyperplane $H_{\boldsymbol{\tau}}$ (*step a*), and to complete with others hyperplanes forming a valid tiling (*step b*): $H_1, \ldots, H_{n-1}$, assuming the depth of the loop kernel is n. Now, it is easy to see that the hyperplane $H_{\boldsymbol{\tau}}$ should be the (n-1)-*th* hyperplane (implemented by L_{it}), any hyperplane H_i being the last one (implemented by L_{par}). Roughly speaking, L_{it} pushes $H_{\boldsymbol{\tau}}$, and L_{par} traverses the current 1D section of $H_{\boldsymbol{\tau}}$.

It remains in *step c* to compute the tile size to fit the fixed FloPoCo operator pipeline depth. If several dependencies exist, the minimum dependence distance must be set to the pipeline depth of the operator, and the other distances gives the number of extra shift registers to be added to the operator to keep the results within the operator pipeline, as seen with the Jacobi 1D example. These three steps are detailed thereafter.

Step a. Find a parallel hyperplane $H_{\boldsymbol{\tau}}$

This can be done with a simple integer linear program (ILP). Here are the constraints:

- $\boldsymbol{\tau}$ must *satisfy every dependence*: $\boldsymbol{\tau} \cdot \boldsymbol{d} > 0$ for each dependence vector $\boldsymbol{d} \in \mathcal{D}$.
- $\boldsymbol{\tau}$ must *reduce the dependence distances*. Notice that the dependence distance is increasing with the radius between $\boldsymbol{\tau}$, and the corresponding dependence vector $\boldsymbol{d}$. Notice that the radius $(\boldsymbol{\tau}, \boldsymbol{d})$ is decreasing with the dot product $\boldsymbol{\tau} \cdot \boldsymbol{d}$, and thus increasing with $-(\boldsymbol{\tau} \cdot \boldsymbol{d})$. Thus, it is sufficient to minimize the quantity $q = \max(-(\boldsymbol{\tau} \cdot \boldsymbol{d}_1), \ldots, -(\boldsymbol{\tau} \cdot \boldsymbol{d}_p))$. So, we build the constraints $q \geq -(\boldsymbol{\tau} \cdot \boldsymbol{d}_k)$ for each k between 1 and p, which is equivalent to $q \geq \max(-(\boldsymbol{\tau} \cdot \boldsymbol{d}_1), \ldots, -(\boldsymbol{\tau} \cdot \boldsymbol{d}_p))$.

With this formulation, the set of valid vectors $\boldsymbol{\tau}$ is an affine cone and the vectors minimizing q tends to have an infinite norm. To overcome this issue, we first minimize the coordinates of $\boldsymbol{\tau}$, which amounts to minimize their sum σ, as they are supposed to be positive. Then, for the minimum value of σ, we minimize q. This amounts to look for the *lexicographic minimum of the vector* (σ, q). This can be done with standard ILP techniques [21]. On the Jacobi 1D example, this gives the following ILP, with $\boldsymbol{\tau} = (x, y)$:

$$\begin{array}{ll}
\min_{\ll} & (x+y, q) \\
\text{s.t.} & (x \geq 0) \wedge (y \geq 0) \\
 & \wedge\, (y - x > 0) \wedge (y > 0) \wedge (x + y > 0) \\
 & \wedge\, (q \geq x - y) \wedge (q \geq -y) \wedge (q \geq -x - y)
\end{array}$$

Step b. Find the remaining tiling hyperplanes

Let us assume a nesting depth of n, and let us assume that $p < n$ tiling hyperplanes $H_{\boldsymbol{\tau}}$, $H_{\boldsymbol{\phi}_1}, \ldots, H_{\boldsymbol{\phi}_{p-1}}$ were already found. We can compute a vector $\boldsymbol{u}$ orthogonal to the vector space spanned by $\boldsymbol{\tau}, \boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_{p-1}$ using the internal inverse method [12]. Then, the new tiling hyperplane vector $\boldsymbol{\phi}_p$ can be built by means of ILP techniques with the following constraints.

- $\boldsymbol{\phi}_p$ must be a *valid tiling hyperplane*: $\boldsymbol{\phi}_p.\boldsymbol{d} \geq 0$ for every dependence vector $\boldsymbol{d} \in \mathcal{D}$.
- $\boldsymbol{\phi}_p$ must be *linearly independent* to the other hyperplanes: $\boldsymbol{\phi}_p.\boldsymbol{u} \neq 0$. Formally, the two cases $\boldsymbol{\phi}_p.\boldsymbol{u} > 0$ and $\boldsymbol{\phi}_p.\boldsymbol{u} < 0$ should be investigated. As we just expect the tiling hyperplanes to be valid, without any optimality criteria, we can restrict to the case $\boldsymbol{\phi}_p.\boldsymbol{u} > 0$ to get a single ILP.

Any solution of this ILP gives a valid tiling hyperplane. Starting from $H_{\boldsymbol{\tau}}$, and applying repeatedly the process, we get valid loop tiling hyperplanes $\mathcal{H} = (H_{\boldsymbol{\phi}_1}, \ldots, H_{\boldsymbol{\phi}_{n-2}}, H_{\boldsymbol{\tau}}, H_{\boldsymbol{\phi}_{n-1}})$ and the corresponding tiling matrix $U_{\mathcal{H}}$. It is possible to add an objective function to reduce the amount of communication between tiles. Many approaches give a partial solution to this problem in the context of automatic parallelization and high performance computing [12, 24, 27]. However how to adapt them in our context is not straightforward and is left for future work.

Step c. Compute the dependence distances

Given a dependence vector $\boldsymbol{d}$ and an iteration $\boldsymbol{x}$ in a tile slice the set of iterations $\boldsymbol{i}$ executed between $\boldsymbol{x}$ and $\boldsymbol{x}+\boldsymbol{d}$ is exactly:

$$D(\boldsymbol{x},\boldsymbol{d}) = \{\boldsymbol{i} \mid U_{\mathcal{H}}\boldsymbol{x} \ll U_{\mathcal{H}}\boldsymbol{i} \ll U_{\mathcal{H}}(x+\boldsymbol{d})\}$$

Remember that $U_{\mathcal{H}}$, the tiling matrix computed in the previous step, is also the intra-tile schedule matrix. By construction, $D(\boldsymbol{x},\boldsymbol{d})$ is a finite union of integral polyhedron. Now, the dependence distance $\Delta(\boldsymbol{d})$ is exactly the number of integral points in $D(\boldsymbol{x},\boldsymbol{d})$. As the dependence distance are constant, this quantity does *not* depend on $\boldsymbol{x}$. The number of integral points in a polyhedron can be computed with the Ehrhart polynomial method [14] which is implemented in the polyhedral library [6]. Here, the result is a degree 1 polynomial in the tile size ℓ_{n-2} associated to the hyperplane H_{n-2}, $\Delta(\boldsymbol{d}) = \alpha\ell_{n-2}+\beta$. Then, given a fixed input pipeline depth δ for the FloPoCo operator, two cases can arise:

- Either we just have *one dependence*, $\mathcal{D} = \{\boldsymbol{d}\}$. Then, solve $\Delta(\boldsymbol{d}) = \delta$ to obtain the right tile size ℓ_{n-2}.
- Either we have *several dependencies*, $\mathcal{D} = \{\boldsymbol{d}_1,\ldots,\boldsymbol{d}_p\}$. Then, choose the dependence vectors with smallest α, and among them choose a dependence vector $\boldsymbol{d}_m$ with a smallest β. Solve $\Delta(\boldsymbol{d}_m) = \delta$ to obtain the right tile size ℓ_{n-2}. Replacing ℓ_{n-2} by its actual value gives the remaining dependence distances $\Delta(\boldsymbol{d}_i)$ for $i \neq m$, that can be sorted by increasing order and used to add additional pipeline registers to the FloPoCo operator in the way described for the Jacobi 1D example (see figure 4(b)).

4.4 Step 2: Generating the Control FSM

This section explains how to generate the FSM that will control the pipelined operator according to the schedule computed in the previous section. A direct hardware generation of loops, would produce multiple synchronized Finite State Machines (FSMs), each FSM having an initialization time (initialize the counters) resulting in an operator stall on every iteration of the outer loops. We avoid this problem by using the Boulet-Feautrier algorithm [13] which generates a FSM whose states are assignments and whose transitions update the loop counters. The method takes as input the tiled iteration domain and the scheduling matrix $U_{\mathcal{H}}$ and uses ILP techniques to generate two functions: First() and Next(). The function First() is actually a constant function, returning the initial state of the FSM with initialized loop counters. The function `Next` is a transition function which updates the loop counters and gives the next state.

The functions First() and Next() are directly translated into VHDL `if` conditions. When the conditions are satisfied, the corresponding iterators are updated and the control signals are set.

The signal assignments in the FSM do not take into account the pipeline level at which the signals are connected. Therefore, we use additional registers to delay every control signal with respect to its pipeline depth. This ensures a correct execution without increasing the complexity of the state machine.

5 Reality Check

Table 1 presents synthesis results for both our running examples, using a large range of precisions, and two different FPGAs. The results presented confirm that precision selection plays an important role in determining the maximum number of operators to be packed on one FPGA. As it can be remarked from the table, our automation approach is both flexible (several precisions) and portable (Virtex5 and StratixIII), while preserving good frequency characteristics.

Table 1. Synthesis results for the full (including FSM) MMM and Jacobi1D codes. Results obtained using using Xilinx ISE 11.5 for Virtex5, and Quartus 9.0 for StratixIII

Application	**FPGA**	**Precision**	**Latency**	**Frequency**	**Resources**		
		(w_E, w_F)	**(cycles)**	**(MHz)**	**REG**	**(A)LUT**	**DSPs**
Matrix-Matrix Multiply `N=128`	Virtex5(-3)	(5,10)	11	277	320	526	1
		(8,23)	15	281	592	864	2
		(10,40)	14	175	978	2098	4
		(11,52)	15	150	1315	2122	8
		(15,64)	15	189	1634	4036	8
	StratixIII	(5,10)	12	276	399	549	2
		(9,36)	12	218	978	2098	4

Jacobi1D
stencil
`N=1024`
`T=1024`
	Virtex5(-3)	(5,10)	98	255	770	1013	

		(8,23)	98	250	1559	1833	
		(15,64)	98	147	3669	4558	
	StratixIII	(5,10)	98	284	1141	1058	
		(9,36)	98	261	2883	2266	
		(15,64)	98	199	4921	3978	

The generated kernel performance for one computing kernel is: 0.4 GFLOPs for matrix-matrix multiplication, and 0.56 GFLOPs for Jacobi, for a 200 MHz clock frequency. Thanks to program restructuring and optimized scheduling in the generated FSM, the pipelined kernels are used with very high efficiency. Here, the efficiency can be defined as the percentage of useful (non-virtual) inputs fed to the pipelined operator. This can be expressed as the ratio $\#(\mathcal{I} \setminus \mathcal{V})/\#\mathcal{I}$, where $\mathcal{I}$ is the iteration domain, as defined in section 4 and $\mathcal{V} \subseteq \mathcal{I}$ is the set of virtual iterations. The efficiency represents more than 99% for matrix-multiply, and more than 94% for Jacobi 1D. Taking into account the kernel size and operating frequencies, tens, even hundreds of pipelined operators can be packed per FPGA, resulting in significant potential speedups.

There exists several manual approaches like the one described in [20] that presents a manually implemented acceleration of matrix-matrix multiplication on FPGAs. Unfortunately, the paper lacks of detailed experimental results, so we are unable to perform correct performance comparisons. Our approach is fully automated, and we can clearly point important performance optimization. To store intermediate results, there approach makes a systematic use of local SRAM memory, whereas we rely on pipeline registers to minimize the use of local SRAM memory. As concerns commercial HLS tools, the comparison is made difficult due to lack of clear documentation as well as software availability to academics.

6 Conclusion and Future Work

In this paper, we have presented a novel approach using state-of-the-art code transformation techniques to restructure the program in order to use more efficiently pipelined operators. Our HLS flow has been implemented in the research compiler Bee, using FloPoCo to generate specialized pipelined floating point arithmetic operators. We have applied our method on two DSP kernels, the obtained circuits have a very high pipelined operator utilization and high operating frequencies, even for algorithms with tricky data dependencies and operating on high precision floating point numbers.

It would be interesting to extend our technique to non-perfect loop nests. This would require more general tiling techniques as those described in [12]. As for many other HLS tools, the HLS flow described in this paper focuses only on optimizing the performances of the computational part. However, experience shows that the performance is often bounded by the availability of data. In future work we plan to focus on local memory usage optimizations by minimizing the communication betweeen the tiles. This can be obtained by chosing a tile orientation to minimize the number of dependencies that crosses the hyperplane. This problem has been partially solved in the context of HPC [24, 12]. However, it is unclear how to apply it in our context. Also, we would like to focus on global memory usage optimizations by adapting the work presented in [9] and [26] to optimize communications with the outside world in a complete system design. Finally, we believe that the scheduling technique can be extended to apply several pipelined operators in parallel.

References

1. Forte design system: Cynthesizer. `http://www.forteds.com`
2. Impulse-C, `http://www.impulseaccelerated.com`
3. ISE 11.4 CORE Generator IP, `http://www.xilinx.com`
4. MegaWizard Plug-In Manager, `http://www.altera.com`
5. Mentor CatapultC high-level synthesis. `http://www.mentor.com`
6. Polylib – A library of polyhedral functions. `http://www.irisa.fr/polylib`
7. Synopsys: Synphony. `http://www.synopsys.com/`
8. Alias, C., Baray, F., Darte, A.: Bee+Cl@k: An implementation of lattice-based memory reuse in the source-to-source translator ROSE. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (2007)
9. Alias, C., Darte, A., Plesco, A.: Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators. An experience with the Altera C2H HLS tool. In: IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) (2010)

10. AutoESL: Autopilot datasheet (2009)
11. Bastoul, C., Cohen, A., Girbal, S., Sharma, S., Temam, O.: Putting polyhedral loop transformations to work. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC) (2003)
12. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM International Conference on Programming Languages Design and Implementation (PLDI) (2008)
13. Boulet, P., Feautrier, P.: Scanning polyhedra without Do-loops. In: IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT) (1998)
14. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In: ACM International Conference on Supercomputing (ICS) (1996)
15. de Dinechin, F.: A flexible floating-point logarithm for reconfigurable computers. Lip research report RR2010-22, ENS-Lyon (2010), `http://prunel.ccsd.cnrs.fr/ensl-00506122/`
16. de Dinechin, F., Joldes, M., Pasca, B., Revy, G.: Multiplicative square root algorithms for FPGAs. In: Field Programmable Logic and Applications. IEEE (2010)
17. de Dinechin, F., Klein, C., Pasca, B.: Generating high-performance custom floating-point pipelines. In: Field Programmable Logic and Applications. IEEE (2009)
18. de Dinechin, F., Pasca, B.: Floating-point exponential functions for DSP-enabled FPGAs. In: Field Programmable Technologies. IEEE (2010), `http://prunel.ccsd.cnrs.fr/ensl-00506125/`
19. de Dinechin, F., Pasca, B., Creţ, O., Tudoran, R.: An FPGA-specific approach to floating-point accumulation and sum-of-products. In: Field-Programmable Technologies. IEEE (2008)
20. Dou, Y., Vassiliadis, S., Kuzmanov, G.K., Gaydadjiev, G.N.: 64-bit floating-point fpga matrix multiplication. In: ACM/SIGDA symposium on Field-Programmable Gate Arrays (FPGA) (2005)
21. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle 22(3), 243–268 (1988)
22. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: Spark: A high-level synthesis framework for applying parallelizing compiler transformations. International Conference on VLSI Design (2003)
23. Irigoin, F., Triolet, R.: Supernode partitioning. In: 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (1988)
24. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (1997)
25. Martin, E., Sentieys, O., Dubois, H., Philippe, J.L.: Gaut: An architectural synthesis tool for dedicated signal processors. In: Design Automation Conference with EURO-VHDL'93 (EURO-DAC) (1993)
26. Plesco, A.: Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators. Ph.D. thesis, École Normale Supérieure de Lyon (2010)
27. Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers (2000)