FPGA-Specific Synthesis of Loop-Nests with Pipelined Computational Cores

Christophe Alias, Bogdan Pasca, Alexandru Plesco

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

FPGA-Specific Synthesis of Loop-Nests with Pipelined Computational Cores

Christophe Alias — Bogdan Pasca — Alexandru Plesco

N° 7674

Juillet 2011

ISSN 0249-6399 ISRN INRIA/RR--7674--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

FPGA-Specific Synthesis of Loop-Nests with Pipelined Computational Cores

Christophe Alias*, Bogdan Pasca †, Alexandru Plesco ‡

Thème : Architecture et compilation
Équipes-Projets Compsys et Arénaire

Rapport de recherche n° 7674 — Juillet 2011 — 33 pages

Abstract: Increases in the capacities and features of FPGAs has opened a new perspective on their use as application accelerators. However, in order for FPGAs to be accepted as mainstream solutions, the long design cycles must be shortened by using high-level synthesis tools in the design process. Current HLS tools targeting FPGAs come with several limitations, and one of them is the efficient use of pipelined arithmetic operators, commonly encountered in high-throughput FPGA designs. We focus here on the efficient generation of FPGA-specific hardware accelerators for regular codes with perfect loop nests where inner statements are implemented as a pipelined arithmetic operator, which is often the case of scientific codes using floating-point arithmetic. We propose a semi-automatic code generation process where the arithmetic operator is identified and generated. Its pipeline information is used to reschedule the initial program execution in order to keep the operator's pipeline as "busy" as possible, while minimizing memory access. Next, we show how our method can be used as a tool to generate control FSMs of multiple parallel computing cores. Finally, we show that accounting for the application's accuracy needs allows designing smaller and faster operators.

Key-words: High-level synthesis, FPGA, data-reuse, perfect loop-nests, pipelined arithmetic operators, floating-point, parallelization, kernel accuracy

* Permanent Research Scientist at INRIA
† Ph.D. student (ARENAIRE team)
‡ Ph.D.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Synthèse haut-niveau de nids de boucles sur FPGA avec des noyaux de calcul pipelinés

Résumé : L'augmentation des fonctionnalités et de la capacité des FPGAs ouvre de nouvelles perspectives pour la conception d'accélérateurs de calcul. Cependant, pour que les FPGAs soient communément acceptés, le cycle de développement – habituellement long – doit être réduit en utilisant des outils de synthèse de haut-niveau. Les outils actuels pour FPGAs ont de nombreuses limitations. En particulier, ils ne parviennent pas à utiliser efficacement les opérateurs arithmétiques pipelinés, fréquement utilisés dans les designs FPGAs. Dans ce rapport, nous nous intéressons à la génération efficace d'accélérateurs matériels sur FPGA, pour les codes de calcul réguliers avec des nids de boucle parfaits et des références affines, dans lesquels les affectations sont implémentées avec un noyau arithmétique pipeliné. Ce type de programme est particulièrement fréquent dans les codes de calcul scientifique en virgule flottante. Nous proposons une technique d'ordonnancement et de génération de code VHDL où le noyau arithmétique est identifié par l'utilisateur, puis généré. La profondeur de pipeline du noyau arithmétique est utilisée pour réordonnancer l'exécution du programme de façon à utiliser le pipeline de façon optimale, tout en minimisant les accès mémoires. Ensuite, nous montrons comment notre méthode peut être utilisée pour générer un automate de contrôle pour plusieurs noyaux arithmétiques fonctionnant en paralléle. Enfin, nous montrons que tenir compte du besoin en précision de l'application permet de construire des accélérateurs plus petits et plus rapides.

Mots-clés : Synthèse de circuits haut-niveau, FPGA, réutilisation de données, nids de boucle parfaits, noyaux arithmétiques pipelinés, virgule flottante, parallelisation, précision de calcul

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 microprocessors: may not accelerate non-standard computations (exponential, logarithm, square-root) and performance suffers when implementing low-grain parallelism due to interprocess communication (2) application-specific integrated circuits (ASICs): the price tag is often too big, (3) Field Programmable Gate Arrays (FPGAs): provide a trade-off between the performances of ASICs and the costs of microprocessors.

FPGAs are memory-based integrated circuits whose functionality can be modified after manufacturing. They are organized as bi-dimensional arrays of logic elements containing small programmable memories connected by a configurable routing network. Modern FPGAs also include “ASIC-like" features like: embedded memories, embedded DSP blocks containing small multipliers, embedded processors etc. All these features combined with increasing capacities allow modern FPGAs to be uses with success as application accelerators.

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. Nevertheless, designing entire systems using these languages is tedious and error-prone. Moreover, it has been recently shown that using generator frameworks such as FloPoCo [1] for designing the arithmetic data-paths of such applications can increase both performance and productivity. What remains in order to globally increase productivity are tools which use these arithmetic operators and efficiently map computations to them.

In order to address this productivity issue, much research has focused on high-level synthesis (HLS) tools [2, 3, 4, 5, 6], which input the system description in higher level language, such as the `C` programming language (C). Unfortunately, so far none of these tools come close to the speedups obtained by the manual design approach. On one hand, the synthesis of arithmetic data-paths, key components of such systems, reduces to assembling library operators. We have proved that manual solutions outperform this process even if state-of-the-art arithmetic operators are used [1]. On the other hand, these tools perform poorly when synthesizing loops having inter-iteration dependencies and where the inner statement involve deeply pipelined arithmetic operators.

One of the most popular forms of arithmetic requiring *deeply pipelined operators* in FPGA designs is *floating-point arithmetic*. Floating-point arithmetic offers a different trade-off between precision, dynamic range and implementation cost than fixed-point arithmetic, classically used in FPGA designs. The implemented operators require more area but offer a better dynamic range, which is often crucial in applications manipulating these type of values (most scientific computing applications).

Some HLS tools supporting standard floating-point arithmetic do exist [4, 5, 3]. They allow synthesizing loop nests having inter-iteration dependencies where the inner statement is an arithmetic operation implemented in floating-point arithmetic. However, performance is very poor due to the deep pipeline of the arithmetic operators which causes the system to stall, waiting for the operation result before starting the next iteration.

In this article, 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 bubbles (void computations). We present here two classic examples: matrix multiplication and the Jacobi 1D relaxation for which we describe the computational kernels and the code transformations used to reschedule their execution. We also discuss execution parallelization opportunities for these computing kernels and the impact of accuracy-aware arithmetic operator design on the operator kernel area. 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 4.5 provide a formal description of our method. Section 5 discusses the different parallelization opportunities, in the context of minimizing communication costs for our two applications. Next, Section 6 discusses the impact of accuracy-aware operator design on the final operator size. Section 7 provides experimental results on the running examples. Finally, Section 8 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. Most of the tools restrict accepted data types to simple ones like integer or fixed point excluding floating-point format. This is mostly due to the low resource utilization of the corresponding arithmetic operators. Another factor can also be attributed to the pipeline depth of the operators. Current high-level synthesis tools use CDFG like internal data structures to represent the program. This representation limits the analysis of loops. Data dependency analysis on these data structures is often limited to the syntactic level analysis (example C2H tool from Altera [7]) or cannot be computed exactly. Even when data dependencies can be computed more accurately, loop code transformation on control and data flow graph (CDFG) like the ones described in [8] are not powerful enough to reschedule loop execution in order to increase data dependency length so that pipelined arithmetic operators can be feed with data at each cycle. One can apply these transformation by hand. We can take for example a code consisting of two nested loops with the outer parallel loop and the inner sequential with loop carried dependencies. Even if the designer can interchange the loops by hand, if the tool cannot detect correctly the parallelism, due to non-fine data dependency analysis, it will still schedule it sequentially inserting voids in the pipelined operators. Tools like C2H permits the designer by using a pragma restrict keyword to specify that two pointers do not alias. One can use two restricted pointers to reference the same array when writing and reading to force eliminate the false data dependency. However, this method works only in some cases and requires deep user knowledge of the underlying tool.

Most of the current high-level synthesis tools like Spark [9], Gaut [2], Synphony [6], Mentor Graphics' CatapultC [10] and others originate from the time when fixed-point formats were sufficient to map most of the applications into silicium. However this is not the case today, when the applications targeted for FPGAs process data having a

wider dynamic range with increased precisions. The high-throughput scenarios FPGAs are used in, require the fixed or floating point operators to be deeply pipelined.

In order to workaround the known weaknesses of fixed-point arithmetic, AutoPilot [4], Impulse-C [3], and Cynthesizer [5] (in SystemC) can synthesize floating-foint (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. Impulse-C can instantiate IEEE-754 SP and DP standard FP operators using Xilinx and Altera libraries. 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 (integer, fixed-point, floating-point or a mix) circuits, we present an automation tool chain implemented in the Bee research compiler [11], and which uses FloPoCo [1], 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.

Another important advantage of fine data dependency analysis is that one can detect and parallelize codes that standard techniques (like the ones used in most HSL tools) cannot. Detecting the parallelism is mandatory but not sufficient to improve performances. One should take into consideration the deployment platform on which the algorithm will run. In our case we use FPGAs that have an advantage compared to most multicore processing systems that one can use fast dedicated lines to communicate between processing elements. In order to ensure a correct computation, we use fine data dependency analysis techniques together with advanced code transformation techniques.

The techniques presented in this article come as a natural evolution of hand-based scheduling techniques applied for the matrix-matrix multiplication [12, 13]. However, our techniques are more general and automatic but also refine the execution scheduling (more accurte FSMs) such that the generated architectures require no buffers (whereas both previous works require buffers) even for codes with more complex dependencies such as the 1D Jacobi kernel.

3 Designing arithmetic kernels using FloPoCo

Arithmetic operators are the key components of loop-nest accelerators, as the accelerator's frequency and area are strongly influenced by those of the arithmetic operator. 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, area and latency (number of pipeline levels) and out task reduces to generating a circuit with just the right frequency thus minimizing area and latency.

Assembling and synchronizing by hand the data-path of the arithmetic operator using subcomponents from common operator libraries or generators such as Xilinx Logicore [14], Altera Megawizard [15] and others offers full control over the choice of subcomponents and their characteristics: implementation, input/output precision, latency etc. which potentially allows building efficient circuits. The drawback lies in the long design cycles needed to build such pipelined system for a user-defined frequency due to the fact that all the subcomponents are parametrized by their latency: if the performance is not acceptable some components need to be pipelined deeper and the system resynchronized.

The approach behind open-source FloPoCo Core Generator[1] [1] is radically different. For a given subcomponent, the user specifies its parameters (as for other core generators), the target running frequency f and a target FPGA (currently several FPGAs from main manufacturers Altera and Xilinx are supported). FloPoCo outputs the operator's description in platform independent and human-readable VHDL. This approach allows to assemble an arithmetic data-path for a target frequency f using subcomponent built for that frequency. *Frequency-driven pipelining* can only be found in recent work by Perry [16] in the Advanced DSP Builder from Altera.

FloPoCo also offers a development framework which allows to assemble the operators available in its library. The framework decouples the task of describing the *functionality* of the arithmetic pipeline from *pipelining* the circuit, minimizing the possibility of flaws and thus enhancing productivity. The framework also offers a testbench suite for validating the implementation against its mathematical description. An alternative automatic solution for assembling floating-point pipelines is given by Langhammer with the Altera Floating-Point Datapath Compiler [17]. The compiler inputs and outputs numbers in IEEE-754 format (SP is discussed) but uses alternative internal representations formats and fuses similar operations clusters, with the main goal of better using the Altera FPGA resources. The drawback of using this compiler is that it would restrict us to Altera FPGAs and to floating-point pipelines although FPGAs allow using mix of integer, fixed and floating-point operators for efficiently performing a given computation. Moreover, as shown in [1] on the $\sqrt{X^2 + Y^2 + Z^2}$ operators, FloPoCo manages to embed more optimizations, at the expense of a longer development time.

The main philosophy of FloPoCo arithmetic data-path design should use operators which allow satisfying the application's accuracy needs [18]: this includes using a mix of integer, fixed, floating-point and possibly other datatypes with application-dictated custom *custom precisions*. A perfect example is the architecture from [19] which combines fixed and floating-point data-types in order to fit the application in one FPGA.

Some of the built-in operators of the ever-increasing FloPoCo operator library are:

- *specialized operators* like squarers [20]. constant multipliers [21], faithfully rounded multipliers with a user-defined accuracy(allow significantly reducing implementation resources) [22] FPGA-specific floating-point accumulators [23].
- a generic fixed-point function evaluator based on polynomial approximations (*FunctionEvaluator*) [24].

[1] http://flopoco.gforge.inria.fr/

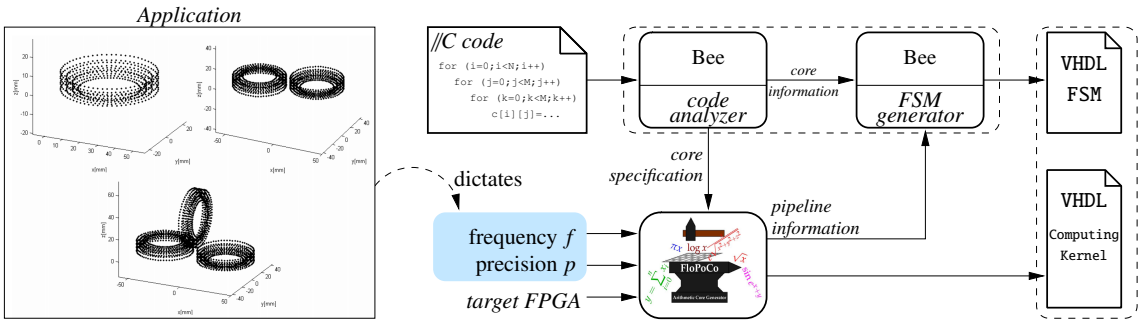


Figure 1: Automation flow

- floating-point functions: square-root [25], logarithm [26], exponential [27] which are implemented using mathematical libraries (libms) in microprocessors and are usually two orders of magnitude slower than the basic floating-point operators $+, \times$.
- dedicated architectures for *coarser operators* which have to be implemented in software in processors, for example $X^2 + Y^2 + Z^2$, and others [1].

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 but also discuss parallelization opportunities in this context. The automation flow for for generating and scheduling the computations onto computational cores is given in Figure 1.

4 Generation of Sequential Hardware with Pipelined Operators

In this section, we present a method to derive automatically an efficient, sequential, hardware using FloPoCo operators in the data-path. The operations are carefully scheduled to keep the FloPoCo operators busy, hence making an optimal use of their pipelines. The input kernel is specified by a naive sequential C program, as depicted in figure 2(a) for matrix multiplication. The user must also specify the pipeline depth for each FloPoCo operator. These are the only input required. Section 4.1 presents the model of programs which can be processed, and review the corresponding methodology. Then, Section 4.2 gives an intuitive explanation of our method on two important examples. Finally, the two steps of our method are formally described in Sections 4.3 and 4.5.

4.1 Program model and background

This section defines the class of programs which can be processed by our method, and precisely review several related notions which are used in the remaining of this paper. For more details the interested reader can consult [28].

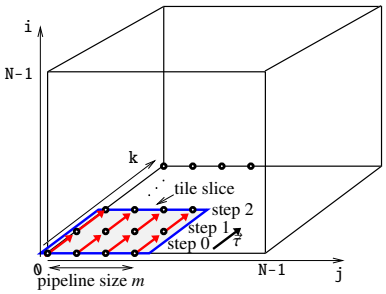
Program model. We consider kernels with a single *perfect loop nest*, that is an imbrication of `for` loops where each level contains either a single `for` loop or a single assignment S. A typical example is the matrix multiply kernel given in figure 2(a). Writing $i_1, ..., i_n$ the loop counters, the vector $\vec{i} = (i_1, ..., i_n)$ is called an *iteration vector*. The set of iteration vectors $\vec{i}$ reached during an execution of the kernel is called an *iteration domain* (see figure 2(b)). The execution instance of S at the iteration $\vec{i}$ is called an *operation* and is denoted by the couple $(S, \vec{i})$. 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. Moreover, the loop bounds and the array indices must be *affine expressions* of surrounding loop counters and structure parameters. For instance, the kernels matrix multiply (figure 2(a)) and jacobi 1D (figure 3(a)) fall in this category. Under these restrictions, the iteration domain $\mathcal{I}$ is invariant whatever the input value is. Also, as loop bounds are affine, the iteration domain $\mathcal{I}$ is always a set of affine lattice points lying in a polytope, usually refered as a $\mathbb{Z}$-polytope. This property makes possible to design a program analysis by means of integer linear programming (ILP) techniques and operations on polytopes.

Dependence vectors. On this program model, the data dependences can be computed at the iteration level. This enables very accurate analysis, like the number of cycles between the source and target of a dependence. As we will see, this capability is absolutely mandatory to take the best profit of FloPoCo pipelined operators. We will assume each data dependence to be *uniform*. This means that each occurence of the dependence is directed by the same vector $\vec{d}$ and must occurs from the iteration $\vec{i}$ to the iteration $\vec{i} + \vec{d}$ for every valid iterations $\vec{i}$ and $\vec{i} + \vec{d}$. In this case, we can represent the data dependence with the vector $\vec{d}$ 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 $\mathcal{D} = \{\vec{d}_1, \dots \vec{d}_p\}$ the set of dependence vectors. With this assumption, the time ellapsed between the production of a data and its use (along a dependence) is constant. As we will see, this important property allow to let the data flow into FIFOs of small constant size from the producer to the consummer, avoiding the use of buffers. Many numerical kernels fit or can be restructured to fit in this model [29]. Particularly, this model 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* $\theta(\vec{i}) = U\vec{i}$ where U is an integral matrix. If there is a dependence from an iteration $\vec{i}$ to an iteration $\vec{j}$, then $\vec{i}$ must be executed before $\vec{j}$: $\theta(\vec{i}) \ll \theta(\vec{j})$. With uniform dependencies, this gives $U\vec{d} \gg 0$ for each dependence vector $\vec{d} \in \mathcal{D}$. Each line $\vec{\phi}$ of U can be seen as the normal vector to an affine hyperplane $H_{\vec{\phi}}$, the iteration domain being scanned by translating the hyperplanes $H_{\vec{\phi}}$ in the lexicographic ordering. An hyperplane $H_{\vec{\phi}}$ *satisfies* a dependence vector $\vec{d}$ if by translating $H_{\vec{\phi}}$ in the direction of $\vec{\phi}$, the source $\vec{i}$ is touched before the target $\vec{i} + \vec{d}$ for each $\vec{i}$, that is if $\vec{\phi}.\vec{d} > 0$. We say that $H_{\vec{\phi}}$ *preserves* the dependence $\vec{d}$ if $\vec{\phi}.\vec{d} \geq 0$ for each dependence vector $\vec{d}$. In

```
1 typedef float fl;
2 void mmm(fl* a, fl* b, fl* c, int N) {
3   int i, j, k;
4   for (i = 0; j < N; j++)
5     for (j = 0; i < N; i++){
6       for (k = 0; k < N; k++)
7         c[i][j] = c[i][j] + a[i][k]*b[k][j
              ];//S
8     }
9 }
```

(a)



(b)

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

that case, the source and the target can be touched at the same iteration. $\vec{d}$ must then be solved by a subsequent hyperplane. We can always find an hyperplane $H_{\vec{\tau}}$ *satisfying* all the dependencies. Any translation of $H_{\vec{\tau}}$ touches in $\mathcal{I}$ a subset of iterations which can be executed in parallel. In the literature, $H_{\vec{\tau}}$ is usually refereed as a *parallel hyperplane*.

Loop tiling. With loop tiling [30, 31], 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 $(H_1, \ldots, H_n)$. The vector $\vec{\phi}_k$ is the normal to the hyperplane H_k and the vectors $\vec{\phi}_1, \ldots, \vec{\phi}_n$ 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 $\mathcal{H} = (H_1, \ldots, H_n)$ is *valid* if each normal vector $\vec{\phi}_k$ preserves all the dependencies: $\vec{\phi}_k.\vec{d} \geq 0$ for each dependence vector $\vec{d}$. 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 $\vec{\phi}_1, \ldots \vec{\phi}_n$. 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 $\vec{d}$ at the iteration $\vec{i}$ is the number of iterations executed between the source iteration $\vec{i}$ and the target iteration $\vec{i} + \vec{d}$. 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 $\vec{d}$ does not depend on the source iteration $\vec{i}$ (see figure 3(b)). Thus, we can write it $\Delta(\vec{d})$. 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.

```
1  int tsi = 2;
2  int tsj = 2;
3  int tsk = 2;
4  int N=4;
5  for (I = 0; i < N/tsi; J++)
6   for (J = 0; J < N/tsj; J++)
7    for (K = 0; K < N/tsk; K++)
8     for (ii = 0; ii < tsi; ii++)
9      for (kk = 0; kk < tsk; kk++)
10      for (jj = 0; jj < tsj; jj++)
11       c[I*tsi+ii][J*tsj+jj] +=
12             a[I*tsi+ii][K*tsk+kk]*b[K*tsk+kk][J*tsj+jj];
```

Listing 1: One valid tiling for the matrix-matrix multiplication

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.

4.2.1 Matrix-matrix multiplication.

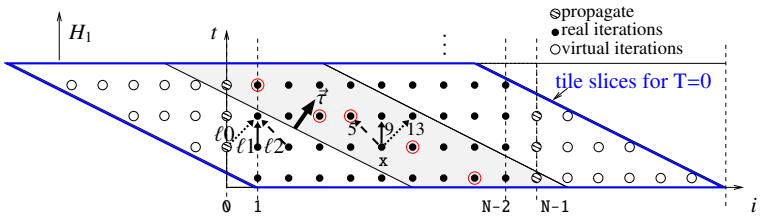
The original code is given in Figure 2(a). The iteration domain is the set of integral points lying into a cube of size `N`, as shown in Figure 2(b). Each point of the iteration domain represents an execution of the assignment `S` with the corresponding values for the loop counters `i`, `j` and `k`. Essentially, the computation boils down to apply sequentially a multiply and accumulate operation $(x, y, z) \mapsto x + (y * z)$ along the `k` axis, 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 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 $\vec{d} = (i_d, j_d, k_d) = (0, 0, 1)$ (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 $H_{\vec{\tau}}$ with $\vec{\tau} = (0, 0, 1)$, which satisfies the data dependency $\vec{d}$. 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

```
1 typedef float fl;
2 void jacobi1d(fl a[T][N]){
3   fl b[T][N];
4   int i,t;
5   for (t = 0; t < T; t++)
6     for (i = 1; i < N-1; i++)
7       a[t][i] = (a[t-1][i-1]+↩
            a[t-1][i]+
8             a[t-1][i+1])/3;
9 }
```

(a)



(b)

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

a way that: between the definition of a variable at iteration $\vec{i}$ and its use at iteration $\vec{i}+\vec{d}$ there are exactly m cycles, i.e. $\Delta(\vec{d}) = m$.

The method consists on applying tiling techniques to reduce the dependence distance (Fig. 2(b)). First, as previously presented, we find a parallel hyperplane $H_{\vec{\tau}}$ (here $\vec{\tau} = (0,0,1)$). 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 $(0,1,0)$), $\mathcal{H} = (H_1, H_2, H_{\vec{\tau}})$. Basically, on this example, the tile width along H_2 is exactly $\Delta(\vec{d})$. 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.

The code corresponding one valid tiling is given in listing 1.

4.2.2 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 $\mathcal{D} = \{\vec{d_1} = (-1,1), \vec{d_2} = (0,1), \vec{d_3} = (1,1)\}$ (Fig. 3(b)). We apply the same tiling method as in the previous example. First, we choose a valid parallel hyperplane $H_{\vec{\tau}}$, with $\vec{\tau} = (t_{\vec{\tau}}, i_{\vec{\tau}}) = (2,1)$. $H_{\vec{\tau}}$ satisfies all the data dependencies of $\mathcal{D}$. Then, we complete $H_{\vec{\tau}}$ 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 $\vec{i}$ must be available 5 iterations later *via* the dependence $\vec{d_1}$, 9

```
1  int T,I,ii,tt, TIME, N;
2  int th, tw;
3  for (T = 0; T < TIME/th; T++)
4   for (I=0; I < N/tw; I++)
5    for (ii=0; ii<tw; ii++)
6     for (tt=0; tt<th; tt++)
7      if (I*tw-2*tt+i==0 || I*tw-2*tt+i==N-1)
8       //propagate
9       a[T*th+tt][I*tw-2*tt+i]=a[T*th+tt-1][I*tw-2*tt+i];
10     else if (I*tw-2*tt+i < 0 || I*tw-2*tt+i > N-1){
11      //dummy: virtual iteration points
12     }else
13      a[T*th+tt][I*tw-2*tt+i] =
14               (a[T*th+(tt-1)][I*tw-2*(tt-1)+(i-1)]+
15                a[T*th+(tt-1)][I*tw-2*(tt-1)+ i ]+
16                a[T*th+(tt-1)][I*tw-2*(tt-1)+(i+1)])*1/3;
```

Listing 2: Code using tiling for 1D Jacobi stencil computation

iterations later *via* dependency $\vec{d}_2$ and 13 iterations later *via* the dependence $\vec{d}_3$. 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`). The C code having the tiled iteration domain is given in listing 2.

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 4.5 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 $H_{\vec{\tau}}$ (*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 $H_{\vec{\tau}}$ 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 $H_{\vec{\tau}}$, and L_{par} traverses the current 1D section of $H_{\vec{\tau}}$.

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 $H_{\vec{\tau}}$

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

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

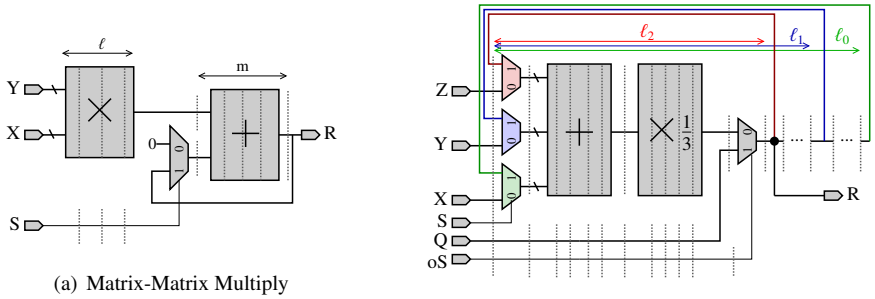
With this formulation, the set of valid vectors $\vec{\tau}$ is an affine cone and the vectors minimizing q tends to have an infinite norm. To overcome this issue, we first minimize the coordinates of $\vec{\tau}$, which amounts to minimize their sum σ, as they are supposed to be positive. Then, for 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 [32]. On the Jacobi 1D example, this gives the following ILP, with $\vec{\tau} = (x, y)$:

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

Step b. Find the remaining tiling hyperplanes

Let us assume a nesting depth of n, and let us assume that $p < n$ tiling hyperplanes $H_{\vec{\tau}}$, $H_{\vec{\phi}_1}, \ldots, H_{\vec{\phi}_{p-1}}$ were already found. We can compute a vector $\vec{u}$ orthogonal to the vector space spanned by $\vec{\tau}, \vec{\phi}_1, \ldots, \vec{\phi}_{p-1}$ using the internal inverse method [33]. Then, the new tiling hyperplane vector $\vec{\phi}_p$ can be built by means of ILP techniques with the following constraints.

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



(a) Matrix-Matrix Multiply

(b) Jacobi 1D

Figure 4: Computational kernels generated using FloPoCo

Any solution of this ILP gives a valid tiling hyperplane. Starting from $H_{\vec{\tau}}$, and applying repeatedly the process, we get valid loop tiling hyperplanes $\mathcal{H} = (H_{\vec{\phi}_1}, \ldots, H_{\vec{\phi}_{n-2}}, H_{\vec{\tau}}, H_{\vec{\phi}_{n-1}})$ 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 [33, 34, 31]. 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 $\vec{d}$ and an iteration $\vec{x}$ in a tile slice the set of iterations $\vec{i}$ executed between $\vec{x}$ and $\vec{x} + \vec{d}$ is exactly:

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

Remember that $U_{\mathcal{H}}$, the tiling matrix computed in the previous step, is also the intra-tile schedule matrix. By construction, $D(\vec{x}, \vec{d})$ is a finite union of integral polyhedron. Now, the dependence distance $\Delta(\vec{d})$ is exactly the number of integral points in $D(\vec{x}, \vec{d})$. As the dependence distance are constant, this quantity does *not* depend on $\vec{x}$. The number of integral points in a polyhedron can be computed with the Ehrhart polynomial method [35] which is implemented in the polyhedral library [36]. Here, the result is a degree 1 polynomial in the tile size ℓ_{n-2} associated to the hyperplane H_{n-2}, $\Delta(\vec{d}) = \alpha\ell_{n-2} + \beta$. Then, given a fixed input pipeline depth δ for the FloPoCo operator, two cases can arise:

- Either we just have *one dependence*, $\mathcal{D} = \{\vec{d}\}$. Then, solve $\Delta(\vec{d}) = \delta$ to obtain the right tile size ℓ_{n-2}.
- Either we have *several dependencies*, $\mathcal{D} = \{\vec{d}_1, \ldots, \vec{d}_p\}$. Then, choose the dependence vectors with smallest α, and among them choose a dependence vector $\vec{d}_m$ with a smallest β. Solve $\Delta(\vec{d}_m) = \delta$ to obtain the right tile size ℓ_{n-2}. Replacing ℓ_{n-2} by its actual value gives the remaining dependence distances $\Delta(\vec{d}_i)$ 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)).

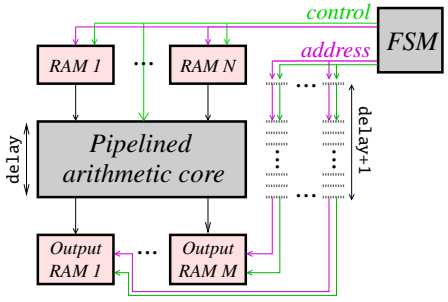


Figure 5: Computing core architecture

4.4 Step 2: Generating the Control

This section explains how to generate the finite state machine (FSM) that will control the computational kernels according to the schedule computed in the previous section. A direct translation 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 [37] which generates a single FSM capturing the whole execution schedule of the loop nest. At each cycle, the resulting FSM executes the next operation scheduled. This allows to respect the timing induced by dependence distances. The states of the Boulet-Feautrier FSM are simply assignment numbers, each transition updating the assignment number and the loop counters signals. The Boulet-Feautrier procedure takes as input the tiled iteration domain and the scheduling matrix $U_{\mathcal{H}}$ and uses ILP techniques to generate two functions. A function First() returning the initial state of the FSM with initialized loop counters. And a function Next() which updates the loop counters and gives the next state. Then, the functions First() and Next() are trivially translated into a VHDL FSM.

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.

4.5 Step 3: Computing core

In this section, we present the general architecture of the computing core which will be used for all the kernels. The architecture is presented in figure 5. It consists of the FSM, the FloPoCo core and multiple memories that the design may require. For the matrix multiplication example there are two memories from which matrices A and B are read and one memory to store the matrix C. In general case we can have N input memories and M output ones.

The FSM generates address and control signals for memories and control signals to FloPoCo core. When generating the FSM, we don't take into consideration the pipelined architecture. Write to memories signals cannot be connected directly. We use loop software pipelining method [8] to insert correct delay registers. Control signals for the FloPoCo core are delayed inside the core. In this case, the pipelined FloPoCo core can be viewed as a pipelined loop basic block. In our case the initiation interval

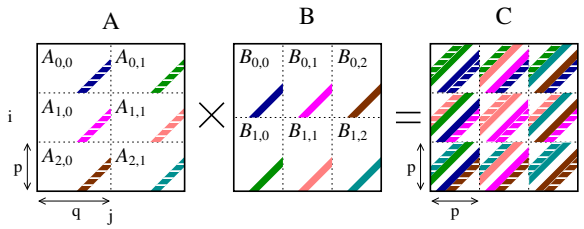


Figure 6: Matrix-matrix multiply using blocking

of the loop is one and the latency is equal to the delay of the FloPoCo core plus one (latency of the memory read). The presented method is suitable for regular kernels where the execution can be statically determined, as the control flow does not depend on values produced in the kernel.

To get an idea of the FSM complexity we present the FSM for Jacobi 1D example in Appendix (Section 9).

5 Parallelization and communication optimization

In the previous sections we have described an effective method for efficiently mapping an entire computational task described by a perfect loop nest to one computing kernel. In this section show how this methodology can be effectively utilized for generating the control FSMs needed for scheduling this task onto *multiple* computing kernels.

5.1 Matrix-matrix multiplication

Parallelizing the matrix-matrix multiplication kernel can be seen as simple due to the fact that both external loops i and j carry no dependencies. However, this is not entirely true if we want this parallelization to be efficient as well, with regard to memory transfers.

A naive implementation of a single computing kernel performing $C = AB$ requires $4N^3$ memory accesses: N^3 ($read(a) + read(b) + read(c) + store(c)$). At each step two elements are ready from A and B together with the destination accumulator from C. After the computation is done, the corresponding element from c is updated in the memory. By using our technique to reschedule the execution of this core we avoid having to read and update c at each iteration step, as its value is stored inside the pipeline's registers: $N^2(N(read(a) + read(b)) + store(c))$.

We can additionally reduce this cost if we are provided with local memory. *Blocking* consists is splitting the input matrices into blocks which are fetched in pairs into the local memory. Figure 6 illustrates this technique. For a given block-size $p \times q$ (where we suppose for simplicity that both p and q divide N) and suppose we are provided with $2(p \times q) + (p \times p)$ local memory for buffering (sufficient to store one block from A,B and C), the external memory requirement is:

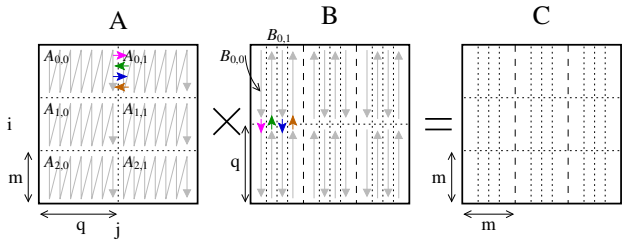


Figure 7: Matrix-matrix multiply blocking applied using our technique. Scheduling of computations is modified in order to minimize external memory usage

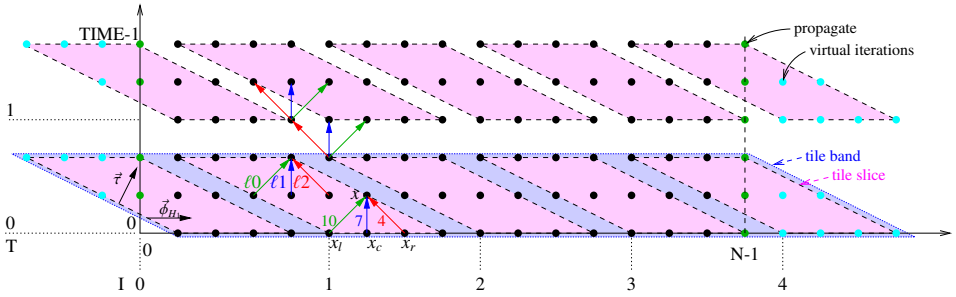


Figure 8: Tiled iteration domain for 1D Jacobi stencil computation

$$M = 2\frac{N}{p}\frac{N}{q}\frac{N}{p}(p \times q) + (2\frac{N}{q} - 1)\frac{N^2}{p^2}(p \times p)$$
$$= 2\frac{N^3}{p} + (2\frac{N}{q} - 1)N^2$$

The technique trades local memory requirement for memory bandwidth. For $p = q = N$ it reduces to storing locally the three matrices $3N^2$ buffer. The bandwidth requirement is $2N^2$ for fetching A and B and N^2 for writing C.

When the execution schedules the processing of consecutive memory blocks in the direction of j: $A_{0,0} \times B_{0,0}, A_{0,1} \times B_{1,0}$ etc. the same block C block will get affected, and is therefore possible to skip its writing to memory until the last product affecting it was processed ($C_{0,0}$ is written to the main memory only when $A_{0,1} \times B_{1,0}$ was complete. This reduces our memory bandwidth to $2\frac{N^3}{p} + N^2$. Now, by applying our scheduling technique, we are able to process entire computation without even needing a buffer for the C block (its values are stored inside the operator's pipeline levels). The current technique requires freezing the computational kernels the time needed to fetch a new pair of blocks from A and B.

Consider the Figure 7 which illustrates how our scheduling algorithm would perform if blocking was used. Note that m denotes the number of stages of our accumulator (see Figure 4(a)). The points executed in the i direction of are on parallel front and therefore have no data dependencies. While m is fixed by the operator's pipeline depth, the size of the internal memory dictates the size of q.

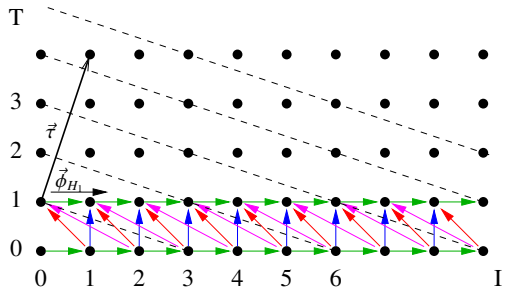


Figure 9: Inter tile slice iteration domain for Jacobi 1D stencil code. The parallel hyperplane has $\vec{\tau} = (1, 3)$ and describes the tile-slices which can be executed in parallel. The dashed lines indicated various translations of the hyperplane $H_{\vec{\tau}}$ showing different levels of parallelism.

When sufficient local memory is available, a second well known technique, *double buffering*, is used to interlacing memory access and computations. Provided we are assigned twice the local memory we need for our enhanced blocking, $2 \times 2(p \times q)$, the idea is to fetch the next set of blocks from A and B for computation at time $t + 1$ while performing the computing stage at time t. This said, when a variable is reused on successive tiles, it is better to load it one time for all, and to avoid reloading it for each tile. An exact solution to this problem has been found recently [38]. The objective now is to try to reuse the same fetched block as much as possible.

The execution schedule is optimized such to maximize the use of the A block buffer. Successive blocks of A and B (A is by far more costly with a size of $m \times q$ whereas B has a size $q \times 1$) are fetched from the memory in the direction of j for A and i for B. Once the edge is reached (say we have finished processing $A_{0,1} \times B1, 0$), we keep $A_{0,1}$ (which would be costly to discard) and we load B1,1 instead. We can clearly execute the accumulation on C iterating from $N - 1$ towards 0. This saves an important amount of external memory accesses particularly when implementing the double buffering technique.

Now, finally we consider using multiple processing elements to accomplish the task. It is easy too see that up to m PEs can work on the same block of A and on m different blocks of B ($B_{ml,m(l+1)-1}$). The local memory requirement is as much $2 \times m \times q$ for such a case (m PEs). The size of m can be increased within reasonable limits due to the embedded memories which can act as shift-registers in modern FPGA devices. Nevertheless, it is much more likely that the external memory bandwidth will be the real limitation.

5.2 One dimensional Jacobi stencil computation

In this section we will present two solutions to parallelize the Jacobi 1D stencil execution. The first solution is based on classical parallel execution of tile slices. Consider the execution of the tile slices in Figure 8. Finding what tile slices can be executed in parallel reduces to finding a hyperplane parallel $H_{\vec{\tau}}$ which in the new iteration domain of the tile slices.

The new iteration domain and the corresponding hyperplane $H_{\vec{\tau}}$ are depicted in Figure 9. The normal vector $\vec{\tau} = (1, 3)$ indicates that the maximum degree of parallelism is

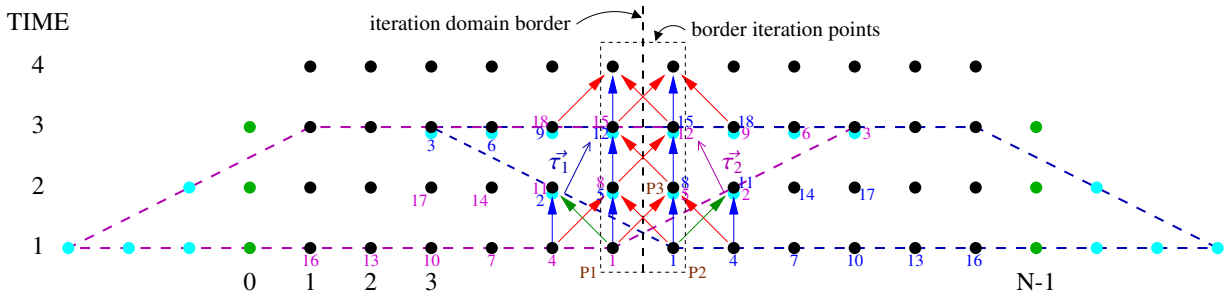


Figure 10: An alternative to executing the Jacobi Kernel using 2 processing elements.

$\lceil N/3 \rceil$. One could increase this to $\lceil N/2 \rceil$ at the expense of performing a different tiling than Figure 8 shows. In the new tiling the tile slices at $T = 1$ would be described by the transition of the same hyperplane $H_{\vec{\tau}}$ as for $T = 0$. This increase the complexity of the border conditions (where we propagate or execute virtual points). We believe that the complexity of the conditions in such an implementation would severely affect the performance of our FSM and we did not consider it further.

Our second proposed parallelization solution will be described next. It was initially supposed to be example-specific, however its execution can be extended to some reduced set of application classes presenting dependence symmetries. The benefits of this solution are: a wider degree of parallelism in execution and a reduced local memory size.

Figure 10 presents the basic principle behind our proposed solution for two PEs. The iteration domain is split into two parts (suppose for clarity that N is even in this example): right part is tiled as previously described in Figure 8 and the left part part tiling is mirrored (symmetrical) to that on the right.

The tile slices intersect the neighboring iteration domains. The set of points described by this intersection represent virtual iteration points.

The border iteration points carry the dependencies between the tile slices of neighboring iteration domains. On these points, the green incoming dependence represents a datum computed by neighboring PE which must be communicated. Thanks to the symmetry of the execution schedule, two symmetric iteration points are executed at the same time. This means that two symmetric border iteration points are executed at the same time. Consider for example the iteration points executed at time 1 on Figure 10, say P1 on the left and P2 on the right, and consider the red dependence starting from P1 to a point P3 executed by the right PE. The corresponding datum should be communicated exactly at the execution of P3, which is the same as the symmetric of P3 in the left PE. This means that the left PE should communicate the datum as for a vertical dependence.

From the architecture perspective this involves widening the green multiplexer of each accelerator with one input from the neighboring blue extraction point and modifying the select line of the multiplexer so to fetch the correct data for these border points.

Figure 11 illustrates the simplicity of this architecture. When recursively instantiating multiple pairs of accelerators the tails of the tile slices will similarly overlap. The border iteration point at this intersections will be solved by the blue dependency from

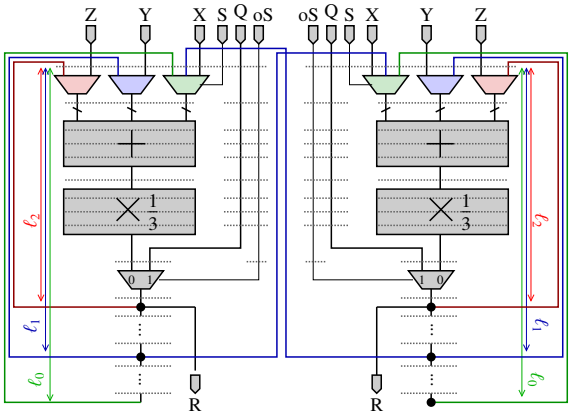


Figure 11: Architecture for the second proposed parallelization of Jacobi 1D

neighbor. Consequently, the red multiplexer will have a third input fed from from the second neighbor's blue dependency.

Notice that this method could be easily applied to any stencil computation. The only difficulty is to insert a wire to communicate the data at the relevant time. Indeed, it can happen that the symmetric of P3 is not targetted by a dependence starting from P1. In this case, the execution distance with P1 should be computed as in the step c, and extra wire/registers should be added.

5.3 Lessons

In this section, we have derived by hand several parallel pipelined accelerators by following different methodologies. We have started from the sequential accelerators generated with the technique described in the previous section.

For data parallel examples like matrix multiplication the parallelization is trivial and consists in instantiating multiple parallel computational cores each having assigned a subdomain of the global iteration domain.

Unfortunately, for examples like Jacobi 1D, the parallelization is not trivial. Due to many data dependencies, the parallel hyperplanes are skewed. There exist an infinite number of such parallel hyperplanes. One has to chose a tradeoff between maximizing the parallelism and not increasing dramatically the number of delay registers. The second solution that consists in cutting the domain into subdomains which execute using a mirror-like schedule seems to be more adapted for stencil examples as it benefits the most from FPGA structure and fast direct links between adjacent computational cores. This solution should be used for stencil examples on FPGA platforms and could be easily automatized.

6 Computing kernel accuracy and performance

In this section we show, on our two working examples that the accelerator's implementation cost can be significantly reduced by designing operators which account for the application's accuracy requirements. In other words, given an average target relative

Table 1: Minimum, average and maximum relative error out of a set of 4096 runs, for $N = 4096$, the elements of A and B are uniformly distributed on the positive/entire floating-point axis. The third architecture uses truncated multipliers having an error of 1 `ulp` with `ulp` = 2^{-w_F-6}. Implementation results are given for a Virtex-4 speedgrade-3 FPGA device

Architecture	Sign	Min	Average	Max	Performance
SP in/out, SP intern	+	1.55e-08 (2^{-25})	5.19e-05 (2^{-14})	1.06e-04 (2^{-13})	21 clk, 368MHz, 565 sl., 4 DSP
	±	3.00e-11 (2^{-34})	9.27e-06 (2^{-16})	1.68e-03 (2^{-9})	
SP in/out, DP intern	+	9.34e-10 (2^{-29})	4.72e-07 (2^{-21})	1.49e-06 (2^{-19})	32 clk, 308MHz, 1656 sl., 16 DSP
	±	3.00e-11 (2^{-34})	3.99e-06 (2^{-17})	8.42e-04 (2^{-10})	
SP in/out, $w_F + 6$ intern	+	1.11e-10 (2^{-33})	5.29e-07 (2^{-20})	1.64e-06 (2^{-19})	22 clk, 334MHz, 952 sl., 1 DSP
	±	3.02e-11 (2^{-34})	5.14e-06 (2^{-17})	1.29e-03 (2^{-9})	

error (which roughly gives average number of valid result bits) we give here an heuristic for choosing the intermediary floating-point formats based on a worst case error analysis. The validity of these heuristics is then tested on several examples.

6.1 Matrix-matrix multiplication

Let's consider the matrix-matrix multiplication $C \leftarrow AB$, where the elements of these matrices are floating-point numbers having w_E bits for representing the exponent and w_F bits for representing the fraction.

The standard iterative operator used in matrix-matrix multiplication performs $\sum_{k=0}^{N-1} a_{i,k}b_{k,j}$. For relatively small values of N this sum can be performed in parallel. For larger values of N an iterative operator $c_{i,j} \leftarrow c_{i,j} + a_{i,k}b_{k,j}, k \in 0..N-1$ is used.

The iterative operator implementation requires assembling one FP multiplier and one FP adder which serves as an accumulator. First, we consider that the elements of the input matrices A and B are exact and the instantiated FP operators employ the round-to-nearest rounding mode (the result of a calculation is rounded to the nearest floating-point number).

We denote by $fl(\cdot)$ the evaluation in floating-point arithmetic of an expression and we assume that the basic arithmetic operators $+, -, \cdot, /$ satisfy:

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), |\delta| \leq \texttt{ulp}/2$$

In plain words we state that the maximum rounding error introduced by one of the above basic operations is bounded by 1/2 `ulp` and is in average 1/4 `ulp`.

During the iterative calculation of $c_{i,j}$ (a dot product between one vector of A and one of B) the rounding errors build-up at each iteration. Possible cancellations at each iteration prevent us from finding a practical static error bound in the general case. Therefore, we decide to provide an approximate static error bound, for each element of c by discarding the cancellation effects [39]. Let's consider as an example the dot product between two vector having two elements:

$$\begin{aligned}
\widehat{p_0} &= a_0b_0(1+\delta_0)\\
\widehat{p_1} &= a_1b_0(1+\delta_1)\\
\widehat{s_0} &= (\widehat{p_0} + \widehat{p_1})(1+\delta_2)\\
&= a_0b_0(1+\delta_0)(1+\delta_2) + a_0b_0(1+\delta_1)(1+\delta_2)
\end{aligned}$$

From here on we don't wish to distinguish between the δ_i so we use a notation due to Higham [39] which denotes products of the form $(1+\delta_i)...(1+\delta_{i+k-1})$. with $(1\pm\delta)^k$. Using this new notation, the error the N-length dot-product kernel is:

$$\begin{aligned}\widehat{c_N} &= (\widehat{c_{N-1}} + a_{i,N-1}\ b_{N-1,j}(1\pm\delta))(1\pm\delta)\\ &= a_{i,0}\ b_{0,j}(1\pm\delta)^N + \sum_{k=1}^{N-1} a_{i,k}\ b_{k,j}(1\pm\delta)^{N+1-k}\end{aligned}$$

A simplified way to express this, due to Higham [39] is using the following notation:

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1+\theta_n, \rho_i \in \{-1,1\}$$

where:

$$|\theta_n| \leq \frac{nu}{1-nu} = \gamma_n$$

The dot product can then be written as:

$$\widehat{c_N} = a_{i,0}\ b_{0,j}(1+\theta_N) + \sum_{k=1}^{N-1} a_{i,k}\ b_{k,j}(1+\theta_{N+1-k})$$

The error will exhibit the largest value when all sub-products have the same magnitude, and the rounding errors will all have the same sign. We will denote this bound by Δ. A well known rule of thumb [39] states that given an error bound Δ, the average error will roughly be $\sqrt{\Delta}$. The number of invalid bits due to roundings alone is bounded by $\log_2(\Delta)$ and is equal, on average to $\log_2(\sqrt{\Delta})$. This value was indeed validated experimentally as presented in Table 1. which reports the minimum, average and maximum relative errors for the vector product, the basic block in the matrix-multiplication algorithm. The input vectors have been populated using positive random numbers for one set of tests, and both positive and negative random numbers for the second set, uniformly distributed on the corresponding floating-point axis (uniformly distributed exponents).

The average relative error reported for a standard single-precision architecture using positive inputs (in order to avoid the effects of cancellation) is of the order 2^{-14}. The error bound obtained using equation 6.1 is about 4100 `ulp`. Using the previously mentioned rule of thumb, we expect that the average relative error in this case to be $\sqrt{4100} \approx 64.03$. Therefore the number of invalidated bits is equal to $\lceil\log_2(64.03)\rceil = 7$. Which gives an expected average relative error of 2^{-16} which is close to the 2^{-14} obtained experimentally.

The second architecture listed in table 1 processes the same SP input data using double-precision operators. The result is finally rounded back to single-precision. As expected, the accuracy of this architecture is improved, at a significant increase in operator size.

The third architecture processes the same SP input data using internal operators with a slightly larger precision ($wF+6$ bits). Additionally, the floating-point multiplier is is implemented using the truncated multipliers [22] (allow reducing the number of DSP blocks over classical implementations). Due to the extended fraction, the `ulp`

Table 2: Minimum, average and maximum relative error for elements of an array in the Jacobi stencil code over a total set of 4096 runs, for $T = 1024$ iterations in the time direction. The numbers are uniformly distributed within wF exponent values. Implementation results are given for a Virtex-4 speedgrade-3 FPGA device

Architecture	**Min**	**Average**	**Max**	**Performance**
SP	1.29e-11 (2^{-35})	2.56e-06 (2^{-18})	5.24e-04 (2^{-10})	32 clk, 395MHz, 954 slices
SP in/out, DP int.	1.90e-11 (2^{-38})	2.12e-08 (2^{-25})	5.83e-08 (2^{-24})	44 clk, 308MHz, 2280 slices
SP in/out, $w_F + 3$ int	1.78e-11 (2^{-35})	6.97e-08 (2^{-23})	4.53e-06 (2^{-17})	31 clk, 313MHz, 1716 slices

value for this architecture is 2^{-29}. Accounting for the lower multiplier accuracy and the final conversion back to single precision, this architecture should still be roughly 2^6 times more accurate than the SP version. Indeed, experimental results presented in table 1 confirm that the average relative error for this implementation is of the order of 2^{-20}, 2^6 times smaller than the 2^{-14} for SP.

The second row for each architecture presents same relative error values when the input numbers are uniformly distributed on the entire floating-point axis (positive and negative) making cancellations possible. In average, each run had 7 cancellations. It can be observed that in such a situation, the three different architectures report similar numbers for the relative errors. Improving accuracy in such a case could be accomplished by avoiding cancellations as much as possible, allowing the computing unit to reorder the operations on the fly. Unfortunately, the proposed scheduling solution requires deterministic execution of operations which will not be the case in such an architecture.

6.2 One dimensional Jacobi stencil computation

The Jacobi stencil computation offers similar optimization opportunities. The main statement executes the averaging of three consecutive members of array a at time t to update the middle index at time $t + 1$.

We can model the impact of the rounding errors on this code using the arithmetic model previously introduced. Consider the assembly of standard floating-point operators.

$$\begin{aligned}\widehat{a}_{t+1,k} =&(((\widehat{a}_{t,k-1} + \widehat{a}_{t,k-1})(1+\delta_1) + \widehat{a}_{t,k+1})(1+\delta_2)\frac{1}{3})(1+\delta_3)\\ =&\frac{1}{3}\left(\widehat{a}_{t,k-1}(1+\theta_3) + \widehat{a}_{t,k}(1+\theta_3) + \widehat{a}_{t,k+1}(1+\theta_2)\right)\end{aligned}$$

The error bound after T steps is of the order θ_{3T}. In the case of an FPGA architecture, this error bound can be reduced to θ_{2T} by using a 3-input adder:

$$\begin{aligned}\widehat{a}_{t+1,k} =&((\widehat{a}_{t,k-1} + \widehat{a}_{t,k-1} + \widehat{a}_{t,k+1})(1+\delta_1) \times \frac{1}{3})(1+\delta_2)\\ =&\frac{1}{3}\left(\widehat{a}_{t,k-1}(1+\theta_2) + \widehat{a}_{t,k}(1+\theta_2) + \widehat{a}_{t,k+1}(1+\theta_2)\right)\end{aligned}$$

Using the same rule or thumb we estimate that the average error for a single-precision implementation with two floating-point adders and one constant multiplier

will be $2^{-23+5} = 2^{-18}$ ($\lceil \log_2(\sqrt{|\theta_{3T}|}) \rceil = 5$). This is indeed confirmed by the data presented in Table 2.

The our specific implementation (third line in table 2) uses a fused 3-input adder in order to enhance accuracy by saving one rounding error. Moreover, it uses an extended format of $wF+3$ bits. The average error in `ulps` one would expect from this implementation is $\lceil \log_2(\sqrt{|\theta_{2T}|}) \rceil = 4$ which invalidates 4 lower bits. Fortunately, the extended precision should absorb 3 of those, leaving the relative error of the order 2^{-22}. This is indeed confirmed by Table 2.

6.3 Lessons

The heuristic we propose is very simple, works for codes involving the basic operations: $+, -, \times, \div, \sqrt{x}$ working in floating-point arithmetic. The first task consists in defining the average accuracy requirement of the application (how many bits we expect, on average to be valid in our result), which we denote by γ. Why this average number of bits and not the worst case accuracy? Because in floating-point arithmetic, due to cancellations (subtraction of two very close values) errors can be amplified theoretically at every subtraction, possibly loosing all the result's accuracy.

Next, we express the accumulation of rounding errors (by discarding the possible amplifying effect of cancellations) using the model of floating-point arithmetic previously introduced (the interested reader should check the excellent book by Higham [39]). This gives us a worst case relative error (considering that no cancellations have amplified any error in the process) which we denote by Δ. We use the rule-of-thumb presented in [39]: the *average relative error* of the result is roughly equal to $\sqrt{\Delta}$. The average number of invalidated bits, due to this error is $\zeta = \lceil \log_2(\sqrt{\Delta}) \rceil$. The working precision we chose for our circuit is therefore $\psi + \zeta$ in order to attain an average output accuracy of ψ.

7 Reality Check

Table 3 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.

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.

Table 4 presents synthesis results of the parallelization for both our running examples on the StratixIII FPGA using the single precision format. As expected, due to massive parallelism and no inter parallel process communication, for matrix multiplication

Table 3: Synthesis results for the full (including FSM) MMM and Jacobi1D codes. Results obtained using using Xilinx ISE 11.5 for Virtex5, and QuartusII 9.0 for StratixIII

Application	FPGA	Precision (w_E, w_F)	Latency (cycles)	Frequency (MHz)	Resources 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	-

Table 4: Synthesis results for the parallelized MMM and Jacobi1D. Results obtained using using Quartus II 10.1 for StratixIII with $w_E = 8, w_F = 23$

Application	Par. factor	Frequency (MHz)	Resources REG	(A)LUT	M9K	DSPs
Matrix-Matrix Multiply N=128	1	308	701	614	3	4
	2	282	1317	999	5	8
	4	303	2473	1789	12	16
	8	302	4842	3291	20	32
	16	281	9582	6291	32	64
Jacobi1D stencil N=1024 T=1024	1	311	1217	1199	9	-
	2	295	2394	2095	21	-
	4	283	4600	3853	38	-
	8	274	9018	7314	69	-
	16	251	17806	14218	132	-

example the scaling in terms of resources is proportional to the parallelization factor. The maximum operating frequency remains fairly constant. Jacobi 1D scales very well too. A small increase in utilized resources is due to the increase in the multiplexer size in order to fit signals from neighbor computational cores. The frequency remains fairly constant. This proves that our method is well suited for FPGA implementation.

There exists several manual approaches like the one described in [40] 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.

8 Conclusion and Future Work

In this article, we have shown how the polyhedral compilation framework can be used to derive efficient hardware accelerators assuming a datapath with pipelined arithmetic

operators. We target FPGAs as our arithmetic does, but the compilations techniques presented here (scheduling and code generation) are very general and could be applied for other hardware categories. Our work has several contributions.

First, we have presented a novel approach to derive automatically an efficient, sequential, hardware with accurate pipelined arithmetic. We used state-of-the-art polyhedral compilation techniques to reschedule the kernel execution so that the arithmetic pipelines are used optimally. 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.

Second, we have shown how efficient parallel hardware can be designed, starting from automatically derived sequential hardware. Particularly, we show how to produce a parallel hardware for stencil computations in a semi-automatic way. As a bonus, the communications between processing elements are minimal with our scheme.

Finally, we have presented a heuristic method that given the average target accuracy for an application allows dimensioning the internal floating-point arithmetic data-path to obtain this accuracy. This technique can be easily automated and integrated in the same compiler tool. The savings in terms of resource usage implied by this technique are significant.

In the future, it would be interesting to extend our technique to non-perfect loop nests. This requires to consider each assignment as a process, the whole kernel being a network of communicating processes. Several model of process networks can be investigated, depending on the communication medium between processes (FIFOs or buffers).

As for many other HLS tools, the HLS flow described in this article focuses on optimizing the computational part, assuming the availability of the data. We have shown in a previous work [38] how to generate the hardware to prefetch the data from the DDR, with minimal DDR accesses and local memory size. Again, this technique works well for perfect loop nest, but its extension to unperfect loop nest remains a challenge that must be tackled at the same time as kernel scheduling presented in the previous paragraphs.

References

[1] F. de Dinechin, B. Pasca, Designing custom arithmetic data paths with FloPoCo, IEEE Design and Test.

[2] E. Martin, O. Sentieys, H. Dubois, J.-L. Philippe, Gaut: An architectural synthesis tool for dedicated signal processors, in: Design Automation Conference with EURO-VHDL'93 (EURO-DAC), 1993. doi:10.1109/EURDAC.1993.410610.

[3] Impulse-C.
URL http://www.impulseaccelerated.com

[4] AutoESL, Autopilot datasheet (2009).

[5] Forte Design Systems: Cynthesizer, http://www.forteds.com.

[6] Synopsys: Synphony, http://www.synopsys.com/.

[7] Nios II C2H Compiler User Guide, version 9.1. http://www.altera.com (Nov. 2009).

[8] J. M. P. Cardoso, P. C. Diniz, Compilation Techniques for Reconfigurable Architectures, 2009. doi:10.1007/978-0-387-09671-1.

[9] S. Gupta, N. Dutt, R. Gupta, A. Nicolau, Spark: A high-level synthesis framework for applying parallelizing compiler transformations, International Conference on VLSI Designhttp://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/ICVD.2003.1183177 doi:http://doi.ieeecomputersociety.org/10.1109/ICVD.2003.1183177.

[10] Mentor CatapultC high-level synthesis, http://www.mentor.com.

[11] C. Alias, F. Baray, A. Darte, 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.

[12] L. Zhuo, V. K. Prasanna, High performance linear algebra operations on reconfigurable systems, in: ACM/IEEE conference on Supercomputing, IEEE, 2005. doi:http://dx.doi.org/10.1109/SC.2005.31.

[13] M. R. Bodnar, J. R. Humphrey, P. F. Curt, J. P. Durbano, D. W. Prather, Floating-point accumulation circuit for matrix applications, in: International Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2006, pp. 303–304. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/FCCM.2006.41 doi:http://doi.ieeecomputersociety.org/10.1109/FCCM.2006.41.

[14] ISE 11.4 CORE Generator IP.
URL http://www.xilinx.com

[15] MegaWizard Plug-In Manager.
URL http://www.altera.com

[16] S. Perry, Model based design needs high level synthesis: a collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2009, pp. 1202–1207.
URL http://portal.acm.org/citation.cfm?id=1874620.1874909

[17] M. Langhammer, Floating point datapath synthesis for FPGAs, in: International Conference on Field Programmable Logic and Applications, 2008, pp. 355 –360. doi:10.1109/FPL.2008.4629963.

[18] F. de Dinechin, J. Detrey, I. Trestian, O. Creţ, R. Tudoran, When FPGAs are better at floating-point than microprocessors, Tech. Rep. ensl-00174627, École Normale Supérieure de Lyon, http://prunel.ccsd.cnrs.fr/ensl-00174627 (2007).

[19] O. Creţ, F. de Dinechin, I. Trestian, R. Tudoran, L. Creţ, L. Văcariu, FPGA-based acceleration of the computations involved in transcranial magnetic stimulation, in: Southern Programmable Logic Conference, IEEE, 2008, pp. 43–48.

[20] F. de Dinechin, B. Pasca, Large multipliers with fewer DSP blocks, in: Field Programmable Logic and Applications, IEEE, 2009.

[21] N. Brisebarre, F. de Dinechin, J.-M. Muller, Integer and floating-point constant multipliers for FPGAs, Application-Specific Systems, Architectures and Processors, IEEE International Conference on 0 (2008) 239–244. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/ASAP.2008.4580184 doi:http://doi.ieeecomputersociety.org/10.1109/ASAP.2008.4580184.

[22] S. Banescu, F. de Dinechin, B. Pasca, R. Tudoran, Multipliers for floating-point double precision and beyond on FPGAs, in: International Workshop on Higly-Efficient Accelerators and Reconfigurable Technologies (HEART), ACM, 2010.

[23] F. de Dinechin, B. Pasca, O. Creţ, R. Tudoran, An FPGA-specific approach to floating-point accumulation and sum-of-products, in: Field-Programmable Technologies, IEEE, 2008.

[24] F. de Dinechin, M. Joldes, B. Pasca, Automatic generation of polynomial-based hardware architectures for function evaluation, in: 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Rennes, 2010.

[25] F. de Dinechin, M. Joldes, B. Pasca, G. Revy, Multiplicative square root algorithms for FPGAs, in: Field Programmable Logic and Applications, IEEE, 2010.

[26] F. de Dinechin, A flexible floating-point logarithm for reconfigurable computers, Lip research report RR2010-22, ENS-Lyon (2010).
URL http://prunel.ccsd.cnrs.fr/ensl-00506122/

[27] F. de Dinechin, B. Pasca, Floating-point exponential functions for DSP-enabled FPGAs, in: Field Programmable Technologies, IEEE, 2010.
URL http://prunel.ccsd.cnrs.fr/ensl-00506125/

[28] P. Feautrier, C. Lengauer, The polyhedron model, Encyclopedia of Parallel Computing.

[29] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, O. Temam, Putting polyhedral loop transformations to work, in: International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2003.

[30] F. Irigoin, R. Triolet, Supernode partitioning, in: 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1988.

[31] J. Xue, Loop Tiling for Parallelism, Kluwer Academic Publishers, 2000.

[32] P. Feautrier, Parametric integer programming, RAIRO Recherche Opérationnelle 22 (3) (1988) 243–268.

[33] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: ACM International Conference on Programming Languages Design and Implementation (PLDI), 2008.

[34] A. W. Lim, M. S. Lam, Maximizing parallelism and minimizing synchronization with affine transforms, in: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1997.

[35] P. Clauss, 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.

[36] Polylib – A library of polyhedral functions, `http://www.irisa.fr/polylib`.

[37] P. Boulet, P. Feautrier, Scanning polyhedra without Do-loops, in: IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), 1998.

[38] A. Plesco, Program transformations and memory architecture optimizations for High-Level Synthesis of hardware accelerators, Ph.D. thesis, École Normale Supérieure de Lyon (2010).

[39] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd Edition, SIAM, Philadelphia, PA, 2002.

[40] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev, 64-bit floating-point FPGA matrix multiplication, in: ACM/SIGDA symposium on Field-Programmable Gate Arrays (FPGA), 2005.

9 Appendix

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_signed.all;
5
6  entity test is
7    generic(Nc0       : integer := 12;
8            Nc1       : integer := 12;
9            Nc2       : integer := 12;
10           RAMADDRBITSA : integer := 10
11           );
12   port(
13     clk             : in std_logic;
14     reset           : in std_logic;
15     flopoco_feedback : out std_logic;
16     flopoco_ins_input : out std_logic;
17     addressA0       : out std_logic_vector (RAMADDRBITSA-1 downto 0);
18     addressA1       : out std_logic_vector (RAMADDRBITSA-1 downto 0);
19     addressA2       : out std_logic_vector (RAMADDRBITSA-1 downto 0);
20     addressAw       : out std_logic_vector (RAMADDRBITSA-1 downto 0);
21     weA             : out std_logic;
22     done            : out std_logic
23     );
24 end test;
25
26 architecture fsm of test is
27   type  states is (start, stop, S0);
28   signal state, next_state    : states;
29   signal counter_0, next_counter_0 : std_logic_vector(Nc0-1 downto 0);
30   signal counter_1, next_counter_1 : std_logic_vector(Nc1-1 downto 0);
31   signal counter_2, next_counter_2 : std_logic_vector(Nc2-1 downto 0);
32   signal condlt               : std_logic_vector (Nc1-1 downto 0);
33   signal actual_ii            : std_logic_vector (Nc1-1 downto 0);
34   signal actual_iim1          : std_logic_vector (Nc1-1 downto 0);
35   signal actual_iip1          : std_logic_vector (Nc1-1 downto 0);
36 begin
37   actual_iim1 <= actual_ii - 1;
38   actual_iip1 <= actual_ii + 1;
39   addressA0 <= actual_iim1(RAMADDRBITSA-1 downto 0) when ((actual_ii >= 0) and (actual_ii ↩
         <= 1023)) else (others => '0');
40   addressA1 <= actual_ii(RAMADDRBITSA-1 downto 0) when ((actual_ii >= 0) and (actual_ii <= ↩
         1023)) else (others => '0');
41   addressA2 <= actual_iip1(RAMADDRBITSA-1 downto 0) when ((actual_ii >= 0) and (actual_ii ↩
         <= 1023)) else (others => '0');
42   addressAw <= actual_ii(RAMADDRBITSA-1 downto 0) when ((actual_ii >= 0) and (actual_ii <= ↩
         1023)) else (others => '0');
43
44   automaton : process(clk, reset)
45   begin
46     if clk = '1' and clk'event
47     then
48       if reset = '1' then
49         state <= start;
50       else
51         state    <= next_state;
52         counter_0 <= next_counter_0;
53         counter_1 <= next_counter_1;
54         counter_2 <= next_counter_2;
55       end if;
56     end if;
57   end process automaton;
58
59   condlt   <= (counter_2-(counter_0(Nc0-6 downto 0)&"00001"));
60   actual_ii <= (counter_1 - (condlt(Nc1-2 downto 0) & '0'));
61
62   flopoco_ins_input <= '1' when ((actual_ii = 0) or (actual_ii = 1023)) else '0';
63 -- -- purpose: drives flopoco ins input
64 -- -- type  : combinational
65 -- -- inputs : actual_ii
66 -- -- outputs: flopoco_ins_input
67
```

```
68   flopoco_feedback <= '0' when (counter_2 = counter_0(Nc0-6 downto 0)&"00001") else '1';
69  -- -- purpose: drives flopoco feedback signal
70  -- -- type  : combinational
71  -- -- inputs : counter_0, counter_2
72  -- -- outputs: flopoco_feedback
73
74     weA <= '1' when ((counter_2 = counter_0(Nc0-6 downto 0)&"00001" + 31) and (actual_ii > ↩
           0) and (actual_ii < 1023)) else '0';
75  -- -- purpose: enables writes to memory
76  -- -- type  : combinational
77  -- -- inputs : counter_2, counter_0
78  -- -- outputs: weA
79
80   first_and_next : process(state, counter_0, counter_1, counter_2)
81   begin
82     done <= '0';
83     next_state   <= stop;
84     next_counter_0 <= counter_0;
85     next_counter_1 <= counter_1;
86     next_counter_2 <= counter_2;
87
88     case state is
89       when stop =>
90         done <= '1';
91       when start =>
92         if (true)
93         then
94           next_state   <= S0;
95           next_counter_0 <= conv_std_logic_vector(0, Nc0);
96           next_counter_1 <= conv_std_logic_vector(0, Nc1);
97           next_counter_2 <= conv_std_logic_vector(1, Nc2);
98         end if;
99       when S0 =>
100        if ((counter_0(Nc0-6 downto 0)&"11111" >= counter_2) and (counter_0(Nc0-6 downto 0)&↩
             "00000" <= counter_2) and (31 >= counter_0) and (1084 >= counter_1) and (↩
             counter_1 >= 0))
101        then
102          next_state   <= S0;
103          next_counter_0 <= counter_0;
104          next_counter_1 <= counter_1;
105          next_counter_2 <= conv_std_logic_vector(1, Nc2)+counter_2;
106        end if;
107        if ((1083 >= counter_1) and (counter_0 (Nc0-6 downto 0)&"11111" < counter_2) and (↩
             counter_0 (Nc0-6 downto 0)&"00000" <= counter_2) and (31 >= counter_0) and ↩
             (1084 >= counter_1) and (counter_1 >= 0))
108        then
109          next_state   <= S0;
110          next_counter_0 <= counter_0;
111          next_counter_1 <= conv_std_logic_vector(1, Nc1)+counter_1;
112          next_counter_2 <= conv_std_logic_vector(1, Nc2)+(counter_0 (Nc0-6 downto 0)&"00000"↩
               );
113        end if;
114        if ((30 >= counter_0) and (1083 < counter_1) and (counter_0 (Nc0-6 downto 0)&"11111"↩
             < counter_2) and (counter_0 (Nc0-6 downto 0)&"00000" <= counter_2) and (31 >= ↩
             counter_0) and (1084 >= counter_1) and (counter_1 >= 0))
115        then
116          next_state   <= S0;
117          next_counter_0 <= conv_std_logic_vector(1, Nc0)+counter_0;
118          next_counter_1 <= conv_std_logic_vector(0, Nc1);
119          next_counter_2 <= conv_std_logic_vector(33, Nc2)+(counter_0 (Nc0-6 downto 0)&"00000↩
               ");
120        end if;
121        if (((-counter_0 (Nc0-6 downto 0)&"00000")+counter_2 < 0) and (31 >= counter_0) and ↩
             (1084 >= counter_1) and (counter_1 >= 0))
122        then
123          next_state   <= S0;
124          next_counter_0 <= counter_0;
125          next_counter_1 <= counter_1;
126          next_counter_2 <= conv_std_logic_vector(1, Nc2)+(counter_0 (Nc0-6 downto 0)&"00000"↩
               );
127        end if;
128        if ((30 >= counter_0) and (1084 < counter_1) and (counter_1 >= 0))
129        then
130          next_state   <= S0;
```

```
131         next_counter_0 <= conv_std_logic_vector(1, Nc0)+counter_0;
132         next_counter_1 <= conv_std_logic_vector(0, Nc1);
133         next_counter_2 <= conv_std_logic_vector(33, Nc2)+(counter_0 (Nc0-6 downto 0)&"00000↩
                ");
134       end if;
135       if ((31 >= counter_0) and (1083 >= counter_1) and (counter_1 < 0))
136       then
137         next_state   <= S0;
138         next_counter_0 <= counter_0;
139         next_counter_1 <= conv_std_logic_vector(1, Nc1)+counter_1;
140         next_counter_2 <= conv_std_logic_vector(1, Nc2)+(counter_0 (Nc0-6 downto 0)&"00000"↩
              );
141       end if;
142     when others =>
143   end case;
144  end process first_and_next;
145 end fsm;
```

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399