High Performance Computing with FPGAs

Erik H. D'HOLLANDER [a,1], Kristof BEYLS [b]
[a] *Ghent University*
Department of Electronics and Information Systems
9000 Ghent, Belgium
[b] *ARM Ltd., Cambridge, UK*

Abstract
Field-programmable gate arrays represent an army of logical units which can be organized in a highly parallel or pipelined fashion to implement an algorithm in hardware. The flexibility of this new medium creates new challenges to find the right processing paradigm which takes into account the natural constraints of FPGAs: clock frequency, memory footprint and communication bandwidth. In this paper first the use of FPGAs as a multiprocessor on a chip or its use as a highly functional coprocessor are compared, and the programming tools for hardware/software codesign are discussed. Next a number of techniques are presented to maximize the parallelism and optimize the data locality in nested loops. This includes unimodular transformations, data locality improving loop transformations and use of smart buffers. Finally, the use of these techniques on a number of examples is demonstrated. The results in the paper and in the literature show that, with the proper programming tool set, FPGAs can speed up computation kernels significantly with respect to traditional processors.

Keywords. FPGA, data locality, high performance, loop transformations

Introduction

Modern FPGAs consist of logic blocks, gates, memories, ALUs and even embedded processors, which can be arbitrarily interconnected to implement a hardware algorithm. Because of their flexibility and growing capabilities, field programmable gate arrays are the topic of intensive research in high-performance computing.

The high reconfigurability and inherent parallelism creates a huge potential to adapt the device to a particular computational task. At the same time, the embodiment of a hardware algorithm constitutes a departure from the classical Von Neumann or Harvard processor architecture. New computing paradigms need to be explored in order to exploit FPGAs to their full potential [8,22]. This involves a thorough knowledge of the design constraints, the development environment, the hardware description tools and the methodology for mapping an algorithm onto the hardware. On the other hand, traditional performance metrics such as instructions per cycle, clock speed, instructions

[1]Corresponding author: Erik H. D'Hollander, e-mail: dhollander@elis.ugent.be

per second, have no direct meaning for an algorithm that is directly executed in hardware [26,21,35]. In order to give an overview of the possibilities and challenges of this new and rapidly evolving technology, the following topics will be covered. First the hardware of the FPGAs is discussed. Next the different constellations of FPGAs used in high-performance computing are explored. An important aspect is the software tools required to configure the hardware, in particular the number of high-level languages which facilitate hardware-software codesign. Finally, a number of program transformations are described which improve the execution speed, data locality and performance of the hardware designs.

1. Architecture, computing power and programmability

Semantically, a Field-Programmable Gate Array (FPGA) means an array of gates which can be arbitrarily interconnected by a configuration program that describes the paths between the logic components. After programming, the FPGA is able to implement any logic function or finite state machine (see fig. 1).

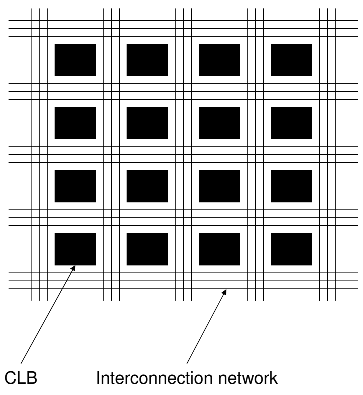

Figure 1. Schematic representation of an FPGA architecture: CLBs (configurable logic blocks) and interconnection network.

Since its conception in the 1980s, the granularity and the complexity of the logic blocks has much evolved. Apart from the fine-grained AND, XOR and NOT gates, the most important blocks are lookup tables or LUTs, which are able to implement a logic function of 4 to 6 inputs. Flip-flops are able to store the state of a calculation, and turn the FPGA into a finite state machine. In addition to the fine-grained cells, most FPGAs have medium grain four-bit ALUs which can be joined to form ALUs of arbitrary precision, as well as coarse grain components, e.g. word size ALUs, registers and small processors with an instruction memory. All these components can be arbitrarily interconnected using a two-dimensional routing framework. More recently, heterogeneous FPGAs contain embedded multiplier blocks, larger memories and even full-fledged microprocessors, such as the PowerPC in the Vertex family of Xilinx, all integrated in one chip. These developments make the FPGA all the more attractive.

1.1. FPGAs versus ASICs

The 'algorithm in hardware' concept may also be implemented in an application-specific integrated circuit. ASICs are used to improve the computational tasks by implementing algorithmic functionality into optimized HPC hardware, e.g. a JPEG encoder [28].

Similar design steps

Both FPGAs and ASICs have a similar design cycle, which involves the behavioral description of the hardware, the synthesis, placement and routing steps. First the algorithm is described in a hardware description language, such as VHDL[2] or Verilog. Next the behavioral description is converted into a hardware realization using logic synthesis. This step is comparable to the translation of a high-level program into machine instructions. However the results of the translation process is not an executable program, but an algorithm implemented in hardware. The synthesis step uses the logic components available in the FPGA or the ASIC target, similar to a compiler using the machine instructions available in a processor. Next, the components are mapped or placed onto the substrate and finally in the routing step the components are interconnected.

Different design implementation

There are significant differences between FPGAs and ASICs.

- The major advantage of an FPGA is that the design cycle is greatly reduced. The synthesis, mapping and routing is done using an integrated toolset which produces a bitstream file. The bitstream file describes the way in which the logic components are interconnected and configured. This file is loaded into static RAM and programs the FPGA at start up in a number of milliseconds. The whole process from hardware description to bitstream generation takes an order of minutes or hours. On the other hand, the development of an ASIC prototype requires weeks or months for a comparable design cycle. The prototype has to be tested and validated, and possibly resubmitted to generate a new design.
- The FPGA is reconfigurable, implementing a new bitstream means a new hardware design. Therefore, FPGAs can be reused for a new task in an HPC system. The reconfiguration of an FPGA can even occur at runtime. Using reprogramming, the same FPGA can be used in different phases of a program to carry out compute intensive tasks.
- On the other hand, ASICs consume less power, less area, and deliver a higher performance than FPGAs. This is not a critical limitation, since successful static or fixed FPGA designs may be burned into ASICs.

1.2. FPGA characteristics and constraints

The versatility of FPGAs comes at a price: there is no clear-cut computing paradigm for FPGAs. Reconfigurable computing may embody the traditional processor as a soft-core processor, act as an independent finite state machine, or operate as a special-purpose processor in conjunction with a CPU or even other FPGAs.

[2]VHDL = VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

FPGAs typically have a small onboard memory, no cache and a slow connection to external memory. Although onboard memory is a critical resource (typically less than 100Mb block RAM or BRAM), the memory is dual-ported and the aggregated memory bandwidth is in order of 100 GB/s [39]. The absence of a cache avoids cache coherency problems, but the nonuniform memory hierarchy has to be taken into account. The use of a high-speed scratchpad memory may be used to transparently optimize the data access [3,25]. Novel techniques such as data reuse analysis in conjunction with a scratchpad memory [23] have shown to outperform cache based architectures. FPGAs are bound to optimize data locality for an efficient operation, e.g. by reusing data in fast memory, and optimizing the use of registers, see section 7 .

Most FPGA designs are limited to fixed point arithmetic, because floating-point designs are complex. Furthermore, the clock frequency depends on the characteristics of the design and is typically one order of magnitude lower than the clock frequency of current processors.

To summarize, the versatility and programmability, together with the unbounded parallelism make FPGAs an ideal choice to map parallel compute kernels onto fast dedicated hardware designs. Topical issues are:

- what computing paradigm is most suitable?
- which approach is best fitted to program HPC FPGAs?
- how to partition a program into a hardware (FPGA) and software (processor) codesign?
- what type of applications benefits most from a hardware/software architecture?
- which program transformations optimize FPGA-based parallelism and data locality?

These questions are addressed in the following sections, and answers, results and new concepts from academia and industry are presented.

2. FPGA computing paradigms

2.1. FPGA as multiprocessor: MPSoC

Traditional computer architectures are built using a data path and a control path. In a typical instruction cycle, the control decodes the instruction and configures the data path; the data path performs the operations on the data and actually executes the instruction. The configuration of the data path by the control ensures that the proper instruction is executed. The flexibility and size of the FPGAs allows to duplicate the processor architecture as a soft core circuit. Modern FPGAs allow to instantiate 20 and more soft core processors, or *soft multiprocessors* [24]. Attractive as they may be, these designs face competition with hardware *Multiprocessor Systems on a Chip* (MPSoCs), which have a higher performance and lower energy consumption. As a matter of fact, the microprocessor companies deliver multicores today and announced that multicore performance is building up [10]. However, the reconfigurability of FPGAs make this platform an attractive testbed for future multicores [31]. The RAMP Gold (Research Accelerator for Multiple Processors) uses a multithreaded Sparc instruction set to run applications on a multicore prototype architecture simulated on an array of Xilinx Virtex FPGAs.

Message passing or shared memory?

Multiple processors communicate and collaborate either by storing data and semaphores in shared memory or by exchanging messages over dedicated channels. A shared memory between two or more processors requires that there is one common address space. Setting up a shared memory involves first an addressing logic which accesses the block RAMs spread over the FPGA, and secondly a common bus with arbitration logic controlling the access of the different processors to the shared memory. In [38] a shared memory based multiprocessor with an hierarchical bus structure has been presented, yielding a speedup of 3.2 on 4 processors. In [2] a testbed is developed to emulate a multiprocessor with shared memory and a cache coherency protocol. The system is used as a research environment to study new parallel architectures.

Despite the rising interest in multi-cores and multiprocessors on a chip, a multiprocessor on an FPGA is bound to face difficulties due to lack of support for a large shared memory. In contrast, the regularity of the logic and interconnections on an FPGA makes it more suitable to function as a pipelined data path instead of implementing a full-functional processor. In fact, most FPGA developers have opted to provide FPGAs with embedded processors, including intellectual property (IP) cores for several types of buses linking the embedded processor with the rest of the FPGA. Examples are the On-chip Peripheral Bus (OPB) and Fast Simplex Link (FSL) buses for the PowerPC processor in the Virtex Xilinx FPGAs. However, no arbitration logic is yet provided for a multi-master bus, and most solutions opt for a point-to-point link between the processors, using message passing.

Considering the message passing alternative, FPGAs have many input-output connections and are very suitable to operate as processing elements in a network of computers, or as an accelerator for dedicated operations in a supercomputer, such as the Cray XD1 [36]. In this area several initiatives have been taken to implement the message passing interface, MPI, as a communication fabric onto the FPGA. This allows FPGAs to communicate with each other using well-defined standards. Examples are TMD-MPI [32] implementing a lightweight subset of MPI, which is designed for systems with limited or no shared memory, no operating system and low processing overhead, such as Toronto's TMD scalable FPGA-based multiprocessor [29]. A multi-FPGA system consists of 9 soft MicroBlaze processors interconnected with a Fast Serial bidirectional Link (FSL) on a single FPGA and a megabit transceiver (MGT) connecting several FPGAs. A speedup of 32 on 40 processors executing Jacobi's algorithm for the heat equation is reported [33]. In [40] an FPGA based coprocessor is presented that implements MPI primitives for remote memory access between processors of a multiprocessor.

2.2. FPGA as accelerator

Since on-chip memory is limited, large data sets need to be stored off-chip, e.g. in a processor's memory. Furthermore, using the hardware/software codesign concept, an FPGA can be easily configured to perform specialized computations using a tailored hardware algorithm. One of the major bottlenecks in such a configuration is the communication bandwidth gap between the processor running at GHz and the FPGA running at MHz speed. Cray uses the RapidArray communication processor which delivers a 3.2 GB/s bidirectional interconnect with 6 Virtex-II Pro FGPAs per chassis [9]. This allows to download a computation kernel, such as a tiled matrix multiply, which executes 9 times faster on the FPGAs than on the XD1's own Opteron CPU [6].

Silicon Graphics uses the RC100 blade [34] to speed up applications with reconfigurable computing. A blade contains dual Xilinx Virtex 4 LX200 FPGAs and the bandwidth to the processor memory is 6.4 GB/s.

Another area where the FPGA excels is its use as simulator of new computer architectures. The design of new architectural features such as prediction buffers, cache sizes, number and function of processing elements requires a cycle-accurate simulation. By construction, FPGAs are excellently placed to emulate many of the features studied in new architectural designs. The use of FPGAs to speed up the simulations has two major advantages:

1. the emulation is orders of magnitude faster than simulation;
2. the emulated architecture can be monitored directly with hardware probes located in the FPGA at the points of interest.

The whole concept of using FPGAs for hardware design is present in the RAMP prototyping system [2], which serves as a testbed for a large number of projects. In another publication [37], the simulation of multiprocessors or CMPs is expedited by orders of magnitude using FPGAs.

2.3. FPGA as high-performance coprocessor

A proficient use of FPGAs is as a collaborative specialized hardware algorithm running under control of a separate processor. This is the best of both worlds: a processor is used to control the flow of the algorithm and the FPGA is used to execute a specialized fast hardware implementation of the algorithm. High performance computers are equipped with reconfigurable FPGAs to operate as fast specialized computing elements, accessed, controlled and addressed by the associated processor. The FPGAs are either connected to a global shared memory (e.g. the SGI-Altix/RASC [34]) or directly to a companion high performance processor (e.g. the Cray-XD1 [9]). The first configuration has more possibilities to assign a farm of FPGAs for a particular job, while the second organization provides a faster and direct link between the processor and the associated FPGA. Both approaches have proven to be able to provide several orders of magnitude speedup on the right applications, e.g. DES breaking or DNA and protein sequencing [16].

To obtain these huge speedups requires a blend of a number of hardware, software and application characteristics, which are summarized as follows.

- A tight high-speed, low latency connection between processors and FPGAs.
- A compute-intensive problem with inherently massive regular parallelism in fixed-point arithmetic. Floating-point IP cores take a lot of chip area and are much slower than integer arithmetic.
- The problem must have good data locality. The data movement between the processor and the FPGA as well as within an FPGA remains slow with respect to the low-level parallel operations in the FPGA. Many successful FPGA applications operate in a single program multiple data (SPMD) fashion, by replicating a compute kernel many times and processing streams of data. Since FPGAs have a limited amount of memory, the data used for a computation should be fetched only once, such that all the computations on the data are finished in one computing step and then the data can be discarded.

- There is a need for a high level programming environment which is able to seamlessly integrate the hardware/software codesign.
- The problem lends itself easily to a hardware/software partitioning in which the software runs on the host and the hardware is executed on the FPGA.

3. FPGA programming languages

The lowest language level for programming FPGAs is VHDL. However VHDL is too hardware specific and can be regarded as the assembly language of reconfigurable computing. The lowest system-level language in wide use is C. In recent years many C-dialects have been developed to program FPGAs. Examples are Handel-C [1,27], Streams-C [17], Impulse-C [30], SystemC [18] and SPARK [20]. The major difference between these languages is their adherence to the ANSI C standard on the one hand or a C-like language with new pragmas and keywords for hardware synthesis on the other hand. The advantage of all approaches over VHDL or Verilog is the higher level of abstraction, which fosters more productivity.

The most well known C-like language with a strong hardware flavor is SystemC. SystemC is managed by the open SystemC Initiative (OSCI), and the language was recently recognized as IEEE standard Std 1666 2005. SystemC exists as a C++ class library which permits hardware-specific objects to be used within C++. The specific SystemC identifiers start with "sc_". In particular, events, sensitivity, modules, ports and hardware data types allow to express concurrency, signals and hardware structure. Handel-C has its roots in Occam, a language using the communicating sequential processes (CSP) paradigm to program transputers. Handel-C uses specific keywords to specify hardware, e.g. signals, interfaces, memory types and clocks. In [7] a test is discussed where a team of software developers without prior hardware knowledge develops an image processing algorithm in hardware using the Handel-C to VHDL compiler. The resulting hardware runs 370 times faster than the software algorithm.

Languages which comply with ANSI C mostly adhere to a well-defined computing paradigm such that the machine dependent characteristics can be mostly hidden for the programmer. An example of such a language is Streams-C or its descendant Impulse C, where the hardware part is programmed as a C function and compiled for execution on an FPGA. The communication between the software process and the hardware algorithm is based on streams of data. The code generation for the processor and the configuration generation for the FPGA is produced in one compile phase. Impulse C also generates the stubs to mediate the traffic between processor and FPGA. Impulse C is one of the languages available on the Cray and SGI supercomputers with FPGA accelerators.

4. Program transformations for parallelism

High-performance computing with FPGAs is most successful for applications with high parallelism and high data locality. In addition the compute kernel running on the FPGA should have a limited control flow in order to avoid pipeline stalling or synchronization overhead. In the area of bioinformatics or cryptography there are applications with obvious parallelism. However many common programs with inherent parallelism need to be transformed in order to map the computations on an FPGA. In these cases nested loops are often a prime candidate for a hardware implementation when the proper conditions are met. Even when there is no obvious parallelism, there exist methods to reorient the computations in order to improve the parallelism and data locality at the same time. In the following, an approach using unimodular transformations is presented, which finds the maximum number of parallel iterations and orders the computations to optimize the data locality.

4.1. Loops with uniform dependencies

Consider a perfect loop nest L:

```
for{$I_1 = 1..n_1$}
  for{$I_2 = 1..n_2$}
    ...
      for{$I_n = 1..n_n$}
        $S(I_1, I_2, ...I_n)$;
```

This loop is represented as $L = (I_1, I_2, \ldots, I_n)(S)$ where S represents the statements in the loop body and column vector $I = [I_1 \ldots I_n]^T$, with $1 \leqslant I_i \leqslant N_k;\ i = 1..n$, contains the loop indices.

Two iterations $S(I)$ and $S(J)$ are *dependent*, $S(I)\delta S(J)$, if both iterations access the same location, and at least one iteration writes into that location. Suppose iteration $S(I)$ precedes iteration $S(J)$ in lexicographical order (i.e. the execution order of the program). Several dependence relations are defined. If iteration $S(J)$ writes data which is used by iteration $S(I)$, then $S(J)$ is data dependent on $S(I)$. If iteration $S(J)$ writes data into the same location that is read by iteration $S(I)$, then $S(J)$ is anti-dependent on $S(I)$. If iterations $S(J)$ and $S(I)$ write into the same location, then $S(J)$ is output dependent on $S(I)$.

The *dependence distance vector* between two dependent iterations $S(I)$ and $S(J)$ is defined as

$$d = J - I$$

For each pair of array elements where one is a write, the dependence distance vector is the distance between the two elements. E.g. the dependence distance vectors of a loop with statement

$$A[I_1, I_2] = A[I_1 - 4, I_2 + 2] + A[I_1 - 4, I_2 - 2]$$

are $d_1 = [4, -2]^T$ and $d_2 = [4, 2]^T$. The *dependence matrix D* contains the dependence distance vectors of the loop, i.e. $D = [d_1 \ldots d_m]$ where m is the number of dependence distance vectors.

In many loops, the dependence between the array indices is fixed and the corresponding dependence matrix is constant. Loops with a constant dependence matrix have *uniform dependencies*. In these loops the array indices have the form $i + c$ where c is a constant. A canonical form of a uniform loop nest is:

```
for{$I$}
  $S(I) = F(S(I-d_1), S(I-d_2), \ldots, S(I-d_m));$
```

The loop variables S in iterations I are functions of the loop variables S in iterations $I-d_1, \ldots, I-d_m$. The dependencies between the iterations are expressed by the index expressions $I-d_i$, i.e. iteration I depends on iterations $I-d_1$, $I-d_2 \ldots I-d_m$.

Consider dependence matrix $D = [d_1 \ldots d_m]$. Then the dependence relations satisfy the following lemmas:

Lemma 1 *Two index points I_1 and I_2 are dependent if and only if*

$$I_1 - I_2 = Dy, \quad y = (y_1 y_2 \ldots y_m)^T \in \mathbb{Z}^m \tag{1}$$

□

In other words, the distance between two dependent iterations is a linear combination of the distance vectors in the dependence matrix D. Starting from an arbitrary iteration I_0, the following lemma 2 finds the set of all dependent iterations.

Lemma 2 *The iterations I dependent on a particular iteration I_0 are given by the following equation*

$$I = I_0 + Dy, \quad y \in \mathbb{Z}^m \tag{2}$$

□

Equation (2) defines a subset of dependent iterations. A loop is partitioned in a number of subsets. Every subset is executable in parallel, but the iterations in each subset need to be executed sequentially. A few questions arise:

- How are the parallel sets identified?
- What loop transformation is needed to execute the sets in parallel?

Both issues are solved by a unimodular transformation of the dependence matrix, followed by a regeneration of the loops traversal in lexicographical order.

4.2. Unimodular transformation

A *unimodular matrix* is a square matrix of integer elements with a determinant value of +1 or -1. A useful property of a unimodular matrix is that its inverse is also unimodular. Likewise, the product of two unimodular matrices is also unimodular.

Definition 1 *Unimodular transformation of the index set*

A uniform transformation of the index set $I \in \mathbb{Z}^n$ is the index set $Y = UI$ with $U_{n \times n}$ a unimodular matrix. □

Unimodular transformations are of interest for constructing parallel partitions because:

1. There is a 1-1 mapping between the index sets I and $Y = UI$.

2. The dependencies remain invariant after a unimodular transformation, i.e.

$$I_1\delta(D)I_2 \Leftrightarrow UI_1\delta(UD)UI_2.$$

   where $\delta(D)$ and $\delta(UD)$ denote dependencies with respect to dependence matrix D and UD respectively.
3. There exist algorithms to convert a dependence matrix D to a triangular or diagonal form.
4. The maximal parallel partitions of triangular or diagonal matrices is known.

The algorithm to transform a dependence matrix into unimodular form is based on an ordered sequence of elementary (unimodular) row- or column-operations:

- exchange two rows;
- multiply a row or column by -1;
- add an integer multiple of one row or column to another row or column.

The elementary transformations are obtained by a left (for rows) or right (for columns) multiplication with a unimodular matrix representing an elementary operation. Multiplication of the elementary matrices yields the unimodular transformation matrices U and V such that $D^t = DV$ or $D^d = UDV$ with D^t and D^d the conversion of the dependence matrix D into triangular or diagonal form. The algorithms for these transformations are given in [12].

Lemma 3 *The triangular dependence matrix $D^t = DV$ with V unimodular, represents the same dependencies as the dependence matrix D*

Proof. With $D = D^tV^{-1}$ the dependence equation (2) of two dependent iterations I_1 and I_2 becomes

$$I_1 - I_2 = D^tV^{-1}y, \quad y \in \mathbb{Z}^m$$

Now, let $y' = V^{-1}y$, then

$$I_1 - I_2 = D^ty', \quad y' \in \mathbb{Z}^m$$

which expresses that I_1 and I_2 are dependent under dependence matrix D^t. □

As a result, it is possible to parallelize the loops and calculate the new loop boundaries using the triangular dependence matrix.

4.3. Loop partitioning

With a triangular dependence matrix $D^t = DV$, obtained by a unimodular transformation of the dependence matrix D, the dependence set, i.e. the iterations dependent on a particular iteration I_0 are given, using equation (2):

$$
\begin{aligned}
&I_1 = I_{0,1} + D^t_{11} y_1 \\
&I_2 = I_{0,2} + D^t_{21} y_1 + D^t_{22} y_2 \\
&\ldots \\
&I_n = I_{0,n} + D^t_{n1} y_1 + \ldots + D^t_{nn} y_n
\end{aligned}
$$

Now, for an arbitrary iteration I, a *unique label* $I_0 = [I_{0,1}, I_{0,2}, I_{0,3}]$ with $I_{0,i} \in \{0, \ldots, D^t_{ii} - 1\}$, is calculated as follows:

$$
\begin{aligned}
I_{0,1} &= I_1 \mod D^t_{11} \\
y_1 &= (I_1 - I_{0,1})/D^t_{11} \\
I_{0,2} &= (I_2 - D^t_{21} y_1) \mod D^t 22 \\
y_2 &= (I_2 - I_{0,2} - D^t_{21} y_1)/D^t_{22} \\
&\ldots \\
I_{0,i} &= (I_i - \sum_{j=1}^{i-1} D^t_{ij}) \mod D^t_{ii} \\
y_i &= (I_i - I_{0,i} - \sum_{j=1}^{i-1} D^t_{ij})/D^t_{ii}
\end{aligned}
$$

Each label generates a different set of dependent iterations, using equation (2). The number of different labels, and therefore the parallelism $|L|$ of the loop L is

$$|L| = \prod_{i=1}^{n} D^t_{ii} = |\det(D^t)| = |\det(D)|,$$

since the unimodular transformation $D^t = DV$ maintains the absolute value of the determinant $\det(D)$.

4.4. Example

Consider the following loop:

```
for{$I_1 = 1..16$}
  for{$I_2 = 1..16$}
    for{$I_3 = 1..16$}
      $a(I_1, I_2, I_3) = A(I_1 - 1, I_2 + 2, I_3 - 4) + A(I_1 - 2, I_2 - 4, I_3 + 1)$
                        $-A(I_1, I_2 - 4, I_3 - 2);$
```

A unimodular transformation of the dependence matrix D

$$D = \begin{bmatrix} 1 & 2 & 0 \\ -2 & 4 & 4 \\ 4 & -1 & 2 \end{bmatrix}$$

yields $D^t = DV$ with

$$D^t = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 4 & 0 \\ 4 & 2 & 13 \end{bmatrix} \text{ and } V = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix}$$

Now the loop can be refactored as follows:

1. Construct parallel outer loops, generating a label I_0 for each parallel dependence set. The step size of the outer loops equals the diagonal elements of D^t.
2. Construct sequential inner loops, which generate the dependent iterations for label I_0.
3. Maintain the body of the original loop.

This gives:

```
$\textbf{for\_all}\{I_{0,1} = 1..1\}$
  $\textbf{for\_all}\{I_{0,2} = 1..4\}$
    $\textbf{for\_all}\{I_{0,3} = 1..13\}$
      $\textbf{for}\{I_1 = I_{0,1}..16, \textbf{step } 1\}$
        $y_1 = (I_1 - I_{0,1})/1$
        $I_{2,min} = 1 + (I_{0,2} - 1 - 2 * y_1) \mod 4$
        $\textbf{for}\{I_1 = I_{2,min}..16, \textbf{step } 4\}$
          $y_2 = (I_2 - I_{0,2} + 2 * y_1)/4$
          $I_{3,min} = 1 + (I_{0,3} - 1 + 4 * y_1 + 2 * y_2) \mod 13$
          $\textbf{for}\{I_3 = I_{3,min}..16, \textbf{step } 13\}$
            $A(I_1, I_2, I_3) = A(I_1 - 1, I_2 + 2, I_3 - 4) + A(I_1 - 2, I_2 - 4, I_3 + 1)$
                              $-A(I_1, I_2 - 4, I_3 - 2);$
```

There are 52 parallel iterations in the three outer loops. Each parallel iteration consists of three sequential loops.

5. Program transformations for locality

Since data movement in FPGAs is expensive, it is important to ensure that the calculations on the same data are conglomerated. The data locality is measured by the reuse distance:

Definition 2 *Reuse distance*

The reuse distance is defined as the number of distinct data elements accessed between the use and the reuse of the same data [15]. □

E.g. given an access pattern $A(1), A(3), A(5), A(1) \ldots$ the reuse distance of $A(1)$ is 3.

When the reuse distance is smaller than the available memory, no reload of the data is necessary. The unimodular loop transformation has a beneficial effect on the data locality, because related iterations are conglomerated in a number of serial loops. For the example in section 4.4, the reuse distance before and after the transformation is shown in figure 2.

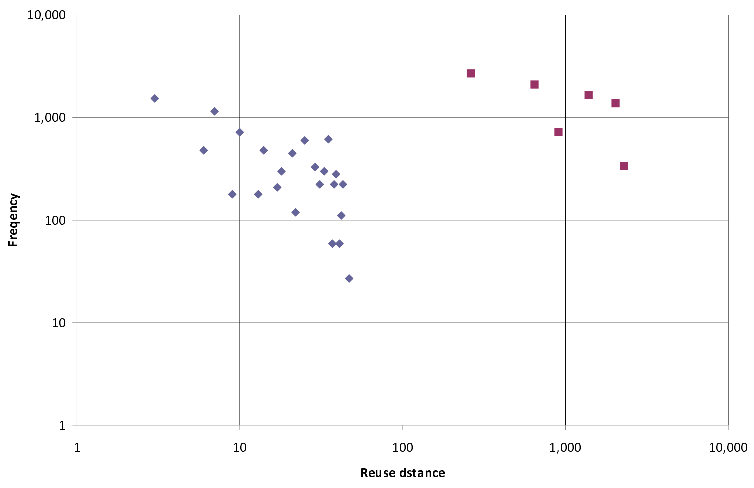

Figure 2. Reuse distance frequency in the example program before and after a unimodular transformation in a log-log scale. Squares are reuses of the original program, diamonds represent reuses after the unimodular transformation. The program transformation reduces the reuse distance almost 50 times.

The maximum reuse distance of the original and transformed program is respectively 2298 and 47. The unimodular transformation therefore decreased the reuse distance by a factor of almost 50.

The execution of the loops in parallel assumes that the participating FPGAs have a parallel access to a large shared memory. Moreover, the loop control and the modulo calculations shown in the example 4.4 create some overhead, which hampers the efficiency. In a hardware/software codesign configuration, the FPGAs are connected to the processor via a streams channel. A further unimodular transformation allows to get rid of the modulo operation and stream the data to the different cooperating FPGAs.

5.1. Data streaming using unimodular transformations

Consider a unimodular transformation by which the dependence matrix is brought into the diagonal form: $D^d = UDV$. The dependence equation (2) becomes

$$UI = UI_0 + UD^d y', \quad y' \in \mathbb{Z}^m, \tag{3}$$

considering that $UDy = UDVV^{-1}y = D^d y'$ with $y' = V^{-1}y$ an arbitrary integer vector in $\mathbb{Z}^m$. Let $Y = UI$ be an iteration in the transformed iteration space and denote $Y_0 = UI_0$ the label of the dependence set $Y_0 + D^d y'$. Since dependent iterations Y differ by a multiple of the diagonal elements, a unique label for the dependence set of Y is Y_0 such that

$$Y_{0,i} = Y_i \mod D^d_{ii}, i = 1 \ldots m. \tag{4}$$

It now becomes possible to assign an iteration I to a processing element as follows:

1. calculate the transformed index, $Y = UI$,
2. calculate the iteration label Y_0 according to equation (4),
3. assign the iteration to a processor according to the formula:

$$p = 1 + (Y_{0,1} + D^d_{11}(Y_{0,2} + D^d_{22}(Y_{0,3} + \ldots + D^d_{n-1,n-1}(Y_{0,n}) \ldots))) \mod p_{max} \tag{5}$$

where p_{max} is the actual number of FPGA processing elements available for the computation. Now in a hardware/software codesign environment, the processor executing the parallel loops, calculates the iteration label and processor number p and sends the data of the sequential loops to the streaming channel of FPGA processing element p, see figure 3.

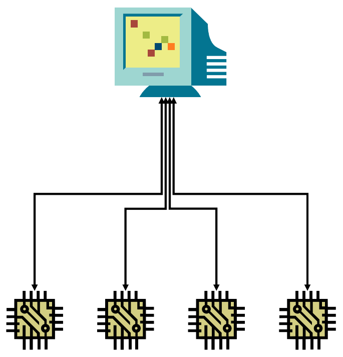

Figure 3. Streams operation: the parallel loops are assigned to FPGAs using formula 5.

The lexicographical ordering is maintained and each FPGA operates in a pipelined fashion on the incoming data stream.

6. Program transformations enhancing locality

When unimodular techniques described in the previous section are not applicable, the data locality in loops may still significantly be improved by a number of well known loop transformations.

6.1. Loop fusion

When the same data is used in adjacent loops, it is useful to merge the loops such that the data is traversed only once. E.g. the reuse distance of A[i] drops from M to 0 in the following loops:

$\textbf{for}\{i = 1..M\}$ $...A[i] + t...$ $\textbf{for}\{j = 1..M\}$ $...A[j] + v...$ $A[1]A[2]...A[M]A[1]...A[M]$	$\textbf{for}\{i = 1..M\}$ $...A[i] + t...$ $...A[i] + v...$ $A[1]A[1]A[2]...A[M]$

6.2. Loop interchange

When data is reused between two iterations of an outer loop, the reuses can be coalesced by interchanging the outer and the inner loop. In the following example, this decreases the reuse distance from N to 0.

$\textbf{for}\{i = 1..M\}$ $\textbf{for}\{j = 1..N\}$ $...A[j]...$ $A[1]A[2]...A[N]A[1]$	$\textbf{for}\{i = 1..N\}$ $\textbf{for}\{j = 1..M\}$ $...A[j]...$ $A[1]A[1]...A[1]A[2]$

6.3. Loop tiling

When data is reused both in the inner and the outer loop, loop interchange is not helpful, because it will improve the inner loop accesses, but worsen the outer loop accesses. Loop tiling improves the locality in both loops by an extra loop, minimizing the data access in the inner loops to a small fraction of the data set.

$\textbf{for}\{i = 1..M\}$ $\textbf{for}\{j = 1..N\}$ $...A[i] + B[j]...$ $A[1]B[1]A[1]B[2]...B[N]A[2]B[1]$	$\textbf{for}\{t = 1..N, \textbf{step } 10\}$ $\textbf{for}\{i = 1..M\}$ $\textbf{for}\{j = t.. \min(t + 10, N)\}$ $...A[i] + B[j]...$ $A[1]B[1]A[1]B[2]...B[11]A[2]B[12]..$

6.4. SLO: suggestions for data locality

In complex programs it is often not easy to find the parts with poor data locality, and even more so to find the right program transformation which improves the locality. E.g. current cache profilers are able to indicate the region where cache misses occur, but this is usually not the place where a program transformation improves the locality.

One of the reasons is that the cache profilers point at the endpoint of a long use/reuse chain, i.e. the cache miss. A new locality profiler, SLO (Suggestions for Locality Optimizations) [4,5] measures the reuse distance and registers a miss when the reuse distance exceeds the memory size. At the same time, SLO analyzes the code executed between use and reuse, and finds the loops where a transformation will have the highest impact on the reuse distance. Next, a suggestion is given to fuse, interchange or tile these loops.

Since SLO is based only on the reuse distance, the profiling is independent of the memory architecture and the suggestions can be applied equally well to optimize the use of FPGA memory. In [11] SLO has been successfully applied to find and refactor loops of a 2-dimensional inverse discrete wavelet transformation (2D-IDWT) used in an FPGA-based video decoder.

7. Optimizing data reuse with smart buffers

Many C to VHDL compilers, e.g. Streams-C, Impulse C or Handel-C apply the streams paradigm to use the FPGA as a hardware procedure. For example, in Impulse C, a procedure is first tested locally for correctness, and next the hardware version of the procedure is generated for implementation on the FPGA. Software and hardware stubs are created for streaming the data to and from the FPGA. Many multimedia and signal processing applications are suitable for streaming execution, in which a signal or an image is sent to the FPGA and results are fed back. In the FPGA, the data is stored in block RAM. In the previous section, it was shown that the data locality can be optimized by suitable transformations. An additional optimization for streaming applications is the use of a smart buffer [19]. A smart buffer is a number of named registers equal to the number of variables used in the computation kernel of the application. The idea is to organize the data in such a way that they are fetched only once from the block RAM, and used repeatedly in the smart buffer for all iterations operating on the fetched data. Consider for example the following N-tap FIR filter program:

```
$S = 0$
for$\{i = 0; i < N; i{+}{+}\}$
  $S = S + A[n-i] * \mathit{coeff}[i]$
$\mathit{Out}[n] = S$
```

With N=5, the array element $A[n]$ is used in five consecutive iterations to calculate $\mathit{Out}[n], \mathit{Out}[n+1], \ldots, \mathit{Out}[n+4]$. A C to VHDL compiler will fetch $A[n]$ five times from block RAM. In fact, five elements of the compute kernel are fetched from block RAM in each iteration. This creates a large overhead. By storing the elements in the registers, the access is much faster, and only one new element from block RAM is needed per iteration. This however requires the rewriting of the loop kernel computations five times, and replacing the array elements with named registers.

Since the block RAM access is the slowest operation, the use of a smart buffer achieves an almost fivefold speedup. In [14], the code produced by the Impulse C compiler has been adapted for use of a smart buffer, both for one dimensional and two dimensional applications.

In two dimensional applications such as image processing, the smart buffer consists of a window of registers representing the pixels used in the calculation. For example, in edge detection, the window consists of the pixels surrounding pixel A[i][j]. A two dimensional smart buffer creates an additional complication in that there are no results in each time step, because three data elements have to be fetched before a new result is available. This delay can be shortened by unrolling the inner loop in such a way that two or three adjacent rows are treated in the same unrolled iteration. In [14] it is shown that the application of the smart buffer technique on a FIR filter and an edge detection algorithm yield a speedup of respectively 4.14 and 4.99, using a small Spartan-3 FPGA board.

8. Conclusion

The use of FPGAs for high performance computing has drawn a lot of attention in the scientific world, at different levels of the algorithmic, application and hardware side. The computer architects have recognized the usefulness of FPGAs, as is evidenced by supercomputing companies like Cray and Silicon Graphics, which have successfully embedded FPGAs in their systems [9,34]. In the academic world, the paradigms by which FPGAs can be used have shown that there are plenty of opportunities [13]. Even so, we are still looking for one consolidating paradigm which tries to synthesize the findings of different approaches. Finally, the compilers need to be aware of specialized program transformations that deal with the characteristics of FPGAs, notably the slower processing clock, the limited amount of block RAM memory and the implications of the streaming communication, which seems to be the prevailing connection between processor and FPGA. In any case, FPGAs and reconfigurable computing will remain a very important building block for parallel and high-performance computing.

References

[1] Agility. *Handel-C Language Reference Manual.* Agility, 2008.

[2] Hari Angepat, Dam Sunwoo, and Derek Chiou. RAMP-White: An FPGA-based coherent shared memory parallel computer emulator. In *8th Annual Austin CAS Conference*, March 8 2007.

[3] Kristof Beyls and Erik H. D'Hollander. Cache remapping to improve the performance of tiled algorithms. In *Euro-Par*, pages 998–1007, 2000.

[4] Kristof Beyls and Erik H. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *Proceedings of the 2nd International Conference on High Performance Computing and Communications (HPCC)*, volume 4208, pages 220–229, Munchen, 9 2006. Springer.

[5] Kristof Beyls and Erik H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, 2009.

[6] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAS. In *PPoPP'07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 101–111, New York, NY, USA, 2007. ACM.

[7] Yi Chen, Nick Gamble, Mark Zettler, and Larry Maki. FPGA-based algorithm acceleration for S/W designers. Technical report, SBS Technologies, March 2004.

[8] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[9] Cray. XD1 Datasheet, 2004.

[10] Mache Creeger. Multicore CPUs for the masses. *Queue*, 3(7):64–ff, 2005.

[11] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized FPGA implementations. *Transactions on High Performance Embedded Architectures and Compilers I*, 4050:159–178, 7 2007.

[12] Erik H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, Jul 1992.

[13] Erik H. D'Hollander, Dirk Stroobandt, and Abdellah Touhafi. Parallel computing with FPGAs - concepts and applications. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15, pages 739–740, Aachen, 9 2007. IOS Press.

[14] F. Diet, E. H. D'Hollander, K. Beyls, and H. Devos. Embedding smart buffers for window operations in a stream-oriented C-to-VHDL compiler. In *4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, pages 142–147, Jan. 2008.

[15] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 245–257, New York, NY, USA, 2003. ACM.

[16] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69+, FEB 2008.

[17] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. In *FPGA'01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field Programmable Gate Arrays*, pages 134–140, New York, NY, USA, 2001. ACM.

[18] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[19] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *LCTES'04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers and Tools for Embedded Systems*, pages 249–256, New York, NY, USA, 2004. ACM.

[20] Sumit Gupta, Nikil D. Dutt, Rajesh K. Gupta, and Alexandru Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design*, pages 461–466, 2003.

[21] Brian Holland, Karthik Nagarajan, Chris Conger, Adam Jacobs, and Alan D. George. RAT: a methodology for predicting performance in application design migration to FPGAs. In *HPRCTA'07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, pages 1–10, New York, NY, USA, 2007. ACM.

[22] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: efficient realization of streaming applications on FPGAs. In *CASES'08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 41–50, New York, NY, USA, 2008. ACM.

[23] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In *DATE'04: Proceedings of the conference on Design, Automation and Test in Europe*, page 10202, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An automated exploration framework for fpga-based soft multiprocessor systems. In *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 273–278, New York, NY, USA, 2005. ACM.

[25] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC'02: Proceedings of the 39th conference on Design automation*, pages 628–633, New York, NY, USA, 2002. ACM.

[26] Seth Koehler, John Curreri, and Alan D. George. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parallel Computing*, 34(4-5):217–230, 2008.

[27] Peter Martin. A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, 2001.

[28] Markos Papadonikolakis, Vasilleios Pantazis, and Athanasios P. Kakarountas. Efficient high-performance ASIC implementation of JPEG-LS encoder. In *DATE'07: Proceedings of the conference on Design, Automation and Test in Europe*, pages 159–164, San Jose, CA, USA, 2007. EDA Consortium.

[29] Arun Patel, Christopher A. Madill, Manuel Saldaña, Chris Comis, Regis Pomes, and Paul Chow. A scalable FPGA-based multiprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 111–120, 2006.

[30] David Pellerin and Scott Thibault. *Practical FPGA programming in C*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2005.

[31] RAMP. Research Accelerator for Multiple Processors, http://ramp.eecs.berkeley.edu/.

[32] Manuel Saldaña and Paul Chow. TMD-MPI: An MPI implementation for multiple processors across multiple fpgas. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL 2006)*, pages 1–6, 2006.

[33] Manuel Saldaña, Daniel Nunes, Emanuel Ramalho, and Paul Chow. Configuration and programming of heterogeneous multiprocessors on a multi-FPGA system using TMD-MPI. In *3rd International Conference on ReConFigurable Computing and FPGAs 2006 (ReConFig'06)*, pages 1–10, 2006.

[34] SGI. SGI-RASC RC100 Blade. Technical report, Silicon Graphics Inc., 2008.

[35] Sunil Shukla, Neil W. Bergmann, and Jurgen Becker. QUKU: A FPGA based flexible coarse grain architecture design paradigm using process networks. *Parallel and Distributed Processing Symposium, International*, 0:192, 2007.

[36] Justin L. Tripp, Anders A. Hanson, Maya Gokhale, and Henning Mortveit. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 27, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical FPGA-based framework for novel CMP research. In *FPGA'07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field Programmable Gate Arrays*, pages 116–125, New York, NY, USA, 2007. ACM.

[38] Wei Zhang, Gao-Ming Du, Yi Xu, Ming-Lun Gao, Luo-Feng Geng, Bing Zhang, Zhao-Yu Jiang, Ning Hou, and Yi-Hua Tang. Design of a hierarchy-bus based MPSoC on FPGA. In T.-A. Tang, G.-P. Ru, and Y.-L. Jiang, editors, *8th International Conference on Solid-State and Integrated Circuit Technology*, page 3 pp., Piscataway, NJ, USA, 2006. IEEE.

[39] Ling Zhuo and Viktor K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 2, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Sotirios G. Ziavras, Alexandros V. Gerbessiotis, and Rohan Bafna. Coprocessor design to support MPI primitives in configurable multiprocessors. *Integration, the VLSI Journal*, 40(3):235–252, 2007.