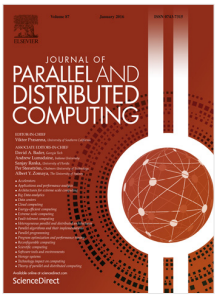
Accepted Manuscript



Efficiency analysis methodology of FPGAs based on lost frequencies, area and cycles

Jan Lemeire, Bruno da Silva, An Braeken, Jan G. Cornelis, Abdellah Touhafi

PII: S0743-7315(17)30324-6
DOI: https://doi.org/10.1016/j.jpdc.2017.11.012
Reference: YJPDC 3784

To appear in: *J. Parallel Distrib. Comput.*

Received date : 30 June 2016
Revised date : 20 October 2017
Accepted date : 15 November 2017

Please cite this article as: J. Lemeire, B. da Silva, A. Braeken, J.G. Cornelis, A. Touhafi, Efficiency analysis methodology of FPGAs based on lost frequencies, area and cycles, *J. Parallel Distrib. Comput.* (2017), https://doi.org/10.1016/j.jpdc.2017.11.012

***Highlights (for review)**

HIGHLIGHTS

- A methodology to study the impact of overheads on runtime performance is proposed.
- Three types of efficiency are introduced - area efficiency, frequency efficiency and cycle efficiency - and combined to define a global efficiency.
- Analytical formulas are presented to measure and to compute the respective efficiencies.

Efficiency Analysis Methodology of FPGAs based on Lost Frequencies, Area and Cycles.

Jan Lemeire[a,b,c,*], Bruno da Silva[a,b], An Braeken[a], Jan G. Cornelis[b,c], Abdellah Touhafi[a]

[a] *Dept. of Industrial Sciences (INDI), Vrije Universiteit Brussel (VUB), Pleinlaan 2, B-1050 Brussels, Belgium*
[b] *Dept. of Electronics and Informatics (ETRO), Vrije Universiteit Brussel (VUB), Pleinlaan 2, B-1050 Brussels, Belgium*
[c] *Data Science Dept., iMinds, Technologiepark 19, 9052 Zwijnaarde, Belgium*

Abstract

We propose a methodology to study and to quantify efficiency and the impact of overheads on runtime performance. Most work on High-Performance Computing (HPC) for FPGAs only studies runtime performance or cost, while we are interested in how far we are from peak performance and, more importantly, why. The efficiency of runtime performance is defined with respect to the ideal computational runtime in absence of inefficiencies. The analysis of the difference between actual and ideal runtime reveals the overheads and bottlenecks. A formal approach is proposed to decompose the efficiency into three components: frequency, area and cycles. After quantification of the efficiencies, a detailed analysis has to reveal the reasons for the lost frequencies, lost area and lost cycles. We propose a taxonomy of possible causes and practical methods to identify and quantify the overheads. The proposed methodology is applied on a number of use cases to illustrate the methodology. We show the interaction between the three components of efficiency and show how bottlenecks are revealed.

Keywords: FPGA, lost cycle analysis, performance efficiency, High-Performance Computing, High-Level Synthesis, Vivado HLS

*Corresponding author
Email address: jlemeire@etrovub.be (Jan Lemeire)

1. Introduction

Field-Programmable Gate Arrays (FPGAs) can be used as accelerators that provide a high computational performance combined with power efficiency. In this context, insight into all performance aspects is crucial. Traditionally, a performance analysis is, however, mostly limited to measuring or estimating performance (e.g. throughput), comparing performance with that of CPUs or GPUs, identifying the overheads and measuring the performance per watt. See for instance the HPC applications reported in [1]. Insight into how good the performance of a proposed design is, is often lacking. The goal of our endeavor is to put forward a formal methodology to analyze the **efficiency** of FPGA implementations. The methodology intends to explain and quantify why peak performance is not obtained, i.e. why the efficiency is lower than 100%. As we want to get the *maximal performance* out of an FPGA for a certain algorithm, we want know to *how far* we are from the peak performance, *why* the peak performance is not reached and whether improvement is possible. The factors that cause inefficiencies are the overheads. Insight into efficiency will help developers in improving FPGA implementations and comparing different implementations for a given algorithm.

The proposed methodology starts with defining the peak performance and the efficiency of an FPGA implementation. The global efficiency is then decomposed into 3 components (frequency, area and cycles) which can be used to quantify the efficiency losses and steer the identification of the different reasons for efficiency losses. We show that lost frequencies, area or cycles can be identified and analyzed separately although they are not independent when optimizing the performance: changing one component might affect another component.

The main scenario discussed in this paper is to achieve maximal computational performance. We focus on maximum computational throughput regardless of cost, power or other considerations. Nevertheless, our methodology can also be used for alternative scenarios such as striving to deliver the maximum possible performance within a space and/or power budget (Performance-per-Watt). Because we focus on computational performance, we limit ourselves to implementations that are compute bound rather than memory bound.

We start by discussing related work. Then, in Section 3 we propose our methodology. It is followed by discussion of its practical usage. Section 5 analyzes the overheads responsible for inefficiencies. Application on FPGA implementations is demonstrated in Section 6.

2. Related work

FPGAs are used for HPC in several domains, such as bioinformatics [2], linear algebra [3], stock market analysis [4] and image processing. It is shown that FPGAs can provide significant speedups in these domains [1]. The researchers report runtime performance and speedups when compared to CPUs. Efficiency and the corresponding bottlenecks (limiting factors) are often not addressed, as the performance analysis is in most cases only devoted to the exploration of the design space. Skalicky et al. [5] for instance provides an analytical model of the performance of several pipelined linear algebra designs which intends to identify design bottlenecks and improve performance. They focus on estimating execution time. They state that the 'performance of a computation depends on the implementation's efficient use of available resources', but are unable to give figures on the efficiency [6]. Another methodology, called Reconfigurable computer Amenability Test (RAT), is intended to model the critical set of algorithms and platform attributes in order to estimate the performance of a specific design, not a generic algorithm [7]. By measuring the resource consumption, RAT seeks to determine the scalability of an application design. In summary, RAT provides a methodology for rapidly analyzing an application's design compatibility with a specific FPGA platform. It estimates the throughput of an accelerator based on parameters like the interconnect's speed, amount of data to be transferred, the number of operations performed per data element and the clock frequency of the accelerator. Again, they do not estimate efficiency, although their work could easily be extended with the efficiency analysis proposed in this paper. On the other hand, they also consider bandwidth, while we concentrate on computational performance.

Interesting work in efficiency analysis is [8]. They designed a framework for runtime performance analysis of High-Level Language applications for FPGAs, including

an automated tool for performance analysis. It is able to determine the main bottlenecks and to recognize common performance problems such as potentially slow communication functions or idle hardware processes through instrumentation, measurement, analysis, and visualization. Indeed, instrumentation enables access to application data at runtime. Through tracing, occurrences of events are logged together with any associated data. In this way they are able to count the number of cycles spent waiting for a transfer to complete. It enables the tracking of overheads due to control hardware employed to maintain program order, pipelines, and communication channels. Our methodology provides the overarching metrics which allows to put the overheads in context and *quantify their impact on the global performance*.

The work of Koehler et al. [9, 10] on bottleneck detection is related to the search for the causes of the bottlenecks, which belongs to the second part of our methodology. They define a bottleneck as some portion of the application that reduces performance for the application as a whole. It is the most important work on enumerating and categorizing bottlenecks for FPGA applications. In our methodology we link bottlenecks to the decrease in efficiency and as such quantify the impact of overheads on the runtime performance. Koehler et al. [9, 10] estimate the possible speedup of removing bottlenecks by a kind of simulation which is based on the traced execution profile. The same approach is possible in our methodology, as will become clear when analyzing the lost cycles.

Also the work on design space exploration [18, 19, 23] lacks estimates for the efficiency. Sirowy and Forin [18] show the impact of optimization strategies on the global runtime but do not discuss the effect on area and cycle efficiencies. Zhong et al. [23] propose two definitions for area efficiencies. One as a sum over all component types of the ratio of used components versus number of components (Eq. 18). A second definition takes the maximum over all these ratios. We prove in Sec. 3.6.1 that the impact of these ratios on global efficiency is more complex than a sum or a maximum (Eq. 25). Our work adds a quantified efficiency analysis in a rigorous and 'complete' way.

As we will discuss, our analysis is influenced by concepts of a performance analysis in parallel computing. The work of Beltran et al. [11] defines performance and

efficiency metrics for FPGA-based multiprocessor systems. As base reference they consider uniprocessor performance. Efficiency is defined as speedup divided by the number of processors. This is one of the fundamental definitions in parallel computing [12]. We are different, we define efficiency as compared to what performance the FPGA's computing elements could deliver in ideal situations.

Our methodology is also based on the philosophy of Crovella and LeBlanc's Lost Cycle Analysis [13]. The idea is that a number of instructions have to be executed (called the useful work). The reference ideal situation is based on these useful instructions: a number of cycles (L_{opt}) are needed to execute these instructions. We assume no overhead. It relates to the sequential implementation of the algorithm which runs on a single processor. This ideal situation is compared with the actual number of cycles (L_{imp}) of the implementation under study. Then the overhead is $L_{imp} - L_{opt}/p$ with p the number of processors. This overhead is expressed in *lost cycles*: all p processors consume L_{imp} cycles of which L_{opt}/p are necessary to execute the useful instructions. The other cycles could have been used. Consequently, these cycles present the overhead of the implementation. A performance analysis should study these lost cycles and try to identify their causes.

3. Formal definitions of the efficiency analysis methodology

Before defining and decomposing performance efficiency, we start by defining a model of an FPGA and the implementation under study.

Table 1 summarizes all concepts of the methodology.

3.1. FPGA model

An FPGA consists of R_j components of type j. The maximum frequency at which the FPGA can run is f_{peak}. An FPGA requires a certain number of components to implement an operation. As there are several types of components and several configurations possible, each type of operation i is mapped onto vectors of resources. This is also referred to as the *cost vector* of an operation. For the sake of simplicity, we assume that only 1 type of component is needed to execute an operation instead of considering

	FPGA	
R_j	number of components of type j	p.5
f_{peak}	maximum frequency	p.5
r_j^i	number of components of type j needed to execute instruction type i	p.7
$\lambda^{i,j}$	latency of issuing instruction of type i on component type j	p.7
$\lambda_{op}^{i,j}$	operational latency ($= r_j^i \cdot \lambda^{i,j}$)	p.8
	Implementation	
N_{op}^i	useful operations of type i	p.7
U	percentage of FPGA area that is used	p.9
$R_{imp}^{i,j}$	number of components j that are used for the useful computations i	p.9
	Optimal performance	
T_{opt}	optimal runtime based on the total FPGA	p.8
T'_{opt}	optimal runtime when using U percentage of the FPGA area	p.9
R_{opt}^i	the number of components that each instruction uses for the optimal configuration	p.12
	Execution	
T_{run}	actual runtime of the implementation	p.9
f_{imp}	the actual frequency	p.9
L_{imp}	the number of cycles used to execute the implementation	p.9
	Efficiency	
$\mathcal{E}$	total FPGA efficiency	p.9
$\mathcal{E}'$	occupied FPGA efficiency	p.9
$\mathcal{E}_{freq}$	frequency efficiency	p.10
$\mathcal{E}_{area}^j$	area efficiency	p.10
$\mathcal{E}_{area}^{\prime j}$	used area efficiency	p.10
$\mathcal{E}_{cycle}^{i,j}$	cycle efficiency	p.10

Table 1: Overview of the different parameters of the efficiency analysis with page number of definition. Index i refers to instruction type and j to component type. The index is dropped when representing aggregated values or when only one type is considered.

complex cost vectors. We denote with r^i_j the number of components of type j that are needed to execute an operation of type i. The parameter r^i_j is set to infinity if the component in unable to execute the operation. In reality components of multiple types might be needed to execute an operation. Another simplification is the assumption that cost vectors are constant, while in practice they sometimes depend on factors such as the target frequency. Each r^i_j comes with a certain *issue latency* $\lambda^{i,j}$. The issue latency is defined as the number of cycles after which the execution of the next operation can be initiated. Note that the issue latency is different from the completion or end-to-end latency, which equals the total number of cycles required to terminate the complete execution.

As we will see, for analyzing the efficiency it is sufficient to concentrate on the components and operations that limit the performance. The others can be disregarded in the analysis. In many cases this will greatly simplify the performance analysis.

3.2. Useful work of an implementation

For the implementation under study, we first have to identify how many operations have to be executed for each operation type i (e.g. additions, multiplications, ...). This is denoted by N^i_{op}. We focus on the operations inherently present in the algorithm while discarding all overheads that are caused by the implementation. We call them the *useful operations*. The choice of which instructions are useful is somewhat arbitrary. For instance, considering loop control operations as useful is a matter of choice. Operations that are not counted as useful contribute to the overhead and decrease the overall efficiency. Sometimes it is interesting to know the loop control overhead. In parallel computing we regard the sequential algorithm as the reference algorithm of which all instructions are necessary and 'useful' [13]. Additional instructions introduced by a parallel implementation are considered overhead and not useful. Here we can also consider a sequential C-implementation of the algorithm as the reference. The useful operations are independent from the implementation. They are 'inevitable' and the minimal number of operations required to execute the algorithm.

To explain our methodology and its philosophy, we start by assuming that the implementation under study consists of 1 operation type (e.g. floating-point) which can be

executed by just 1 component type. Later we will extend it to heterogeneous operations and components.

3.3. Peak performance

The efficiency of an implementation will be defined with respect to the optimal performance that would be reached in the ideal case. The implementation under study has to execute N_{op} useful operations. If there are R components available and r components are needed to execute the operation under study with an issue latency of λ (as we focus on 1 type of operation and component, we drop the subscripts i and j), the ideal run time would be:

$$T_{opt} = \frac{N_{op} \cdot \lambda}{f_{peak} \cdot \lfloor R/r \rfloor} \approx \frac{N_{op} \cdot \lambda \cdot r}{f_{peak} \cdot R} = \frac{N_{op} \cdot \lambda_{op}}{f_{peak} \cdot R} \tag{1}$$

where $\lfloor R/r \rfloor$ gives the number of computational units that can be made. Since r is often small and R large, the approximation error will remain small. The approximation eases the further elaboration. λ_{op} is defined as the product of issue latency and the number of components required to execute the operation ($\lambda_{op} \triangleq \lambda \cdot r$). We call it the *operational latency*. For the analysis it is equivalent whether two components are needed to execute an operation with an issue latency of one cycle, or one component can do it with an issue latency of two cycles.

Despite the fact that attaining the theoretical peak performance is not realistic [21], it offers a reference or yardstick. In the ideal case, each component can start a useful operation each λ cycles. At first we do not consider any practical issues that would prevent a component from doing so. This is done in the next step, when we analyze the efficiency, i.e. the reasons why the ideal case is not possible. By comparing the actual execution with the ideal case this gives us the necessary insight into all issues to consider when using FPGAs for HPC. The values obtained for peak performance using this method should not be used to represent the achievable performance of a given device. We do not want to put forward this peak performance as a realisable peak performance, but as a yardstick to compare the performance of actual implementations and to be used for the discussion of inefficiencies. Another advantage of this definition

is that the proposed approach offers a **methodological** way of performing a qualitative and quantitative efficiency analysis.

3.4. Total FPGA Efficiency

The total FPGA efficiency of an implementation is defined as

$$\mathcal{E} \triangleq T_{opt}/T_{run} \tag{2}$$

where T_{run} is the actual runtime of the implementation. Alternatively, the efficiency can be calculated based on the performance (expressed in operations per second).

$$Performance_{peak} = f_{peak} \cdot R/\lambda_{op} \tag{3}$$

$$Performance_{imp} = N_{op}/T_{run} \tag{4}$$

$$\mathcal{E} = \frac{Performance_{imp}}{Performance_{peak}} = \frac{N_{op} \cdot \lambda_{op}}{T_{run} \cdot f_{peak} \cdot R}. \tag{5}$$

3.5. Occupied FPGA Efficiency

The performance efficiency targets an effective use of the whole FPGA. However, often one wants to optimize the used area. For instance, when for energy reasons, one wants to limit area consumption. With U we denote the fraction of the FPGA area that is used for the implementation. If we define $R' = U \cdot R$, then the ideal run time of Eq. 1 becomes

$$T'_{opt} = \frac{N_{op} \cdot \lambda_{op}}{f_{peak} \cdot R'} \, . \tag{6}$$

Occupied FPGA efficiency is then defined as

$$\mathcal{E}' \triangleq T'_{opt}/T_{run} \, . \tag{7}$$

It follows that $\mathcal{E} = U \cdot \mathcal{E}'$. Depending on the main optimization goal, one of both efficiencies should be considered. This consideration will be further discussed in Sec. 6.1.

3.6. Efficiency Decomposition

The efficiency is decomposed into three basic components (time, area and frequency). Define R_{imp} ($\leq R'$) as the number of components that are used for the

useful computations. Define f_{imp} as the actual frequency at which the FPGA executes the implementation, and L_{imp} the number of cycles used to execute. The relation with the runtime is given by:

$$T_{run} = L_{imp}/f_{imp} \tag{8}$$

It follows that the performance efficiency can be decomposed as

$$\mathcal{E} = \frac{N_{op} \cdot \lambda_{op}/f_{peak}/R}{L_{imp}/f_{imp}} \tag{9}$$

$$= \frac{N_{op} \cdot \lambda_{op}}{L_{imp} \cdot R_{imp}} \cdot \frac{R_{imp}}{R} \cdot \frac{f_{imp}}{f_{peak}} \tag{10}$$

$$= \mathcal{E}_{freq} \cdot \mathcal{E}_{area} \cdot \mathcal{E}_{cycle} \tag{11}$$

These components allow us to analyse overheads in detail. The frequency efficiency is known at design time:

$$\mathcal{E}_{freq} \triangleq f_{imp}/f_{peak} \, . \tag{12}$$

The area efficiency is defined as the number of FPGA components that participate in doing the useful computations, divided by the total number of FPGA components:

$$\mathcal{E}_{area} \triangleq R_{imp}/R \, . \tag{13}$$

If occupied FPGA efficiency is considered, R' should be considered in the definitions and area efficiency becomes:

$$\mathcal{E}'_{area} \triangleq R_{imp}/R' \, . \tag{14}$$

In the following subsections one has to substitute R with R' to retrieve the definitions for used are efficiency. The cycle efficiency is defined as

$$\mathcal{E}_{cycle} \triangleq \frac{N_{op} \cdot \lambda_{op}}{L_{imp}.R_{imp}} = \frac{\lambda_{op}}{\lambda_{imp}} \tag{15}$$

With λ_{imp} the *average operational latency* for the useful computations on the R_{imp} components:

$$\lambda_{imp} = \frac{L_{imp}.R_{imp}}{N_{op}} \tag{16}$$

Note that $N_{op} \cdot \lambda_{op}/R_{imp}$ equals L_{opt}, the number of cycles needed at peak performance, given only R_{imp} components are used. It follows that $\mathcal{E}_{cycle} = L_{opt}/L_{imp}$.

So, besides measuring the global efficiency by comparing the runtimes (Eq. 2), the 3 basic components of efficiency can be obtained separately with Eq. 12, 13 and 15. Multiplying the 3 efficiency components should give the same value for the global efficiency. Each loss in efficiency is due to some *overheads*. The second phase of the analysis consists of identifying the causes for losses in frequency, area and cycles. This is tackled in Sec. 5.

3.7. Extensions to multiple operations and component types

Our methodology supports multiple operations and multiple component types. Subscript i is used to denote operation type and j to denote component type. For the general heterogeneous case, the efficiency components are defined as:

$$\mathcal{E}_{freq} \triangleq f_{imp}/f_{peak} \tag{17}$$

$$\mathcal{E}^{j}_{area} \triangleq \sum_{i}^{I} R^{i,j}_{imp}/R^{j} \tag{18}$$

$$\mathcal{E}'^{j}_{area} \triangleq \sum_{i}^{I} R^{i,j}_{imp}/(U.R^{j}) \tag{19}$$

$$\mathcal{E}^{i,j}_{cycle} \triangleq \frac{N^{i,j}_{op} \cdot \lambda^{i,j}_{op}}{L_{imp} \cdot R^{i,j}_{imp}} \tag{20}$$

where $N^{i,j}_{op}$ is the number of instructions of type i executed on component j such that $N^{i}_{op} = \sum_j N^{i,j}_{op}$. The same computational unit can be constructed out of different components. Frequency efficiency is an overall efficiency, while area efficiency is per component type and cycle efficiency per component and operation type.

In the following we derive the peak performance and the equations that relate the efficiency components with the global efficiency (defined by Eq. 2).

3.7.1. One operation type and multiple component types

Given the component types, the ideal runtime becomes (subscript j denotes component type, we drop subscript i):

$$T_{opt} = N_{op}/(\sum_{j} R^{j}/\lambda^{j}_{op})/f_{peak} \tag{21}$$

The global efficiency turns out to be a kind of weighted average of area and cycle efficiency, where N_{op}^j/N_{op} is the weight.

$$Efficiency = \frac{T_{opt}}{T_{run}} \tag{22}$$

$$= \frac{N_{op}/(\sum_j R^j/\lambda_{op}^j)/f_{peak}}{L_{imp}/f_{imp}} \tag{23}$$

$$= \frac{1}{\sum_j \frac{R^j \cdot L_{imp}}{\lambda_{op}^j \cdot N_{op}}} \cdot \frac{f_{imp}}{f_{peak}} \tag{24}$$

$$= \frac{1}{\sum_j \frac{N_{op}^j}{N_{op}} \frac{1}{\mathcal{E}_{cycle}^j \cdot \mathcal{E}_{area}^j}} \cdot \mathcal{E}_{freq} \tag{25}$$

3.7.2. Multiple operation types and one component type

N_{op}^i denote the useful operations per instruction type i. Since we focus on only one component type, we take the cost vector with the minimal λ_{op}^i if minimal implementations would be possible. For optimal execution, the components are divided among the different instruction types according to $N_{op}^i \cdot \lambda_{op}^i$:

$$T_{opt} = \frac{\sum_i N_{op}^i \cdot \lambda_{op}^i}{R \cdot f_{peak}} \tag{26}$$

Let R_{opt}^i be the number of components that each instruction uses. To reach peak performance, we should divide the available components according to the following rate:

$$R_{opt}^i = \frac{N_{op}^i . \lambda_{op}^i}{\sum_i N_{op}^i . \lambda_{op}^i} \tag{27}$$

In the implementation under study, R_{imp}^i components are used to execute instructions of type i. The impact of the local efficiencies on the global efficiency is given by the following equations:

$$\mathcal{E} = \frac{(\sum_i^I N_{op}^i \cdot \lambda_{op}^i)/R/f_{peak}}{L_{imp}/f_{imp}} \tag{28}$$

$$= \frac{\sum_i N_{op}^i \cdot \lambda_{op}^i/R}{L_{imp}} \cdot \frac{f_{imp}}{f_{peak}} \tag{29}$$

$$= \sum_i (\mathcal{E}_{cycle}^i \frac{R_{imp}^i}{R}) \cdot \mathcal{E}_{freq} \tag{30}$$

$$= \sum_i (\mathcal{E}_{cycle}^i \frac{R_{imp}^i}{\sum_i^I R_{imp}^i}) \cdot \mathcal{E}_{area} \cdot \mathcal{E}_{freq} \tag{31}$$

The factors $\frac{R^i_{imp}}{\sum_i R^i_{imp}}$ are based on the actual 'distribution' of the components across the total number of components. This rate might be different than the optimal distribution given by Eq. 27. When more components are devoted to an operation than the ideal balance, the cycle efficiency will be lower than that of the other operations. Either there is more overhead, or the reason is an unequal distribution: the other operations are executed at maximal performance, but the components executing one of the operations cannot be kept busy because too many resources were reserved for it.

Eq. 29 shows that it makes sense to define an aggregate cycle efficiency as

$$\mathcal{E}_{cycle} = \sum_i \frac{N^i_{op} \cdot \lambda^i_{op}}{R \cdot L_{imp}} \tag{32}$$

The global efficiency remains a multiplication of the 3 efficiency components.

3.7.3. Multiple instruction types and multiple component types

Finally, we consider the most general case in which each component type can execute several instruction types. The optimal configuration (mapping of instructions on components) determines the peak performance. Some of the component types will be fully used. They determine the peak performance. Peak performance is reached with a configuration that minimizes the runtime:

$$T_{opt} = \arg\min_{conf}(\arg\max_{i,j}((N^{i,j}_{op}\lambda^{i,j}_{op}/R^{i,j})/f_{peak})) \tag{33}$$

With $conf$ iterating over all possible configurations resulting in different values for $N^{i,j}_{op}$ and $R^{i,j}$. Also, $\sum_j N^{i,j}_{op} = N^i_{op}$ and $\sum_i R^{i,j} = R^j$.

The component that bounds peak performance is denoted by $j = J$ and the operation by $i = I$, then efficiency becomes:

$$\mathcal{E} = \frac{(N^{I,J}_{op}\lambda^{I,J}_{op}/R^{I,J}_{opt})/f_{peak}}{L_{imp}/f_{imp}} \tag{34}$$

$$= \frac{N^{I,J}_{op} \cdot \lambda^{I,J}_{op}/R^{I,J}_{opt}}{L_{imp}} \cdot \frac{f_{imp}}{f_{peak}} \tag{35}$$

$$= \mathcal{E}^{I,J}_{cycle}\frac{R^{I,J}_{imp}}{R^{I,J}_{opt}} \cdot \mathcal{E}_{freq} \tag{36}$$

$$= \mathcal{E}^{I,J}_{cycle}\frac{R^{I,J}_{imp}}{\sum_I R^{I,J}_{imp}} \cdot \frac{R^J}{R^{I,J}_{opt}} \cdot \mathcal{E}^J_{area} \cdot \mathcal{E}_{freq} \tag{37}$$

The weight factors in the equation have an explicit meaning: the actual rate of used components times the inverse of the rate of the optimal configuration. The product is 1 if they are equal. An actual implementation will typically be based on a different configuration. If more components are used ($\frac{R^{I,J}_{imp}}{\sum_I R^{I,J}_{imp}} > \frac{R^{I,J}_{opt}}{R^J}$), then the cycle efficiency is increased. The total efficiency can, however, not be greater than one since such a non-optimal configuration will induce additional overheads of another type.

As discussed in Sec. 5.3, the values I and J that determine the optimal runtime do not mean that they are the bounding instruction and component type. Other instruction and component types might have to be considered as well.

4. Practical usage

Our efficiency analysis is not linked to any particular step in the design flow. However, the accuracy of the performance values is determined by at what stage is the data collected. Figure 1 shows the impact of optimizations on the throughput based on the stage of Xilinx's design flow [15]. Modifications at high-level have the highest impact on the final performance. It motivates a deeper analysis of the potential designs at an early stage before going through the whole design flow. Additionally, High-Level Synthesis (HLS) tools offer a fast design-space exploration, which facilitates such an analysis. The examples used to introduce our analysis consider traditional design tools such as Xilinx ISE and the Xilinx HLS tool called Vivado HLS. This HLS tool accepts C based languages like C, C++ or SystemC as input and converts each source code into a synthesizable RTL module.

Vivado HLS generates reports with estimations of the FPGA resource utilization, latency, and throughput of the resulting RTL module. The HLS design can go through the next stages (RTL, place and route) which will produce more accurate statistics on the design. Together with the Analysis viewer (discussed later) it provides enough information to obtain the parameters of our analysis:

- R_j, f_{peak}, $\lambda^{i,j}_{op}$: These hardware parameters are usually specified in technical reports.

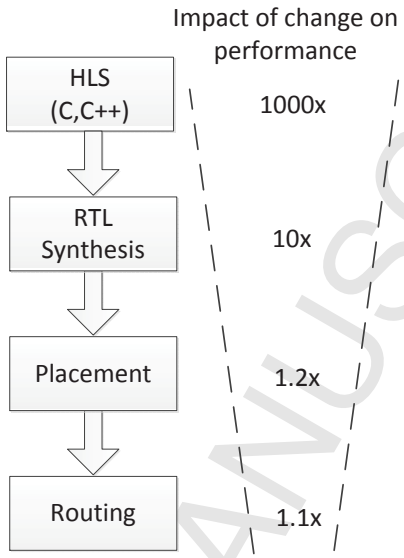


Figure 1: *Design flow and level of the impact of the design decisions on performance.*

- N_{op}^{i}: The number of useful operations is retrieved from the Vivado HLS report, based on the number of instances, and from the Analysis viewer, by measuring how many times a particular operation i is executed per iteration. The number of iterations is known (or estimated for variable loops). Both values define the total number of useful operations of type i. One can separate useful operations from overheads (such as control operations) in two ways. First, one can label the source code parts with the useful operations (such as the computations of a loop body) in Vivado HLS such that the computational units are labeled in the generated report. An alternative is to count useful operations in C code. The difference with the numbers from the HLS tool represents the number of overhead operations.

- R_{imp}^{i}: The resource consumption per operation is reported in the Analysis viewer.

- f_{imp}, L_{imp} and T_{run}: The cycles and frequency is reported by Vivado HLS, the runtime can be calculated from them.

From these parameters, the optimal runtime can be calculated (Eq. 1, Eq. 26, Eq. 21 or Eq. 33), the global efficiency (Eq. 2) as well as the three efficiency components (Eq. 17, Eq. 18, Eq. 32) and Eq. 20 for a detailed lost cycle efficiency.

5. Overhead analysis

Now that our methodology has clearly defined what deteriorates the global performance - namely lost frequencies, lost area and lost cycles - we want to dive deeper into the analysis. The question arises what causes these lost performances. A causal explanation has a *counterfactual interpretation*: if the cause could be eliminated, we expect the lost performance to disappear. In the sense that if 100 lost cycles are due to C, then by removing bottleneck C, the 100 lost cycles would disappear, cycle efficiency would increase accordingly and hence the global efficiency. The counterfactual increase of performance is called $speedup$ by Koehler et al. [9]. For instance, non-overlapped communication will block some computational components and hence induces lost cycles. In this case we have to identify the reason of the non-overlapped communication. This might be non-ideal communication (below the potential bandwidth) which causes long transfer periods.

In this section we establish a classification of possible causes for lost frequencies, lost area and lost cycles. We want to know the reasons for a frequency drop and the role of the used components that are not used for useful computations. We want to label each lost cycle with the actual reason of being idle. In this sense we will analyze each component in turn.

5.1. Lost frequencies

An FPGA is forced to function at a lower frequency because the design takes up a large percentage of the available logic resources, increasing the critical path which determines the clock rate.

5.2. Lost area

In the equations we considered the area (R_{imp}) that contributes to the execution of useful instructions. The rest of the components can either be used for other, non-useful

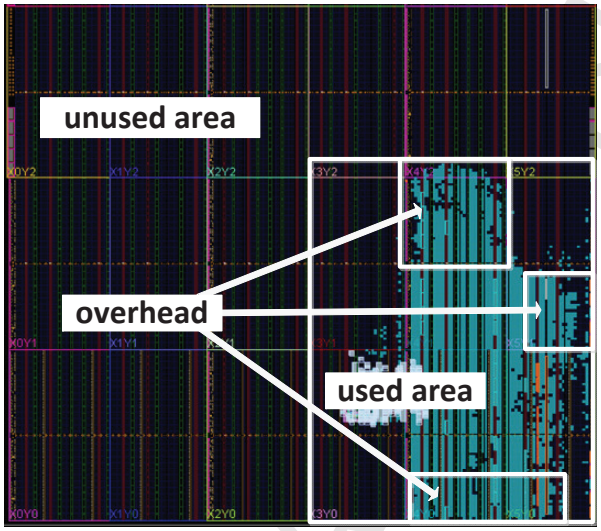


Figure 2: *The area of an FPGA is partly used for doing useful computations or overhead operations.*

operations or not used at all. As shown in Fig. 2, we call the former *overhead area* and the latter *unused area*.

Classification of lost area

1. Unused area
   (a) No more replication possible due to depletion of other area (which is necessary for more replication). Here, another resource type is (at least partially) bounding the performance. Therefore it is useful to keep track of the usage of all resources. This resource might be part of the cost vector or needed for support functionality such as memory or control logic.
   (b) Prevent frequency drop
2. Used area, instead of doing useful operations, components are being used for
   (a) routing
   (b) control
   (c) memory

When considering performance efficiency, area that is not used is considered 'lost'

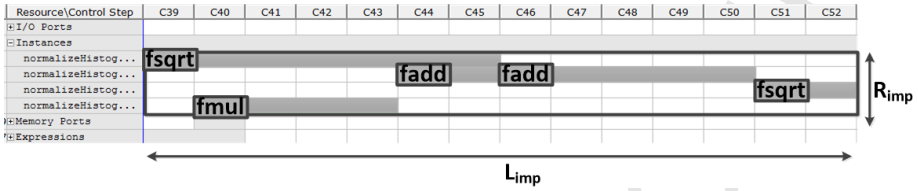


Figure 3: *Vivado HLS Analysis viewer reflecting the used and lost cycles.*

because it *could* have been used to increase the performance. When optimizing used area efficiency, the unused area is not considered overhead.

5.3. Lost cycles

The concept of lost cycles comes from the overhead analysis in parallel computing established by Crovella and LeBlanc [13]. On CPUs, the frequency is constant and one does not consider area. So the only inefficiencies that remain are cycles of the processors that are not used to execute useful instructions. The rationale of a lost cycle is that ideally the processor could issue an operation during that cycle, but it didn't because of 'this or that'. The 'this or that' is what we are interested in: the reasons for not reaching the peak performance. In parallel computing, when the parallel runtime takes x cycles, each processor can exploit these x cycles usefully. An efficiency analysis is devoted to analyze the portion of the x cycles that are not used usefully. For FPGAs we want to apply the same idea: R_{imp} components are devoted to the useful operations. The execution takes L_{imp} cycles. Thus, $R_{imp} \cdot L_{imp}$ is the total number of cycles at which an operation *could* be executed on the useful area. In reality, when executing the N_{op} operations, only $\lambda \cdot r_j^i \cdot N_{op}$ cycles were effectively used. The ratio of the former to the latter defines the cycle efficiency (expressed by Eq. 15). The lost cycle analysis is therefore performed on the execution profile, checking the cycle consumption of each R_{imp} component. Note that the other components should not be considered since they are already counted in the area efficiency.

This is illustrated by Fig. 3 showing the execution profile provided by the Vivado HLS analysis viewer. The viewer details the schedule of the design's execution. It shows how the resources are consumed, the I/O access and what is executed at every

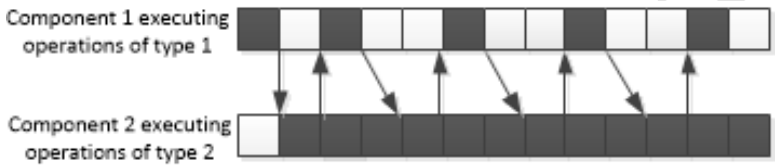


Figure 4: *Execution profile with 2 components having dependent operations. Component 2 is bounding the performance. Cycles in gray are useful, in white are idle.*

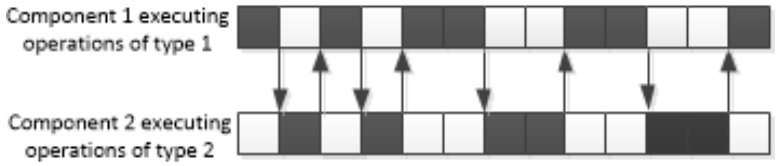


Figure 5: *Execution profile with 2 components having dependent operations. The idle periods can be filled with an overlapping iteration through pipelining. Cycles in gray are useful, in white are idle.*

clock cycle. The top bar reflects the control states of the execution, where each state corresponds to one clock cycle. Four different floating-point operations are used to compute a L2 normalization of HOG (which is discussed next). Each floating-point operation consumes dedicated resources; particularly specific IP-cores which are mostly implemented with DSPs. Every control step (at the top bar) represents one state of the Finite State Machine (FSM) and consumes one clock cycle. Cycles are colored gray when the components are executing the operation. However, only the issue latency (λ = 1 cycle) has to be considered as a useful cycle, since operations can be pipelined. This happens for instance at clock cycle 46, when a new addition is started and overlaps with the previous addition initiated at cycle 44. For the lost cycle analysis we have to consider the R_{imp} components that are performing the useful operations and have L_{imp} cycles at their disposal. From these $R_{imp} \times L_{imp}$ cycles only 5 cycles were devoted to issuing new operations. All the other cycles are lost cycles because they could have been used.

Classification of lost cycles:

1. **imperfect execution of operations**: happens with longer issue latencies than normal, e.g. with memory access.
2. **idle cycles**: component cannot proceed with the next operation due to a *dependency* with another component. A dependency is caused when the data to be processed is not ready yet. It can be static or dynamic (control, synchronization, ...). There are 2 possibilities: either the other component is bounding the performance or there is not enough parallelism to overlap operations.
   A. **bounded by another instruction** on a certain component, as shown in Fig. 4. The bounding instructions could be useful instructions as well as data movement or overhead operations.
      I. due to **non-optimal execution** (e.g. non-optimal data movement)
      II. due to an **imbalance**: another, better configuration is possible. Moving operations to other components will lead to fewer idle cycles and a better global performance. This was discussed in Subsection 3.7.2.
   B. **not enough parallelism**, as shown in Fig. 5. Since the involved components all exhibit idle periods, the idle periods could be filled with overlapping iterations through pipelining.
      I. **no independent iterations** in the algorithm: the algorithm is inherently sequential. Then the lost cycles are due to non-overlapping useful or overhead instructions (data movement, loop control, ...)
      II. insufficient concurrent execution possible due to **resource limitations**.
      III. **imperfect overlap**: although there is sufficient parallelism, the concurrent execution cannot prevent the idle cycles.

To analyze idle cycles and differentiate between cases of type A and type B, the lost cycle analysis must be performed per component and not per component type. This will be shown in the first example of the next section.

Note that the same analysis applies to the performance of GPUs: either GPU programs are compute bound, memory bound or latency bound [14]. The latter happens when there are not enough concurrent hardware threads to hide all latencies. This happens because of resource limitations.

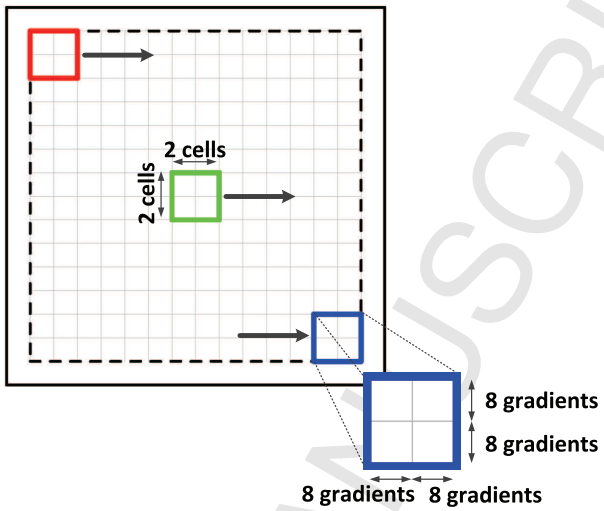


Figure 6: *Graphical description of HOG. Example of how the sliding windows can be processed in parallel since each individually processes a block of* 2×2 *cells.*

6. Applying the methodology in practice.

Our methodology is evaluated for several designs on a Xilinx Virtex6-LX240T. The tool chain is composed of the Xilinx HLS tool called Vivado HLS 2014.4 and the Xilinx ISE 14.7, in order to support our target FPGA. The resources available on this FPGA are 768 DSPs, 150k LUTs, 301k FFs and 832 Block RAMs. For our Virtex6 the maximum measured frequency that DSPs can operate is 484 Mhz.

Despite that our methodology is independent of the development tool, we considered the use of Vivado HLS for some of our examples for the sake of simplicity. This type of tools accelerate the design-space exploration thanks to the high-level representation of the algorithms and to the large set of available optimizations of such HLS tools. The optimizations considered are pipelining, partial loop unrolling and optimizing the I/O interface.

6.1. Efficiency analysis of a real-world HPC FPGA implementation

Our methodology is firstly applied to a real-world algorithm, a Histogram-Oriented Gradients (HOG) descriptor.

Algorithm description

The HOG algorithm is one of the most popular algorithms for object detection. The first step, the HOG descriptor is a sliding window algorithm that processes the gradient orientations and magnitudes obtained for each pixel from a pre-processed image (Figure 6). The output generated from the HOG descriptor is used by a classifier algorithm such as a Support-Vector Machine (SVM) to assign a matching score to the descriptor. The HOG algorithm is an application with high computational demands that needs to be executed on a hardware accelerator such as GPU [16] or FPGA [17] to achieve real-time object detection.

Although the whole object detection requires several steps, we only target the descriptor calculations since it is the performance limiting step. The HOG descriptor processes gradients obtained by calculating the orientation and the magnitude gradients for each pixel of the original image. The image is divided in square cells of 8 by 8 pixels, and a sliding window of 2 by 2 cells is slid over the image (Figure 6). The sliding window is moved one cell at a time and four histograms corresponding to the cells contained by this window are computed. Two loops, L0 and L1 are defined to traverse all the gradients of the 2 by 2 cells. The histograms are generated based on the orientation and magnitude values of each gradient. For each cell, the histogram bin is computed and the contribution in terms of the gradient orientation and magnitude is added to the histogram bin. Each combination of orientation and magnitude not only determines which histograms must be incremented but is also used to determine the value of the increment. Since there is an overlap of one cell for each sliding window, each cell contributes to the histogram of four sliding windows. These calculations demand several floating-point operations, which dominate the overall execution time. The HOG implementation analyzed in this paper is the floating-point version of the one presented in [17].

FPGA implementations

The C/C++ code describing HOG is compiled and synthesized using the Vivado HLS 2014.4 tool. The first implementation is one without optimizations. Secondly, the inner loop L1 is pipelined and, thirdly, the outer loop L0 is pipelined. Here we discuss the optimization of a single core. As shown in our previous work [17], this core can then

be replicated. For this reason we will try to optimize occupied FPGA efficiency $\mathcal{E}'$.

Efficiency analysis

The reports from Vivado HLS are used to create a table such as Table 2, where the parameters of the efficiency analysis, derived and calculated as explained in Sec. 4 are summarized. The upper table shows the detailed resource consumption and cycle efficiencies (which are calculated with Eq. 32). Since the DSPs are the limiting resources, we focus on this component. Two types of floating point instructions have to be considered: additions and multiplications. Used area (U) is calculated based on consumed DSPs. All consumed DSPs participate in the additions or multiplications, $\mathcal{E}'_a$ is therefore 100% for the three implementations. The lower table shows the aggregate results. The aggregated cycle efficiency $\mathcal{E}_c$ is based on Eq. 20 and is a weighted average of the detailed cycle efficiencies (see Sec. 3.7.2). Note that the occupied FPGA efficiency $\mathcal{E}'$ can be calculated in two ways: as T'_{opt}/T_{run} or as the multiplication of the three efficiency components $\mathcal{E}_f$, $\mathcal{E}'_a$ and $\mathcal{E}_c$.

Because of not overlapping iterations, the first version has a very low efficiency. The first optimization results in the highest efficiency. The second optimization is consuming more area; it leaves less area for replication. This is reflected in a lower $\mathcal{E}'$, while the run time is nearly the same.

Accuracy validation

Vivado HLS cannot guarantee that the reported numbers of the HLS design are correct before proceeding with the next steps towards the actual implementation (e.g. it does not know what the actual routing delays will be). To validate the results, we proceeded with the RTL design and place and route stage of the standalone core generated by Vivado HLS. The I/Os are assigned to the available pins of the FPGA and the target clock frequency is determined by the maximum reported frequency in Vivado HLS. The recalculated efficiency values are shown in Table 3. It shows that the only difference in terms of efficiency is a decrement of frequency by about 10% for the first and third implementation. The resource consumption is slightly different since the estimation of the area consumption is based on a component library. For this example, $\mathcal{E}'_a$ remains the same because the consumed DSP remains the same. Nevertheless, the estimations provided by Vivado HLS tool are good a good reference in terms of efficiency.

Impl	i	N_{op}^{i}	λ_{op}^{i}	DSP	LUT	FF	$L_{imp}[cc]$	$\mathcal{E}_{c}^{i}[\%]$
No optim	Add/Sub	3072	2	2	212	227		13.5
	Mul	1024	3	3	135	128		4.5
	Mul	512	3	3	135	128		2.3
	Mul	512	3	3	135	128		2.3
	Mul	512	3	3	135	128	22306	2.3
	Sub	256	2	2	212	227		1.1
	Sub	256	2	2	212	227		1.1
	Sub	256	2	2	212	227		1.1
	Total	**-**	**-**	**20**	**1388**	**1420**		
Pipeline L1	Add/Sub	1280	2	2	212	227		49
	Add/Sub	2560	2	2	212	227	2601	98
	Mul	2560	3	3	135	128		98
	Total	**-**	**-**	**7**	**1101**	**553**		
Pipeline L0	Add	16	2	2	212	227		0.6
	Add/Sub	1632	2	2	212	227		63.0
	Add/Sub	1456	2	2	212	227		56.1
	Add/Sub	384	2	2	212	227		14.8
	Add/Sub	272	2	2	212	227	2591	10.5
	Add/Sub	64	2	2	212	227		2.5
	Mul	1312	3	3	135	128		50.6
	Mul	1264	3	3	135	128		48.8
	Total	**-**	**-**	**18**	**1542**	**1618**		

Impl	$U[\%]$	$T'_{opt}[ms]$	$T_{run}[ms]$	$f_{imp}[MHz]$	$\mathcal{E}_{f}[\%]$	$\mathcal{E}'_{a}[\%]$	$\mathcal{E}_{c}[\%]$	$\mathcal{E}'[\%]$
No optim	2.60	15.8e-4	0.1887	118.2	24.42	100	3.44	**0.84**
Pipeline L1	0.91	4.53e-4	0.02494	104.28	21.54	100	84.36	**18.2**
Pipeline L0	2.34	1.76e-4	0.02485	104.28	21.54	100	32.97	**7.1**

Table 2: *The reports from Vivado HLS are used to analyze the efficiency of three HOG implementations: useful operations and resource consumption (top) and efficiency components (bottom). The peak frequency is 484MHz and the FPGA contains 768 DSPs.*

Impl	$U[\%]$	$T'_{opt}[ms]$	$T_{run}[ms]$	f_{imp}	$\mathcal{E}_f[\%]$	$\mathcal{E}'_a[\%]$	$\mathcal{E}_c[\%]$	$\mathcal{E}'[\%]$
No optim	2.60	1.58e-3	0.205	109.46	22.62	100	3.44	**0.77**
Pipeline L1	0.91	4.53e-3	0.02494	104.21	21.53	100	84.36	**18.2**
Pipeline L0	2.34	1.76e-3	0.0292	91.32	18.87	100	32.97	**6.02**

Table 3: *Efficiencies obtained after the placement and routing of the design solutions presented in Table 2.*

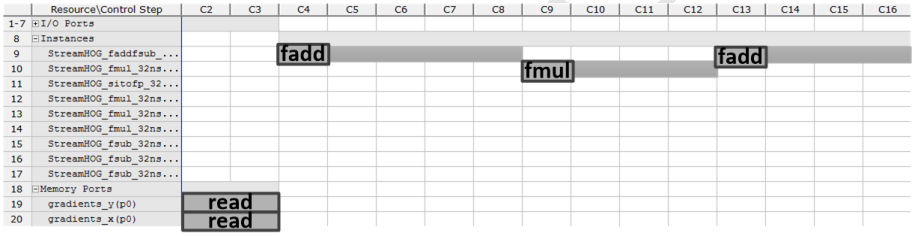


Figure 7: *Analysis viewer of the HOG descriptor without any optimization.*

Lost cycle analysis

The next step in our methodology is to identify the overheads, the reasons for the efficiency drops. The low area efficiency indicate that it might be possible to have more pipelining or more replication of the HOG blocks. The frequency drop is caused by the length of the critical path.

Regarding the cycle efficiency, we identify the lost cycles based on the classification in Sec. 5.3. The poor cycle efficiency of the non-optimized first implementation is due to lack of overlapping iterations (type 2.B.I). This also becomes apparent in Fig. 7, which depicts part of the Analysis viewer. In this part there are only 3 useful cycles.

Cycle efficiency is much better for the second implementation. The 98% of the second and third unit show almost complete utilization. Only the first unit is underutilized (49%). This is due to an imbalance in the distribution of the instructions among the components (type 2.A.II). A better distribution could in principle increase the performance. Imbalances also clearly appear for the third version, 'Pipeline L0'. 4 of the units have a utilization between 50% and 60%, while 4 units between 0.6% and 14.5%. As none of the units attain an efficiency close to 100%, more overlap seems to be pos-

Impl	$U[\%]$	$T'_{opt}[ms]$	$T_{run}[ms]$	f_{imp}	$\mathcal{E}_f[\%]$	$\mathcal{E}'_a[\%]$	$\mathcal{E}_c[\%]$	$\mathcal{E}'[\%]$
Pipeline L0	2.34	1.76e-3	0.0292	91.3	18.9	100	32.97	**6.02**
Limiting adders	2.60	1.58e-3	0.0324	91.3	18.9	100	26.0	**4.87**
Partitioning memory	3.39	1.21e-3	0.031	49.9	10.3	100	37.9	**3.90**

Table 4: *Efficiencies obtained after optimizing version 'Pipeline L0' (first row) by limiting the number of adders and partitioning the memory.*

sible (type 2.B.I). However, close inspection of the execution profile in the Analysis viewer reveals that the lost computational cycles are due to the memory components which are fully busy during those cycles (type 2.A).

Optimization

Finally, we try to optimize the design based on the aforementioned bottlenecks. Two optimizations were tried on the 'Pipeline L0' version. First, to remove the imbalance of the DSPs doing additions, we forced Vivado HLS to limit the used DSPs doing additions to 4 computational units. Secondly, we let Vivado HLS partition the memory to overcome the memory bottleneck. The results of both are reported in Table 4. We copied the results for the original 'Pipeline L0' version in the first row. However, no optimization results in a better performance. Limiting the adders resulted in more multipliers: 4 instead of the original 2. This decreased the cycle efficiency instead of increasing it. Memory partitioning increased the area consumption (7 adders and 4 multipliers) which were better used (higher cycle efficiency). But, conversely, the obtained frequency had to be almost halved.

6.2. Interaction of Efficiency Components

As clearly demonstrated by the HOG use case, the three efficiency components, $\mathcal{E}_c$, $\mathcal{E}_a$ and $\mathcal{E}_f$, are not independent. Changing or optimizing one component might affect another component. We discuss the 3 possible interactions.

6.2.1. Area and Frequency

The dependency between area and frequency efficiency is demonstrated by constructing a benchmark to attain the theoretical peak performance. A cascade of single-

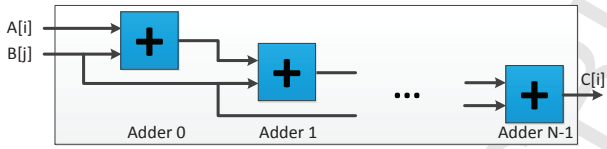


Figure 8: *Our benchmark consists of a cascade of a variable number of single precision floating-point adders.*

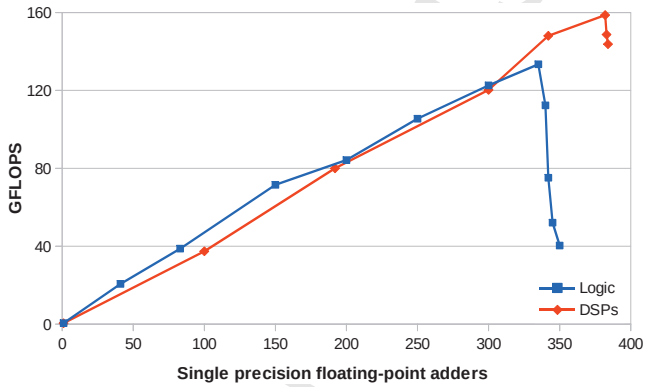


Figure 9: *Evolution of the floating-point performance by generating single precision adders with DSPs or with logic resources.Values obtained after placement and routing.*

precision floating-point adders is built as proposed in [21].

Fig. 8 shows the benchmark used to measure the attainable floating-point performance on a particular FPGA. These IPs are implemented as single precision floating-point adders configured to consume DSPs or logic resources (8). In order to maximize the floating-point performance, the best choice is to use the add/subtract operation. This floating-point operation can be implemented on an FPGA using DSPs and/or logic resources. The FPGA vendors' floating-point intellectual property (IP) user guide already offers an estimation of logic consumption and maximum frequency.

The performance is obtained after the placements and routing of the handmade VHDL benchmark, showing the real attainable performance and reflecting not only the routing congestion but also the impact on the frequency. Such effects can hardly be estimated at high-level, since the resource consumption are based on component's

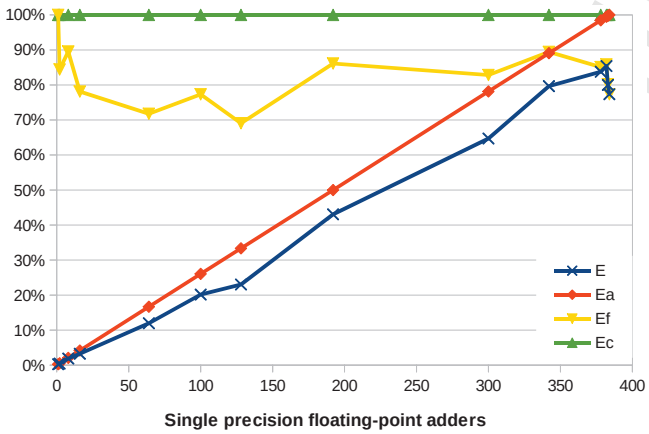


Figure 10: *Evolution of the efficiencies when increasing the number of single precision adders with DSPs.Values obtained after placement and routing.*

estimations. As a result, effects like routing congestion cannot be accurately estimated. Nevertheless, our model is designed to be applied at any level of design. Therefore, despite the results would be more accurate when using values after placement and route, the overall effort increases and a higher implementation time is needed.

The theoretical way to estimate the peak performance is by consuming all the available DSPs operating at their theoretical maximum frequency. However, by benchmarking the floating-point peak performance is not achieved when all DSPs are consumed. Fig. 9 shows how the floating-point performance increases linearly up to a certain point where the frequency starts to decrease due to routing congestion. The peak is achieved just before the maximum frequency starts to drop drastically when all DSPs are consumed.

Our Virtex6 lx240t offers 768 DSPs, or 384 single point additions, able to operate at 484 MHz. This represents a theoretical floating-point peak performance of 185.8 GFLOPS. Our measurements show that the peak is close to 158 GFLOPS when using only DSPs, while using only logic resources it approximates to 133 GFLOPS. Consequently, only 85% or 71% of the peak performance using DSPs or logic is respectively achieved respectively.

Fig. 10 shows how our benchmark exploits the available resources in order to reach

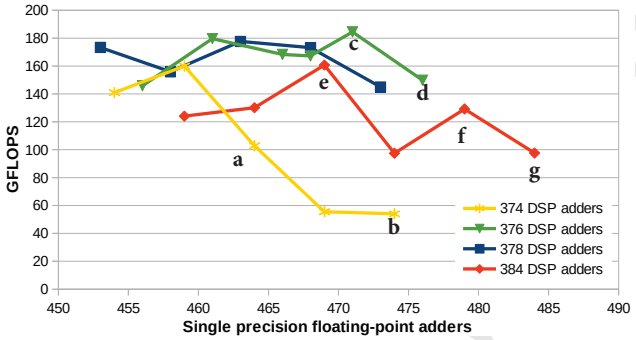


Figure 11: *Trade-off consuming DSPs and logic resources to obtain the single precision floating-point peak performance. By reducing the DSP consumption more logic is available for routing and additional adders. While up to 484 adders can be implemented, the peak performance is dominated by the maximum frequency. Values obtained after placement and routing.*

the highest efficiency when using DSPs for building the adders. The increment of the consumed area to execute useful operations leads to a higher area efficiency, but not necessarily to the highest performance. $\mathcal{E}_c$ remains constant at the highest value since there are no lost cycles due to the pipelining of the adders. Pipelining allows to start a new addition after the issue latency of the previous operation. Therefore,a floating-point operation is executed each λ cycles. The global efficiency is determined by $\mathcal{E}_a$ and $\mathcal{E}_f$. $\mathcal{E}_a$ increases linearly with the number of DSPs used to implement single precision floating-point adders. $\mathcal{E}_f$ starts to decrease just before $\mathcal{E}_a$ reaches the highest efficiency as shown in Fig. 10. The increment of additions due to a higher consumption of DSPs leads to a routing congestion, which enlarges the critical path and decreases the maximum frequency. Consequently, it is not possible to reach 100% of efficiency. Due to the dependency between area and frequency, the highest performance is only achievable assuming a trade-off between those parameters.

The strategy to reach the peak performance proposed in [21] is to consume all available DSPs by building as many adders as possible while dedicating the remaining logic resources to build more adders. An adder can be built with either 2 DSPs and about 212 LUTs, or with 385 LUTs. We neglect the amount of flipflops because they are not constraining the implementations. The equations when using a mix of component

	$N_{op}^{D\&L}$	N_{op}^{LUT}	f_{imp}	$Perf_{imp}$	$\mathcal{E}$	$\mathcal{E}_f$	$\mathcal{E}_a^{DSP}$	R_{imp}^{LUT}	$\mathcal{E}_a^{LUT}$	$\mathcal{E}_c$
a	374	90	222	103	37.7	45.8	97	118636	79	100
b	374	110	305	148	54.2	63.1	97	123751	82	100
c	376	95	392	184	67.7	80.9	98	119615	79	100
d	376	100	315	150	54.9	65.1	98	122691	81	100
e	384	85	343	161	58.9	70.8	100	118624	79	100
f	384	95	270	129	47.3	55.8	100	120182	80	100
g	384	100	202	98	35.8	41.7	100	122000	81	100

Table 5: *Efficiencies obtained by constructing $N_{op}^{D\&L}$ adders with DSPs and Logic, and N_{op}^{LUT} adders with only Logic (LUTs). Each row correspond to a point in the graph of Fig. 11. f_{imp} is in MHz and $Perf_{imp}$ is in GFLOPS. Peak performance is 272 GFLOPS.*

types were discussed in Sec. 3.7.1, although not when cost vectors have to be considered. With cost vectors, the calculation of the optimal runtime is in general less straight forward than Eq. 21, since one has to find the optimal usage of the different components first. In our case it is fairly simple, one first consumes all DSPs, 374 adders are built, and then uses the remaining LUTs for additional adders. This results in theoretically 185 additional adders and a peak performance of 272 GFLOPS. In practice it was only possible to synthesize 110 additional adders after consuming all DPSs. Figure 11 shows that the peak performance rounds 184.5 GFLOPS when using only 752 out of 768 DSPs, and combined with remaining logic. Next, we apply our methodology to acquire insight into the obtained efficiency. Table 5 summarizes the efficiency values for 7 points of Fig. 11. After filling the adder pipeline, each adder executes 1 operation each cycle. Cycle efficiency is then 100%. $Perf_{imp}$ is the product of frequency and the number of adders. Efficiency is calculated by comparing the attained performance with the theoretical peak performance of 272 GFLOPS. $\mathcal{E}_a^{DSP}$ and $\mathcal{E}_a^{LUT}$ represent the fraction of components used (Eq. 18), $R_{imp}^{DSP} = 2.N_{op}^{D\&L}$ and the given R_{imp}^{LUT}. Note that the total efficiency can also be calculated from its components with Eq. 25.

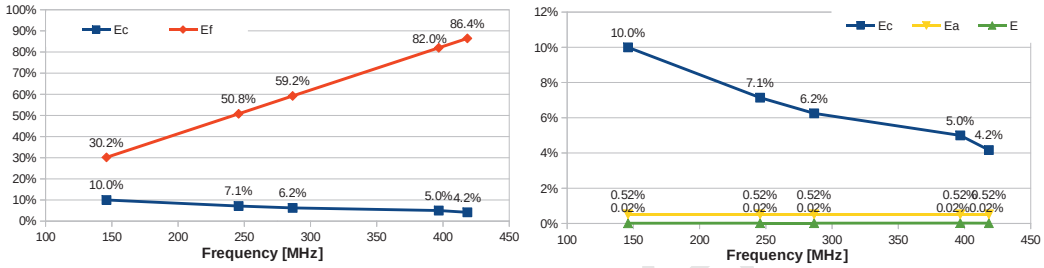


Figure 12: *Evolution of $\mathcal{E}_{cycle}$ when increasing $\mathcal{E}_{freq}$ for a matrix size of 32 × 32.*

6.2.2. Cycles and Frequency

Changing the frequency affects the total number of cycles and cycle efficiency. This is demonstrated with a fundamental linear-algebra operation, a matrix multiplication. We consider a matrix multiplication of a matrix A of m rows by k columns and a matrix B of k rows by n columns resulting in a matrix C of m rows by n columns. We assume all elements of matrix C to be equal to 0 before the computation begins. The implementation we consider, consists of three nested loops: L0, L1 and L2. L0 iterates over all rows of matrix C, while L1 iterates over all elements (columns) of the row corresponding to the current iteration of L0. An element at a given row and column is determined by computing the scalar product of the corresponding row of matrix A and the corresponding column of matrix B. For this purpose loop L2 adds the products of corresponding elements of said row and column to the target element of matrix C. Given that no intermediate values are used, each iteration of L2 accesses an element of matrices A, B and C. For the sake of simplicity, we will only consider square matrices.

Fig. 12 shows the evolution of $\mathcal{E}_c$ when increasing $\mathcal{E}_f$ with and without optimizations. Notice how the increment of $\mathcal{E}_f$ reduces $\mathcal{E}_c$. The increment on $\mathcal{E}_f$ is obtained by increasing the target frequency of the Vivado HLS design, which forces the tool to achieve a lower maximum clock period for every compilation. The frequency defines the clock period, which is the basic time unit. Therefore, a higher number of clock cycles is needed to execute the same code because of the increment of the frequency.

$\mathcal{E}_c$ decreases because of two main reasons:

- The impact of the lost cycles increases since their number increases due to a

shorter clock period.

- A higher frequency demands additional logic, mainly registers, to decrease the critical clock path. This leads to overhead, resulting in additional lost cycles.

6.2.3. Area and Cycles

As has been detailed before, $\mathcal{E}$ reflects the quality of an implementation, while the area, latency and frequency reflect where the design needs to be improved. The analysis of the peak performance for a floating-point matrix multiplication reveals that the limiting efficiency is $\mathcal{E}_a$. The original design does not fully exploit the available area.

Our efficiency analysis shows that pipelining loops is the most effective optimization for this algorithm, $\mathcal{E}_c$ achieves the highest efficiency. The overall efficiency, however, can be improved by increasing $\mathcal{E}_a$ or $\mathcal{E}_f$. The increment of the resource consumption increases $\mathcal{E}_a$ because these resources are dedicated to compute useful operations in parallel. On the other hand, $\mathcal{E}_f$ can increase up to a certain limit, as has been previously shown.

Optimizations at memory level need to be made to increase $\mathcal{E}_a$. In the previous design, memory was accessed serially, but with the proper optimization it is possible to reduce the memory accesses. By partitioning the memory blocks, the memory can be accessed in parallel. Partitioning the memory allows to operate in parallel, consuming more resources for useful operations and reducing the overall execution cycles. However, only the proper level of partitioning leads to the peak performance.

Table 6 shows how the overall efficiency increases when the memory is partitioned. The greater parallelism increases the resources dedicated to execute useful operations, improving $\mathcal{E}_a$. However, $\mathcal{E}_c$ slightly decreases. Since most of the lost cycles are constant when pipelining this algorithm [22], their relative impact increases when the execution time decreases. Thus, while increasing the parallelism, more area is used but the impact of control overhead increases. Actually, this is an example of *Amdahl's law*. Loop control overhead is constant, it does not decrease with more parallelism. Therefore, its impact on the overall efficiency increases with more parallelism.

Partition Level	$\mathcal{E}_f[\%]$	$\mathcal{E}_a[\%]$	$\mathcal{E}_c[\%]$	$\mathcal{E}[\%]$
1	59.20	0.52	99.28	**0.31**
2	59.20	1.04	98.47	**0.61**
4	59.20	2.08	96.89	**1.19**
8	59.20	4.17	93.97	**2.32**
16	59.20	8.33	88.62	**4.37**
32	59.20	16.67	79.56	**7.85**

Table 6: *Efficiency analysis of a* 32×32 *matrix multiplication when pipelining L1 and with different levels of memory partitioning.*

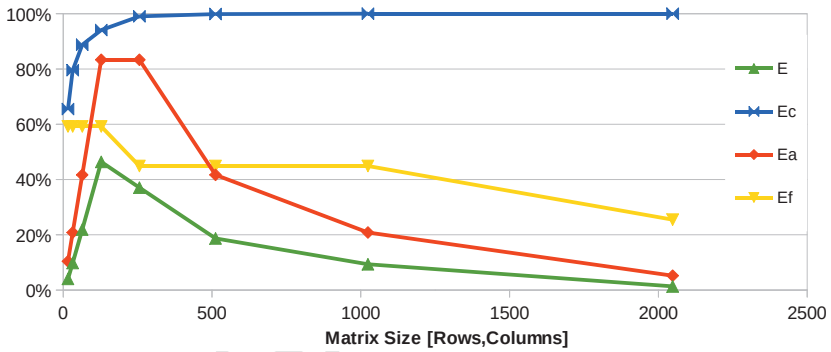


Figure 13: *Evolution of the efficiencies for each matrix size. The values have been obtained from the most efficient design with a particular matrix size. The optimizations applied are pipelining of the loop L1, several target frequencies and different levels of memory partition.*

6.3. Bottleneck identification

Our efficiency analysis helps in identifying bottlenecks. The results in the previous example only considers matrix multiplication of 32×32 matrices. Despite that the maximum parallelism has been reached, $\mathcal{E}_a$ only reaches 16.7%. Here we analyze what happens when processing large matrices. Our analysis targets the optimizations in the middle loop L1, because, as shown in the previous section, the highest performance is obtained through loop pipelining and memory partitioning. Consequently, these are the optimizations considered for this analysis.

Fig. 13 shows the evolution of the efficiencies in function of matrix size for the ma-

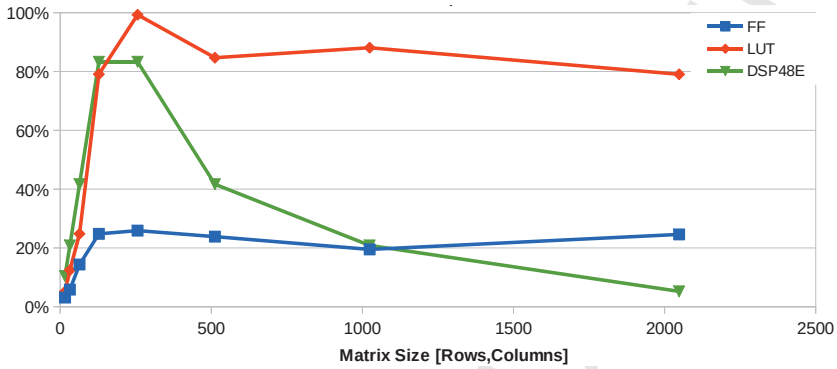


Figure 14: *Resource consumption $\mathcal{E}_u^j$ in function of matrix size. The optimizations applied are pipelining of loop L1, several target frequencies and different levels of memory partition.*

trix multiplication implementation. As depicted in Table 6, $\mathcal{E}_c$ slightly decreases due to the latency overhead introduced by the extra control logic required for the memory partitioning. When increasing the matrix size, the impact of this overhead is reduced and $\mathcal{E}_c$ converges to 100%. Regarding $\mathcal{E}_a$, the level of the memory partition is defined by the matrix size or by the area consumption. By default, 5 DSPs are consumed to execute the single floating-point additions and multiplications. The level of the memory partition increases the DSP consumption, and consequently $\mathcal{E}_a$. The maximum level of the memory partition is close to 150. This is obtained by considering the DSP consumption of each operation and the available DSPs on the target FPGA. Consequently, the highest $\mathcal{E}_a$ is expected to be achieved by completely partitioning a matrix of 150×150 floating-point elements.

Fig. 14 shows the limiting resource based on the matrix size. Thus, matrices smaller than 150×150 elements are limited by the number of available DSPs, while larger matrices are limited by LUTs. A high level of memory partition for large matrices is causing LUTs to become the limiting factor. Matrices larger than 150×150 elements consume all the available DSPs plus certain number of LUTs. For instance, matrices larger than 256×256 do not support the same level of memory partition than for matrices of 150×150 due to the LUTs consumption. This fact results in a reduction of $\mathcal{E}_a$ due to a lower level of memory partition. It decreases the number of parallel

operations and, therefore, reduces the number of consumed DSPs. $\mathcal{E}_f$ is also affected by this turning point. The achievable design frequency decreases with the increment of the matrix size due to the additional resource consumption. The resource overhead is slightly reduced when the target frequency is decreased, allowing higher partition levels of the memory until the LUT limit is reached again. Only the proper combination of both parameters leads to the highest efficiency, and, therefore, the peak performance for this algorithm. Notice that the decrease of $\mathcal{E}_f$, however, is not as abrupt as for $\mathcal{E}_a$.

7. Conclusions

When using FPGAs for HPC, one should compare the obtained performance with the peak performance and identify what blocks optimal execution. For this, we proposed a formal methodology to study the efficiency of an FPGA implementation. Our work provides a formal umbrella to complement existing work by extending the performance analysis with a quantification and decomposition of the efficiency. The value of the methodology is demonstrated with several studies. We were able to identify bottlenecks in different types of implementations. Next, we compared different alternatives in order to find the best compromise. It is also shown how the interrelations between area, frequency, and performance can be better understood thanks to our methodology.

Nevertheless, the utility of this methodology will depend on its ability to be automated and integrated into a tool.

References

[1] W. Vanderbauwhede and K. Benkrid, Editors, "High-Performance Computing Using FPGAs", Springer 2013.

[2] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics", Design & Test, IEEE 31.1: 19-30, 2014.

[3] S. Skalicky, S. Lopez, M. Lukowiak and C. Wood, "Mission control: A performance metric and analysis of control logic for pipelined architectures on FPGAs",

In ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on (pp. 1-6). IEEE. 2014.

[4] E. Gerlein, T. M. McGinnity, A. Belatreche, S. Coleman and Y. Li, "Multi-agent pre-trade analysis acceleration in FPGA", In Computational Intelligence for Financial Engineering & Economics (CIFEr), IEEE Conference on (pp. 262-269), 2014.

[5] S. Skalicky, S. Lopez, M. Lukowiak, J. Letendre and M. Ryan, "Performance Modeling of Pipelined Linear Algebra Architectures on FPGAs", 9th International Symposium, ARC 2013.

[6] S. Skalicky, S. Lopez and M. Lukowiak, "Performance modeling of pipelined linear algebra architectures on FPGAs", Computers & Electrical Engineering, Elsevier, Volume 40, Issue 4, 2014.

[7] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, A. D. George, "RAT: a methodology for predicting performance in application design migration to FPGAs", In Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07, (pp. 1-10). ACM, 2007.

[8] J. Curreri, S. Koehler, B. Holland and A. D. George, "Performance analysis with high-level languages for high-performance reconfigurable computing", In Field-Programmable Custom Computing Machines (FCCM), 2008.

[9] S. Koehler and A. D. George, "Performance Visualization and Exploration for Reconfigurable Computing Applications", ERSA, 2010.

[10] S. Koehler, G. Stitt, and A. D. George. Platform-aware bottleneck detection for reconfigurable computing applications. ACM Transactions on Reconfigurable Technology and Systems, Vol. 4, No. 3, 2011.

[11] M. Beltran, A. Guzman and F. Sevillano, "High level performance metrics for FPGA-based multiprocessor systems", Performance Evaluation, Vol. 67, No. 6, pp. 417-431, 2010.

[12] A. Grama, A. Gupta, G. Karypis and V. Kumar, "Introduction to Parallel Computing", Benjamin-Cummings, 2003.

[13] M. Crovella and T. J. LeBlanc, "Parallel performance using lost cycles analysis", Proceedings of the conference on Supercomputing, 1994.

[14] S. Hong and H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, ACM SIGARCH Computer Architecture News, 2009.

[15] "UltraFast Design Methodology Guide for the Vivado Design Suite", Available online: *http://www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pd*, Xilinx Inc., November 2015

[16] V. Prisacariu and I. Reid, "fastHOG-a real-time GPU implementation of HOG", Department of Engineering Science, Technical Report 2310/09, Oxford University, 2009.

[17] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, and J. Lemeire. "Comparing and combining GPU and FPGA accelerators in an image processing context", 23rd International Conference on Field programmable Logic and Applications, IEEE, 2013.

[18] S. Sirowy and A. Forin, "Where's the Beef? Why FPGAs Are So Fast", Technical Report MSR-TR-2008-130, Microsoft 2008.

[19] B. So, M. W. Hall, and Pedro C. Diniz. "A compiler approach to fast hardware design space exploration in FPGA-based systems." Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation, New York, NY, USA, 2002.

[20] "Vivado Design Suite, High-Level Synthesis User Guide", Available online: *http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug902-vivado-high-level-synthesis.pdf*, Xilinx Inc., 2014.

[21] "Technical White Paper: Understanding Peak Floating-Point Performance Claims", Available online: *http://www.altera.com/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf*, Altera Corporation, 2014

[22] B. da Silva, J. Lemeire, A. Braeken and A. Touhafi, "A Lost Cycles Analysis for Performance Prediction using High-Level Synthesis", International Symposium on Applied Reconfigurable Computing (pp. 334-342), Springer, 2016.

[23] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra and Smail Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis", 32th IEEE International Conference on Computer Design (ICCD), 2014.

***Author Biography & Photograph**

Bruno T. Da Silva Gomes is a Ph.D student at Vrije Universiteit Brussel (Belgium) under the supervision of Pr. Abdellah Touhafi. He obtained his degree in Telecommunications at the University of Vigo (Spain), specialized in Electronics and Telematics, and he completed his master thesis at iMEC (Belgium). For more than two years he has worked as researcher in the Signal department of the Univesity of Vigo (Spain) implementing different architectures for DvB receivers on an FPGA. He is currently finishing his Ph.D in the topic of High-Level Synthesis for High-Performance Computing applications.

Abdellah Touhafi obtained his bachelor's degree in Electronics, option: Computer systems at IHAM Antwerp. He has a Masters degree in Electronics from the VUB Brussels and a PhD in Applied Sciences: Scalable Run-Time Reconfigurable Computing Systems also at the VUB. He currently is a full time professor at the VUB and the leader of the Rapptor Lab: Reconfigurable Architectures, Parallel Processing and Telecommunications Oriented Research.

An Braeken obtained her MSc Degree in Mathematics from the University of Gent in 2002. In 2006, she received her PhD in engineering sciences from the KULeuven at the research group COSIC (Computer Security and Industrial Cryptography). In 2007, she became professor at Erasmushogeschool Brussel in the Industrial Sciences Department. Prior to joining the Erasmushogeschool Brussel, she worked for almost 2 years at a mangement consulting company BCG. Her current interests include cryptography, security protocols for sensor networks, secure and private localization techniques, and FPGA implementations.

Jan Lemeire is professor at the Department of Electronics and Informatics (ETRO) and the Department of Industrial Sciences (INDI) at the Vrije Universiteit Brussel (VUB). He is responsible for the bachelor courses on informatics and electronics, as well as for the master courses on computer architecture and parallel systems. He is the author of several publications in top journals, as well as book chapters, on parallel computing and probabilistic graphical models. Within the field of parallel computing, his areas of expertise are MPI, GPU computing, performance analysis and performance models. The results of this research can be found at www.gpuperformance.org. Jan Lemeire is co-founder of the Personal SuperComputing Competence Center which aims at lowering the thresholds for researchers and companies for exploiting GPUs. His interests in the field of probabilistic graphical models include causal analysis and learning algorithms in the context of modeling static and dynamic systems.

Jan G. Cornelis graduated as Master of Science in Engineering in 1996 at the University of Leuven. He worked for more than 10 years in the private sector. He returned to the academic world in 2011 after graduating as Master of Science in 2010 at the Vrije Universiteit Brussel. He currently works at the ETRO Department of this university, working on projects and performing research that both revolve around GPU programming and performance.