# A Fused Hybrid Floating-Point and Fixed-Point Dot-product for FPGAs *

Antonio Roldao Lopes and George A. Constantinides
{aroldao,g.constantinides}@ic.ac.uk

Electrical & Electronic Engineering,
Imperial College London,
London SW7 2BT,
England

**Abstract.** Dot-products are one of the essential and recurrent building blocks in scientific computing, and often take-up a large proportion of the scientific acceleration circuitry. The acceleration of dot-products is very well suited for Field Programmable Gate Arrays (FPGAs) since these devices can be configured to employ wide parallelism, deep pipelining and exploit highly efficient datapaths. In this paper we present a dot-product implementation which operates using a hybrid floating-point and fixed-point number system. This design receives floating-point inputs, and generates a floating-point output. Internally it makes use of a configurable word-length fixed-point number system. The internal representation can be tuned to match the desired accuracy. Results using a high-end Xilinx FPGA and an order 150 dot-product demonstrate that, for equivalent accuracy metrics, it is possible to utilize 3.8 times fewer resources, operate at 1.62 times faster clock frequency, and achieve a significant reduction in latency when compared to a direct floating-point core based dot-product. Combining these results and utilizing the spare resources to instantiate more units in parallel, it is possible to achieve an overall speed-up of at least 5 times.

## 1 Introduction

The dot-product computation, also known as vector scalar product or vector-by-vector multiplication, is a basic operation in linear algebra. This operation is also a building block in other fundamental algebraic operations such as matrix-by-vector, and matrix-by-matrix multiplications. All these operations are recurrent and central to many scientific algorithms, which range from solution finding for systems of linear equations [1] to the generation of complex biomedical images [2]. With their prolific employment, and often intensive computational requirements, it is important to explore methods that allow the acceleration of this basic operation.

* The authors would like to acknowledge the support of the EPSRC (Grant EP/C549481/1 and EP/E00024X/1) and the contributions of Dr. Eric Kerrigan.

In computer arithmetic there are a two widely used number systems, fixed-point and floating-point [3]. The former representation is commonly found in digital signal processors. This number system is fast, requires a modest amount of resources, and is well suited to modern commercial FPGA architectures. The latter number system is more flexible and has the advantage of supporting a wide range of values. Nonetheless, the digital logic required by floating-point is more complex, and this complexity comes at a significant performance and silicon cost.

Power consumption has become an important issue in accelerating scientific computing using typical high performance microprocessors, hence it has become increasingly important to explore alternative means of acceleration. This has positioned Field Programmable Gate Arrays (FPGAs) as a key component in the exploration of highly optimized reconfigurable architectures where speed-up can be provided by exploring wide-parallelism, deep-pipelining, fast and efficient datapaths, and through the usage of customized number systems.

In a typical microprocessor a floating-point unit can take operands from any source computation. But in a custom hardware accelerator, we have prior knowledge of the source and relationship between each of the operands. We can therefore take advantage of this knowledge and only perform normalizations, denormalizations, and alignments [4] when necessary. We extend this reasoning to allow for a customized hardware block where the interfaces abide by floating-point standards but internally the design is optimized for efficient FPGA implementation.

The main contributions of this paper are thus:

- a parameterizable hybrid floating-point and fixed-point dot-product design,
- a report on performance, latency, and resource utilization results for the proposed hybrid design (FX) and a standard floating-point (FP) core based design,
- an empirical error analysis for the absolute and relative solution error for both FX and FP designs,
- a comparison of the solution accuracy versus resource utilization trade-offs reported by FX and FP demonstrating that in most of the design space the FX implementation is superior.

After discussing the relevant background in Section 2, we present an overview of the hybrid dot-product method in Section 3. Section 4 details the proposed hardware design as well as a direct floating-point dot-product implementation. In the same section, we make a comparison between resource utilization, performance, and latency for the implementations. Section 5 describes a precision study based on a Monte-Carlo simulation. Section 6 presents the most significant findings of this work, showing that the proposed design offers a superior trade-off between accuracy and resource utilization. Section 7 concludes the paper.

## 2 Background

A dot-product is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = r, \tag{1}$$

where $a$ and $b$ are both vectors of order $n$, and $r$ is the resulting real-valued scalar quantity. This operation involves adding-up the pairwise products of the two vectors, and requires $2n-1$ scalar operations. Taking advantage of the associativity of addition over the real numbers, dot-products can be highly parallelized and deeply-pipelined, as shown in Fig. 1, and used in [1, 4, 5].

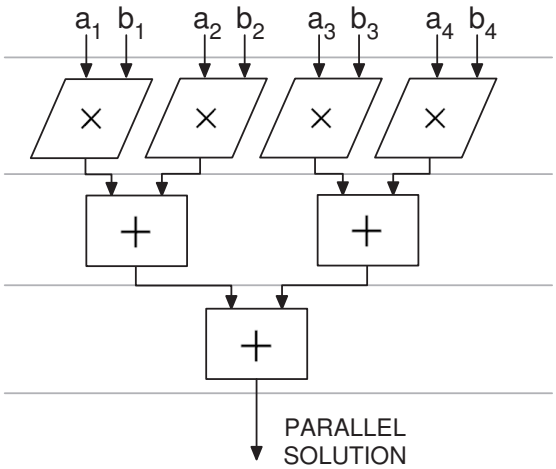



**Fig. 1.** A parallel dot-product of order 4.

A number $x$ is represented in floating-point using the notation in (2), where $S$ represents a sign bit, $M$ a mantissa, and $E$ an exponent.

$$x = (-1)^S \times 1.M \times 2^{E-bias} \tag{2}$$

In accordance with the IEEE 754 standard for binary FP arithmetic the mantissa is an unsigned number and a normalized floating-point number will always have a single one in the most-significant-bit (MSB) position. This bit is implicit and therefore the mantissa does not need to store it. The exponent is represented as a non-negative integer from which a constant *bias* is subtracted.

Floating-point multiplications require a number of stages besides the multiplication of the mantissas and the addition of exponents. These stages include the normalization, rounding, and re-normalization of the mantissa, if necessary [3]. In the case of floating-point additions two initial stages are required, one to detect which exponent is the highest, and another to align the mantissa of the lowest number to the same magnitude of the larger number. These stages are illustrated in Fig. 2.

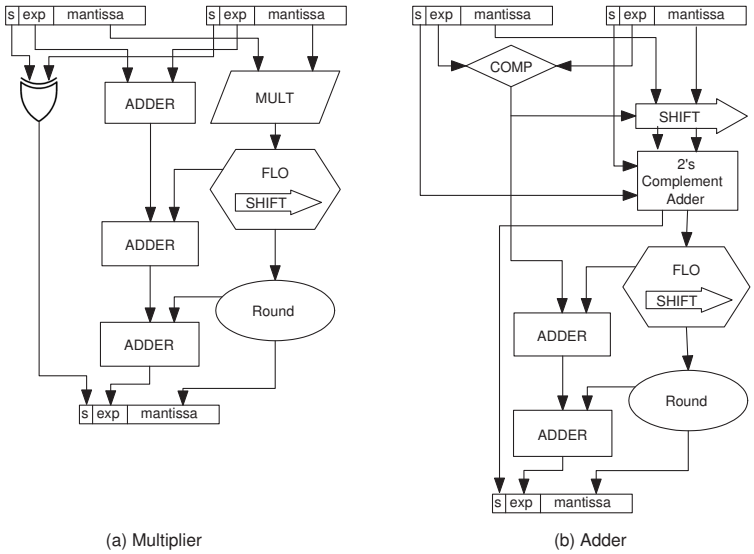



**Fig. 2.** Floating-point multiplier and adder diagrams showing the alignment stage in the adder and the normalization, rounding and re-normalization stages on both operations. FLO represents "finding leading one".

Floating-point arithmetic defined by the IEEE 754 standard [3] applies to atomic scalar operations, not to composite computations such as dot-product. As such, because of the non-associativity of floating-point addition, re-ordering of operands changes roundoff error. Thus we should see floating-point realizations of dot-products as producing a "family" of possible arithmetic accuracies rather than one single accuracy. Our scheme aims to be indistinguishable from this family under an appropriate measure of accuracy, while out-performing a straight-forward floating-point core based implementation.

In the fully parallelized and deeply pipeline dot-product circuit depicted in Fig. 1, where each floating-point operation output is connected to an adder input, there is a recurrent connection between a normalization, rounding and re-normalization circuit and a mantissa alignment circuitry. This recurrent logic

consumes significant resources and increases the latency of such operations. In this work we address this wastage and propose a dot-product design which fuses the entire dot-product datapath. The design inputs two floating-point number vectors and generates a floating-point output. Internally it makes use of a configurable word-length fixed-point number system, which eliminates the recurrent normalization and alignment stages present in straight-forward floating-point implementation. In this proposed implementation, the customizable word-length in the adder reduction-tree increases by 1-bit at each stage, automatically avoiding overflow. The corrected exponent is separately calculated and produced at the output of the dot-product circuit.

In previous related work, Underwood has performed a study which compared the performance of dot-products in FPGAs and CPUs [6]. In this 2004 paper it was predicted that FPGA-based floating-point operations would overtake CPUs by at least an order of magnitude by 2009. In this paper, we demonstrate that such performance is indeed possible by configuring FPGA resources into highly optimized parallel and pipelined datapaths.

In [7], a number of reduction circuits, which are an inherent part of dot-product operation, are studied. For these circuits, the authors have set three fundamental requirements. These include: no stalling on multiple input sets of arbitrary size, the number of adders must be fixed, and buffers are only allowed to grow up to a reasonable size. The proposed designs successfully address the target requirements, and solves the problem arising in summation-reduction using high latency pipelined floating-point cores. In contrast our design aims at reducing overall latency and minimize area consumption by deviating from a floating-point core-based design.

In [8], the authors propose a floating-point fused dot-product unit. This unit is limited to order 2, nonetheless significant improvements are reported. These improvements include a reduction of latency by 80% when compared to a fully parallel approach and 50% when compared to a serial approach. In terms of resource utilization, the fused approach saves 42% when compared to a fully parallel implementation.

The FloPoCo project [9] is an open-source initiative which provides a framework to automatically generate floating-point operators. The precision of these operators can be customized to exploit and maximize the flexibility provided by FPGA. The generated VHDL code is synthesizable and portable to any mainstream FPGA architecture. However there is no primitive to generate a highly efficient dot-product datapath.

In [4], Langhammer proposes a tool that automatically fuses a floating-point datapath. In this paper it is reported that efficiencies gained by fusing the entire datapaths result in a typical 50% improvement in terms of logic, latency, and

power reduction. The focus is on an automated flow taking best advantage of the underlying architecture for non-standard floating-point, rather than on the application-specific hybrid representation we discuss.

In this work we present a fused-hybrid dot-product design that can provide up to a 5 times speed-up in throughput as well as a reduction in latency by 5 times.

## 3 Hybrid dot-product design

The proposed design considers the dot-product operation in its entirety. This allows for the elimination of redundant circuitry that is present in straightforward floating-point core based implementations. This redundant circuitry, as illustrated in the Section 2, consumes significant resources, accounts for the comparatively high latency, and increases the complexity of placement and routing which in turn lowers the overall operating frequency. Our proposed design is fully parallelized and deeply pipelined, and can receive a new set of input vectors at each clock cycle.

This proposed design, as depicted in Fig. 3, comprises of a number of interconnected blocks. At the inputs it receives two sets of $n$ single precision floating-point numbers. Each of these floating-point numbers is partitioned into its sign-bit, exponent-bits, and mantissa-bits. At this initial partitioning, the leading hidden bit which can be 1 or 0, depending on the exponent, is concatenated on to the mantissa. Subsequently a pair-wise multiplication is performed between corresponding mantissas of each vector, exponents are added together, and respective sign-bits XORed. At the output of the multiplication stage, the mantissas can be truncated or expanded into a desirable internal word-length. This allows us to trade-off resource utilization with solution accuracy, as described in Section 5. In parallel, all the exponents are compared in a tree with $\lceil \log_2 n \rceil$ stages. The highest exponent becomes the reference and all the mantissas are aligned to this reference. After this alignment, the mantissas are converted into 2's complement number representation. In the next stage, the sum-reduction is performed on the mantissas, and at each level of this reduction-tree the word-length is increased by 1-bit, to prevent overflow. After this summation, the fixed-point number is reconverted to sign and magnitude, aligned so as to drop the leading one, and the exponent corrected. From these values, an IEEE 754 compliant floating-point number is generated and output.

## 4 Implementation

We have implemented a number of designs; the first set of designs is a reference core based floating-point implementation, which we will refer to as "FP$n$" for an order-$n$ dot-product; the other set of designs are based on our proposed scheme,

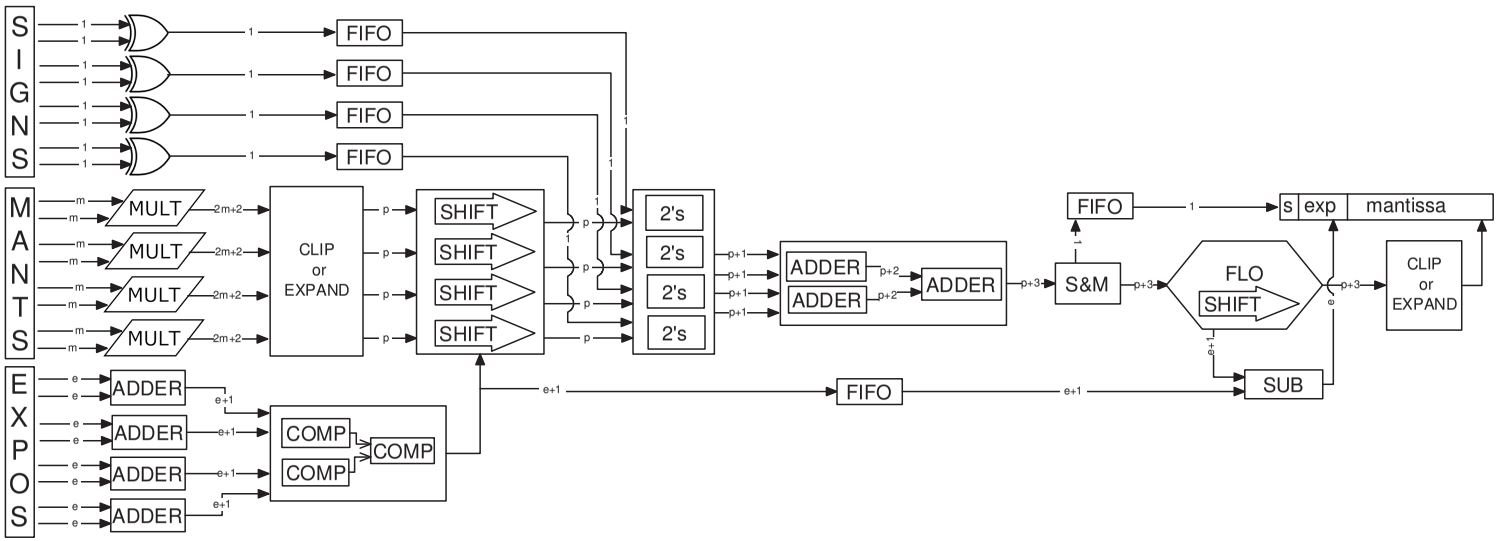



**Fig. 3.** The proposed hybrid floating-point and fixed-point dot-product circuit for $n = 4$. In this circuit, data flows from left to right, and each signal is annotated with its word-length. Floating-point numbers are multiplied together to produce $n(2m + 2)$-bit mantissas. After this multiplication, these mantissas are truncated or expanded to the desired internal precision $p$. Subsequently, all mantissas are aligned to the highest exponent. After the alignment, all mantissas are converted to the two's complement representation and added together. With the resulting sum, the first leading one (FLO) is searched and its position is used to produce an aligned mantissa and the final corrected exponent. The aligned mantissa may be truncated or expanded before being output. The FLO block generates an output in a single clock cycle, and this unit is the one that relatively exhausts the highest number of resources.

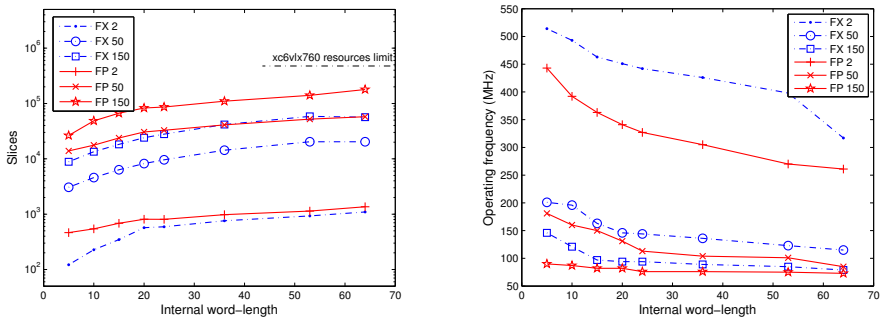
as described in the previous section, we will refer to these as "FX$n$". Both sets were targeted, placed and routed onto a Xilinx Virtex6LX760-2 FPGA, and the toolchain used was Xilinx ISE version 11.3. This toolchain was set to optimize for speed with maximum effort. These parameterizable designs can also be trivially modified to operate on Altera FPGAs.

For all designs, the input and output format uses IEEE 754 single precision floating-point. However, in both sets of designs, it is possible to trade-off area against accuracy; in the FP designs this can be done by using a greater precision internally to the dot-product, *e.g.* double precision. In our design, this trade-off is achieved by varying the fixed-point word-length, $p$.

### 4.1 Resource utilization

To measure resource utilization as a function of the internal word-length, $p$, we have selected Look-Up-Tables (LUTs) slices as the resource of interest because these are the limiting factor on the target FPGA. In Fig. 4(a) there are three lines describing slice utilization for each the two sets of implementations. Resource utilization growth is linear with input vector order. This growth becomes quadratic when varying the internal word-length. Both sets of designs utilize the same number of DSP48 blocks.

It is important to note that a fixed-point implementation with $p$ internal bits will, in general, have a different solution accuracy when compared to a floating-point implementation of $p$ internal bits. This issue is addressed in Section 5, where we show our approach provides a superior trade-off for comparable accuracy.




(a) Slice utilization (b) Operating frequency

**Fig. 4.** Resource utilization and performance for the Virtex6 LX760.

### 4.2 Performance and Latency

The operating frequency as a function of internal word-length, $p$, has been depicted in Fig. 4(b) for both sets of implementations. Since these circuits are fully pipelined and parallelizable their performance in terms of operations per second is given by $(2n-1) \times$ frequency.

The latencies for the FX and the FP sets of circuits, as a function of input vectors orders $n$, are described in (3) and (4) respectively. In (4), $L_m$ and $L_s$ represent the latencies of the individual floating-point cores for multiplication and addition, respectively. These latencies vary with word-length, and in the case of 24-bit mantissas (IEEE 754 single precision), $L_m$ is 8 and $L_s$ is 12, when utilizing the floating-point modules from the Xilinx CoreGen library. Thus the latency of the proposed scheme is also superior when $n \geq 5$.

$$\text{FX Latency}(n) = 2\lceil \log_2 n \rceil + 29 \tag{3}$$

$$\text{FP Latency}(n) = L_m + L_s \lceil \log_2 n \rceil \tag{4}$$

## 5 Precision Study

In any finite number representation numbers have a limited accuracy. This accuracy can be improved by increasing the word-length, which in bit-parallel hardware translates to more resources being allocated. In this section, we create two random test-benches for vectors of order 2 and 150. Each of these test-benches comprises of 10000 randomly generated pairs of vectors. Each of these vectors comprises of single precision floating-point numbers with exponents uniformly distributed between -50 and 50, and mantissas uniformly distributed over their natural range.

We have also implemented a software dot-product which emulates the hardware. This software utilizes the MPFR library which is a C library for multiple-precision floating-point computations with correct rounding [10]. Utilizing this software and emulating the dot-product with 128-bits of internal precision, we have generated our reference values. From these values we were able to calculate the error as function of word-length and of resource utilization, as described in the following section.

### 5.1 Word-length and Error

Running the 10000 randomly generated vectors and varying the word-length we have produced the two plots as depicted in Fig. 5; the first plot shows circuits with input vectors of order 2; the other plot depict circuits with input vectors of order 150. Note that since the error is data dependent, we use the measures of accuracy given in (5) where $S$ denotes the input test set; $test_i$ is the result

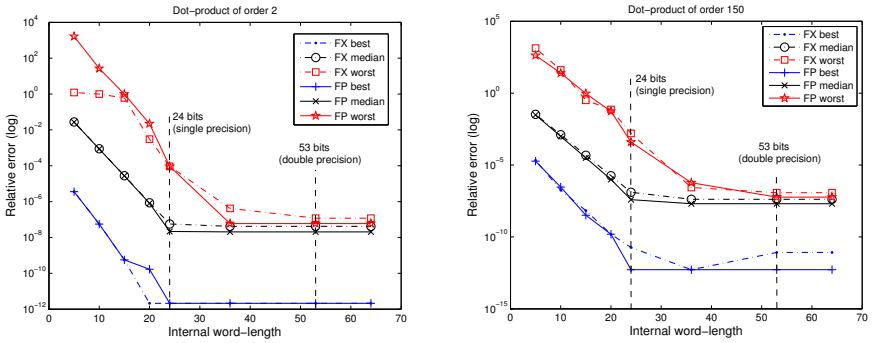



**Fig. 5.** Error as a function of word-length for dot-product order 2 and 150.

produced by the unit under test; and $ref_i$ is the result produced by our high accuracy MPFR implementation.

$$\text{error} \ = min/max/median_{i \in S} \ |\frac{ref_i - test_i}{ref_i}| \tag{5}$$

It is possible to observe that the floating-point error lines become flat for word-lengths significantly greater than 24-bits. This reflects the precision of the input values which are all single precision.

## 6 Error and Resource Utilization

In the previous section we demonstrated how the internal word-length affected the solution accuracy. In this section we translate word-length into resource utilization and show how it is possible to trade-off resources with solution accuracy. It is further demonstrated that by using our proposed design, it is possible to achieve the same median accuracy, in terms of relative and absolute error, while saving significant resources when compared to the FP design. This is demonstrated through the four plots in Fig. 6. For example, using our proposed implementation, with 24 internal bits, it is possible to achieve a better solution accuracy than using the straight-forward floating-point implementation with 20 internal bits, while consuming almost 4 times less resources. The solution accuracy of our design levels-off at a slightly lower accuracy then the straight-forward floating-point scheme. Therefore, beyond this level of accuracy, the only option is to use the FP implementation. Apart from this extreme case, our proposed design provides a significantly better trade-off between solution accuracy and resource utilization for all of its accuracy range. These plots were generated using median error values, however it is also important to note that both the minimum and maximum error values provide similar trade-offs.

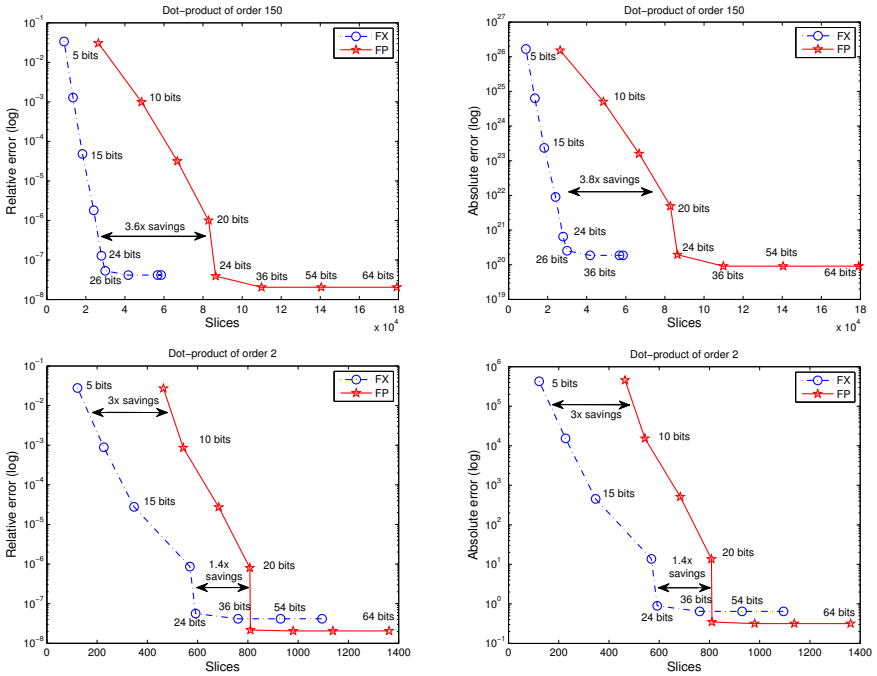



**Fig. 6.** Error as a function of slices utilization for dot-product of order 2 and 150.

## 7 Conclusions

This paper proposes new FPGA-based dot-product design which takes advantage of deep-pipelining, wide-parallelism, highly-efficient datapaths, and makes use of a hybrid number representation system. This design inputs and outputs floating-point numbers, but internally makes use of a customizable dynamic fixed-point (FX) number representation. It analyzes and compares the resource utilization, performance and latency of the a hybrid design and a direct core based floating-point design. An empirical study of solution accuracy as a function of resource utilization is presented. From this study it is demonstrated that the newly proposed design can provide better solutions utilizing significantly fewer resources.

It is demonstrated that it is possible to utilize 3.8 times fewer resources, operate at 1.62 times faster clock frequency, and achieve a significant reduction in latency when compared to a floating-point based dot-product. Combining these results and utilizing the spare resources, to instantiate more units in parallel, it is possible to achieve an overall speed-up of at least 5 times.

Future work could be focused on further improvements to the accuracy, resource utilization, and latency by using various number partitioning schemes.

## References


1. D. Boland, and G. Constantinides, "An FPGA-based Implementation of the MINRES algorithm," in *Proc. of Field Programmable Logic*, 2008, pp. 379–384.
2. K. Junaid, and G.Ravindrann, "FPGA Accelerator For Medical Image Compression System," in *Proc. of International Conference on Biomedical Engineering*, 2006, pp. 396–399.
3. IEEE, "754 Standard for Binary Floating-Point Arithmetic," http://grouper.ieee.org/groups/754/, Accessed on 25/10/2009, 1985.
4. M. Langhammer, "Floating point datapath synthesis for FPGAs," in *Proc. of Field Programmable Logic and Applications*, 2008, pp. 355–360.
5. A. Roldao, and G. Constantinides, "High Throughput FPGA-based Floating Point Conjugate Gradient Implementation," Proc. of Applied Reconfigurable Computing, 2008, pp. 75–86.
6. K. Underwood, and S. Hemmert, "Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS Performance," in *Proc. of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 219–228.
7. L. Zhuo, G. Morris, and V. Prasanna, "High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, Oct. 2007, pp. 1377–1392.
8. H. Saleh, and E. Swartzlander, "A Floating-point Fused Dot-product Unit," in *Proc. of IEEE International Conference on Computer Design*, 2008, pp. 427–431.
9. F. Dinechin, "FloPoCo is a generator of arithmetic cores (Floating-Point Cores)," http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/, Accessed on 19/10/2009.
10. Collaborative Project, "Multi-precision Floating Point Library," http://www.mpfr.org/, Accessed on 02/01/2009.