FlexCore: Massively Parallel and Flexible Processing for Large MIMO Access Points

Christopher Husmann, Georgios Georgis, and Konstantinos Nikitopoulos, *University of Surrey;* Kyle Jamieson, *Princeton University and University College London*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/husmann

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

FlexCore: Massively Parallel and Flexible Processing for Large MIMO Access Points

Christopher Husmann[1], Georgios Georgis[1], Konstantinos Nikitopoulos[1], and Kyle Jamieson[2]

[1]5G Innovation Centre, Institute for Communication Systems, University of Surrey
[2]Princeton University and University College London

Abstract

Large MIMO base stations remain among wireless network designers' best tools for increasing wireless throughput while serving many clients, but current system designs, sacrifice throughput with simple linear MIMO detection algorithms. Higher-performance detection techniques are known, but remain off the table because these systems parallelize their computation at the level of a whole OFDM subcarrier, sufficing only for the less-demanding linear detection approaches they opt for. This paper presents FlexCore, the first computational architecture capable of parallelizing the detection of large numbers of mutually-interfering information streams at a granularity below individual OFDM subcarriers, in a nearly-embarrassingly parallel manner while utilizing any number of available processing elements. For 12 clients sending 64-QAM symbols to a 12-antenna base station, our WARP testbed evaluation shows similar network throughput to the state-of-the-art while using an order of magnitude fewer processing elements. For the same scenario, our combined WARP-GPU testbed evaluation demonstrates a 19× computational speedup, with 97% increased energy efficiency when compared with the state of the art. Finally, for the same scenario, an FPGA-based comparison between FlexCore and the state of the art shows that FlexCore can achieve up to 96% better energy efficiency, and can offer up to 32× the processing throughput.

1 Introduction

One of the most important challenges in the design of next-generation wireless communication systems is to meet users' ever-increasing demand for capacity and throughput. Large Multiple Input-Multiple Output (MIMO) systems with spatial multiplexing are one of the most promising ways to satisfy this demand, and so feature in emerging cellular [1] and local-area [21, 22] networking standards. For example, in LTE-Advanced and 802.11ac, up to eight antennas are supported at the access point (AP).

In a system employing spatial multiplexing, multiple transmit antennas send parallel information streams concurrently to multiple receive antennas. While the technique can be used both in the uplink and the downlink of multi-user MIMO networks, here we focus on the uplink case, where several users concurrently transmit to a multi-antenna AP. The need for high throughput in the uplink stems from the emergence of new applications for wireless networks, such as video internet telephony, wireless data backup, and Internet of Things devices, that have shifted the ratio between uplink and downlink traffic quantities towards the former.

However, simply having enough antennas at the AP does not always suffice: to fully realize MIMO's potential throughput gains, the AP must effectively and efficiently demultiplex mutually-interfering information streams as they arrive. Current large MIMO AP designs such as Argos [38], BigStation [54] and SAM [41], however, use linear methods such as *zero-forcing* and *minimum mean squared error* (MMSE). These methods have the advantage of very low computational complexity, but suffer when the MIMO channel is poorly-conditioned, as is often the case when the number of user antennas approaches the number of antennas at the AP [32].

On the other hand, *Sphere Decoder*-based MIMO detectors can boost throughput over linear methods significantly [8, 32], even in cases where the number of user antennas approaches the number of AP antennas. The cost of such methods, however, is their increased computational complexity: compute requirements increase exponentially with the number of antennas [19, 24], soon becoming prohibitive. Indeed, processing complexity is a significant issue for any advanced wireless communication system. While the clock speed of traditional processors is plateauing [12], emerging hardware architectures including GPUs support hundreds of cores, presenting an opportunity to parallelize the processing load, along with the challenge of how to do so most efficiently. BigStation [54] is an example of a system that handles the processing load of a large MIMO system using multiple commodity servers, exploiting parallelism down to the OFDM subcarrier level. Since BigStation uses zero-forcing methods to decode clients' transmissions, this level of parallelism suffices for the AP to be able to decode multiple incoming transmissions and send acknowledgement frames back to the client without letting the wireless medium become idle for more than the amount of time prescribed by the 802.11 standards [22]. The cost, however, is diminished wireless performance, due to linear zero-forcing detection.

Antennas	**Throughput**	**Complexity**
$\mathbf{2\times 2}$	45 Mbit/s	1.2 GFLOPS
$\mathbf{4\times 4}$	100	13
$\mathbf{6\times 6}$	162	105
$\mathbf{8\times 8}$	223	837

Table 1— A summary comparison of the throughput achieved and computational rate required for a Sphere Decoder implementation [32].

Table 1 shows the number of floating point operations per second that a single processing core must maintain to perform optimal, *maximum-likelihood*, depth-first Sphere decoding,[1] as in [32], when processing OFDM symbols at the same rate they arrive over the air, for typical Wi-Fi system parameters.[2] The table is parametrized on the size of the MIMO system (number of clients times number of AP antennas), highlighting the exponential increase in floating point operations per second required to keep up with linearly-increasing numbers of clients. By the time a Sphere decoder reaches eight clients, it must meet a rate on the order of 10^3 GFLOPS, saturating, for example, Intel's Skylake core i7 architecture, whose arithmetic subsystem achieves an order of magnitude less computational throughput [23]. Clearly, the answer to this performance mismatch is to decompose and parallelize the problem, and the aforementioned previous work has made progress in this direction, dedicating one core to each OFDM subcarrier. But as the number of clients grows, because of the mismatch between the exponential growth in required computational rate and the much slower growth of the frequency bandwidth Wi-Fi systems use in general, there is a need for a new family of MIMO detectors that allows parallelization *below* the subcarrier level using a small number of parallel processing elements.

To meet the processing needs of large, high-performance MIMO APs, we present *FlexCore*, an asymptotically-optimal, massively-parallel detector for large MIMO systems. FlexCore reclaims the wasted throughput of linear detection approaches, while at the same time parallelizing processing below the subcarrier level, thus enabling it to meet the tight latency requirements required for packetized transmissions. In contrast to existing low-complexity Sphere decoder architectures [4], FlexCore can exploit any number of available processing elements and, for however many are available, maximize throughput by allocating processing elements only to the parts of the decoding process most likely to increase wireless throughput. Consequently, as additional processing elements are provisioned, FlexCore continues to improve throughput. By this design, FlexCore can avoid unnecessary computation, allocating only as many processing elements as required to approach optimal (*maximum-likelihood*) MIMO wireless throughput performance. FlexCore's computation proceeds in a nearly "embarrassingly" parallel manner that makes parallelization efficient for a broad set of implementation architectures, in particular GPUs, even allowing parallelization across devices. To achieve this, we present novel algorithms to:

1. Choose which parts of the Sphere decoder tree to explore, through a "pre-processing" step, and then,
2. efficiently allocate the chosen parts of the Sphere decoder tree to the available processing elements.

FlexCore's *pre-processing* step is a low-overhead procedure that takes place only when the transmission channel significantly changes. It narrows down which parts of the Sphere decoder tree the system needs to explore to decode the clients' transmissions, *i.e.*, find a *solution* to the decoding problem. The pre-processing step identifies the "most promising" candidate solutions in a probabilistic manner, and occurs *a priori*, without knowing the signals received from the clients themselves, but instead based on the knowledge of the transmission channel and the amount of background noise present. In this part of the design, FlexCore introduces a new probabilistic model to identify the most promising candidate solutions and an indexing technique wherein the tree nodes are labeled by *position vectors*. We also introduce a novel pre-processing tree structure and tree search distinct from the traditional Sphere decoder tree search. These new techniques allow us to efficiently identify the most promising candidate solutions.

FlexCore's *allocation* step maps each of the chosen paths in the Sphere decoder's search tree to a single processing element, spreading the load evenly. While this previously required redundant calculations across parallel tasks as well as multiple sorting operations, FlexCore skips them by introducing a new node selection strategy.

Roadmap. The rest of this paper is structured as follows. We start with a primer of the Sphere decoder in §2, followed by a description of its design in §3. In §4 we present our implementation strategy on both GPUs and FPGAs, highlighting FlexCore's computational flexibility

[1]Simulated results for Rayleigh channel, 16-QAM and 13 dB SNR.

[2]20 MHz frequency bandwidth, resulting in a number of OFDM subcarriers N_c on the order of 50.

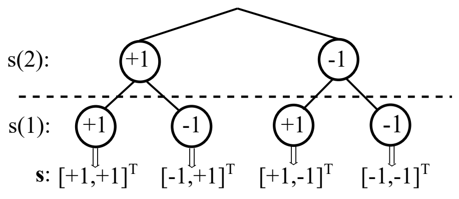

Figure 1— Sphere decoding tree for two transmit antennas, each sending a binary-modulated wireless signal.

across different platforms. Section 5 presents our algorithmic performance evaluation based on both over-the-air experiments and trace-based simulations. In the same section, separate GPU- and FPGA-based evaluations follow, and the applicability of our FlexCore implementations to LTE in terms of computation time is discussed. In §6 we discuss relevant related work, before concluding in §7.

2 Primer: The Sphere Decoder

In order to put FlexCore's high-level design into context, we now introduce the basic principles of the Sphere decoder, then briefly describe the *fixed complexity sphere decoder* (FCSD), an approximate Sphere decoder whose design is amenable to parallelization.

When transmitting a symbol vector $\mathbf{s}$ in an OFDM-MIMO system with N_t transmit and N_r receive antennas (*i.e.*, $N_t \times N_r$ MIMO), with $N_r \geq N_t$, the vector of data arriving at the receive antennas on a particular OFDM subcarrier is $\mathbf{y} = \mathbf{Hs} + \mathbf{n}$, with $\mathbf{H}$ being the $N_r \times N_t$ MIMO channel matrix. The N_t elements of the transmit vector $\mathbf{s}$ belong to a complex constellation Q (*e.g.*, 16-QAM) consisting of $|Q|$ elements. The vector $\mathbf{n}$ represents an N_r-dimensional white Gaussian noise vector. The Sphere decoder transforms the *maximum-likelihood* (ML) problem[3] into an equivalent tree search [44]. In particular, by a QR-decomposition of the MIMO channel matrix ($\mathbf{H} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthonormal and $\mathbf{R}$ an upper triangular matrix), the ML problem can be transformed into $\hat{\mathbf{s}} = \arg\min_{\mathbf{s} \in |Q|^{N_t}} \|\bar{\mathbf{y}} - \mathbf{Rs}\|^2$, with $\bar{\mathbf{y}} = \mathbf{Q}^*\mathbf{y}$. The corresponding tree has a height of N_t and a branching factor of $|Q|$. As shown in Fig. 1 for a 2×2 MIMO system with binary modulation, each level l of the tree corresponds to the symbol transmitted from a certain antenna. Each node in a certain level l is associated with a partial symbol vector $\mathbf{s}_l = [s(N_t - l), \ldots, s(N_t)]^T$ containing all possibly-transmitted symbols from senders down to and including this level, and is characterized by its partial Euclidean distance [44]

$$c(\mathbf{s}_l) = \left[y(l) - \sum_{p=l}^{N_t} R(k,p) \cdot s(p) \right]^2 + c(\mathbf{s}_{l+1}), \qquad (1)$$

with $R(k,p)$ being the element of $\mathbf{R}$ at the kth column and the pth row, and $c(\mathbf{s}_{N_t+1}) = 0$. The ML problem then transforms into finding the tree leaf node with the minimum $c(\mathbf{s}_1)$; the corresponding tree path $\mathbf{s}_1$ is the ML estimate. Many approaches explore the Sphere decoding tree in a depth-first order via which, in contrast to breadth-first approaches, they can guarantee ML performance, and they can adjust their complexity to the channel condition [8, 14, 32]. However, depth-first tree search is strictly sequential, making parallelization extremely difficult: a naive parallel evaluation of all Euclidean distances in order to minimize processing latency would be impractical. For example, for an 8×8 MIMO with 64-QAM modulation, this approach would require performing 2.8×10^{14} Euclidean distance calculations in parallel.

[3] In other words, the problem of finding the most likely set of symbols the transmitting antennas sent.

Fixed Complexity Sphere Decoder. The *Fixed Complexity Sphere Decoder* (FCSD) [4] is an approximate Sphere decoder algorithm: instead of examining (and pruning) all possible Sphere decoder leaf nodes (*i.e.*, all possible solutions), the FCSD visits only a predefined set of them. Specifically, the FCSD visits all nodes at the top L levels of the tree, while for the remaining $N_t - L$ levels, only visits the child node with the smallest partial Euclidean distance (*i.e.*, the branching factor is reduced to one). Since the FCSD visits a predefined set of leaf nodes, it can visit all these in parallel, returning the leaf node with the minimum partial distance as its final answer.

There are however, three important drawbacks to the FCSD's approach. First, the required number of parallel processes has to be a power of the order of the QAM constellation, so the FCSD cannot efficiently adjust to the number of the available processing elements. Second, the visited tree paths are not necessarily the ones that are the most likely to constitute the correct solution, which means that much of the available processing power is not efficiently allocated. Finally, the FCSD cannot differentiate between favorable channel conditions, where even linear approaches would give near-ML performance, and unfavorable channel conditions, where linear approaches are not efficient. In the next section, we describe FlexCore's techniques to address this problem by visiting just the most promising tree paths, maximizing throughput for a given number of processing elements and exploiting any number of available processing elements, not just numbers that are a power of the constellation size.

3 Design

As shown in Fig. 2, FlexCore's architecture consists of two major components: the *pre-processing* module which identifies the most promising tree paths as a function of the MIMO channel and the background noise power, and the *parallel detection* module that actually allocates tree paths to processing elements when the AP decodes an incoming signal. Since FlexCore evaluates the most promising paths by accounting for the transmission channel and background noise, it needs to re-execute pre-processing

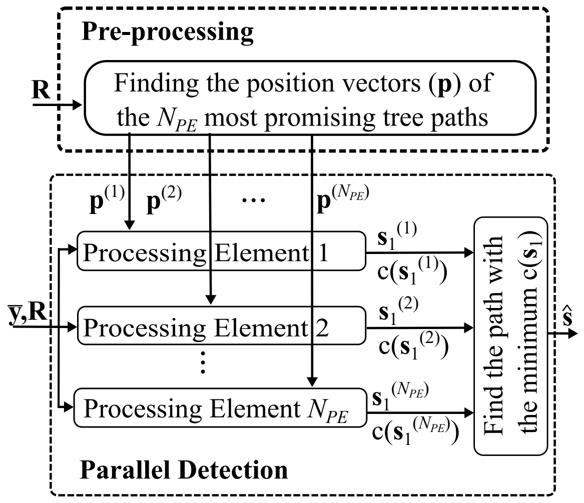
Pre-processing
R
Finding the position vectors (p) of the N_{PE} most promising tree paths
$\mathbf{p}^{(1)}$ $\mathbf{p}^{(2)}$ … $\mathbf{p}^{(N_{PE})}$
y,R
Processing Element 1
Processing Element 2
Processing Element N_{PE}
$\mathbf{s}_1^{(1)}$ $c(\mathbf{s}_1^{(1)})$
$\mathbf{s}_1^{(2)}$ $c(\mathbf{s}_1^{(2)})$
$\mathbf{s}_1^{(N_{PE})}$ $c(\mathbf{s}_1^{(N_{PE})})$
Find the path with the minimum $c(\mathbf{s}_1)$
$\hat{\mathbf{s}}$
Parallel Detection

Figure 2— Block Diagram of FlexCore.

only when the transmission channel changes, similarly to the QR decomposition that is required for Sphere decoding. However, as we show below, the delay introduced by pre-processing is insignificant compared to that of the QR decomposition.

3.1 Pre-processing module

FlexCore's pre-processing uses the notion of a *position vector* **p**, which uniquely describes all the possible tree paths relative to the (unknown) received signal. The position vector is of equal size to the Sphere decoder tree height: each of its elements $p(l)$ takes an integer value ranging from 1 to $|Q|$ that describes the position of the corresponding node at the lth level of the Sphere decoder tree as a function of its index, when sorting the nodes in ascending Euclidean distance order.

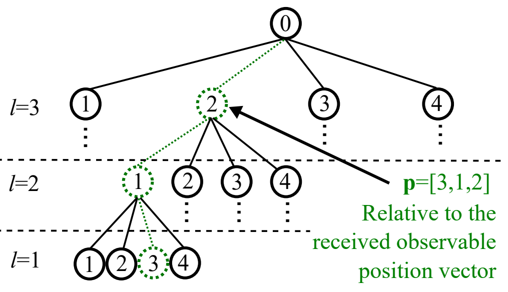

Figure 3— Sorted search for tree for 3 transmit antennas and 4-QAM modulation. The path with the the position vector $\mathbf{p} = [3,1,2]$ is highlighted (green, dashed).

Independent channel example. To illustrate the basic principle behind pre-processing, we present the following simplified example. Suppose that a vector **s** of two binary symbols is transmitted via two independent Gaussian noise channels, with noise powers σ_l^2 with $l = 1,2$ and $\sigma_2^2 \geq \sigma_1^2$, with the symbol $s(l)$ being transmitted

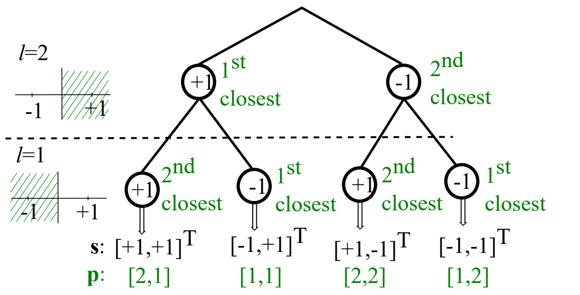

Figure 4— FlexCore's pre-processing and most-promising path selection for the **independent channel example** (two transmit antennas, binary modulation).

through the channel l. In FlexCore each parallel channel corresponds to a level of the Sphere decoding tree.[4] All possible transmitted symbol combinations are shown in Fig. 4. It is known from detection theory that the best decoding method for this example is to choose the symbols lying on the same side of the x-axis (positive or negative) as the received signals lie. That means that the most likely solution in the tree of Fig. 4 is the path consisting of the first-closest symbols to the received signal (on each parallel channel) and, therefore, its position vector is $\mathbf{p} = [1,1]$. It can also be shown that the corresponding probability of including the correct vector is $P_c(\mathbf{p} = [1,1]) = (1 - P_e(1))(1 - P_e(2))$, with $P_e(l)$ being the error probability of binary modulation for a noise variance of σ_l^2. The second most likely path to include the correct solution is for a case where on one parallel channel the symbol lies on the same side of the x-axis as the received observable (*i.e.*, the tree path includes the closest symbol to the received observable) and on the other channel the symbol and received observable lie on different sides of the x-axis (*i.e.*, the tree path includes the second-closest symbol to the received observable). This means that the second-most-promising path is either the path $\mathbf{p} = [1,2]$ or the path $\mathbf{p} = [2,1]$.[5] Finally, the least-promising path is $\mathbf{p} = [2,2]$ with $P_c(\mathbf{p} = [2,2]) = P_e(1) \cdot P_e(2)$.[6]

After performing the pre-processing step, and when the actual received signal **y** is available, detection takes place. In the case that our system has only two available processing elements, FlexCore calculates the Euclidean distances for the two most promising paths $\mathbf{p} = [1,1]$ and $\mathbf{p} = [1,2]$. Then, the detection output is the vector with the smallest calculated Euclidean distances (Fig. 2).

Generalizing to the MIMO channel. In a similar manner to the independent Gaussian channels case, for the actual Sphere decoding tree, and for any QAM constella-

[4] In the Sphere decoding case, however, the levels are not independent but, as we show in the Appendix, the following approach still applies.

[5] Since we assumed that $\sigma_2^2 \geq \sigma_1^2$, then $P_c(\mathbf{p} = [1,2]) \geq P_c(\mathbf{p} = [2,1])$ and therefore $(1 - P_e(1))P_e(2) \geq P_e(1)(1 - P_e(2))$.

[6] We note again that the position vector identifies these tree paths in terms *relative* to the signal that the AP will later receive, instead of identifying absolute tree paths, hence pre-processing is possible *a priori*.

tion, the corresponding probabilities can be approximated as

$$P_c(\mathbf{p}) \approx \prod_{l=1}^{N_t} P_l\left(p(l)\right) \tag{2}$$

with

$$P_l\left(p(l)\right) = \left(1 - P_e(l)\right) \cdot \left(P_e(l)\right)^{(p(l)-1)} \tag{3}$$

and

$$P_e(l) = \left(2 + \frac{2}{\sqrt{|Q|}}\right) \cdot \operatorname{erfc}\left(\frac{|R(l,l)| \cdot \sqrt{E_s}}{\sigma}\right) \tag{4}$$

where *erfc* is the complementary error function, which can be calculated on-the-fly or pre-calculated using a look-up table, E_s is the power of the transmitted symbols, σ^2 is the noise variance, and $p(l)$ is the l^{th} element of $\mathbf{p}$. We defer a mathematical justification to the Appendix.

3.1.1 Finding the most promising position vectors

FlexCore needs to identify the set $\mathscr{E}$ consisting of the N_{PE} most promising position vectors, with N_{PE} being the number of available processing elements. An exhaustive search over all possible paths becomes intractable for dense constellations and large antenna numbers. Therefore, we translate the search into a new *pre-processing tree* structure (distinct from the Sphere decoder tree) and propose an efficient traversal and pruning approach that substantially reduces pre-processing complexity.

We now explain how to construct and traverse the pre-processing tree with an example: Fig. 5 shows its structure for three transmit antennas. Each node in the tree can be described by a position vector and its likelihood P_c (Eq. 2). Tree construction begins by setting the tree

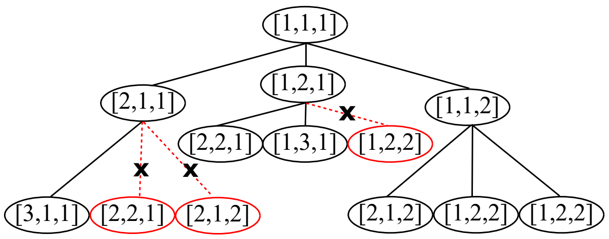

Figure 5— A *pre-processing tree* construction for three transmit antennas.

root to a node whose position vector consists of only ones ($\mathbf{p} = [1,1,\dots,1]$) since this will always be the "most promising" one, regardless of the MIMO channel. Then we expand the tree root node, namely, we construct its child nodes and calculate their P_c values. To find the w^{th} child node ($w \in [1,\dots,N_t]$), we increment the w^{th} element of the parent's position vector by one, as shown in Fig. 5. To avoid duplication of pre-processing tree nodes, when expanding a node whose position vector has been generated by increasing its l^{th} element, the w^{th} children nodes ($w \in [l+1,\dots,N_t]$) are not expanded. To avoid unnecessary computations while calculating the P_c values of the children nodes, we further observe that for two position vectors $\tilde{\mathbf{p}}$ and $\mathbf{p}$ that only differ in their w^{th} component, their probabilities are related by $P_c(\tilde{\mathbf{p}}) = P_c(\mathbf{p}) \cdot P_e(w)$. After expanding the tree root, we include its position vector in the set $\mathscr{E}$ of most promising position vectors and we store all the children nodes of the expanded node and their P_c values in a sorted candidate list $\mathbb{L}$ (of descending order in P_c). Tree traversal continues by expanding the node with the highest P_c value in $\mathbb{L}$. The expanded node is then removed from $\mathbb{L}$, its position vector is added to $\mathscr{E}$ and the node's children are appended to $\mathbb{L}$. Whenever $|\mathbb{L}|$ exceeds N_{PE}, we remove from $\mathbb{L}$ the $|\mathbb{L}| - N_{PE}$ nodes with the lowest P_c values. The process continues until $|\mathscr{E}| = N_{PE}$. Finally, we introduce a *stopping criterion* in order to terminate the tree search if the sum of the P_c values of the vectors currently in $\mathscr{E}$, is larger than a predefined threshold. An example case is discussed in Section 5.

Pre-processing complexity. In terms of this process' complexity, we first calculate the error probabilities ($P_e(l)$ in Eq. 4), which can be computed once and reused several times during pre-processing. Then, we require in the worst case N_t real multiplications per expanded node. Since the maximum amount of expanded nodes is at most N_{PE}, the maximum complexity in terms of real multiplications is $(N_{PE} \cdot N_t)$. In very dense constellations (*e.g.*, 256-, 1024-QAM) a rather large number of parallel processing elements may be required to reach near ML-performance. In such challenging scenarios, sequential execution of the pre-processing phase may introduce a significant delay. However, our simulations have shown that a parallel expansion of the nodes with the highest probabilities P_c in $\mathbb{L}$ is possible with negligible throughput loss compared to a sequential implementation, provided that the ratio of available processing elements N_{PE} to the number of nodes expanded in parallel is greater than ten. As a result, the latency of the pre-processing step for large MIMO systems is insignificant compared to that of the QR decomposition. In MIMO systems with dynamic channels and user mobility, the most promising paths will vary in time. Therefore, and as validated in [17], in such cases reliable channel estimates are still required to preserve the gains of spatial multiplexing. FlexCore will then leverage these estimates to recalculate the most promising paths, together with the traditionally required channel-based pre-processing (*e.g.*,

	Pre-Processing			Detection	
	QR/ZF	FlexCore		FlexCore	
		$N_{PE}=32$	$N_{PE}=128$	$N_{PE}=32$	$N_{PE}=128$
8×8	≈2048	102	301	4608	18432
12×12	≈6912	136	391	9984	39936
Parallelizability	-	3	12	32	128

Table 2— Complexity in real multiplications and "parallelizability" of Pre-Processing and FlexCore detection.

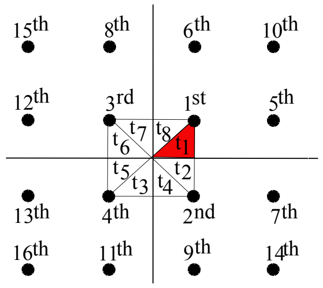

Figure 6— Detection square and triangles 1-8 for 16-QAM and approximate predefined symbol ordering, calculated when the received symbol is within triangle t_1.

channel inversion for linear detection or QR decomposition for Sphere decoder-based detection). In such dynamic channels, pre-processing complexity requirements can become comparable to those of FlexCore's detection, as shown in Table 2. Latency requirements can then be determined by the required sequential QR decomposition (or channel inversion for linear detection).

3.2 Core Allocation and Parallel Detection

To perform detection in parallel, each of the calculated position vectors in $\mathscr{E}$ has to be allocated to a different processing element. FlexCore's pre-processing has already identified the position vectors as a function of their Euclidean distance sorted order. For example, if the corresponding position vector is $\mathbf{p} = [3,1,2]$, then the tree path to be processed consists of the node with the second smallest Euclidean distance at the top level, the smallest in the second level and the third smallest in the first level. Typically, finding the node (*i.e.*, QAM symbol) with the third smallest Euclidean distance to the observable would require exhaustive calculation of all Euclidean distances at a specific level (*e.g.*, for 64-QAM it would require 63 unnecessary Euclidean distance calculations). In order to avoid these unnecessary computations, we exploit the symmetry of the QAM constellation, defining an approximate predefined order based on the relative position of the "effective" received point $\tilde{y}_l$ in the QAM constellation.

$$\tilde{y}_l = \left(y_l - \sum_{p=1+l}^{N_t} R(l,p) \cdot s(p) \right) / R(l,l). \qquad (5)$$

We calculate the approximate predefined symbol order by assuming that the "effective" received point lies in a square which is centered at the center of the QAM constellation and of side length equal to the minimum distance between consecutive constellation symbols as in Fig. 6. We then split the square into eight triangles t_i ($i = 1,\dots,8$) and via computer simulations, compute the most frequent sorted order for "effective" received points lying in these triangles (as a function of their relative position to the center of the square), storing the order in a look-up table. Fig. 6 shows the resulting approximate order for a 16-QAM constellation when the received point lies within t_1. We note that the constellation's symmetrical properties allow us to store the order of just a single triangle (*e.g.*, for t_1) since the order for all other triangles will be just circularly shifted (with a center one of the constellation points). During actual decoding, at each level we identify the relative position of the square as well as the relative position of the received point within the square. We then identify the symbol with the k^{th} smallest distance by using the predefined ordering. If, however, the latter points to a symbol that is not part of the constellation (*i.e.*, the center of the grid is not the same as the center of the constellation), then the corresponding Euclidean distance calculation unit is deactivated.

4 Implementation

To showcase the versatility and efficiency of FlexCore we implement it on FPGAs and GPUs. In particular, since FlexCore focuses available processing resources to the most promising parts of the Sphere decoding tree, we demonstrate that it can consistently outperform state-of-the-art implementations, regardless of the underlying platform. We first evaluate FlexCore's gains when realized on GPUs based on the Compute Unified Device Architecture (CUDA) programming model [31]. These types of many-core architectures are among the more challenging technologies to display FlexCore's actual gains, as the developer does not have direct control over the allocation of processing elements, but can instead control the way parallel threads are generated.

Additionally, we implement FlexCore on FPGAs, a platform that is less programmable than a GPU but more able to be tailored to FlexCore's design. FPGA implementation of fixed-complexity detection schemes has been examined in the literature [2, 5, 26, 49], enabling low latency and high throughput detection. Apart from high performance at a low power envelope [16], FPGAs allow greater flexibility for the examined fixed complexity schemes. For instance, the designer may choose to implement in parallel a fraction or even a single path of the total paths required. Therefore, FPGAs are more effective in highlighting FlexCore's computational flexibility.

GPU-based detector implementation. To implement FlexCore we have extended the MIMOPACK library [34] by introducing support for single-precision floating-point computations, and therefore we have achieved improved processing and memory transfer throughput, and reduced storage requirements. To further improve transfer throughput we added support for non-pageable host memory allocation. Finally, we provided support for streams, a means of concurrent execution on GPUs through overlapping asynchronous (i.e., in practice independent) operations.

The parallel FCSD generates $N_{sc} \times |Q|^L$ threads on the GPU, with L being the number of levels to be fully expanded and N_{sc} the number of subcarriers to be processed in parallel, given it is supported by the memory of the GPU. To facilitate parallel computations, storage for all position vectors is allocated in advance by the library. Our FlexCore implementation generates $N_{sc} \times |\mathscr{E}|$ threads. We note that FlexCore has a higher workload compared with FCSD due to the additional arithmetic/branching operations and their application to the topmost level of the tree. Memory-wise, compared to MIMOPACK's FCSD our FlexCore implementation requires three additional Host to Device (H2D) transfers: a) two $|Q| \times 4$-byte wide involving the order of a single triangle and b) one $N_{sc} \times N_t \times |\mathscr{E}|$-byte wide containing the tree paths. Since the latter matrix essentially represents the position vectors, its values can be limited to single bytes for $|Q| \leq 256$.

FPGA-Based Detection Design. FlexCore was designed with latency minimization in mind and thus, we present a pipelined parallel architecture and the implementation of both FCSD's and FlexCore's detection engines. For the purpose of consistency, we consider our processing element as the fully-instantiated logic required to process a whole Sphere decoder path from the top to the bottom level of the tree. To implement the fixed-complexity schemes, we have designed modular, low-complexity and highly parameterizable fixed-point architectures using Verilog RTL code. We have explicitly designed each branch at every level, replicated branches and connected levels in a structural manner. In order to save resources, in the topmost FCSD level we employ constant complex coefficient multipliers (CCMs) which perform a hard-wired integer multiplication of $R(l,l)$ by the complex constellation point value. Moreover, to avoid the division in Eq. 5, we consider the effective received point as $\tilde{y}_l \cdot R(l,l)$, (for all comparisons involving the constellation plane, we multiply the latter's values by $R(l,l)$). To save DSP48 resources (Xilinx's embedded FPGA multiplication/arithmetic logic unit) and to maintain a low utilization of the generic FPGA fabric, we employ multiple constant coefficient multipliers (MCMs) similar to the ones described in [11] (*i.e.*, employing indices for $\mathbf{s}$).

Fig. 7 shows an overview of the detection engines' architecture and modular design principle which allows for reusability, fair comparison and an instantiation of an arbitrary number of detection paths. Bit widths and pipeline levels are parameterizable at instantiation time. All levels of FlexCore are implemented by replicating the branches displayed in the top left of Fig. 7. The FCSD's design follows a very similar approach allowing the reuse of FlexCore's modules, apart from the topmost FCSD level, where for its fully parallel implementation we designed and instantiated $|Q|$ CCMs in parallel. Note that the high level architecture of every branch remains almost identical to the one displayed in Fig. 7, apart from its $\tilde{y}_l \cdot R(l,l)$ unit, whose complexity increases as we approach the bottom tree levels. Additionally to the FCSD, FlexCore's branches include pipeline registers to store the desired closest point offset to the received vector and non-pipelined registers storing the order for a single triangle. FlexCore's slicer computes the midpoint value and index instead of the actual constellation point, forwarding the results to the FlexCore engine which outputs the detected constellation point index. The l^2 norm unit (Fig. 7, bottom-left) is designed to directly employ two cascaded DSP48 FPGA slices, one in multiplication mode and the second in multiply-add mode. Finally, a parameterizable (regarding both the supported width and the number of elements) pipelined minimum tree unit outputs the detected solution. At a minimum level of pipeline (*i.e.*, submodule i/o and three registers per embedded multiplier), the total FCSD latency is 95 up to 150 clock cycles ($N_t = 8$ and 12 respectively). The FlexCore engine induces an additional minimum latency of 5 cycles per level.

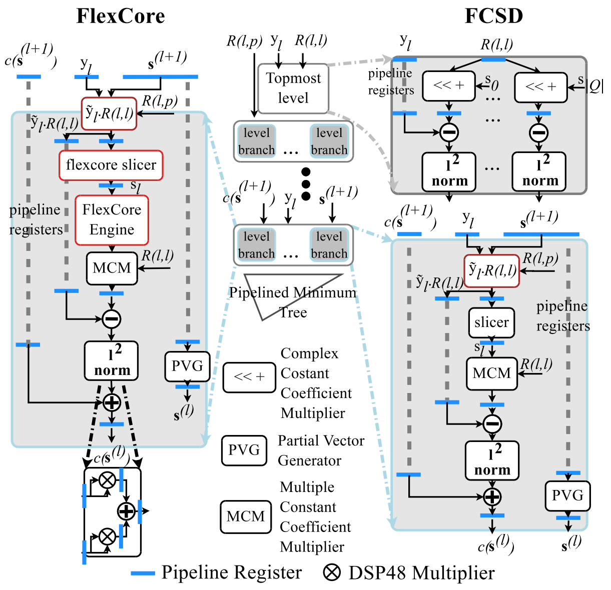

Figure 7— FPGA architecture of proposed detection engines: FlexCore (left) and FCSD (right).

5 Evaluation

In this section we discuss FlexCore's algorithmic performance and implementation aspects compared to the state-of-the-art. We first evaluate FlexCore's throughput performance on our WARP v3 testbed. Based on these results, we then jointly assess FlexCore's algorithmic and GPU implementation performance. Finally, we provide a design space exploration of high-performing detection for various parallelization factors on the state-of-the-art Xilinx Virtex Ultrascale xcvu440-flga2892-3-e FPGA. Our evaluation focuses on scenarios where the channel is static over a packet transmission and it does not account for the pre-processing complexity. Since the pre-processing task needs to take place any time the channel changes (see

§3), the corresponding overhead can be easily calculated based on the assumed channel dynamics.

5.1 Throughput Evaluation

Methodology and setup. To evaluate FlexCore's throughput gains we use Rice's WARP v3 radio hardware and WARPLab software. We employ 16- and 64-QAM modulation with the 1/2 rate convolutional coding of the 802.11 standard. Each user transmits 500-kByte packets over 20 MHz bandwidth channels within the 5 GHz ISM band in indoor (office) conditions. We implement an OFDM system with 64 subcarriers, 48 of which are used to transmit payload symbols, similarly to the 802.11 standard. Eight- and 12-antenna APs are considered, with the distance between co-located AP antennas to be approximately 6 cm. For the eight-antenna AP case, evaluations have been made purely by over-the-air experiments involving all necessary estimation and synchronisation steps (*e.g.*, channel estimation). For the 12-antenna AP case, and due to restrictions on the available hardware equipment, evaluation is performed via trace-driven simulation. To collect the corresponding MIMO channel traces, we have separately measured (over the air) and combined the received channel traces of single-antenna users to 12-antenna APs (1×12) . Fig. 8 displays a graphical overview of our testbed with the positions of the eight- and 12- antenna APs. Similarly to [32], the individual SNRs of the scheduled users differ by no more than 3 dB. This minimizes the condition number of the channel (a low condition number is an indicator of a favorable channel) but therefore also limits the potential gains of FlexCore and Sphere decoding approaches in general. For all our evaluations, the examined SNR is such that an ML decoder reaches approximately the practical packet error rates (PER_{ML}) of 0.1 and 0.01 when the number of active users is equal to the number of the AP antennas. For the realization of both FlexCore and FCSD we employ both the sorted QR decompositions of [4] and [13] and we show the best achievable throughput.

FlexCore's throughput for $N_t = N_r$. Fig. 9 shows the achievable network throughput of FlexCore, FCSD and the trellis-based parallel decoder introduced in [50], for several numbers of available processing elements, against the throughput achieved by exact ML detection and linear MMSE detection. The evaluation is based on the assumption that minimum latency is targeted. Therefore, each parallel element is only allocated to one parallel task. In the case of FlexCore and FCSD, each processing element is used to calculate the Euclidean distance of a single tree path per received MIMO vector. In [50] each processing element calculates the partial Euclidean distance of each constellation point. As a result, [50] would also require a fixed number of processing elements, equal to the QAM constellation's size. In practice, for all schemes, a processing element could be used multiple times to carry out multiple parallel tasks sequentially, but this would result in an increase in latency. In agreement with the literature [32, 38, 54], Fig. 9 demonstrates that linear detection results in a poor throughput when $N_t = N_r$.[7] It also shows that while the trellis-based method of [50] outperforms MMSE, it is consistently inferior to FCSD and FlexCore, in all evaluated scenarios. In addition, it requires a fixed number of processing elements, and is therefore unable to scale its performance with the number of available processing elements. Due to these limitations, in the rest of the paper we focus on comparing FCSD and FlexCore.

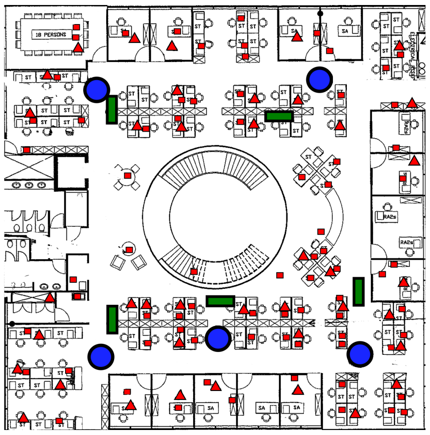

Figure 8— Testbed floorplan (circles: 8-antenna APs, rectangles: 12-antenna APs, triangles: single-antenna users transmitting to 8-antenna APs, squares: single-antenna users transmitting to 12-antenna APs).

Fig. 9 shows that, in contrast to FSCD, FlexCore operates for any number of available processing elements and it consistently improves throughput when increasing the available processing elements. On the other hand, and as discussed in §2, the FCSD can fully exploit processing elements as long as their number is a power of the order of the employed QAM constellation. Figure 9 also shows that for a given number of available processing elements, FlexCore consistently outperforms FCSD in terms of throughput. When 12 users transmit 16-QAM symbols to a 12-antenna AP, at an SNR such that $PER_{ML} = 0.1$ (SNR=13.5 dB), when 196 parallel elements per subcarrier are available, FlexCore can provide nearly 2.5× the throughput of the FCSD. In addition, due to FlexCore's pre-processing, which focuses the available processing power to the tree paths that are most likely to increase wireless throughput, FlexCore requires significantly fewer processing elements than FCSD to reach the same throughput. For example, in a 12×12 64-QAM MIMO system and at an SNR such that $PER_{ML} = 0.01$ (SNR=21.6 dB), FlexCore requires 128 parallel paths to

[7] We note that MMSE can achieve better throughput if we allow $N_t < N_r$. This is shown in [32], as well as in Fig. 10.

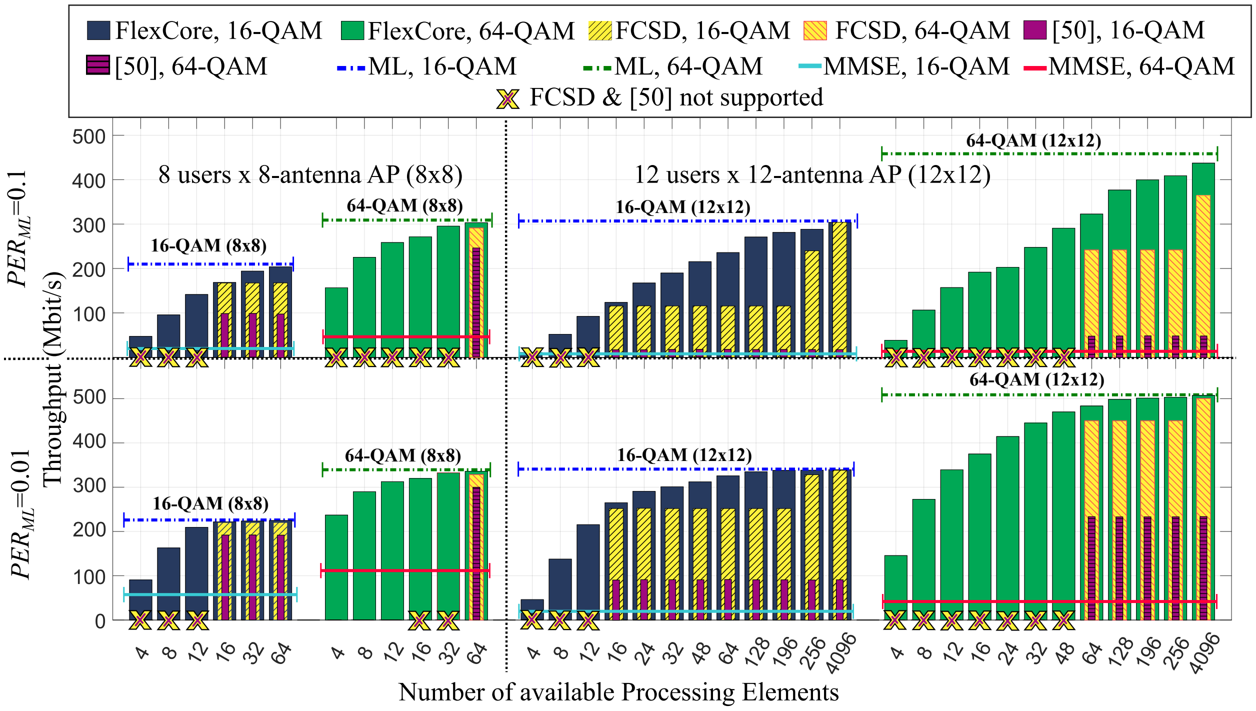

Figure 9— Achievable network throughput of FlexCore, FCSD and trellis-based decoder [50] for minimum processing latency, as a function of the available processing elements, compared to optimal (ML) detection and MMSE.

reach 95% of the ML-bound, whereas FCSD requires 4096. Fig. 9 also shows that, in principle, the gains of FlexCore against the FCSD increase when the transmission conditions become more challenging. Namely, when the number of antennas and the QAM constellation's order increase, and when the SNR decreases (and therefore the PER of the ML solution increases).

FlexCore's throughput v. number of users. The bars in Fig. 10 show the achieved network throughput of FlexCore against the throughput of Geosphere and MMSE, as a function of the number of active users simultaneously transmitting 64-QAM symbols to a 12-antenna AP at an SNR such that $PER_{ML} = 0.01$ (SNR=21.6 dB). We assume that 64 processing elements per sub-carrier are available for FlexCore. However, we also consider an adjustable version of FlexCore (*a-FlexCore*) that from the 64 available processing elements, uses as many as required so that the sum of the P_c values of the corresponding most promising paths becomes 0.95. As expected [38,54], Fig. 10 shows that MMSE is almost optimal only when the number of active users is significantly smaller than the number of AP antennas. In contrast, exact or approximate ML methods, including FlexCore, can support numbers of users that are similar to the number of the AP antennas and still scale network throughput. This ability to reclaim the unexploited throughput of linear detectors separates FlexCore from prominent large MIMO architectures such as [38,54]. Fig. 10 also shows that in contrast to the previously proposed parallel schemes, FlexCore has the ability to adjust the number of activated processing elements and therefore the overall complexity to the channel conditions. When the number of active users is significantly smaller than the one of the AP antennas, where the MIMO channel is well-conditioned and linear detection methods also perform well, a-FlexCore reduces the number of active processing elements to almost one, resulting in an overall complexity similar to that of linear methods.

5.2 Algorithmic/GPU-Based Evaluation

Methodology and setup. We compare FlexCore's combined kernel execution and memory transfer times against MIMOPACK's parallel and single-threaded FCSD employing CUDA 7.5 and OpenMP. We choose MIMOPACK's FCSD over other state-of-the-art GPU implementations such as [10], since MIMOPACK is available as open-source. Implementing both detectors on the same library leads to the most fair comparison of the underlying algorithms, which is the aim of this work. We then evaluate detection performance, based on our previous assessment, in the context of the LTE standard [1]. GPU simulations are executed on the Maxwell-based GTX 970 device CPU simulations based on the OpenMP library are executed on the octacore FX-8120 x86_64 general purpose processor using 16 GB of RAM. Our profiling setups involve $|Q| \in \{16, 64\}$, $N_t \in \{2, ..., 16\}$ and $64 \leq N_{sc} \leq 168,140$ (256 thread blocks).

FlexCore's speedup gains. Fig. 11 overlays FlexCore's

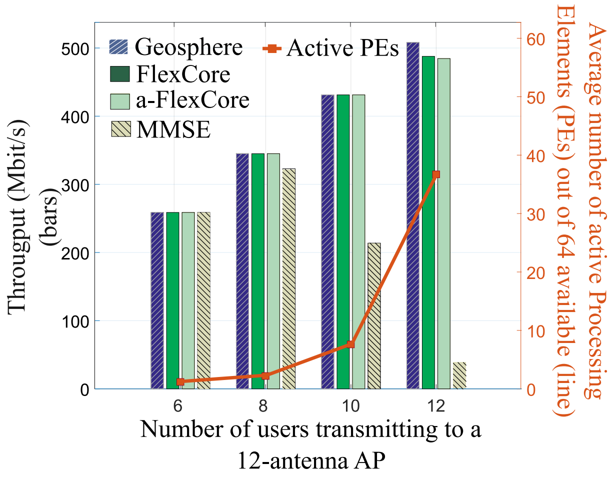

Figure 10— *Bars*: Network throughput of FlexCore and a-FlexCore with 64 available processing elements against Geosphere and MMSE, for a 12-antenna AP with six to 12 simultaneous users. *Line*: Corresponding average number of activated processing elements for a-FlexCore.

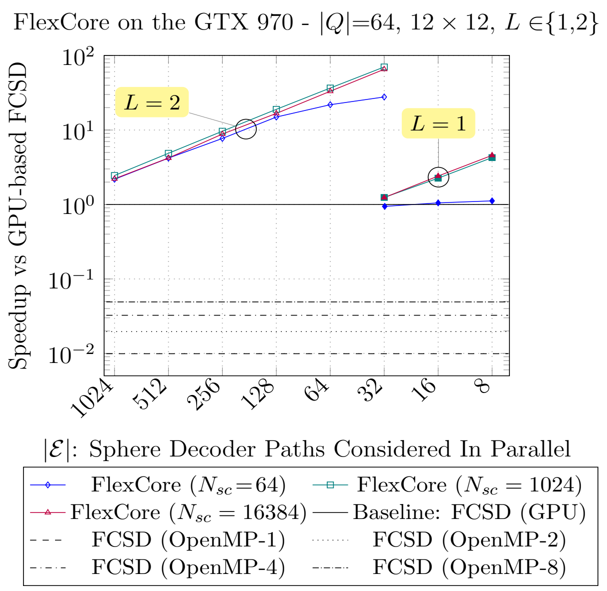

Figure 11— FlexCore's speedup vs FCSD (GPU/CPU).

speedup in 12×12 scenarios using 64-QAM modulation against the case where FCSD fully expands $L \in \{1,2\}$ levels (Sec. 2). The solid horizontal line depicts the GPU-based FCSD (baseline reference), while dashed and dotted lines display the results of CPU-based execution for a varying number of OpenMP threads (denoted as OpenMP-1, OpenMP-2 etc.). The horizontal axis lists the number of Sphere decoder tree paths considered in parallel by FlexCore.

Fig. 11 shows that the GPU-based FCSD is at least 21× faster than the 8-threaded CPU version which in turn provides a maximum speedup of 5.14× over single-threaded execution (*i.e.*, a 64.25% parallel efficiency). Thus, the many-core implementation and our MIMOPACK enhancements benefit both detection schemes. GPU results also show an expected sublinear increase in relative speedup as the thread ratio increases. At the SNR for which PER_{ML} is 0.01, (Fig. 9), FlexCore's speedup against the GPU-based FCSD increases up to 19×, as we require just 128 parallel paths to reach the same performance (Fig. 11, $L=2$). Speedup is maximized when we process in parallel a sufficient number of sphere decoder paths and/or subcarriers (*e.g.*, $N_{sc} \geq 1024$ at high occupancy). When jointly assessing performance and power by employing the Joules per bit index (computed as Power (W) / Processing Throughput (bps)), FlexCore is 17.3% ($N_t = 8$) up to 51.3% ($N_t = 12$) more energy efficient. For $L=2$, FlexCore's energy advantage increases by 97.5%.

FlexCore for LTE. To obtain system context, we assess computation time, including data transfers, with respect to the 3GPP LTE standard [1]. LTE requires that a 10 ms frame contains 20 timeslots, each with a 500 µs duration (a total of 140× the number of occupied subcarriers). Fig. 12 shows the corresponding 64-QAM SNR loss for Successive Interference Cancellation (SIC) [47], the FCSD and FlexCore compared to ML detection, based on the supported number of paths. For $N_t = 8$ and when employing 8 streams, FlexCore supports 105 down to 4 paths for the two extremes of the LTE modes for which the SNR loss is 0.2 to 2.1 dB. In the $N_t = 12$ case, the corresponding SNR loss becomes 0.9 to 9.8 dB (68 down to 2 paths). In the case of SIC (essentially a single-path FlexCore), the loss can be up to 11.9 dB. Notice that even though FlexCore's threads have a higher workload compared to FCSD's, the latter's inherent lack of flexibility significantly limits support to just the 1.25 MHz LTE mode for $L=1$ at $N_t \in \{8,12\}$. We note that when $L=2$, the FCSD fails to meet the LTE requirement for $N_t = 8$, $|Q| = 16$ and the more demanding cases.

5.3 Algorithmic/FPGA-Based Evaluation

Methodology and setup. We first present the implementation cost, achieved frequency and power consumption of a single processing element, considering $|Q| = 64$, $N_t \in \{8,12\}$ at the minimum level of pipeline (Sec. 4) and 16-bits width. Then, based on the results of Sec. 5.1, we jointly evaluate power and FPGA processing throughput. Note that the number of processing elements M that can be instantiated in practice on an FPGA is limited by the latter's available resources. Due to our pipelined design though, the number of processing elements does not need to be equal to the number of Sphere decoder paths. Thus, we explore performance for varying values of M at a 5.5 ns delay (the minimum supported by both detection engines up to the number of instantiated processing elements). We note that due to host system memory lim-

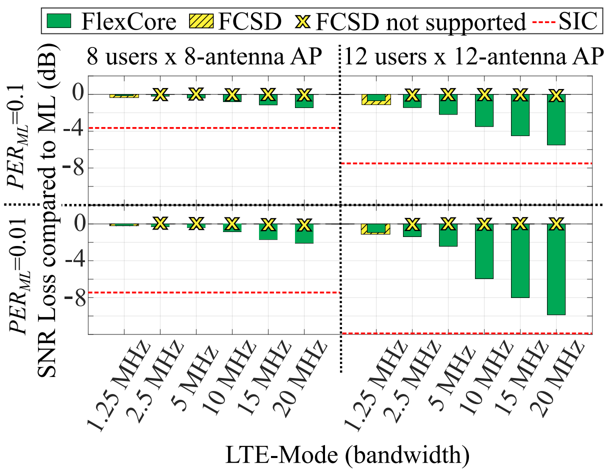

Figure 12— FlexCore, FCSD and SIC on the GPU at 64-QAM, against the ML SNR, considering the detection latency requirements of the several LTE modes.

$N_t \times N_t$	64-QAM	CLB	LUTs Logic	LUTs Mem	LUTs FF-pairs	DSP48 slices	f_{max} (MHz)	Power (W)
8×8	FlexCore	3206	15276	1187	5363	16	312.5	6.82
	FCSD	2187	11320	713	4717	16	370.4	6.54
12×12	FlexCore	5795	28810	2497	11415	24	312.5	9.157
	FCSD	4364	23252	1537	10501	24	370.4	9.04

Table 3— Single processing element on the XCVU440-flga2892-3-e for FlexCore and the FCSD at 64-QAM. FlexCore's path increases area-delay product on average only by 73.7 to 57.8% ($N_t = 8$ and $N_t = 12$ respectively).

itations, $M \leq 32$ in the case of FlexCore (both antenna setups) and $M \leq 32$, $M \leq 64$ respectively for the FCSD at $N_t = 12$ and $N_t = 8$. We estimate power through Xilinx's Power Estimator, under worst-case static power conditions at 100% utilization. Results are compared using the area-delay product and the efficiency index of Joules/bit.

FlexCore's single-path cost. Our FPGA implementation shows that FlexCore's significant advantage in terms of numbers of required processing elements (see Fig. 9), comes at a small implementation overhead per processing element. Table 3 presents implementation results for $M = 1$, where a CLB is a Configurable Logic Block, containing Look-up Tables (LUTs), Flip-Flops (FFs) and distributed RAM. In fact, the processing element overhead tends to decrease as N_t increases; in the case of FlexCore and FCSD respectively for $N_t = 12$, there is a 1.81× and 1.99× area delay product increase compared with the $N_t = 8$ case. Timing analysis shows that logic delay is below 2 ns; the rest is attributed to routing.

Flexibility revisited: Multi-Path performance. FlexCore's design allows detection by employing an arbitrary number of Sphere decoder paths and for $M = 32$, its processing throughput on this device can reach 13.09 Gbps when 32 paths need to be processed, to 3.27 Gbps (128

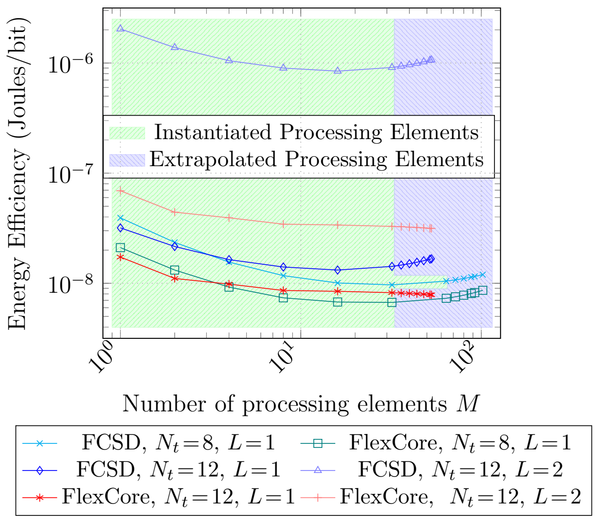

Figure 13— FPGA energy efficiency exploration on the XCVU440-flga2892-3-e under the same FlexCore and FCSD network throughput requirements (Fig. 9, $|Q| = 64$).

required paths). We remind that for 12×12 MIMO, FlexCore with 32 and 128 paths reaches the same network throughput as the FCSD does with 64 and 4096 paths respectively (see Fig. 9). To support the 20 MHz LTE bandwidth for 32 and 128 Sphere decoder paths, three or more and nine or more FlexCore processing elements need to be respectively instantiated at 5.5 ns. In contrast, as the FCSD requires at least $|Q|$ PD outputs (to visit all nodes on all tree levels for $L = 1$), its processing throughput is $\frac{log_2(|Q|) \cdot N_t \cdot f_{max}}{|Q|/M}$ Mbps (f_{max} is the maximum operating frequency in MHz). Even when we instantiate $M = 64$ processing elements, the FCSD fails to meet the 20 MHz LTE bandwidth for $L = 2$ (4096 required paths) and it requires instantiation of twice as many processing elements as FlexCore to support the 1.25 MHz mode. It is thus less area and energy efficient. Fig. 13 displays the energy efficiency of the proposed detection engines in terms of J/bit for a varying number of processing elements to reach the same network throughput (see Fig. 9). For extrapolating the numbers of processing elements that can be instantiated, a 75% maximum device utilization [3] is assumed in order to retain performance by avoiding routing congestion. Overall, the FCSD requires on average 1.54× up to 28.8× more J/bit ($N_t = 8$, $L = 1$ and $N_t = 12$, $L = 2$ cases respectively). These results illustrate FlexCore's significant efficiency advantage. To put GPU performance into perspective, our FPGA implementations achieve a two and three orders of magnitude higher energy efficiency for $L = 1$ and $L = 2$, respectively.

Discussion. We have shown that FlexCore can efficiently exploit any number of available processing elements, and can outperform linear detection methods even when a small number of processing elements is available. Compared to the FCSD, FlexCore can provide significant throughput gains for the same number of processing elements and it can achieve near-optimal performance with a number of processing elements that can be more than an order of magnitude less. FlexCore's modest detection requirements translate to a GPU implementation speedup of 19$\times$ (64-QAM, 12×12 MIMO at 21.6 dB) against the FCSD when the latter fully expands two levels (Fig. 9 and 11). FlexCore's flexibility allows it to operate in varying conditions, in which the FCSD fails to meet the LTE's timing requirements. To the best of our knowledge, FlexCore is the first Sphere-decoding-based detection scheme that can support all LTE bandwidths, while providing performance better than the one of SIC, even for 12×12 MIMO systems on a GPU [33,35,40,50,51]. Our FPGA implementation results reveal that FlexCore is 34% to 96% more energy efficient than the proposed FCSD implementation and can reach a throughput of 13 Gbps on a state-of-the-art device. Also to the best of our knowledge, these are the first FCSD-based FPGA implementations for 8×8 and larger antenna arrays [2,5,26,49].

6 Related Work

The exploitation of parallelism alone to reduce processing latency is not a novel approach: indeed, implementations of all kinds of Sphere decoders (e.g., both breadth-first and depth-first) involve some level of parallelism. However, existing approaches either take a limited or inflexible level of parallelism or they perform parallel processing takes place in an suboptimal, heuristic manner, without accounting for the actual transmission channel conditions.

Parallelism at a distance calculation level. Both depth-first [8, 20, 45] and breadth-first Sphere decoder implementations, including the *K-Best sphere decoders* [9,18,28,30,36,37,46,48], calculate multiple Euclidean distances in parallel any time they change tree level. In addition, after performing the parallel operations, the node or list of nodes with the minimum Euclidean distance needs to be found, which requires a significant synchronization overhead between the parallel processes. This level of exploited parallelism is fixed, predetermined, non-flexible, with high dependencies which are related to the specific architectural design. In addition, in K-Best Sphere decoders the value of K, which is predetermined, needs to increase for dense constellations and large numbers antennas, making K-best detection inappropriate for dense constellations and large MIMO systems. Using FlexCore's approach we can adaptively select the value of K, which will differ per Sphere decoding tree level.

Parallelism at a higher than a Euclidean distance level. Khairy *et al.* [27] use GPUs to run in parallel multiple, low-dimensional (4×4) Sphere decoders but without parallelizing the tasks or the data processing involved in each. Jósza *et al.* [25] describe a hybrid *ad hoc* depth-first, breadth-first GPU implementation for low dimensional sphere decoders. However, their approach lacks theoretical basis and cannot prevent visiting unnecessary tree paths that are not likely to include the correct solution. In addition, since the authors do not propose a specific tree search methodology, their approach is not extendable to large MIMO systems. Yang *et al.* [52, 53] propose a VLSI/CMOS multicore combined distributed/shared memory approach for high-dimensional SDs, where SD partitioning is performed by splitting the SD tree into subtrees. But partitioning is heuristic, and their approach requires interaction between the parallel trees, thus making it inflexible. In addition, the required communication overhead among the parallel elements makes the approach inefficient for very dense constellations and inappropriate for a GPU implementation.

Other detection approaches. Local area search approaches [29,39,42] could also be used for the detection of large MIMO systems, but they are strictly sequential and require several iterations, resulting in increased latency. Lattice reduction techniques [15] are also prohibitive for large MIMO systems due to their sequential manner and high complexity ($\mathcal{O}(N_t^4)$).

7 Conclusion and Future Work

We have described FlexCore, a computationally-flexible method to consistently and massively parallelize the problem of detection in large MIMO systems, as well as similar maximum-likelihood detection problems. Our FlexCore GPU implementation in 12×12 64-QAM MIMO, enjoys a 19$\times$ computational speedup and 97% increased energy efficiency compared with the state-of-the-art. Furthermore, according to the best of our knowledge, our FlexCore's GPU implementation is the first able to support all LTE bandwidths and provide detection performance better than SIC, even for 12×12 MIMO systems. Finally, our implementation and exploration of FlexCore on FPGAs showed that for the same MIMO system, its energy efficiency surpasses that of the state-of-the-art by an order of magnitude. A promising next step is to extend FlexCore to "soft-detectors" as in [7,43].

8 Acknowledgments

This material is based upon work supported by the UK's Engineering and Physical Sciences Research Council (EPSRC Ref. EP/M029441/1), by the National Science Foundation under Grant No. 1617161 and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement No. 279976. The Authors would like to thank the members of University of Surrey 5GIC (http://www.surrey.ac.uk/5GIC) for their support. Finally, we thank our shepherd Lin Zhong and the anonymous reviewers for their insightful feedback.

References

[1] 3GPP LTE Encyclopedia: An Introduction to LTE, 2010.

[2] K. Amiri, C. Dick, R. Rao, J. R. Cavallaro. Flex-sphere: An FPGA configurable sort-free sphere detector for multi-user MIMO wireless systems. *Software Defined Radio Forum (SDR)*, 2008.

[3] D. Amos, A. Lesea, R. Richter. *FPGA-based prototyping methodology manual: best practices in design-for-prototyping*. Synopsys Press, 2011.

[4] L. Barbero, J. Thompson. Fixing the complexity of the sphere decoder for MIMO detection. *IEEE Transactions on Wireless Communications*, **7**(6), 2131–2142, 2008.

[5] L. G. Barbero, J. S. Thompson. FPGA design considerations in the implementation of a fixed-throughput sphere decoder for MIMO systems. *IEEE International Conference on Field Programmable Logic and Applications*, 1–6, 2006.

[6] J. R. Barry, E. A. Lee, D. G. Messerschmitt. *Digital Communication*. Springer Science & Business Media, 2004.

[7] J. Boutros, N. Gresset, L. Brunel, M. Fossorier. Soft-input soft-output lattice sphere decoder for linear channels. *IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 3, 1583–1587, 2003.

[8] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, H. Bolcskei. VLSI implementation of MIMO detection using the sphere decoding algorithm. *IEEE Journal of Solid-State Circuits*, **40**(7), 1566–1577, 2005.

[9] S. Chen, T. Zhang, Y. Xin. Relaxed K-best MIMO signal detector design and VLSI implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **15**(3), 328–337, 2007.

[10] T. Chen, H. Leib. GPU acceleration for fixed complexity sphere decoder in large MIMO uplink systems. *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 771–777, 2015.

[11] Chia-Hsiang Yang, D. Markovic. A flexible DSP architecture for MIMO sphere decoding. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **56**(10), 2301–2314, 2009.

[12] R. Courtland. Transistors could stop shrinking in 2021. *IEEE Spectrum*, **53**(9), 9–11, 2016.

[13] V. K. D Wübben, J Rinas, K. Kammeyer. Efficient algorithm for decoding layered space-time codes. *IEEE Electronics letters*, **37**(22), 1348–1350, 2001.

[14] E. Viterbo, and E. Biglieri. A universal decoding algorithm for lattice codes. *Proceedings of GRETSI*, 611–614. Juan-les-Pins, France, 1993.

[15] Y. H. Gan, C. Ling, W. H. Mow. Complex lattice reduction algorithm for low-complexity full-diversity MIMO detection. *IEEE Transactions on Signal Processing*, **57**(7), 2701–2710, 2009.

[16] G. Georgis, G. Lentaris, D. Reisis. Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. *Journal of Real-Time Image Processing*, 1–28, 2016.

[17] R. E. Guerra, N. Anand, C. Shepard, E. W. Knightly. Opportunistic channel estimation for implicit 802.11 af MU-MIMO. *Teletraffic Congress (ITC 28), 2016 28th International*, vol. 1, 60–68. IEEE, 2016.

[18] Z. Guo, P. Nilsson. Algorithm and implementation of the K-best sphere decoding for MIMO detection. *IEEE Journal on Selected Areas in Communications*, **24**(3), 491–503, 2006.

[19] B. Hassibi, H. Vikalo. On the sphere-decoding algorithm I. Expected complexity. *IEEE Transactions on Signal Processing*, **53**(8), 2806–2818, 2005.

[20] C. Hess, M. Wenk, A. Burg, P. Luethi, C. Studer, N. Felber, W. Fichtner. Reduced-complexity MIMO detector with close-to ML error rate performance. *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 200–203. ACM, 2007.

[21] IEEE Standard for Information Technology, part 11: wireless LAN Medium Access Control (MAC) and physical layer (PHY) specifications, amendment 4: enhancements for very high throughput for operation in bands below 6 GHz (IEEE Std 802.11ac-2013).

[22] IEEE Standard 802.11: wireless LAN medium access control and physical layer specifications, 2012.

[23] Intel's Skylake Core i7: a performance look, 2016. http://techgage.com/article/intels-skylake-core-i7-6700k-a-performance-look.

[24] J. Jaldén, B. Ottersten. On the complexity of sphere decoding in digital communications. *IEEE Transactions on Signal Processing*, **53**(4), 1474–1484, 2005.

[25] C. M. Józsa, G. Kolumbán, A. M. Vidal, F.-J. Martínez-Zaldívar, A. González. New parallel sphere detector algorithm providing high-throughput for optimal MIMO detection. *Procedia Computer Science*, **18**, 2432 – 2435, 2013.

[26] M. S. Khairy, M. M. Abdallah, S.-D. Habib. Efficient FPGA implementation of MIMO decoder for mobile WiMAX system. *IEEE International Conference on Communications*, 1–5, 2009.

[27] M. S. Khairy, C. Mehlführer, M. Rupp. Boosting sphere decoding speed through Graphic Processing

Units. *European Wireless Conference (EW)*, 99–104, 2010.

[28] Q. Li, Z. Wang. Improved K-best sphere decoding algorithms for MIMO systems. *IEEE International Symposium on Circuits and Systems*, 1159–1162, 2006.

[29] S. K. Mohammed, A. Chockalingam, B. S. Rajan. A low-complexity near-ML performance achieving algorithm for large MIMO detection. *IEEE International Symposium on Information Theory*, 2012–2016, 2008.

[30] S. Mondal, A. Eltawil, C. A. Shen, K. N. Salama. Design and implementation of a sort-free K-best sphere decoder. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **18**(10), 1497–1501, 2010.

[31] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable parallel programming with CUDA. *ACM Queue Magazine*, **6**(2), 40–53, 2008.

[32] K. Nikitopoulos, J. Zhou, B. Congdon, K. Jamieson. Geosphere: Consistently turning MIMO capacity into throughput. *Proceedings of the ACM SIGCOMM*, 631–642. ACM, 2014.

[33] T. Nyländen, J. Janhunen, O. SilvÃl'n, M. Juntti. A GPU implementation for two MIMO-OFDM detectors. *IEEE International Conference on Embedded Computer Systems (SAMOS)*, 293–300, 2010.

[34] C. Ramiro, A. M. Vidal, A. Gonzalez. MIMOPack: a high-performance computing library for MIMO communication systems. *The Journal of Supercomputing*, **71**(2), 751–760, 2014.

[35] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, A. M. Vidal. Fully parallel GPU implementation of a fixed-complexity soft-output MIMO detector. *IEEE Transactions on Vehicular Technology*, **61**(8), 3796–3800, 2012.

[36] M. Shabany, P. G. Gulak. Scalable VLSI architecture for K-best lattice decoders. *IEEE International Symposium on Circuits and Systems*, 940–943, 2008.

[37] M. Shabany, K. Su, P. G. Gulak. A pipelined scalable high-throughput implementation of a near-ML K-best complex lattice decoder. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 3173–3176, 2008.

[38] C. Shepard, H. Yu, N. Anand, L. Li, T. Marzetta, R. Yang, L. Zhong. Argos: Practical many-antenna base stations. *Proceedings of ACM Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[39] N. Srinidhi, S. K. Mohammed, A. Chockalingam, B. S. Rajan. Near-ML signal detection in large-dimension linear vector channels using reactive tabu search. *arXiv preprint arXiv:0911.4640*, 2009.

[40] D. Sui, Y. Li, J. Wang, P. Wang, B. Zhou. High throughput MIMO-OFDM detection with Graphics Processing Units. *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 2, 176–179, 2012.

[41] K. Tan, *et al.* SAM: Enabling practical spatial multiple access in wireless LAN. *Proceedings of ACM Conference on Mobile Computing and Networking (MobiCom)*, 2009.

[42] K. V. Vardhan, S. K. Mohammed, A. Chockalingam, B. S. Rajan. A low-complexity detector for large MIMO systems and multicarrier CDMA systems. *IEEE Journal on Selected Areas in Communications*, **26**(3), 473–485, 2008.

[43] H. Vikalo, B. Hassibi, T. Kailath. Iterative decoding for MIMO channels via modified sphere decoding. *IEEE Transactions on Wireless Communications*.

[44] E. Viterbo, J. Boutros. A universal lattice code decoder for fading channels. *IEEE Transactions on Information Theory*, **45**(5), 1639–1642, 1999.

[45] M. Wenk, L. Bruderer, A. Burg, C. Studer. Area-and throughput-optimized VLSI architecture of sphere decoding. *IEEE/IFIP 18th VLSI System on Chip Conference (VLSI-SoC)*, 189–194, 2010.

[46] M. Wenk, M. Zellweger, A. Burg, N. Felber, W. Fichtner. K-best MIMO detection VLSI architectures achieving up to 424 Mbps. *IEEE International Symposium on Circuits and Systems*, 4 pp.–1154, 2006.

[47] P. W. Wolniansky, G. J. Foschini, G. Golden, R. A. Valenzuela. V-BLAST: An architecture for realizing very high data rates over the rich-scattering wireless channel. *IEEE URSI International Symposium on Signals, Systems, and Electronics (ISSSE)*, 295–300, 1998.

[48] K. wai Wong, C. ying Tsui, R. S. K. Cheng, W. ho Mow. A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, III–273–III–276, 2002.

[49] B. Wu, G. Masera. A novel VLSI architecture of fixed-complexity sphere decoder. *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, 737–744, 2010.

[50] M. Wu, S. Gupta, Y. Sun, J. R. Cavallaro. A GPU implementation of a real-time MIMO detector. *IEEE Workshop on Signal Processing Systems*, 303–308, 2009.

[51] M. Wu, Y. Sun, S. Gupta, J. R. Cavallaro. Implementation of a high throughput Soft MIMO

Detector on GPU. *Journal of Signal Processing Systems*, **64**(1), 123–136, 2011.

[52] C. H. Yang, D. Marković. A 2.89mW 50GOPS 16x16 16-core MIMO sphere decoder in 90nm CMOS. *IEEE European Solid-State Circuits Conference (ESSCIRC)*, 344–347, 2009.

[53] C. H. Yang, T. H. Yu, D. Marković. A 5.8mW 3GPP-LTE compliant 8x8 MIMO sphere decoder chip with soft-outputs. *IEEE Symposium on VLSI Circuits (VLSIC)*, 209–210, 2010.

[54] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, Y. Zhang. Bigstation: enabling scalable real-time signal processing in large MU-MIMO systems. *ACM SIGCOMM Computer Communication Review*, **43**(4), 399–410, 2013.

A Position Vector Error Probability Approximation

For the top sphere decoding tree layer ($l = N_t$), the probability of the first closest symbol to the effective received point $\tilde{y}(N_t)$ not to be the transmitted symbol is equivalent to the corresponding symbol error rate over an AWGN channel, or [6]

$$P_e(N_t) = \left(2 + \frac{2}{\sqrt{|Q|}}\right) \cdot \text{erfc}\left(\frac{R(N_t,N_t)\cdot\sqrt{Es}}{\sigma}\right), \quad (6)$$

Then, the probability of the first closest symbol to the received to be the transmitted one is $P_{N_t}(1) = 1 - P_e(N_t)$. Calculating the probability for the kth (with $k > 1$) closest to the received symbol to be the one transmitted would require real-time two-dimensional integrations since an analytical solution is infeasible. Instead, we approximate the problem based on the observation that the inter-symbol distance in QAM constellations scales nearly in a square-root manner, as a function of the position index k related to the received signal.

Then we make the approximation that the decision boundaries (D_k) would scale in a similar manner. That is

$$D_k = \sqrt{c \cdot k}, \quad (7)$$

where c is a positive and real constant. Then,

$$P_{N_t}(k) = P\left(D_{k-1} < |n_{N_t}| \leq D_k\right)$$
$$= P\left(|n_{N_t}| \leq \sqrt{c\cdot(k)}\right) - P\left(|n_{N_t}| \leq \sqrt{c\cdot(k-1)}\right). \quad (8)$$

Since the amplitude of the noise sample (n_{N_t}) is Rayleigh distributed

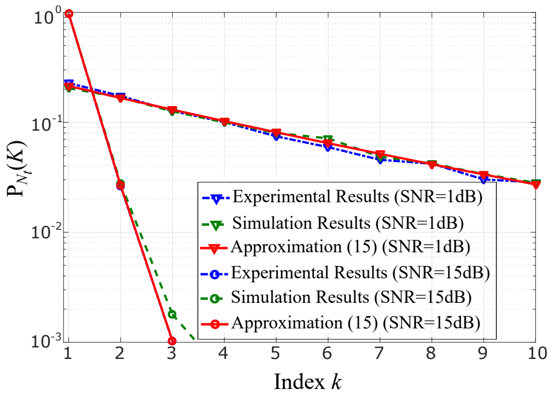

Figure 14— Comparison between (11) (solid), simulation (dashed, Gaussian noise) results for $P_{N_t}(k)$, and experimental results (dashed-dotted, WARP platform) for the probabilities $P_{N_t}(k)$ at various SNRs.

$$P_{N_t}(k) = \exp\left(-\frac{c\cdot(k-1)}{\sigma^2_{n_t}}\right) - \exp\left(-\frac{c\cdot(k)}{\sigma^2_{n_t}}\right)$$
$$= \exp\left(-\frac{c\cdot(k-1)}{\sigma^2_{n_t}}\right)\cdot\left[1 - \exp\left(-\frac{c}{\sigma^2_{n_t}}\right)\right] \quad (9)$$

Applying the above for $k = 1$ is $1 - P_e(N_t)$, with $P_e(N_t)$ defined in (6), therefore, for both equations to hold,

$$P_e(N_t) = \exp\left(-\frac{c}{\sigma^2_l}\right). \quad (10)$$

Accordingly, the probability that the kth closest constellation point to the observable of the top level ($l = N_t$) is the transmitted one can be expressed as

$$P_{N_t}(k) = (1 - P_e(N_t)) \cdot (P_e(N_t))^{(k-1)}. \quad (11)$$

Fig. 14 compares the theoretical estimates of the "per-level" probabilities P_{N_t} to the ones obtained by simulations as well as to the ones obtained by actual experiments using our WARP v3 platform implementation (described in Section 5.1). It shows that our theoretical model is very accurate in all SNR regimes.

It can be easily shown that, the above equation does hold for any sphere decoding tree level, given that all the higher layers include the correct solutions (the correct transmitted vector). This is because the effect of the correct solution can be easily removed in terms of successive interference cancellations. As as result, the probability P_c can be calculated as in equations (2), (3) and (4) (Sec. 3).