POSTPRINT VERSION. The final version is published here :
Sirois, S., Ahmed Ouameur, M. et Massicotte, D. (2022). High level synthesis strategies for ultra fast and low latency matrix inversion implementation for massive MIMO processing. Integration (Vol. 82, p. 29-36).
doi: https://doi.org/10.1016/j.vlsi.2021.08.011

High Level Synthesis Strategies for Ultra Fast and Low Latency Matrix Inversion Implementation for Massive MIMO Processing

Samuel Sirois, Messaoud Ahmed Ouameur and Daniel Massicotte

Université du Québec à Trois-Rivières, Department of Electrical and Computer Engineering,
3351, Boul. des Forges, Trois-Rivières, Québec, Canada
Laboratoire des Signaux et Systèmes Intégrés
Chaire de recherche sur les signaux et l'intelligence des systèmes haute performance

Abstract—**In this paper, we discuss the implementation strategies of an explicit matrix inversion technique based on the recursive Gram matrix inversion update (RGMIU) algorithm. These strategies are explicitly chosen to optimise the design for high throughput dictated by the enhanced mobile broad band (eMBB) use case and by the low latency imposed by the ultra-reliable low-latency communications (URLLCs) use case. The RGMIU algorithm is recently proposed to implement the zero forcing (ZF) problem encountered in massive multiple-input multiple-output (MIMO) detection task. We therefore compare and analyse the performance in terms of symbol error rate (SER) against popular implicit and explicit methods such as optimised coordinate descent (OCD), Gauss-Seidel (GS) and Neumann series expansion (NSE) algorithms. To determine the optimal word length, a fixed-point analysis shows that 16 bits are enough to reach a floating-point precision. We, thereafter, use Vivado High-Level Synthesis tools and optimization directives to implement the RGMIU algorithm based on three strategies to infer key insights and design trade-offs from the resource's utilization, latency, throughput and energy efficiency.**

Index Terms— **Massive MIMO, Recursive Gram Matrix Inversion Update (RGMIU), MIMO detection, Zero Forcing (ZF), Fixed-point analysis, high-level synthesis, Vivado HLS, throughput, latency.**

I. Introduction

Being a promising concept for future cellular networks, massive multiple-input multiple-output (MIMO) has now made its way to 5G as one of the means to substantially improve both spectral and energy efficiencies [1]. As a matter of fact, base stations (BSs) with 64 fully digital transceiver chains are commercially deployed and the key component of massive MIMO has made its way into the 5G standard [2], [3]. Nevertheless, the authors in [4] have pointed out that massive MIMO implementation continues to be at least as exciting as massive MIMO theory. Massive MIMO is a form of multiuser MIMO where the number of serving antennas at the BS is an order of magnitude larger than the number of user terminals (UTs) served within each radio resource element. Given the large number of antennas, reliance on time division duplex (TDD) channel reciprocity is essential [1].

Under favorable channel conditions and/or as the number of antennas increase, the UTs' channels are mutually orthogonal which makes linear processing (detection and precoding), such as maximum ratio combining (MRC), zero forcing (ZF) and minimum mean square error (MMSE) detection techniques, optimal [5]. The detection/precoding problem based on ZF or MMSE technique is an arithmetic operation with cubic computational complexity in the order of the matrix dimension. To reduce the implementation complexity, matrix inversion approximations such as Neumann series expansion (NSE) is proposed in [6]. Recently, a technique based on Gauss-Seidel (GS) was shown to outperform NSE due to its fast convergence at considerably low computational complexity [7]. However, this comes at the expense of higher latency and lower throughput [7]. It has actually been shown that the NSE performance degrades as the number of UTs increases [8]. To counter the load increase effect, GS can still afford using more iterations while maintaining lower computational complexity, albeit at the expense of reduced throughput [7]. It has therefore been argued to resort to exact matrix inversion [8].

High throughput application-specific integrated circuit (ASIC) is designed for the NSE based detector in [9]. The ASIC achieves 3.8 Gbps for 128 antenna BS and 8 users for single carrier frequency division multiple access (SCFDMA). The NSE based detector is also implemented on a Xilinx Virtex-7 FPGA in [6]. The FPGA design achieves 600 Mbps for 128 antenna BS serving 8 UTs (i.e. 128×8 system). A detector based on the conjugate gradient (CG) algorithm has been proposed in [10] and achieved a significant complexity reduction. However, to speed up the convergence rate and improved performance, a hybrid detector based on the CG algorithm and the Jacobi method has been proposed in [11]. The CG-based detector is also implemented in Xilinx Virtex-7 FPGA for a 128×8 system [12]. On the other hand, the GS-based method has been proposed wherein the initial solution is based on the NSE of two terms [7] whereas, its parallel architecture is implemented in [13]. Even though the GS method can reduce the complexity to be $O(K^2)$ [14], however, due to the GS internal sequential iteration dependencies, it is not well suited for parallel implementation [7], [15].

On the other hand, it has also been argued that these centralized processing techniques still impose stringent constraints on the interconnects bandwidth between the massive MIMO radio heads (RHs) and the central processing unit (CPU). Distributed, or decentralized, massive MIMO processing has been introduced to overcome such limitations [16] and [17]. Unfortunately, the decentralized processing

computational complexity, and hence the energy efficiency, are also of concern [18]. On the other hand, to support ultra-reliable low-latency communications (URLLCs), low latency and high-throughput processing is required. As such, we focus on implementing the core matrix inversion technique based on recursive Gram matrix inversion update (RGMIU) method [19].The inversion of the Gram matrix is performed by exploiting matrix inversion update of a matrix in the form of $\mathbf{H}^H\mathbf{H}$ when a new column is added/updated to a complex-valued matrix $\mathbf{H}$ [19]. Even though the implementation of the RGMIU technique has been presented in [19], the contribution of this paper is the way the algorithm is implemented. In fact, the RGMIU method is completely unrolled to leverage a maximum throughput performance. We can add that the implementation in [19] was designed for low resources consumption. Also, the order of magnitude of the obtained results can justify the importance of this paper as it will be shown in the next sections. Herein, direct matrix inversion based on Cholesky decomposition is considered as a reference to the performance and computational complexity standpoint.

A comprehensive review, comparison and discussion of the existing linear precoding mechanisms for massive MIMO according to different cell scenarios have been presented in [8]. It also discussed some standing challenges which related to the design of precoding mechanisms and practical implementations. Low complexity precoders suffered from a considerable performance loss, while a complicated precoder design is more difficult to implement practically. On the other hand, an extensive survey on detection algorithms related to massive MIMO systems is presented in [20]. The particular focus is on performance and complexity trade-off as well as the practical implementation of detection algorithms.

In this paper,

- We simulate and compare performances in terms of symbol error rate (SER) for four popular algorithms, namely the Gauss-Seidel (GS) implicit method, the optimized coordinate descent (OCD) implicit method with BOX equalization, the Neumann series expansion (NSE) of order 3 explicit approximation method and the RGMIU explicit method. The goal is to introduce the reader to other state-of-the-art algorithms and to show how the RGMIU performs relatively to these algorithms. This is an extension to [19] by adding the OCD method to make the paper self-contained including fixed point simulations.
- The main focus of this work will be on the RGMIU algorithm implementation which is not considered in [19]. We use Vivado High-Level Synthesis tools and optimization directives to implement the RGMIU algorithm based on three strategies to infer key insights and design trade-offs from the resource's utilization, latency, throughput and energy efficiency.

The paper is organized as follows: Section II summarizes the detection problem. Section III presents the simulation results and discusses the performances of the four methods. Section IV shows the implementation results on Vivado HLS in terms of design strategies, fixed-point analysis, resources consumption, latency/throughput and energy efficiency for the RGMIU algorithm. Finally, a conclusion is drawn.

II. Signal Model

Considering a massive MIMO system with K UTs and M BS antennas, the received signals at the BS can be modeled as

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{n}\,, \tag{1}$$

where $\mathbf{y} \in \mathbb{C}^{M\times 1}$ is a vector containing the signals received by each BS antenna, $\mathbf{H} = \begin{bmatrix}\mathbf{h}_1 & \mathbf{h}_2 & \cdots & \mathbf{h}_K\end{bmatrix}$ is the matrix channel for a given sub-carrier with every column representing the channel between every antenna and one particular UT so that $\mathbf{H} \in \mathbb{C}^{M\times K}$, $\mathbf{s} = \begin{bmatrix}s_1 & s_2 & \cdots & s_K\end{bmatrix}^T$ is the vector containing the K symbols sent by the K UTs and $\mathbf{n} \in \mathbb{C}^{M\times 1}$ is a complex gaussian white noise with zero mean and variance σ^2. The Gram matrix and the matched filter vector resulting from the projection of the vector $\mathbf{y}$ onto the column space of $\mathbf{H}$ are computed as follows

$$\mathbf{G} = \mathbf{H}^H\mathbf{H}\,, \tag{2.a}$$

$$\mathbf{y}_{MF} = \mathbf{H}^H\mathbf{y}\,, \tag{2.b}$$

where $\mathbf{G}$ is the Gram matrix and $\mathbf{y}_{MF}$ is the matched filter vector. Therefore, the new square system of equations can be written as

$$\mathbf{y}_{MF} = \mathbf{G}\mathbf{s} + \mathbf{H}^H\mathbf{n}\,. \tag{3}$$

It is possible to explicitly solve equation (3) as follows

$$\hat{\mathbf{s}} = \left(\mathbf{G}\right)^{-1}\mathbf{y}_{MF} \tag{4}$$

where $\hat{\mathbf{s}}$ is the vector containing the estimated value of the symbols sent by the UTs. In fact, hard or soft data output decision needs to be done on $\hat{\mathbf{s}}$ depending on if forward error correction process is used or not to ensure that the estimated symbols are contained within the same set (constellation) as the one of $\mathbf{s}$. For simplicity, our analysis will be restricted to $\hat{\mathbf{s}}$ by computing the symbol error rate (SER) on hard output decision which consists of making a decision based on the symbol in the constellation set that is the closest to its estimated value $\hat{\mathbf{s}}$ in terms of Euclidian distances.

It is also possible to implicitly solve it by using iterative methods as discussed in the first section. Nevertheless, the procedure that consists of solving equation (3) is known as a zero forcing (ZF) solution. By adding the variance σ^2 to every diagonal entry of the Gram matrix and solving the same system, we could have derived a more robust technique called minimum mean square error (MMSE) problem but since it is often hard to get the value of σ^2 in practice, we chose the simplicity and use ZF algorithm. Finally, hard output decision can be written as

$$\hat{s}_i^{hard} = \arg\min_{z\in \mathrm{O}} \left|\hat{s}_i - z\right| \quad i = 1,\ldots,K \tag{5}$$

where O is the constellation set. With this metric in hand, it is possible to process the SER by computing the ratio of the wrong decisions over the total number of symbols sent. As the

main focus is to implement the RGMIU algorithm, Table 1 outline such technique for the sake of safe contained reference where the details are found in [19]. To give the reader a better intuition behind this method, the RGMIU algorithm is said to be recursive in the sense that the inverse of the Gram Matrix is recursively built from a smaller Gram matrix inverse that is growing every iteration. This growing matrix corresponds to $\mathbf{B}$ in table 1. Each iteration of the RGMIU algorithm adds a row and a column to $\mathbf{B}$ up until the full inverse of the Gram matrix is obtained. That way, the starting point of the algorithm could be an already known inverse of a one dimension smaller Gram matrix and only one iteration would be required to find the new inverse. This is typically the case when the channel matrix does not change and a new UT is added to the group. This behavior is opposite to most classical iterative algorithms that need to redo a set of iterations all over again when a new UT is considered.

TABLE 1. THE RECURSIVE GRAM MATRIX INVERSION UPDATE (RGMIU) ALGORITHM

INPUT: $\mathbf{G}=\mathbf{H}^H\mathbf{H}$		
Precompute the scalar inverse as a starting point:		
$\mathbf{B}_1=\left(\mathbf{G}_{1,1}\right)^{-1}$	// $\mathbf{B}_1$ is an 1×1 matrix (scalar)	
FOR $m=1$ **to** $K-1$ **DO:**		
1.	$\mathbf{z}=\mathbf{G}_{m+1,m+1}$	// $\mathbf{G}_{m+1,m+1}$ is the element at row and column $m+1$
2.	$\mathbf{y}_1=\mathbf{G}_{1:m,m+1}$	// $\mathbf{G}_{1:m,m+1}$ corresponds to rows 1 to m in column $m+1$
3.	$\mathbf{y}_2=\mathbf{B}_m\mathbf{y}_1$	
4.	$c=1/\left(\mathbf{z}-\mathbf{y}_1^H\mathbf{y}_2\right)$	
5.	$\mathbf{y}_3=c\,\mathbf{y}_2$	
6.	$\mathbf{\Gamma}=\mathbf{B}_m+c\,\mathbf{y}_2\mathbf{y}_2^H$	
7.	$\mathbf{B}_{m+1}=\begin{bmatrix}\mathbf{\Gamma} & -\mathbf{y}_3\\ -\mathbf{y}_3^H & c\end{bmatrix}$	// $\mathbf{B}$ is a growing matrix
END DO		
OUTPUT: $\mathbf{B}_K=\left(\mathbf{H}^H\mathbf{H}\right)^{-1}$		

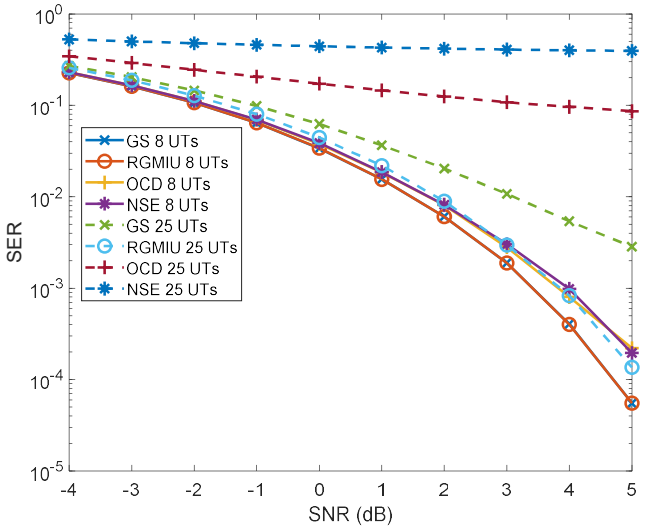
III. SIMULATION RESULT

Our simulations were done with 128 BS antennas and 8 or 25 UTs. The channel matrix is modeled with independent and identically distributed (i.i.d.) random variable with zero mean and unit variance and we used a 64 QAM constellation for the symbols with an average transmitted power of 1.

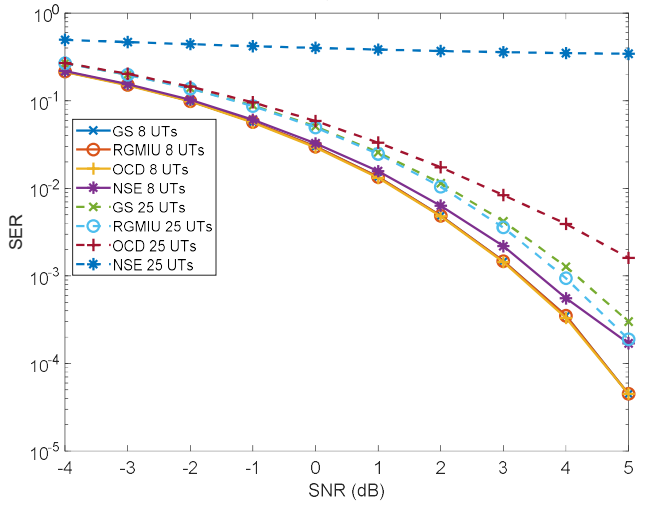
Before, looking at the simulation results, we can predict some behaviors of these algorithms. Firstly, since the GS and the RGMIU are both solving the ZF problem, we can guess that with enough iterations, the GS algorithm will reach the same performances as the RGMIU method. Also, as mentioned in [19], the OCD algorithm has near MMSE performances without having access to the value of σ^2. That being said, we should expect the OCD to have the smallest SER after a certain number of iterations.

Fig. 1 shows the SER for the four algorithms with the GS and OCD having 2 and 3 iterations respectively for 8 and 25 UTs. As expected, for 8 UTs, it takes at least two iterations for the GS algorithm to reach the same performances as the RGMIU algorithm. It also takes three iterations for the OCD method to get the same results as RGMIU and GS. The reason why OCD does not outperform the other algorithms is because the channel matrix $\mathbf{H}$ is not correlated. That way, the diversity at the BS is maximized and the system of equations is easy to solve with a good accuracy. Our goal was simply to get a good idea on how these algorithms perform relatively to one another. On the other hand, the order 3 NSE algorithm never reaches the performances of the exact matrix inversion and has a catastrophic SER with 25 UTs.

Of particular interest is the fact that for 25 UTs, OCD and GS need more iterations to reach the performances of the RGMIU. In Fig. 1b, OCD and GS do not reach the SER of RGMIU with three iterations. In general, the number of iterations for implicit methods will depend on the ratio of the number of BS antennas over the number of UTs which is a disadvantage because the complexity of the algorithm is hard to predict. In contrast, the



a)



b)

Fig. 1 SER versus SNR of the GS, NSE, OCD and RGMIU with a) two iterations and b) three iterations for GS and OCD.

complexity of the RGMIU can be known in advance because it takes as many iterations as the number of UTs minus one to inverse the Gram matrix. Also, the results of the RGMIU will always be accurate because its output is the exact Gram matrix inverse. On the other hand, the order 3 NSE will be accurate only when the ratio of the number BS antennas over the number UTs is high.

In the end, all the presented techniques solve for the ZF detector. Some methods approximate the Gram matrix inverse and others implicitly solve the system of equations. The results tend to show that approximation methods may not solve the ZF problem with good accuracy and that implicit iterative methods performances can vary depending on the number of iterations applied in the process. The over all goal here is to show that direct matrix inversion should be chosen to obtain the best trade-off in terms of performances, complexity and reliability.

From another point of view, the implementation trade-off between algorithms is mainly about the results in terms of throughput versus resource consumption. A computational intensive algorithm can be implemented with low resources consumption but the throughput results will generally be very low. The opposite can also be true. In fact, it is shown in [19] that the RGMIU method is in the midfield in terms of computational complexity, being below the NSE technique and above the GS algorithm.

In this sense, RGMIU is partly chosen for its low inter-iteration dependencies which generally reduce resource consumption regarding the algorithm complexity. In [8], implementations of the CG and the GS can only achieve the throughput of 20 Mb/s and 48 Mb/s respectively as opposed to the Neumann series expansion technique that can reach a theoretical throughput of 621 Mb/s. In all these three algorithms, the preprocessing steps corresponding to compute the Gram matrix is part of the implementation. This shows that the bottleneck of the design in terms of throughput is the core algorithm itself and not the preprocessing steps. On the other hand, the resource consumption on the chosen FPGA for the NSE implementation uses 34% of LUTs, 19% of FFs and 28% of DSP48Es as opposed to the two other methods that barely use above 5% of each of these hardware components.

On the other hand, as it is shown in [7], another GS implementation could reach the throughput of 732 Mb/s but at the expense of higher and similar resources consumption as the NSE algorithm whereas, by fully optimizing the RGMIU algorithm to reach excellent throughput performances, the resources consumption for 8 mobile users are more than acceptable (will be shown Section IV, Table 2) with potential throughput results above what is normally expected. The only problem that remains is the implementation of a preprocessing and a post-processing unit that will not act as the bottleneck to the design.

IV. Implementation

We used Vivado HLS 2019.1 to implement our designs [21]. All solutions were validated with RTL Cosimulation and we used the xc7vx690t from the Virtex 7 family as the reference FPGA.

A. Implementation strategies

Before directly talking about design strategies, it is important to mention here that we have only implemented the core of the RGMIU algorithm. In other words, the Gram matrix and matched filter vector preprocessing phase is not implemented as well as the post-processing multiplication of the matrix inverse with the matched filter vector. We only wanted to optimize the core algorithm because as we will see later, it has the capabilities to reach huge throughput data detection, but the preprocessing and the post-processing units cannot keep up with the data rate at which the core can operate without a big increase in resources utilization on FPGA. Our goal is to show the potential of the core algorithm. On the other hand, the complete implementation of the RGMIU algorithm on an ASIC, which can reach higher clock frequencies than FPGA, could be realistic in the sense that the preprocessing and post-processing phases could operate at a higher clock frequency than the core algorithm in order to use it at its full capacity.

Three implementation strategies were used [22]. The first one, which we refer as S1, uses a different hardware for every iteration. It is important to note here that the RGMIU algorithm has a growing vector and matrix sizes every iteration so that loops boundaries in the code are variable depending on the current iteration. With that in mind, we found out that the best way to translate this to Vivado HLS was to create template functions in C++ with the template parameter being the loop boundaries so that every time a template function is called with a new template parameter, Vivado HLS understands that it needs to create new hardware for this function call. Every function was written in such a way that it could be pipelined with an initiation interval of one. Without the function template, we would have needed to declare as many functions as there are iterations times the number of steps in one iteration with the only difference being the loop boundaries parameter values. Some may argue that we could have used the FUNCTION_INSTANTIATE pragma instead which is equivalent in some way to the template functions, but we found out that in our design, this directive was interfering with the PIPELINE pragma. We then created a top function which called all the template functions one after the other [22]. This top function was pipelined, and the template functions were inlined to increase the flow of data and to maximize the performances. In total, there were five template functions corresponding to equations (3) to (7) in Table 1 and the value of the template parameter went from 1 to $K-1$. Also, since the loop boundaries are different for every iteration in Table 1, the hardware resources used are also different for every iteration because most of the loops are unrolled by the pipeline directive in the top function. Fig. 2 shows the architecture of the first implementation strategy.

The second strategy (S2) is basically the same as the first strategy except that the design is heavily pipelined with more registers so that the clock signal can reach higher frequencies which in turn enables a higher potential throughput. The downsides of this are the increases in resources consumption and a higher latency. To translate this to Vivado HLS, we just reused the first strategy solution and we set a clock constrain with a smaller period. Even if it is straightforward to pass from

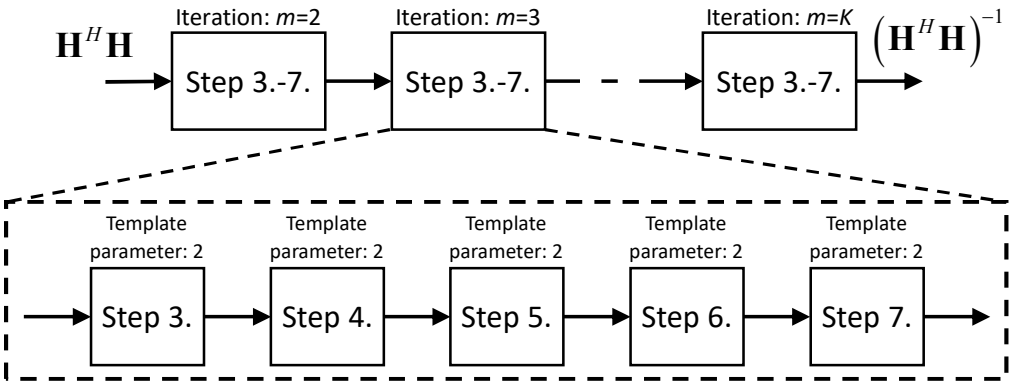


Fig. 2 RGMIU implementation flat architecture

S1 to S2, the authors judge that the results are still worth mentioning since the maximum clock frequency can be significantly increased. Otherwise, the general architecture of S2 is the same as the one of S1 depicted in Fig. 2.

The third strategy (S3) is designed in such a way that resources consumption is similar in terms of percentage for the DSP48E, the flip-flops (FF) and the look-up tables (LUT). The downside of this method, as we will see in the Section IV.D, is that the throughput is generally cut in half. To deploy this strategy, we used the ALLOCATION pragma to limit the number of DSP48E in the top function, so that FFs and LUTs are needed to compensate this limitation. The fact that S3 uses less DSP48E can give more room to other implementations that need explicitly this type of resources to reach good performances in terms of throughput on the same FPGA. That is not the case with S1 and S2. Also, S3 is the only possible implementation with 12 users. Once again, the general architecture of S3 is the same as the one of S1 depicted in Fig. 2.

To sum up in more details, the power of the RGMIU algorithm resides in the fact that it does not contain major sequential inter-iteration dependencies, as opposed, for example, to the GS method [7]. This allows implementing this method with a small pipeline initiation interval without excessive usage of hardware resources. The smaller the initiation interval, the higher the throughput is, since an initiation interval of one means a new valid input can be processed every new clock cycle. Also, as stated before, the number of iterations that the algorithm needs to go through to obtain the exact inverse of the Gram matrix is equal to the number of mobile users minus one. These simple facts greatly simplify the complexity of the hardware implementation under Vivado HLS.

Each equation from equations (3) to (7) corresponding to one iteration in table 1 can be described as a simple function in C++ with a template parameter used to indicate the current iteration. Once this is done, each array that is contained in these functions are partitioned with the ARRAY_PARTITION pragma. Then, the whole function is pipelined with the PIPELINE pragma and inlined with the INLINE pragma. The combination of partitioning and pipelining allows Vivado HLS to effectively pipeline every loop inside a given function to maximize its throughput while inlining prevents the creation of bottle necks between each function call. Then, a main top function is created to call all functions corresponding to one iteration of the RGMIU algorithm a number of times equal to the number of mobile users minus one. This main function is also pipelined to ensure smooth data transfers between function calls. In other words, the main idea of the RGMIU method implementation on Vivado HLS was to first subdivide the algorithm in simple elements and optimized them as much as possible to then glue them together in an efficient way to avoid bottle necks. At this stage the only thing missing is the input/output management. Basically, the input of the design is a FIFO corresponding to every element of the Gram matrix in parallel and the output is corresponding, in a similar manner, to every element of the inverse of the GRAM matrix in parallel. The INTERFACE pragma was used to achieve this.

The key idea behind an efficient implementation of RGMIU is to correctly subdivide the algorithm and add simple optimization pragmas from the bottom-up. This is where the power of Vivado HLS resides. It is therefore worth reminding that the implementation results are the main contribution of this paper, even if the path to obtain them remains quite intuitive but not trivial.

The optimization strategies were also quite intuitive to implement once the algorithm is well subdivided into small function. The first strategy was simply obtained by letting Vivado HLS freely set the clock constrains and the resources allocation. Then, from this first baseline strategy, the second strategy design was obtained by specifying a smaller target clock in the project solution settings. Finally, the third strategy was derived from the first strategy by using the ALLOCATION pragma in the main top function to limit the number of DSP48Es to a maximum value.

That being said, we also tried to implement a fourth strategy which consisted of reusing the same hardware for the last two iterations. The goal was to dramatically reduce the resources consumption since the DSP48E utilization is a cubic function with respect to the number of iterations as we will see in the Section IV.E. When we implemented it, there was indeed a big improvement in the resources consumption, but the throughput was getting too slow so that we have decided to abandon the idea.

Before continuing to the next subsection, one thing we did not try but that could have been interesting is a pipeline interleaving strategy as they did in [8]. Indeed, since the algorithm loop depicted in Table 1 has inter-iteration dependencies, this prevents traditional pipelining except if we completely unroll it (different hardware for every iteration) as we did in S1, S2 and S3. To overcome this problem without a complete unroll, it would have been possible to always reuse the same hardware that represents one iteration and to create several pipeline stages inside it so that the core can accept a new Gram matrix input every clock cycle up until all the pipeline stages are filled. Thereafter, the algorithm would need to finish its $K-1$ iterations before being able to accept a new batch of inputs. Also, it would have been possible to create a hardware that computes more than one iteration to be able to have more pipeline stages in the design and a higher global throughput at the expense of higher resources utilization. This pipeline interleaving strategy is reserved for future work.

B. *Fixed-point analysis*

The fixed-point analysis was done with 16 bits word lengths. Most of the internal variables used an integer part of one signed bit and 15 fractional bits. Fortunately, no shifts were needed to reduce the dynamic range and the common operators in Vivado HLS were able to automatically align the binary point of two variables having different fractional bit width. That being said, since DSP48Es in FPGAs can accept a maximum bit width of 18, our design was not panelized in terms of resources consumption. For all complex value multiplications, the complex multiplier is composed of 4 real multipliers and 2 real adders.

We have been able to reach a good precision such that the difference between the SER curve of floating and fixed-point never exceed 0.5 dB. Indeed, Fig. 3 shows the comparison in terms of SER for floating and fixed-point variables with a bit width of 15 and 16. Clearly, 16 bits is the minimum required bit width to obtain results below 0.5 dB.

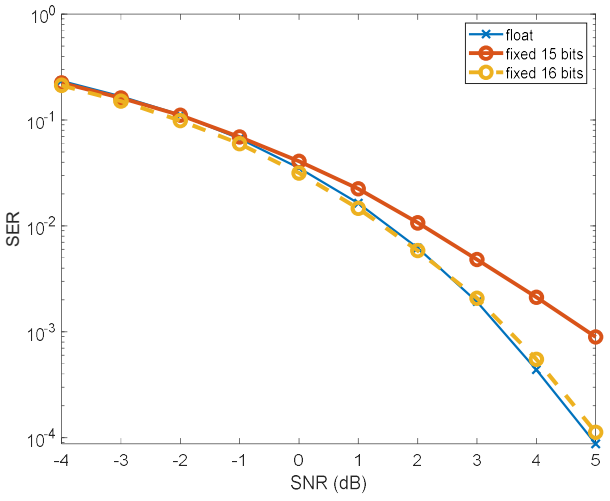


Fig. 3 SER of the RGMIU algorithm for 8 UTs with floating and fixed-point precision.

C. *Resources utilization estimates*

The resources utilization for all strategies in terms of DSP48Es, FFs and LUTs is depicted in Table 2 for different numbers of UTs. Looking only at S1 and S2, the number of DSP48Es used is independent of the employed strategy. This is totally normal because these are used for multiplication and multiplier-accumulate (MAC) operations which are independent of the level of pipelining. The DSP48Es are only dependent on the number of UTs. This being said, we can derive a formula that connects these two parameters together. By looking at the scheduler of Vivado HLS, we have determined that the required number of DSP48Es, namely $d(l)$, for the l^{th} iteration is defined as follow

$$d(l) = 8l^2 + 4l - 2 \tag{6}$$

With that in mind, the total number of DSP48Es, namely D, is defined by

$$D = \sum_{l=1}^{L} d(\mathrm{l}) = \frac{16L^3 + 36L^2 + 8L}{6} \tag{7}$$

where L is the total number of iterations, which corresponds to the number of UTs minus one. Equation (7) shows that the number DSP48Es follows a cubic order which can be a limiting factor for bigger designs with more than 8 UTs. This is why we have implemented S3 to repartition the resources in such a way that we are not limited by the number of DSP48Es for 12 UTs. The resources consumption of S3 in terms of LUTs and FFs is way higher than S1, but it is much less in terms of DSP48Es. This comes from the fact that Vivado HLS used LUTs and FFs to create the same logic response as DSP48Es. As it will be seen in the next subsection, the throughput in S3 is smaller because DSP48Es are designed and optimized to accept a high clock frequency. When the algorithm can use all the DSP48E it needs, the overall throughput can be maximized in contrast as when it cannot. The obtained results of S3 are worth mentioning to show the flexibility of the implementation. On the other hand, as we could expect, S2 utilizes more FFs and LUTs than S1 because it has more pipeline stages. Finally, as mentioned in the implementation strategies Section IV.E, S3 uses resources in a balanced way. In addition to allowing us to implement a design with 12 UTs, it significantly reduces the number of DSP48Es for 8 UTs by a factor of 13%.

D. *Latency and throughput estimates*

Referring to the results obtained from Vivado HLS post synthesis estimations in Table 2, the latency between S1 and S2 are similar but the highest throughputs are achieved by S2. Fig. 4 compares the latency and throughput for S1, S2 and S3 in a 64 QAM modulation. For 8 UTs, we are talking about a potential latency of 0.966 µs and a potential throughput of 18.18 Gb/s with S1. These numbers are conditional to the fact that the preprocessing and the post-processing units can keep up with this data rate. In a similar fashion, for 8 UTs with S2, it is theoretically possible to reach the latency of 1.04 µs and the throughput of 24.82 Gb/s. It is interesting to note that since the implementations of all strategies are pipelined with a unitary input interval, the minimum clock period estimated by Vivado HLS in Table 2 also corresponds to the Gram matrix inversion throughput period. That way, the estimated

TABLE 2. ESTIMATED RESOURCES, LATENCY AND THROUGHPUT

Strategy	Minimum clock period (ns)	Number of UTs	Estimated latency (µs)	Estimated Throughput (Gb/s)	DSP48Es	LUTs	FFs
S1	2.64 ± 0.38	4	0.428	9.09	130 (3%)	12192 (2%)	18065 (2%)
		8	0.966	18.18	1218 (33%)	60216 (13%)	95177 (10%)
S2	1.934 ± 0.13	4	0.404	**12.41**	130 (3%)	14726 (3%)	32829 (3%)
		8	1.039	**24.82**	1218 (33%)	84282 (19%)	210569 (24%)
S3	2.577 ± 0.38	8	0.905	9.31	750 (20%)	84098 (19%)	161753 (18%)
		12	1.495	13.97	2400 (66%)	289930 (66%)	534689 (61%)

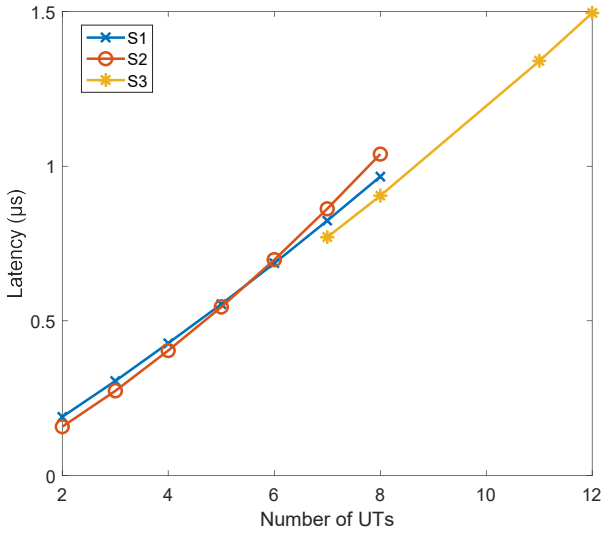


Fig. 4 Latency versus the number of UTs for implementation strategies S1, S2 and S3.

throughput in Gb/s can be calculated by multiplying the number of UTs with the number of bits in the constellation (6 bits for 64 QAM) and by dividing the result with the estimated clock period. To continue, both throughput and latency are linear with the number of UTs. This is totally normal for the throughput because the number of UTs is the only variable multiplicative factor. Since the algorithm is completely unrolled (All loops flatten), the limiting factor for the clock frequency is the level of pipelining which is independent of the number of users. The only thing that changes is the resources consumption and the over all latency since 7 iterations are flatten with 8 UTs in contrast of only 3 with 4 UTs. On the other hand, by looking at the design scheduler in Vivado HLS with S1 for example, we concluded that the latency was approximately linear. Indeed, since there is one division of 26 cycles for the initialization of B and that every new iteration there is a new division of 26 cycles (equation (4) in Table 1) and parallel operations that take the latency of approximately 20 cycles for the first few iterations and that grows very slowly, the sum of all these terms gives a rough linear function. Fig. 4 also shows the latency for S3. The resources trade-off had the downside effect of approximately cutting in half the throughput if we compare it with S1. In terms of latency, S1, S2 and S3 are similar with S3 taking a little less clock cycles. Of particular interest is the case when a new UT is added or when an existing UT has a new channel. Indeed, if a new UT is included, the starting point of the algorithm is the $K \times K$ already computed matrix inverse. In other words, every time a new UT is added, the algorithm only needs to do one iteration before finding the new inverse. Also, as shown in [19], when the channel changes for one UT, only a partial computation equivalent in terms of latency of twice (removing the UT and then adding it with its new CSI) the last iteration of the algorithm needs to be done. Other algorithms need to redo all the calculations to get to the new inverse. Table 3 shows the approximative latency for different configurations for updating the matrix inverse when one UT has a new channel. This approximation is computed by multiplying by 2 the result obtained by the difference between the total latency of two designs with $n-1$ and n UTs, where n is arbitrary. The obtained latency gains are not neglectable and can represent a big advantage for time-critical applications.

TABLE 3. ESTIMATED LATENCY TO UPDATE ONE UT

Strategy	Number of UTs	Estimated latency to replace one UT (µs)
S1	4	0.243
	6	0.264
	8	**0.285**
S2	4	**0.259**
	6	0.306
	8	**0.352**
S3	8	**0.201**
	12	0.236

E. Energy efficiency estimates

We have been able to compute an estimate of the power consumption of our designs with the help of the Xilinx Power Estimator spreadsheet. Post place and route timing simulations to estimate energy efficiency would be worth in a complete end to end implementation case. The goal here is just to give a rough idea of the power estimates of our design. The main contribution of the paper is still the throughput results obtained with the RGMIU algorithm. Table 4 presents the energy efficiency for our three strategies and for different numbers of UTs. This efficiency is shown in terms of watts (W) and Gigabits per Joule (Gb/J). The first thing to notice is that the energy efficiency is decreasing with the number of UTs no

TABLE 4. ENERGY EFFICIENCY VS OPTIMIZATION STRATEGY

Strategy	Number of UTs	Power consumption estimates (W)	Energy efficiency estimates (Gb/J)
S1	4	1.16	7.837
	6	2.541	5.367
	8	5.03	**3.615**
S2	4	2.068	6.001
	6	5.306	3.508
	8	11.34	**2.189**
S3	8	6.291	**1.480**
	12	16.809	0.831

matter which strategy is used. Clearly, since S2 is heavily pipelined, the design clock can reach a higher frequency so that the power consumption is heavily increased. From S1 to S2, for 8 UTs in a 64 QAM modulation, we pass from approximately 3.615 Gb/J to 2.189 Gb/J so that the trade-off for S2 is a higher throughput for a lower energy efficiency and a higher resources consumption. On the other hand, since S3 needs to compensate the fixed limit of DSP48Es with a lot of LUTs and FFs, the power consumption is higher compared to S1. Indeed, we get a 1.480 Gb/J efficiency for 8 UTs in a 64 QAM modulation. The trade-off then becomes a sacrifice of throughput and energy consumption for a more balanced resources utilization. It is important to note here that the efficiency is computed for the core algorithm only. By adding the preprocessing and the post-processing units, the energy efficiency would be a little bit less than previously computed.

V. CONCLUSION

To conclude, we have compared in simulations the performances of four different types of algorithm detection in terms of error symbol rate (SER). We have shown that the performances of iterative algorithms are sensitive to the number of iterations chosen but when it is carefully chosen, the GS algorithm has the same SER performances as the RGMIU method and the OCD algorithm has the potential to get better results due to the fact that it can reach near MMSE performances. On the other hand, NSE algorithm has very poor results when the number of UTs increase. Nevertheless, we have done these simulations to support the point that direct matrix inversion always leads to predictable and accurate results.

Also, we have implemented the RGMIU algorithm on Vivado HLS and have shown the great potential of this method to reach high throughput data detection in a massive MIMO system. The design was done on 16 bits words and performances for three different designs in terms of resources consumption, latency/throughput and energy efficiency were compared to bring out the possible trade-offs.

Since our proposed implementations concern the core of the RGMIU algorithm, it would not be fair to directly compare their performances with others known techniques in the literature simply because we assume a preprocessing and a post-processing unit that can keep up with the core algorithm data rate. Just by looking at the results, throughputs in the order of magnitudes of Gb/s are obtained as opposed to Mb/s in the literature. Resources consumption is also difficult to directly compare since preprocessing and post processing are not considered in our design.

A more in-depth study needs to be done to accelerate the pre and post-processing steps by either approximating the Gram matrix, increasing their clock rate or completely getting around these units.

Our future work will also consist of implementing the pipeline interleaving strategy for RGMIU as well as the GS, OCD and NSE algorithms to present a full comparison between all these methods.

ACKNOWLEDGMENTS

This work has been funded by the Natural Sciences and Engineering Research Council of Canada, Prompt, Canadian Foundation for Innovation, the CMC Microsystems and NUTAQ innovation.

REFERENCES

[1] E. Björnson, L. Sanguinetti, H. Wymeersch, J. Hoydis, and T.L. Marzetta, "Massive MIMO is a Reality – What is Next?: Five Promising Research Directions for Antenna Arrays," *Digital Signal Processing*, vol. 94, 2019, pp. 3–20.

[2] E. Björnson, "A look at an LTE-TDD Massive MIMO product," http://ma-mimo.ellintech.se/2018/08/27/ Accessed 7 July 2020.

[3] P. von Butovitsch, D. Astely, C. Friberg, A. Furuskär, B. Göransson, B. Hogan, J. Karlsson, and E. Larsson, "Advanced antenna systems for 5G networks. Ericsson white paper," https://www.ericsson.com/4a8a87/assets/local/publications/white-papers/10201407_wp_advanced_antenna_system_nov18_181115.pdf (Accessed 7 July 2020

[4] Y. Qu, A. Lozano, and A. Gatherer, "Nine Communications Technology Trends for 2019," Communication society technology news, 2019. https://www.comsoc.org/publications/ctn/nine-communications-technology-trends-2019 (Accessed 7 July 2020)

[5] H.Q. Ngo, "*Massive MIMO: fundamentals and system designs*," Ph.D. Thesis. Linköping University Electronic Press, 2015.

[6] M. Wu, B. Yin, G. Wang, C. Dick, J.R. Cavallaro, and C. Studer, "Large-scale MIMO detection for 3GPP LTE: algorithms and FPGA implementations," *IEEE Journal of Selected Topics in Signal Processing*, vol. 8, no. 5, 2014, pp. 916–929.

[7] Z. Wu, C. Zhang, Y. Xue, S. Xu, and X. You, "Efficient architecture for soft-output massive MIMO detection with Gauss-Seidel method," IEEE Int. Symp. on Circuits and Systems, May 2016, pp. 1886–1889.

[8] M. Wu, C. Dick, J.R. Cavallaro, and C. Studer, "High-throughput data detection for massive MU-MIMO-OFDM using coordinate descent," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 12, 2016, pp. 2357–2367.

[9] B. Yin, M. Wu, G. Wang, C. Dick, J. R. Cavallaro, and C. Studer, "A 3.8Gb/s large-scale MIMO detector for 3GPP LTE-advanced," Proc. IEEE Int. Conf. Acoust., Speech, Signal Proc., May 2014, pp. 3879–3883.

[10] J. Zhou, Y. Ye, and J. Hu, "Biased MMSE soft-output detection based on Jacobi method in massive MIMO," Proc. IEEE Int. Conference on communication problem-solving, Dec 2014, pp. 442–445.

[11] W. Song, X. Chen, L. Wang, and X. Lu, "Joint conjugate gradient and Jacobi iteration based low complexity precoding for massive MIMO systems," IEEE International Conference on Communication, July 2016, pp. 1–5.

[12] B. Yin, M. Wu, J. R. Cavallaro, and C. Studer, "VLSI design of largescale soft-output MIMO detection using conjugate gradients," IEEE International Symposium on Circuits and Systems, May 2015, pp. 1498–1501.
[13] Z. Wu, Y. Xue, X. You, and C. Zhang, "Hardware efficient detection for massive MIMO uplink with parallel Gauss-Seidel method," International Conference on Digital Signal Processing, August 2017, pp. 1–5.
[14] Z. Zhang, J. Wu, X. Ma, Y. Dong, Y. Wang, S. Chen, and X. Dai, "Reviews of recent progress on low-complexity linear detection via iterative algorithms for massive MIMO systems," IEEE International Conference on Communication, July 2016, pp. 1–6.
[15] X. Qin, Z. Yan, and G. He, "A near-optimal detection scheme based on joint steepest descent and Jacobi method for uplink massive MIMO systems," IEEE Communication Letter, vol. 20, no. 2, Feb. 2016, pp. 276–279.
[16] C. Jeon, K. Li, J.R Cavallaro and C. Studer, "Decentralized Equalization with Feedforward Architectures for Massive MU-MIMO," *IEEE Transactions on Signal Processing*, vol. 67, no. 17, 2019, pp. 4418–4432.
[17] M. Ahmed Ouameur, and D. Massicotte, "Efficient Distributed Processing for Large Scale MIMO Detection," *European Signal Processing Conference (Eusipco)*, A Coruna, Spain, 2-6 Sept. 2019, pp. 1–4.
[18] M. Ahmed Ouameur and D. Massicotte, "Deep Autoencoder for Interconnect's Bandwidth Relaxation in Large Scale MIMO-OFDM Processing", 2019, https://arxiv.org/abs/1907.12613. (Accessed 7 July 2020)
[19] M. Ahmed Ouameur, D. Massicotte, A. M. Akhtar and R. Girald, "Performance Evaluation and Implementation Complexity Analysis Framework for ZF Based Linear Massive MIMO Detection," Wireless Networks Journals, Springer, pp. 1–15, April 2020.
[20] M. A. Albreem, M. Juntti and S. Shahabuddin, "Massive MIMO Detection Techniques: A Survey," IEEE Communications Surveys & Tutorials, vol. 21, no. 4, 2019, pp. 3109-3132.
[21] Xilinx, Vivado Design Suite User Guide, High-Level Synthesis UG902 (v2019.1) July 12, 2019.
[22] Xilinx, SDx Pragma Reference Guide UG1253 (v2019.1) June 5, 2019.