Elsevier Editorial System(tm) for Digital Signal Processing
Manuscript Draft

Manuscript Number:

Title: Parallel Sphere Detector algorithm providing optimal MIMO detection on massively parallel architectures

Article Type: Regular Paper

Keywords: MIMO; parallel ML detection; sphere detector; hybrid tree search; GP-GPU

Corresponding Author: Mr. Csaba Máté Józsa, M.Sc.

Corresponding Author's Institution: Pázmány Péter Catholic University

First Author: Csaba M Józsa, M.Sc.

Order of Authors: Csaba M Józsa, M.Sc.; Géza Kolumbán, D.Sc.; Antonio M Vidal, Ph.D.; Francisco J Martínez-Zaldívar, Ph.D.; Alberto González, Ph.D.

Abstract: Multiple-input multiple-output (MIMO) systems have attracted considerable attention in wireless communications because they offer a significant increase in data throughput and link coverage without additional bandwidth requirement or increased transmit power. The price that has to be paid is the increased complexity of hardware components and algorithms. Recent results have proved that Massively Parallel Architectures (MPAs) are able to solve computationally intensive tasks in a very efficient manner. The Sphere Detector (SD) algorithm solves the problem of Maximum Likelihood (ML) detection for MIMO channels by significantly reducing the search space of the possible solutions. The main drawback of the SD algorithm is in its sequential nature, consequently, running it on MPAs is very inefficient. In order to overcome the limited possibilities of the SD algorithm, a new Parallel Sphere Detector algorithm is proposed here. It implements a novel hybrid tree search method where (i) the algorithm parallelism is assured by the alternating use of depth-first search (DFS) and breadth-first search (BFS) algorithms and (ii) the search is combined with a path metric based sorting on each intermediate stage. The PSD algorithm is able to adjust its memory requirements and extent of parallelism to fit a wide range of parallel architectures. Mapping details for MPAs are proposed, by giving the details of thread dependent, highly parallel building blocks of the algorithm. Based on the proposed building blocks, performance is evaluated on a General-Purpose Graphics Processing Unit (GP-GPU). In order to achieve high-throughput several levels of parallelism are introduced and different scheduling strategies are considered.

Cover Letter

Pázmány Péter Catholic University
Faculty of Information Technology

UNIVERSIDAD POLITECNICA DE VALENCIA

Dr. Ercan E. Kuruoglu
Editor-in-Chief
Digital Signal Processing
Elsevier

Budapest, December 5, 2013

Dear Editor Prof. Kuruoglu,

Together with this letter you will find the manuscript of the paper entitled *Parallel Sphere Detector algorithm providing optimal MIMO detection on massively parallel architectures*, by C. M. Józsa, G. Kolumbán, A. M. Vidal, F. J. Martínez-Zaldivar and A. González, to be considered for publication in the Journal of Digital Signal Processing.

With the submission of this manuscript I would like to declare the followings:

- All authors of this research paper have directly participated in the planning, execution, or analysis of this study;
- All authors of this paper have read and approved the final version submitted;
- The contents of this manuscript have not been copyrighted or published previously;
- The contents of this manuscript are not now under consideration for publication elsewhere;
- The contents of this manuscript will not be copyrighted, submitted, or published elsewhere, while acceptance by the Journal is under consideration.

We think that the submitted manuscript is relevant for the journal, because it has a special emphasis on statistical signal processing methodology such as Bayesian signal processing. In multiple-input multiple-output (MIMO) systems Maximum Likelihood (ML) detection is performed based on Bayesian signal processing. The ML detection problem drops down to the Integer Least Squares (ILS) problem. In this paper we propose a Parallel Sphere Detector (PSD) algorithm, which overcomes the problems caused by the sequential nature of the traditional Sphere Detector algorithms, and we prove that ILS problems can be efficiently solved with the proposed PSD algorithm. The algorithm can adjust the extent of parallelism and its resource needs by its parameters, thus it can be adjusted to a wide range of modern parallel architectures. Mapping details to General-Purpose Graphics Processing Units (GP-GPU) are also given.

The contact address for any subject related to this submission is:

Csaba Máté Józsa
Faculty of Information Technology
Pázmány Péter Catholic University
Práter str. 50/A
1083 Budapest, Hungary

e-mail: jozsa.csaba@itk.ppke.hu
Tel: +36 30 924-5724
Fax: +36 1 886-4724

Pázmány Péter Catholic University
Faculty of Information Technology

UNIVERSIDAD POLITECNICA DE VALENCIA

We hope that the paper do comes up to your expectation not just in terms of originality of content as well as in terms of submission standards. Expecting the manuscript to be considered worth publishing in your journal,

Yours Sincerely,

C. M. Józsa, G. Kolumbán, A. M. Vidal, F. J. Martínez-Zaldivar and A. González

**Highlights (for review)**

- A novel parallel sphere detector algorithm for ML detection is proposed.
- The algorithm relies on a hybrid tree search method suitable for multi/many-core architectures.
- Thread dependent, highly parallel building blocks for every stage of the algorithm are designed and implemented.
- Several levels of parallelism are identified and exploited to reach peak detection throughput.
- Static and dynamic work distribution strategies are developed and compared.
- The performance of the proposed parallel sphere detector algorithm is evaluated on state-of-the-art GP-GPUs.

# Parallel Sphere Detector algorithm providing optimal MIMO detection on massively parallel architectures

Csaba M. Józsa[a], Géza Kolumbán[a], Antonio M. Vidal[b], Francisco J. Martínez-Zaldívar[c], Alberto González[c]

[a] *Faculty of Information Technology, Pázmány Péter Catholic University, Práter str. 50/A, 1083 Budapest, Hungary*
[b] *Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Camino de Vera s/n, 46022 València, Spain*
[c] *Departamento de Comunicaciones, Universitat Politècnica de València, Camino de Vera s/n, 46022 València, Spain*

**Abstract**

Multiple–input multiple-output (MIMO) systems have attracted considerable attention in wireless communications because they offer a significant increase in data throughput and link coverage without additional bandwidth requirement or increased transmit power. The price that has to be paid is the increased complexity of hardware components and algorithms. Recent results have proved that Massively Parallel Architectures (MPAs) are able to solve computationally intensive tasks in a very efficient manner. The Sphere Detector (SD) algorithm solves the problem of Maximum Likelihood (ML) detection for MIMO channels by significantly reducing the search space of the possible solutions. The main drawback of the SD algorithm is in its sequential nature, consequently, running it on MPAs is very inefficient. In order to overcome the limited possibilities of the SD algorithm, a new Parallel Sphere Detector algorithm is proposed here. It implements a novel hybrid tree search method where (i) the algorithm parallelism is assured by the alternating use of depth-first search (DFS) and breadth-first search (BFS) algorithms and (ii) the search is combined with a path metric based sorting on each intermediate stage. The PSD algorithm is able to adjust its memory requirements and extent of parallelism to fit a wide range of parallel architectures. Mapping details for MPAs are proposed, by giving the details of thread dependent, highly parallel building blocks of the algorithm. Based on the proposed building blocks, performance is evaluated on a General-Purpose Graphics Processing Unit (GP-GPU). In order to achieve high-throughput several levels of parallelism are introduced and different scheduling strategies are considered.

*Keywords:* MIMO, parallel ML detection, sphere detector, hybrid tree search, GP-GPU

## 1. Introduction

The most important driving forces in the development of wireless communications are the need for higher link throughput, higher network capacity and improved reliability. The limiting

*Email addresses:* jozsa.csaba@itk.ppke.hu (Csaba M. Józsa), kolumban.geza@itk.ppke.hu (Géza Kolumbán), avidal@dsic.upv.es (Antonio M. Vidal), fjmartin@dcom.upv.es (Francisco J. Martínez-Zaldívar), agonzal@dcom.upv.es (Alberto González)

factors of such systems are equipment cost, radio propagation conditions and frequency spectrum availability. Research in information theory has revealed that important improvements can be achieved in data rate when multiple antennas are applied at both the transmitter and receiver sides. The key feature of MIMO [1] systems is the ability to turn multipath propagation, traditionally a pitfall of wireless transmissions, into a benefit for the user. The success of MIMO lies in the fact that the performance of wireless systems is improved by orders of magnitude at no cost of extra spectrum requirement. MIMO techniques can increase the robustness of wireless communication systems by transmitting different representations of the same data stream on different parallel transmit branches, or they can achieve a higher throughput by transmitting independent data streams on different transmit branches simultaneously and within the same frequency band. The price that has to be paid is the increased complexity of detection hardware components and algorithms. The complexity of detection algorithms used over different receiver structures depends on many factors, such as antenna configuration, modulation order, channel, coding, etc.

The manycore parallel architectures, such as GP-GPUs or Field Programmable Gate Arrays (FPGAs), are playing a prominent role in computer sciences because of their general purpose, high computational performance and cheap price. The trend is that market leading smart phones use sophisticated GP-GPUs, and the use of high-performance GP-GPU clusters are more and more common. Research conducted in several scientific areas has shown that the GP-GPU approach is very powerful which offers a considerable improvement in system performance at a low cost.

In this paper we propose a new parallel SD algorithm, that uses a novel hybrid tree search method to enable parallel processing, thus making it suitable for MPAs. Several papers [2], [3], [4], [5], [6], [7] are available in the literature focused on finding a near-ML solution where a significant decrease in computational complexity is achieved. However, our goal is to find the optimal ML solution. The drawback of finding the ML solution is the increased and variable complexity.

The paper is organized as follows. Section 2 defines the model of MIMO system considered here. Section 3 describes the SD algorithm and shows how its detection complexity can be reduced. Section 4 introduces the PSD algorithm proposed here and compares the SD and PSD algorithms from an algorithmic point of view. The novel hybrid tree search method is also analyzed and a detailed description of the building blocks of the PSD algorithm is provided. Section 5 gives a brief overview of CUDA programming model. Section 6 highlights the importance of multilevel parallelism. Two computing load distribution strategies are presented and it is showed how they can be applied on a GP-GPU with multi-stream configuration. Section 7 evaluates the performance of the PSD algorithm proposed by giving simulation results on the achieved average detection throughput and the distribution of work over the available threads. Finally, Section 8 concludes the main results.

## 2. System Model

A MIMO system consists of $n$ transmit and $m$ receive antennas as shown in Fig. 1. The transmit antennas are sending a complex signal vector $\tilde{\mathbf{s}}_\mathbf{t}$ of size $n$ during one symbol period. The components of $\tilde{\mathbf{s}}_\mathbf{t} = (\tilde{s}_1, \tilde{s}_2, ..., \tilde{s}_n)^T$ are drawn from a complex symbol set $\tilde{\Omega}$. The received complex symbol vector $\tilde{\mathbf{y}} = (\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_m)^T$ is expressed as

$$\tilde{\mathbf{y}} = \tilde{\mathbf{H}}\tilde{\mathbf{s}}_\mathbf{t} + \tilde{\mathbf{v}} \tag{1}$$

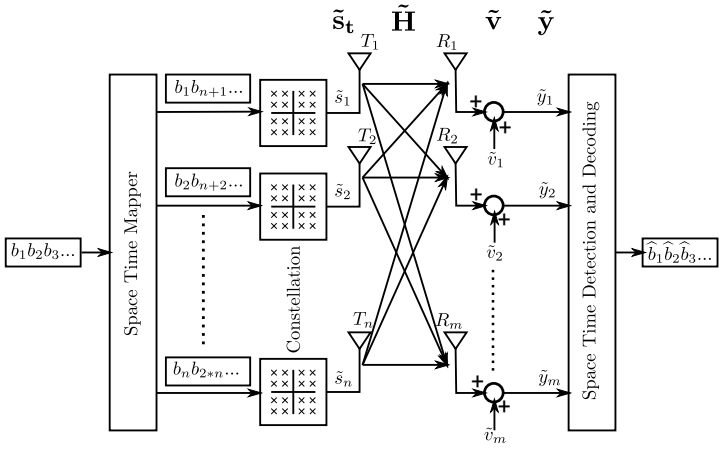



Figure 1: MIMO system model.

where $\tilde{\mathbf{v}} = (\tilde{v}_1, \tilde{v}_2, ..., \tilde{v}_m)^T$, an independent and identically distributed (i.i.d.) circular symmetric complex multivariate Gaussian random variable $CN(0, \mathbf{K})$ with the covariance matrix $\mathbf{K} = \sigma_n^2\mathbf{I_m}$, models the channel noise. The entries $\tilde{h}_{ij}$ of the channel matrix $\tilde{\mathbf{H}}$ are assumed to be i.i.d. zero-mean complex Gaussian variables with unit variance. We assume a block-fading channel where the channel-fading matrix remains quasi-static within a fading block, but it is independent between successive fading blocks.

The ML detection estimates the transmitted complex symbol vector as

$$\tilde{\mathbf{s}}_\mathbf{ml} = \arg\min_{\tilde{\mathbf{s}}\epsilon\tilde{\Omega}^\mathbf{n}} \|\tilde{\mathbf{y}} - \tilde{\mathbf{H}}\tilde{\mathbf{s}}\|^2. \tag{2}$$

In order to simplify the problem, the original complex representation of the system model, presented in Eq. (1), is transformed into an equivalent real-valued model at the cost of increasing its dimension:

$$\mathbf{y} = \mathbf{H}\mathbf{s}_\mathbf{t} + \mathbf{v} \tag{3}$$

where

$$\mathbf{y} = \begin{pmatrix} \Re(\tilde{\mathbf{y}}) \\ \Im(\tilde{\mathbf{y}}) \end{pmatrix}_{M\times 1}, \mathbf{s_t} = \begin{pmatrix} \Re(\tilde{\mathbf{s}}_\mathbf{t}) \\ \Im(\tilde{\mathbf{s}}_\mathbf{t}) \end{pmatrix}_{N\times 1}, \mathbf{v} = \begin{pmatrix} \Re(\tilde{\mathbf{v}}) \\ \Im(\tilde{\mathbf{v}}) \end{pmatrix}_{M\times 1}, \mathbf{H} = \begin{pmatrix} \Re(\tilde{\mathbf{H}}) & -\Im(\tilde{\mathbf{H}}) \\ \Im(\tilde{\mathbf{H}}) & \Re(\tilde{\mathbf{H}}) \end{pmatrix}_{M\times N}$$

moreover $M = 2 \cdot m$ and $N = 2 \cdot n$.

For the real valued system the ML solution is

$$\mathbf{s_{ml}} = \arg\min_{\mathbf{s}\epsilon\Omega^N} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2 \tag{4}$$

where $\mathbf{y}, \mathbf{H}, \mathbf{s_t}, \mathbf{s_{ml}}$ are all real-valued quantities and $\Omega$ is a real-valued signal set. Equation (4) shows that the maximum likelihood estimate of the symbol vector is found by solving an integer least-squares (ILS) problem which is analogous to finding the closest lattice point of lattice $\mathbf{\Lambda} = \{\mathbf{Hs} : \mathbf{s}\epsilon\Omega^N\}$ to a given point $\mathbf{y}$ [8], [9]. In lattice theory this problem is often referred to as the closest lattice point search (CLPS) [10], [11]. The exhaustive search implementation of ML

detection has a complexity that grows exponentially with both the number of elements in the signal set $\Omega$ and the number of antennas. Consequently, the required computational performance becomes unattainable. For general lattices the problem has been shown to be NP-hard [12]. However, significant complexity reduction can be achieved by exploiting the structure of the lattice as shown in [13], [14].

## 3. The Sphere Detector Algorithm

The fundamental aim of the SD algorithm is to restrict the search to lattice points that lie within a certain sphere of radius $d$ around a given received symbol vector. Reducing the search space will not affect the detection quality, because the closest lattice point inside the sphere will also be the closest lattice point for the whole lattice. The reduction of the search space is necessary in order to reduce the high computational complexity required by the ML detection.

The complexity analysis of the SD algorithm has been thoroughly investigated by the researchers, for a few good examples refer to [15], [16], [17], [18], [19]. It has been shown that the complexity of the SD algorithm is directly proportional to the number of lattice points explored. The search space is highly influenced by the chosen sphere radius. Choosing a small radius may result in an empty sphere, while the choice of a too large radius may lead to an increased complexity. For an arbitrary lattice $\mathbf{\Lambda}$ the search of the optimal (covering) radius requires a number of steps that grows exponentially [20] with the dimension of the lattice. Thus, this approach is not feasible for real systems.

In order to exploit the advantage of the search space reduction, a good enumeration strategy is needed. In [21] Pohst proposed an efficient way of enumerating lattice points inside a sphere. Pohst's method was first implemented in digital communications by Viterbo and Biglieri [22]. Important speedups have been achieved by Schnorr and Euchner [14] by refining the Pohst method. Agrel et al. in [11] showed that the Schnorr-Euchner strategy can be efficiently used for CLPS. The benefit of the SD algorithm lies in the enumeration strategy of the lattice points. A detailed description of the SD algorithm can be found in [23] and the pseudo-code of the SD algorithm is presented in Algorithm 1.

Without any loss of generality we may assume that (i) $N = M$, i.e., the number of transmit antennas equals to that of receive ones, and (ii) the channel matrix has full rank. Furthermore, we assume that perfect channel state information (CSI) is available at the receiver. Taking the *unconstrained least-squares* solution

$$\hat{\mathbf{s}} = \mathbf{H}^{-1}\mathbf{y} \tag{5}$$

of the equivalent real system and applying QR factorization to the real channel matrix $\mathbf{H} = \mathbf{QR}$, the ML solution (4) can be rearranged as

$$\begin{aligned} \mathbf{s_{ml}} &= \arg\min_{\mathbf{s}\epsilon\Omega^N} \|\mathbf{y} - \mathbf{Hs}\|^2 \\ &= \arg\min_{\mathbf{s}\epsilon\Omega^N} (\mathbf{s} - \hat{\mathbf{s}})^{\mathbf{T}}\mathbf{H}^{\mathbf{T}}\mathbf{H}(\mathbf{s} - \hat{\mathbf{s}}) \\ &= \arg\min_{\mathbf{s}\epsilon\Omega^N} (\mathbf{s} - \hat{\mathbf{s}})^{\mathbf{T}}(\mathbf{QR})^{\mathbf{T}}(\mathbf{QR})(\mathbf{s} - \hat{\mathbf{s}}) \\ &= \arg\min_{\mathbf{s}\epsilon\Omega^N} \|\mathbf{R}(\mathbf{s} - \hat{\mathbf{s}})\|^2 \end{aligned} \tag{6}$$

where matrix **Q** is orthogonal and matrix **R** upper triangular. The lattice point **Hs** is included by the sphere $S(\mathbf{y}, d)$ with center point **y** and radius $d$ if the following inequality is satisfied

$$\|\mathbf{R}(\mathbf{s}-\hat{\mathbf{s}})\|^2 \leqslant d^2 \tag{7}$$

$$\left\| \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1N} \\ 0 & r_{22} & \cdots & r_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{NN} \end{pmatrix} \begin{pmatrix} s_1 - \hat{s}_1 \\ s_2 - \hat{s}_2 \\ \vdots \\ s_N - \hat{s}_N \end{pmatrix} \right\|^2 \leqslant d^2.$$

Instead of evaluating all possible symbol combinations we exploit the upper triangular property of matrix **R** and define a recursion based on the dependency hierarchy of the terms.

In order to give a deeper insight, let $\mathbf{s_i^N} \triangleq (s_i, s_{i+1}, \cdots, s_N)^T$, referred to as partial symbol vector, denote the last $N-i+1$ components of the vector **s** and let $M(\mathbf{s_i^N}) = \sum_{j=i}^{N} \left| \sum_{k=j}^{N} r_{jk}(s_k - \hat{s}_k) \right|^2$ define the metric of $\mathbf{s_i^N}$. The recursion starts at level $N$ and a solution candidate is found when the first level is reached. In every iteration a partial symbol vector $\mathbf{s_i^N}$ is *expanded*. During the *expansion* one symbol $s_{i-1} \in \Omega$ is selected from the symbol set and it is added to the partial symbol vector as follows $\mathbf{s_{i-1}^N} = (s_{i-1}, s_i, \cdots, s_N) = (s_{i-1}, \mathbf{s_i^N})$. The *evaluation* of the new partial symbol vector $\mathbf{s_{i-1}^N}$ is the computation of the path metric $M(\mathbf{s_{i-1}^N})$. If the conditions are met, namely, when $M(\mathbf{s_{i-1}^N}) < d^2$, a new symbol $s_{i-2} \in \Omega$ has to be selected for the next dimension. If not then the symbol $s_{i-1}$ chosen previously is discarded and a new symbol for the same level is chosen from the signal set.

A possible solution is found if a complete symbol vector $\mathbf{s_1^N}$ satisfies the condition $M(\mathbf{s_1^N}) < d^2$. The solution with the smallest metric is the ML solution. If a too small initial radius is chosen the solution is not found, consequently, the process has to be restarted with a higher radius.

Based on the above description of the SD algorithm an analogy with bounded tree search can be found. The partial symbol vectors $\mathbf{s_i^N}$ can be regarded as tree nodes at level $i$. The symbol vectors $\mathbf{s_1^N}$ are the leaves of the tree and the weight of each node is defined by the symbol vector metric $M(\mathbf{s_i^N})$. The continuous change of the partial symbol vector $\mathbf{s_i^N}$ is analogous to a DFS. The condition given in Eq. 7 can be regarded as the bounding criteria.

## 4. The Parallel Sphere Detector algorithm

Section 3 concluded that the SD algorithm can be regarded as a branch and bound tree search problem. Khairy et al. showed in [4] that significant speed-up can be achieved by executing multiple sequential sphere decoders simultaneously. However, to achieve even a better performance it is mandatory to redesign the sequential algorithm using several effective parallel design patterns in order to exploit all advantages of parallel computing capabilities of multi-core and many-core architectures. The design process is complex, because the new parallel algorithm has to be general enough to satisfy the requirements and limitations imposed by the different parallel architectures.

### *4.1. Key Concepts of the PSD algorithm*

The key concepts of the PSD algorithm design process are as follows:

**Algorithm 1** SD algorithm for estimating $\mathbf{s_{ml}} = (s_1, s_2, \cdots, s_N)$

**Require:** $\hat{\mathbf{s}}, \mathbf{R}, |\Omega|$

**procedure** DEFINITION AND INITIALIZATION OF VARIABLES
  **for** $j = 1$ **to** $N$ **do**
    $\mathbf{eval_j} \leftarrow |\mathbf{\Omega}|$ ▷ Number of partial symbol vector evaluations on level j after a node expansion
    $\mathbf{buf_j}[\mathbf{eval_j}] = \{\}$ ▷ Denotes an empty buffer of size $\mathbf{eval_j}$ for level $j$
    $\mathbf{off_j} \leftarrow 0$ ▷ Offset of processing on level j for buffer $\mathbf{buf_j}$
  **end for**
  $\mathbf{buf_N} \leftarrow$ EXPAND(()) ▷ Expand the root () of the tree and update $\mathbf{buf_N}$
  PROCESS($i \leftarrow N - 1$)
**end procedure**
**procedure** PROCESS($i$)
  **while** $i < N$ **do**
    **if** $\mathbf{off_{i+1}} < \mathbf{eval_{i+1}}$ **then**
      $\mathbf{s_{i+1}^N} \leftarrow \mathbf{buf_{i+1}}(\mathbf{off_{i+1}})$
      **if** $M(\mathbf{s_{i+1}^N}) < d^2$ **then** ▷ Evaluate $M(\mathbf{s_{i+1}^N})$ and check if $\mathbf{s_{i+1}^N}$ is inside the sphere $S(y, d)$
        $\mathbf{buf_i} \leftarrow$ EXPAND($\mathbf{s_{i+1}^N}$) ▷ Expand partial symbol vector $\mathbf{s_{i+1}^N}$ of the tree and update $\mathbf{buf_i}$
        **if** $i = 1$ **then**
          Find symbol vector $\mathbf{s_{ml}'}$ in $\mathbf{buf_1}$ with minimum path metric
          $d_{temp}^2 \leftarrow \|\mathbf{R}(\mathbf{s_{ml}'} - \hat{\mathbf{s}})\|^2$
          **if** $d_{temp}^2 < d^2$ **then** $d^2 \leftarrow d_{temp}^2$ and $\mathbf{s_{ml}} \leftarrow \mathbf{s_{ml}'}$ **end if**
          $i \leftarrow i + 1$
        **else**
          $\mathbf{off_{i+1}} \leftarrow \mathbf{off_{i+1}} + 1$, $i \leftarrow i - 1$
        **end if**
      **else**
        $\mathbf{off_{i+1}} \leftarrow \mathbf{off_{i+1}} + 1$
      **end if**
    **else**
      $\mathbf{off_{i+1}} \leftarrow 0$, $i \leftarrow i + 1$
    **end if**
  **end while**
**end procedure**
**procedure** EXPAND($\mathbf{s_i^N}$) ▷ The input is the partial symbol vector to be expanded
  **for** $j = 0$ **to** $|\Omega| - 1$ **do**
    **if** $\mathbf{s_i^N} = ()$ **then** ▷ When expanding the root node the partial symbol vector is empty
      $\mathbf{s_N^N} \leftarrow \Omega[j]$
      $\mathbf{buf_N}[j] \leftarrow \mathbf{s_N^N}$
    **else**
      $s_{i-1} \leftarrow \Omega[j]$
      $\mathbf{s_{i-1}^N} \leftarrow (s_{i-1}, \mathbf{s_i^N})$
      $\mathbf{buf_{i-1}}[j] \leftarrow \mathbf{s_{i-1}^N}$
    **end if**
  **end for**
**end procedure**

1. Consider an arbitrary lattice $\mathbf{\Lambda}$. The search of the optimal (covering) radius requires a number of steps that grows exponentially [20] with the dimension of the lattice, thus its use is not practical. A possible solution to the initial radius problem is the Zero Forcing (ZF) radius, since this radius guarantees the existence of at least one lattice point. However, it may happen that this choice of radius will yield too many lattice points lying inside the sphere. The ZF radius is defined as follows $d = \|\mathbf{y} - \mathbf{H}\hat{\mathbf{s}}_\mathbf{B}\|$ where $\hat{\mathbf{s}}_\mathbf{B} = \lfloor \hat{\mathbf{s}} \rceil$ is the Babai estimate and operator $\lfloor \cdot \rceil$ slices each element of the input vector to the closest symbol in the symbol set $\Omega$.
   However, one of our design objectives is to make the detection completely independent of the size of initial radius and to ensure that the detection process does not have to be restarted with an increased radius. By defining the initial radius as $d = \infty$ the above condition is fulfilled, however the SD algorithm becomes an exhaustive search. Consequently, the radius has to be updated during the detection process.
   In order to update the radius a leaf node has to be found, afterwards, the radius is adjusted to the path metric of leaf node as follows $d^2 = \mathbf{M}(\mathbf{s}_\mathbf{1}^\mathbf{N})$. If the path metric of the leaf is small, then the search efforts required are significantly reduced. Lai et al. in [24] have examined different hybrid tree search algorithms where the detection process has been started with BFS and continued as a DFS based on the branch metric of expanded nodes. The hybrid tree searching technique resulted in a near-ML detection in a few iterations. Consequently, the concept of hybrid search should be used for finding a small metric leaf node, i.e., a good initial radius.
2. The redesigned SD algorithm should support parallel architectures. This can be achieved by introducing a new work generation and distribution mechanism that is able to keep all the processing elements busy continuously. By expanding and evaluating multiple symbol vectors simultaneously the above goal could be reached. However, the extent of parallelism should be controlled by well defined parameters so that the new algorithm can adjust itself to any multi-core or many-core architecture.
3. Parallel architectures may have different memory hierarchies. In order to make the use of faster but usually smaller sized memories possible, the algorithm should have different parameter configuration in order to assure the efficient use of memory.

### *4.2. General description*

As stated previously, the parallelism of the SD algorithm is achieved by a hybrid tree search. The branching factor of the tree is equal to the size $|\mathbf{\Omega}|$ of symbol set. The depth of the tree depends on the number $M$ of receive antennas. Recall, it was assumed that $N = M$, i.e., the number of transmit antennas equals to that of receive ones.

Algorithm 2 gives a high level overview of the PSD algorithm. The definitions of the parameters used to describe the PSD algorithm are given in Table 1. The key parameters that determine the overall performance of the algorithm are: $\mathbf{lvl_{nr}}$, $\mathbf{lvl_x}$ and $\mathbf{exp_{lvl_x}}$. These parameters define the tree traversal process, determine the memory usage and, consequently, influence (i) the speed of reaching a leaf node, (ii) the metric of the first leaf node and (iii) the number of iterations required to find the optimal solution.

To get a better insight Table 2 shows a few valid parameter sets for different system configurations. Configurations 1, 2 and 3 have the same parameters $\mathbf{lvl_x}$, $\mathbf{exp_{lvl_x}}$ but because the different size of the symbol set is different, a significant change in memory requirements is obtained. Note, different parameters have to be used for the various system configurations and symbol sets.

Table 1: Definition of parameters used in PSD algorithm.

| | | |
|---|---|---|
| Tree traversal parameters | | |
| $\mathbf{lvl_{nr}}$ | the total number of tree levels where partial symbol vectors are evaluated | $0 < \mathbf{lvl_{nr}} \leq N$ |
| $\mathbf{lvl_x}$ | levels assigned for partial symbol vector evaluation | $\mathbf{lvl_0} = N + 1$, $\mathbf{lvl_{lvl_{nr}}} = 1$ $\mathbf{lvl_x} > \mathbf{lvl_{x+1}}$ |
| $\mathbf{exp_{lvl_x}}$ | number of partial symbol vectors expanded simultaneously on level $\mathbf{lvl_x}$ | $\mathbf{exp_{lvl_0}} = 1$ $\mathbf{exp_{lvl_x}} \leq \mathbf{eval_{lvl_x}}$ |
| $\mathbf{eval_{lvl_x}}$ | number of partial symbol vectors needed to be evaluated on level $\mathbf{lvl_x}$ after the expansion of partial symbol vectors on level $\mathbf{lvl_{x-1}}$ | $\mathbf{eval_{lvl_x}} = \mathbf{exp_{lvl_{x-1}}} * \lvert\Omega\rvert^{(\mathbf{lvl_{x-1}}-\mathbf{lvl_x})}$ |
| $\mathbf{max_{lvl_x}}$ | maximum number of partial symbol vectors on level $\mathbf{lvl_x}$ | $\mathbf{max_{lvl_x}} = \lvert\Omega\rvert^{(\mathbf{lvl_0}-\mathbf{lvl_x})}$ |
| Algorithm parameters | | |
| $\mathbf{tt}$ | total number of threads assigned for detection | |
| $\mathbf{t_{id}^k}$ | thread with identifier $\mathbf{k}$ | |
| $\mathbf{buf_{lvl_x}}$ | buffer for the evaluated partial symbol vectors $\mathbf{s_{lvl_x}^N}$ on level $\mathbf{lvl_x}$ | $size(\mathbf{buf_{lvl_x}}) = \mathbf{eval_{lvl_x}}$ |
| $\mathbf{off_{lvl_x}}$ | offset of processing on level $\mathbf{lvl_x}$ for $\mathbf{buf_{lvl_x}}$ | $0 \leq \mathbf{off_{lvl_x}} \leq \mathbf{eval_{lvl_x}}$ |
| $\mathbf{s_{lvl_x}^{N<j>}}$ | partial symbol vector on level $\mathbf{lvl_x}$ where $j$ is the index of the partial symbol vector in buffer $\mathbf{buf_{lvl_x}}$ | $0 \leq j < \mathbf{eval_{lvl_x}}$ |
| $\mathbf{vt_{lvl_x}}$ | virtual thread identifier calculated from $\mathbf{lvl_x}$, $\mathbf{t_{id}^k}$ and $\mathbf{tt}$ | Based on Eq. 9 |
| $\mathbf{vb_{lvl_x}}$ | virtual block identifier calculated from $\mathbf{lvl_x}$, $\mathbf{t_{id}^k}$ and $\mathbf{tt}$ | Based on Eq. 10 |

**Algorithm 2** High-level overview of the PSD algorithm

1: Expand distinct levels of nodes and evaluate them. ▷ This ensures enough computational load to maintain the cores active.
2: Repeat until a leaf level is reached

   1. Sort the previously expanded nodes by their path metric.
   2. Expand nodes further from a subset of previously sorted nodes for the following distinct level.

3: When a leaf level is reached

   1. Find the minimum metric leaf and update the sphere radius.
   2. Proceed with the rest of the nodes evaluated at the previous level.

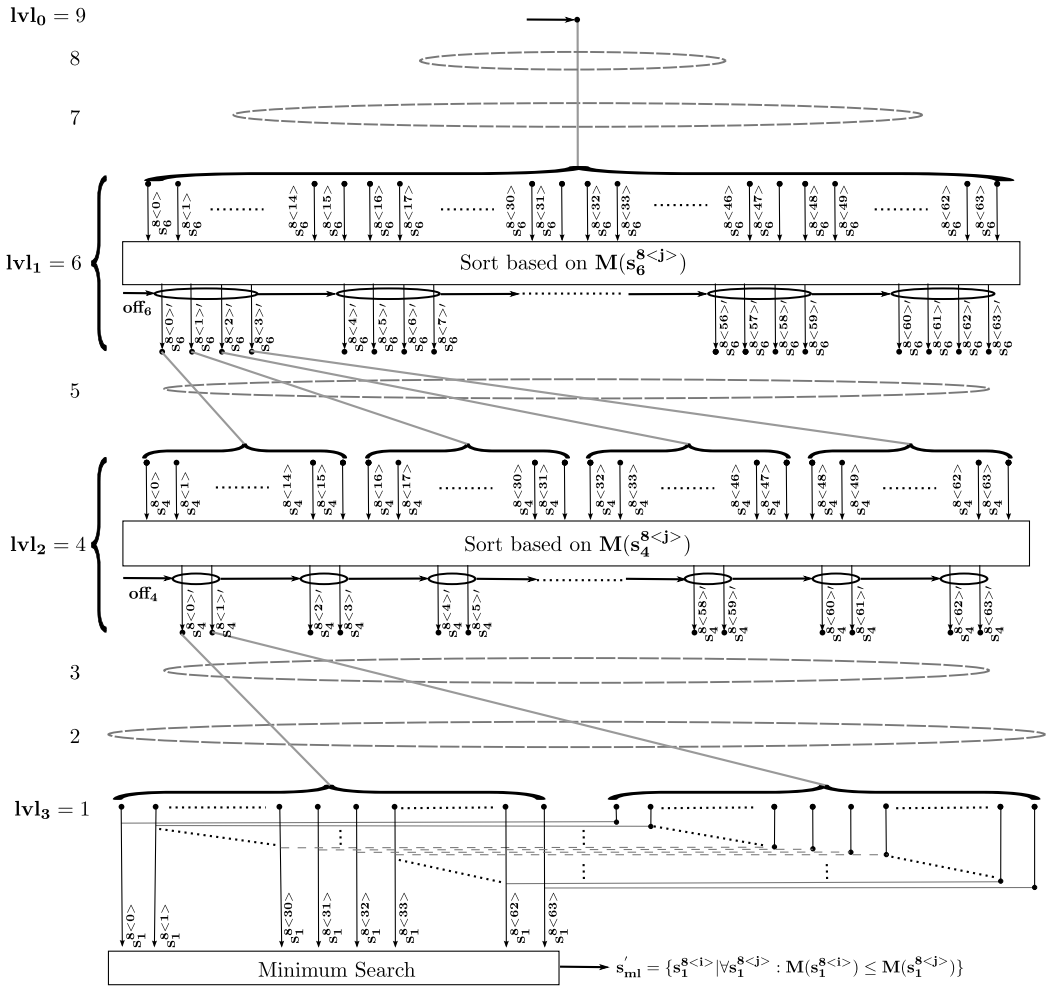



Figure 2: Structure of PSD algorithm for a 4x4 MIMO system with $|\Omega| = 4$.

Table 2: Various valid PSD algorithm configurations.

| Configuration | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Antennas | 2x2 | 2x2 | 2x2 | 4x4 | 4x4 | 4x4 |
| Symbol set size | 2 | 4 | 8 | 4 | 8 | 8 |
| $\mathbf{lvl_{nr}}$ | 2 | 2 | 2 | 3 | 4 | 4 |
| $\mathbf{lvl_0}$ | 5 | 5 | 5 | 9 | 9 | 9 |
| $\mathbf{lvl_1}$ | 2 | 2 | 2 | 6 | 7 | 7 |
| $\mathbf{lvl_2}$ | 1 | 1 | 1 | 4 | 6 | 6 |
| $\mathbf{lvl_3}$ | 0 | 0 | 0 | 1 | 3 | 2 |
| $\mathbf{lvl_4}$ | 0 | 0 | 0 | 0 | 1 | 1 |
| $\mathbf{exp_{lvl_0}}$ | 1 | 1 | 1 | 1 | 1 | 1 |
| $\mathbf{exp_{lvl_1}}$ | 4 | 4 | 4 | 4 | 2 | 2 |
| $\mathbf{exp_{lvl_2}}$ | 0 | 0 | 0 | 2 | 3 | 3 |
| $\mathbf{exp_{lvl_3}}$ | 0 | 0 | 0 | 0 | 4 | 4 |
| $\mathbf{eval_{lvl_1}}$ | 8 | 64 | 256 | 64 | 64 | 64 |
| $\mathbf{eval_{lvl_2}}$ | 8 | 16 | 32 | 64 | 16 | 16 |
| $\mathbf{eval_{lvl_3}}$ | 0 | 0 | 0 | 128 | 768 | 12288 |
| $\mathbf{eval_{lvl_4}}$ | 0 | 0 | 0 | 0 | 256 | 32 |
| $\sum_{\mathbf{x=1}}^{\mathbf{lvl_{nr}}} \mathbf{eval_{lvl_x}}$ | 16 | 80 | 288 | 256 | 1104 | 12400 |

Figure 2 shows the PSD schematic for *configuration* 4 defined in Table 2. The levels referred below are identified on the left side of the figure. The detection process starts from the root of the tree on level $\mathbf{lvl_0} = 9$. The partial symbol vector is empty on this level.

One of the key features of the PSD algorithm is the tree traversal process. That means that instead of evaluating the path metrics $M(\mathbf{s_8^{8<j>}})$ of partial symbol vectors $\mathbf{s_8^{8<j>}}$ on level 8, as done in the SD algorithm, the first node evaluation takes place at $\mathbf{lvl_1} = 6$. By expanding the root node of the tree, $\mathbf{eval_{lvl_1}} = 64$ partial symbol vectors are generated and evaluated on level $\mathbf{lvl_1} = 6$. Note, levels 8 and 7 are skipped, thus there is no symbol vector expansion and evaluation on those levels.

After evaluating the obtained partial symbol vectors $\mathbf{s_6^{8<j>}}$, a sorting is applied based on their path metrics $M(\mathbf{s_6^{8<j>}})$. The sorted symbol vectors are denoted as $\mathbf{s_6^{8<j>'}}$, and $\mathbf{s_6^{8<0>'}}$ is the partial symbol vector with the lowest metric. When moving towards to the next level $\mathbf{lvl_2} = 4$, the $\mathbf{exp_{lvl_1}} = 4$ best metric partial symbol vectors are selected and expanded from the previous level $\mathbf{lvl_1} = 6$. As a result, the partial symbol vectors $\mathbf{s_4^{8<j>}}$ are generated.

If the inequality $\mathbf{M}(\mathbf{s_{lvl_x}^{N<off_{lvl_x}>}}) < d^2$ does not hold, then instead of increasing the corresponding offset $\mathbf{off_{lvl_x}}$ the search is stopped on that level and the offset's value is updated to 0 and the search is continued on $\mathbf{lvl_{x-1}}$. The search can be stopped on a specific level because the partial symbol vectors are sorted by their path metric. Thus if $M(\mathbf{s_{lvl_x}^{N<j>}}) > d^2$ then the remaining partial symbol vectors will have a higher path metric.

The selection, expansion, evaluation and sorting steps discussed above are repeated until the last level $\mathbf{lvl_3} = 1$ is reached. Upon reaching the last level, the symbol vector with the lowest metric has to be found. At level $\mathbf{lvl_3} = 1$, instead of sorting, a minimum search is performed. If a symbol vector $\mathbf{s_1^8}$ with the lowest metric satisfies the condition $\mathbf{M}(\mathbf{s_1^8}) < d^2$, then a new ML candidate has been found. If an ML candidate already exists from a previous iteration then it is compared with the new candidate and the one with the smaller metric will become the new solution $\mathbf{s_{ml}} = \mathbf{s_1^8}$ and the sphere radius is adjusted. The further flow of the detection process is similar to the flow of a SD algorithm.

By sorting on every stage the lowest path metric partial symbol vectors are found and the search is continued by expanding these. With this greedy strategy, where on each stage locally

optimal choices are made, an approximate of the global optimal solution may be found, thus the updated radius metric significantly reduces the search space. This is the reason why the initial condition $d^2 = \infty$ is admissible.

Algorithm 3 gives a detailed and precise description of the PSD algorithm. To make the comparison of SD and PSD algorithms as easy as possible the same notation is used in Algorithms 1 and 3. Both algorithms are divided into three main procedures: (i) *Definition and Initialization of Variables*, (ii) control of tree *Traversal Process* and (iii) *Expand and Evaluate* the tree nodes. The main differences between the SD and PSD algorithms are highlighted in Table 3.

In the *Definition and Initialization of Variables* procedure the main steps are as follows: (i) memory allocation for buffers on different levels, (ii) generating data for the first buffer and (iii) starting the tree traversal process. As shown in Table 3, the number of buffers is equal with the levels of the tree and each buffer has a constant size that is equal with the number of symbols in the symbol set in the SD algorithm. In the PSD algorithm, the number of buffers is equal to $\mathbf{lvl_{nr}}$ and the size of buffers depends on both $\mathbf{lvl_x}$ and $\mathbf{exp_{lvl_x}}$ parameters.

The *Traversal Process* procedure controls the tree traversal and in case of finding a leaf node with smaller path metric than found previously it updates the radius. The traversal process is implemented in a very different manner in the PSD and SD algorithms. While the breadth traversal of the tree, controlled by the offset variables $\mathbf{off_{lvl_x}}$, is always one in the SD algorithm, the PSD algorithm changes the offset variables based on the number of paths chosen on a specific level as follows from $\mathbf{off_{lvl_x}} \leftarrow \mathbf{off_{lvl_x}} + \mathbf{exp_{lvl_x}}$. The depth traversal of the tree is controlled by the parameters $\mathbf{lvl_x}$. While in the SD algorithm the difference between consecutive levels is always one, i.e., $\mathbf{lvl_x} - \mathbf{lvl_{x+1}} = 1$, the PSD can skip levels if $\mathbf{lvl_x} - \mathbf{lvl_{x+1}} > 1$. Using this technique the leaf nodes can be reached faster.

The *Expand and Evaluate* procedure is responsible for generating the new partial symbol vectors and to evaluate their metric. During the expansion of a tree node their child nodes are defined, i.e., the partial symbol vector denoting the tree node is updated with new symbols that are representing the child nodes. The evaluation of a partial symbol vector is the calculation of its path metric. A detailed description of this process is given in Section 4.3. Depending on the parameters chosen, the amount of newly expanded and evaluated partial symbol vectors can be significantly higher in the PSD algorithm as that in the SD one. More details are given in Table 3. Since different nodes can be expanded and evaluated independently from each other, this work can be done in parallel. Because the generated work can be controlled with well defined parameters the PSD algorithm can be adjusted to several computing platforms.

### *4.3. The main building blocks of the PSD algorithm*

The operation principle and structure of the PSD algorithm has been discussed in the previous section. All computations done on one level in the PSD algorithm shown in Fig. 2 are performed by the *Expand and Evaluate Pipeline* (EEP) depicted in Fig. 3. First a detailed description of EEP is given then the iterative implementation of PSD algorithm with EEP blocks is discussed. For a detailed description of variables used by EEP refer to Table 1.

The stages of the EEP are as follows: (i) preparation of data sets for the partial symbol vectors, referred to as *Preparatory Block*, (ii) preparation of partial symbol vectors, referred to as *Selecting, Mapping and Merging Block*, (iii) metric calculation for each partial symbol vector, referred to as *Path Metric Evaluation Block*, (iv) sorting based on the calculated path metrics or finding the symbol vector with the smallest path metric, referred to as *Searching or Sorting Block*.

The operation principle of the EEP is given in the following subsections:

**Algorithm 3** PSD algorithm for estimating $\mathbf{s_{ml}} = (s_1, s_2, \cdots, s_N)$

**Require:** $\hat{\mathbf{s}}, \mathbf{R}, |\Omega|, \mathbf{lvl_{nr}}, \mathbf{lvl_{0,1,2,\cdots,lvl_{nr}}}, \mathbf{exp_{lvl_0,lvl_1,\cdots,lvl_{nr-1}}}, \mathbf{tt}$

**procedure** DEFINITION AND INITIALIZATION OF VARIABLES
  **for** $j = 1$ **to** $\mathbf{lvl_{nr}}$ **do**
    $\mathbf{eval_{lvl_j}} \leftarrow \mathbf{exp_{lvl_{j-1}}} \cdot |\Omega|^{\mathbf{lvl_{j-1}} - \mathbf{lvl_j}}$ ▷ Number of partial symbol vector evaluations on level $\mathbf{lvl_j}$
    Let $\mathbf{buf_{lvl_j}}[\mathbf{eval_{lvl_j}}] = \{\}$ ▷ Denote an empty buffer of size $\mathbf{eval_{lvl_j}}$ for level $\mathbf{lvl_j}$
    $\mathbf{off_{lvl_j}} \leftarrow 0$ ▷ Offset of processing on level $\mathbf{lvl_j}$ for buffer $\mathbf{buf_{lvl_j}}$
  **end for**
  $\mathbf{buf_{lvl_1}} \leftarrow$ EXPAND AND EVALUATE($\{()\}$) ▷ Expand the root node () of the tree and update $\mathbf{buf_{lvl_1}}$
  TRAVERSAL PROCESS($i \leftarrow 2$)
**end procedure**
**procedure** TRAVERSAL PROCESS($i$)
  **while** $i > 1$ **do**
    **if** $\mathbf{off_{lvl_{i-1}}} < \mathbf{eval_{lvl_{i-1}}}$ **then**
      **if** $M(\mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}>}}_{\mathbf{lvl_{i-1}}}) < d^2$ **then** ▷ Where $\mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}>}}_{\mathbf{lvl_{i-1}}}$ is the element of $\mathbf{buf_{lvl_{i-1}}}$ at index $\mathbf{off_{lvl_{i-1}}}$
        **if** $i = \mathbf{lvl_{nr}}$ **then**
          $\mathbf{s'_{ml}} \leftarrow$ EXPAND AND EVALUATE($\{\mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}>}}_{\mathbf{lvl_{i-1}}}, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+1>}}_{\mathbf{lvl_{i-1}}}, \cdots, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+(exp_{lvl_{i-1}}-1)>}}_{\mathbf{lvl_{i-1}}}\}$)
          $d^2_{temp} \leftarrow \|\mathbf{R}(\mathbf{s'_{ml}} - \hat{\mathbf{s}})\|^2$
          **If** $d^2_{temp} < d^2$ **then** $d^2 \leftarrow d^2_{temp}$ and $\mathbf{s_{ml}} \leftarrow \mathbf{s'_{ml}}$ **end if**
          $i \leftarrow i - 1$
        **else**
          $\mathbf{buf_{lvl_i}} \leftarrow$ EXPAND AND EVALUATE($\{\mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}>}}_{\mathbf{lvl_{i-1}}}, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+1>}}_{\mathbf{lvl_{i-1}}}, \cdots, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+(exp_{lvl_{i-1}}-1)>}}_{\mathbf{lvl_{i-1}}}\}$)
          $\mathbf{off_{lvl_{i-1}}} \leftarrow \mathbf{off_{lvl_{i-1}}} + \mathbf{exp_{lvl_{i-1}}}$, $i \leftarrow i + 1$
        **end if**
      **else**
        $\mathbf{off_{lvl_{i-1}}} \leftarrow 0$, $i \leftarrow i - 1$
      **end if**
    **else**
      $\mathbf{off_{lvl_{i-1}}} \leftarrow 0$, $i \leftarrow i - 1$
    **end if**
  **end while**
**end procedure**
**procedure** EXPAND AND EVALUATE($\{\mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}>}}_{\mathbf{lvl_{i-1}}}, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+1>}}_{\mathbf{lvl_{i-1}}}, \cdots, \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+(exp_{lvl_{i-1}}-1)>}}_{\mathbf{lvl_{i-1}}}\}$) ▷ The input is the array of partial symbol vectors to be expanded
  **for** $n = 0$ **to** $\lceil \mathbf{eval_{lvl_i}}/\mathbf{tt} \rceil - 1$ **do**
    $ind \leftarrow \mathbf{t^k_{id}} + n \cdot \mathbf{tt}$
    $\mathbf{vt_{lvl_i}} \leftarrow (\mathbf{t^k_{id}} + n * \mathbf{tt}) \mod |\Omega|^{(\mathbf{lvl_{i-1}} - \mathbf{lvl_i})}$ ▷ Virtual thread identifier based on Eq. 9
    $\mathbf{vb_{lvl_i}} \leftarrow \lfloor (\mathbf{t^k_{id}} + n * \mathbf{tt}) / |\Omega|^{(\mathbf{lvl_{i-1}} - \mathbf{lvl_i})} \rfloor$ ▷ Virtual block identifiers based on Eq. 10
    $\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{i-1}}} = \mathbf{s}^{\mathbf{N<off_{lvl_{i-1}}+vb_{lvl_i}>}}_{\mathbf{lvl_{i-1}}} \leftarrow \mathbf{vb_{lvl_i}}$ ▷ Select partial symbol vector $\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{i-1}}}$ from the input array based on $\mathbf{vb_{lvl_i}}$
    $\mathbf{s}^{(\mathbf{lvl_{i-1}}-1)}_{\mathbf{lvl_i}} = (s_{lvl_i}, \cdots, s_{(lvl_{i-1}-2)}, s_{(lvl_{i-1}-1)}) \leftarrow \mathbf{vt_{lvl_i}}$ ▷ Create partial symbol vector $\mathbf{s}^{(\mathbf{lvl_{i-1}}-1)}_{\mathbf{lvl_i}}$ based on $\mathbf{vt_{lvl_i}}$
    $\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_i}} \leftarrow (s_{lvl_i}, \cdots, s_{(lvl_{i-1}-2)}, s_{(lvl_{i-1}-1)}, \mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{i-1}}}) = (\mathbf{s}^{(\mathbf{lvl_{i-1}}-1)}_{\mathbf{lvl_i}}, \mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{i-1}}})$ ▷ Merge $\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{i-1}}}$ and $\mathbf{s}^{(\mathbf{lvl_{i-1}}-1)}_{\mathbf{lvl_i}}$
    $\mathbf{buf_{lvl_i}}[ind] = \mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_i}}$
  **end for**
  **if** $\mathbf{lvl_i} = 1$ **then**
    **return** $\mathbf{s'_{ml}}$, which is the minimum path metric symbol vector in $\mathbf{buf_{lvl_i}}$
  **else**
    **return** $\mathbf{buf_{lvl_i}}$, where the partial symbol vectors are sorted based on the path metric $M(\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_i}})$
  **end if**
**end procedure**

Table 3: Comparison of the PSD and SD algorithms

| | *Definition and Initialization of Variables* | |
|---|---|---|
| | Number of buffers used | Accumulated buffer size |
| SD | $N$ | $N \cdot \lvert\Omega\rvert$ |
| PSD | $0 < \mathbf{lvl_{nr}} \leq N$ | $\sum_{\mathbf{x=1}}^{\mathbf{lvl_{nr}}} \mathbf{exp_{lvl_{x-1}}} \cdot \lvert\Omega\rvert^{(\mathbf{lvl_{x-1}}-\mathbf{lvl_x})}$ |
| | *Traversal process* | |
| | Horizontal traversal | Vertical traversal |
| SD | $\mathbf{off_x} \leftarrow \mathbf{off_x} + 1$ | $\mathbf{lvl_x} - \mathbf{lvl_{x+1}} = 1$ |
| PSD | $\mathbf{off_{lvl_x}} \leftarrow \mathbf{off_{lvl_x}} + \mathbf{exp_{lvl_x}}$ | $1 \leq \mathbf{lvl_x} - \mathbf{lvl_{x+1}} \leq N$ |
| | *Expand and Evaluate* | |
| | Newly evaluated partial symbol vectors in one iteration | |
| SD | $\lvert\Omega\rvert$ | |
| PSD | $\mathbf{exp_{lvl_{x-1}}} \cdot \lvert\Omega\rvert^{(\mathbf{lvl_{x-1}}-\mathbf{lvl_x})}$ | |

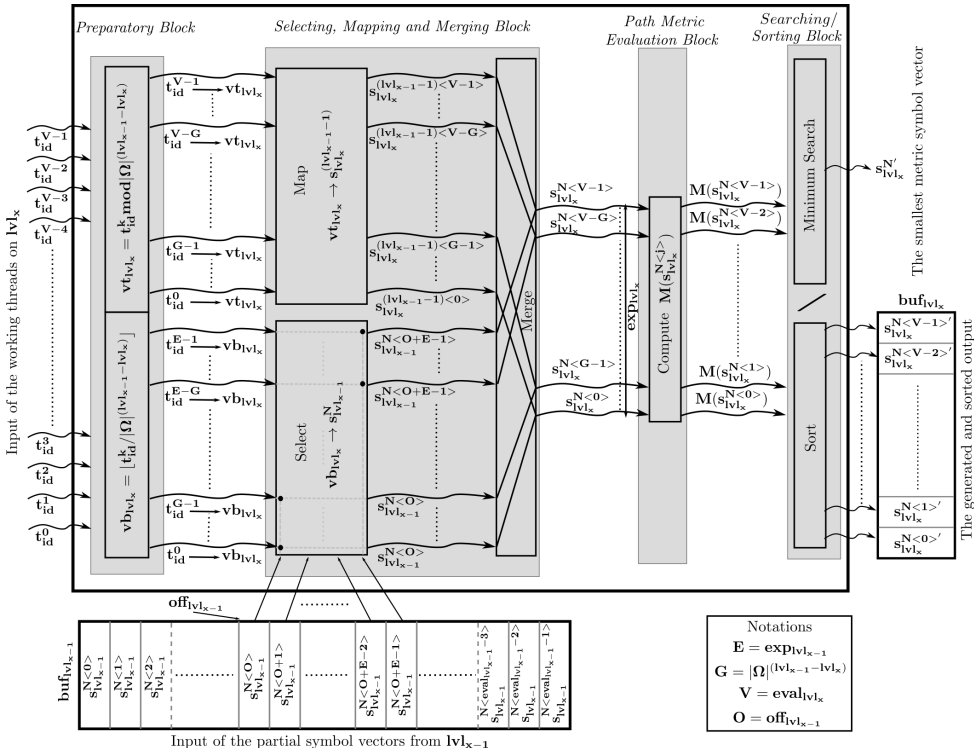



Figure 3: The *Expand and Evaluate* pipeline of the PSD algorithm.

*4.3.1. Preparatory Block*

In order to form a symbol vector $\mathbf{s}^{\mathbf{N<j>}}_{\mathbf{lvl_x}}$ on level $\mathbf{lvl_x}$ parameters such as $\mathbf{vt_{lvl_x}}$ and $\mathbf{vb_{lvl_x}}$ have to be defined. The work assigned for one thread depends on the number of symbol vectors needed to be evaluated on a given level and on the number of the threads launched. If the condition

$$\mathbf{eval_{lvl_x}} \leq \mathbf{tt} \tag{8}$$

is met then one symbol vector has to be evaluated by one thread. Otherwise one thread has to be assigned to process at most $\lceil \mathbf{eval_{lvl_x}}/\mathbf{tt} \rceil$ number of symbol vectors. A full BFS will take place on level $\mathbf{lvl_x}$ in the case when the condition $\mathbf{exp_{lvl_{x-1}}} = \mathbf{max_{lvl_{x-1}}}$ holds because all the symbol vectors on level $\mathbf{lvl_x}$ are expanded simultaneously. Assuming that $\mathbf{eval_{lvl_x}}$ is divisible by $\mathbf{tt}$, two sets are defined for each thread $\mathbf{t^k_{id}}$: (i) set $VT^k_{lvl_x}$ containing the virtual thread identifiers and (ii) set $VB^k_{lvl_x}$ containing the virtual block identifiers. The virtual identifiers are computed in the following manner:

$$\begin{aligned} VT^k_{lvl_x} = \{\mathbf{vt_{lvl_x}} | \mathbf{vt_{lvl_x}} = (\mathbf{t^k_{id}} + n \cdot \mathbf{tt}) \mod |\mathbf{\Omega}|^{(\mathbf{lvl_{x-1}} - \mathbf{lvl_x})}, \\ n = 0 : \lceil \mathbf{eval_{lvl_x}}/\mathbf{tt} \rceil - 1\}, \end{aligned} \tag{9}$$

$$\begin{aligned} VB^k_{lvl_x} = \{\mathbf{vb_{lvl_x}} | \mathbf{vb_{lvl_x}} = \lfloor (\mathbf{t^k_{id}} + n \cdot \mathbf{tt})/|\mathbf{\Omega}|^{(\mathbf{lvl_{x-1}} - \mathbf{lvl_x})} \rfloor, \\ n = 0 : \lceil \mathbf{eval_{lvl_x}}/\mathbf{tt} \rceil - 1\} \end{aligned} \tag{10}$$

where $\lfloor x \rfloor$ is the largest integer not greater than x and $\lceil x \rceil$ is the smallest integer not less than x.

Each thread has to compute its own set of identifiers for every level. This first block, referred to as *Preparatory Block*, is completed when each thread has finished computing the virtual identifiers.

*4.3.2. Selecting, Mapping and Merging Block*

In the *Selecting, Mapping and Merging* block the task is to generate symbol vectors $\mathbf{s}^{\mathbf{N<j>}}_{\mathbf{lvl_x}}$ for the level $\mathbf{lvl_x}$.

In the *Selecting* phase, $\mathbf{exp_{lvl_{x-1}}}$ number of previously evaluated symbol vectors $\mathbf{s}^{\mathbf{N}}_{\mathbf{lvl_{x-1}}}$ are selected from $\mathbf{buf_{lvl_{x-1}}}$ serving as inputs to this process. The *selection* is done based on the virtual block identifiers and the corresponding offset $\mathbf{off_{lvl_{x-1}}}$. The virtual block identifiers $\mathbf{vb_{lvl_x}}$ are computed based on Eq. 10. Each $\mathbf{vb_{lvl_x}} \in VB^k_{lvl_x}$ serves as an index of the input partial symbol vector array. The selected partial symbol vector $\mathbf{s}^{\mathbf{N<j>}}_{\mathbf{lvl_{x-1}}}$ is the element at index $j$ in the input buffer $\mathbf{buf_{lvl_{x-1}}}$ and $j = \mathbf{off_{lvl_{x-1}}} + \mathbf{vb_{lvl_x}}$.

In the *Mapping* phase the goal is to create partial symbol vectors $\mathbf{s}^{\mathbf{lvl_{x-1}-1<j>}}_{\mathbf{lvl_x}}$ based on the $\mathbf{vt_{lvl_x}} \in VT^k_{lvl_x}$ virtual thread identifiers. In order to achieve this, each $\mathbf{vt_{lvl_x}}$ will be transformed to a binary vector of length $\log_2 |\mathbf{\Omega}| \cdot (\mathbf{lvl_{x-1}} - \mathbf{lvl_x})$. Let $B$ denote the transformation of a natural number to a binary vector of size $l$

$$B : (\mathbb{N}, l \in \mathbb{N}) \rightarrow \mathbb{B}^l = \{0,1\}^l. \tag{11}$$

Let the binary vector $\mathbf{b^l}$ denote the result of transformation $B$ with inputs $\mathbf{vt_{lvl_x}}$ and $\log_2 |\mathbf{\Omega}| \cdot (\mathbf{lvl_{x-1}} - \mathbf{lvl_x})$:

$$\mathbf{b^l} = B(\mathbf{vt_{lvl_x}}, \log_2 |\mathbf{\Omega}| \cdot (\mathbf{lvl_{x-1}} - \mathbf{lvl_x})). \tag{12}$$

In vector $\mathbf{b^l}$, $(\mathbf{lvl_{x-1}} - \mathbf{lvl_x})$ number of binary groups of size $\log_2 |\Omega|$ are available. A one-to-one mapping between the binary groups and the symbol set elements is defined. Therefore, while iterating over the groups of binary elements, $\mathbf{b}_{(i\cdot\log_2|\Omega|):((i+1)\cdot\log_2|\Omega|-1)} \rightarrow s_i \in \Omega$ are selected and the partial symbol vector $\mathbf{s}_{\mathbf{lvl_x}}^{\mathbf{lvl_{x-1}-1}} = (s_{lvl_x}, s_{lvl_x+1}, \cdots, s_{(lvl_{x-1}-lvl_x)-1})$ is formed.

In the *Merging* phase the result of the selection and mapping is merged, namely, each selected vector $\mathbf{s}_{\mathbf{lvl_{x-1}}}^{\mathbf{N<j>}}$ and mapped symbol vector $\mathbf{s}_{\mathbf{lvl_x}}^{\mathbf{lvl_{x-1}-1<j>}}$ is merged as

$$\begin{aligned}\mathbf{s}_{\mathbf{lvl_x}}^{\mathbf{N<j>}} &= (\mathbf{s}_{\mathbf{lvl_x}}^{\mathbf{lvl_{x-1}-1<j>}}, \mathbf{s}_{\mathbf{lvl_{x-1}}}^{\mathbf{N<j>}}) \\ &= (s_{lvl_x}, \cdots s_{lvl_{x-1}-1}, s_{lvl_{x-1}}, \cdots, s_N).\end{aligned} \tag{13}$$

*4.3.3. Path Metric Evaluation Block*

In the *Path Metric Evaluation* block, the metric of created partial symbol vectors is computed. This is one of the most time-consuming steps, but the path metric is computed in parallel by several threads. Consequently, a significant speed-up can be achieved. Further speed-up can be achieved if the path metric of partial symbol vectors $M(\mathbf{s}_{\mathbf{lvl_{x-1}}}^{\mathbf{N}})$ computed previously are stored and only the contribution of the newly created partial symbol vectors $M(\mathbf{s}_{\mathbf{lvl_x}}^{\mathbf{lvl_{x-1}-1}})$ to the overall metric is computed.

*4.3.4. Searching or Sorting Block*

The last block of the EEP is one of the most important stages during the detection. Depending on the level of processing either sorting or a minimum search is applied. The minimum search is applied only when the detection has reached the last processing level, while sorting is applied on all other levels. The use of the two algorithms is motivated by the lower complexity required by the minimum search algorithm. Recall, when the last level of the tree is reached then the task is to find the symbol vector with the smallest path metric.

As discussed in Subsection 4.2, the complexity of the algorithm can be reduced by adjusting the radius of the sphere after finding a leaf node of the tree. Sorting the buffers based on the path metric of symbol vectors and applying the hybrid searching strategy makes the finding of a leaf node possible after a few iterations. Several parallel algorithms exist in the literature that can exploit the parallel architectures in order to sort and search arrays [25], thus the high computational power of these devices can be also utilized at this stage.

*4.4. Application of the EEP pipeline*

Recall, the EEP depicted in Fig. 3 implements one level of PSD algorithm. To implement the entire PSD algorithm the EEP is used in an iterative manner as shown in Fig. 4. Note, depending on the processing level the EEP outputs are (i) the sorted partial symbol vectors placed in $\mathbf{buf}_{\mathbf{lvl_x}}$ or (ii) the symbol vector with the smallest path metric. The inputs for this process are (i) the number of threads **tt** available for the processing and (ii) $\mathbf{exp}_{\mathbf{lvl_{x-1}}}$ number of previously computed partial symbol vectors retrieved from $\mathbf{buf}_{\mathbf{lvl_{x-1}}}$. In the last stage of the EEP a candidate ML solution might be returned.

## 5. The CUDA programming model

The programming of the GP-GPU devices has became popular since Nvidia published the Compute Unified Device Architecture (CUDA) parallel programming model. Traditional CPUs

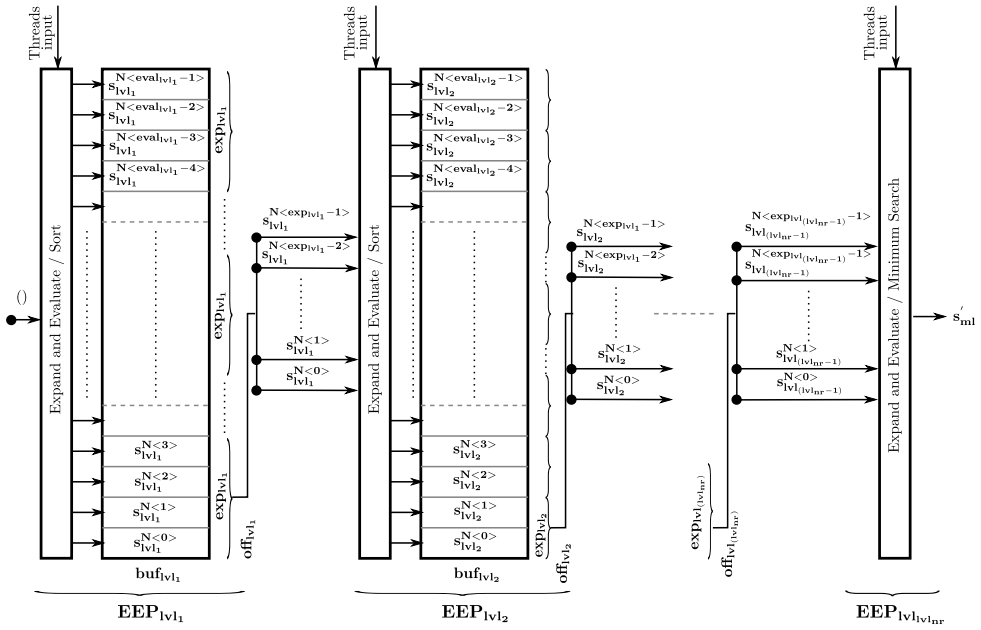



Figure 4: The iterative use of the *Expand and Evaluate* pipeline.

are able to execute only a few threads but with relatively high clock rate. In contrast GP-GPUs have parallel architecture and are able to support the execution of thousands of threads with a relatively slower speed.

An extensive description of CUDA programming and optimization techniques can be found in [26]. The main entry points of GP-GPU programs are referred to as *kernels*. These kernels are executed N times in parallel by N different *CUDA threads*. CUDA threads are grouped in *thread blocks* (TBs). The number of threads in a TB is limited, however, multiple equally-shaped TBs can be launched simultaneously. A *grid* is a collection of TBs. Either the threads in the TB or the TBs in the grid can have a one-dimensional, two-dimensional or three-dimensional ordering.

The cooperation between the threads is realized with the help of multiple memory spaces that differ in size, latency and visibility. In CUDA the following hierarchy of memory levels are defined: (i) *private*, (ii) *shared*, (iii) *global*, (iv) *constant* and (v) *texture* memory.

There are situations where specific threads have to wait for result generated by other threads. Therefore, threads within a TB can be *synchronized*. In order to continue the execution each thread has to reach the synchronization point. There is no similar mechanism to synchronize TBs in a grid. When a kernel finishes its execution it can be regarded as a global synchronization of the TBs.

The Nvidia GP-GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). The TBs of the grid are distributed to the SMs with available execution capacity by the grid management unit. An important metric of the SMs usage is *occupancy*. The occupancy metric of each SM is defined as the number of active threads divided by the maximum number of threads. Groups of 32 threads, called *warps*, are executed together. The maximum number of TBs running simultaneously on a multiprocessor are limited by the maximum number of warps or registers, or by the amount of shared memory used by the kernel.

In order to concurrently execute hundreds of threads, the SMs employ the *Single-Instruction, Multiple-Thread (SIMT)* architecture. A warp executes one common instruction at a time. In the case of branching, the warp will serially execute each branch path. In order to achieve full efficiency, divergence should be avoided. Applications manage concurrency through *streams*. A stream is a sequence of commands that are executed in order. Different streams may execute their commands out of order with respect to one another or concurrently. Thus, launching multiple kernels on different streams is also possible which can be very efficient when kernels can be launched independently from each other.

## 6. Levels of parallelism and CUDA mapping details

As described in Section 5, a grid is defined before launching a CUDA kernel. A grid may contain several TBs and each TB may contain several threads. Concurrent kernel executions are also possible for some devices using multiple streams. Hence, multiple levels of parallelism are available. The main challenge during the implementation is to well define the parallel possibilities of the system model, the parallel architecture and to make the correct bounding of these.

The *algorithm level* parallelism is the effective distribution of the work among the threads in a TB. The computationally intensive parts of the algorithm are (i) the expansion and evaluation of the symbol vectors and (ii) the sorting. The *Expand and Evaluate* procedure is highly parallel. Every thread in the thread block is working at this point. The PSD algorithm through its parameters is able to adjust the generated work, thus the algorithm can be easily adapted to different architectures.

For the sorting stage several parallel sorting algorithms can be used. In the PSD algorithm the sorting is done with the use of sorting networks [27], [25], [28]. Due to their data-independent structure, their operation sequence is completely rigid. This property makes this algorithm parallelizable for the GP-GPU architecture. The minimum search algorithm relies on the parallel scan algorithm [29].

Each TB launched is a one dimensional block with **tt** number of threads. In order to get fast detection, access time to global memory has to be minimized. A good solution is to store the heavily used $\mathbf{buf_{lvl_x}}$ arrays in the shared memory. If all $\mathbf{buf_{lvl_x}}$ buffers are stored in the shared memory then a more severe limitation may be imposed on the parameters $\mathbf{lvl_x}$ and $\mathbf{exp_{lvl_x}}$ because the size of shared memory is significantly smaller than that of the global memory. Recall, shared memory used by a TB is proportional to the sum $\sum_{\mathbf{x=1}}^{\mathbf{lvl_{nr}}} \mathbf{eval_{lvl_x}}$ of the evaluated nodes at different levels. The excessive use of shared memory can lead to occupancy degradation and, consequently, one SM can execute only a lower number of TBs at the same time. In case of GP-GPUs a good trade-off has to be found among the algorithm parameters and the resources of the SMs. Since different GP-GPUs have different memory configurations, the optimal algorithm parameters depend heavily on the device used.

Our model presented in Section 2 assumes block-fading channel where the fading process is constant for a block of symbols and changing independently from a block to another. The block of symbol vectors for which the fading process is constant is called a *fading block*. A transmitted frame of length $\mathbf{L}$ symbols is affected by $\mathbf{F}$ independent fading realizations, i.e., channel matrices, resulting in a block of length $\mathbf{l} = \lceil \mathbf{L}/\mathbf{F} \rceil$ symbols being affected by the same fading realization. It can be seen that multiple symbol vectors have to be processed simultaneously for one transmitted frame.

The *system level* parallelism is implemented by the parallel processing of fading blocks of a transmitted frame, consequently, the number of kernels launched is equal to the number of

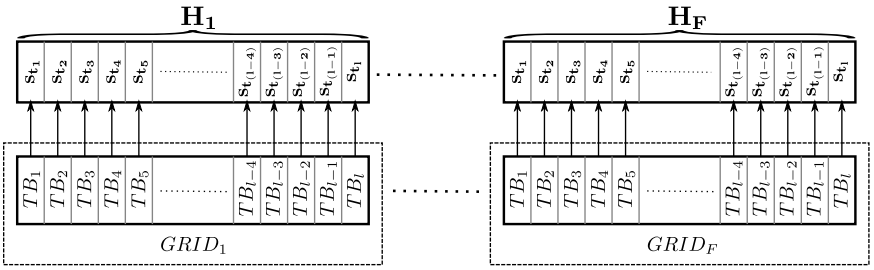



Figure 5: System level parallelism - equal distribution of the computing load.

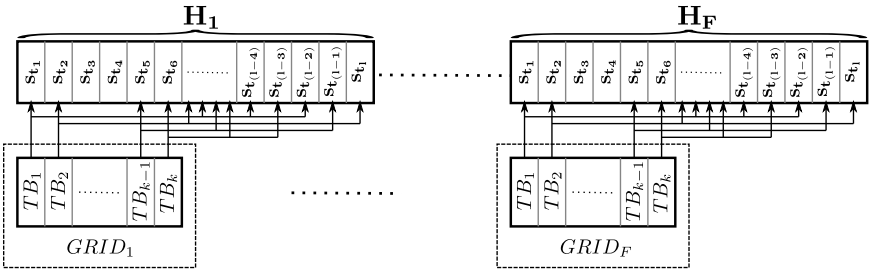



Figure 6: System level parallelism - dynamic distribution of the computing load.

independent fading blocks. Different bounding strategies among the TBs of one grid and symbol vectors of a fading block are shown in Figs. 5 and 6. In the first case the number of TBs in one grid is equal to the number of symbol vectors belonging to the same channel matrix. The drawback of this straightforward bounding is the high number of TBs because the resources of the GP-GPU will be available for a long time duration only for one kernel. Consequently, the overlapping execution of concurrent kernels will be limited. In the second case the number of TBs in a grid is significantly smaller than the number of symbol vectors in one group. The work for a TB is dynamically distributed, namely, when the detection of one symbol vector is finished, the TB evaluates the next unprocessed symbol vector. Because the detection time of the different symbol vectors may differ significantly, the number of symbol vectors to be processed by one TB is also different. Having a lower number of TBs in one grid makes possible to launch TBs from other grids if there are free GP-GPU resources. The drawback of this approach is the increased complexity of the algorithm caused by the dynamic distribution of the work among the TBs.

The *device level* parallelism in GP-GPUs is achieved by launching multiple kernels simultaneously on different streams. By exploiting the advantage of device level parallelism, a significant decrease in the computational time can be achieved. To demonstrate the importance of overlapping execution of multiple kernels, a simplified TB scheduling is shown in the following. Consider a GP-GPU with only one SM and assume that this SM is capable of running only four TBs simultaneously as shown in Fig. 7. Consider a kernel with a grid configuration of four TBs. The kernel is finished when every TB has completed its task. In this example the execution of $TB_1$ is finished at time $t_1$, afterwards the 25% of the cores are idle. The worst case is when the execution of $TB_2$ is finished because the 75% of the available cores in the SM are idle. Because of the wasted resources the overall performance is degraded. If a new TB from a different kernel could be launched after the execution of $TB_1$ is finished then the resources of the GP-GPU would

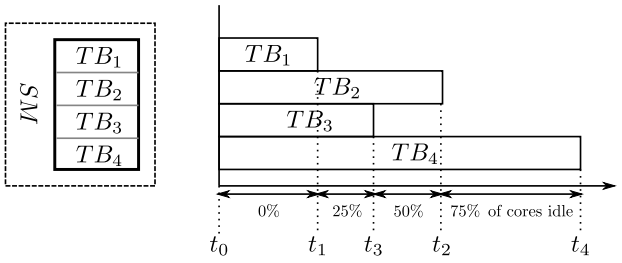



Figure 7: Thread blocks scheduling.

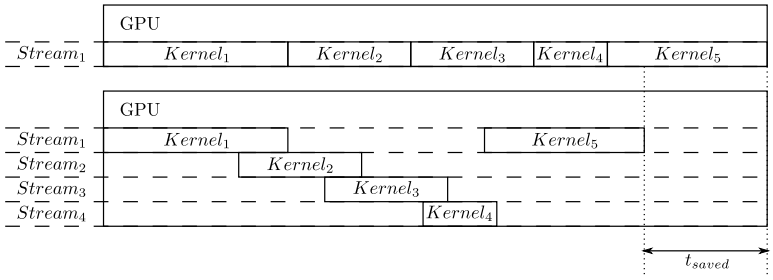



Figure 8: Scheduling of the grids using the SS and MS strategies.

be fully exploited.

The idle time of the cores can be minimized by exploiting the multi-stream features of the selected GP-GPUs. Figure 8 shows the scheduling for *single stream* (SS) and *multiple streams* (MS) execution. The SS strategy launches the kernels in succession and avoids overlapping execution. As shown in Fig. 8, MS exploits the overlapping execution of kernels and minimizes the idle time of the cores. Note, the amount of overlapping depends on the occupancy of the kernels and the number of TBs launched in each kernel. In Section 7 the performance of SS and MS strategies are compared and evaluated.

## 7. Performance results

A major issue in ML detection is handling of its varying complexity. Channel matrices with high condition numbers or low SNRs may increase the complexity of the algorithm, consequently, the running time of different kernels may differ significantly. In order to evaluate the average detection throughput of the PSD algorithm, $\mathbf{F} = 8000$ independent fading realizations with $\mathbf{l} = 600$ symbol vectors for each fading realization were generated and evaluated. The average throughput is determined based on $\sim 5 \cdot 10^6$ processed symbol vectors.

Table 4: Kepler GK104 architecture main characteristics

| CUDA cores | Threads / Warp | Max warps / SM | Max threads / SM | Max TBs / SM | Max registers / thread | Max threads / TB | Max shared memory / SM |
|---|---|---|---|---|---|---|---|
| 1536 | 32 | 64 | 2048 | 16 | 63 | 1024 | 48 Kbytes |

Table 5: Results of parameter optimization obtained for 2*x*2 and 4*x*4 MIMO systems for SNR = 20 dB.

| $N$ | $M$ | $\|\Omega\|$ | $\mathbf{lvl_{nr}}$ | $\mathbf{lvl_0}$ | $\mathbf{lvl_1}$ | $\mathbf{lvl_2}$ | $\mathbf{lvl_3}$ | $\mathbf{lvl_4}$ | $\mathbf{exp_{lvl_0}}$ | $\mathbf{exp_{lvl_1}}$ | $\mathbf{exp_{lvl_2}}$ | $\mathbf{exp_{lvl_3}}$ | **tt** | $Mbit/s$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 1 | 5 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 16 | 163 |
| 2 | 2 | 4 | 1 | 5 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 128 | 268 |
| 2 | 2 | 8 | 2 | 5 | 3 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 64 | 153 |
| 4 | 4 | 2 | 2 | 9 | 7 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 64 | 169 |
| 4 | 4 | 4 | 3 | 9 | 6 | 4 | 1 | 0 | 1 | 4 | 1 | 0 | 64 | 121 |
| 4 | 4 | 8 | 4 | 9 | 7 | 5 | 3 | 1 | 1 | 4 | 4 | 2 | 128 | 24 |

The performance of PSD algorithm was evaluated on a GeForce GTX690 GP-GPU built on Kepler GK104 [30] architecture. The main parameters of the GK104 architecture are given in Table 4.

It was shown in Sec. 4 that a good trade-off has to be found between the algorithm parameters and the resources of SMs. The most important quality measure of a radio link is its SNR. The throughput achieved by the PSD algorithm proposed here are compared to that of published in the literature [31] - [32] where results are given for SNR = 20 dB. In order to make a fair comparison with the results presented in the literature we optimized the algorithm parameters for 20 dB SNR. Table 5 summarizes the result of the parameter optimization for different MIMO systems.

Parameters shown in Table 5 were applied here for the throughput measurements. In order to measure the efficiency of MS execution presented in Fig. 6 grids with 64 TBs were used and the TBs were launched on 32 streams. Average detection throughputs achieved by the SS and MS strategies are shown in Fig. 9 for $2 \times 2$ and $4 \times 4$ MIMO systems where the size of the symbol set was set to $|\mathbf{\Omega}| = 2, 4, 8$.

Figure 9 shows that the average throughputs do not depend on SNR for 2x2 MIMO system with $|\mathbf{\Omega}| = 2$ and 4. This is due to the low number of symbol vectors to be evaluated on the last tree level. That low number of symbol vectors can be processed simultaneously without computing any partial symbol vector. The throughput is higher for $|\mathbf{\Omega}| = 4$ because the number of transmitted bits is doubled compared to the case of $|\mathbf{\Omega}| = 2$ and the processing time required is not significantly higher. With the increasing number of antennas and symbol set size, the detection throughput is getting lower. This is mainly caused by the exponential increase of total number of nodes. A $15\% - 30\%$ increase in average throughput is achieved by enabling the overlapping execution of kernels on multiple streams.

Another important metric of the SD algorithms is the average number of expanded nodes. Figure 10 compares the average number of visited nodes in each thread when the SD, PSD and Automatic Sphere Decoder (ASD) algorithms are used. We compare our results with that of the ASD algorithm because it was shown in [33] that ASD expands the minimum number of the nodes as the number of antennas or size of symbol set are increasing. Recall, the SD and ASD algorithms are sequential algorithms, consequently, the tree search can be performed by only a single thread and there is no chance to expand and evaluate multiple nodes simultaneously. In contrast, the PSD algorithm is able to distribute the work among multiple threads. However, the total number of symbol vectors to be expanded and evaluated is higher. Table 5 shows the number **tt** of total threads used for different MIMO systems.

Figure 10 shows that the PSD algorithm requires a significantly lower average number of symbol vectors to be processed by one thread in every MIMO configuration. For a $4 \times 4$ MIMO system and a symbol set of size $|\Omega| = 8$ the signal space has $1.6 \cdot 10^7$ symbol vectors. If the SNR = 5 dB the PSD expands $\sim 260$ nodes per thread while the ASD expands $\sim 7500$ nodes per thread. As a result the total work of a thread running the PSD algorithm is reduced by 97%.

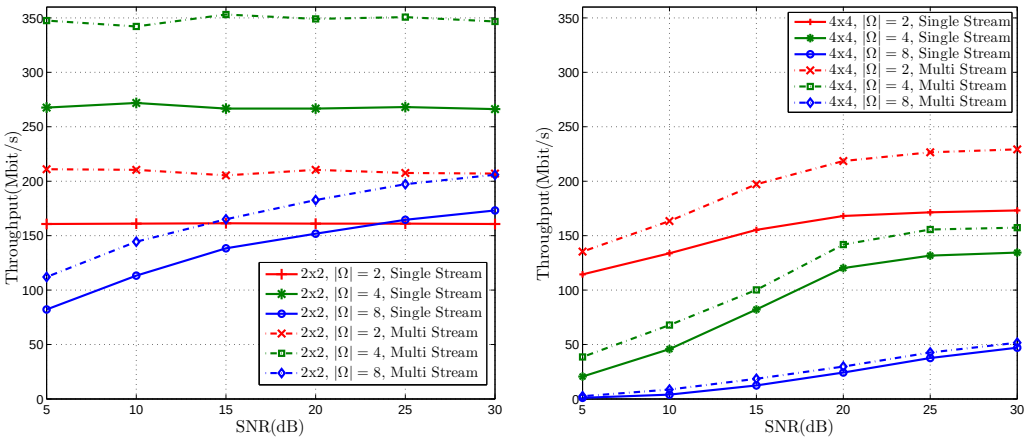



Figure 9: PSD average throughput for (a) 2x2 and (b) 4x4 MIMO obtained with SS and MS strategies.

Table 6: Throughput comparison of existing ML algorithms.

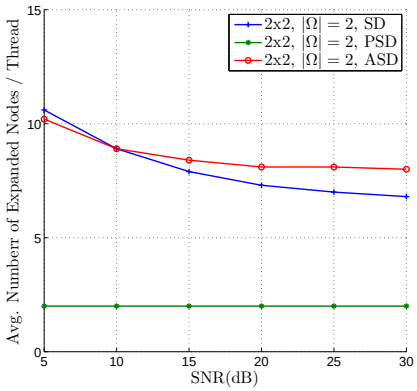
| Reference | [31] | [34] | [35] | [32] | PSD | PSD |
|---|---|---|---|---|---|---|
| BER performance | ML | | | | | |
| Antennas | 4x4 | | | | | |
| Symbol set size | $\|\Omega\| = 4$ | $\|\Omega\| = 2$ | $\|\Omega\| = 4$ | $\|\Omega\| = 4$ | $\|\Omega\| = 4$ | $\|\Omega\| = 2$ |
| Technology | ASIC | ASIC | ASIC | FPGA | GPU | GPU |
| Throughput | 38 Mbps | 50 Mbps | 73 Mbps @SNR = 20 dB | 81.5 Mbps @SNR = 20 dB | 141 Mbps @SNR = 20 dB | 218 Mbps @SNR = 20 dB |

If the SNR = 20 dB, the work of a thread running the PSD algorithm is reduced by 95%. The distribution of work makes the PSD algorithm efficient despite the fact that the total number of symbol vectors to be processed is higher than that of in the SD and ASD algorithms. The processing of more symbol vectors in total can be regarded as the price of enabling the use of many-core architectures.

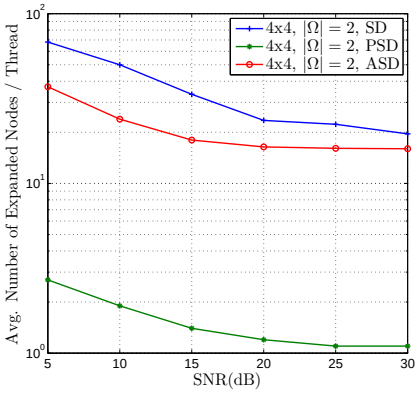
Throughput is the most important performance measure of a digital demodulator. Recall, the ML detection assures the best BER in Additive White Gaussian Noise (AWGN) channels. Table 6 compares the throughput of PSD algorithm running on the GeForce GTX690 GP-GPU proposed here with other alternatives published in the literature [31], [32], [34], [35]. Only those results are considered here that focus on finding the optimal solution. Table 6 shows that the PSD algorithm proposed here outperforms all of them.

## 8. Conclusions

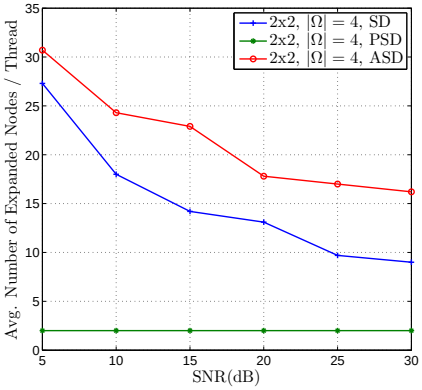
This work aimed to enable the efficient usage of multi-core and many-core architectures in wireless MIMO communications systems by solving the optimal ML detection problem on a GP-GPU platform. In the literature many near-ML algorithms exist, however we imposed the condition of finding the optimal ML solution. Because the complexity of the ML detection grows exponentially with both the size of the signal set and the number of antennas, we wanted to use modern MPAs to solve this problem. The main drawback of the original SD algorithm



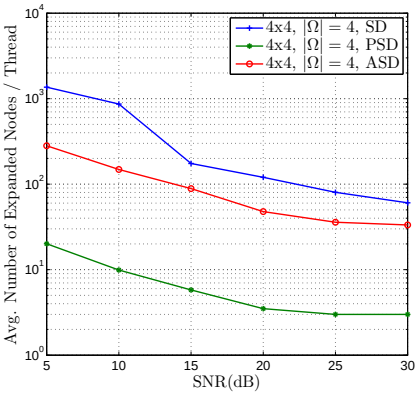

(a) 2x2 MIMO and $|\boldsymbol{\Omega}| = 2$



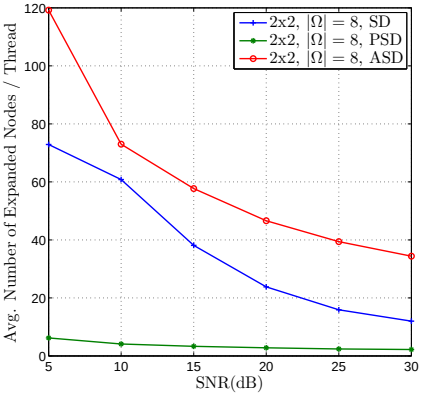

(b) 4x4 MIMO and $|\boldsymbol{\Omega}| = 2$



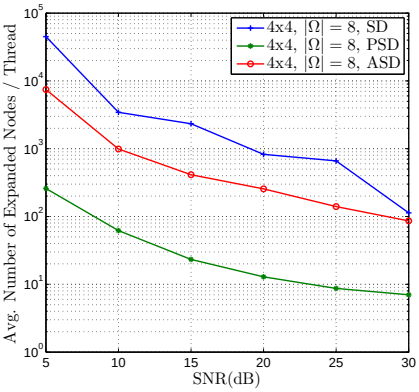

(c) 2x2 MIMO and $|\boldsymbol{\Omega}| = 4$




(d) 4x4 MIMO and $|\boldsymbol{\Omega}| = 4$




(e) 2x2 MIMO and $|\boldsymbol{\Omega}| = 8$




(f) 4x4 MIMO and $|\boldsymbol{\Omega}| = 8$

Figure 10: Average Number of Expanded Nodes per Thread for different MIMO systems and signal sets in the SD, ASD and PSD algorithms.

is its sequential nature, thus running it on MPAs is very inefficient. In order to overcome the limitation of the SD algorithm, we designed and implemented the new parallel SD algorithm.

The new PSD algorithm is based on a novel hybrid tree traversal where algorithm parallelism is achieved by the alternating use of DFS and BFS strategies, referred to as hybrid tree search, combined with path metric based sorting on the intermediate stages. The novel hybrid tree search algorithm makes possible the simultaneous processing of high number of symbol vectors and overcomes the problem of the initial radius. The most important feature of the new PSD algorithm is that it assures a good balance between the total number of processed symbol vectors and the extent of parallelism by adjusting its parameters. In modern MPAs complex memory hierarchies are available, enabling the use of smaller but faster memories. The PSD algorithm is able to adjust its memory requirements by the algorithm parameters and the allocated memory is kept constant during the processing. The above mentioned properties of the PSD algorithm makes it suitable for a wide range of parallel computing devices. In contrast, the sequential SD algorithm can not fully exploit the resources of a parallel architecture because the generated computational load is always constant.

We identified further levels of parallelism: (i) a higher system level parallelism and (ii) a GPGPU specific device level parallelism. The system level parallelism is implemented by parallel processing of the fading blocks in a transmitted frame. We considered the (i) equal and (ii) dynamic computing load distribution strategies and we showed that by applying the dynamic distribution of the computing load in a multi-stream environment a $15-30\%$ boost in the average throughput is achieved.

Addition to the new PSD algorithm its efficient implementation on a GeForce GTX 690 GPGPU also has been demonstrated. As shown in Fig. 9, the peak throughput achieved by the PSD algorithm proposed was as high as 350 Mbit/s and 218 Mbit/s for a 2x2 MIMO system using a symbol set $|\mathbf{\Omega}| = 4$ and a 4x4 MIMO system implemented with a symbol set $|\mathbf{\Omega}| = 2$ respectively. Throughputs published in literature for 4x4 MIMO system for $|\Omega| = 2$ and 4 presented in Table 3 varies from 38 to 81.5 Mbit/s. The 141 and 218 Mbit/s throughputs achieved by the new PSD algorithm offers a significant performance improvement.

The average number of expanded nodes per thread have been also analyzed and it has been shown that the PSD algorithm is doing much less processing in one thread compared to the SD and ASD algorithms. For 4x4 MIMO systems the work of a thread, i.e., the number of expanded nodes, has been reduced by $90-97\%$. Consequently, the goal of efficient work distribution was achieved.

## Acknowledgments

This work has been supported by the grants TÁMOP-4.2.1./B-11/2/KMR-2011-0002, TÁMOP-4.2.2/B-10/1-2010-0014 from the Hungarian Government, K84045 from the Hungarian Research Fund (OTKA), CICYT TEC2012-38142-CO4 from the Spanish Government and PROMETEO/2009/013 project from the Generalitat Valenciana.

## References

[1] E. Biglieri, R. Calderbank, A. Constantinides, A. Goldsmith, A. Paulraj, H. V. Poor, MIMO Wireless Communications, Cambridge University Press, New York, NY, USA, 2007.

[2] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, A. Vidal, Fully Parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector, Vehicular Technology, IEEE Transactions on 61 (8) (2012) 3796–3800.

[3] L. Barbero, J. Thompson, Fixing the Complexity of the Sphere Decoder for MIMO Detection, Wireless Communications, IEEE Transactions on 7 (6) (2008) 2131–2142.
[4] M. Khairy, C. Mehlfuhrer, M. Rupp, Boosting sphere decoding speed through Graphic Processing Units, in: Wireless Conference (EW), 2010 European, IEEE, 2010, pp. 99–104.
[5] M. El-Khamy, M. Medra, H. M. ElKamchouchi, Reduced complexity list sphere decoding for mimo systems, Digital Signal Processing (0) (2013) –. doi:http://dx.doi.org/10.1016/j.dsp.2013.10.023.
[6] C.-E. Chen, W.-H. Chung, Computationally efficient near-optimal combined antenna selection algorithms for v-blast systems, Digital Signal Processing 23 (1) (2013) 375 – 381.
[7] G. Romano, D. Ciuonzo, P. S. Rossi, F. Palmieri, Low-complexity dominance-based sphere decoder for mimo systems, Signal Processing 93 (9) (2013) 2500 – 2509.
[8] M. Damen, H. El Gamal, G. Caire, On maximum-likelihood detection and the search for the closest lattice point, Information Theory, IEEE Transactions on 49 (10) (2003) 2389–2402.
[9] J. H. Conway, N. J. A. Sloane, E. Bannai, Sphere-packings, lattices, and groups, Springer-Verlag, Inc., New York, NY, USA, 1987.
[10] A. Murugan, H. El Gamal, M. Damen, G. Caire, A unified framework for tree search decoding: rediscovering the sequential decoder, Information Theory, IEEE Transactions on 52 (3) (2006) 933–953.
[11] E. Agrell, T. Eriksson, A. Vardy, K. Zeger, Closest point search in lattices, Information Theory, IEEE Transactions on 48 (8).
[12] D. Micciancio, S. Goldwasser, Complexity of lattice problems: a cryptographic perspective, The Kluwer international series in engineering and computer science, Kluwer Academic, 2002.
[13] U. Fincke, M. Pohst, Improved Methods for Calculating Vectors of Short Length in a Lattice, Including a Complexity Analysis, Mathematics of Computation 44 (170) (1985) 463–471.
[14] C. P. Schnorr, M. Euchner, Lattice basis reduction: Improved practical algorithms and solving subset sum problems, Mathematical Programming 66 (1994) 181–199.
[15] H. Vikalo, B. Hassibi, On the sphere-decoding algorithm II. Generalizations, second-order statistics, and applications to communications, Signal Processing, IEEE Transactions on 53 (8) (2005) 2819–2834.
[16] B. Hassibi, H. Vikalo, On the sphere-decoding algorithm I. Expected complexity, Signal Processing, IEEE Transactions on 53 (8).
[17] J. Jalden, B. Ottersten, On the complexity of sphere decoding in digital communications, Signal Processing, IEEE Transactions on 53 (4) (2005) 1474 – 1484.
[18] J. Fink, S. Roger, A. Gonzalez, V. Almenar, V. Garcia, Complexity assessment of sphere decoding methods for MIMO detection, in: Signal Processing and Information Technology (ISSPIT), 2009 IEEE International Symposium on, 2009, pp. 9 –14.
[19] M. Myllylä, M. Juntti, J. R. Cavallaro, Implementation aspects of list sphere decoder algorithms for mimo-ofdm systems, Signal Processing 90 (10) (2010) 2863 – 2876. doi:http://dx.doi.org/10.1016/j.sigpro.2010.04.014.
URL http://www.sciencedirect.com/science/article/pii/S0165168410001611
[20] P. van Emde-Boas, Another NP-complete partition problem and the complexity of computing short vectors in a lattice, Report. Department of Mathematics. University of Amsterdam, Department, Univ., 1981.
[21] M. Pohst, On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications, ACM SIGSAM Bulletin 15 (1) (1981) 37–44.
[22] E. Viterbo, E. Biglieri, A universal decoding algorithm for lattice codes, in: 14 Colloque sur le traitement du signal et des images, FRA, 1993, GRETSI, Groupe d'Etudes du Traitement du Signal et des Images, 1993.
[23] E. Viterbo, J. Boutros, A universal lattice code decoder for fading channels, Information Theory, IEEE Transactions on 45 (5) (1999) 1639 –1642.
[24] K. Lai, J. Jia, L. Lin, Hybrid Tree Search Algorithms for Detection in Spatial Multiplexing Systems, Vehicular Technology, IEEE Transactions on (99) (2011) 1–1.
[25] P. Kipfer, R. Westermann, GPU Gems, Vol. 2, Addison Wesley Professional, 2005, Ch. 46, pp. 733–746.
[26] NVIDIA Corporation, CUDA C Programming Guide (2012).
[27] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems), Addison-Wesley Professional, 2005.
[28] K. E. Batcher, Sorting networks and their applications, 1968, pp. 307–314.
[29] H. Nguyen, GPU Gems 3, 1st Edition, Addison-Wesley Professional, 2007.
[30] NVIDIA Corporation, GTX 680 Kepler (GK104) Whitepaper (2012).
[31] D. Garrett, L. Davis, S. ten Brink, B. Hochwald, G. Knagge, Silicon complexity for maximum likelihood MIMO detection using spherical decoding, Solid-State Circuits, IEEE Journal of 39 (9) (2004) 1544–1552.
[32] X. Huang, C. Liang, J. Ma, System architecture and implementation of MIMO sphere decoders on FPGA, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 16 (2) (2008) 188–197.
[33] Su, K., Efficient Maximum Likelihood Detection for Communication over Multiple Input Multiple Output Channels, Master's thesis, University of Cambridge (2005).

[34] N. Felber, W. Fichtner, A. Burg, A 50 MBPS 4x4 maximum likelihood decoder for multiple-input multiple-output systems with QPSK modulation, in: Icecs 2003: Proceedings Of The 2003 10Th Ieee International Conference On Electronics,Circuits And Systems, Vol. 1, IEEE, 2003, pp. 332–335.

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