Jozsa, CM.; Domene Oltra, F.; Vidal Maciá, AM.; Piñero Sipán, MG.; González Salvador, A. (2014). High performance lattice reduction on heterogeneous computing platform. Journal of Supercomputing. 70(2):772-785. doi:10.1007/s11227-014-1201-2.

Noname manuscript No.
(will be inserted by the editor)

High Performance Lattice Reduction on Heterogeneous Computing Platform

Csaba M. Józsa · Fernando Domene · Antonio M. Vidal · Gema Piñero · Alberto González

Received: date / Accepted: date

Abstract The lattice reduction (LR) technique has become very important in many engineering fields. However, its high complexity makes difficult its use in real-time applications, especially in applications that deal with large matrices. As a solution, the Modified Block LLL (MB-LLL) algorithm was introduced in [10], where several levels of parallelism were exploited: (i.) *coarse-grained parallelism* was achieved by applying the block-reduction concept presented in [15] and (ii.) *fine-grained parallelism* was achieved through the Cost Reduced All-Swap LLL (CR-AS-LLL) algorithm introduced in [10].

In this paper, we present the Cost Reduced MB-LLL (CR-MB-LLL) algorithm, which allows to significantly reduce the computational complexity of the MB-LLL by allowing the relaxation of the first LLL condition while executing the LR of submatrices, resulting in the delay of the GS coefficients update and by using less costly procedures during the boundary checks. The effects of complexity reduction and implementation details are analyzed and discussed for several architectures. A mapping of the CR-MB-LLL on a heterogenenous platform is proposed and it is compared with implementations running on a *dynamic parallelism* enabled GPU and a *multi-core* CPU. The mapping on the architecture proposed allows a dynamic scheduling of kernels where the overhead introduced is hidden by the use of several CUDA streams. Results show that the execution time of the CR-MB-LLL algorithm on the heterogeneous platform outperforms the multi-core CPU and it is more efficient than the CR-AS-LLL algorithm in case of large matrices.

Keywords Lattice Reduction · LLL · GPU · CUDA · OpenMP

Csaba M. Józsa
Faculty of Information Technology, Pázmány Péter Catholic University, Hungary
E-mail: jozsa.csaba@itk.ppke.hu

Fernando Domene, Gema Piñero, Alberto González
Inst. Telecommunications and Multimedia Applications (iTEAM), Universitat Politècnica de València, Spain
E-mail: {ferdool,gpinyero,agonzal}@iteam.upv.es

Antonio M. Vidal
Dept. of Information Systems and Computation, Universitat Politècnica de València, Spain
E-mail: avidal@dsic.upv.es

1 Introduction

The application of lattice reduction (LR) as a preconditioner of various signal processing algorithms plays a key role in several fields: communications, cryptography, image processing, etc. Given a basis, LR consists of finding another basis whose vectors are more orthogonal and shorter, in the sense of Euclidean norm, than the original ones. The Minkowski or Hermite-Korkine-Zolotareff reductions are the techniques that obtain the best performance in terms of reduction, but also the ones with a higher computational cost. Both techniques require the calculation of the shortest lattice vector, which has been proved to be NP-hard (see [17] and references therein).

In order to reduce the computational complexity of LR techniques, a polynomial time algorithm was proposed by Lenstra, Lenstra and Lovász, known as the LLL algorithm [11]. This algorithm can be seen as a relaxation of Hermite-Korkine-Zolotareff conditions [4] or an extension of Gauss reduction [17] and obtains the reduced basis by applying two different operations over the original basis: size-reduction (linear combination between columns) and column swap. Although further reduction techniques have been proposed afterwards, the LLL algorithm is the most used due to the good trade-off between performance and computational complexity.

Regarding the hardware implementation of the LLL algorithm, several solutions can be found in the literature. Implementations that make use of LR to improve the detection performance of multiple antenna systems can be found in [3, 8, 13, 16]. In [16], an LR-aided symbol detector for multiple-input multiple-output (MIMO) and orthogonal frequency division multiple access (OFDMA) is implemented using 65 nm ASIC technologies. A field-programmable gate array (FPGA) implementation of a variant of the LLL algorithm, the Clarkson's algorithm, is presented in [3], whose main benefit is the computational complexity reduction without significant performance loss in MIMO detection. More recently, [8] makes use of a Xilinx XC4VLX80-12 FPGA for implementing LR-aided detectors, whereas [13] uses an efficient VLSI design based on a pipelined architecture.

In [10] low level, fine-grained parallelism was implemented by the CR-AS-LLL algorithm, where the low processing time is assured by an efficient work distribution, minimizing the idle time of the launched threads. Based on the parallel block-reduction concept presented in [15], a higher level, coarse-grained parallelism can be applied as an extra level of parallelism. The idea is to subdivide the original lattice basis matrix in several smaller submatrices and perform an independent LR on them followed by a boundary check between adjacent submatrices. The MB-LLL algorithm proposed in [10] implements a parallel processing of the submatrices by using the parallel CR-AS-LLL algorithm for the LR of every submatrix.

The implementations of the previous references make use of only one architecture to calculate the LR of a basis. However, a better performance can be obtained by combining different architectures, what is known as heterogeneous computing. Among the different combinations, the use of CPU and GPU is probably the most popular since it can be found in most of the computers.

In this paper, we present the Cost Reduced MB-LLL (CR-MB-LLL) algorithm in order to further reduce the computational complexity of the MB-LLL algorithm [10]. The main idea behind the CR-MB-LLL algorithm is the relaxation of the first LLL condition while executing the LR for the submatrices, resulting

in the delay of the GS coefficients update and by using less costly procedures when performing the boundary checks. The effects of this complexity reduction are evaluated on different architectures.

A mapping of the CR-MB-LLL algorithm on a heterogeneous platform consisting of a CPU and a GPU is proposed and it is compared with implementations running on a GPU with *dynamic parallelism* (DP) capability and a multi-core CPU architecture. The proposed architecture allows a dynamic scheduling of kernels where the overhead introduced by host-device communication is hidden by the use of CUDA streams. Results show that the CR-MB-LLL algorithm executed on the heterogeneous platform outperforms the DP-based GPU and multi-core implementations.

The algorithm mapping on different parallel architectures is very challenging, since the number of processing cores, latency and size of the different memory hierarchies and available cache is different. The mapping details of the CR-AS-LLL and CR-MB-LLL algorithms to different parallel architectures is also presented with a special emphasis on the work distribution among the threads and the efficient memory utilization.

The paper is organized as follows. In Section 2 a brief introduction to LR is given. The architecture dependent mapping details of the CR-AS-LLL and CR-MB-LLL are presented in Section 3, followed by their performance evaluations on different platforms in Section 4. Finally, conclusions are stated in Section 5.

2 Problem description

A real-valued lattice $L = \left\{ \sum_{i=1}^{n} x_i \underline{b}_i \;\middle|\; x_i \in \mathbb{Z}, i = 1, \cdots, n \right\}$ is a discrete additive subgroup of $\mathbb{R}^n$, where $\underline{b}_1, \underline{b}_2, \cdots, \underline{b}_n \in \mathbb{R}^n$ are linearly independent vectors and $\mathbb{Z}$ denotes the set of integers. Let matrix $\mathbf{B} = (\underline{b}_1, \ldots, \underline{b}_n) \in \mathbb{R}^{n \times n}$ denote the full (column) ranked basis of the lattice. Let $\mathbf{B}^* = (\underline{b}_1^*, \ldots, \underline{b}_n^*) \in \mathbb{R}^{n \times n}$ denote the associated orthogonal basis of $\mathbf{B}$, calculated by the Gram-Schmidt (GS) orthogonalization process as $\underline{b}_1^* = \underline{b}_1$ and $\underline{b}_i^* = \underline{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \underline{b}_j^*$ for $2 \leq i \leq n$, where $\mu_{i,j} = (\underline{b}_i, \underline{b}_j^*)/(\underline{b}_j^*, \underline{b}_j^*)$ for $1 \leq j < i \leq n$, also called the GS coefficients and $(\cdot, \cdot)$ denotes the ordinary dot product on $\mathbb{R}^n$. Thus, matrix $\mathbf{B}$ can be expressed as $\mathbf{B} = \mathbf{B}^* \cdot \mathbf{U}$, where matrix $\mathbf{U}$ is upper triangular with unit diagonal, and whose element (i, j) above the diagonal is given by the GS coefficient $\mu_{j,i}$.

Definition 1 Given a lattice $\mathbf{L} \in \mathbb{R}^n$ with basis $\mathbf{B} \in \mathbb{R}^{n \times n}$, associated orthogonal basis $\mathbf{B}^* \in \mathbb{R}^{n \times n}$, and GS coefficients $\mu_{i,j}$, $\mathbf{B}$ is called *LLL-reduced* if the following conditions are satisfied:

$$|\mu_{i,j}| \leq \frac{1}{2} \qquad \text{for } 1 \leq j < i \leq n \tag{1}$$

$$\|\underline{b}_i^* + \mu_{i,i-1} \underline{b}_{i-1}^*\| \geq \delta \|\underline{b}_{i-1}^*\| \qquad \text{for } 1 < i \leq n,\ \frac{3}{4} \leq \delta < 1 \tag{2}$$

3 Parallel Lattice-Reduction algorithms and their mapping to parallel architectures

Since, the LLL algorithm shows a highly sequential behavior, multiple levels of parallelism have to be identified and exploited in order to efficiently parallelize

this algorithm. Dividing the problem in several sub-problems that can be executed concurrently, can be regarded as one level of parallelism. In addition, if a sub-problem could benefit from a multi-threaded environment it can be regarded as a second level of parallelism. Previous parallel LR implementations, such as the ones presented in [1, 2, 12], have focused only on multi-core architectures. The main drawback of the low number of threads offered by modern CPUs (compared to GPUs) is that low level parallelism can not be efficiently exploited. During an algorithm design, low level parallelism is usually omitted and the levels of parallelism are also restricted. In case of GPUs, the high number of CUDA cores makes possible the parallel execution of a high number of threads leading to significant performance improvements.

3.1 Mapping details of the CR-AS-LLL algorithm

Basically, LR consists of a succession of swaps between vectors of the basis and some operations to decrease their norms. The order in which the swaps are applied in the LLL algorithm is limiting in a parallel framework. Villard in [14] introduced the *any swap reduction* concept, that enables simultaneous basis swaps and served as a basis for future parallel implementations. In [12], the concept of delaying the *size reductions* was introduced. In the CR-AS-LLL algorithm, further computational cost is saved by rearranging and delaying the frequently used *size reduction* procedure.

Procedures **SimpleSizeReduce**, **SimpleSwap** and **Swap** are defined in order to give an accurate description of the CR-AS-LLL algorithm.

Procedure 1 (SimpleSizeReduce($\mathbf{B_i}, k, l$)) *Given a lattice generator matrix* $\mathbf{B}$ *and the associated GS coefficients matrix* $\mathbf{U}$*, if the condition* (1) *is not satisfied, i.e.* $|\mu_{k,l}| > \frac{1}{2}$*, the following updates have to be applied:*

- $\mu = \lceil \mu_{k,l} \rfloor$, $\mu_{k,l} = \mu_{k,l} - \mu$, $\underline{b}_k = \underline{b}_k - \mu \underline{b}_l$.

Procedure 2 (SimpleSwap($\mathbf{B_i}, k$)) *Given a lattice generator matrix* $\mathbf{B}$*, the associated orthogonal basis* $\mathbf{B}^*$ *and GS coefficients matrix* $\mathbf{U}$*, if the condition* (2) *is not satisfied, or equivalently* $\|\underline{b}_k^*\|^2 < (\delta - \mu_{k,k-1}^2)\|\underline{b}_{k-1}^*\|^2$*, the following updates have to be applied:*

- *swap* $\underline{b}_k$ *with* $\underline{b}_{k-1}$
- $\underline{b}_{k-1}^{*'} = \underline{b}_k^* + \mu_{k,k-1}\underline{b}_{k-1}^*$, $\mu_{k,k-1}' = (\underline{b}_{k-1}^*, \underline{b}_{k-1}^{*'})/\|\underline{b}_{k-1}^{*'}\|^2$, $\underline{b}_k^{*'} = \underline{b}_{k-1}^* - \mu_{k,k-1}'\underline{b}_{k-1}^{*'}$
- $\underline{b}_{k-1}^* = \underline{b}_{k-1}^{*'}$,$\underline{b}_k^* = \underline{b}_k^{*'}$,$\mu_{k,k-1} = \mu_{k,k-1}'$

Procedure 3 (Swap($\mathbf{B_i}, k$)) *Given a lattice generator matrix* $\mathbf{B}$*, the associated orthogonal basis* $\mathbf{B}^*$ *and GS coefficients matrix* $\mathbf{U}$*, if the condition* (2) *is not satisfied, or equivalently* $\|\underline{b}_k^*\|^2 < (\delta - \mu_{k,k-1}^2)\|\underline{b}_{k-1}^*\|^2$*, the following updates have to be applied:*

- *perform* **SimpleSwap**(k)
- *swap* $\mu_{k,j}$ *with* $\mu_{k-1,j}$*, for* $1 \leq j < k-1$,
- $\begin{pmatrix} \mu_{i,k-1} \\ \mu_{i,k} \end{pmatrix} = \begin{pmatrix} \mu_{i,k-1}\mu_{k,k-1}' + \mu_{i,k}\|\underline{b}_k^*\|^2/\|\underline{b}_{k-1}^{*'}\|^2 \\ \mu_{i,k-1} - \mu_{i,k}\mu_{k,k-1} \end{pmatrix}$ *for* $k+1 \leq i < n$.

Algorithm 1 CR-AS-LLL OpenMP pseudocode

Input: $[\mathbf{B_1}, \mathbf{B_2}, \ldots, \mathbf{B_m}]$, $[\mathbf{B_1^*}, \mathbf{B_2^*}, \ldots, \mathbf{B_m^*}]$, $[\mathbf{U_1}, \mathbf{U_2}, \ldots, \mathbf{U_m}]$, δ
Output: $[\mathbf{B_1}, \mathbf{B_2}, \ldots, \mathbf{B_m}]$ as LLL reduced basis
$maxT \leftarrow$ set the maximum number of available OpenMP threads
$simMat \leftarrow$ set the number of matrices processed simultaneously
$TPM = maxT/simMat$ ▷ The number of threads for parallel processing one matrix
#pragma omp parallel numthreads($simMat$) {
  $grp \leftarrow$ set current thread id
  $odd = true$, $even = true$, $off = 0$, $i = grp \cdot (m/simMat)$
  #pragma omp parallel numthreads(TPM) shared($odd, even, off$) firstprivate(grp) {
    while $(i < (grp + 1) \cdot MPG)$ **do**
      while (odd or $even$) **do**
        #pragma omp single {
          if $off == 0$ **then** $odd = false, off = 1$ **else** $even = false, off = 0$ **end if**
        }
        #pragma omp for reduction(||:odd,even)
        for $k = 2 + off$ **to** n **step** 2 ▷ Embarrassingly parallel for all k
          Update GS coefficient $\mu_{k,k-1}$ and **SimpleSizeReduce**($\mathbf{B_i}$,k,$k-1$)
          if $\|\underline{b}_k^*\|^2 < (\delta - \mu_{k,k-1}^2)\|\underline{b}_{k-1}^*\|^2$ **then**
            Perform **SimpleSwap**($\mathbf{B_i}$,k)
            if($off == 0$) **then** $even = true$ **else** $odd = true$ **end if**
          end if
        end for
      end while
      #pragma omp barrier
      Update all of the GS coefficients of $\mathbf{B_i}$ and perform **SimpleSizeReduce** if necessary ▷ Highly parallel
      #pragma omp single {$i \leftarrow i + 1, odd = true, even = true$}
    end while
  }
}

In Alg. 1, the OpenMP implementation of the CR-AS-LLL is presented. Two-level parallelism is implemented based on a nested parallelism construct. The outer level parallelism starts the concurrent processing of $simMat$ number of lattice basis and the inner parallel construct is responsible for the parallel LR of a basis with TPM number of threads. As the outer level parallelism is expanded, namely $simMat$ is increased, the threads available for the parallel LR are decreased.

When mapped to the GPU, the performance of the CR-AS-LLL algorithm depends on the efficiency of the *work distribution* among the available GPU threads and the implementation of the most frequently used operations, such as *dot products*, *size reductions* and *column swaps*. In Alg. 2, the CUDA pseudo-code is presented with a two dimensional thread block TB(T_x, T_y) configuration, where T_x and T_y denotes the number of threads in the x and y dimension. The kernel is launched with a one dimensional *grid* whose size is determined by the number of basis processed simultaneously. The number of threads T_y is defined based on the size of the original basis, i.e. $T_y = \max(n/2, 32)$. By enabling the usage of $T_x = \max(n, 32)$ threads in the x dimension, the threads that belong to the same y dimension will form a *warp*, consequently the global memory loads and stores issued by the threads of the warp will be coalesced. The y dimension also defines the extent of paralellism. The iteration variable of the for loop is increased in every iteration by $T_y \cdot 2$. In other words, in every phase the threads with the same id_y have to reduce and swap at most $n/(T_y \cdot 2)$ vectors.

Algorithm 2 CR-AS-LLL CUDA kernel pseudocode - processing of one lattice basis $\mathbf{B_i}$ with 2 dimensional thread block configuration $TB(T_x, T_y)$

Input: $\mathbf{B_i}$,$\mathbf{B_i^*}$,$\mathbf{U}$, δ and thread identifiers id_x, id_y
Output: $\mathbf{B_i}$ as a LLL reduced basis
Definition of shared arrays $buf_1[T_y][T_x], buf_2[T_y][T_x], \mu[T_y]$
Definition of shared variables $odd = true, even = true$ and private variable off
Copy the elements above the diagonal from $\mathbf{U}$ to shared array $U_\setminus[n-1]$
while *odd* or *even* **do** ▷ $T_x \cdot T_y$ threads are working on the while loop
    $off = (off + 1) \mod 2$
    for $k = id_y * 2 + 1 + off$ **to** n **step** $k \mathrel{+}= T_y * 2$ **do**
        Call **DotProduct**($\mathbf{b^*_{k-1}}$,$\mathbf{b^*_{k-1}}$,$buf_1[id_y][]$), **DotProduct**($\mathbf{b_k}$,$\mathbf{b^*_{k-1}}$,$buf_2[id_y][]$)
        Threads with ($id_x == 0$) set $U_\setminus[k-1] = buf_2[id_y][0]/buf_1[id_y][0]$
        if $|U_\setminus[k-1]| > 0.5$ **then** ▷ Check reduction criteria
            Threads with ($id_x == 0$) set $\mu[id_y] = \lceil U_\setminus[k-1] \rfloor$
            Call **SimpleSizeReduce**($\mathbf{b_k}, \mathbf{b_k}, \mathbf{b_{k-1}}, \mu[id_y]$)
        end if
        Call **DotProduct**($\mathbf{b^*_k}$,$\mathbf{b^*_k}$,$buf_2[id_y][]$)
        if $buf_2[id_y][0] < (\delta - U_\setminus[k-1]^2) \cdot buf_1[id_y][0]$ **then**
            Call **SimpleSwap**($\mathbf{b_k}, \mathbf{b_{k-1}}, buf_1[id_y][]$)
            Call **SimpleSizeReduce**($\mathbf{b^{*\prime}}, \mathbf{b^*_k}, \mathbf{b^*_{(k-1)}}, U_\setminus[k-1]$)
            Call **DotProduct**($\mathbf{b^{*\prime}}$,$\mathbf{b^{*\prime}}$,$buf_1[id_y][]$), **DotProduct**($\mathbf{b^*_{k-1}}$,$\mathbf{b^{*\prime}}$,$buf_2[id_y][]$)
            Threads with ($id_x == 0$) set $U_\setminus[k-1] = buf_2[id_y][0]/buf_1[id_y][0]$ and set *odd* or *even* to *true* depending on the off variable
            Call **SimpleSizeReduce**($\mathbf{b^*_k}, \mathbf{b^*_{(k-1)}}, \mathbf{b^{*\prime}}, U_\setminus[k-1]$) and update $\mathbf{b^*_{(k-1)}} = \mathbf{b^{*\prime}}$
        end if
    end for
    Synchronize threads
end while
Copy the $U_\setminus$ to the diagonal elements of $\mathbf{U}$
Update the rest of GS coefficients based on the procedures and methods presented above
procedure DotProduct($v_1, v_2, buf[T_x]$) ▷ The result is stored in buf at index 0
    $buf[id_x] = 0$
    for $i = id_x$ **to** n **step** $i \mathrel{+}= T_x$ **do** $buf[id_x] \mathrel{+}= v_{1i} \cdot v_{2i}$ **end for**
    for $stride = T_x/2$ **to** $stride > 0$ **step** $stride >>= 1$ **do**
        if $id_x < stride$ **then** $buf[id_x] \mathrel{+}= buf[id_y][stride + id_x]$ **end if**
    end for
end procedure
procedure SimpleSizeReduce(v_1, v_2, v_3, μ)
    for $i = id_x$ **to** n **step** $i \mathrel{+}= T_x$ **do** $v_{1i} = v_{2i} - \mu \cdot v_{3i}$ **end for**
end procedure
procedure SimpleSwap($v_1, v_2, buf[T_x]$)
    for $i = id_x$ **to** n **step** $i \mathrel{+}= T_x$ **do**
        (i.) $buf[id_x] = v_{1i}$, (ii.) $v_{1i} = v_{2i}$, (iii.) $v_{2i} = buf[id_x]$
    end for
end procedure

The elements of matrices $\mathbf{B}, \mathbf{B}^*, \mathbf{U}$ are stored in the global memory of the GPU. Since the size of shared memory is limited, it is not possible to load the entire matrices in this low latency memory. Furthermore, the excessive use of shared memory is decreasing the occupancy, resulting in performance degradation. Shared buffers $buf_1[T_y][T_x]$ and $buf_2[T_y][T_x]$ are allocated in order to efficiently compute the dot products and the vector norms. In order to avoid unnecessary access to the global memory, the GS coefficients right above the diagonal are also stored in the shared buffer $U_\setminus$ due to their frequent access.

A major difference between the CUDA and OpenMP mapping lies in the implementation of the size reductions, dot products and swaps. In the CUDA implementation, because of the two dimensional TB configuration, T_x number of threads are working in every procedure. For example, in the dot product calculation every thread has to do n/T_x number of multiplications and the result of the multiplication is added to the shared memory buffer. When the execution of all the thread finishes, a parallel prefix sum is applied on the buffer in order to conclude the dot product computation. In case of the OpenMP implementation, only one thread is working in the computation of a dot product since the number of threads are limited.

3.2 Mapping details of the CR-MB-LLL algorithm

As stated in the previous sections the MB-LLL algorithm allows to split a large matrix in several smaller submatrices where parallel LR is performed in a block-wise manner with the parallel LR algorithm CR-AS-LLL. Once the LR of the submatrices is finished, the boundaries between adjacent submatrices are checked and finally the GS coefficients outside the initial groups are updated. The main condition is to keep every submatrix as an LLL-reduced matrix throughout the processing. In [10], a detailed description of the MB-LLL algorithm is shown.

The proposed CR-MB-LLL algorithm further reduces the computational complexity of the MB-LLL algorithm. In the MB-LLL algorithm, the submatrices affected by a boundary swap have to be LLL-reduced and the GS coefficients have to be updated. Moreover, in order to maintain the LLL conditions in the submatrices affected by a boundary swap, the **Swap** procedure has to be performed. The complexity reduction is achieved by eliminating the GS coefficients update in the submatrices after the execution of the CR-AS-LLL and the usage of the **SimpleSwap** procedure instead of **Swap** in case of a boundary swap. Since the GS coefficients are updated only when the ordering condition (2) is met for every column vector, the processing time can be considerably reduced.

In the following, algorithm mappings are proposed for a (i.) GPU, (ii.) multi-core CPU architecture and (iii.) a heterogeneous system.

The GPU mapping of the CR-MB-LLL algorithm is similar to the one presented in Section 3.1, since the procedures used are performed with a two dimensional TB configuration even in the case of a boundary check. The main difference is that *dynamic parallelism* (DP) enables the launch of new kernels from the GPU without returning the program flow control to the CPU. DP is a feature that was introduced in CUDA 5.0 and device compute capability 3.5 is required.

The schematic of the kernels scheduling implementing the DP is shown in Fig. 1. The CPU launches the *Block* kernel. The size of the grid is equal to the number of matrices that are simultaneously processed and the number of threads in one TB is equal to the number of submatrices. In this case, every thread has to prepare the data for the corresponding submatrices and launch the *CR-AS-LLL* kernel. The kernel has to be relaunched if the LLL conditions were broken by a boundary swap, which can be solved by tracking state variables placed in the global memory. When all the submatrices are reduced, the *Boundaries Check (BC)* kernel is launched. Since the operations performed in this section are dot products and column swaps, the thread configuration of the TB is the same as

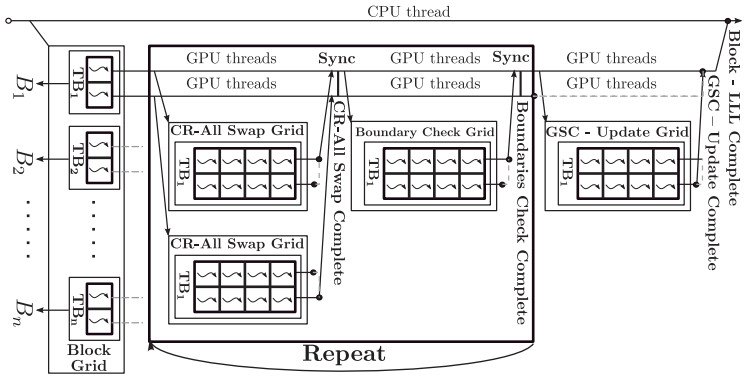

Fig. 1: Kernels scheduling on *dynamic parallelism* enabled GPU for the CR-MB-LLL algorithm.

in case of the *CR-AS-LLL* kernel. The *CR-AS-LLL* and *BC* kernels are repeated until there are no swaps on the boundaries. Because one matrix is assigned to one TB in the parent *Block* kernel, the processing of the different matrices can be done simultaneously despite the variable number of iterations. Finally, the GS coefficients outside the blocks are updated with the *GSC-Update* kernel and the size-reduction is performed wherever is needed.

Algorithm 3 The mapping of the CR-MB-LLL on the heterogeneous platform

Input: $[\mathbf{B_1}, \mathbf{B_2}, \ldots, \mathbf{B_m}]$, δ, block-size l, T number of OpenMP threads
Output: $[\mathbf{B_1}, \mathbf{B_2}, \ldots, \mathbf{B_m}]$ as LLL reduced basis
#pragma omp parallel {
$mpt = m/T$ ▷ The number of matrices that have to processed by one thread
$bpm = n/l$ ▷ The number of blocks per matrix
Assign a CUDA $stream_{id}$ to the current CPU thread with identifier id
Define arrays $matIndD[mpt]$ on the GPU and $matIndH[mpt]$ on the host ▷ The indexes of the unprocessed matrices are stored in these arrays
for $i = 0$ **to** mpt **step** $i{+}{+}$ **do** $matIndH[i] = id \cdot mpt + i$ **end for**
Define arrays $boundaryExchD[mpt \cdot bpm]$ and $boundaryExchH[mpt \cdot bpm]$
while $mpt > 0$ **do**
    Asynchronously copy $matIndH$ to $matIndD$ on $stream_{id}$
    Launch $CR - AS - LLL$ kernel on $stream_{id}$ with grid size $grid_{lll} = mpt \cdot bpm$ and $TB(T_x, T_y)$ ▷ The CR-AS-LLL is performed on the submatrices, without updating the GS coefficients
    Launch the $BoundaryCheck$ kernel on $stream_{id}$ with grid size $grid_{bc} = mpt \cdot (bpm - 1)$ and $TB(B_x, B_y)$ ▷ The LLL conditions (1) and (2) are checked on the boundary of two adjacent submatrices. In case if the conditions are not met the **SimpleSwap** is executed instead of the **Swap** procedure.
    Asynchronously copy $boundaryExchD$ to $boundaryExchH$ on stream id
    Synchronize CPU thread with $stream_{id}$
    if There was no boundary exchange for one matrix **then**
        Remove the matrix index from $matIndH$ and $mpt \leftarrow mpt - 1$ ▷ The CPU threads have to process the result of the boundary exchange
    end if
end while
Launch the $GSC - Update$ kernel on $stream_{id}$ ▷ In this kernel all the GS coefficients are updated and size reduction is performed where necessary.
}

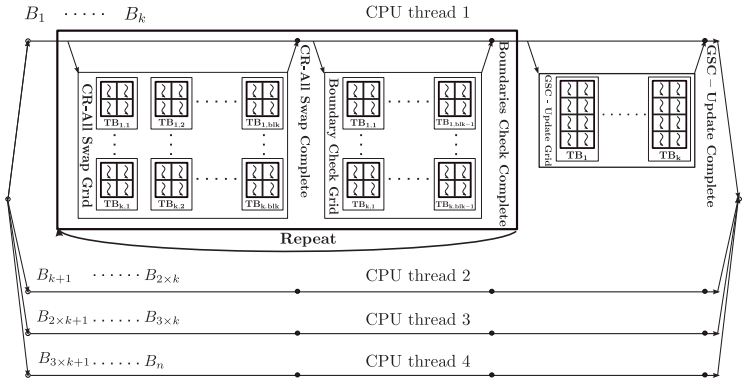

Fig. 2: Kernels scheduling on the heterogeneous platform for the CR-MB-LLL algorithm.

The schematic of the heterogeneous platform is shown in Fig. 2. The CPU threads are responsible for launching *CR-AS-LLL*, *BC* and *GSC-Update* kernels, update the state variables and implement the control logic of the dynamic scheduling. The mapping of the CR-MB-LLL algorithm on the heterogeneous platform is presented in Alg. 3.

A different CUDA stream is assigned for every CPU thread, making possible the concurrent kernel execution and reducing the idle time of the CUDA cores. Before launching the *CR-AS-LLL* and *BC* kernels, the CPU thread updates the $matIndD$ array placed in the GPU's global memory to specify which matrices need further processing. The size of the grid is dynamically adjusted according to the number of non-processed matrices in every iteration. After the *Boundary Check* kernel is executed and the $boundaryExchH$ is updated on the host, the CPU thread checks if the LR of any matrix is finished. If LLL reduced matrices are found, the $matIndH$ is updated and consequently the size of the grids assigned to the *CR-AS-LLL* and *BC* kernels is decreased. The *GSC-Update* kernel starts after all the matrices assigned to one CPU thread have been completely processed.

The control structure required by the multi-core architecture is similar to the one presented in the heterogeneous platform. The difference is that instead of launching GPU kernels, the master threads fork a specified number of slave threads that are processing the submatrices in parallel. The parallel LR of the submatrices is performed according to Alg. 1. In this case the very limited number of CPU threads restrict the exploitation of several levels of parallelism.

4 Evaluation results

In this section we present the performance results of the proposed algorithms. The computations were done in single-precision floating point arithmetic and parameter $\delta = 0.75$ was used for the LLL condition (2). Block-Toeplitz matrices have been considered to evaluate the performance of the different implementations. These type of matrices are usually used in wireless communications [17].

Figure 3 shows the computational times of the MB-LLL algorithm based on the architectures discussed in Section 3.2 for different matrix dimensions, where

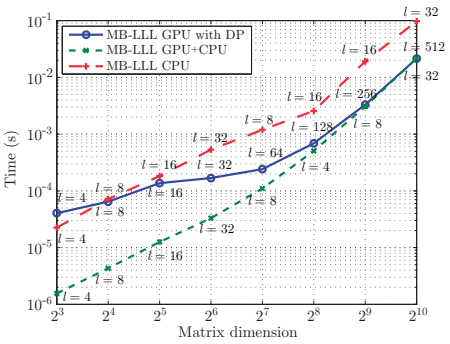

Fig. 3: Computational time of the MB-LLL algorithm on different architectures, where l denotes the size of the processed blocks.

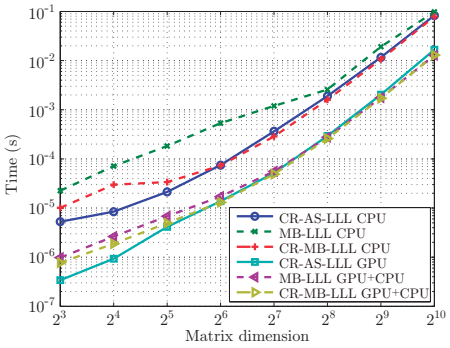

Fig. 4: Computational times of algorithms CR-AS-LLL, MB-LLL and CR-MB-LLL for matrix dimensions $2^3 - 2^{10}$.

l denotes the size of the processed blocks. The performance measurements were evaluated with all the possible block sizes and the best configuration is shown. The architectures used for the computational time measurements are the Nvidia Tesla K20 (DP capability) and an Intel Core i7-3820 processor. The processing times show similar performance for large matrices when the GPU is involved. However, the heterogeneous platform clearly outperforms the solution based on DP in the case of small matrices. This gap is caused by the overhead required when launching kernels from kernels with DP and the limited overlapping execution of kernels on different streams. The conclusion is that the data transfer between CPU and GPU required by the heterogeneous system is less time consuming than the overhead of the kernel launch with DP and the limitation of the concurrent execution of kernels on different streams.

Figure 4 compares the average computational time of the CR-AS-LLL, MB-LLL and CR-MB-LLL algorithms for different matrix dimensions. The algorithms were evaluated on the Intel Core i7-3820 CPU, the Nvidia GeForce GTX 690 GPU and the heterogeneous system containing the previously mentioned CPU and GPU. In [7], it was shown that the GTX 690 has a better performance than the K20 when performing LR, thus the DP is not required. Regarding the GPU and comibned CPU+GPU implementations the following conclusions can be drawn: (i.) the computational time of the CR-MB-LLL is $25-40\%$ lower in case of small and medium-sized matrices compared to the MB-LLL algorithm and the performance is similar in case of larger matrices, (ii.) the CR-AS-LLL performs better than the CR-MB-LLL in case of small matrices, however for large matrices the block concept implemented in the CR-MB-LLL achieves 30% speed-up compared to the CR-AS-LLL and (iii.) the systems using the GPU outperform the CPU for every matrix dimension with speed-ups ranging from 6 to 15.

Regarding the CPU implementations the following conclusions can be drawn: (i.) the CR-MB-LLL always outperforms the MB-LLL algorithm with speed-ups ranging from 2 to 7, (ii.) the CR-AS-LLL algorithm performs better than the MB-LLL and CR-MB-LLL for matrices with low dimensions ($2^3 - 2^6$), (iii.) the computational time of the CR-MB-LLL is $10-20\%$ lower in case of larger matrices compared to the CR-AS-LLL.

Ref	Algorithm	Architecture	4×4	8×8	64×64	1024×1024
[3]	Clarksons Algorithm	Virtex-II-Pro FPGA	4.2×10^{-6}	x	x	x
[9]	Complex LLL	Virtex-5 FPGA	0.79×10^{-6}	x	x	x
[6]	Brun's Algorithm	ASIC 250 nM	0.07×10^{-6}	x	x	x
[5]	Reverse Siegel LLL	Virtex-4 FPGA	0.18×10^{-6}	x	x	x
[5]	Reverse Siegel LLL	ASIC 130 nM	0.04×10^{-6}	x	x	x
[1]	SB-LLL	ADRES	0.17×10^{-6}	x	x	x
This work	CR-AS-LLL	GTX690 GPU	x	0.33×10^{-6}	1.37×10^{-5}	1.67×10^{-2}
This work	CR-MB-LLL	GTX690 GPU + Intel i7-3820 CPU	x	0.77×10^{-6}	1.30×10^{-5}	1.28×10^{-2}

Table 1: Performance comparison of different lattice reduction implementations.

A surprising result is that, while the CR-MB-LLL achieves a significant speed-up compared to the MB-LLL for the CPU architecture, the same is not true for GPU architecture. This fact is due to several reasons. The computational complexity reductions for the CR-MB-LLL affect only the *CR-AS-LLL* and *BC* kernels. However, in case of large matrices, the *GSC-Update* kernel is taking the major part of the processing time. This kernel has to access the global memory frequently and these accesses have a high latency. In case of the CPU, this problem is alleviated by the high speed memory access and the large amount of available cache for CPU.

The performance of LR mostly depends on the precision of the computation, the size and type of the basis matrix and the architecture used. In Table 1 performance of existing implementations are presented. Previous research mostly focused on small matrices. In [2] performance measures for higher dimension matrices are presented as well, however the total runtime of the algorithm is not specified.

5 Conclusions

In this paper, we proposed the CR-MB-LLL algorithm and we have compared it with the CR-AS-LLL and MB-LLL algorithms presented in [10]. The idea behind the CR-MB-LLL algorithm is the relaxation of LLL condition (1) for the sub-matrices, resulting in the delay of the GS coefficients update when executing the LR and the replacement of the **Swap** procedure by the less costly **SimpleSwap** procedure when performing the boundary checks.

The CR-MB-LLL algorithm has been evaluated on several architectures: a multi-core architecture, a GPU with DP capability and a heterogeneous platform based on a CPU and GPU. Results show that mapping the CR-MB-LLL algorithm on the heterogeneous architecture reduces the computational time by 30% compared to the CR-AS-LLL in case of large matrices, whereas implementations involving GPUs achieve speed-up factors from $6 - 15$ compared to the multi-core CPU architecture. The MB-LLL algorithm achieves speed-up factors ranging from $5 - 25$ when launched on the proposed heterogeneous platform compared to the DP-based GPU implementation for matrix dimensions ranging $2^3 - 2^6$.

It was shown that the efficiency of the CR-MB-LLL is significantly affected by the architectures used. The CR-MB-LLL is $1.5 - 7$ times faster compared to the MB-LLL algorithm when launched on multi-core CPU architecture, however the CR-MB-LLL is only at most 1.4 times faster compared to the MB-LLL when launched on the GPU architecture. This is mainly because the computational complexity reductions introduced in the CR-MB-LLL algorithm affect the *CR-AS-LLL* and *BC* kernels. However, in case of large matrices the *GSC-Update* kernel is

taking the major part of the processing time with frequent accesses to the global memory of the GPU. In case of the CPU, the memory access has a lower latency and the available cache for CPU is significantly bigger, causing different speed-ups of the same algorithm on the different architectures.

Acknowledgements Financial support for this study was provided by grants TÁMOP-4.2.1./B-11/2/KMR-2011-0002, TÁMOP-4.2.2/B-10/1-2010-0014 from the Pázmány Péter Catholic University, European Union ERDF, Spanish Government through TEC2012-38142-C04-01 project and Generalitat Valenciana through PROMETEO/2009/013 project.

References

1. Ahmad, U., Amin, A., Li, M., Pollin, S., Van der Perre, L., Catthoor, F.: Scalable block-based parallel lattice reduction algorithm for an SDR baseband processor. In: Communications (ICC), 2011 IEEE International Conference on (2011)
2. Backes, W., Wetzel, S.: Parallel lattice basis reduction - the road to many-core. In: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on (2011)
3. Barbero, L.G., Milliner, D.L., Ratnarajah, T., Barry, J.R., Cowan, C.: Rapid prototyping of Clarkson's lattice reduction for MIMO detection. In: Communications, 2009. ICC'09. IEEE International Conference on, pp. 1–5 (2009)
4. Bremner, M.R.: Lattice basis reduction: An introduction to the LLL algorithm and its applications. CRC Press (2012)
5. Bruderer, L., Studer, C., Wenk, M., Seethaler, D., Burg, A.: VLSI implementation of a low-complexity LLL lattice reduction algorithm for MIMO detection. In: Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on (2010)
6. Burg, A., Seethaler, D., Matz, G.: VLSI implementation of a lattice-reduction algorithm for multi-antenna broadcast precoding. In: Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on, pp. 673–676 (2007)
7. Domene, F., Józsa, C.M., Vidal, A.M., Piñero, G., Gonzalez, A.: Performance analysis of a parallel lattice reduction algorithm on many-core architectures. In: Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering (2013)
8. Gestner, B., Zhang, W., Ma, X., Anderson, D.: Lattice reduction for MIMO detection: From theoretical analysis to hardware realization **58**(4), 813–826 (2011)
9. Gestner, B., Zhang, W., Ma, X., Anderson, D.V.: VLSI implementation of a lattice reduction algorithm for low-complexity equalization. In: Circuits and Systems for Communications, 2008. ICCSC 2008. 4th IEEE International Conference on, pp. 643–647 (2008)
10. Józsa, C.M., Domene, F., Piñero, G., González, A., Vidal, A.M.: Efficient GPU implementation of lattice-reduction-aided multiuser precoding. In: Wireless Communication Systems (ISWCS 2013), Proceedings of the Tenth International Symposium on (2013)
11. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen **261**(4), 515–534 (1982)
12. Luo, Y., Qiao, S.: A parallel LLL algorithm. In: Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering, pp. 93–101 (2011)
13. Shabany, M., Youssef, A., Gulak, G.: High-throughput 0.13-μm CMOS lattice reduction core supporting 880 Mb/s detection. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **21**(5) (2013)
14. Villard, G.: Parallel lattice basis reduction. In: Papers from the international symposium on Symbolic and algebraic computation, ISSAC '92. ACM, New York, NY, USA (1992)
15. Wetzel, S.: An efficient parallel block-reduction algorithm. In: Algorithmic Number Theory, pp. 323–337. Springer (1998)
16. Wu, D., Eilert, J., Liu, D.: A programmable lattice-reduction aided detector for MIMO-OFDMA. In: Circuits and Systems for Communications, 2008. ICCSC 2008. 4th IEEE International Conference on, pp. 293–297 (2008)
17. Wubben, D., Seethaler, D., Jaldén, J., Matz, G.: Lattice reduction **28**(3), 70–91 (2011)