This is a preprint of the article "Lazy Parallel Kronecker Algebra-Operations on Heterogeneous Multicores" by W. Sodsong, R. Mittermayr, Y. Park, B. Burgstaller and J. Blieberger. The article was published in the proceedings of the International European Conference on Parallel and Distributed Computing, EuroPar 2017, pp. 538-552.
The final publication is available at Springer via https://doi.org/10.1007/978-3-319-64203-1_39.

Lazy Parallel Kronecker Algebra-Operations on Heterogeneous Multicores

Wasuwee Sodsong[1], Robert Mittermayr[2], Yoojin Park[1], Bernd Burgstaller[1], and Johann Blieberger[2]

[1] Yonsei University, Korea,
wasuwee.s@yonsei.ac.kr,{yoojin1.park,bburg}@cs.yonsei.ac.kr
[2] Vienna University of Technology, Austria
{robert,blieb}@auto.tuwien.ac.at

Abstract. Kronecker algebra is a matrix calculus which allows the generation of thread interleavings from the source-code of a program. Thread interleavings have been shown effective for proving the absence of deadlocks. Because the number of interleavings grows exponentially in the number of threads, deadlock analysis is still a challenging problem.
To make the computation of thread interleavings tractable, we propose a lazy, parallel evaluation method for Kronecker algebra. Our method incorporates the constraints induced by synchronization constructs. To reduce problem size, only interleavings legal under the locking behavior of a program are considered. We leverage the data-parallelism of Kronecker sum- and product-operations for multicores and GPUs. Proposed algebraic transformations further improve performance. For one synthetic and two real-world benchmarks, our GPU implementation is up to 5453× faster than our multi-threaded version. Lazy evaluation significantly reduces memory consumption compared to both the sequential and the multicore versions of the SPIN model-checker.

Keywords: Kronecker algebra, Lazy evaluation, Deadlock detection, Heterogeneous multicores, GPUs

1 Introduction

The complexity of software-development for multicore processors has raised the interest in verification techniques for multi-threaded applications. To prove a property of a multi-threaded program, e.g., deadlock freedom, all possible thread interleavings must be considered. The number of interleavings increases exponentially in the number of threads. This combinatorial explosion is referred as the state explosion problem. All state-of-the-art methods suffer from the state explosion problem, including model checking (see, e.g., [4]).

Kronecker algebra is a matrix calculus that has been applied to model multi-threaded shared-memory systems [2, 10, 11, 16]. Kronecker algebra encodes the control-flow graphs (CFGs) of threads and synchronization primitives as adjacency matrices. In applying combinations of Kronecker sum and product operations, all interleavings of the underlying threads can be generated (see Sect. 2).

Related to the state explosion problem, the order of adjacency matrices grows exponentially in the number of threads. It has been observed in prior, unpublished work [10] that it is not necessary to compute adjacency matrices in their entirety: the use of synchronization constructs in a multi-threaded program induces constraints that can be exploited to greatly limit the number of thread interleavings that must be considered by static program analysis.

Kronecker algebra operations have been devised that are able to capture the constraints on possible thread interleavings resulting from semaphore-based producer-consumer synchronization and from mutual exclusion using mutexes and semaphores. Kronecker algebra can be applied with higher-level, monitor-like synchronization constructs such as Ada's protected objects, and with barriers [9, 10, 3, 12].

We propose a lazy evaluation method for Kronecker algebra operations, which computes only thread interleavings which are legal under the synchronization behavior of a given program. This lazy evaluation scheme greatly reduces problem sizes by considering only those distinctive portions of each adjacency matrix, which represent the legal thread interleavings of the program at hand.

We found the Kronecker matrix calculus to contain a vast amount of data-parallelism. Our lazy evaluation method is able to harness this parallelism on multicore CPUs and GPUs. Algebra transformations for particular matrix instances further enhance the performance of our matrix operations.

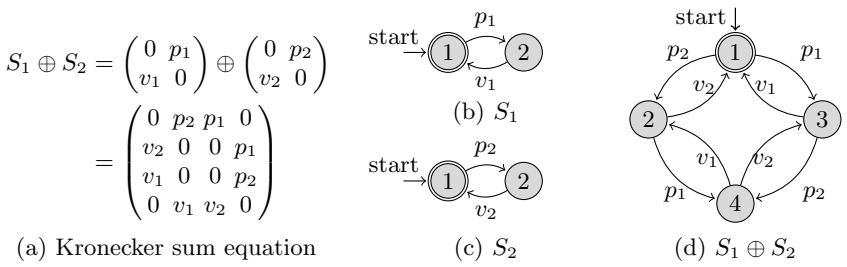
This paper thus makes the following contributions:

1. We devise a two-step lazy evaluation scheme for Kronecker sum- and product-operations: expression trees are constructed and then evaluated lazily.
2. We provide Kronecker algebra operations optimized for multicore CPUs.
3. We devise an execution scheme that utilizes both the multicore CPU and the GPU. This scheme conducts lazy evaluation of Kronecker algebra operations on the GPU. CPU cores are used to maintain the computed thread interleavings, and for coordinating the GPU-based evaluation process.
4. We perform an extensive evaluation, showing that the GPU implementation is up to 5453× faster than our multi-threaded CPU implementation.

This paper is organized as follows. The relevant background on Kronecker algebra is discussed in Sect. 2. We provide an overview of our execution scheme in Sect. 3. Our multicore CPU and GPU implementations are discussed in Sect. 4. Experimental results are provided in Sect. 5. We review the related work in Sect. 6 and draw our conclusions in Sect. 7.

2 Background

For the verification of concurrent systems using Kronecker algebra, threads and semaphores are represented by CFGs. The usual CFG representation consists of nodes representing basic blocks and edges representing transfer of control. Because our matrix calculus manipulates CFG edges, we move basic blocks onto



(a) Kronecker sum equation (b) S_1 (c) S_2 (d) $S_1 \oplus S_2$

Fig. 1. (a) A Kronecker sum $S_1 \oplus S_2$ of two binary semaphores S_1 (b) and S_2 (c) generates all possible interleavings (d).

the incoming edges of a node. Each CFG is encoded as an adjacency matrix, and CFG edges are labeled by the elements of a semiring [7, 10].

Example CFGs of binary semaphores and their adjacency matrix representations are depicted in Fig. 1. Each matrix row-index corresponds to the node ID of the tail of a CFG edge, and the matrix column-index corresponds to the node ID of the head of a CFG edge. For example, the first row of the result adjacency matrix, 0 p_2 p_1 0, specifies that there is an edge labelled "p_2" from Node 1 to Node 2 and another edge labelled "p_1" from Node 1 to Node 3. Multiple outgoing edges represent branches in a program. The Kronecker matrix calculus interprets the CFG of a thread as a deterministic finite automation (DFA). Nodes of the CFG represent DFA states, and edges represent the transitions of the DFA. The DFA's start-state corresponds to the entry-node of the CFG, and the accepting state to the CFG's exit node. In the remainder of this section we provide an overview of Kronecker algebra operations. Details and proofs of the stated properties can be found in [9, 10, 3, 12].

Kronecker product. Given an m-by-n matrix A and a p-by-q matrix B, their Kronecker product of size mp-by-nq is defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}. \tag{1}$$

The Kronecker product $A \otimes B$ represents a DFA where A and B execute in lock-step (i.e., A and B perform each transition simultaneously).

Kronecker sum. Given a square matrix A of order m and B of order n, their Kronecker sum denoted by $A \oplus B$ is a matrix of order $m \times n$ defined by

$$A \oplus B = A \otimes I_n + I_m \otimes B, \tag{2}$$

where I_m and I_n are identity matrices of order m and n respectively. The Kronecker sum generates all possible interleavings of DFAs A and B. Fig. 1 demonstrates a Kronecker sum of two binary semaphores S_1 and S_2. The semaphore DFAs are depicted in Fig. 1(b) and Fig. 1(c). Their sum $S_1 \oplus S_2$ is defined in Fig. 1(a). Fig. 1(d) depicts all possible interleavings of S_1 and S_2.

A concurrent system is defined by a tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where $\mathcal{T}$ is a set of threads, $\mathcal{S}$ is a set of synchronization primitives, and $\mathcal{L}$ are labels. For the ensemble of k threads $T^{(i)} \in \mathcal{T}$, we obtain a matrix T representing the thread interleavings. Similarly, for all synchronization primitives $S^{(j)} \in \mathcal{S}$, we obtain a matrix S representing the r interleaving semaphores.

Selective Kronecker product. Given an m-by-n matrix A and a p-by-q matrix B, we call $A \oslash_L B$ their selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let $A \oslash_L B = (a_{i,j}) \oslash_L (b_{r,s}) = (c_{t,u})$, where

$$c_{(i-1)\cdot p+r,(j-1)\cdot q+s} = \begin{cases} l \text{ if } a_{i,j} = b_{r,s} = l,\ l \in L, \\ 0 \text{ otherwise.} \end{cases}$$

The selective Kronecker product synchronizes identical labels $l \in \mathcal{L}_s$ of the left and right matrices. It ensures that a semaphore operation in the left operand is paired with the operation in the right operand.

An adjacency matrix representing a program P can be computed by

$$P = T \oslash_{\mathcal{L}_\mathrm{S}} S + T_{\mathcal{L}_\mathrm{V}} \otimes I_{o(S)}. \tag{3}$$

Therein, $\mathcal{L} = \mathcal{L}_\mathrm{V} \cup \mathcal{L}_\mathrm{S}$ describes the set of CFG edge labels $\mathcal{L}$ which is composed from the label-set $\mathcal{L}_\mathrm{S}$ representing calls to synchronization primitives (e.g., p, v), and the label-set $\mathcal{L}_\mathrm{V}$ that represents the remaining, non-synchronizing computations of a program (sets $\mathcal{L}_\mathrm{V}$ and $\mathcal{L}_\mathrm{S}$ are disjoint). Intuitively, given a concurrent system $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, matrix P describes all possible interleavings of the thread ensemble $\mathcal{T}$ under the constraints imposed from synchronizing with the synchronization primitives $\mathcal{S}$ [10].

3 Kronecker Algebra Evaluation

A deadlock manifests through unreachable components in a program's adjacency matrix [10]. Fig. 2(a) depicts a CFG with an invalid use of a binary semaphore: the p()-operations on the right path constitute a self-deadlock. With the second p()-operation, the semaphore is unobtainable, which results in self-deadlock at Node 6; Nodes 5 and 8 are unreachable from the starting node.

Considering two CFGs of m and n nodes, both the space- and time-complexity of a Kronecker sum or product operation that produces an adjacency matrix of m-by-n nodes is $\mathcal{O}(m^2n^2)$. An additional operation to a CFG of k nodes increases the complexities to $\mathcal{O}(k^2m^2n^2)$. Hence, the space- and time-complexities grow exponentially in the number of Kronecker operations.

Lazy evaluation of Kronecker operations delays all computations until proven to be required. (Only reachable nodes are analyzed.) In our example in Fig. 2(b), the reachable nodes (i.e., successors) from the starting node, Node 1, are represented by the non-zero entries in Row 1 of the 8×8 result-matrix, i.e., Node 4 and Node 6. Node 4 has one successor, Node 7. Nodes 6 and 7 have no successors and all reachable nodes have been visited, thus the analysis terminates. Because we only require non-zero elements of the result matrix, we restrict the evaluation of a Kronecker operation to the non-zero elements of its operands.

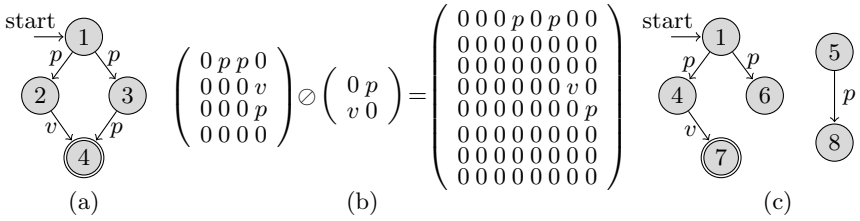


Fig. 2. Kronecker analysis of a CFG (a) with a self-deadlock (right path) on a binary semaphore; Kronecker algebra operation (b), and the resulting graph (c) showing an unreachable path (Node 5 and Node 8) from the starting node, Node 1. The self-deadlock manifests through the non-reachability of the exit-node (Node 7) from Node 6.

Lazy evaluation consists of two steps: a Kronecker expression is converted to an expression tree, followed by the lazy evaluation of the tree's operations. We employ sparse matrices for the adjacency matrix representation of CFGs and semaphores. They are operands of the algebra computation and constitute the leaf nodes of a tree. Successors of a node are retrievable by reading a sparse matrix. Kronecker operations constitute internal nodes. The intermediate results are stored as lazy matrices, which are represented by the algebra operator and pointers to the operands (i.e., children) in the tree. No actual computation happens until the result matrix is required, which allows us to optimize expression trees as they are constructed.

Lazy evaluation starts once the result matrix is required. Algorithm 1 computes all successors of a node. Node IDs and successor IDs of parent operations are constructable from IDs of child nodes and vice versa. Hence, from the starting node, we recursively find corresponding IDs of child nodes (Line 6–7) until the leaves are encountered. Successor IDs of leaf nodes are retrievable as they are presented as sparse matrices. Consequently, the successor IDs of parent nodes, then, are constructed in a bottom-up fashion. Newly found successors will later be evaluated. When no more successors are found, evaluation terminates. To keep track of processed nodes, we hash node IDs in a hash-table.

Kronecker Sum Optimizations: We can compute a Kronecker sum in a single step, to improve the formulation from Equation (2). An example sum of order two is stated in Equation (4). We observe three properties of the result matrix:

- The elements along the diagonal are the summations of the diagonal elements of the operands A and B.
- The first Kronecker product term in Equation (2) produces a dot product of elements from matrix A and I_n. The elements of matrix A are spread along the diagonal of sub-matrices of size n. Successors of a given node ID, x, must be n elements apart with an initial offset of $(x-1)\%n$.

Algorithm 1. Successor search

```
1 list succ(Matrix M, T node):
2  if M.getOperation != +:
3   sizeR= M.getRightMatSize
4   lid=((node-1)/sizeR)+1
5   rid=((node-1)%sizeR)+1
6   lsu=succ(M.leftMat, lid)
7   rsu=succ(M.rightMat, rid)
8  switch(M.getOperation):
9   case +:
10   su=succ(M.leftMat, node)
11   + succ(M.rightMat, node)
12  case ⊗: for l in lsu:
13   for r in rsu:
14    su.push((l-1)*sizeR+r)
15  case ⊘: for l in lsu:
16   for r in rsu:
17    if(label(l)=label(r)):
18     su.push((l-1)*sizeR+r)
19  case ⊕: for l in lsu:
20   su.push((l-1)*sizeR+rid)
21  for r in rsu:
22   su.push((lid-1)*sizeR+r)
23  case leafNodes:
24   su=readSparseMat(M, node)
25 return su
```

Algorithm 2. Successor search for Kronecker sums of same-sized operands

```
1 list KSumSuccs(SparseMat M,
2                T node,
3                int totalSize,
4                int nrKSums):
5 offset=0
6 size=M.size
7 for i in (0:nrKSums+1):
8  stripe=totalSize/size
9  leaf=succ(M, ⌊node/stripe⌋% M.size)
10 for su in leaf:
11  T sid=offset
12       +(su*stripe)
13       +(node%stripe)
14  su.push(sid)
15 offset+=⌊node/(offset+stripe)⌋*stripe
16 size*= size
17 return su
```

- The second Kronecker product term in Equation (2) duplicates matrix B along the diagonal of the result matrix. Given a node x, the successors must be located between 0 to n from $\lfloor \frac{(x-1)}{n} \rfloor * n$.

$$A \oplus B = \left(\begin{array}{cc|cc} a_1+b_1 & b_2 & a_2 & 0 \\ b_3 & a_1+b_4 & 0 & a_2 \\ \hline a_3 & 0 & a_4+b_1 & b_2 \\ 0 & a_3 & b_3 & a_4+b_4 \end{array}\right) = \left(\begin{array}{c|c} (a_1 \cdot I) + B & a_2 \cdot I \\ \hline a_3 \cdot I & (a_4 \cdot I) + B \end{array}\right) \quad (4)$$

From these properties, the successors of a Kronecker sum operand can be computed in one operation (see Algorithm 1). We further optimize sum operations for multiple adjacency matrices of the same order with non-zero values located at the same locations. This case mainly applies to sums of semaphores. Fig. 3 depicts the results of such Kronecker sums of two and three binary semaphores. These sums create a pattern that separates the p and v operations of different semaphores. Binary semaphore p and v are located at the upper-right and lower-left entry of a 2×2 adjacency matrix respectively. In Fig. 3(b), p_1 and v_1 of $S1$, the left most operand, appear along the diagonal of 4×4 sub-matrices at the upper-right and lower-left quadrants of the result matrix. Similarly, p_2

$$\left(\begin{array}{cc|cc} 0 & p_2 & p_1 & 0 \\ v_2 & 0 & 0 & p_1 \\ \hline v_1 & 0 & 0 & p_2 \\ 0 & v_1 & v_2 & 0 \end{array}\right) \qquad \left(\begin{array}{ccc|ccc} 0 & p_3 & p_2 & 0 & p_1 & 0 & 0 & 0 \\ v_3 & 0 & 0 & p_2 & 0 & p_1 & 0 & 0 \\ \hline v_2 & 0 & 0 & p_3 & 0 & 0 & p_1 & 0 \\ 0 & v_2 & v_3 & 0 & 0 & 0 & 0 & p_1 \\ \hline v_1 & 0 & 0 & 0 & 0 & p_3 & p_2 & 0 \\ 0 & v_1 & 0 & 0 & v_3 & 0 & 0 & p_2 \\ \hline 0 & 0 & v_1 & 0 & v_2 & 0 & 0 & p_3 \\ 0 & 0 & 0 & v_1 & 0 & v_2 & v_3 & 0 \end{array}\right)$$

(a) $S_1 \oplus S_2$ (b) $S_1 \oplus S_2 \oplus S_3$

Fig. 3. Kronecker sum of (a) two and (b) three binary semaphores.

and v_2 of $S2$ appear along the diagonal of 2×2 sub-matrices of the upper-left and lower-right 4×4 matrices. The pattern repeats for the remaining Kronecker sum operations, where the size of sub-matrices is reduced by half after each operation. Hence, given a node ID, the total number of Kronecker sums and the shape of the operands' sparse matrices, we can compute successors of Kronecker sums of same-sized matrices using Algorithm 2.

The lazy evaluation scheme only evaluates nonzero entries of reachable nodes in adjacency matrices. Hence, the time complexity is reduced to successor search. Considering a Kronecker operation produces a result matrix of size m-by-n nodes, a successor search takes up to $\mathcal{O}(pq)$, where the left child's matrix has p $\leq$ m successors and the right child's matrix has q $\leq$ n successors. The overall complexity is $\mathcal{O}(rpq)$, with r $\leq$ mn reachable nodes. In practice, the memory requirement is reduced to the number of reachable nodes, and memory occupied by nodes queued in unprocessed node queues.

4 Parallel Kronecker Algebra

The computation of finding successor nodes has a dependency between a node and its successors. Therefore, lazy evaluation is well-suited for parallelization where a worker thread processes per node instead of per successor. We employ hash-tables to record nodes that have already been processed, to avoid duplication of work. For example, with the CFG from Fig. 1(d), Node 2 will be encountered twice: as a successor of Node 1 and Node 4.

4.1 Multi-threaded CPU Implementation

With our lazy evaluation scheme, each worker thread maintains a local work-queue to store to-be-processed nodes. A new set of successors will be discovered during the evaluation of a node. We employ a hash-function which hashes the node IDs of the newly-discovered successors onto the work-queues of the worker threads. With the help of this hash-function, a worker thread will distribute the newly-discovered successors among the work-queues of all worker threads. We have implemented each work-queue as a lock-free queue, employing

the `boost::lockfree::queue` template of the boost C++ library (version 1.61). We use the row-index of a node in the adjacency matrix as the node ID (key) with this hashing operation. The hash-function guarantees that a node is assigned to exactly one worker.

Note that a worker thread will use the hash-tables of processed nodes to avoid duplication of work (i.e., re-processing of nodes). Because each worker thread processes a unique set of node IDs, it maintains a local hash-table of processed nodes. Only if a node is not already contained in the hash-table, it will be assigned for processing. If the table already contains a given node, it implies that this node has already been processed or is currently in a work-queue. Contrary, successful hashing (i.e., entering a new node into the hash-table) indicates that this node has not been encountered yet and must be assigned to a worker thread for processing.

4.2 GPU Implementation

As shown in Algorithm 1, the CPU implementation contains complex control-flow constructs including a switch-case statement and recursion. Because of potential branch divergence, such control-flow constructs are prone to low performance on a GPU. However, because all GPU threads will work on the same matrix algebra operation, they will branch to the same arm in the switch-case statement. Therefore, we found that the switch-case statement incurred very low branch divergence with our GPU kernel.

Similarly to the multicore CPU computations, we designed our GPU kernel such that one thread processes one node. A GPU has a considerably smaller memory than a CPU. As problem sizes grow, GPU memory is insufficient to keep track of all processed nodes. Hence, we maintain all computed thread interleavings on the CPU, using multiple threads. The list of processed nodes obtained from the GPU is unsorted. Thus, we utilize one hash-table of processed nodes in this implementation. Because the hash-table is highly contended, we employ lock-free synchronization to keep synchronization overhead low. We use libcuckoo, which is a lock-free hash-table of competitive performance [8]. In analogy to the multicore CPU implementation, if the table already contains a given node, it implies that this node has been processed or is currently in a work-queue. Contrary, successful hashing indicates a newly discovered node which must be assigned to a worker thread for processing.

Preprocessing: In the initial stage of the parallel execution, we prepare an expression tree and adjacency matrices for the GPU. Problem sizes vary in the number of input CFGs and CFG sizes. Each of these CFGs can be relatively small. A binary semaphore is represented as an adjacency matrix of size 2×2. Merging all matrices into a large buffer further decreases the data transfer overhead between CPU and GPU.

CFG edges are associated with labels in the form of strings. The length of such strings depends on the input-problem and hence cannot be determined at

Algorithm 3. Successor search on a GPU

```
1 void succ(T *BigSucc,T node,
2           optTree *opt,T *su):
3  nid[0]=node
4  for i in (1:NumOpts):
5   pid=nid[opt[i].parent]
6   nid[i]=getID(opt[i],pid)
7  for i in (NumOpts:0:-1):
8   su=succ(opt,nid[i],BigSucc)
```

Algorithm 4. Estimation of the maximum number of successors

```
1 T maxSucc(LazyMat M, T node):
2  lsu=maxSucc(M.leftMat,node)
3  rsu=maxSucc(M.rightMat,node)
4  switch(M.getOperation):
5   case +: n=lsu+rsu
6   case ⊗: n=lsu*rsu
7   case ⊘: n=min(lsu,rsu)
8   case ⊕: n=lsu*rsu
9  return n
```

compile-time. Performing string operations such as label comparisons with selective Kronecker products or string concatenation with matrix addition can be highly difficult and inefficient on a GPU. Thus, we re-code all string labels to a numeric representation. All labels are known at the time of constructing a problem's basic matrices. We implemented a lookup table which converts between string labels and numbers during the initial execution stage.

Eliminating Recursion: Because of the limited maximum recursion depth of GPUs, we replaced the recursive calls of our CPU implementation (Algorithm 1) by two loops as depicted in Lines 4 and 7 in Algorithm 3. All intermediate data, passed between recursive calls, is kept on stacks. The first loop pre-calculates node IDs at all levels of an expression tree in a top-down fashion. The node IDs of all intermediate and leaf matrices of the expression tree are stored in a fixed-length array. Based on the node ID information of the first loop and the known successors of the leaves, the second loop determines a list of successors in a bottom-up fashion. Function `succ()` in Line 8 is similar to Algorithm 1, without the recursive calls in Lines 6–7, and the successor of the plus operation being a combination of successors of previously calculated successors of a left- and right-hand child. The number of successors can vary between nodes. We preserve the maximum number of slots for each operation. The maximum number of successors can be estimated beforehand by Algorithm 4. Notably, the selective Kronecker product finds an exact label match of successors of the left and right child nodes. Thus, the number of matches cannot exceed the minimum number of labels of the two children.

Pipelined Execution Scheme: For further performance improvements, we exploit pipeline-parallelism between the CPU and the GPU. The GPU computation is split into iterations, where each iteration computes a fixed number of nodes. We choose heuristically optimal sizes according to GPU performance and memory usage. Depending on the expression tree, the stacks that store intermediate data on the GPU occupied most GPU memory.

Hashing on CPU cores happens in parallel to the GPU kernel computations. Because hashing requires a list of newly discovered successors from the GPU, it

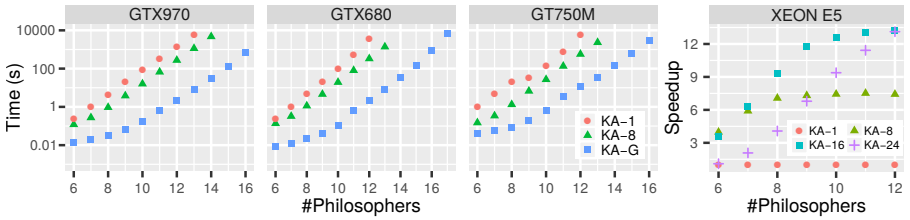


Fig. 4. Execution times on the GPU systems and speedup over KA-1 on the multicore system for the Dining Philosophers problem

is not guaranteed to retrieve data in time from the current iteration. Hence, we delayed the CPU computation by one iteration. CPU threads always hash nodes from the previous iteration.

5 Experimental Results

We have evaluated our method on one multicore system and three desktop/embedded GPU systems (platform specifications are stated in Table 1). With performance measurements, we have omitted file I/O and GPU context creation times, because they do not reflect the actual computation times. For measurement consistency, we do not stop our analysis when a deadlock is detected. Rather, we have the analysis compute all possible, legal thread interleavings (the entire problem size). Our rationale was that with parallel execution, where multiple nodes are being analyzed concurrently, the node processing order may vary, and a deadlock may be detected earlier or later.

We perform deadlock analysis on one synthetic and two real-world examples: (1) Dijkstra's Dining Philosopher's, (2) Linux kernel threads, and a (3) railway system. We verify the correctness of our optimizations to the unoptimized implementation proposed in [10]. With the first problem, we employed n philosophers with n forks placed between them. Each philosopher constitutes an individual thread; each fork is represented by a binary semaphore. This set-up will result in the well-known deadlock if all n philosophers pick up their left fork simultaneously. For the remainder of this section, we refer to the multi-threaded implementation as KA-N for N threads, and to the GPU implementation as KA-G.

We use the sequential implementation as our yardstick. Fig. 4 shows the obtained execution times on the GPU systems. Note that the y-axis is in log-scale: as the number of philosophers (threads) increases, the number of nodes to be processed grows exponentially. Five philosophers generate 392 nodes reachable from the starting node, while 17 philosophers result in more than 662 million reachable nodes. Once a problem becomes sufficiently large to fully utilize the GPU, the GPU implementation shows the best performance. Due to the long execution time of the single-threaded CPU version, we were unable to measure our yardstick up to as many philosophers as with the parallel versions. The GPU implementation is 140×, 176× and 51× faster than the multi-threaded imple-

mentation on the GTX 970, GTX 680 and GT 750M systems. Scalability of the multi-threaded version is depicted for the XEON E5 platform. The speedups of KA-8 and KA-16 over single-threaded execution saturate at 7.5× and 13.1×, respectively. For 12 philosophers, KA-24 is on par with KA-16, but on an upward trend (whereas KA-16's scalability is already saturated).

We applied Kronecker algebra to detect deadlocks in Linux kernel threads (for Linux kernel version 3.10). We analyze two ensembles of three and five kernel threads that share eight and ten locks. LLVM was used to inline called functions into kernel threads and to obtain CFGs. Computations unrelated to synchronization were pruned from CFGs, because they are irrelevant for deadlock analysis. The details of this automated conversion are outside the scope of this paper and explained in [15].

Kronecker-based deadlock analysis of railway systems has been introduced in [14]. Train routes and track sections can be viewed as thread CFGs and semaphores. At most one train can use a section at any time. During the occupation, a section is locked as if using a semaphore p()-operation. Once the train leaves the section, the lock is released. We analyze a train system with six train routes and twelve sections illustrated in Fig. 5. Each route constitutes a thread. E.g., Route 1 is defined as $L_1 = p_5, v_2, p_6, p_9, v_5, v_6, p_{11}, v_9, p_{12}, v_{11}, v_{12}$. The double-slip switch, located between sections 4, 6, 7 and 9, is replaced with two switches connected by Section 6 of zero-length. If a train is too long to fit in a section, a sufficient number of sections ahead must be reserved. E.g., in Route L1, because Section 6 has zero-length, Section 9 is reserved ahead of time.

The analysis times obtained for Linux kernel threads and the railway system are depicted in Table 2. Sequential execution was intractable and has been excluded. The speedups of the GPU implementation of up to 5453× are proportional to the problem sizes and memory requirements. The GPU kernel computes in parallel with the hashing of processed nodes on the CPU. Hence, the GPU computation is "*hidden*" behind the CPU computation.

We compared our GPU implementation, KA-G, to SPIN, the state-of-the-art model checker [5] (version 6.4.5), in Fig. 6 and Table 2. We conducted experi-

Table 1. Evaluation Platform Specifications.

		GTX680	**GT750M**	**GTX970**	**Xeon E5**
CPU	Model	i7-3770k	i7-4850HQ	i7-6700	Xeon E5-2697
	# cores	4 (8 threads)	4 (8 threads)	4 (8 threads)	28 (56 threads)
	Memory	32 GB	16 GB	16 GB	256 GB
	Clock freq.	3.5 GHz	2.3 GHz	3.4 GHz	1.8 GHz
	Bandwidth	15.9 GB/s	11.4 GB/s	20.4 GB/s	12.0 GB/s
GPU	Model	GeForce GTX 680	GeForce GT 750M	GeForce GTX 970	N/A
	# cores	1536	384	1664	
	Memory	2048 MB	1971 MB	4029 MB	
	Clock freq.	1006 MHz	926 MHz	1329 MHz	
	Bandwidth	12.1 GB/s	6.3 GB/s	12.7 GB/s	

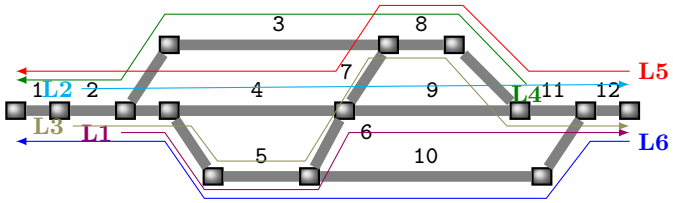


Fig. 5. A train system with six train routes and twelve track sections.

Table 2. Performance of multi-threaded SPIN (SPIN-8) and our multi-threaded (KA-8) and GPU (KA-G) implementations on three practical problems (times in seconds).

		Linux (3 threads)	Linux (5 threads)	Railway
GTX970	SPIN-8	N/A	N/A	1.64
	KA-8	154	68024	4.15
	KA-G	0.30	14.64	0.050
GTX680	SPIN-8	N/A	N/A	1.68
	KA-8	181	77107	4.89
	KA-G	0.29	14.14	0.049
GT750M	SPIN-8	N/A	N/A	2.05
	KA-8	316	144321	5.86
	KA-G	0.52	69.10	0.066

ments using one (SPIN-1) and eight threads (SPIN-8). For Dining Philosopher's, KA-G is up to 1.9× faster than SPIN-1. However, it is struggling to outperform the multi-threaded version of SPIN. In the analysis of railway systems, however, KA-G is 33× faster than multi-threaded SPIN.

We have observed that the Kronecker algebra approach consumes considerably less memory than SPIN. SPIN reports its total memory usage, and we use the PAPI library and the CUDA API to observe the memory consumption of our approach. The smaller CPU memory footprint allows our approach to analyze larger problem sizes. E.g., the performance comparison reported in Fig. 6 had to be limited to a maximum of 15 philosophers; With 16 philosophers, SPIN-1's memory consumption exceeded the 32 GB RAM of our test-platform, and performance trashed as a result of excessive swapping. (We stopped SPIN on this test-run after 48 hours.) In comparison, KA-G stores some intermediate buffers in the GPU memory space; consequently, our approach handled 17 philosophers in less than 24 GB of RAM. SPIN-8 has higher memory requirements and is thus unable to handle more than 14 philosophers.

The relative CPU and GPU memory consumption of KA-G to SPIN is shown in Fig. 7. Up to 12 philosophers, the GPU computation of KA-G consumes more memory than SPIN-1, because memory must be reserved for each GPU thread. The GPU kernel operates in iterations, where all buffers are reused. Thus, the GPU memory requirement does not grow with the problem size. In fact, the GPU memory consumption grows linearly in proportion to the number of Kronecker algebra operations while the CPU memory consumption grows in the number of

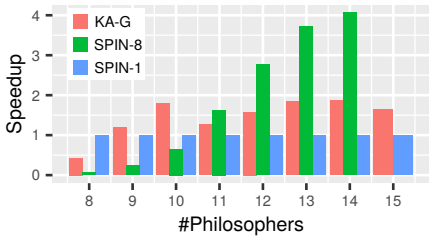


Fig. 6. Speedup comparison of our GPU implementation to SPIN.

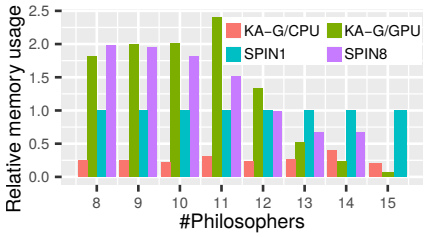


Fig. 7. Relative memory consumption comparison of KA-G and SPIN-8 with respect to SPIN-1

reachable nodes. Because the CPU and GPU memory spaces are disjoint and the memory consumption on the GPU does not constrain the problem size, we only compare our CPU memory consumption to SPIN. Our implementation consumes up to 4.8× and 8.1× less memory than SPIN-1 and SPIN-8, respectively.

6 Related Work

Kronecker algebra-based concurrent program verification has been introduced in [10] and subsequently extended to support worst-case execution time analysis [11, 13], and the analysis of protected objects, semaphores and barriers [9, 10, 3, 12]. Unlike prior work, this paper addresses performance improvements of Kronecker algebra operations, to cope with real-world static analysis problem sizes. The SPIN model checker [5, 6] is a verification tool for concurrent programs. Spin employs state-transition graphs and depth-first search to check a program's safety and liveness properties. Bartocci [1] has parallelized SPIN on a GPU, such that computed thread interleavings are located in the GPU's memory. The limited memory space of GPUs constrains the solvable problem size to 15 philosophers. Kronecker algebra allows us to store thread interleavings on the CPU and thereby analyze larger problem sizes.

7 Conclusions

We have optimized Kronecker algebra operations for heterogeneous multicores. Our two-step lazy-evaluation approach constructs expression trees, followed by parallel, lazy evaluation. Lazy evaluation substantially speeds up analysis by omitting all unreachable nodes. Our pipelined execution scheme performs lazy Kronecker algebra operation evaluation on the GPU and uses the CPU to maintain the computed thread interleavings. Our experiments show speedups of up to 5453× over the multicore CPU implementation. Our implementation consumes up to 8.1x less memory than SPIN and can therefore analyze larger problem sizes.

Acknowledgments. This research was supported by the Austrian Science Fund (FWF) project I 1035N23, and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015M3C4A7065522.

References

1. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-parallel SPIN Model Checker. In: 2014 International SPIN Symposium on Model Checking of Software. pp. 87–96. ACM (2014)
2. Buchholz, P., Kemper, P.: Efficient Computation and Representation of Large Reachability Sets for Composed Automata. Discrete Event Dyn. Systems 12(3), 265–286 (2002)
3. Burgstaller, B., Blieberger, J.: Kronecker Algebra for Static Analysis of Ada Programs with Protected Objects. In: Ada-Europe International Conference on Reliable Software Technologies. vol. 8454, pp. 27–42 (2014)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Progress on the State Explosion Problem in Model Checking. In: Informatics - 10 Years Back. 10 Years Ahead. pp. 176–194. Springer (2001)
5. Holzmann, G.J.: The Model Checker SPIN. vol. 23. IEEE Computer Society (1997)
6. Holzmann, G.J., Bosnacki, D.: The Design of a Multicore Extension of the SPIN Model Checker. IEEE Transactions on Software Engineering 33, 659–674 (2007)
7. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. Springer (1986)
8. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14, ACM (2014)
9. Mittermayr, R., Blieberger, J.: Static Partial-Order Reduction of Concurrent Systems in Polynomial Time. In: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. pp. 619–633. Springer (2008)
10. Mittermayr, R., Blieberger, J.: Shared Memory Concurrent System Verification using Kronecker Algebra. Tech. Rep. 183/1-155, TU Vienna, `http://arxiv.org/abs/1109.5522` (Sept 2011)
11. Mittermayr, R., Blieberger, J.: Timing Analysis of Concurrent Programs. In: 12th International Workshop on Worst-Case Execution Time Analysis. pp. 59–68 (2012)
12. Mittermayr, R., Blieberger, J.: Kronecker algebra for static analysis of barriers in Ada. In: Ada-Europe International Conference on Reliable Software Technologies. pp. 145–159. Springer (2016)
13. Mittermayr, R., Blieberger, J.: Deadlock and wcet analysis of barrier-synchronized concurrent programs. Computing pp. 1–22 (2017)
14. Mittermayr, R., Blieberger, J., Schöbel, A.: Kronecker algebra-based deadlock analysis for railway systems. PROMET-Traffic&Transportation 24(5), 359–369 (2012)
15. Park, Y.: Kronecker algebra-based deadlock analysis in the Linux kernel. Tech. rep., Yonsei University, `http://elc.yonsei.ac.kr/publications/KernelDeadlockAnalysis.pdf` (Dec 2016)
16. Plateau, B.: On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In: ACM SIGMETRICS. vol. 13, pp. 147–154 (1985)