Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Stephen W. Abell

Entitled
Parallel Acceleration of Deadlock Detection and Avoidance Algorithms on GPUs

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Dr. John Jaehwan Lee	
Chair	
Dr. Brian King	
Dr. Stanley Chien	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. John Jaehwan Lee

Approved by: Dr. Brian King 06/26/2013
Head of the Graduate Program Date

PARALLEL ACCELERATION OF DEADLOCK DETECTION AND AVOIDANCE ALGORITHMS ON GPUS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Stephen W. Abell

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2013

Purdue University

Indianapolis, Indiana

ACKNOWLEDGMENTS

First and foremost, I must thank my advisor Dr. John Lee. His dedication to innovative and thorough research is what led to the accomplishments in this thesis. I must thank him for our many meetings, technical discussions, and paper revisions. All of which have helped me grow into a more thorough engineer.

I must thank my family and girlfriend for their undying support and confidence in me. They have always stood beside me in my decision to seek higher education and believed in my abilities as an engineer.

Lastly, I would like to thank Sherrie Tucker. Sherrie has always been helpful when it comes to logistical aspects of my graduate career and has always kept me on track. Also, her seemingly endless supply of coffee and candy has been a huge aid in my success as a masters student.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Abell, Stephen W. MSECE, Purdue University, August 2013. Parallel Acceleration of Deadlock Detection and Avoidance Algorithms on GPUs. Major Professor: Dr. John Jaehwan Lee.

Current mainstream computing systems have become increasingly complex. Most of which have Central Processing Units (CPUs) that invoke multiple threads for their computing tasks. The growing issue with these systems is resource contention and with resource contention comes the risk of encountering a deadlock status in the system. Various software and hardware approaches exist that implement deadlock detection/avoidance techniques; however, they lack either the speed or problem size capability needed for real-time systems.

The research conducted for this thesis aims to resolve issues present in past approaches by converging the two platforms (software and hardware) by means of the Graphics Processing Unit (GPU). Presented in this thesis are two GPU-based deadlock detection algorithms and one GPU-based deadlock avoidance algorithm. These GPU-based algorithms are: (i) GPU-OSDDA: A GPU-based Single Unit Resource Deadlock Detection Algorithm, (ii) GPU-LMDDA: A GPU-based Multi-Unit Resource Deadlock Detection Algorithm, and (iii) GPU-PBA: A GPU-based Deadlock Avoidance Algorithm.

Both GPU-OSDDA and GPU-LMDDA utilize the Resource Allocation Graph (RAG) to represent resource allocation status in the system. However, the RAG is represented using integer-length bit-vectors. The advantages brought forth by this approach are plenty: (i) less memory required for algorithm matrices, (ii) 32 computations performed per instruction (in most cases), and (iii) allows our algorithms to handle large numbers of processes and resources. The deadlock detection algorithms also require minimal interaction with the CPU by implementing matrix storage and algorithm computations on the GPU, thus providing an interactive service type of behavior. As a result of this approach, both algorithms were able to achieve speedups over two orders of magnitude higher than their serial CPU implementations (3.17-317.42x for GPU-OSDDA and 37.17-812.50x for GPU-LMDDA). Lastly, GPU-PBA is the first parallel deadlock avoidance algorithm implemented on the GPU. While it does not achieve two orders of magnitude speedup over its CPU implementation, it does provide a platform for future deadlock avoidance research for the GPU.

1 INTRODUCTION

1.1 Problem Statement

Modern computing platforms have grown increasingly complex over the past decade. The advent of multi-cored and multi-threaded Central Processing Units (CPUs) have reintroduced a common problem regarding resource contention, deadlock. In the past, many software-based deadlock detection/avoidance approaches were devised that handled both single-unit and multi-unit request systems. The issue with these deadlock detection solutions was that they were not capable of determining events that led to deadlock in a deterministic and expedited manner. These solutions typically had poor run-time complexities on the order of $\mathcal{O}(M \times N)$, $\mathcal{O}(N^2)$, or $\mathcal{O}(N^3)$, where M and N are process and resource counts, respectively.

As a result, researchers began developing hardware solutions to the deadlock detection problem. The key advantage to these hardware techniques was their ability to exploit parallelism in the Resource Allocation Graph (RAG) while performing deadlock detection computations. Since these parallel computations took place in hardware, the algorithms devised had low run-time complexities, i.e., $\mathcal{O}(log_2(min(M,N)))$ and $\mathcal{O}(1)$. The only issue with applying these hardware solutions to real world systems was the inability to handle increasingly large numbers of processes and resources. By increasing the number of processes and resources, the size of the hardware solutions would grow polynomially, as would their cost.

Furthermore, in the past 7 years their has been a paradigm shift in the way researchers, scientists, and software engineers handle parallel computations. Graphics card manufacturers (Nvidia and ATI) realized, with the help of the hardware and software communities, that their massively parallel hardware platforms, or Graphics Processing Units (GPUs), could be used as a large vector processor. Nvidia then created their Compute Unified Device Architecture (CUDA) framework. This development has given researchers the capabilities of massively parallel processors in a discrete package that fits into most common personal computers (PC). This leads to the research conducted for this thesis. By thinking about the previously developed hardware algorithms in terms of software, it was hypothesized that those same algorithms could be developed for the modern GPU. As a result, several deadlock detection/avoidance algorithms could be utilized for real world systems that would yield large speedups with respect to a CPU implementation. Since the GPU is a secondary device to the CPU, limited interaction with the CPU would be required so that it may continue performing its normal tasking.

1.2 Terminology

This section defines terms and theorems in the CUDA framework, deadlock detection and graph theory domains that are applicable to this thesis.

1.2.1 Definitions in the CUDA Framework

Definition 1.2.1 API is the acronym for an Application Programming Interface.

Definition 1.2.2 CUDA is the acronym for Nvidia's Compute Unified Device Architecture. The CUDA framework consists of both the hardware architecture and the parallel programming model provided by Nvidia.

Definition 1.2.3 GPU is the acronym for a Graphics Processing Unit.

Definition 1.2.4 A Streaming Multiprocessor (SM) is a multi-cored processing unit on the GPU.

Definition 1.2.5 A Stream Processor (SP) is one of many execution units on the SM of a GPU.

Definition 1.2.6 A thread is a single path of execution that takes place on an SP inside of an SM.

Definition 1.2.7 A block is a grouping of threads that are scheduled on multiple SP's inside of a single SM.

Definition 1.2.8 A grid consists of many groupings of thread blocks that can be distributed across many SP's of many SM's.

Definition 1.2.9 A warp is the lowest level grouping of threads that are scheduled on the GPU. The warp consists of 32 threads.

Definition 1.2.10 A kernel is the programmer derived software function to be launched on the GPU.

1.2.2 Definitions in the Deadlock Domain

Definition 1.2.11 Deadlock is a situation in which a set of processes are permanently blocked for competing for system resources or communicating with each other. The following are necessary and sufficient conditions for deadlock to occur:

Condition 1 Mutual exclusion condition: a resource (unit) is either assigned to one process or it is available.

Condition 2 Hold and wait condition: processes already holding resources may request additional resources.

Condition 3 No preemption condition: only a process holding a resource can release it.

Condition 4 Circular wait condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

Definition 1.2.12 Deadlock detection is a way of dealing with deadlock that tracks resource allocation and process states to find deadlock, and rolls back and restarts one or more of the processes in order to remove the deadlock.

Definition 1.2.13 A single-unit resource is a resource that serves at most one process at a time. This means while a single-unit resource is serving a process, all other processes requesting this resource must wait.

Definition 1.2.14 A multi-unit resource is a resource that can serve one or more processes at the same time. All processes are served with the same or similar functionality.

Definition 1.2.15 An active process is a process which has no outgoing edge (no pending resource request) but may have incoming edges (granted resources).

Definition 1.2.16 A system is in an expedient state if any request for available units is granted immediately.

Definition 1.2.17 A single-unit request system is a system in which a process may request only one unit at a time, and thus has at most one outstanding request for a single unit of some resource.

Definition 1.2.18 An H-Safe sequence is an enumeration $p_1, p_2, \ldots, p_N$ of all processes in the system, such that for each i=$1, 2, \ldots, N$, the resources that p_i may request are a subset of the union of resources that are currently available and resources currently held by $p_1, p_2, \ldots, p_{i-1}$ [1] [2].

Definition 1.2.19 An H-Safe state exists if and only if there exists an H-Safe sequence $p_1, p_2, \ldots, p_N$. If there is no H-Safe sequence, the system is in an H-Unsafe state [2].

1.2.3 Definitions in the Graph Theory Domain

In this section, we first describe the Resource Allocation Graph (RAG), followed by definitions from graph theory that are used throughout the algorithms in this thesis.

Definition 1.2.20 Let $P = \{p_1, p_2, \ldots, p_M\}$ be a set of M requestors or processes that may request and/or hold a number of resources at any given time.

Definition 1.2.21 Let $Q = \{q_1, q_2, \ldots, q_N\}$ be a set of N resources that provide specific functions usable by processes. Each resource consists of a fixed number (one or more) of units to supply.

Definition 1.2.22 Let the set of nodes V be $P \cup Q$, which is divided into two disjoint subsets P and Q such that $P \cap Q = \emptyset$. Another notation for the set of nodes is, $V = \{v_1, v_2, \ldots, v_l\}$.

Definition 1.2.23 Let G be a set of grant edges. Let an ordered pair (q_j, p_i) be a grant edge where the first node is a resource $q_j \in Q$, the second node is a process $p_i \in P$, and q_j has been granted to p_i. Thus, a set of grant edges G can be written as $G = \{(q_j, p_i)|\ j \in \{1, 2, \ldots, n\},\ i \in \{1, 2, \ldots, m\}$, and resource q_j has been granted to process $p_i\}$. An ordered pair (q_j, p_i) can also be represented by $q_j \rightarrow p_i$.

Definition 1.2.24 Let R be the set of request edges. Let an ordered pair (p_i, q_j) be a request edge where the first node is a process $p_i \in P$, the second node is a resource $q_j \in Q$, and p_i has requested q_j but has not yet acquired it. Thus, a set of request edges R can be written as $R = \{(p_i, q_j)|\ i \in \{1, 2, \ldots, m\},\ j \in \{1, 2, \ldots, n\}$, and process p_i is blocked for requesting resource $q_j\}$. An ordered pair (p_i, q_j) can also be represented by $p_i \rightarrow q_j$.

Definition 1.2.25 Let the set of edges E be $R \cup G$. Another notation used is, $E = \{e_1, e_2, \ldots, e_h\}$.

Definition 1.2.26 A particular resource allocation situation in a given system with processes and resources can be abstracted by a Resource Allocation Graph (RAG). A RAG is a directed graph $\theta = \{V, E\}$, such that V is a non-empty set of nodes defined in Definition 1.2.22, and E is a set of edges defined in Definition 1.2.25. The RAG is also split into two separate adjacency matrices: Adjacency Request (AR) and Adjacency Grant (AG), which hold the resource request and grant information, respectively. In this thesis we assume a RAG handles single-unit resources (see Definition 1.2.13).

Definition 1.2.27 The Adjacency Request matrix is a component of the RAG. It contains request edges (see Definition 1.2.24) that denote that a process p_i has requested a resource q_j.

Definition 1.2.28 The Adjacency Grant matrix is the second component of the RAG. It contains grant edges (see Definition 1.2.23) that denote that a resource q_j has been granted to process p_i.

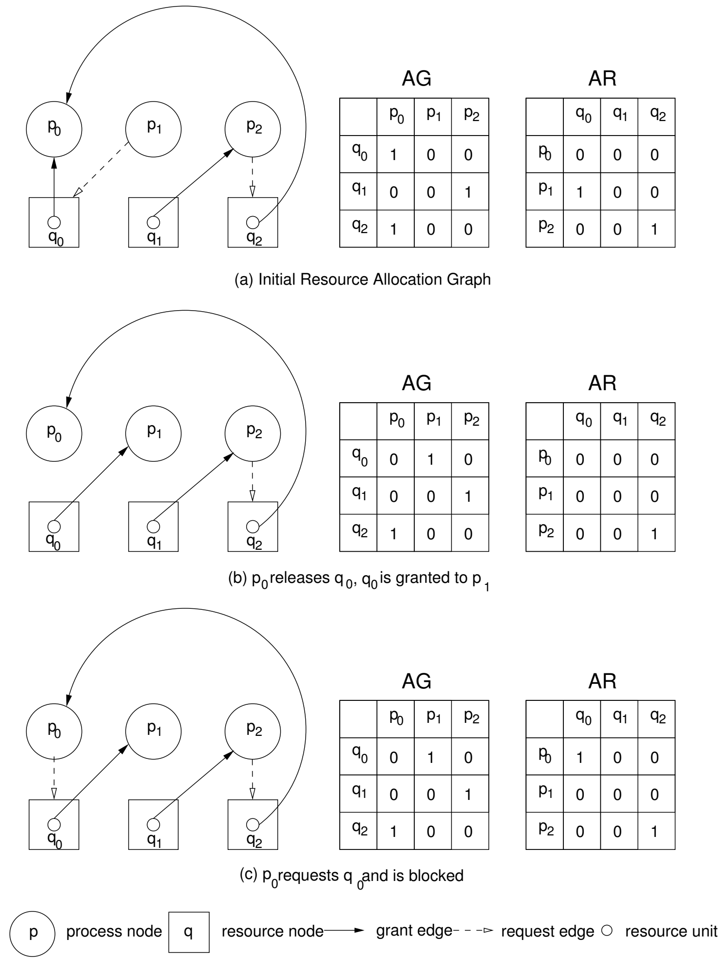
Summary of AG/AR for RAG Handling Single-Unit Resources

$$AG[j][i] = \begin{cases} 1 & \text{if } \exists q_j \rightarrow p_i, \\ 0 & \text{otherwise.} \end{cases} \qquad AR[i][j] = \begin{cases} 1 & \text{if } \exists p_i \rightarrow q_j, \\ 0 & \text{otherwise.} \end{cases}$$

RAG Example

Figure 1.1(a) shows an example RAG of a 3×3 system consisting of three processes (p_0, p_1, p_2) and three resources (q_0, q_1, q_2). Since we assume a RAG handles a single-unit system, each resource has one unit and grant edge weights do not exceed 1. Accompanying the RAG in Figure 1.1 are the associated adjacency matrices AG and AR to reflect resource allocation status in the system.

In the RAG, there exist three *resource grant* edges ($q_0 \rightarrow p_0$, $q_1 \rightarrow p_2$, and $q_2 \rightarrow p_0$) and two *resource request* edges ($p_1 \rightarrow q_0$ and $p_2 \rightarrow q_2$). Further, by looking at Figure 1.1(b), it can be seen that under the resource release event where p_0 releases q_0, q_0 is then granted to the blocked process p_1. This is an example of the system being



AG

	p_0	p_1	p_2
q_0	1	0	0
q_1	0	0	1
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	0	0	0
p_1	1	0	0
p_2	0	0	1

(a) Initial Resource Allocation Graph

AG

	p_0	p_1	p_2
q_0	0	1	0
q_1	0	0	1
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	0	0	0
p_1	0	0	0
p_2	0	0	1

(b) p_0 releases q_0, q_0 is granted to p_1

AG

	p_0	p_1	p_2
q_0	0	1	0
q_1	0	0	1
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	1	0	0
p_1	0	0	0
p_2	0	0	1

(c) p_0 requests q_0 and is blocked

Figure 1.1: A 3×3 RAG incurring resource events.

in an *expedient* state (see Definition 1.2.16). Notice also that AG and AR have been updated accordingly. Lastly, in Figure 1.1(c), process p_0 requests q_0 and is blocked due to q_0 having been granted to process p_1.

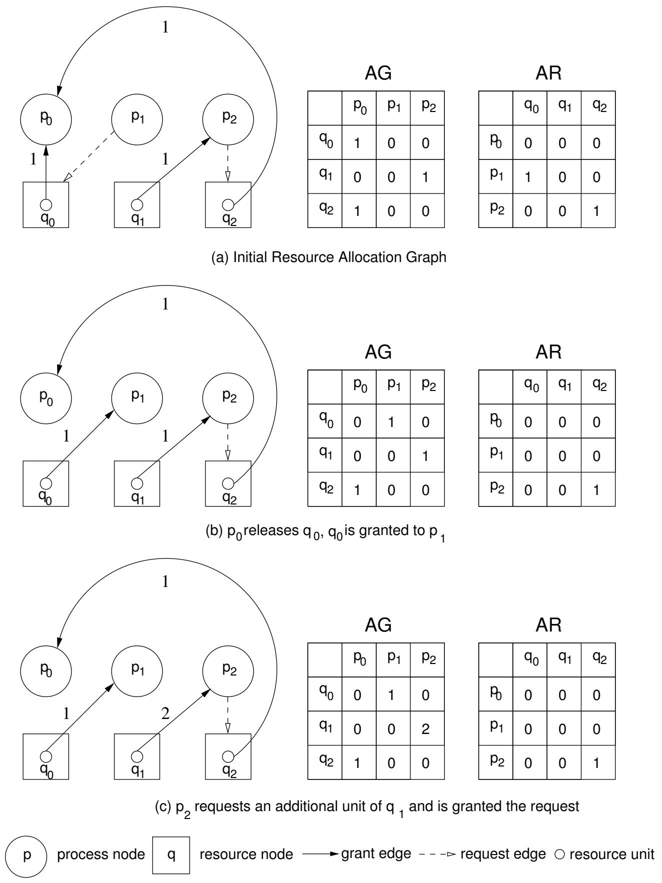
Definition 1.2.29 A weighted RAG is a Resource Allocation Graph (see Definition 1.2.26) whose grant edges can have a weight (or value) greater than 1. In other words, a weighted RAG handles multi-unit resources (see Definition 1.2.14).

Summary of AG/AR for Weighted RAG Handling Multi-Unit Resources

$$AG[j][i] = \begin{cases} w(q_j, p_i) & \text{if } \exists q_j \rightarrow p_i, \\ 0 & \text{otherwise.} \end{cases} \qquad AR[i][j] = \begin{cases} 1 & \text{if } \exists p_i \rightarrow q_j, \\ 0 & \text{otherwise.} \end{cases}$$

Weighted RAG Example

Figure 1.2(a) shows an example RAG of a 3×3 system consisting of three processes (p_0, p_1, p_2) and three resources (q_0, q_1, q_2). Since the weighted RAG handles multi-unit systems, the edge weights in the RAG may exceed 1. Accompanying the RAG in Figure 1.2 are the associated adjacency matrices AG and AR to reflect the resource allocation status in the system. In the RAG, there exist three *resource grant* edges ($q_0{\rightarrow}p_0$, $q_1{\rightarrow}p_2$, and $q_2{\rightarrow}p_0$) and two *resource request* edges ($p_1{\rightarrow}q_0$ and $p_2{\rightarrow}q_2$).



AG

	p_0	p_1	p_2
q_0	1	0	0
q_1	0	0	1
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	0	0	0
p_1	1	0	0
p_2	0	0	1

(a) Initial Resource Allocation Graph

AG

	p_0	p_1	p_2
q_0	0	1	0
q_1	0	0	1
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	0	0	0
p_1	0	0	0
p_2	0	0	1

(b) p_0 releases q_0, q_0 is granted to p_1

AG

	p_0	p_1	p_2
q_0	0	1	0
q_1	0	0	2
q_2	1	0	0

AR

	q_0	q_1	q_2
p_0	0	0	0
p_1	0	0	0
p_2	0	0	1

(c) p_2 requests an additional unit of q_1 and is granted the request

Figure 1.2: A 3×3 Weighted RAG incurring resource events.

Further, by looking at Figure 1.2(b), it can be seen that under the resource release event where p_0 releases q_0, q_0 is then granted to the blocked process p_1. This is an example of the system being in an *expedient* state (see Definition 1.2.16). Notice also that AG and AR have been updated accordingly. Lastly, in Figure 1.2(c), process p_2 requests an additional unit of q_1 and the resource is granted (assuming q_1 has available units), increasing the weight of the grant edge $q_1 \rightarrow p_2$. If no additional units of q_1 were available, the resource request $p_2 \rightarrow q_1$ would be blcoked.

Definition 1.2.30 A bipartite graph is a graph whose nodes can be divided into two disjoint sets V_1 and V_2 such that every edge connects a node in V_1 and one in V_2. That is, there is no edge between two nodes in the same set.

Definition 1.2.31 A cycle is made up of a simple path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_{j-1} \rightarrow v_j$ and an additional edge $v_j \rightarrow v_1$.

Definition 1.2.32 A node v_i is reachable from a node v_j if there exists a path that starts from v_j and ends at v_i. Thus, v_i is called a reachable node of v_j [3].

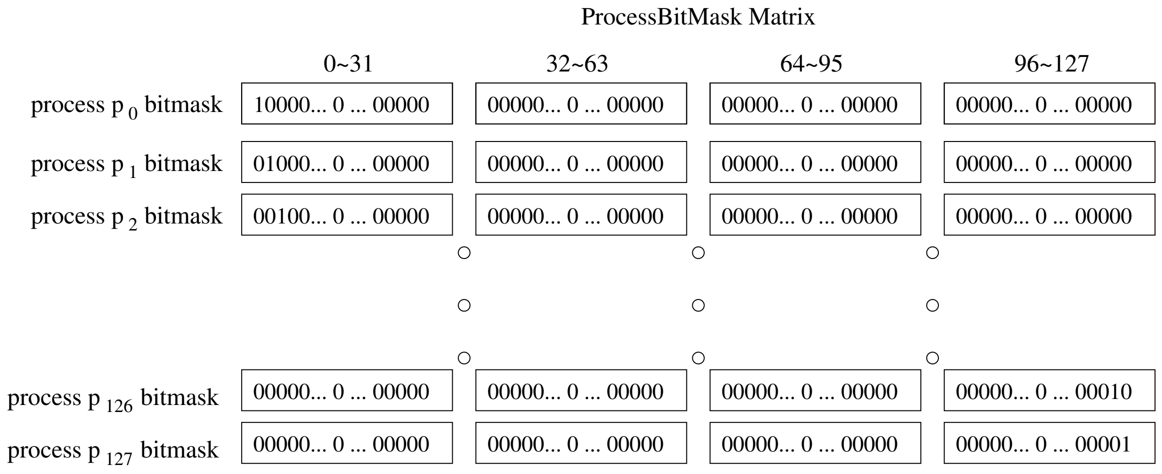
Definition 1.2.33 The reachable set of node v_j is a set of nodes such that a path exists from v_j to every node in the reachable set [3].

Definition 1.2.34 A knot is a nonempty set K of nodes such that the reachable set of each node in K is exactly K [3].

Definition 1.2.35 A sink node is a node that has at least one incoming edge but does not have any outgoing edge.

Definition 1.2.36 The process bitmask matrix is an $M \times M$ matrix in which the i^{th} row is equal to p_i's bitmask. The process bitmask of process p_i is an M-bit binary value 0. . .010. . .0 in which only the i^{th} bit is 1, and all others are zeros.

Example of a process bit-mask matrix



ProcessBitMask Matrix

	0~31	32~63	64~95	96~127
process p_0 bitmask	10000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000
process p_1 bitmask	01000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000
process p_2 bitmask	00100... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000
process p_{126} bitmask	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00010
process p_{127} bitmask	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00000	00000... 0 ... 00001

Figure 1.3: The process bit-mask matrix represented with bit-vectors.

Definition 1.2.37 A sink bitmask for the reachable sink nodes of a resource q_j is an M-bit binary value 0...0101...0 in which its i^{th} bit is 1 if process p_i is a reachable sink node of q_j in the RAG, where $1 \leq i \leq M$. Otherwise, the i^{th} bit is 0.

1.2.4 Theorems for Deadlock Detection

Defined here are two theorems that provide theoretical support for the deadlock algorithms which were used to construct the GPU-based algorithms described in this thesis.

Theorem 1.2.38 A single-unit request system consisting of only single-unit resources is in a deadlock state if and only if its corresponding RAG contains a cycle.

Proof The proof of this theorem can be found in [3].

Theorem 1.2.39 A single-unit request system consisting of general resources (i.e., including both single-unit and multi-unit resources) is in a deadlock state if and only if its corresponding RAG contains a knot.

Proof The proof of this theorem can be found in [3].

1.3 Related Work

There have been a multitude of software-based deadlock detection algorithms proposed in the past that handle resource events in single-unit resource systems. In 1972, Holt [3] first introduced a RAG-based deadlock detection approach with an $\mathcal{O}(M \times N)$ run-time complexity. Following this development, Leibfried [4] devised an algorithm that utilized the adjacency matrix. Leibfried's approach used matrix multiplication in order to determine reachability information which led to an algorithm with an $\mathcal{O}(M^3)$ run-time complexity. Later, Kim and Koh [5] devised a tree-based algorithm that improved upon the prior deadlock detection run-time. Their tree-based algorithm was able to detect deadlock in $\mathcal{O}(1)$ run-time; however, the caveat to this approach was that it required an $\mathcal{O}(M+N)$ run-time for the resource release phase of the algorithm. The completion of this release phase was required for the algorithm to handle the next invocation of deadlock detection.

Regarding systems with multi-unit resources, Shoshani et al. devised a deadlock detection algorithm in 1969 that achieved a run-time complexity of $\mathcal{O}(M^2 \times N)$ where M and N are the number of processes and resources, respectively [6]. This approach also utilized the Resource Allocation Graph (RAG) to maintain resource allocation information in the system. The use of the RAG as an abstraction of resource allocation information in systems became quite popular in the deadlock detection domain following this development. As a result, in 1972, Holt devised his own approach to deadlock detection in multi-unit resource systems using the RAG. Holt's approach led to an improved run-time of $\mathcal{O}(M \times N)$ [3]. Then in 1997, Kim introduced an algorithm that was split into multiple phases: preparation and detection [7]. The preparation phase of this algorithm was required for future iterations of detection following a resource event. The result of Kim's work yielded an algorithm with a deadlock preparation run-time of $\mathcal{O}(M \times N)$ and a deadlock detection run-time of $\mathcal{O}(1)$.

In recent years, there has been a progression towards parallel hardware-based algorithms to detect deadlock. These algorithms are deterministic and have accomplished low run-time complexities in hardware. In 2001, Shui et al. devised a deadlock detection approach in hardware that utilized parallel hardware to perform matrix reductions, leading to an overall run-time complexity of $\mathcal{O}(min(M,N))$ [8]. Following the development by Shui et al., a new approach to deadlock detection in single-unit resource systems was developed by Xiao and Lee in 2007, known as HDDU [9]. This algorithm had a deadlock detection run-time of $\mathcal{O}(1)$ and a detection preparation run-time of $\mathcal{O}(min(M,N))$. Later, Xiao and Lee developed a new approach to classifying resource events in single-unit resource systems. This development led to a new algorithm known as OSDDA [10]. By utilizing the new classification of resource events, deadlock preparation was able to be completed in $\mathcal{O}(1)$ time. As a result, OSDDA was able to achieve an overall run-time complexity of $\mathcal{O}(1)$ in hardware.

The single-unit resource hardware deadlock detection approaches mentioned above ultimately led to the development of deadlock detection algorithms in hardware for multi-unit resource systems. In 2007, Xiao and Lee developed the first multi-unit hardware approach to deadlock detection, known as MDDU [11]. The deadlock preparation run-time of MDDU was $\mathcal{O}(min(M,N)$ and deadlock detection run-time was $\mathcal{O}(1)$ in hardware. This algorithm was the first of its kind to have a $\mathcal{O}(1)$ deadlock detection run-time. Following this development, Xiao and Lee then devised a new approach to classifying resource events in a multi-unit resource system which led to the development of the $log_2(min(M,N))$ deadlock detection algorithm known as LMDDA [12].

Now that prior work regarding deadlock detection has been discussed, we'll move on to discussing prior work in deadlock avoidance. In 1965, Dijkstra introduced the Bankers Algorithm (BA) for systems with a single resource with multiple units [1]. Since most systems have more than a single resource available, BA laid the framework for deadlock avoidance algorithms. Habermann, in 1969, improved BA to support multiple resources with multiple instances of each resource [2]. Habermann's approach was software-based and had a run-time of $\mathcal{O}(N \times M^2)$, where M and N are the process and resource amounts, respectively. This approach was improved upon in 1972 by Holt. Holt's approach improved the algorithm's run-time, leading to a $\mathcal{O}(M \times N)$ run-time for BA [3].

Much like the progression towards hardware-based approaches to deadlock detection, a similar progression occurred regarding deadlock avoidance. In 2004, Lee designed a parallel hardware approach to BA which resulted in a run-time complexity of $\mathcal{O}(N \times min(M,N))$ [13]. The only caveat to this algorithm was that it was able to operate solely on single-instance resource systems. As a follow up to this work, Lee and Mooney then proceeded to develop a $\mathcal{O}(N)$ deadlock avoidance algorithm in hardware known as the Parallel Banker's Algorithm Unit (PBAU) [14]. PBAU was able to handle multi-resource systems containing multiple instances of each resource.

1.4 Motivation

One may inquire about the purpose of developing such algorithms for the GPU over the CPU when modern CPUs have parallel programming primitives (such as OpenMP) and feature vector architectures such as MMX and SSE. The answer is quite simple: our CPUs are meant for general purpose tasks. The goal of a deadlock detection/avoidance algorithm should be to act as an *interactive service* to the normal operation of the CPU. If such algorithms were to be designed and utilized on the CPU using its parallel programming capabilities, it would severely detract from the usability of the CPU. Not only would the CPU not be able to spawn the amount of threads that a GPU is capable of, thus not being able to fully exploit the parallel nature of the problem, but it also would take longer to execute and thus other critical tasks would not be able to take place concurrently.

For these reasons, this thesis proposes three algorithms: GPU-OSDDA, GPU-LMDDA, and GPU-PBA. GPU-OSDDA is the first single-unit deadlock detection algorithm implemented on the GPU platform. This algorithm stores its adjacency request and grant matrices, as well as other algorithm matrices and vectors, in integer length bit-vectors. This along with the bit-wise operators utilized to solve computations of deadlock detection has allowed this algorithm to operate on large systems with increasing numbers of processes and resources (in the thousands or tens of thousands). It has accomplished this while also being able to run extremely fast, thus yielding a single-unit deadlock detection algorithm applicable to real-world systems.

The second algorithm, GPU-LMDDA, is the also the first multi-unit deadlock detection algorithm implemented on the GPU platform. Much like GPU-OSDDA, GPU-LMDDA stores all but one of its algorithm's matrices in integer length bit-vectors. This also has allowed GPU-LMDDA to adopt bit-wise operators to perform calculations involved in deadlock detection for multi-unit systems. The result of this work yielded a multi-unit deadlock detection algorithm that was able to achieve substantial speedups over the CPU, as well as handle up to 1024 processes or resources concurrently.

The last of the proposed algorithms, GPU-PBA, is the first GPU-based deadlock avoidance algorithm. By utilizing the GPU's parallel architecture, many computations involved in deadlock avoidance were able to be parallelized. As a result, GPU-PBA was able to achieve speedups over a CPU-based implementation of the Banker's Algorithm.

For all of these algorithms, the CPU's only responsibility is to pass resource event information to the GPU for computation. In this way, GPU-OSDDA, GPU-LMDDA, and GPU-PBA serve as interactive services to the CPU. When the words *interactive service* are mentioned, it refers to the limited interaction that the CPU is required to have with our algorithms. All of the aforementioned algorithms are meant to run in the background, receiving resource event information from the operating system and then provide notification to the CPU regarding the status of the resource event (deadlock occurred or request denied). In this way, our algorithms provide an unobtrusive notification to the CPU (or operating system) regarding the state of its resource events.

1.5 Thesis Organization

In Chapter 2, the disparity between serial and parallel computing is discussed. This is followed by a discussion of the CUDA hardware architecture and software model. Chapters 3, 4, 5 discuss GPU-OSDDA, GPU-LMDDA, and GPU-PBA, respectively, as well as their implementation details and results. Chapter 6 provides a summary of this thesis and its technical contributions.

2 PARALLEL COMPUTING AND THE CUDA FRAMEWORK

2.1 Serial vs. Parallel Computing and Flynn's Taxonomy

In the past, most software programs have been written for Central Processing Units (CPUs) that follow the Single-Instruction Single-Data (SISD) architecture. This implies that the software programs are written in a serial manner; executing one instruction after the next. This architecture, however, does not advocate task concurrency in terms of the task's execution. Until the advent of multi-core CPUs, SISD machines implemented context switching (or time slicing) and pipelining in order to simulate task concurrency. The arrival of multi-core CPUs, parallel instructions sets (SSE and MMX), and massively parallel architectures (GPU) have enabled increasing amounts of task execution concurrency. In 1996, Michael Flynn introduced a method to classify various computer architectures, known as 'Flynn's Taxonomy' [15]. Table 2.1 summarizes these classifications and a further description of each architecture is provided in the following sub-sections.

Table 2.1: Flynn's Taxonomy

Architecture	Description
SISD	single instruction, single data
SIMD	single instruction, multiple data
MISD	multiple instruction, single data
MIMD	multiple instruction, multiple data

2.1.1 Single-Instruction Single-Data (SISD)

The SISD architecture is the most well-known class of computing architecture today. Software written for these machines were meant to execute in a serial manner. Figure 2.1 depicts this execution model. SISD architectures are thought to have no layers of concurrency; however, concurrency may exist if the architecture implements pipelining. Pipelining allows the CPU to perform multiple phases of processing (of different instructions) simultaneously. According to [15], this does not achieve concurrency of execution, but does achieve concurrency of processing. As a result, modern CPUs adopted the pipelining to increase their concurrent processing capability.

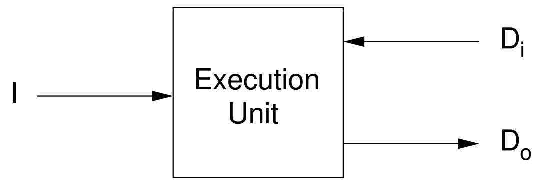


Figure 2.1: SISD Execution Model

The concurrency of execution is often referred to as Instruction-Level Parallelism (ILP). There are two common sub-architectures of SISD that aid in the concurrency of instruction execution: *superscalar* and *very long instruction word (VLIW)*. Both of these techniques schedule different operations to execute concurrently based on the analysis of the dependencies between the operations in the instruction stream [15]. The primary difference between the two is that the *superscalar* technique determines dependencies dynamically at run-time while the *VLIW* technique determines them statically at compile-time. Most modern CPUs are of the *superscalar* variety, as they are more easily adapted to current software.

2.1.2 Single-Instruction Multiple-Data (SIMD)

The SIMD architecture has become increasingly popular in modern day CPUs and mobile devices. This architecture typically has a single instruction operate on a multitude of data elements in parallel, hence single-instruction multiple-data. Figure 2.2 illustrates the execution model of the SIMD architecture.

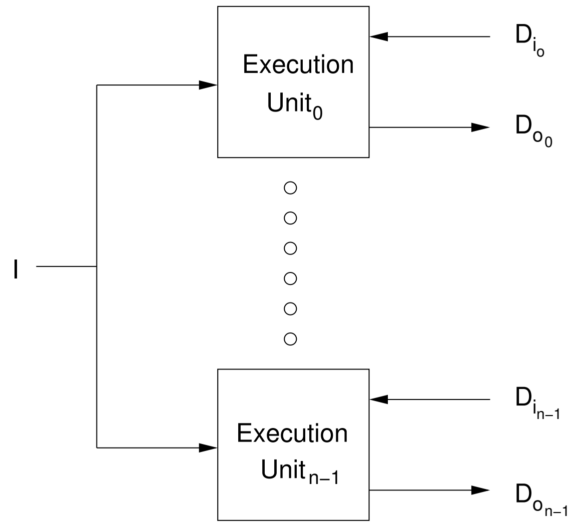


Figure 2.2: SIMD Execution Model

Flynn [15] further defines two SIMD sub-architectures: *array* and *vector* processors. *Array* processors feature hundreds or thousands of simple processors operating in parallel on sequential data elements [15]. Prior to the mid-2000s, there were no immediate problems requiring such an architecture. However, with the arrival of increasingly complex graph algorithms, medical applications, and graphics rendering techniques, the *array* processor design has found its way into the hands of everyday users in the form of GPU and General Purpose Graphics Processing Units (GPGPU). These architectures are also referred to as massively parallel processors (MPP). In contrast, the *vector* architecture typically has a single processing element that focuses its operation on smaller data sets. *Vector* processors rely on heavy pipelining and high clock rates to reduce overall latency of operations [15].

2.1.3 Multiple-Instruction Single-Data (MISD)

The MISD architecture is the most underutilized architecture, if thought as a standalone parallel architecture. The MISD architecture is composed of independently executing functional units operating on a single data stream, forwarding results from one functional unit to the next [15]. Figure 2.3 illustrates the execution model of the MISD architecture. This should sound much like the pipelining concept mentioned previously because on an abstract level it is pipelining. The difference between a pipelined architecture and the MISD architecture is that the pipelined architecture is a partition of assembly instructions into pipeline stages that pass results between successive stages. In contrast, the MISD architecture aims to pass full instruction results between execution units.

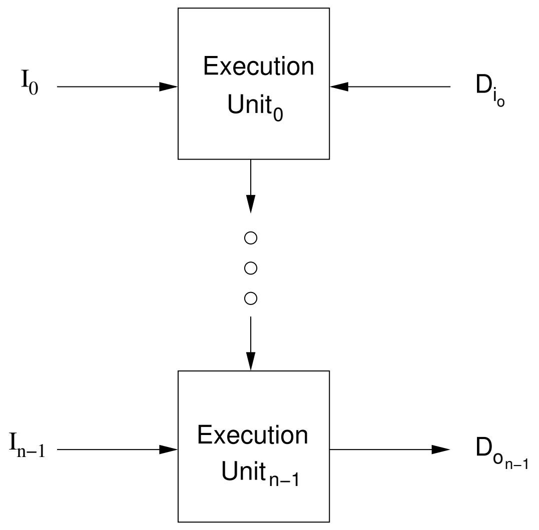


Figure 2.3: MISD Execution Model

According to [15], most programming constructs do not easily map programs into the MISD organization. For this reason, the abstract idea of MISD (pipelining) has been integrated into modern day computing systems, but the MISD architecture as a whole has been more or less ignored for real-world systems.

2.1.4 Multiple-Instruction Multiple-Data (MIMD)

The MIMD architecture is the most well-known parallel architecture available, seen in modern day multi-core CPUs. This architecture is typically comprised of independent, homogeneous processing units that communicate with each other through network interconnects or a shared memory construct (cache). Figure 2.4 illustrates the execution model of the MIMD architecture.

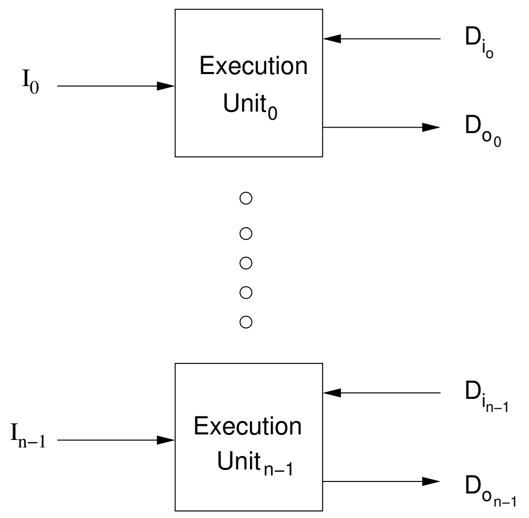


Figure 2.4: MIMD Execution Model

The MIMD architecture has its processing units handle independent instructions acting on independent data sets. The most significant issues with implementing such an architecture are mainlining memory and cache coherency [15]. Mainlining memory refers to the programmer-visible memory references seen while writing software, and how memory values may change when writing software across many processing units [15]. As mentioned previously, many of these homogeneous processing units are connected through a cache. When multiple processing units access the same memory locations, it is critical that all processing units see the same memory value, resulting in cache coherency. Modern CPUs solve these issues with a combination of software and hardware techniques for mainlining memory and exclusively hardware techniques for cache coherency [15].

2.2 CUDA Overview

The introduction of the Compute Unified Device Architecture (CUDA) by Nvidia has brought a whole new level of parallel programmability to the mainstream public. In the past, the GPU (a massively parallel processor) has been used solely for graphics rendering, an inherently parallel task. Since Nvidia has developed the CUDA hardware architecture and a scalable and powerful programming model, it has enabled normal users to offload heavy parallel computations to their CUDA-capable GPUs. By modeling their architecture off the familiar SIMD architecture, the GPU has the potential of operating on many thousands of data elements in parallel. This powerful architecture has unleashed never before seen capabilities in the consumer software development market. This has allowed developers to write applications that exploit the GPU's parallel execution capabilities.

For purposes of this thesis, only the two latest Nvidia architectures will be covered: *Fermi* and *Kepler*. These two architectures, while both providing significant compute capability, have a significant difference in their hardware design and subset of their software features. The following Sections 2.3 and 2.4 provide detailed information regarding the CUDA hardware architecture and software model in Nvidia's *Fermi* and *Kepler* series GPUs.

2.3 CUDA Hardware Architecture Details

The following Sections 2.3.1 and 2.3.2 describe in detail hardware features of the two latest Nvidia GPU architectures: *Fermi* and *Kepler*. Section 2.3.1 describes the *streaming multiprocessor* (SM) and *stream processor* (SP). The abstraction of threads, blocks, grids, and warps are also discussed as a subset of the SM and SP modules. Following that, in section 2.3.2, is a description of the CUDA memory hierarchy and the implications each memory type produces.

2.3.1 Streaming Multiprocessors

In any CUDA capable GPU, there exist a number of streaming multiprocessors (SM). The SM is the building block of the GPU. Within each SM, there are tens or hundreds of streaming processors (SP). The SPs are where the stream of execution takes place. Any time that an SP is given work, it is signified by a thread. These threads are scheduled in batches of 32 and are called *warps*. Each SM has its own set of warp-schedulers which allocate work to the SPs. Nvidia adopted a scheduling model that it dubs SPMD (single-program, multiple-data), which is based off of the SIMD architecture discussed in Section 2.1.2 [16]. Figures 2.5 and 2.6 depict the layout of the SM(X) in both the *Fermi* and *Kepler* architectures [17, 18]. Discussed next are the concepts of *threads*, *blocks*, *grids*, and *warps*, all of which provide a hardware abstraction of the CUDA hardware model.

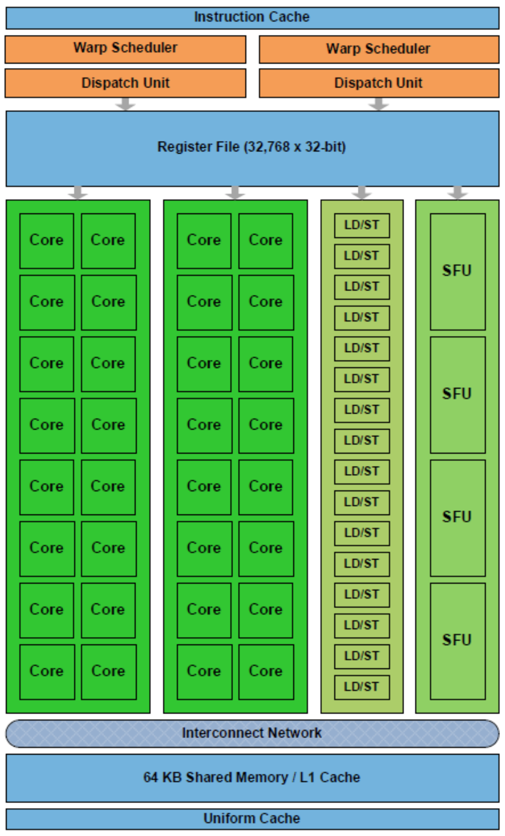


Figure 2.5: Fermi SM Block Diagram (Courtesy of Nvidia)

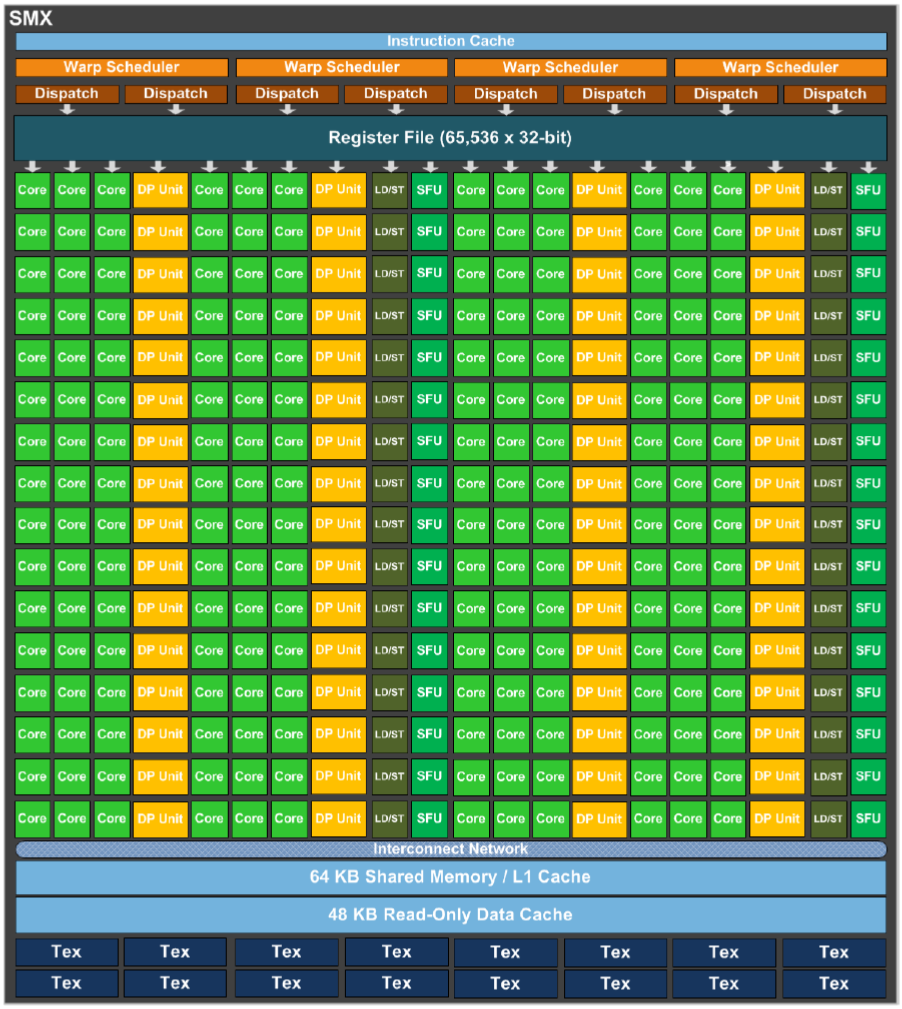

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 2.6: Kepler SMX Block Diagram (Courtesy of Nvidia)

Threads

Threads are the fundamental building block of a parallel program [16]. As most are familiar, uni-processors typically have a single *thread* of execution. This thread will compute a single result for a single data point per instruction. GPUs on the other hand are comprised of tens, hundreds, or thousands of SPs (synonymous with uni-processor in terms of instruction execution) that allow them to operate on tens, hundreds, or thousands of data points per instruction. The *thread* runs in a single

SP of a single SM. The thread scheduling units (*warps*) are distributed to the SPs of an SM by the warp schedulers available in the SM. Each thread is then allocated a partition of registers. These registers are on-chip, and have very high bandwidth and low latency. Section 2.3.2 further covers the registers available to each thread.

Blocks

A *block* is an abstraction for a set of threads on the GPU. While each thread's scope is a single SP, a block's scope is a single SM, i.e. many threads run on an SM. There are limitations to the number of threads one may schedule per block and also per SM, and this is determined by the target hardware. In our case, our thread limit per block on both the *Fermi* and *Kepler* architecture is 1024 threads. One of the differences between the *Fermi* and *Kepler* architectures is that while *Fermi* may schedule only 1536 threads per SM, *Kepler* may schedule 2048 threads per SM. This difference implies that *Kepler* may have two concurrent blocks running of size 1024 threads while *Fermi* may only run a single block. By launching multiple blocks, we are essentially launching (# of blocks $\times$ # of threads) streams of execution. This allows us to compute (# of blocks $\times$ # of threads) data elements in parallel per SM(X).

Grids

A *grid* in CUDA is an abstraction of multiple blocks on the GPU. Similar to how the *block* was a level higher in terms of scope than a *thread*, a *grid* has a higher scope than a *block*. The *grid* in CUDA hardware has scope of the entire GPU, or in other words, a *grid* may be decomposed across many SMs and SPs. Since the GPU is limited in the number of threads per block and blocks per SM it may launch, if a problem is in need of additional parallelism (or has more data points than currently available threads), we are able to specify more dimensions to the grid (x,y, and z). This enables more blocks, and thus, more threads to be scheduled per GPU. The method of scheduling multiple dimensions to a grid also applies to threads, which may aid in problem decomposition. Note that while we may schedule many more threads by adding additional dimensions to a *grid*, this does not necessarily imply that more threads are running concurrently.

Warps

Cook [16] states that the warp is the basic unit of execution on the GPU. Understanding how the GPU handles this thread grouping is important in order to properly utilize GPU resources. Since the GPU is essentially a collection of SIMD vector processors and each *warp* is executed together, ideally, a single fetch from memory for the current instruction is needed [16]. Upon fetching the instruction, the GPU is able to broadcast that instruction to each SP (thread) in the *warp*.

There exists one common programming paradigm that can significantly lower performance on the GPU, that is branching. The conditional branch in Algorithm 1 is a common practice in serial CPU programming. On the GPU, however, it becomes very costly.

Algorithm 1 Warp Branching

```
1: if test_condition_if then
2:     execute_if();
3: else
4:     execute_else();
5: end if
```

If the threads within a *warp* take separate execution paths for the conditional, the separate conditional (*if* and *else*) pieces will not be executed in parallel. Therefore, some of the threads within the warp may execute the *if* block and the other threads will become inactive. This inactive state effectively cuts utilization of the GPU hardware by half. When the initial threads complete the *if* block, the remaining threads will execute the *else* block, while the initial threads become inactive. The typical rule of thumb is that for every level of divergence in a conditional block (say N), our device utilization drops by a factor of N. So for a conditional code-block with 3 conditional statements, our utilization would be $\frac{1}{3}$, or 33%.

2.3.2 CUDA Memories

The crux to writing efficient parallel programs utilizing the GPU is understanding the GPU's memory hierarchy. Nvidia's CUDA architecture supplies the developer with several memory types, each type serving its own purpose with different on/off chip placements and latency requirements. Table 2.2 and Table 2.3 provide a summary of all on-board memory types available in the CUDA hardware architecture and their access times [19].

Table 2.2: CUDA Device Memory Features

Memory	**Location On/Off Chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached only on devices of compute capability 2.x					

Table 2.3: CUDA Memory Access Times

Storage Type	**Registers**	**Shared Memory**	**Texture Memory**	**Constant Memory**	**Global Memory**
Bandwidth	~8 TB/s	~1.5 TB/s	~200 MB/s	~200 MB/s	~200 MB/s
Latency	1 cycle	1 to 32 cycles	~400 to 600	~400 to 600	~400 to 600

To facilitate understanding of the memory hierarchy, Figure 2.7 depicts the memory spaces on the GPU device [19]. The rest of this section summarizes typical memory usage scenarios and additional information on registers available in the CUDA architecture.

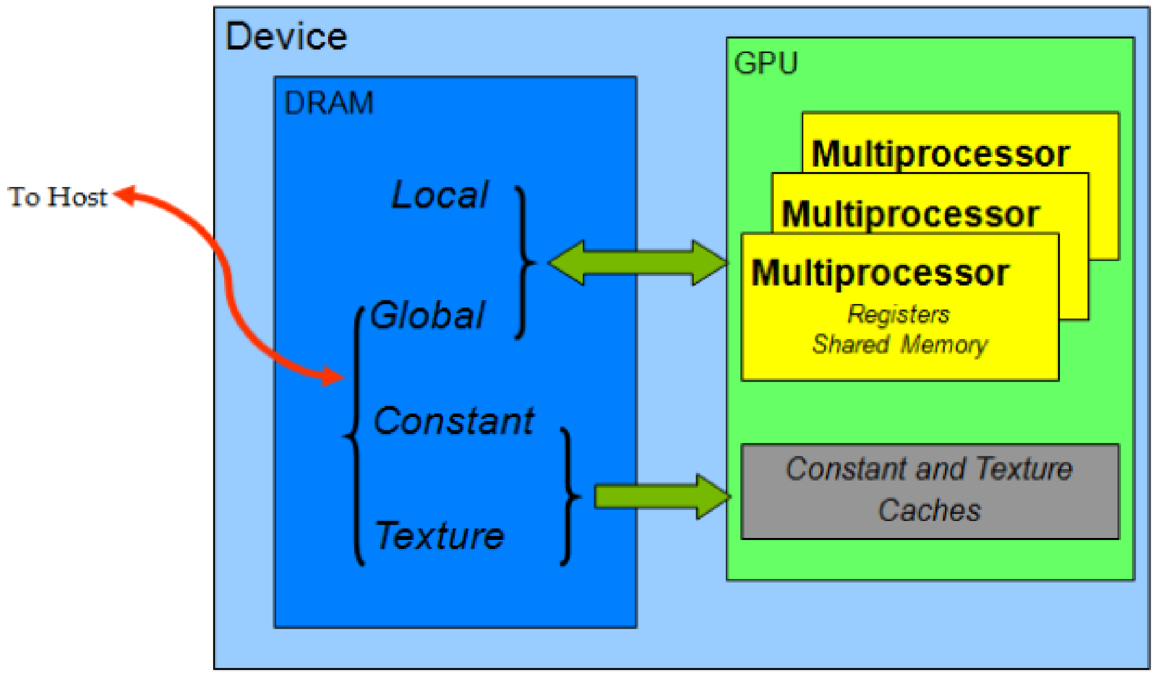


Figure 2.7: CUDA Memory Spaces (Courtesy of Nvidia)

Global Memory Coalescing and Caching

According to Nvidia [19], global memory coalescing is the most important performance consideration to take into account when writing parallel code for CUDA-capable GPUs. If done correctly, global memory loads and stores initiated by a warp may be handled in a single memory transaction when certain requirements are met. These requirements can be summarized as follows:

> The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads of a warp [19].

By default, the GPU's L1 cache is enabled and all transactions are cached through it. This cache is comprised of 128-byte cache lines. Next, three examples are given to illustrate a warp that performs coalesced accesses in a single 128-byte L1-cache line, unaligned sequential addresses fitting into two 128-byte L1-cache lines, and misaligned sequential addresses that fall within five 32-byte L2-cache lines.

Figure 2.8 shows the simplest case of memory coalescing in CUDA. Every k-th thread in a warp will access the k-th word in an L1-cache line. If in this case each thread addresses a four byte value, then an entire warp is satisfied by a single 128-byte transaction from the L1-cache (hashed area in figure is the cache line).

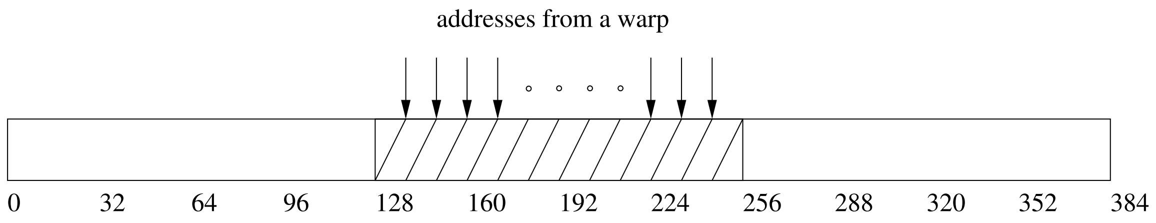


Figure 2.8: Coalesced global memory accesses fitting into a single 128-byte L1-Cache line

In the second example, seen in Figure 2.9, the threads within a warp access sequential addresses but are not aligned on the 128-byte cache line boundary. This forces the GPU to fetch two 128-byte cache lines from the L1-cache.

The last example in Figure 2.10 is considered a non-caching transaction, in that it bypasses the L1-cache and uses the L2-cache [19]. The L2-cache utilizes 32-byte cache lines. In the event that misaligned (but sequential) addresses are requested, it forces the GPU to fetch five 32-byte L2-cache lines to satisfy the 128-byte request.

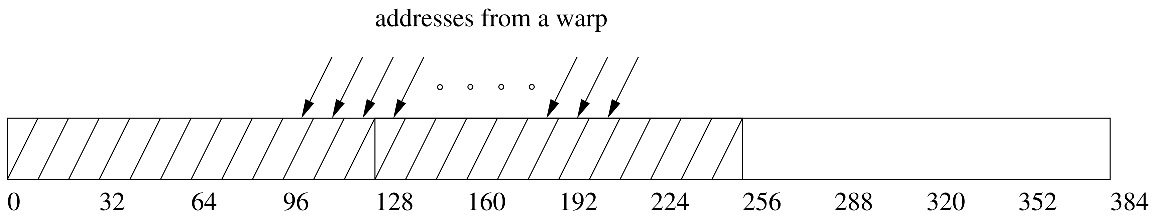


Figure 2.9: Unaligned sequential addressing fitting into two 128-byte L1-Cache lines

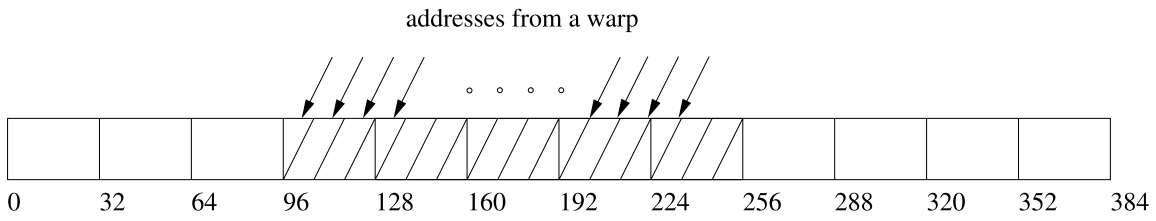


Figure 2.10: Misaligned sequential addressing fitting into five 32-byte L2-Cache lines

Shared Memory and Bank Conflicts

As denoted in Table 2.2, the GPU's shared memory resides on-chip. Since shared memory is on-chip, it has a much higher bandwidth and lower latency than all other memories on the GPU with the exception of registers. However, the potential speedup granted by using shared memory may suffer in the event that shared memory bank conflicts occur. The structure of shared memory and bank conflicts are discussed next.

Shared memory is able to achieve high bandwidth for concurrent memory accesses because it is split into equally sized memory modules (or *banks*) that can be accessed simultaneously [19]. In the event that the GPU loads or stores n addresses that span n shared memory banks, the loads and stores of all n addresses may be handled simultaneously [19]. The simultaneous handling of shared memory accesses yields an effective bandwidth n times higher than bandwidth of a single shared memory bank. If there are multiple memory requests (for different addresses) that access the same shared memory bank, these accesses are serialized. In the event of a bank conflict consisting of m requests, the GPU hardware splits the memory requests into

m separate conflict-free requests. This action would effectively decrease the shared memory bandwidth by a factor of m. The exception to this rule is when multiple threads within a *warp* access the same memory location. If this occurs, the memory location is broadcasted to all requesting threads.

The two architectures we focus on in this thesis (*Fermi* and *Kepler*) are of compute capability 2.x and 3.x, respectively. In the 2.x compute GPUs, each shared memory bank has a bandwidth of 32 bits per two clock cycles and successive 32-bit words are assigned to successive memory banks [19]. The 3.x compute capability GPUs have two separate banking modes: successive 32-bit words or successive 64-bit words are assigned to successive banks. The 3.x compute capable card could have a maximum bandwidth per bank equal to 64 bits per clock cycle [19].

Register Usage

As denoted by Table 2.3, registers offer the highest bandwidth and lowest latency of all the memory types available on the CUDA platform. Typically, registers do not consume additional clock cycles per instruction but delays may occur due to read-after-write (RAW) dependencies and register memory bank conflicts [19]. The latency of such a dependency is approximately 24 cycles, but this latency may be hidden by running simultaneous threads. For example, devices of compute capability 2.0, which have 32 CUDA cores per SM(X), may require up to 768 threads, or 24 warps, to completely hide this latency (32 CUDA cores per SM $\times$ 24 cycles of latency = 768 active threads to cover latency) [19].

In terms of optimally utilizing registers, NVCC (CUDA compiler) and the hardware thread scheduler will attempt to schedule instructions in such a way to avoid register memory bank conflicts [19]. There are a large number of registers available per SM(X), but the available registers are partitioned among all threads scheduled to that SM(X). For fine tuned control over register usage, Nvidia provides a compiler switch, *-maxrregcount=N*, that places an upper bound on the number of registers a thread may have allocated to it (in this case, N) [19]. Generally, the best results are achieved by launching thread blocks that have a thread count equal to a multiple of 64.

2.4 CUDA Programming Model Details

The CUDA programming model (CUDA C) is split into two separate APIs: driver and runtime. The driver API is not within the scope of this thesis, but it should be known that it has the same capabilities as the runtime API with some advanced low level features. The runtime API will be discussed here, as it is the most commonly used and provides a higher level of abstraction to the GPU hardware.

2.4.1 Core Software Instrinsics

CUDA C is simply an extension of the familiar C programming language. Nvidia [20] has allowed the programmer to define functions, known as *kernels*, that are executed N times in parallel by N different threads on the GPU. In contrast, the CPU typically launches a single thread to execute a function. The similarities of the GPU's execution model with that of a SIMD architecture (seen in Section 2.1.2) are realized by how the GPU kernels execute instructions. The GPU kernel launches a series of serialized instructions that operate on a vector of (or many) data elements, just like the SIMD architecture.

The CUDA kernel (or function) is defined using the *__global__* declaration specifier, followed by *void* as the return type since CUDA kernels do not return values. In order to specify the number of blocks (i.e., the grid size) and the number of threads per block (i.e., the block size) for a kernel launch, the CUDA framework provides the triple chevron execution configuration syntax, $\lll ... \ggg$. In its most basic form, the execution configuration syntax is populated with two arguments: 1.) the number of blocks and 2.) the number of threads per block. As mentioned in Section 2.3.1, multiple dimensions of both blocks and threads may be invoked. The arguments to the execution configuration syntax are of type *dim3*, which is a 3-component vector [20]. When the *dim3* variable is used within a kernel it allows for thread identification by a one-dimensional, two-dimensional, or three-dimensional thread index known as *threadIdx*. Utilizing the multi-dimensional thread and block intrinsics allows for easier mapping of threads to a problem space (i.e., vector, matrix, volume). The following summarizes multi-dimensional thread usage within a single block.

> The index of a thread and its thread ID relate to each other in a straightforward way. For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x,D_y), the thread ID of a thread of index (x,y) is $(x+yD_x)$; for a three-dimensional block of size (D_x,D_y,D_z), the thread ID of a thread index (x,y,z) is $(x+yD_x+zD_xD_y)$ [20].

Algorithm 2 illustrates the usage of a two-dimensional thread block to perform an addition operation between two matrices A and B. In lines 13 and 14, the dim3 component vector is used to establish the number of blocks launched per grid and the number of threads launched per dimension. This is followed by the kernel invocation in line 15. Starting on line 4 is the kernel that will perform the matrix addition. Lines 4 and 5 generate the indices for the x and y dimensions of matrices A and B. Notice that the vector directions (x,y) are referenced using the threadIdx variable appended with **.x** or **.y**, depending on the dimension to be used. Line 6 performs the matrix addition between A and B, storing the result in the matrix C.
Declaring registers to hold the values of the threadIdx variables are not necessary. However, doing so greatly increases code readability.

Note that there is a limit to the number of threads that may be invoked per block. In both the *Fermi* and *Kepler* architectures the thread limit per block is 1024 threads. If additional threads are needed, which is typical, more blocks should be scheduled to handle the additional parallelism. Similar to the prior algorithm, Algorithm 3 utilizes multiple blocks to handle a larger matrix size. Assume for the computation that M and N are of size 256. In lines 14-15, the *dim3* vector is initialized with 16 in the x and

Algorithm 2 Multi-Dimensional Threads - Kernel Invocation

```
1:  // Kernel Definition
2:  __global__ void Add2DVector(int A[M][N], int B[M][N], int C[M][N])
3:  {
4:      int x = threadIdx.x;
5:      int y = threadIdx.y;
6:      C[x][y] = A[x][y] + B[x][y];
7:  }
8:
9:  int main()
10: {
11:     ...
12:     // Kernel launched with a single block of M * N * 1 threads
13:     dim3 blocksPerGrid(1,1,1);
14:     dim3 threadsPerBlock(M,N,1);
15:     Add2DVector≪blocksPerGrid,threadsPerBlock≫(A, B, C);
16:     ...
17: }
```

y directions. The number of blocks required in both directions depends on the quotient of M or N and the number of threads launched in the associated directions. Line 16 performs the kernel invocation using the blocksPerGrid and threadPerBlock *dim3* variables that were created. Inside the kernel, lines 4 and 5 build the x and y indices into the matrices. Since multiple blocks (with multiple dimensions) were launched, we utilize the *blockIdx* variable with **.x** or **.y** appended to obtain the appropriate block ID. We then multiply the block ID by the number of threads within each block, denoted by *blockDim*. The thread ID is then added to yield the global index into the matrices. Line 6 performs a check on the indices x and y to verify computations stay in a valid memory range, which is followed by the final matrix addition in line 7.

When scheduling multiple thread blocks in a kernel invocation, all thread blocks must be able to execute independently [20]. Threads within a single block may communicate with each other via *shared memory* as discussed in Section 2.3.2. If the threads within the block need to synchronize at some point (to avoid data hazards), Nvidia provides the *__syncthreads()* intrinsic function. The *__syncthreads()* function creates a barrier where all threads within a block must wait to perform further action

Algorithm 3 Multi-Dimensional Blocks and Threads - Kernel Invocation

```
// Kernel Definition
__global__ void Add2DVector(int A[M][N], int B[M][N], int C[M][N])
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    if($x < M$ and $y < N$)
        C[x][y] = A[x][y] + B[x][y];
}

int main()
{
    ...
    // Kernel launched with 16x16 (256) thread blocks
    dim3 threadsPerBlock(16,16,1);
    dim3 blocksPerGrid(M/threadsPerBlock.x,N/threadsPerBlock.y,1);
    Add2DVector<<<blocksPerGrid,threadsPerBlock>>>(A, B, C);
    ...
}
```

[20]. After all threads have synchronized on this function, the threads within that block may continue their execution.

There are no synchronization primitives between blocks while a kernel is executing. The only way that threads across multiple blocks may communicate is via *global memory*. If multiple invocations of a CUDA kernel are required, synchronization is handled by the CPU by using the *cudaDeviceSynchronize()* intrinsic function following a kernel launch. Since all kernel calls are asynchronous, this function forces the CPU to halt its execution until the invoked kernel has completed.

When deciding on the optimal number of threads and blocks to schedule for a problem, some experimentation may be needed. However, Nvidia has provided the following common practices:

- Threads per block should be a multiple of the warp size to avoid wasting computation on under-populated warps and to facilitate coalescing [19].

- A minimum number of 64 threads per block should be used, and only if there are multiple concurrent blocks per SM(X) [19].
- Between 128 and 256 threads per block is a better choice and a good initial range for experimentation with different block sizes [19].
- Use several (3 or 4) smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call *__syncthreads()* [19].

2.4.2 Occupancy

Instructions executed by a thread are done so in sequence. By having additional warps execute while others are paused or stalled allows the GPU to hide memory latency and keep the hardware active. The metric that relates the number of active warps on an SM(X) to the maximum number of active warps is known as *occupancy* [19]. Maintaining a high level of occupancy is never bad practice because it does hide memory latency. However, there is a diminishing rate of return for having additional occupancy above a certain point.

For purposes of calculating occupancy, the number of registers claimed per thread is a primary factor [19]. On the *Kepler* architecture, for example, there are 65,536 registers available per SM(X) and each SM(X) may have up to 2048 resident threads (64 warps $\times$ 32 threads per warp). This implies that to achieve 100% occupancy, each thread may claim at most 32 registers. Determining the occupancy of a kernel can be complicated across different GPU architectures. The number of available registers per SM(X), simultaneous resident threads on an SM(X), the number of blocks running per SM(X), and the register allocation granularity are all factors that need to be considered when calculating occupancy [19]. Since many of these factors change across architectures, the relationship between register usage and occupancy is difficult to determine. There has been research performed that determined that higher occupancy does not always mean better performance [21].

As a result, testing and experimentation are the only true way to maximize performance with regards to occupancy.

Nvidia [19] does provide two tools to help developers determine the occupancy of kernels, they are the CUDA Occupancy Calculator and the Nvidia Visual Profiler's Achieved Occupancy metric. By evaluating kernels with one of these two tools, developers are able to make more efficient decisions regarding the number of blocks and threads to launch.

2.5 Chapter Summary

This chapter has discussed the differences between serial and parallel computing, followed by the well-known computer architectures described by Flynn's Taxonomy. By understanding Flynn's classification of the SIMD architecture, the current GPU programming platform is better understood.

Next, the hardware architecture details of the Fermi and Kepler GPU architectures were covered in detail. How the GPU schedules its threads, interacts with memory, and the implications of working with several memory types were covered in order to facilitate explanation of the algorithms within this thesis. Finally, the CUDA programming model was discussed to explain how the CUDA software framework provides abstractions of the GPU hardware. After researching both the hardware architecture and software model details, the most performance critical techniques and optimizations have been applied to the algorithms in this thesis.

3 GPU-OSDDA: A GPU-BASED DEADLOCK DETECTION ALGORITHM FOR SINGLE-UNIT RESOURCE SYSTEMS

3.1 Introduction

This chapter discusses GPU-OSDDA, a GPU-based approach to deadlock detection in single-unit resource systems. The following section states the system assumptions and provides background information regarding the core methodology of GPU-OSDDA. GPU-OSDDA adopts the resource event classification scheme found in the hardware-based deadlock detection algorithm known as $\mathcal{O}(1)$ *Single-Unit Deadlock Detection Algorithm* (OSDDA) [10]. Section 3.2.2 includes the underlying theory of OSDDA and discusses how it classifies and handles resource events, as well as its $\mathcal{O}(1)$ overall run-time capability in hardware. This is followed by our algorithm design and its implementation in Section 3.3. Following the design discussion is the Experimentation and Results Section (Section 3.4) which details the run-times and speedups achieved by GPU-OSDDA.

3.2 Background

3.2.1 Assumptions and Terms

We here define a single-unit resource system as an $m \times n$ system, where m and n are the number of processes and resources, respectively. The proposed algorithm adheres to the following assumptions:

1. Each resource contains a single unit (see Definition 1.2.13). Thus, a cycle in the RAG is a necessary and sufficient condition for deadlock [10].

2. One process requests one resource at a time (see Definition 1.2.17). Thus, a process is blocked as soon as it requests an unavailable resource [3].

3. A resource is granted to a process immediately if the resource is available. As a result, the entire system is always in an *expedient* state (see Definition 1.2.16) [3].

4. Resource events are managed centrally (e.g., by the OS).

To aid computations of deadlock detection in GPU-OSDDA, we define the *sink process node* (see Definition 1.2.35) and the *active process* (see Definition 1.2.15).

3.2.2 Underlying Theory of OSDDA

OSDDA is truly unique because its overall algorithm run-time is $\mathcal{O}(1)$ in hardware. It is able to achieve this by performing parallel computations on a RAG based on the classification of resource events in the system. The three types of resource events in OSDDA are: *granted resource request*, *blocked resource request*, and *resource release* [10]. The resource events and deadlock detection capability of OSDDA are briefly discussed in Sections 3.2.2 and 3.2.2.

Resource Events

Let us first discuss the *resource request granted* event type. For this event to occur, when a process p_i requests a resource q_j, q_j has to be available (not granted to another process) and p_i needs to be an active process. This means that resource q_j must not have any incoming or outgoing edge and process p_i may have incoming edges but no outgoing edge (since the system is in an expedient state). If these criteria are satisfied, resource q_j may be granted to process p_i. This event causes q_j to change its reachable sink node. The *resource request granted* scenario is illustrated in Figure 3.1.

Next we will discuss the *resource request blocked* event. When a process p_i requests a resource q_j and there are no available unit of q_j, the process p_i is blocked. Prior to

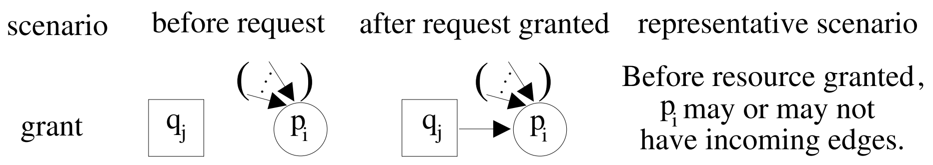


Figure 3.1: Scenarios of *resource request granted* events (Courtesy of Xiao and Lee [10]).

the request, process p_i has to be an *active process*, and thus, p_i has no outgoing edge when the request is made. By definition of an *active process*, p_i could have already been granted resources and as a result have incoming edges. Furthermore, resource q_j has an outgoing edge (as it is not available) and may have incoming edges (pending requests). As a result, two scenarios of the *resource request blocked* event exist with an illustration provided in Figure 3.2:

1. Block (i) - Before request blocked, p_i has no incoming edges; q_j may or may not have incoming edges [10].

2. Block (ii) - Before request blocked, p_i has incoming edges; q_j may or may not have incoming edges [10].

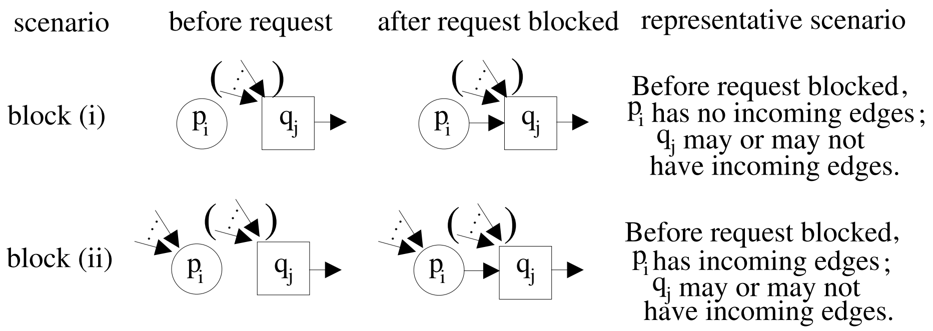


Figure 3.2: Scenarios of *resource request blocked* events (Courtesy of Xiao and Lee [10]).

Finally, we'll discuss the *resource release* event. For process p_i to release its resource, it must be an *active process* by having no outgoing edge. While servicing the *resource release* event, the algorithm must determine if resource q_j has any pending resource requests (incoming edges). If q_j has a pending request (incoming edge) from a process p_t, q_j is granted to p_t after the release (due to the system being in an expedient state). Depending on if resource q_j has pending requests, two separate *resource release* scenarios exist with an illustration provided in Figure 3.3.

1. Release (i) - Before resource released, p_i may have one or more incoming edges; q_j has no incoming edges [10].

2. Release (ii) - Before resource released, p_i may have one or more incoming edges; q_j has one or more incoming edges; p_t may or may not have incoming edges. After release, q_j is assigned to p_t [10].

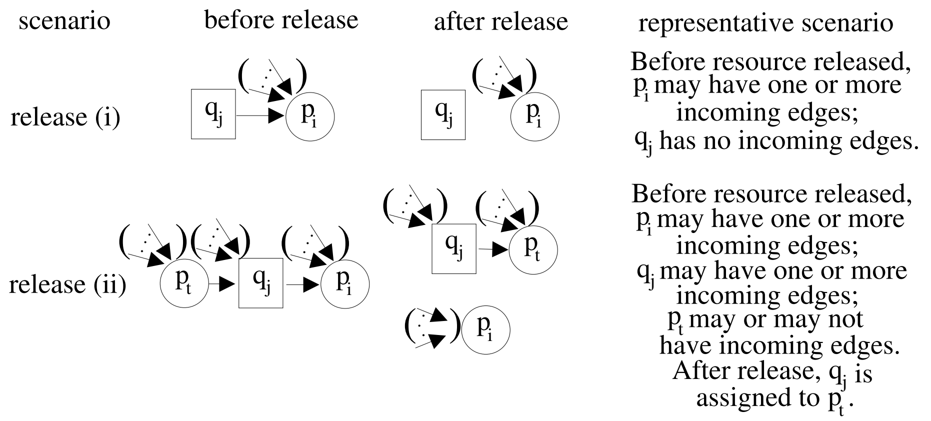

scenario
before release
after release
representative scenario
release (i)
q_j
p_i
Before resource released, p_i may have one or more incoming edges; q_j has no incoming edges.
release (ii)
p_t
Before resource released, p_i may have one or more incoming edges; q_j may have one or more incoming edges; p_t may or may not have incoming edges. After release, q_j is assigned to p_t.

Figure 3.3: Scenarios of *resource release* events (Courtesy of Xiao and Lee [10]).

O(1) Deadlock Detection

It is known that as long as the *sink process node* for every resource in the system can be identified, then deadlock can be detected in $\mathcal{O}(1)$ time in OSDDA [5] [9]. We know the reachable *sink process node* of a resource q_j is process p_i if and only if p_i is a *sink process node* and a path from resource q_j to process p_i exists. A cycle occurs in the system when the *sink process node* (say p_i) of a resource (say q_j) requests the resource. By the system assumptions of OSDDA, a cycle in the RAG is a necessary and sufficient condition for deadlock, and thus, under this scenario, a deadlock exists.

Prior work in [9] adopted the same deadlock detection approach as seen in [5], but this approach required reachable sink node information to be re-computed after every resource event. The re-computation of the *sink process nodes* in the system has a run-time complexity of $\mathcal{O}(m+n)$ in [5], $\mathcal{O}(\min(m,n))$ in [9], and $\mathcal{O}(1)$ in [10], where m and n are the process and resource counts, respectively. To achieve $\mathcal{O}(1)$ run-time of deadlock detection, OSDDA maintains the sink information for all resources in the system for use in upcoming invocations of deadlock detection. The sink information is stored in a matrix known as Sink.

Furthermore, for a release (ii) event, the OSDDA algorithm needs to identify resources on the sub-tree of the released resource (q_j) as well as those on the sub-tree of the process acquiring q_j [9]. For this, OSDDA utilizes the ReachableResource or RR and the ReachableProcess or RP matrices to maintain information on what resources and processes are reachable from every resource, respectively [9]. The matrices are defined as follows:

$$Sink[j][i]_{nxm} = \begin{cases} 1 & \text{if } p_i \text{ is } q_j\text{'s reachable sink node,} \\ 0 & \text{otherwise.} \end{cases}$$

$$RR[j][k]_{nxn} = \begin{cases} 1 & \text{if a path exists from resource } q_j \text{ to } q_k \\ & \text{or } k = j, \\ 0 & \text{otherwise.} \end{cases}$$

$$RP[j][i]_{nxm} = \begin{cases} 1 & \text{if a path exists from resource } q_j \text{ to } p_i, \\ 0 & \text{otherwise.} \end{cases}$$

3.3 GPU-OSDDA Design

3.3.1 Introduction

During the development process of our preliminary version of GPU-OSDDA, we implemented two versions, one with characters and the other with integers to represent our matrices. For both the character and integer based approaches, time was spent optimizing and tweaking code to ensure that occupancy was high, coalesced memory accesses were occurring, and threads were kept busy. This enabled us to maintain a high Instructions Per Cycle (IPC) ratio and maximize the memory bandwidth for our problem. We utilized the Nvidia Compute Visual Profiler to gauge our results at each step and came to a point where we were satisfied with the optimizations. However, with all optimizations complete, we were only able to achieve 3-24X speedup over our CPU implementation, dubbed CPU-OSDDA. These speedups, while an improvement, did not grant us the kinds of speedups we were looking for.

We then decided that a radically different approach was necessary in order to maximize our speedups and yield an algorithm that would be applicable to real world systems. Thus, we decided to implement our entire algorithm with integer length bit-vectors. This approach would reduce our memory footprint by a factor of 32, thus allowing for an increasing amount of processes and resources the algorithm could handle, as well as simplify and accelerate the bit-wise computations of our algorithm. Advantages of this approach are discussed throughout the remaining subsections as well as how GPU-OSDDA handles each resource event type.

3.3.2 Bit-Vector Design

Since GPU-OSDDA is based off of a single-unit system, all values that indicate the state of a process or resource in the system may be represented as binary values (0,1). In this case, instead of using an 8 or 32-bit variable to hold a 1-bit value, we bit-pack 32 processes or resources into a single 32-bit unsigned integer. Figure 3.4 shows how we would create a 128×128 adjacency matrix using 32-bit unsigned integers, where each box in the image represents a 32-bit unsigned integer. Similarly, we could create an adjacency matrix where the rows/columns are reversed.

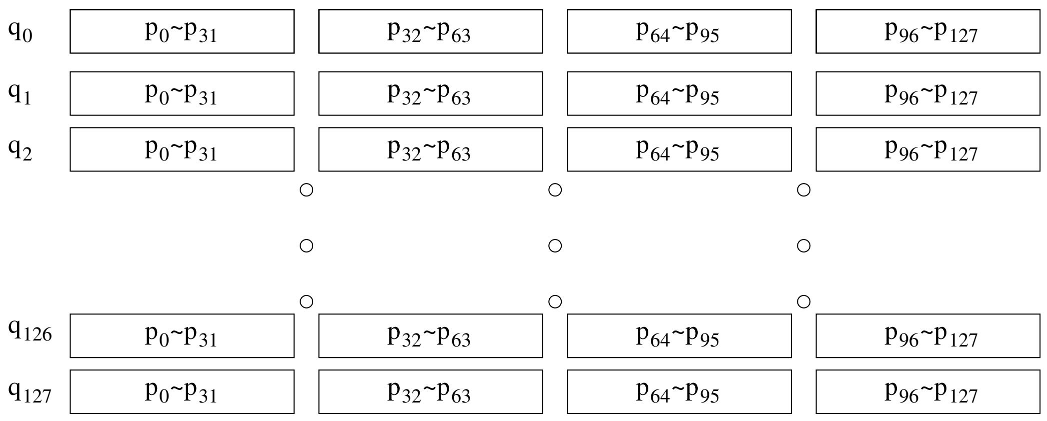


Figure 3.4: A 128x128 Bit-Vector Adjacency Matrix

Additionally, Table 3.1 describes variables used throughout the remaining algorithm descriptions using Figure 3.4.

Table 3.1: Common variables used throughout GPU-OSDDA.

Variable	Description	Values in Figure 3.4	Comment
INTS_PER_ROW (IPR)	The number of integers in a bit-packed row	4	Used to calculate row index
INTBITS	The number of bits per unsigned integer	32	Used to determine integer and bit to alter
LIPR	Equivalent to $log_2(INTS\_PER\_ROW)$	2	Used in multiply bit-shift calculations
LINTBITS	Equivalent to $log_2(INTBITS)$	5	Used in divide bit-shift calculations

Now that all algorithm critical information has been discussed, we present the overall kernel structure with the pseudo-code in Algorithm 4. The pseudo-code presents the kernels called upon each resource event type. In lines 4-5, GPU-OSDDA handles the *resource request granted* event.

The *Request_Granted* kernel launches a single block containing a single thread to perform the updates discussed in Section 3.3.3.

Algorithm 4 Overall Kernel Structure

```
 1: // Refer to Table 3.1 for variable definitions
 2: // Where TILE_DIM equals M÷INTBITS or N÷INTBITS (32 bits per unsigned int)
 3:
 4: if Resource Request Granted then
 5:     Request_Granted⋘1,1⋙(event, AG, Sink, RP)
 6: else if Resource Request Blocked then
 7:     DeadlockCheck_Init⋘1,1⋙(event, Sink, AR, deadlock)
 8:     if deadlock == false then
 9:         BitMatrix_Transpose⋘TILE_DIM,TILE_DIM⋙(AG_tile, AG)
10:         tileTranspose⋘TILE_DIM,TILE_DIM⋙(AG_trans, AG_tile)
11:         Row_Reduction⋘1,IPR/2⋙(AG_trans[event→p*IPR], phold)
12:         if phold == false then
13:             Request_Blocked⋘N,IPR⋙(event, Sink_temp, Sink, RR, RP)
14:         end if
15:     else
16:         Handle Deadlock
17:     end if
18: else if Resource Released then
19:     Release_Resource⋘1,1⋙(event, AG)
20:     BitMatrix_Transpose⋘TILE_DIM,TILE_DIM⋙(AR_tile, AR)
21:     tileTranspose⋘TILE_DIM,TILE_DIM⋙(AR_trans, AR_tile)
22:     Row_Reduction⋘1,IPR/2⋙(AR_trans[event→q*IPR], pwait)
23:     if pwait == 0 then
24:         Update_Sink_RP⋘1,1⋙(event, Sink, RP)
25:     else
26:         Update_AG_AR⋘1,1⋙(event, AG, AR)
27:         Release_Update_Reachability⋘N,IPR⋙(event, Sink, RP, RR)
28:     end if
29: else
30:     Not a valid event
31: end if
```

Lines 6-16 in Algorithm 4 handle the *resource request blocked* event. The *DeadlockCheck_Init* kernel in line 7 launches a single block containing a single thread. It checks the system for deadlock utilizing the Sink[ ] matrix previously discussed and updates the *deadlock* flag accordingly. Line 8 checks the system's deadlock status. If deadlock exists, the algorithm notifies the CPU of the deadlock status. Otherwise, lines 9-10 performs a bit-wise matrix transpose on the AG[ ] matrix, which allows for coalesced memory accesses in the *Row_Reduction* kernel in line 11. The reduction kernel launches a single block with a number of threads equal to the number of integers per matrix row (INTS_PER_ROW or IPR) divided by two. The reduction kernel determines if the blocked process holds any additional resources. If the process does not hold additional resources, sink nodes do not change (block (i) event, see Figure 3.2) and no additional computation is needed. If the process does hold additional resources, we launch the *Request_Blocked* kernel in line 13. This kernel launches N blocks with *INTS_PER_ROW* threads per block to facilitate parallel computation. The *resource request blocked* functionality is discussed in detail in Section 3.3.4, while the *BitMatrix_Transpose*, *tileTranspose* and *Row_Reduction* kernels are discussed in Section 3.3.6.

Lastly, lines 17-26 handle the *resource release* event. In line 19, the *Resource_Release* kernel launches a single block with a single thread to perform the update to AG[ ] reflecting the resource release. Lines 20-22 perform a similar function to what was done in lines 9-11, except this time we transpose and check the AR[ ] matrix. The *Row_Reduction* kernel tells us if any processes are waiting for the released resource. If no processes are waiting, we launch the *Update_Sink_RP* kernel in line 24, which performs the updates on the Sink[ ] and RP[ ] matrices. Otherwise, we grant the released resource to a new process with the *Update_AG_AR* kernel in line 26. Following the resource grant, we update the reachability information in the system by calling the *Release_Update_Reachability* kernel in line 27. The *release resource* event is discussed in detail in Section 3.3.5.

In the GPU-OSDDA algorithm, we set the size of our matrices to powers of two, so that we can perform bit-shifts either left (multiplication) or right (division) for index calculations. Note that M and N do not need to be equal in size, but they must be powers of two. If M and N are not equal, the INTS_PER_ROW value will change depending on which matrix we address. For ease of explanation in this paper, we assume that M and N are equal.

3.3.3 Handling a Resource Request Granted Event

To handle a resource request granted event, GPU-OSDDA launches a kernel with a single block containing a single thread. The computation involved in this event does not advocate parallelism; however, GPU-OSDDA manages and maintains the RAG on the GPU which makes it necessary to launch this small kernel. Algorithm 5 shows the assignments made in our kernel. Since we utilize a bit-packing technique to represent all algorithm matrices, we have to use a special method of referencing the correct process/resource pair for assignment. Figure 3.5 provides an illustration of the indices used to address the matrices in the *Request_Granted* kernel while Figure 3.6 summarizes the actions taken by the kernel.

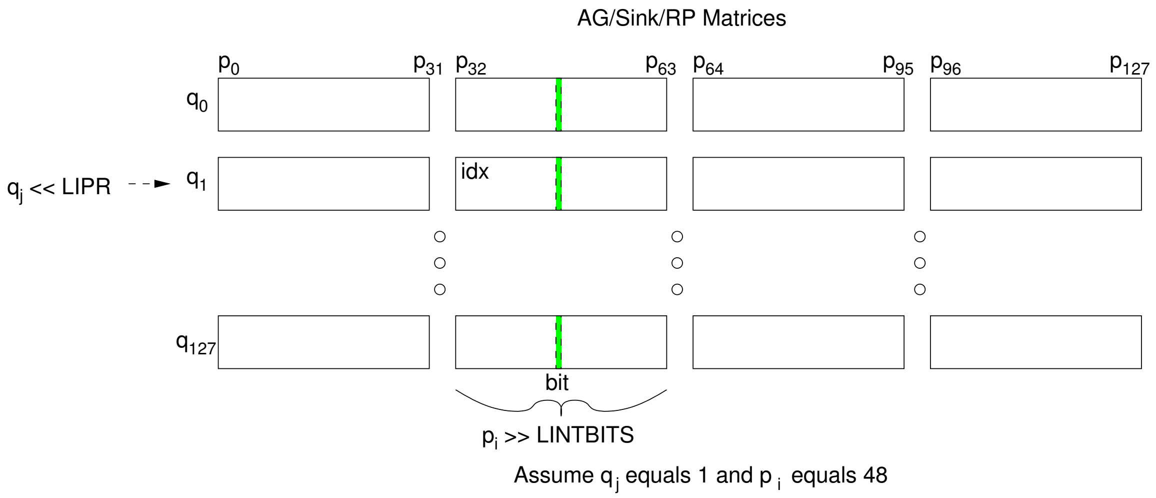


Figure 3.5: Illustration of computation to update AG/Sink/RP Matrix (Resource Granted).

In order to find the correct bit corresponding to the process in the grant event, handled by the Request_Granted computation, we perform in line 3 the modulo of p_i by INTBITS (see Figure 3.5 for illustration of *bit* variable). After obtaining the correct bit in the integer, we find the exact integer index to be altered in each adjacency matrix. This is computed in line 4 by multiplying the row we want (q_j) by the size of each row (note, this is actually a left shift), which is INTS_PER_ROW (refer

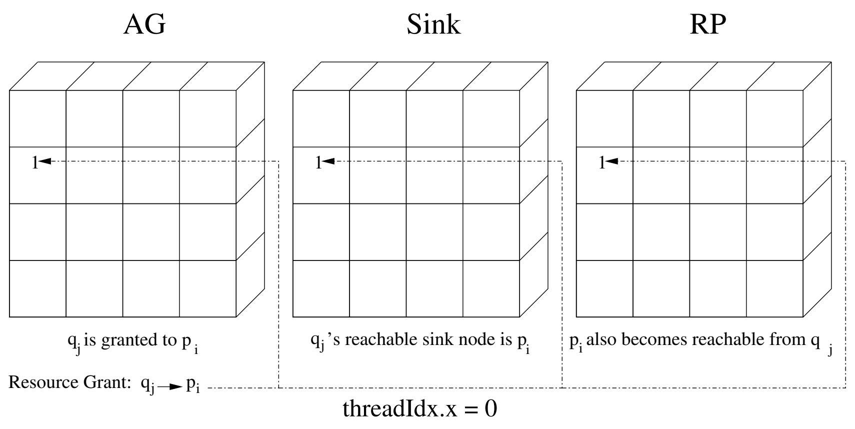

AG
Sink
RP
1
1
1
q_j is granted to p_i
q_j's reachable sink node is p_i
p_i also becomes reachable from q_j
Resource Grant: q_j → p_i
threadIdx.x = 0

Figure 3.6: Resource Request Granted

to Figure 3.4 that has INTS_PER_ROW equal to 4). To this we add the quotient of the process number divided by INTBITS (note, this division is actually a right shift). The sum of these two numbers yields the index of the integer we want to alter in the adjacency matrix. In Figure 3.5, this value is equivalent to the *idx* variable.

In order to perform assignments to the adjacency matrices, we perform bit-wise OR computations with the appropriate mask. The mask is created by shifting a 1 into the location specified by the bit variable we calculated in a prior step. Upon performing the bit-wise OR operations in lines 8-10, our Request_Granted kernel is complete.

Algorithm 5 Request_Granted⋘1,1⋙

```
1:  // Refer to Table 3.1 for variable definitions
2:  // Determine index variables - integer index and bit to alter
3:  $bit \leftarrow p_i \mod INTBITS$
4:  $idx \leftarrow q_j \ll LIPR + p_i \gg LINTBITS$
5:
6:  // Update AG[ ], Sink[ ], and RP[ ] to reflect resource grant
7:  // All assignments are bit-wise computations using index variables
8:  $AG[idx] \mid= (1 \ll (INTBITS - (bit + 1)))$
9:  $Sink[idx] \mid= (1 \ll (INTBITS - (bit + 1)))$
10: $RP[idx] \mid= (1 \ll (INTBITS - (bit + 1)))$
```

As a summary, in the Request_Granted kernel, it can be seen that the AG[ ] matrix is updated to reflect the assignment $q_j \rightarrow p_i$. Similarly, by resource q_j being granted to p_i, p_i becomes the new reachable sink node of resource q_j, denoted by the Sink[ ] matrix. It follows that process p_i is reachable from resource q_j as denoted by the RP[ ] matrix assignment.

3.3.4 Handling a Resource Request Blocked Event

GPU-OSDDA handles a resource request blocked event through several stages. The initial step, which we denote as DeadlockCheck_Init (Algorithm 6), checks whether or not the requesting process is the current sink node of the requested resource. According to [10], if a resource request event occurs where the requesting process is the current sink node of the resource being requested, a cycle forms in the RAG and a deadlock occurs. Otherwise, AR[ ] is updated to reflect the blocked request of $p_i \rightarrow q_j$.

Algorithm 6 DeadlockCheck_Init⋘1,1⋙

```
// Refer to Table 3.1 for variable definitions
// Determine index variables - integer index and bit to alter
// We have two sets of indices as Sink[ ] is resource×process
// and AR[ ] is process×resource
$sbit \leftarrow p_i \mod INTBITS$
$abit \leftarrow q_j \mod INTBITS$
$sidx \leftarrow q_j \ll LIPR + p_i \gg LINTBITS$
$aidx \leftarrow p_i \ll LIPR + q_j \gg LINTBITS$

// If current requested resource's sink node is the requesting process
// then a deadlock exists. Otherwise, block the request by updating AR[ ].
if $(Sink[sidx]\ \&\ (1 \ll (INTBITS - (sbit + 1))) == 1)$ then
    Update Deadlock Flag
else
    $AR[aidx] \mathrel{|}= (1 \ll (INTBITS - (abit + 1)))$
end if
```

The DeadlockCheck_Init kernel utilizes a similar technique seen in the Request_Granted kernel to determine the indices needed for its computations. First, we determine which bit needs to be checked and/or set in the kernel. We first determine the bit to be checked in the Sink[ ] matrix in line 5. Since we want to check a single bit in an integer, we perform the modulo of process p_i by INTBITS. Similarly, we need a bit for the AR[ ] matrix. The reason for building two separate indices is that the Sink[ ] and AR[ ] matrices take the form of *resource* × *process* and *process* × *resource*, respectively. For the AR[ ] matrix, we gain the bit to check by performing the modulo of resource q_j by INTBITS in line 6. As can be seen in Algorithm 6, we continue by constructing two separate global indices; *sidx* and *aidx* in lines 7-8. The *sidx* index yields the position of the integer we want to check in the Sink[ ] matrix, while the *aidx* index yields the position of the integer for assignment in the AR[ ] matrix. The combination of both the global integer index and the associated bit index enables us to check or alter a single bit in the appropriate adjacency matrix. If a deadlock occurs (checked in line 12), then we update the deadlock detection flag in line 13 for the CPU to handle the deadlock event. Otherwise, the resource request is blocked in line 15 by updating the value corresponding to the request $p_i \rightarrow q_j$ in the AR[ ] matrix.

If a deadlock does not occur, reachability information of the RAG needs to be updated if the requesting process holds additional resources. Otherwise, the reachable sink nodes do not change, so no additional computation is necessary. The task of updating reachability information for a RAG is computationally expensive, unlike $\mathcal{O}(1)$ of OSDDA [10]. Nevertheless, our implementation of the reachability update computation benefits greatly from the parallelism offered by the GPU and is further accelerated by our bit-vector approach to the algorithm. Algorithm 7 shows the pseudo-code for our Request_Blocked kernel, which performs the reachability update. As can be seen by the kernel overview in Algorithm 4, we launch N blocks with INTS_PER_ROW threads per block. This kernel structure allows us to perform all bit-wise computations in this kernel simultaneously, except for the serialization of the Sink[ ], RR[ ], and RP[ ] updates per thread. The bit-vector approach we implement allows us to perform computations for 32 processes or resources per integer index in an adjacency matrix. This approach granted us an exponential speedup in the run-time of our algorithm, which will be depicted in the Experimentation and Results section of this paper. The first step in our Request_Blocked kernel is to determine the indices in our adjacency matrices (lines 3-7).

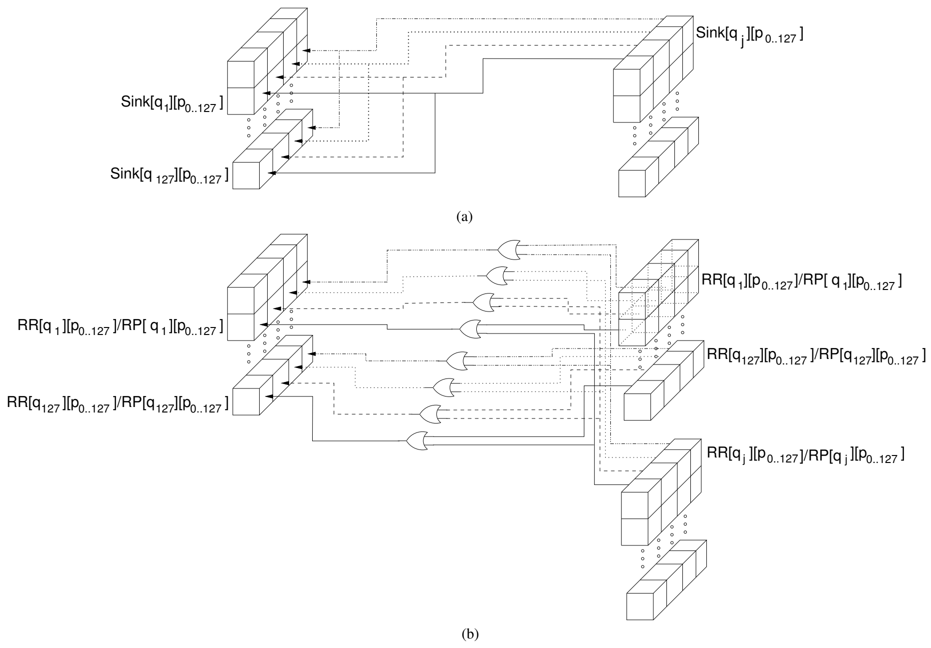
We also allocate a temporary sink matrix, Sink_temp[ ], on the GPU which takes on the values of the Sink[ ] matrix. The Sink_temp[ ] matrix is used to check values of the Sink[ ] matrix without the risk of race conditions between the read and write cycles of the Sink[ ] matrix. After performing our check in line 12, the Sink[ ], RR[ ], and RP[ ] matrices are updated according to [10] in lines 13-15. Line 13 makes the sink node of all resources on the sub-tree of p_i equal to q_j's sink node. Then in lines 14-15, the resources on p_i's sub-tree include the reachable resources and processes of q_j. Following Algorithm 7, a summary of the operations performed for the Request_Blocked kernel is provided.

Algorithm 7 Request_Blocked⋘N,IPR⋙

```
1:  // Refer to Table 3.1 for variable definitions
2:  // Indexing variables for reachability update computation
3:  $row \leftarrow blockIdx.x$
4:  $col \leftarrow p_i \gg LINTBITS$
5:  $bit \leftarrow p_i \mod INTBITS$
6:  $tid \leftarrow threadIdx.x$
7:  $idx \leftarrow row \ll LIPR + col$
8:
9:  // For all the resources that belong to the sub-tree of pi,
10: // their sink nodes are now set to $q_j$'s; their reachable
11: // resource and process nodes include $q_j$'s.
12: if $((Sink\_temp[idx]\ \&\ (1 \ll (INTBITS - (bit+1)))) == 1)$ then
13:     $Sink[row \times IPR + tid] \leftarrow Sink[q_j \times IPR + tid]$
14:     $RR[row \times IPR + tid] \leftarrow RR[row \times IPR + tid] \mid RR[q_j \times IPR + tid]$
15:     $RP[row \times IPR + tid] \leftarrow RP[row \times IPR + tid] \mid RP[q_j \times IPR + tid]$
16: end if
```

As in [10], for all resources belonging to p_i's sub-tree, their sink nodes are set to q_j's sink node. Their RR[ ] and RP[ ] matrices are also updated to include q_j's reachable nodes. The biggest advantage we gain during this computation is that per each assignment or bit-wise OR operation, we effectively update 32 process/resource pairs per matrix per thread. Figure 3.7 depicts the operations taking place during the reachability computation, where resources q_1 and q_{127} have a sink node of p_i and q_j is assumed to be q_0. Figure 3.7(a) depicts line 13 in the Request_Blocked kernel where the reachable sink nodes are updated. Figure 3.7(b) depicts lines 14-15 in the Request_Blocked kernel where the reachable processes and resources of p_i's sub-tree resources are updated to include q_j's.

In our implementation, each row of a matrix is handled per block with all columns being handled per thread. In the bottom part of the figure, the RR[ ]/RP[ ] matrix is split into four rows in order to better show the handling of the computation. Furthermore, we assume that the resource q_j falls within the first row (since q_j is equivalent to q_0) in order to make the diagram easily understood. In the bottom part of the figure, we show the logic diagram for a single matrix, but know that an identical operation is occurring for both matrix RR[ ] and RP[ ], respectively.



Sink[q_j][p_0..127]
Sink[q_1][p_0..127]
Sink[q_127][p_0..127]
(a)
RR[q_1][p_0..127]/RP[q_1][p_0..127]
RR[q_127][p_0..127]/RP[q_127][p_0..127]
RR[q_j][p_0..127]/RP[q_j][p_0..127]
(b)

Figure 3.7: Calculation to determine reachability in RAG.

(a) Update Reachable Sink Node (Line 13)

(b) Update Reachable Processes and Resources (Lines 14-15)

3.3.5 Handling a Resource Release Event

In handling a *resource release* event, GPU-OSDDA first has process p_i release resource q_j by updating AG[ ], as the pseudo-code in Algorithm 8 depicts. We first determine the bit that represents the process p_i releasing resource q_j in line 3. Following a familiar procedure, we compute p_i modulo INTBITS to represent our process bit. Then to determine the integer index into the AG[ ] matrix that we need to alter, we compute q_j left-shifted by $log_2(INTS\_PER\_ROW)$ to give us the proper row of AG[ ] in line 4. We then add this to p_i right-shifted by $log_2(INTBITS)$ to give us the column integer that we want to address. This sum then yields the index to address in AG[ ]. From there, the operation in line 7 performs a bit-wise AND operation that updates AG[ ] reflecting that process p_i released resource q_j.

Algorithm 8 Release_Resource⋘1,1⋙

```
1: // Refer to Table 3.1 for variable definitions
2: // Determine index variables - integer index and bit to alter
3: $bit \leftarrow p_i \mod INTBITS$
4: $idx \leftarrow q_j \ll LIPR + p_i \gg LINTBITS$
5:
6: // Release resource from AG matrix by clearing the bit
7: $AG[idx]\ \&= \sim(1 \ll (INTBITS - (bit + 1)))$
```

GPU-OSDDA then checks if a process is waiting on q_j by performing a reduction on AR_trans[q_j][ ] (transpose of the AR[ ] matrix). In the event that this reduction returns 0, it informs us that no process is waiting on q_j and that it belongs to a release event (i) explained in Section 3.2.2 (see Figure 3.3). From there, GPU-OSDDA updates the Sink[ ] and RP[ ] matrices to indicate that q_j has no sink and that p_i is no longer reachable from q_j. Algorithm 9 depicts the update process of Sink[ ] and RP[ ]. Since both Sink[ ] and RP[ ] are *resource*×*process* matrices, we are able to utilize the same bit and index to update necessary information. Notice the procedure in lines 3-4 in Algorithm 9 is similar in terms of finding the appropriate bit and index. After computing this information, we perform a bit-wise AND operation in lines 7 and 9 to clear the corresponding bit in the adjacency matrices.

Algorithm 9 Update_Sink_RP⋘1,1⋙

```
1:  // Refer to Table 3.1 for variable definitions
2:  // Determine index variables - integer index and bit to alter
3:  $bit \leftarrow p_i \mod INTBITS$
4:  $idx \leftarrow q_j \ll LIPR + p_i \gg LINTBITS$
5:
6:  // $q_j$ is isolated; thus $q_j$ has no sink - clear bit
7:  $Sink[idx] \;\&= \;\sim(1 \ll (INTBITS - (bit + 1)))$
8:  // $p_i$ is no longer reachable from $q_j$ either - clear bit
9:  $RP[idx] \;\&= \;\sim(1 \ll (INTBITS - (bit + 1)))$
```

If the reduction of AR[ ][q_j] is not equal to 0, this indicates that the release event is a release (ii) event (see Figure 3.3). In this case, GPU-OSDDA updates AG[ ] and AR[ ] to indicate that the released resource is granted to a waiting process. Algorithm 10 depicts the update process for AG[ ] and AR[ ]. Since the AG[ ] and AR[ ] matrices take the form of *resource* $\times$ *process* and *process* $\times$ *resource*, respectively, they both need their own bit and global index variables to update the correct bit. As performed for all of our updates thus far, we find the correct bit by computing the modulo of the bit we want with INTBITS in lines 5-6. Following that, lines 7-8 perform familiar computations to find the integer index into the adjacency matrix that we want to update. Finally, we update AG[ ] and AR[ ] by performing bit-wise OR operations on the corresponding index and bit in lines 11-12.

Algorithm 10 Update_AG_AR⋘1,1⋙

```
1:  // Refer to Table 3.1 for variable definitions
2:  // Determine index variables - integer index and bit to alter
3:  // We have two sets of indices as AG[ ] is resource×process
4:  // and AR[ ] is process×resource
5:  $tbit \leftarrow p_t \mod INTBITS$
6:  $qbit \leftarrow q_j \mod INTBITS$
7:  $tidx \leftarrow p_t \ll LIPR + q_j \gg LINTBITS$
8:  $qidx \leftarrow q_j \ll LIPR + p_t \gg LINTBITS$
9:
10: // $q_j$ is now granted to $p_t$ - set appropriate bits
11: $AG[qidx] \mathrel{|}= (1 \ll (INTBITS - (qbit + 1)))$
12: $AR[tidx] \mathrel{|}= (1 \ll (INTBITS - (tbit + 1)))$
```

Following this step, reachability and sink information needs to be updated. Algorithm 11 provides our pseudo-code in handling the update process. The Release_Update_Reachability kernel takes advantage of the parallelism provided by the GPU. For this kernel, we launch N blocks with a number of threads equal to INTS_PER_ROW for each block. The kernel also utilizes the same methods as prior kernels to check, set, and clear bits in our adjacency matrices.

To start, in line 3 we create an array in shared memory called *newSink*[ ], which we use to update sink information later in the kernel. Lines 7-8 assign the block and thread variables to row and tid, respectively, which are used for calculating the bit, column, and index variables. It can be seen that the same familiar process to find needed bits (lines 9-11), columns (lines 12-14), and indices (lines 15-17) has been performed. This kernel, however, has every block handle a row in the matrices involved in computation. In lines 20-24, the newSink shared variable is populated to hold the new sink node, i.e., process p_t. In line 27, we check the RR[ ] matrix to obtain all of q_j's sub-tree resources. After finding all of q_j's sub-tree resources, we assign them the new sink node of p_t in line 29. Since p_i released resource q_j, p_i is no longer reachable from q_j and is removed from the RP[ ] matrix in line 34. In line 36, we check if p_t's sub-tree resources were able to reach q_j. If p_t's sub-tree resources were able to reach q_j, q_j is no longer reachable so we remove q_j from the RR[ ] matrix in line 38. Otherwise, for q_j's sub-tree resources that were not reachable to p_t, p_t becomes reachable and the RP[ ] matrix is updated to reflect the change in line 41.

As a summary, the Release_Update_Reachability kernel updates all sink nodes in q_j's sub-tree to p_t. The process p_i that released the resource is no longer reachable from q_j and its sub-tree, so the RP[ ] matrix is updated accordingly. The final steps in our computation require that all p_t's sub-tree resources that were previously able to reach q_j, be removed from the RR[ ] matrix. Conversely, if q_j's sub-tree resources were not reachable to p_t, p_t now becomes reachable and the RP[ ] matrix is updated.

Algorithm 11 Release_Update_Reachability⋘N,IPR⋙

```
// Refer to Table 3.1 for variable definitions
// Shared variable holds new sink information
__shared__ $newSink[IPR]$

// Determine index variables - integer index and bit to alter
// Multiple indices are needed since we reference three different variables: $p_t$, $p_i$, and $q_j$
$row \leftarrow blockIdx.x$
$tid \leftarrow threadIdx.x$
$tbit \leftarrow p_t \mod INTBITS$
$pbit \leftarrow p_i \mod INTBITS$
$qbit \leftarrow q_j \mod INTBITS$
$tcol \leftarrow p_t \gg LINTBITS$
$pcol \leftarrow p_i \gg LINTBITS$
$qcol \leftarrow q_j \gg LINTBITS$
$tidx \leftarrow row \ll LIPR + tcol$
$pidx \leftarrow row \ll LIPR + pcol$
$qidx \leftarrow row \ll LIPR + qcol$

// Initialize newSink so $p_t$ is the new sink node and synchronize threads
$newSink[tid] \leftarrow 0$
if $tid == tcol$ then
    $newSink[tcol] \mid= (1 \ll (INTBITS - (tbit + 1)))$
end if
__syncthreads()

// For $q_j$ and its sub-tree resources
if $(RR[qidx]\ \&\ (1 \ll (INTBITS - (qbit + 1))) == 1)$ then
    // Assign the new sink node information
    $Sink[row \times IPR + tid] \leftarrow newSink[tid]$
    // Avoids writing to the same location for each thread
    // although the remaining threads go inactive
    if $tid == 0$ then
        // $p_i$ is no longer reachable
        $RP[pidx]\ \&= \sim (1 \ll (INTBITS - (pbit + 1)))$
        // For $p_t$'s sub-tree resources that were able to reach $q_j$
        if $(RP[tidx]\ \&\ (1 \ll (INTBITS - (tbit + 1))) == 1)$ then
            // $q_j$ is no longer reachable
            $RR[qidx]\ \&= \sim (1 \ll (INTBITS - (qbit + 1)))$
        else
            // $p_t$ becomes reachable
            $RP[tidx] \mid= (1 \ll (INTBITS - (tbit + 1)))$
        end if
    end if
end if
```

3.3.6 Supplementary Kernels

One may notice that there were two additional kernels in the overview code that were not discussed in the design section, the BitMatrix_Tranpose and Row_Reduction kernels. The BitMatrix_Transpose kernel was used to transpose our bit-vector matrices in order to ensure coalesced global memory accesses in our Row_Reduction kernels. This is why the BitMatrix_Transpose kernel always precedes the Row_Reduction kernel.

Performing the transpose of a bit-vector matrix can be a complicated task (see Figure 3.8). This computation has been done in [22]; however, we needed to parallelize the computation to some degree and make it applicable to our algorithm. As such, we adopted the bit-vector matrix transpose function seen in [22]. The transpose in [22] only works on a 32-bit×32-bit matrix. To make this solution fit to our problem, we sub-divided the transpose of our matrices into 32-bit×32-bit tiles (seen as TILE_DIM in Algorithm 1). We achieve this by launching TILE_DIM blocks and threads. This ensures that each thread handles a 32-bit×32-bit tile of the matrix. Following the transpose of each 32-bit×32-bit tile, all tiles are transposed to effectively transpose the entire bit-vector matrix (see Figure 3.9). By performing the transpose of our entire matrix in tiles, we were not only able to enable coalesced global memory accesses for following kernels but also able to parallelize the transpose, thus leading to a fast bit-vector matrix transpose operation.

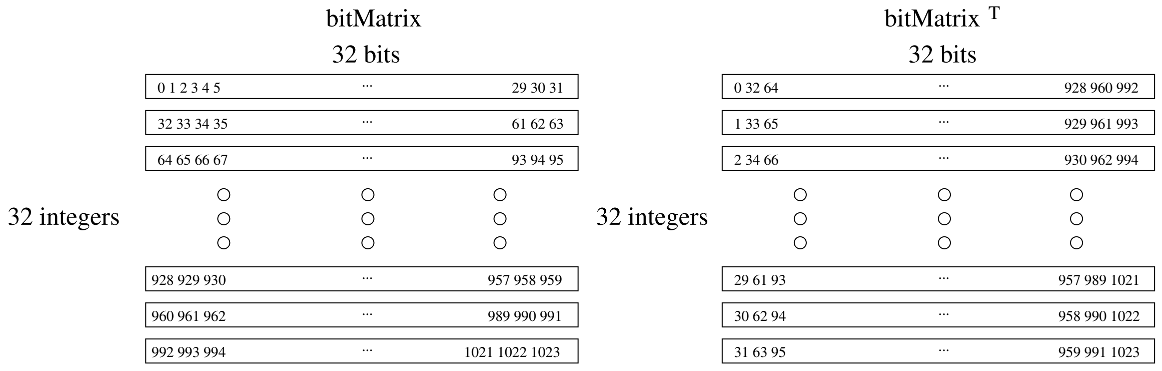


Figure 3.8: Example of a bit-matrix transpose of a 32×32 matrix.

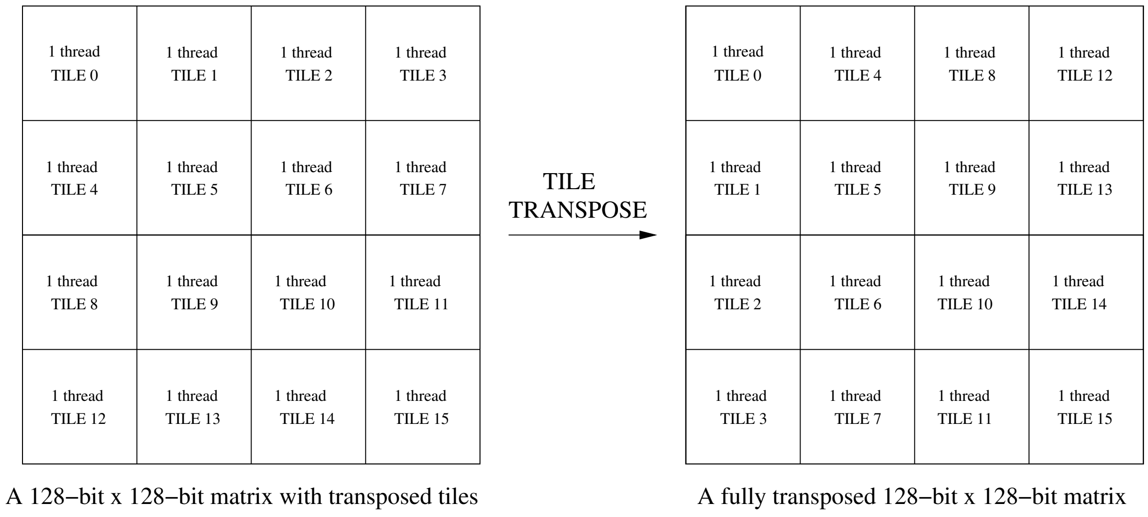


Figure 3.9: Example of full matrix transpose after bit-matrix transpose in tiles.

Our Row_Reduction kernel also greatly benefits from our bit-vector approach. All of the reductions in GPU-OSDDA are used to check to see if the row or column of a particular matrix is zero. This works well with the bit-vector approach. When we perform our reduction, we simply add each integer of a particular row and column together (an add reduction) and determine if the total is zero or not. This allowed us to compute the row or column reduction 32× faster than if we had not taken the bit-vector approach in storing our adjacency matrices. To optimize our reduction kernels even further, we perform the first addition of the reduction when we populate shared memory, thus allowing us to launch half the number of threads required in a standard reduction. From that point, we perform the reduction while unrolling the last warp utilizing the *warpReduce* function found in [23].

While these kernels are supplementary to the core GPU-OSDDA functionality, they provide substantial speedups with regard to the run-time of our algorithm. Without performing the bit-matrix transpose that advocates global memory coalescing in kernels, we would have seen a great loss in efficiency (coalesced global memory accesses would not have occurred) and in run-time.

3.4 Experimentation and Results

A serial version of GPU-OSDDA is first implemented using the C language referred to as CPU-OSDDA. This version utilizes no parallel programming primitives (such as OpenMP) in order to fully demonstrate the capabilities of parallel processing on GPU. The development of a parallel CPU version of our algorithm would provide significant difficulty as the CPU is limited in the number of threads it may spawn to accomplish the massive parallel computations required. In addition, the CPU should be available to handle the normal stream of tasks associated with computing, as mentioned in Section 1.4. By launching the maximum number of threads (or utilizing vectorized instructions) on the CPU, we limit the capability of the processor to perform processing of normal computing tasks.

All experiments were performed on an Intel ® Core i7 CPU @ 2.8 GHz with 12 GB RAM. The CUDA GPU-OSDDA implementation was tested on three different GPUs: GTX 670, Tesla C2050, and Tesla K20c. The GTX 670 has 7 SMXs (1344 CUDA Cores) with 2 GB Global Memory, the Tesla C2050 has 14 SMs (448 CUDA Cores) with 3 GB Global Memory, and the Tesla K20c has 13 SMXs (2496 CUDA Cores) with 5 GB Global Memory.

To verify the correctness of our algorithm, both CPU-OSDDA and GPU-OSDDA were tested against RAGs of different sizes (processes x resources). Once results were verified as correct, more complex and larger RAGs were generated to test the scalability of GPU-OSDDA for larger set sizes. Tables 3.2 and 3.3 show the run-times of CPU-OSDDA, our initial approach to GPU-OSDDA, and the bit-packed approach to GPU-OSDDA, respectively. Figure 3.10 then depicts the associated speedups of each set size on each piece of target hardware. As can be seen by our results, an increase in process and resource counts dramatically increases CPU-OSDDA's run-time. However, GPU-OSDDA scales well with increasing process and resource amounts. Our initial approach without the use of bit-packing was able to achieve speedups of an order of magnitude higher than CPU-OSDDA. The integration of our bit-packing approach, however, enabled us to achieve additional speedups of an order of magnitude higher than our initial approach.

Table 3.2: Run-Time/Speedup of CPU-OSDDA and GPU-OSDDA (Initial)

Input	CPU-OSDDA	GTX 670	Tesla C2050	Tesla K20c
512×512	0.22/0x	0.08/2.75x	0.08/2.75x	0.05/4.40x
1024×1024	2.57/0x	0.26/9.88x	0.27/9.52x	0.24/10.71x
2048×2048	18.60/0x	1.49/12.48x	1.17/15.90x	1.50/12.40x
4096×4096	170.81/0x	10.83/15.77x	8.10/21.09x	11.00/15.53x
8192×8192	1383.97/0x	81.97/16.88x	58.15/23.80x	86.66/15.97x

Table 3.3: Run-Time/Speedup of CPU-OSDDA and GPU-OSDDA (Bit-Packed)

Input	CPU-OSDDA	GTX 670	Tesla C2050	Tesla K20c
512×512	0.22/0x	0.07/3.14x	0.07/3.14x	0.04/5.50x
1024×1024	2.57/0x	0.11/23.36x	0.12/21.42x	0.09/28.56x
2048×2048	18.60/0x	0.26/71.54x	0.33/56.36x	0.23/80.87x
4096×4096	170.81/0x	0.88/194.10x	1.42/120.29x	0.81/210.88x
8192×8192	1383.97/0x	4.36/317.42x	8.52/162.44x	4.68/295.72x

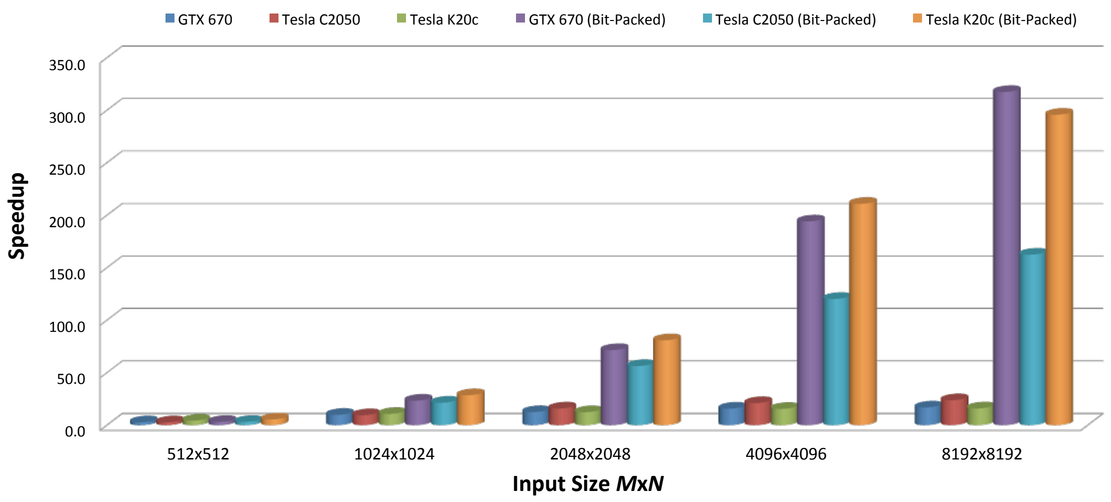


Figure 3.10: GPU-OSDDA Speedup

3.5 Conclusion

A new approach to deadlock detection for single-unit systems on GPU has been devised and developed using CUDA C. By leveraging facts about single-unit systems, we were able to devise a bit-vector technique for storing our matrices which led to efficient algorithmic computations and drastically saved memory space on the GPU. These factors alone allow GPU-OSDDA to handle systems with increasing amounts of processes and resources. Since GPU-OSDDA performs deadlock computation/detection on the GPU, our algorithm acts as an interactive service to resource events occurring on the CPU. Our experimental results show promising speedups in the range of 5-317X, thus making GPU-OSDDA a viable solution to deadlock detection on single-unit resource systems with a large number of processes and resources.

4 GPU-LMDDA: A GPU-BASED DEADLOCK DETECTION ALGORITHM FOR MULTI-UNIT RESOURCE SYSTEMS

4.1 Introduction

This chapter discusses GPU-LMDDA, a GPU-based approach to deadlock detection in multi-unit resource systems. The following section states the system assumptions and provides background information regarding the core methodology of GPU-LMDDA. GPU-LMDDA adopts the resource event classification scheme found in the hardware-based deadlock detection algorithm known as *Logarithmic Multi-Unit Deadlock Detection Algorithm* (LMDDA) [12]. Section 4.2.2 includes the underlying theory of LMDDA and discusses how it classifies and handles resource events, as well as its $log_2(min(m,n))$ overall run-time capability in hardware. This is followed by our algorithm design and its implementation in Section 4.3. Following the design discussion is the Experimentation and Results Section (Section 4.4) which details the run-times and speedups achieved by GPU-LMDDA.

4.2 Background

4.2.1 Assumptions and Terms

A multi-unit resource system (see Definition 1.2.14) is an $m \times n$ system where m and n are the number of processes and resources, respectively. GPU-LMDDA, like the algorithm it is modeled after, utilizes a weighted RAG (see Definition 1.2.29) and its adjacency matrices AG and AR to represent the resource allocation information in the system. This GPU-based approach to LMDDA also adheres to the following system assumptions:

1. Each type of resource has a fixed total number of units [12].
2. A resource unit is granted immediately if it is available. Thus, the entire system is always in an *expedient* state (see Definition 1.2.16) [12].
3. A process requests or releases one resource unit at a time, so called a single-unit request system (see Definition 1.2.17). Thus, a process is blocked as soon as it requests an unavailable resource unit [12].
4. Since the target system has multi-unit resources, a knot in the RAG is a necessary and sufficient condition for deadlock (see Theorem 1.2.39).
5. Resource events are managed centrally (e.g., by the OS).

Like GPU-OSDDA discussed in Chapter 3, GPU-LMDDA also utilizes the *active process* (see Definition 1.2.15) and the *sink process node* (see Definition 1.2.35) to aid the computations of deadlock detection.

4.2.2 Underlying Theory of LMDDA

LMDDA has a slightly different approach to the way it classifies resource events and determines deadlock in a system since it deals with multi-unit resources. It also has the best known run-time of any multi-unit resource deadlock detection algorithm to date. The resource events in LMDDA are comprised of the following: *granted resource requests*, *blocked resource requests*, and *resource release* [12]. In order to determine if a deadlock exists in a RAG consisting of both single-unit and multi-unit resources, the RAG of interest must contain a knot (see Theorem 1.2.39). The resource events, node traversal, and deadlock preparation capability of LMDDA are briefly discussed in the following subsections.

Resource Events

First discussed is the *granted resource request* event type. In the event that a process p_i requests a unit of resource q_j (via Assumption 3 in Section 4.2.1), two circumstances may arise assuming that resource q_j has at least one available unit. If process p_i has not been previously granted a resource of q_j, then q_j is granted (see Definition 1.2.16) to p_i with a weight of 1 (see Definition 1.2.29). On the other hand, if resource q_j has been previously granted to process p_i and an additional unit is requested, then the weight of the grant edge ($q_j \rightarrow p_i$) is increased by one. Depending on if process p_i has already attained a unit of resource q_j, two *granted resource request* scenarios exist with illustrations provided in Figure 4.1:

1. Grant (i) - Before q_j is granted, no unit of q_j has been assigned to p_i. The weight of the granted edge is assigned a value of 1 [12].
2. Grant (ii) - Before q_j is granted, some units of q_j have been assigned to p_i. The weight of the already granted edge is increased by 1 [12].

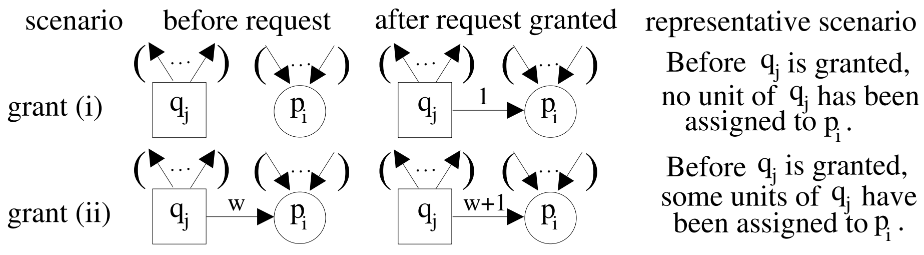

scenario
before request
after request granted
representative scenario
grant (i)
grant (ii)
q_j
p_i
1
w
w+1
Before q_j is granted, no unit of q_j has been assigned to p_i .
Before q_j is granted, some units of q_j have been assigned to p_i .

Figure 4.1: Scenarios of *resource request granted* events (Courtesy of Xiao and Lee [10]).

Discussed next is the *blocked resource request* event type. In this event type, process p_i makes a request for a unit of resource q_j, where p_i is an *active process* (see Definition 1.2.15). When the process p_i is blocked for requesting resource q_j, the request edge $p_i \rightarrow q_j$ is inserted into the RAG. As a result, all processes that are reachable (see Definition 1.2.32) from the requested resource q_j, also become reachable from the resources on process p_i's sub-tree. If p_i has no sub-tree resources, then the reachable processes of every resource in the RAG remain unchanged. The *blocked resource request* events can be handled under a single scenario, with an illustration provided in Figure 4.2:

1. Block (i) - Before the request is blocked, p_i (q_j) may or may not have incoming edges [12].

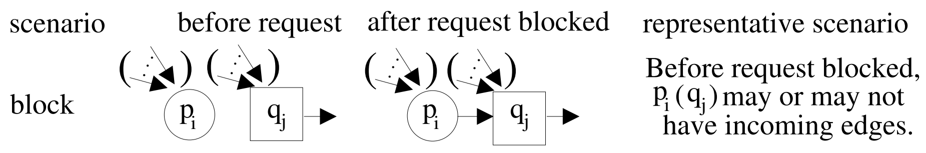


Figure 4.2: Scenarios of *resource request blocked* events (Courtesy of Xiao and Lee [10]).

Last is the *resource release* event type. In this event type, process p_i releases a unit of resource q_j, and the released resource may be granted to a waiting process p_t if q_j has an incoming edge from p_t. The trivial case of the *resource release* event occurs when more than a single unit of resource q_j has been granted to process p_i and q_j has no incoming edges (pending resource requests). If p_i releases a unit of q_j under this scenario, when the resource unit is released, the grant edge $q_j \rightarrow p_i$ still exists. Thus, no edge changes occur in the RAG; there is simply a decrease in the weight of the granted edge. As a result, the reachable processes of every resource remain the same.

If the *resource release* scenario is not like that mentioned above, it can be unclear on how the released resource affects the reachable processes of each resource. For example, say resource q_j has incoming edges (pending requests) prior to the resource release; after the release occurs, q_j is granted to one of the requesting processes (say p_t). While the request edge $p_t \rightarrow q_j$ no longer exists after the *resource release* event, a resource on p_t's sub-tree may still be able to reach process p_i via another path not involving p_t [12]. In these cases, LMDDA utilizes a node hopping mechanism to determine reachable processes from each resource in the RAG (further discussed in the next section). Summarized here are the *resource release* event scenarios with illustrations provided in Figure 4.3:

1. Release (i) - Before a resource is released, more than one unit of q_j were assigned to p_i; q_j has no incoming edges [12]. As a result, the AG[ ] matrix is updated to reflect the resource release, but no update of the RP[ ] matrix is required.

2. Release (ii) - Before a resource is released, more than one unit of q_j were assigned to p_i; q_j has incoming edges [12]. As a result, the AR[ ] and AG[ ] matrices are updated to reflect the resource release and the newly granted resource request ($p_t \rightarrow q_j$). Then reachability information for the RAG is computed by updating the RP[ ] matrix using the node hopping mechanism (discussed in next section).

3. Release (iii) - Before a resource is released, only one unit of q_j was assigned to p_i; q_j may or may not have incoming edges [12]. As a result, the AG[ ] matrix is updated to reflect the resource release, followed by an update of the RP[ ] matrix to determine reachability information in the RAG. If q_j has incoming requests, one of the requests is granted (updating the AR[ ] matrix). Otherwise, the AR[ ] matrix does not change.

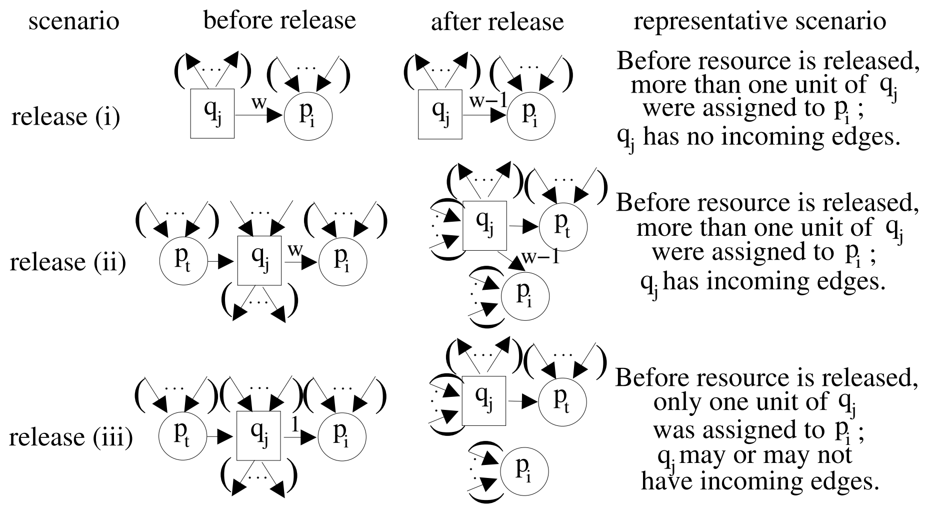

scenario
before release
after release
representative scenario
release (i)
release (ii)
release (iii)
q_j
p_i
p_t
w
w−1
1
Before resource is released, more than one unit of q_j were assigned to p_i ; q_j has no incoming edges.
Before resource is released, more than one unit of q_j were assigned to p_i ; q_j has incoming edges.
Before resource is released, only one unit of q_j was assigned to p_i ; q_j may or may not have incoming edges.

Figure 4.3: Scenarios of *resource release* events (Courtesy of Xiao and Lee [10]).

Node Hopping Mechanism

The node hopping mechanism is used to determine reachable processes from each resource in a RAG. Knowing resource reachability information in the RAG is required to determine if a knot exists in the RAG. By using the reachability information calculated by the node hopping mechanism, the algorithm is able to satisfy the requirements of Theorem 1.2.39.

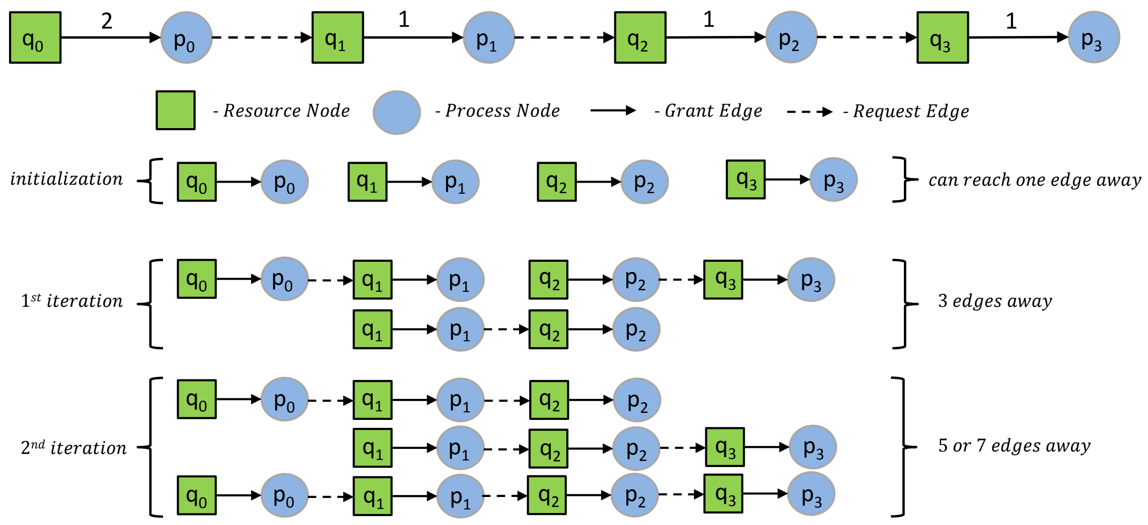


Figure 4.4: An example of the node hopping mechanism finding reachable processes (Courtesy of Xiao and Lee [12]).

First, the closest reachable process from each resource is determined, or in other words, the processes that are directly connected to each resource. In Figure 4.4, this step is known as *initialization*. As an example, the *initialization* step in Figure 4.4 determines that processes p_0, p_1, p_2, and p_3 are reachable from resources q_0, q_1, q_2, and q_3, respectively. In this *initialization* step, all reachable processes up to 1 edge (or hop) away can be determined.

Next, the node hopping mechanism goes through a number of iterations to determine reachability information for the entire RAG. For all upcoming iterations, the node hopping mechanism plugs in request edges to further determine reachability information in the system. Looking at the first iteration in Figure 4.4, if the request edge $p_0 \rightarrow q_1$ is plugged in, then process p_1 is now reachable from resource q_0. Similarly, by plugging in request edges $p_1 \rightarrow q_2$ and $p_2 \rightarrow q_3$, p_2 becomes reachable from q_1 and p_3 becomes reachable from q_2, respectively. Notice, during this first iteration that all reachable processes up to 3 edges away (or 3 hops) can be determined.

For the second iteration of the nodding hopping mechanism, additional request edges are plugged in. In Figure 4.4, the request edge $p_1 \rightarrow q_2$ is plugged in and results in p_2 becoming reachable from q_0 and q_1. Now, all reachable processes of each resource has been determined. The second iteration of the node hopping mechanism is able to determine reachable processes that are either 5 or 7 edges (5 or 7 hops) away from each resource.

Each iteration of the node hopping mechanism builds upon the prior iteration to find reachable processes that are further away by plugging in additional request edges. Since it is not known in advance which request edge connects a set of nodes, all request edges must be tried to determine reachability across the RAG. There are no dependencies between request edge computations, so all computations may be performed in parallel. The node hopping mechanism finds reachable processes that are at most 1, 3, 7, 15, 31, ... edges away from every resource in sequence [12]. From this sequence, it is known that prior to the sth iteration (where s $\geq$ 1), reachable processes within 2^s-1 edges (or hops) away from each resource can be determined. Since all reachable processes are at most $2 \times min(m,n)$-1 edges away from any resource in the RAG [12], to fully determine all reachable processes of every resource in the system, the number of iterations required is equal to $\lceil log_2(min(m,n)) \rceil$ [12].

Deadlock Preparation in $log_2(\boldsymbol{min(m,n)})$ Time

To perform deadlock preparation in $log_2(min(m,n))$ time, LMDDA determines the reachable processes of every resource in the system utilizing the node hopping mechanism described above. The reachable process information is stored in a matrix known as ReachableProcess (RP). The matrix RP is an n by m matrix where RP[j][i] is 1 if p_i is reachable from q_j, or 0 otherwise [12]. RP is summarized as follows:

$$RP[j][i]_{nxm} = \begin{cases} 1 & \text{if a path exists from resource } q_j \text{ to } p_i, \\ 0 & \text{otherwise.} \end{cases}$$

LMDDA also determines which processes are *sink nodes* (see Definition 1.2.35) in the system. The corresponding *sink nodes* are stored in a vector known as Sink. The vector Sink is an m by *1* vector where Sink[i] is 1 if p_i is a sink node. Sink is summarized as follows:

$$Sink[i]_{mx1} = \begin{cases} 1 & \text{if } p_i \text{ is a sink node,} \\ 0 & \text{otherwise.} \end{cases}$$

After determining reachability information of all processes and resources in the system, the deadlock status of the system may be determined. In the event of a *resource request blocked*, *sink nodes* must be determined using the AR matrix (see Definition 1.2.28). Then, a logical AND is performed between Sink[ ] and RP[j][ ]. If the result of this logical AND operation is equal to p_i's bitmask (see Definition 1.2.36), then a deadlock exists according to Theorem 1.2.39.

4.3 GPU-LMDDA Design

4.3.1 Introduction

The initial design of GPU-LMDDA utilized characters to represent the adjacency matrix elements. It was thought that the use of characters would help eliminate as much unused memory space as possible, thus allowing for a larger number of process and resource counts the algorithm could handle. This also reduces the amount of data fetched from memory per global memory access and allows additional data to be held in the L1 and L2 caches of the GPU. After this version of the algorithm was optimized, the maximum speedup achieved over the CPU implementation, dubbed CPU-LMDDA, was 64X.

It was found that the bottle neck of the algorithm was within the implementation of the node hopping mechanism to find reachability information in the system.

Following the development of the bit-vector implementation of GPU-OSDDA (see Chapter 3), it was hypothesized that similar speedups may be achieved by utilizing the bit-vector structure for GPU-LMDDA. However, GPU-LMDDA has several caveats with regards to implementing a bit-vector version. The first is that all matrix elements are not represented solely as a series of 0s and 1s. Since GPU-LMDDA utilizes a weighted RAG (see Definition 1.2.29), the AG matrix may contain values greater than 1. To circumvent this problem, the AG matrix elements were represented with integers, while the rest of the algorithm's matrices were represented with integer length bit-vectors. Another caveat of the bit-vector implementation in GPU-LMDDA is that when updating reachability information, the algorithm is unable to write all consecutive values at once because of the nature of the node hopping mechanism (see Section 4.2.2). As a result there are portions of GPU-LMDDA where writes to a single bit-packed integer must be atomic and are therefore serialized. The code segments where this occurs will be discussed in the forthcoming sections when the algorithm design is described in detail. While GPU-LMDDA provided several interesting problems to overcome, the use of bit-vectors allowed GPU-LMDDA to reduce its memory footprint by approximately a factor of 32. The bit-vector approach also allowed for simplification of reductions and accelerated bit-wise computations of GPU-LMDDA.

4.3.2 Bit-Vector Design

Since GPU-LMDDA is based off of a multi-unit resource system and utilizes a weighted RAG (see Definition 1.2.29), there may be more than one unit of a resource granted to any given process in the system. This implies that all matrices in GPU-LMDDA, with the exception of the AG[ ] matrix, can have their elements represented by binary values (0,1). Instead of using an 8 or 32-bit variable to hold a 1-bit value, we bit-pack 32 processes or resources into a single 32-bit unsigned integer (just as in

GPU-OSDDA). Figure 4.5 shows how we could create a 128×128 element bit-vector matrix using 32-bit unsigned integers. Like the bit-packing method used in GPU-OSDDA, each box in Figure 4.5 represents a 32-bit unsigned integer. Note, Figure 4.5 does not apply to the AG[ ] matrix.

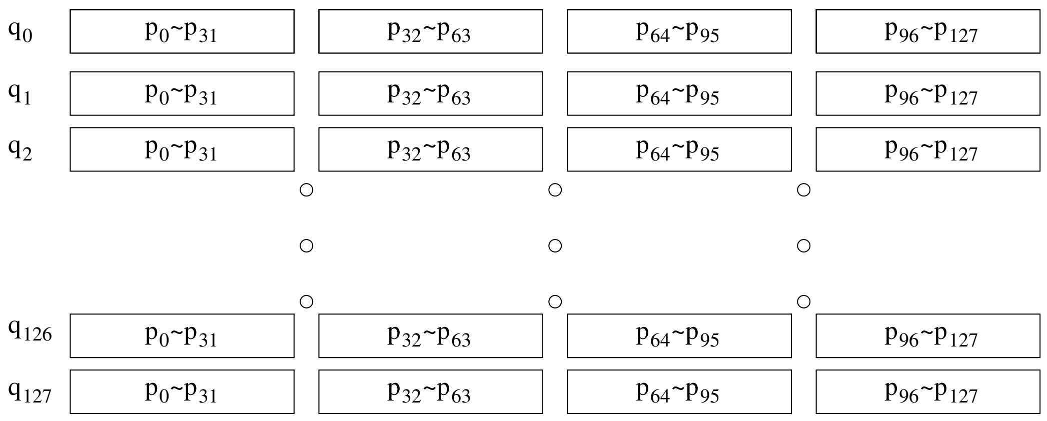


Figure 4.5: A 128x128 Bit-Vector Adjacency Matrix

Table 4.1 provides a description of some key variables used throughout GPU-LMDDA. All descriptions are made with respect to Figure 4.5.

Table 4.1: Common variables used throughout GPU-LMDDA.

Variable	Description	Values in Figure 3.4	Comment
INTS_PER_ROW (IPR)	The number of integers in a bit-packed row	4	Used to calculate row index
INTBITS	The number of bits per unsigned integer	32	Used to determine integer and bit to alter
LIPR	Equivalent to $log_2(INTS\_PER\_ROW)$	2	Used in multiply bit-shift calculations
LINTBITS	Equivalent to $log_2(INTBITS)$	5	Used in divide bit-shift calculations

Now that the core methodology behind GPU-LMDDA has been discussed, the overall kernel structure is shown in Algorithm 12. The pseudo-code presents the kernels that are invoked upon each resource event type, with the event information being contained in the *event* structure variable. The event structure variable consists of the elements in Table 4.2:

Table 4.2: Event structure in GPU-LMDDA.

Variable	Description
p_i	Used to hold process id (i.e., p_i) involved in event
q_j	Used to hold resource id (i.e., q_j) involved in event
p_t	Used to hold process id of the process to be granted resource q_j in the resource release (ii) or (iii) event (i.e., p_t)
event	States the event type (0 = Granted, 1 = Blocked, 2 = Released)

In our pseudo-code description of GPU-LMDDA, for the sake of understanding and simplicity, it is assumed that M and N are equal and are powers of two (to facilitate bit-packing and bit-shifting operations). In lines 5-6, GPU-LMDDA handles the *resource request granted* event. The *Request_Granted* kernel launches a single block containing a single thread to perform the updates discussed in Section 4.3.3.

Lines 7-15 in Algorithm 12 handle the *resource request blocked* event. First, the *Find_Sink_Processes* kernel launches M blocks with *INT_PER_ROW* threads per block. Each block determines if a process is a sink node or not. After finding all sink nodes, GPU-LMDDA launches the *Deadlock_Check* kernel with a single block containing a single thread. This kernel determines if a deadlock exists in the system and alters the *deadlock_flag* accordingly. If a deadlock exists, it would be handled in line 12 of Algorithm 12 (out of the scope of this thesis). Otherwise, as the requester (say p_i) is blocked for the requested resource (say q_j), all processes reachable from the requested resource also become reachable from the resources on the requesting process' sub-tree. The *Update_RP_Blocked* kernel in line 14 handles this case. The kernel is launched with N blocks containing *INT_PER_ROW* threads per block to facilitate parallel computation. All kernels launched to handle the *resource request blocked* event are discussed in detail in Section 4.3.4.

Lastly, lines 16-37 handle the *resource release* event. In line 18, the *Release_Resource* kernel is launched with a single block containing a single thread. The kernel releases the resource from the AG matrix by either decrementing the weight of the grant edge

Algorithm 12 Overall Kernel Structure

```
// Assume TILE_DIM = 32 AND BLOCK_ROWS = 16
// grid variable → dim3 grid (M/TILE_DIM, N/TILE_DIM)
// threads variable → dim3 threads (TILE_DIM, BLOCK_ROWS)

if Resource Request Granted then
    Request_Granted $\lll 1, 1 \ggg$ (event, AG, RP)
else if Resource Request Blocked then
    Find_Sink_Processes $\lll M, INT\_PER\_ROW \ggg$ (SP, AR)
    Deadlock_Check $\lll 1, 1 \ggg$ (event, SP, RP, ProcessBitMask, deadlock_flag)
    Resource_Request_Blocked $\lll M, N \ggg$ (event, AR, SP, RP, ProcessBitMask, deadlock_flag)
    if deadlock_flag == true then
        Handle Deadlock
    else
        Update_RP_Blocked $\lll N, INT\_PER\_ROW \ggg$ (event, RP)
    end if
else if Resource Released then
    // Release the resource
    Release_Resource $\lll 1, 1 \ggg$ (event, AG)
    // Transpose the AR matrix for upcoming kernel
    bitMatrixTranspose $\lll TILE\_DIM, TILE\_DIM \ggg$ (AR_tile, AR)
    tileTranspose $\lll TILE\_DIM, TILE\_DIM \ggg$ (AR_trans, AR_tile)
    // Determine if processes are waiting on the released resource
    Find_Waiting_Processes $\lll 1, INT\_PER\_ROW \ggg$ (event, AR_trans, pwait_flag)
    // Determine the release type: if of type (ii) or (iii), perform reachability computation
    Determine_Release_Type $\lll 1, 1 \ggg$ (event, AG, AR, pwait_flag, reachability_flag)

    if reachability_flag == true then
        // Reset reachability_flag
        reachability_flag ← false
        // Initialize the RP matrix to AG - i.e. All grant edges
        InitRP_Reachability $\lll N, M \ggg$ (AG, RP)

        // Perform iterations of the reachability computation implementing the node hopping mechanism
        for $s = 1$; $2^{s-1} < min(m,n)$; $s{+}{+}$ do
            bitMatrixTranspose $\lll TILE\_DIM, TILE\_DIM \ggg$ (RP_tile, RP)
            tileTranspose $\lll TILE\_DIM, TILE\_DIM \ggg$ (RP_trans, RP_tile)
            Reachability_Computation $\lll N, M \ggg$ (RP, RP_temp, RP_trans, AR)
        end for
    end if
else
    Not a valid event
end if
```

or removing it altogether. Then a transpose of the AR matrix is computed and stored into the AR_trans matrix in lines 20-21 via the *bitMatrixTranspose* and *tileTranspose* kernels (the *bitMatrixTranspose* and *tileTranspose* kernels are further discussed in Section 4.3.6). This is done to facilitate coalesced global memory accesses in the upcoming kernel. The *Find_Waiting_Processes* kernel then launches one block containing *INT_PER_ROW* threads. This kernel reduces row q_j of the AR_trans matrix to determine if any processes are waiting on the released resource. If there is a process waiting, the *pwait_flag* is set to 1. Next, in line 25, the *Determine_Release_Type* kernel is launched with a single block containing a single thread. This kernel checks the resource release event type, and if necessary, removes the pending request edge $p_t \rightarrow q_j$ from the AR matrix and grants the resource q_j to process p_t by updating the AG matrix accordingly. If the *Determine_Release_Type* kernel finds that the resource release event is of release type (ii) or (iii), then the *reachability_flag* is set to true, indicating that reachability information in the system needs to be updated.

After returning to the CPU, it checks if the *reachability_flag* is set to true (line 27), and if so, the *reachability_flag* is reset (line 29) and the *InitRP_Reachability* kernel is launched (line 31). The *InitRP_Reachability* kernel is launched with N blocks containing M threads per block. This kernel initializes RP for the forthcoming Reachability Computation (RC). The CPU then begins iterations of RC. For each iteration, the *bitMatrixTranspose* and *tileTranspose* kernels (line 35-36) are called to build the transpose of RP, placing the result into the matrix RP_trans. This step is performed in order to guarantee coalesced global memory accesses in the upcoming kernel. Following the transpose is the launch of the *Reachability_Computation* kernel in line 37, which is launched with N blocks containing M threads per block. It is this kernel's responsibility to update all reachability information in the system by utilizing the node hopping mechanism seen in Section 4.2.2. Design details of the *resource release* kernels are discussed in Section 4.3.5.

4.3.3 Handling a Resource Request Granted Event

To handle the *resource request granted* event, GPU-LMDDA launches a kernel with a single block containing a single thread. Much like GPU-OSDDA, the *Resource_Request_Granted* kernel does not advocate parallelism since it launches a single thread, but is required for GPU-LMDDA to manage and maintain the RAG on the GPU. Algorithm 13 shows the pseudo-code for the *Resource_Request_Granted* kernel and is followed with an explanation of its statements.

Algorithm 13 Resource_Request_Granted⋘1,1⋙

```
1: // CUDA Indexing Variables
2: $row \leftarrow q_j \ll LIPR$
3: $col \leftarrow p_i \gg LINTBITS$
4: $bit \leftarrow p_i \% INTBITS$
5: $idx \leftarrow row + col$
6:
7: // One additional unit of $q_j$ is assigned to $p_i$
8: $AG[q_j \times M + p_i] = AG[q_j \times M + p_i] + 1$
9: // Process $p_i$ becomes reachable from resource $q_j$
10: $RP[idx] | = (1 \ll (INTBITS - (bit + 1)))$
```

In this kernel, both the RP[ ] and AG[ ] matrices must be updated to reflect the granted resource $q_j \rightarrow p_i$. Since the RP[ ] matrix is bit-packed and the AG[ ] matrix is not, two separate indices are needed to update the matrices. To facilitate understanding of the upcoming discussion, Figure 4.6 illustrates the index variable computations performed.

In lines 2-5, index variables are determined in order to update RP[ ]. In line 2, the proper row is calculated by multiplying q_j by LIPR (also denoted as *row* in Figure 4.6). Since GPU-LMDDA assumes that both M and N are powers of two, it is able to use bit-manipulation techniques to determine index variables. To determine the correct column to update, p_i is divided by LINTBITS in line 3 (the *col* variable is denoted as the blue set of boxes in Figure 4.6). Then, to calculate which bit within the final integer to alter, line 4 performs the modulo of p_i by INTBITS (denoted as the green highlighted portions of blue boxes in Figure 4.6). The final index is computed

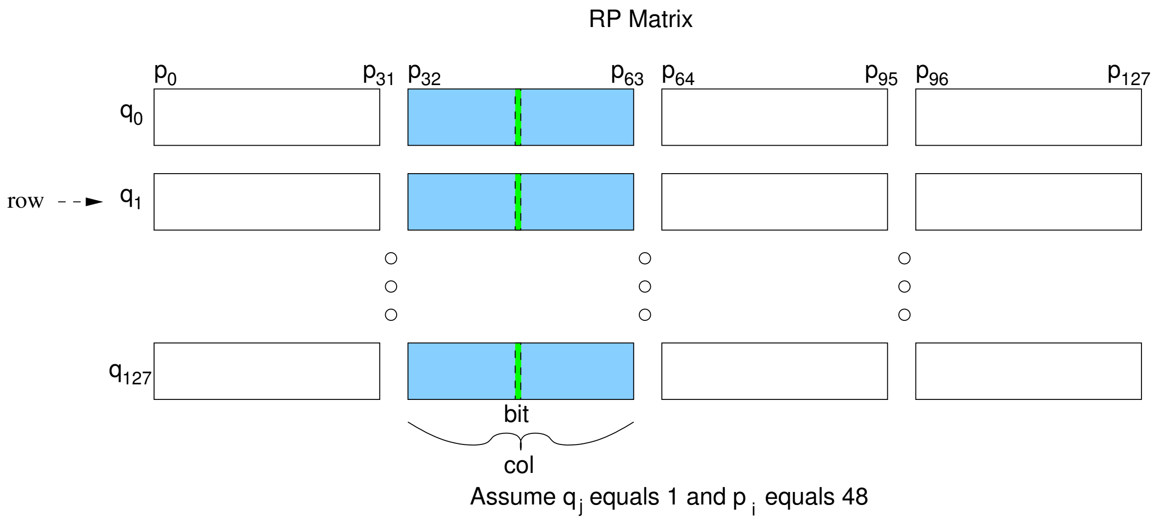


Figure 4.6: Illustration of computation to update RP Matrix (Resource Granted).

in line 5 by adding the row and column variables that were previously calculated. Line 8 handles the update of the AG[ ] matrix by assigning one (or one additional) unit of resource q_j to the requesting process p_i. Since AG[ ] is not bit-packed, the standard method of addressing a linearized matrix is used (i.e., *row* $\times$ *row_width* + *column*). As a result of the granted resource $q_j \rightarrow p_i$, p_i becomes reachable from q_j. To reflect this change, RP[*idx*] is ORed with a bit-mask that updates the correct bit within RP[ ] (line 10). This bit-mask technique is used frequently throughout GPU-LMDDA (much like GPU-OSSDA in Chapter 3) and is accomplished by simply shifting a bit into the desired position and performing a bit-wise operation on the desired data element, in this case, an OR operation so that the correct bit in RP[ ] will be set.

In summary, according to Xiao and Lee [12], the result of a *resource requested granted* event is that the requesting process p_i become reachable from the requested resource q_j. Since the resource request ($p_i \rightarrow q_j$) is granted, the AG[ ] matrix is updated to reflect the grant status ($q_j \rightarrow p_i$).

4.3.4 Handling a Resource Request Blocked Event

GPU-LMDDA handles a *resource request blocked* event in several stages: finding the sink process nodes, checking for deadlock, and updating reachability information if necessary. First, the *Find_Sink_Processes* kernel is launched to find the sink process nodes in the RAG.

Algorithm 14 Find_Sink_Processes≪M,INT_PER_ROW≫

```
1:  // Declare shared variable for reduction
2:  __shared__ $sAR[INT\_PER\_ROW]$
3:
4:  // CUDA Indexing Variables
5:  $bid \leftarrow blockIdx.x$
6:  $tid \leftarrow threadIdx.x$
7:  $idx \leftarrow bid \ll LIPR + tid$
8:  $spidx \leftarrow bid \gg LINTBITS$
9:  $bit \leftarrow bid\%INTBITS$
10:
11: // Read global AR data into shared memory
12: $sAR[tid] = AR[idx]$
13: __syncthreads__()
14:
15: // OR Reduction in shared memory to determine if processes are waiting on $q_j$
16: for $s = 1$; $s < blockDim.x$; $s = s \times 2$ do
17:     if $tid \;\%\; (2 \times s) == 0$ then
18:         $sAR[tid] \mid= sAR[tid + s]$
19:     end if
20:     __syncthreads()
21: end for
22:
23: // Write out compliment of shared data to SP to determine sink status
24: if $tid == 0$ then
25:     if $sAR[0] == 0$ then
26:         $atomicOr(SP[spidx], (unsigned\ \ int)(1 \ll (INTBITS - (bit + 1))))$
27:     end if
28: end if
```

The computations performed in this kernel are illustrated in Figure 4.7. First, a shared vector of length INT_PER_ROW is created in line 2 that holds each row of the AR[ ] matrix in shared memory (of each block).

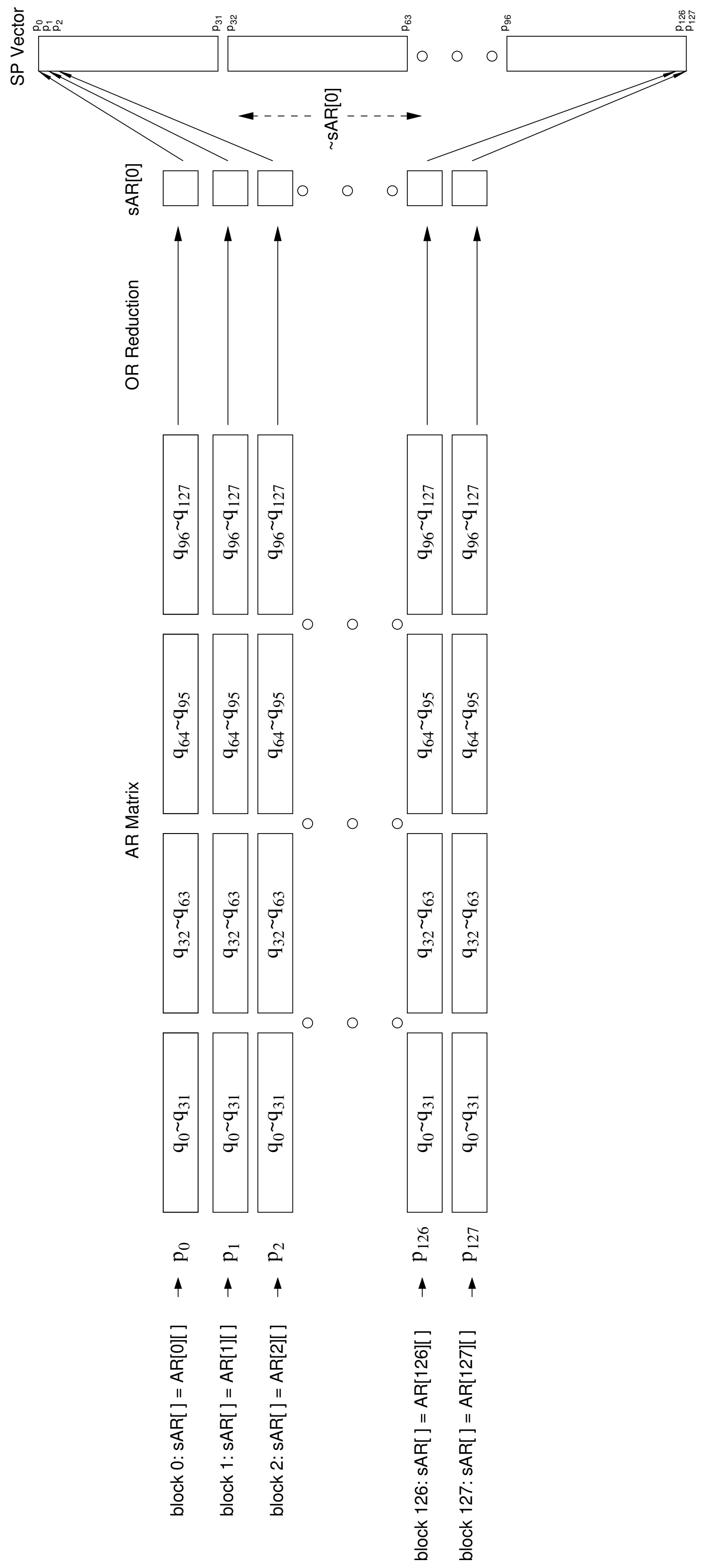


Figure 4.7: Illustration of finding sink process nodes.

The shared memory is used to hasten future reductions in the kernel. Next, indexing variables are determined in lines 5-9. Lines 5 and 6 holds the block index and thread index variables in *bid* and *tid*, respectively. Then, in line 7, a global integer index is determined by multiplying (actually a left shift) *bid* (synonymous with a row) by LIPR and adding to it *tid* (synonymous with a column). Next, an index is declared for each block by dividing (actually a right shift) *bid* by LINTBITS (line 8). This index is created so that each block may write to the correct integer in the SP[ ] vector. Finally, to find the correct bit in the integer to alter in the SP[ ] vector, the modulo of *bid* by INTBITS is performed in line 9.

In lines 12 and 13, the shared vector sAR[ ] is populated with the contents of each row of the AR[ ] matrix (i.e., the shared vector sAR[ ] in block *bid* would be populated with AR[*bid*][ ]) and a *__syncthreads()* is performed. Following the initialization phase of the kernel, a parallel OR reduction of the shared memory vector sAR[ ] is performed in lines 16-21. Starting in line 24, a single thread (thread 0) from each block takes control. If the reduction of the sAR[ ] vector is equal to 0 it indicates that process p_{bid} is a sink node. As a result, by using the *spidx* index and the *bit* variable determined earlier, a 1 is written to the associated bit in the SP[*bid*] element.
If the reduction of sAR[ ] is greater than 0, process p_{bid} is blocked and thus not a sink node; no change is made to the SP[*bid*] element.

After the *Find_Sink_Processes* kernel completes, all process sink nodes have been found for the current RAG. Then the *Deadlock_Check* kernel is invoked with a single block containing a single thread. The pseudo-code for this kernel is provided in Algorithm 15. This kernel first creates a register variable *likecols* in line 2 which will be used later in the kernel to keep track of vector comparisons. Next, index variables *qrow* and *prow* are created in lines 3-4. The *qrow* variable equals the resource q_j times LIPR, yielding the ${q_j}^{th}$ row in an $n \times m$ matrix. The *prow* variable on the other hand equals the process p_i times LIPR, which gives the ${p_i}^{th}$ row in an $m \times n$ matrix. In lines 7-11, a bit-wise AND of the SP[ ] and RP[*qrow*][ ] vectors is performed and compared with ProcessBitMask[*prow*][ ] vector (where *prow* is the bitmask for

Algorithm 15 Deadlock_Check⋘1,1⋙

```
// CUDA Indexing Variables
$likecols \leftarrow 0$
$qrow \leftarrow q_j \ll LIPR$
$prow \leftarrow p_i \ll LIPR$

// Check if $p_i$ is the only reachable sink of $q_j$
for $i = 0$; $i < INT\_PER\_ROW$; $i = i + 1$ do
    if $(SP[i]\ \&\ RP[qrow + i]) == ProcessBitMask[prow + i])$ then
        $likecols = likecols + 1$
    end if
end for

// If likecols is equal to INT_PER_ROW, this means that all columns of
// (SP[ ] & RP[j][ ]) == $p_i$'s bitmask and that a deadlock exists
if $likecols == INT\_PER\_ROW$ then
    $deadlock\_flag \leftarrow 1$
else
    $col \leftarrow q_j \gg LINTBITS$
    $bit \leftarrow q_j \% INTBITS$
    $idx \leftarrow prow + col$

    // Process $p_i$ is now blocked for requesting $q_j$
    $AR[idx] \mid= (1 \ll (INTBITS - (bit + 1)))$
end if
```

process p_{prow}, see Definition 1.2.36). The bit-wise AND result is compared with the ProcessBitMask[$prow$][ ] vector to determine if p_i is the only reachable sink node of q_j.

As illustrated in Figure 4.8, if the result of the bit-wise AND computation is equal to the value in ProcessBitMask[$prow$][ ], then we iterate the *likecols* variable. The loop iterates INT_PER_ROW times so that all elements of the vectors of interest have been compared. In line 15, a comparison of *likecols* and INT_PER_ROW is performed. If *likecols* is equal to INT_PER_ROW, this indicates that all integers of the bit-wise AND computations are equal to the corresponding integers in ProcessBitMask[ ]. As a result of *likcols* being equal to INT_PER_ROW, process p_i is a sink node of the requested resource q_j and a deadlock exists in the system.

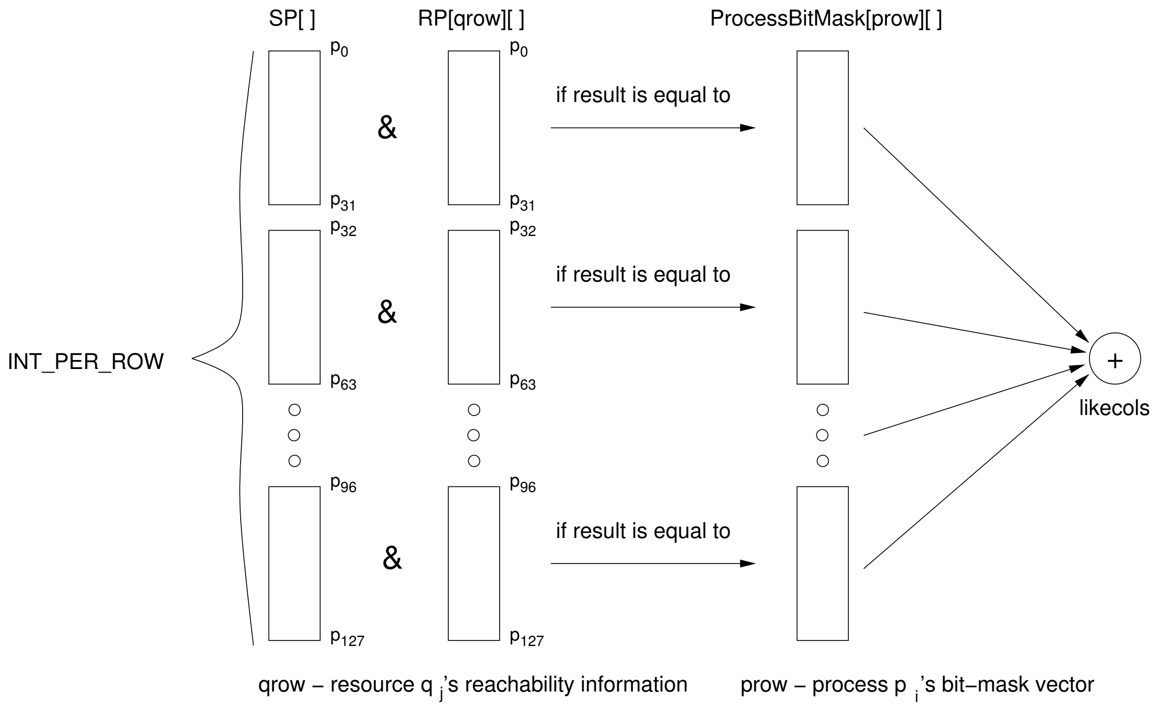


Figure 4.8: Illustration of computation to check for deadlock.

Line 16 then asserts the *deadlock_flag* variable and the kernel is complete. If *likecols* is not equal to INT_PER_ROW, then the requesting process p_i is not a sink node of the requested resource q_j. As a result, p_i needs to be blocked for requesting q_j. Lines 18-20 perform the index calculations to determine what bit to write in the AR[ ] matrix and AR[ ] is subsequently updated in line 23 to reflect the blocked request, $p_i \rightarrow q_j$.

Upon completion of the *Deadlock_Check* kernel, the CPU either handles the deadlock event or GPU-LMDDA continues its operation by updating resource reachability information by launching the *Update_RP_Blocked* kernel (see Figure 4.2). This kernel is invoked with N blocks containing INT_PER_ROW threads per block. What it does is update the reachability information (see Definition 1.2.32) of all resources following the *resource request blocked* event. Figure 4.9 provides an illustration of how each variable in the kernel references indices into the RP[ ] matrix (to be discussed next).

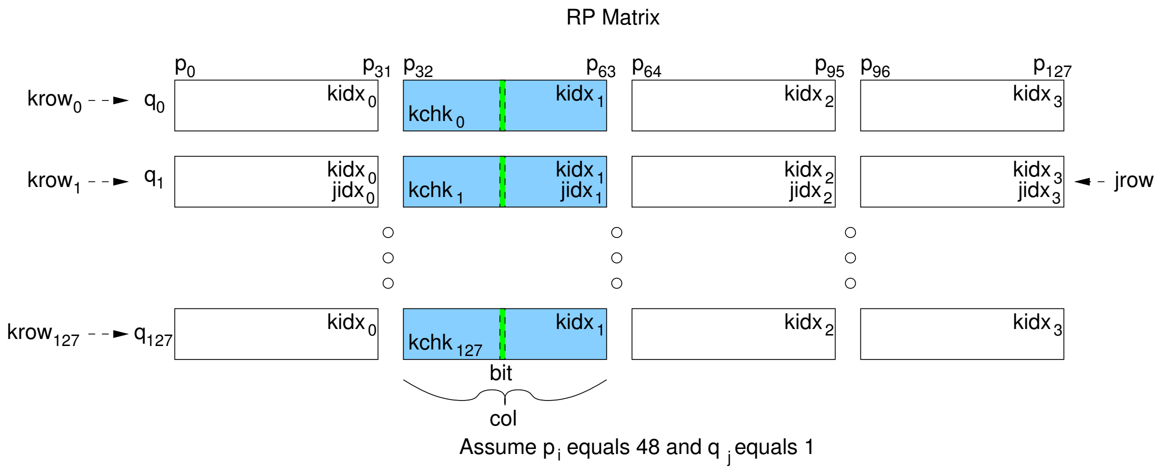

RP Matrix
p0 p31 p32 p63 p64 p95 p96 p127
krow0 q0 kidx0 kchk0 kidx1 kidx2 kidx3
krow1 q1 kidx0 jidx0 kchk1 kidx1 jidx1 kidx2 jidx2 kidx3 jidx3 jrow
krow127 q127 kidx0 kchk127 kidx1 kidx2 kidx3
bit
col
Assume pi equals 48 and qj equals 1

Figure 4.9: Illustration of computation to update RP Matrix (Resource Blocked).

Algorithm 16 Update_RP_Blocked⋘N,INT_PER_ROW⋙

```
1: // CUDA Indexing Variables
2: $col \leftarrow p_i \gg LINTBITS$
3: $bit \leftarrow p_i \% INTBITS$
4: $krow \leftarrow blockIdx.x \ll LIPR$   // Start location of each row
5: $kchk \leftarrow krow + col$   // Each int to be checked in each row
6: $kidx \leftarrow krow + threadIdx.x$   // All elements of such row (dest)
7: $jrow \leftarrow q_j \ll LIPR$   // Start location of $q_j$ row
8: $jidx \leftarrow jrow + threadIdx.x$   // All elements of $q_j$ row (source)
9:
10: // Now all processes reachable from $q_j$ become also
11: // reachable from the resources on $p_i$'s sub-tree
12: if $(RP[kchk]\ \&\ (INTBITS - (bit + 1)) == 1)$ then
13:     $RP[kidx] \mid= RP[jidx]$
14: end if
```

To start, the kernel computes all indexing variables in lines 2-8. In this kernel, each block works on a separate row in the RP[ ] matrix. In line 2, the *col* variable is declared which yields the integer in which process p_i falls. This is done by dividing (actually a right shift) p_i by LINTBITS (result is the blue column in Figure 4.9 under listed assumptions). Then in line 3, the *bit* variable is computed to determine which bit to alter in the integer index (result is the green highlighted portion of the blue box in Figure 4.9). Following a similar process as done previously to find the bit of interest, the modulo of process p_i by INTBITS is performed. Next, we define the *krow* variable in line 4. The *krow* is assigned the row that each block handles by multiplying (actually a left shift) *blockIdx.x* by LIPR.

In the upcoming computation, one bit of one column of every row in RP[ ] needs to be checked to find if process p_i is reachable from any resource. Looking at Figure 4.9, it is assumed that the process of interest is p_{48}, which helps derive the column and bit of RP[ ] to check. The column (denoted as *col* in Figure 4.9) and bit checked under this scenario are shown. To perform this check in the kernel, we continue to create the *kchk* variable in line 5, which is the sum of the *krow* and *col* variables. This *kchk* variable combined with the *bit* variable already determined will allow for all bits of interest (bit column shown in Figure 4.9) of RP[ ] to be checked. As we discussed earlier, if an element of RP[ ] (say RP[k][i]) is 1, then those resources denoted by q_k are reachable to p_i. As a result, we must make all processes reachable from q_j also reachable from the resources on p_i's sub-tree. To do so, we first calculate the variable *jrow* in line 7 by multiplying q_j by LIPR. Following in line 8, the global integer index *jidx* is determined by adding *jrow* and *threadIdx.x*. Figure 4.9 depicts the indices used after creating the *jrow* and *jidx* variables, assuming that q_j is equal to q_1 in the figure. Then in line 12, we check a specific bit (see the *bit* variable in Figure 4.9) within every RP[*kchk*] integer for equality with 1. If it is equal to 1, then that indicates that resource q_{bid} is reachable to process p_i. Then, all processes reachable from q_j become also reachable from the resources (such as q_{bid}) on p_i's sub-tree. Line 13 performs this update in parallel. Since the RP[ ] matrix is bit-packed, each statement (RP[*kidx*] |= RP[*jidx*]) updates 32 bits at a time. Upon completing the update in reachability, the *resource request blocked* event has been completely handled.

Provided here is a summary of the computations involved when GPU-LMDDA handles a *resource request blocked* event. According to [12], the first action taken after the request is to determine if the event would cause the system to enter a deadlock state. This is done first by finding all *sink processes* in the system. To do so, GPU-LMDDA performs an OR reduction on all rows of the AR[ ] matrix. If a row (say p_h) is reduced to 0 (or the negation of the reduction is equal to 1) then process p_h is a *sink process*. All *sink nodes* found are stored in the Sink[ ] vector. Then, the Sink[ ] vector is bit-wise ANDed with the reachable process vector (RP[j][ ]) to determine if the result is equal to p_i's bit-mask. If the result of the computation is equal to p_i's bit-mask, a deadlock occurs in the system. Otherwise, the request is blocked ($p_i \rightarrow q_j$) by updating the AR[ ] matrix to reflect this event (see Figure 4.2). Finally, resource reachability is updated. All processes reachable from q_j become also reachable from the resources on p_i's sub-tree [12]. To update the RP[ ] matrix, in parallel for all $1 \leq k \leq N$, RP[k][ ] is bit-wise ORed with RP[j][ ] and stored in the associated RP[k][ ] memory location.

4.3.5 Handling a Resource Release Event

GPU-LMDDA handles the *resource release* event in several phases. The first phase is the releasing of a unit of resource q_j which may be followed by an update phase of the reachability information of the system if the event is of type release (ii) or (iii) (see Figure 4.3).
To begin, GPU-LMDDA launches the *Release_Resource* kernel whose pseudo-code is provided in Algorithm 17. This kernel is launched with a single block containing a single thread.

Algorithm 17 Release_Resource⋘1,1⋙

```
1: // CUDA Indexing Variables
2: $idx \leftarrow q_j \times M + p_i$
3:
4: // One unit of $q_j$ is released by $p_i$
5: $AG[idx] = AG[idx] - 1$
```

Looking at Algorithm 17, line 2 calculates the global index to alter in the AG[ ] matrix to reflect the release of resource q_j by process p_i. This global index is stored in the *idx* variable and is found by multiplying the row q_j by the length of the row

(M) and adding the column offset (p_i). Next, the resource is released in line 5 by decrementing AG[idx]. This will result in either: i) the weight of the current grant edge being reduced by 1 or ii) the grant edge being removed entirely (i.e., a weight of 0).

Algorithm 18 Find_Waiting_Processes≪1,INT_PER_ROW≫

```
1:  // Declare shared variable for reduction
2:  $sAR[INT\_PER\_ROW]$
3:
4:  // CUDA Indexing Variables
5:  $tid \leftarrow threadIdx.x$
6:
7:  // Populate shared data with data from AR_trans
8:  $sAR[tid] = AR\_trans[q_j \times INT\_PER\_ROW + tid]$
9:  $\_\_syncthreads()$
10:
11: // Perform reduction of row $q_j$ of AR_trans matrix
12: for $s = 1$; $s < blockDim.x$; $s = s \times 2$ do
13:     if $tid \;\%\; (2 \times s) == 0$ then
14:         $sAR[tid] \;|= sAR[tid + s]$
15:     end if
16:     $\_\_syncthreads()$
17: end for
18:
19: // If sAR is not all zeros, then a process is waiting for resource $q_j$
20: if $tid == 0$ then
21:     $sAR[0]\ ?\ pwait\_flag = 1 : pwait\_flag = 0$
22: end if
```

After removing a unit of the released resource from the AG[ ] matrix, GPU-LMDDA must determine if any other processes are waiting on the released resource q_j. To do so, the *Find_Waiting_Processes* kernel is invoked with a single block containing INT_PER_ROW threads. The pseudo-code for this kernel can be found in Algorithm 18 with an illustration of its computations found in Figure 4.10.

To begin the *Find_Waiting_Processes* kernel, a shared vector sAR[ ] is declared of size INT_PER_ROW in line 2 (as done in Algorithm 14). Then in line 5, a variable is created to hold the value contained in *threadIdx.x* named *tid*. Lines 8-9 then populate

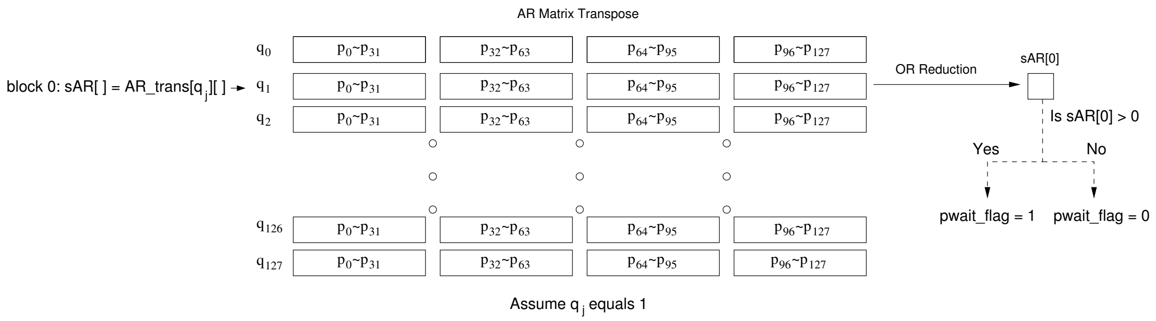


Figure 4.10: Illustration of computation to find waiting processes.

the sAR[ ] vector with the values in the ${q_j}^{th}$ row of the AR_trans[ ] matrix followed by the *__syncthreads()* intrinsic operation. Using shared memory to reduce the ${q_j}^{th}$ row of AR_trans[ ] allows for quicker access to memory, thus making the reduction faster.

In lines 12-17, an OR reduction of the sAR[ ] vector is performed. After the reduction is finished, a single thread (thread 0) takes control, denoted by the conditional check in line 20. If the reduced value of sAR[ ] is not equal to 0, this implies that a process is waiting on resource q_j and the *pwait_flag* is set to 1. If the reduction of sAR[ ] is equal to 0, it implies that no processes are waiting for resource q_j and the *pwait_flag* is set to 0. Upon completion of this kernel, GPU-LMDDA has determined whether or not any processes are waiting on the released resource q_j.

After finding if any processes are waiting for the release of resource q_j, GPU-LMDDA determines whether the event was of release scenario (ii) or (iii) using the *Determine_Release_Type* kernel. This kernel is invoked with a single block containing a single thread. The pseudo-code is provided in Algorithm 19. In Algorithm 19, lines 2-5 determine the indexing variables used in the kernel's calculations. Line 2 calculates the ${p_t}^{th}$ row (p_t is the process that has a potential outstanding request of resource q_j) by multiplying (actually a left shift) p_t by LIPR and storing the result in *row*. In line 3, the integer that represents the ${q_j}^{th}$ column is found by dividing (actually a right shift) q_j by LINTBITS and stored in *col*. Then in line 4, the bit

Algorithm 19 Determine_Release_Type⋘1,1⋙

```
1:  // CUDA Indexing Variables
2:  $row \leftarrow p_t \ll LIPR$
3:  $col \leftarrow q_j \gg LINTBITS$
4:  $bit \leftarrow q_j \% INTBITS$
5:  $idx \leftarrow row + col$
6:
7:  // Check the event type:
8:  // If pwait_flag == 1 - release type (ii)
9:  // If AG[ ] == 0 - release type (iii)
10: if $pwait\_flag == 1 \,||\, AG[q_j \times M + p_i] == 0$ then
11:     $reachability\_flag \leftarrow 1$
12:     // If the resource is reassigned - update matrices
13:     if $q_j$ is reassigned to $p_t$ then
14:         // The request edge $p_t \rightarrow q_j$ is removed
15:         $AR[idx] \;\&= \;\sim(1 \ll (INTBITS - (bit + 1)))$
16:         // A unit of $q_j$ is assigned to $p_t$
17:         $AG[q_j \times M + p_t] = AG[q_j \times M + p_t] + 1$
18:     end if
19: end if
```

which needs to be altered is found by performing the modulo of q_j by INTBITS. A global index *idx* is then calculated by adding the *row* and *col* variables together.

Next, line 10 checks if either there is a process (known as p_t) waiting on the released resource q_j or if the released resource's grant edge has been completely removed from the RAG. If either of these conditions is true, it indicates that the release event is of type (ii) or (iii), respectively (see Figure 4.3). In this case, reachability information of the RAG needs to be updated and thus the *reachability_flag* is set to 1 in line 9. In the event that the resource q_j was reassigned to a process p_t (denoted by the conditional check in line 13), line 15 removes the pending request edge from the AR[ ] matrix by performing a bit-wise AND with the appropriate bit-mask. Then in line 17, the resource q_j is granted to process p_t by incrementing the associated element in the AG[ ] matrix.

When the CPU regains control after the *Determine_Release_Type* kernel completes, it checks the status of the *reachability_flag*. If the *reachability_flag* has been asserted, then a series of calculations, known in their entirety as the Reachability Computation (RC), begins. Otherwise, GPU-LMDDA is finished handling the *resource release* event. Here we assume that reachability information needs to be updated and that the *reachability_flag* has been asserted. In this case, the *InitRP_Reachability* kernel is invoked with N blocks containing M threads per block. The pseudo-code for this kernel is provided in Algorithm 20.

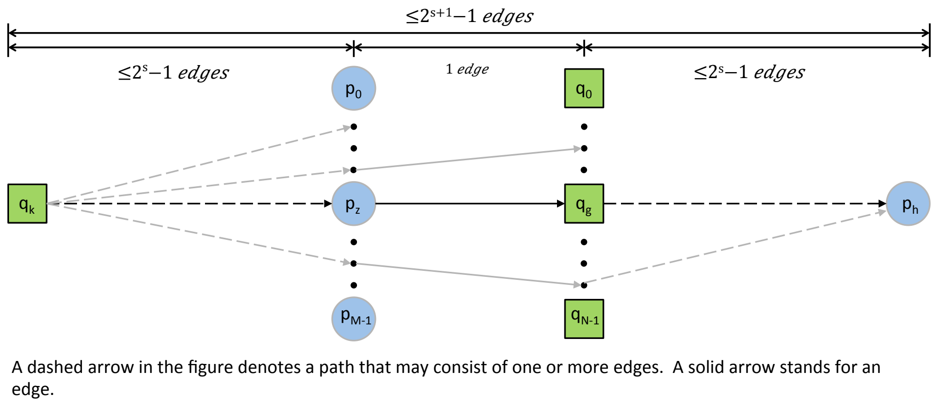
Algorithm 20 InitRP_Reachability$\lll$N,M$\ggg$

```
// CUDA Indexing Variables
$bid \leftarrow blockIdx.x$
$tid \leftarrow threadIdx.x$
$nidx \leftarrow bid \times blockDim.x + tid$
$row \leftarrow bid \ll LIPR$
$col \leftarrow tid \gg LINTBITS$
$bit \leftarrow tid\%INTBITS$
$idx \leftarrow row + col$

// If $AG[q_j][p_i] >= 1$, it means that process $p_i$ is reachable from $q_j$
if $AG[nidx] >= 1$ then
    $RP[idx] \mid= (1 \ll (INTBITS - (bit + 1)))$
end if
```

To prepare the RP[ ] matrix for RC, Algorithm 20 performs the initialization step of the node hopping mechanism (see Section 4.2.2). The algorithm starts by calculating all index variables in lines 2-8. Each matrix row of interest is handled per block in this kernel. Line 2 assigns the value of *blockIdx.x* to the variable *bid.* Then each element per row (of AG) is handled by a thread. Line 3 assigns the value of *threadIdx.x* to the variable *tid.* To build the global index into the AG[ ] matrix (since it is not bit-packed), *bid* (indicating a row) is multiplied by the width of the row (denoted by *blockDim.x*) and added by the thread offset of *tid.* To successfully update the RP[ ] matrix, indices are needed that correctly address the bit-packed matrix as well. Line 5 calculates the row used for addressing RP[ ] and is computed by multiplying *bid* by LIPR and storing the result in *row.* The correct integer to alter in RP[ ] is determined by storing the result of *tid* divided by LINTBITS in the *col* variable (line 6) (see Figure 4.6). To find the bit within the integer to update, line 7 creates the variable *bit* and assigns it the modulo of *tid* by INTBITS. The global bit-packed index into the RP[ ] matrix is then found by creating the *idx* variable in line 8. This global index is calculated by adding the *row* and *col* variables. After finding all index variables, line 11 checks every element in the AG[ ] matrix using the global index *nidx*. If the value at the checked element in AG[ ] is greater than or equal to 1 (indicating a grant edge $q_{bid} \rightarrow p_{tid}$ exists) then that same (resource, process) pair is updated in the RP[ ] matrix. Line 12 performs the update of the RP[ ] matrix if necessary by performing a bit-wise OR computation using the appropriate bit-mask built with the *bit* variable. Now the initialization phase of the node hopping mechanism has been completed.

Next, the CPU initiates iterations of RC. For this computation, GPU-LMDDA calculates whether if each process (say p_h) is reachable (see Definition 1.2.32) from each resource (say q_k), where $1 \leq h \leq M$ and $1 \leq k \leq N$, respectively. This computation follows the methodology described by the node hopping mechanism in Section 4.2.2. Figure 4.11 illustrates the computation of a single RP[k][h] value for a single iteration, and such a computation needs to be performed for all resource-process pairs over many iterations.



≤2^{s+1}−1 edges
≤2^s−1 edges
1 edge
≤2^s−1 edges
p0
q0
qk
pz
qg
ph
pM-1
qN-1
A dashed arrow in the figure denotes a path that may consist of one or more edges. A solid arrow stands for an edge.

Figure 4.11: Illustration of RC for RP[k][h] for a single iteration (Courtesy of Xiao [12]).

Algorithm 21 Reachability_Computation⋘N,M⋙

```
// CUDA Indexing Variables
$k \leftarrow blockIdx.x$
$h \leftarrow threadIdx.x$
$hcol \leftarrow h \gg LINTBITS$
$hbit \leftarrow h\%INTBITS$

// Vector elements indicate whether connections
// have been made between $p_z \rightarrow p_h$
$innertemp[M/INTBITS] = \{0\}$

// By implementing the node hopping mechanism, try to connect any
// request edge in AR to the reachability edge $q_g \rightarrow p_h$
for $z = 0$; $z < M$; $z = z + 1$ do
    for $g = 0$; $g < INT\_PER\_ROW$; $g = g + 1$ do
        if $(AR[z \ll LIPR + g]\ \&\ RP\_trans[h \ll LIPR + g]) > 0$ then
            // Represent the connection of AR[z][g] to $p_h$ with innertemp[z]
            $innertemp[z \gg LINTBITS] \mid= (1 \ll (INTBITS - ((z\%INTBITS) + 1)))$
            break
        end if
    end for
end for

// Now attempt to connect all reachability edges $q_k \rightarrow p_z$ using the RP matrix and innertemp[ ]
for $z = 0$; $z < INT\_PER\_ROW$; $z = z + 1$ do
    if $(RP[k \ll LIPR + z]\ \&\ innertemp[z]) > 0$ then
        $atomicOR(RP\_temp[k \ll LIPR + hcol], (unsigned\ int)(1 \ll (INTBITS - (hbit + 1))))$
        break
    end if
end for
```

Prior to each iteration of RC, GPU-LMDDA computes the transpose of the current RP[ ] matrix and stores the result in the RP_trans[ ] matrix, which facilitates global memory coalescing in the upcoming kernel. The *bitMatrixTranspose* kernel is further discussed in Section 4.3.6. After performing the matrix transpose of RP[ ], the *Resource_Reachability* kernel is invoked with N blocks containing M threads per block. For this kernel, every thread (across all blocks) handles its own reachability computation for each RP[k][h] index. Algorithm 21 provides the pseudo-code for the *Resource_Reachability* kernel.

To start Algorithm 21, index variables k and h are created in lines 2-3 which contain the *blockIdx.x* and *threadIdx.x* variables, respectively. Lines 4-5 then determine indexing variables that allow us to address the bit-packed matrices later in the kernel. Line 4 creates the *hcol* variable that is the quotient of h divided by LINTBITS. The bit which would need to be altered is calculated in line 5 by performing the modulo of h by INTBITS and storing it in the *hbit* variable. Then, line 9 creates the vector *innertemp* of length M/INTBITS, which will be used to keep track of any connection made between a particular process p_z and p_h, where $0 \leq z \leq M-1$ (see Figure 4.11), while performing calculations of the node hopping mechanism. Starting in line 13, the kernel begins computation that attempts to connect edges in AR[z][g] (where $0 \leq z \leq M-1$ and $0 \leq g \leq N-1$) to edges in the current RP[g][h] (where $0 \leq g \leq N-1$) matrix index. Looking at Figure 4.11, this step consists of connecting edges $p_z \rightarrow q_g$ (in AR[ ]) to $q_g \rightarrow p_h$ (in the current RP[ ] matrix). The for-loops iterate over all combinations of $0 \leq z \leq M-1$ and $0 \leq g \leq N-1$ as seen in Figure 4.11. To determine if a connection can be made from p_z to p_h (where $0 \leq z \leq M-1$), line 15 performs the bit-wise AND of every AR[z][g] element with every current RP[g][h] (denoted by RP_trans[h][g]) element. If a connection between p_z in AR[ ] to p_h in the current RP[ ] matrix exists, the value pertaining to that z element in the *innertemp* vector is set to 1 (line 17). The *innertemp* value is set utilizing the familiar bit-wise OR with a bit-mask as performed in prior kernels. The resulting *innertemp* vector now signifies those processes connected to p_h. Figure 4.12 provides an illustration of the computations performed to attempt connecting all z processes (p_z, where $0 \leq$ z $\leq M-1$) to process p_h (see Figure 4.11). Each row (z) of AR undergoes AND computations with row p_h of RP_trans, the result is used to set the z^{th} element of *innertemp*.

After performing computations that try to connect any edge between p_z and p_h, the kernel then tries to connect the resource q_k to one of those processes found and stored in the *innertemp* vector. Figure 4.13 illustrates the computations performed to attempt connecting resource q_k to one of the processes contained in *innertemp*

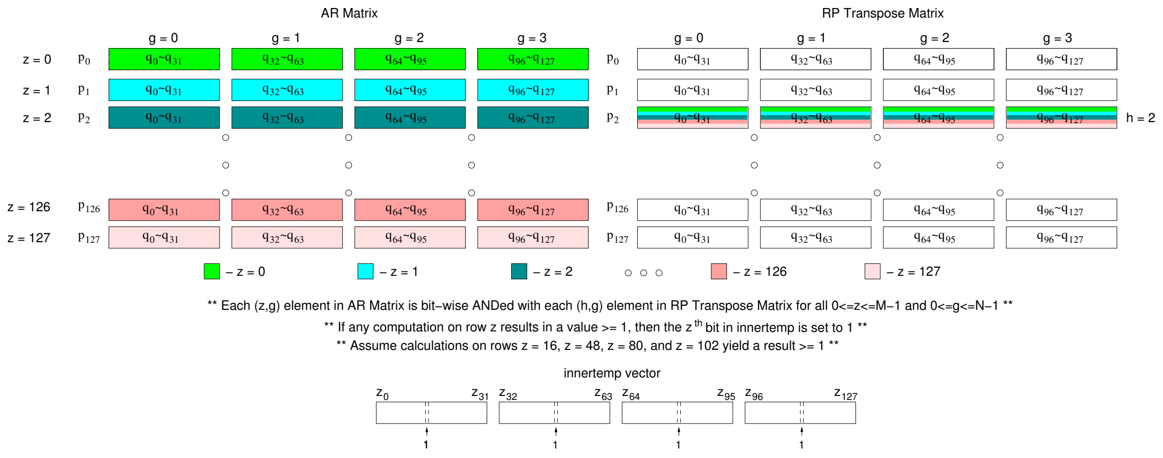

AR Matrix
RP Transpose Matrix
g = 0
g = 1
g = 2
g = 3
z = 0
z = 1
z = 2
z = 126
z = 127
h = 2
– z = 0
– z = 1
– z = 2
– z = 126
– z = 127
** Each (z,g) element in AR Matrix is bit–wise ANDed with each (h,g) element in RP Transpose Matrix for all 0<=z<=M–1 and 0<=g<=N–1 **
** If any computation on row z results in a value >= 1, then the z th bit in innertemp is set to 1 **
** Assume calculations on rows z = 16, z = 48, z = 80, and z = 102 yield a result >= 1 **
innertemp vector

Figure 4.12: Illustration of connecting process z to process h in RC.

where row q_k of the current RP[ ] matrix is bit-wise ANDed with innertemp, and the results are reduced to determine if q_k is reachable to p_h. In line 24, the for-loop starts iterating over all p_z elements in vector *innertemp* and attempts to connect them with q_k using the current RP[ ] matrix. For each iteration of z, the kernel effectively performs 32 comparisons at once since the matrices are bit-packed. If the result of the bit-wise AND operation in line 25 is greater than 0, a connection has been made between resource q_k and process p_h. To signify the connection has been made, an atomic OR operation is performed on the RP_temp[ ] matrix to set the bit pertaining to the connection between different RP[k][h] values (see node hopping mechanism in Section 4.2.2). Since multiple threads may attempt to write to the same RP_temp[ ] integer, the atomic operation is needed to avoid race conditions. Similarly, to avoid race conditions, the results of each iteration of RC are stored into the RP_temp[ ] matrix which holds new values of RP[ ]. Upon completing each iteration of RC, the results held in RP_temp[ ] are copied back into the RP[ ] matrix and GPU-LMDDA continues the next iteration of RC. This final computation will be able to determine if a connection exists between resource q_k and process p_h at the end of the current iteration of RC (i.e., $q_k \rightarrow \ldots \rightarrow p_z \rightarrow \ldots \rightarrow p_h$). Multiple iterations of RC are required to fully determine if some q_k is reachable to some p_h, as each iteration of RC adds edges that are 1,3,7,15,31... edges away per iteration (see Figure 4.4). The maximum number of iterations of RC completed by GPU-LMDDA is $\lceil log_2(min(m,n))\rceil$ as defined by the node hopping mechanism in Section 4.2.2.

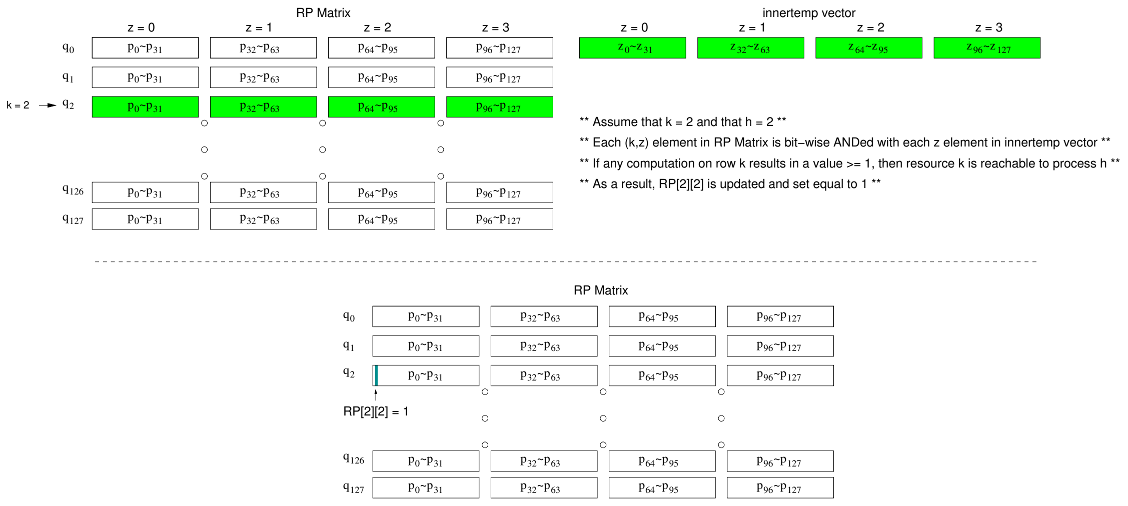

RP Matrix
z = 0
z = 1
z = 2
z = 3
q_0
q_1
k = 2
q_2
q_{126}
q_{127}
p_0~p_{31}
p_{32}~p_{63}
p_{64}~p_{95}
p_{96}~p_{127}
innertemp vector
z_0~z_{31}
z_{32}~z_{63}
z_{64}~z_{95}
z_{96}~z_{127}
** Assume that k = 2 and that h = 2 **
** Each (k,z) element in RP Matrix is bit–wise ANDed with each z element in innertemp vector **
** If any computation on row k results in a value >= 1, then resource k is reachable to process h **
** As a result, RP[2][2] is updated and set equal to 1 **
RP[2][2] = 1

Figure 4.13: Illustration of connecting resource k to process h in RC.

To summarize, GPU-LMDDA starts handling the *resource release* event by removing the released resource from the AG[ ] matrix. Then, GPU-LMDDA needs to determine what type of release event occurred (i, ii, or iii) (see Figure 4.3). First, the column vector of AR[ ][j] undergoes an OR reduction. Then, the result of the OR reduction is compared with 0, and if the result is not equal to 0 or if the grant edge AG[j][i] has been completely removed then the event belongs to release type (ii) or (iii). If one of these two conditions does not occur, the RAG does not change and computation is complete.

Here we assume that the release event is of type (ii) or (iii) and reachability information needs to be updated. If the released resource is reassigned to a waiting process (say p_t), then the request edge ($p_t \rightarrow q_j$) is removed from the AR[ ] matrix and the grant edge ($q_j \rightarrow p_t$) is added (or incremented) in the AG[ ] matrix. The next step is initializing the RP[ ] matrix for computations of RC. To do so, the RP[k][h] elements (where $1 \leq k \leq N$ and $1 \leq h \leq M$) are initialized to 1 if AG[k][h] is greater than or equal to 1. Otherwise, the RP[k][h] elements contain a 0. Next, computations of RC are completed as described in Section 4.2.2.

4.3.6 Supplementary Kernels

In addition to the core GPU-LMDDA kernels discussed previously, there are two additional kernels (or techniques) that GPU-LMDDA uses in its computations. One of which is the *bitMatrixTranspose* and *tileTranspose* kernels. These kernels are used prior to the *Find_Waiting_Processes* kernel and during the iterations of RC to enable coalesced global memory accesses in the forthcoming kernels. Since the matrices to be transposed were bit-packed, the transpose becomes more complex than a standard matrix transpose. It was found that a bit-matrix transpose of 32×32 elements had been performed before by Warren [22] (see Figure 4.14). Like GPU-OSDDA, Warren's approach to the bit-matrix transpose was implemented in parallel to handle blocks of 32×32 elements per thread followed by a tile transpose to effectively transpose the entire matrix (see Figure 4.15). This allowed for a fast and efficient bit-matrix transpose function that facilitated parallelism and global memory coalescing in computations of GPU-LMDDA.

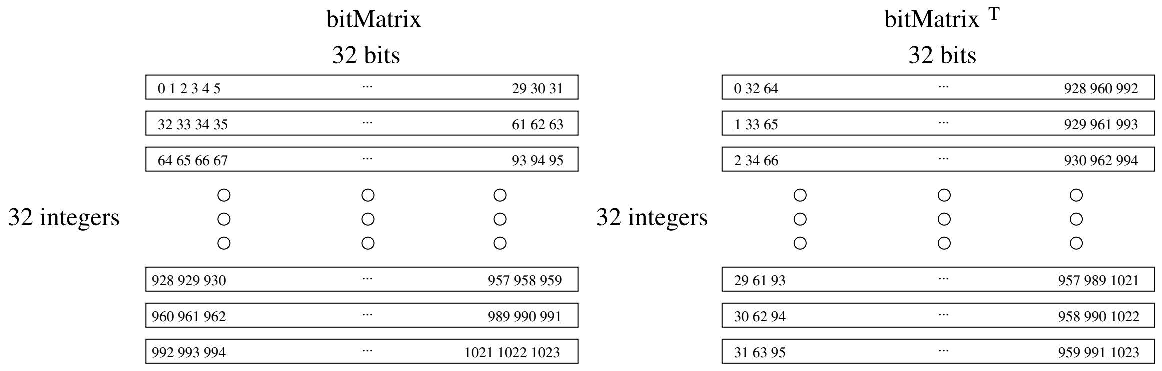


Figure 4.14: Example of a bit-matrix transpose of a 32×32 matrix.

The other kernel (or technique) used in GPU-LMDDA's computations is the reduction. Throughout GPU-LMDDA, the reduction was not performed in its own

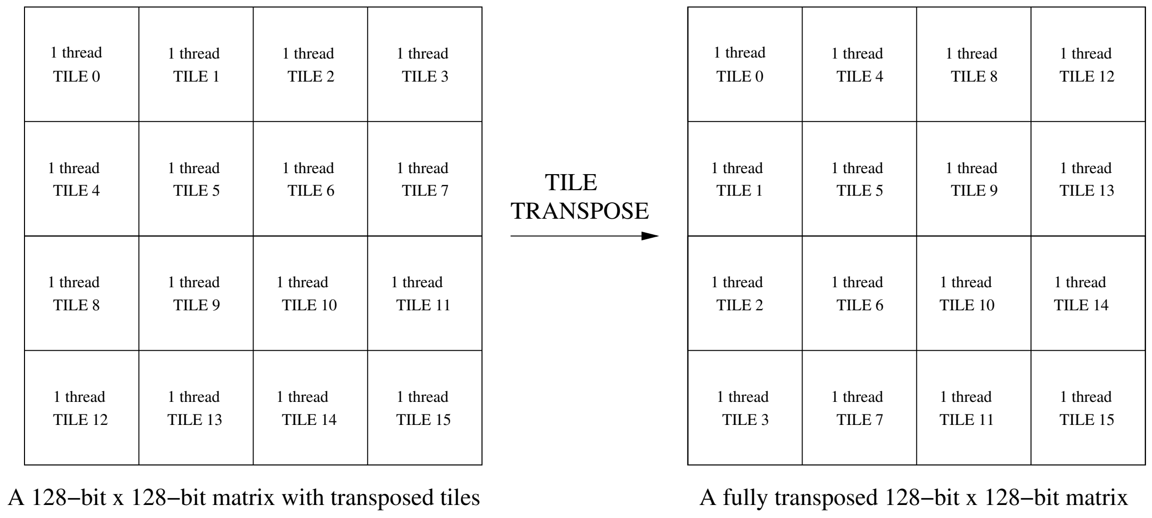


Figure 4.15: Example of full matrix transpose after bit-matrix transpose in tiles.

device function, but directly in the kernel. GPU-LMDDA also had several types of reductions (OR, AND, ADD); regardless, the approach to the reduction is the same. Parallel reduction is also a common operation in parallel programming, so the reduction techniques used in this algorithm were adopted from [23].

4.4 Experimentation and Results

A serial version of GPU-LMDDA is first implemented using the C language referred to as CPU-LMDDA. This version, like CPU-OSDDA, utilizes no parallel programming primitives (such as OpenMP) in order to fully demonstrate the capabilities of parallel processing on GPU. The development of a parallel CPU version of our algorithm would provide significant difficulty as the CPU is limited in the number of threads it may spawn to accomplish the massive parallel computations required. In addition, the CPU should be available to handle the normal stream of tasks associated with computing, as mentioned in Section 1.4. By launching the maximum number of threads (or utilizing vectorized instructions) on the CPU, we limit the capability of the processor to perform processing of normal computing tasks.

All experiments were performed on an Intel ® Core i7 CPU @ 2.8 GHz with 12 GB RAM. The CUDA GPU-LMDDA implementation was tested on three different GPUs: GTX 670, Tesla C2050, and Tesla K20c. The GTX 670 has 7 SMXs (1344 CUDA Cores) with 2 GB Global Memory, the Tesla C2050 has 14 SMs (448 CUDA Cores) with 3 GB Global Memory, and the Tesla K20c has 13 SMXs (2496 CUDA Cores) with 5 GB Global Memory.

To verify the correctness of our algorithm, both CPU-LMDDA and GPU-LMDDA were tested against a small RAG (approximately 8 processes $\times$ 8 resources). Once results were verified as correct, larger RAGs were generated to test the scalability of GPU-LMDDA for larger set sizes. Table 4.3 shows the run-time of CPU-LMDDA versus our initial character approach to GPU-LMDDA. Table 4.4 follows and shows the run-time of CPU-LMDDA versus the final bit-vector implementation of GPU-LMDDA. The speedups associated with our results are then depicted in Figure 4.16 on each piece of target hardware. As can be seen by our results, an increase in process and resource counts dramatically increases CPU-LMDDA's run-time. However, GPU-LMDDA's run-time provides substantial speedups in comparison with the run-time of CPU-LMDDA. The gradual reduction of achieved speedups is due to limitations of available thread blocks when the number of processes and resource increases. A further discussion will be provided in the Future Work section (Section 6.2) of this thesis.

Table 4.3: Run-Time/Speedup of CPU-LMDDA and GPU-LMDDA (Initial)

Input	CPU-LMDDA	GTX 670	Tesla C2050	Tesla K20c
64×64	0.10/0x	0.001/100.00x	0.001/100.00x	0.001/100.00x
128×128	1.80/0x	0.04/45.15x	0.04/44.00x	0.03/59.00x
256×256	32.54/0x	0.57/56.09x	0.70/45.49x	0.50/64.08x
512×512	389.03/0x	9.38/40.47x	12.30/30.63x	8.50/44.77x
1024×1024	6688.14/0x	162.99/40.03x	213.60/30.31x	149.80/43.65x

Table 4.4: Run-Time/Speedup of CPU-LMDDA and GPU-LMDDA (Bit-Packed)

Input	CPU-LMDDA	GTX 670	Tesla C2050	Tesla K20c
64×64	0.10/0x	0.001/100.00x	0.0001/100.00x	0.0001/100.00x
128×128	1.80/0x	0.005/359.00x	0.003/599.00x	0.003/599.00x
256×256	32.54/0x	0.15/215.93x	0.04/812.50x	0.12/270.17x
512×512	389.03/0x	2.78/138.94x	1.22/317.88x	2.18/177.45x
1024×1024	6688.14/0x	93.17/70.78x	175.23/37.17x	77.55/85.24x

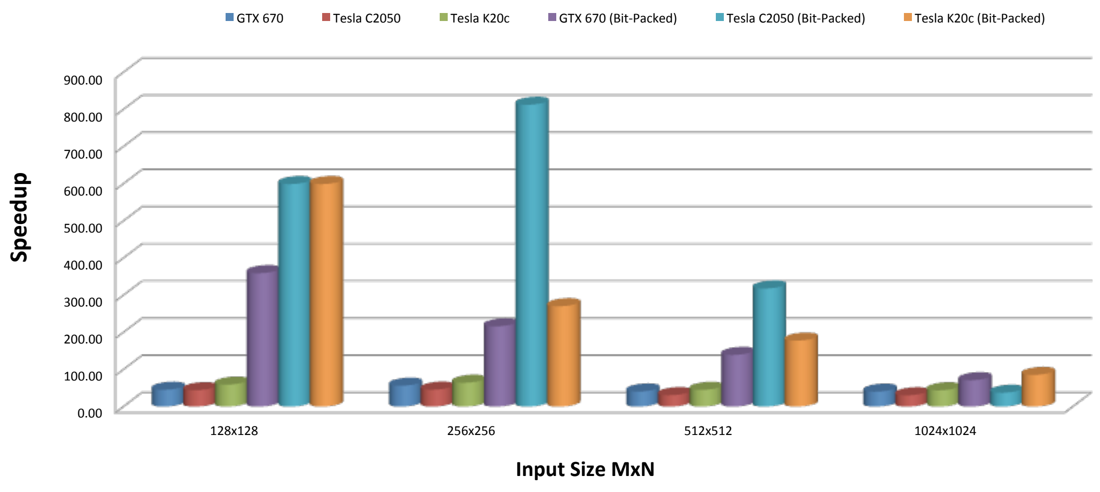


Figure 4.16: GPU-LMDDA Speedup

4.5 Conclusion

A new approach to deadlock detection for multi-unit systems on GPU has been devised and developed using CUDA C. By finding a clever GPU-based solution to the node hopping mechanism, GPU-LMDDA was able to achieve substantial speedups over a CPU-based implementation. The use of bit-vectors for storing matrices led to a greatly reduced memory footprint and efficient algorithmic computations. Both of these factors alone drastically affected the speedups realized and allow GPU-LMDDA to handle a large number of processes and resources.

5 GPU-PBA: A GPU-BASED DEADLOCK AVOIDANCE ALGORITHM

5.1 Introduction

This chapter discusses GPU-PBA, a GPU-based approach to deadlock avoidance in systems with multi-resources containing multiple-instances. Section 5.2 states the system assumptions and provides background information regarding the core methodology of GPU-PBA, which is rooted in the Parallel Bankers Algorithm (PBA). The section then discusses the underlying theory of PBA by revealing how it handles *resource request* and *resource release* events. Since all computations are done in parallel, the best case run-time of PBA is $\mathcal{O}(1)$ as opposed to $\mathcal{O}(N \times M^2)$ of [2] and $\mathcal{O}(N \times M)$ of [3]. It also discusses the H-Safety check which performs the task of determining if an H-Safe sequence exists (system is in an H-Safe state) upon a *resource request* event. The H-Safety check and the H-Safe sequence are terms based on Habermann's approach to deadlock avoidance (see Definitions 1.2.19 and 1.2.18), hence the H (for Habermann) preceding the terms.

After performing the H-Safety check it would be known if the *resource request* is able to be granted and still allow the system to avoid the deadlock condition. This is followed by our algorithm design and its implementation in Section 5.3. Following the design discussion is the Experimentation and Results Section (Section 5.4) which details the run-times and speedups achieved by GPU-PBA.

5.2 Background

5.2.1 Assumptions and Terms

The goal of GPU-PBA is to determine if an H-Safe sequence (see Definition 1.2.18) exists based upon a *resource request* event. If an H-Safe sequence exists, then the system is deemed to be in an H-Safe state (see Definition 1.2.19). The assumptions listed here are required in order to determine if the H-Safe sequence exists:

1. The maximum claim of each resource from each process must be declared in advance [14].

2. There are a fixed number of resources in the system [14].

Additionally, we provide two additional notes regarding the H-Safe sequence.

1. Any H-Safe sequence can be proven never to evolve into deadlock [2].

2. There are no promises about any timing properties such as periods and worst-case execution times of processes [14].

It was noted in the introduction that GPU-PBA handles multi-unit resources containing multiple instances or units. By this, it is meant that processes may request/release multiple resources as well as multiple units of each resource per each resource request/release event. The multiple instance (or multi-unit) resource is defined in Definition 1.2.14.

5.2.2 Underlying Theory of PBA

The PBA algorithm is meant to execute in the event of any *resource request* or *resource release* event. Upon a *resource request* event, PBA determines that the system is "safe" if the *resource request* is granted, i.e., if an H-Safe sequence exists that leads to the system being in an H-Safe state.

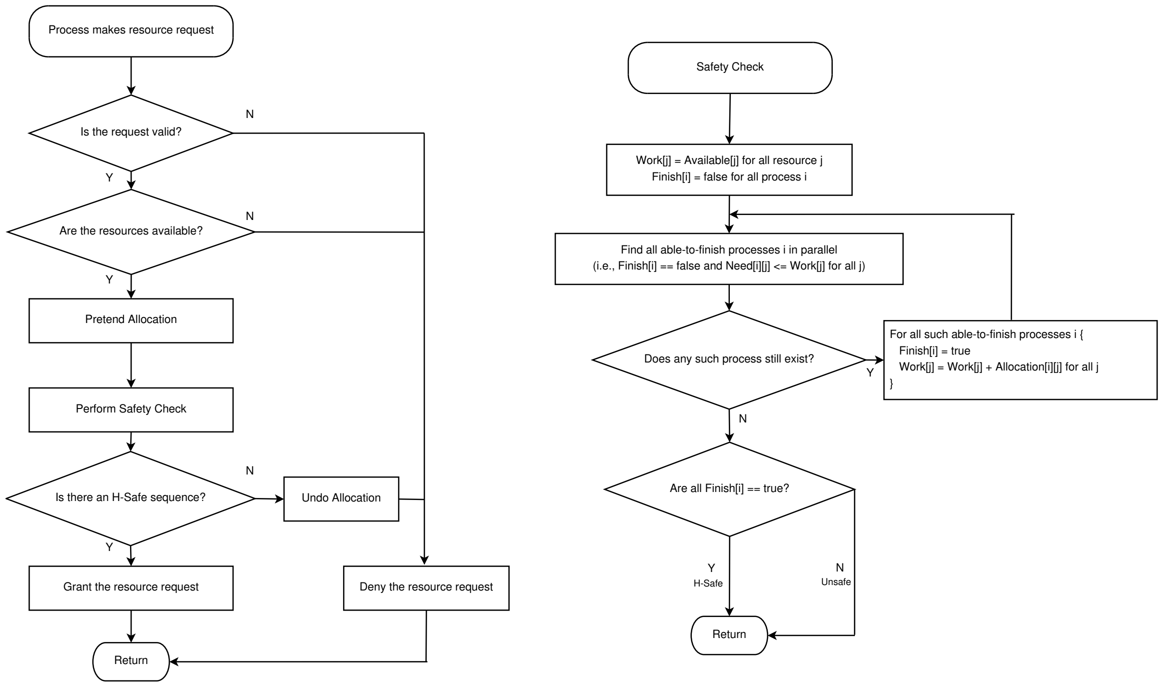
Table 5.1: Data Structures for GPU-PBA

Structure Name	Explanation
request[i][j]	request from process i for resource j
maximum[i][j]	maximum possible demand of process i for resource j
available[j]	current number of unused instances of resource j
allocation[i][j]	process i's current allocation of j
need[i][j]	process i's potential for more j (need[i][j]=maximum[i][j]-allocation[i][j])
work[j]	a temporary storage (vector) for available[j]
finish[i]	potential completeness of process i

Prior to the explanation of the underlying theory of PBA, the algorithm's data structures are introduced in Table 5.1. Table 5.1 provides a list of the data structures utilized in PBA with an explanation of each [14]. It should be noted for the rest of the design explanation of PBA and GPU-PBA that i refers to a process and j refers to a resource.

5.2.3 Computations of PBA and the H-Safety Check

Prior to starting computations of PBA, the algorithm should take as input the maximum resource claim of each process, which is represented by the maximum[i][j] matrix described in Table 5.1. After the system is initialized, PBA may begin handling *resource request* and *resource release* events. The flow chart detailing PBA's behavior in a *resource request* scenario is shown in Figure 5.1. No flow chart is provided for a *resource release* scenario, as the actions taken under this scenario are trivial. The following sub-sections detail each phase of computation from the *resource request* and *resource release* events.



Process makes resource request
Is the request valid?
N
Y
Are the resources available?
N
Y
Pretend Allocation
Perform Safety Check
Is there an H-Safe sequence?
N
Undo Allocation
Y
Grant the resource request
Deny the resource request
Return
Safety Check
Work[j] = Available[j] for all resource j
Finish[i] = false for all process i
Find all able-to-finish processes i in parallel
(i.e., Finish[i] == false and Need[i][j] <= Work[j] for all j)
Does any such process still exist?
Y
For all such able-to-finish processes i {
Finish[i] = true
Work[j] = Work[j] + Allocation[i][j] for all j
}
N
Are all Finish[i] == true?
Y
H-Safe
N
Unsafe
Return

Figure 5.1: PBA Process Flowchart

Resource Request

Computations of PBA are initiated upon a *resource request* event. The first computation PBA performs is called the validity check. For this computation, the number of resources requested by a particular process (say p_i) is compared with its pertaining vector in the need matrix (i.e., need[p_i][ ]). If the number of resources requested for each resource q_j is less than or equal to the process' current number of needed resources, the request is valid. If one or many of the amounts of resources requested is greater than its associated value in the need[ ] matrix, then the request is denied and the next event may be handled.

Assuming that the request made in the prior step is valid, PBA continues to check if the resources requested are available to satisfy the request. To do so, PBA checks the amounts of requested resources against the resource values contained in the available[ ] matrix.

If the number of resources requested is less than or equal to all values in the available[ ] matrix, then resources are available to satisfy the resource request. If there are not enough resources (of one or many of the requested resources) then the request is denied.

If all resources are available to satisfy the request, then PBA temporary allocates the resources to the requesting process and initializes the work[ ] matrix with the values in the available[ ] matrix in preparation for the H-Safety check. In the H-Safety check (see Figure 5.1), the finish[ ] matrix is initialized to false for all values of p_i. It then begins iterations of the safety check to determine if processes are able to finish. To do so, the need[ ] matrix is compared with the work[ ] matrix for all processes (i.e., need$[p_i][j] \leq$ work$[j]$). If for all j, this inequality is satisfied, then the process p_i is able to finish. If the process is able to finish, then the finish$[p_i]$ value is set to true and the process' allocated resources are added back to the work[ ] matrix. Iterations continue until either 1) all processes are able to finish which implies an H-Safe sequence exists, or 2) no more processes are able to finish. If the H-Safety check determines that an H-Safe sequence exists, the resource request is granted. If there is no H-Safe sequence, the temporary allocations are revoked and the resource request is denied.

Resource Release

The *resource release* event is easily handled by PBA. In parallel, PBA adds the released resources back to the availability pool (i.e., available$[j]$ += releasedResources$[j]$). The process that released these resources (say p_i) needs the resources added back to its need pool, indicating that it may require those resources again to execute its task (need$[p_i][j]$ += releasedResources$[j]$). This is followed by removing the released resources from the allocation pool (i.e., allocation$[p_i][j]$ -= releasedResources$[j]$) so that the resources may be claimed by other requesting processes.

5.3 GPU-PBA Design

5.3.1 Introduction

This design section describes, in detail, the GPU-PBA approach to the Parallel Bankers Algorithm. The design that GPU-PBA employs allows for simultaneous computations to be done concurrently, ranging from the validity and resource availability checks to the computations in the H-Safety check. Section 5.3.2 dicusses how GPU-PBA handles the *resource request* event and breaks the overall process down into multiple phases. Each subsection of Section 5.3.2 provides the pseudo-code of each kernel that GPU-PBA uses to handle this event type. Then in Section 5.3.3, GPU-PBA's handling of the *resource release* event is discussed. Pseudo-code for the resource release kernel is also provided and detailed.

5.3.2 Handling a Resource Request

GPU-PBA splits the handling of the *resource request* event into 5 phases: 1) Valid Request Check, 2) Resource Availability Check, 3) Preparation for the H-Safety Check, 4) Computations of the H-Safety Check, and 5) Handling the results of the H-Safety Check. Provided in Algorithm 22 is the overall kernel structure that GPU-PBA follows for a *resource request* event. The H-Safety check kernel structure is detailed separately in Algorithm 23. Make note of two things: 1) there are several computations that are handled outside of GPU kernels for the H-Safety check due to the speed advantage that the CPU has over the GPU for that particular computation and 2) the finish[ ] and atf[ ] vector elements are all initialized to 0 prior to running the H-Safety check.

In Algorithm 22, line 4 performs a check to determine if the resource request that was made is valid by launching the *Check_Valid_Request* kernel. This kernel launches a single block containing J threads (i.e., the number of resources in the system). If this kernel returns with the *cont* variable equal to true, then the resource request

Algorithm 22 GPU-PBA Overall Kernel Structure for Resource Request Events

```
1:  // Check resource event type
2:  if Resource Request then
3:      // Check if the resource request is valid
4:      Check_Valid_Request⋘1,J⋙(event, Need, cont)
5:      if cont == true then
6:          // Check if resources are available to satisfy request
7:          Check_Resource_Availability⋘1,J⋙(event, Available, Wait_Count, cont)
8:          if cont == true then
9:              // Temporarily Allocate the Requested Resources
10:             Temporary_Allocate_Request⋘1,J⋙(event, Available, Allocation, Maximum, Need, Work)
11:             // Declare variable to hold number of processes that are able to finish
12:             total_atf ← 0
13:
14:             // Set finish[ ] and atf[ ] vectors to 0 and Perform Safety Check - see Algorithm 23
15:             The H-Safety Check
16:
17:             // If the request is H-Safe - Continue
18:             // Otherwise, rollback the temporary allocations and deny request
19:             if total_atf != I then
20:                 RollBack_Allocations⋘1,J⋙(event, Available, Allocation, Maximum, Need, Wait_Count)
21:             end if
22:         end if
23:     end if
24: end if
```

is valid. Then in line 7, the *Check_Resource_Availability* kernel is invoked with a single block containing J resources. This kernel checks that all requested resources are available, if so, the GPU-PBA continues to temporarily allocate the requested resources. Otherwise, the resource request is denied.

Assuming the resource request was valid and the requested resources are available, the *Temporary_Allocate_Request* kernel is invoked in line 10. This kernel is launched with a single block containing J threads. The work[ ] matrix is also initialized in this kernel and will be detailed in sub-section *Preparing for the H-Safety Check*. Next, a temporary variable *total_atf* is created that is used to keep track of the number of processes that are able to finish. Then, starting in line 15, the H-Safety Check is performed (see Algorithm 23) to attempt to find an H-Safe sequence.

Algorithm 23 Safety Check Overall Kernel Structure

```
1:  // The H-Safety Check
2:  // Maximum I iterations can occur where only 1 process is
3:  // “able_to_finish” per iteration, if an H-Safe sequence exists
4:  for $i = 0; i < I; i++$ do
5:      Perform_Safety_Check⋘I,J⋙(Need, Work, Finish, atf)
6:
7:      // Copy the atf vector to CPU for sequential scan
8:      for $i = 0; i < I; i++$ do
9:          if $atf[i] == 1$ then
10:             $total\_atf = total\_atf + 1$
11:             // Add resources of able to finish processes to Work[ ]
12:             Add_Finished_To_Work⋘1,J⋙(i, Work, Allocation)
13:         end if
14:     end for
15:
16:     // If all processes are able to finish - break the for loop
17:     if $total\_atf == I$ then
18:         break;
19:     end if
20:     // Set all elements of the atf[ ] vector to 0
21:     $atf[] \leftarrow 0$
22: end for
```

The safety check will perform a maximum number of I iterations to determine if a safe sequence exists. The reason there is a potential of I iterations is due to the fact that a scenario may exist where, per iteration, a single process is deemed “able_to_finish”. If this is the case, and an H-Safe sequence exists, it would take the maximum number of iterations, or I iterations. If no safe sequence is found before or on the I^{th} iteration, then no safe sequence exists for a particular resource request event. In Algorithm 23, line 4 initiates the iterations of the safety check, with the maximum number of iterations set to I. Line 5 then launches the *Perform_Safety_Check* kernel to handle safety check computation in parallel. Make note of the arguments in the *Perform_Safety_Check* kernel call, in particular the *finish* and *atf* arguments. The *finish* argument refers to the *finish* vector mentioned in Section 5.2.2. The *atf* argument is a temporary *finish* vector that tracks the processes that are able to finish per each iteration of the safety check. Between each iteration, the *atf* vector is reset to 0. The *Perform_Safety_Check* kernel is invoked with I blocks (each block performs computations for each process) containing J threads per block (perform computations on each resource) in order to maximize parallelism for this computation and perform all comparisons of the safety check in parallel. Inside the kernel, the *atf* vector is updated to show which processes are able to finish per iteration as previously mentioned. After the completion of the *Perform_Safety_Check* kernel, the *atf* vector is copied back to the CPU and sequentially scanned to find all processes that are able to finish (see lines 8-14). In line 9, if a process (p_i) is able to finish, then the *total_atf* variable is incremented by 1 and p_i's allocated resources are added back to the work pool. Note that more than one process may be able to finish per iteration of the H-Safety check. Line 12 launches the *Add_Finished_To_Work* kernel with a single block containing J threads. After all processes that were able to finish add their allocated resources back to the work pool, then the total number of able to finish process is checked in line 17. If *total_atf* is equal to *I*, this indicates that all processes were able to finish and that an H-Safe sequence exists. The safety check computation then terminates via the *break* statement. Otherwise, the next iteration of the safety check is initiated to find additional able to finish processes.

Looking back at Algorithm 22, if the H-Safety Check completes and there exists an H-Safe sequence, then the variable *total_atf* will be equal to I and all temporary allocations anchor. On the other hand, if no H-Safe sequence exists (the check performed on *total_atf* in line 19 is true), then all temporary allocations are rolled back by the *RollBack_Allocations* kernel in line 20. This kernel is invoked with one block containing J threads. As a result of no H-safe sequence existing, the resource request is denied. Now that the overall kernel structure of GPU-PBA has been covered, the following sub-sections will detail all kernels launched for each phase of a *resource request* event scenario.

Valid Request Check

The first phase of handling the *resource request* event is the valid request check. GPU-PBA launches the *Check_Valid_Request* kernel to handle this validity check. The pseudo-code of this kernel is provided in Algorithm 24.

Algorithm 24 Check_Valid_Request⋘1,J⋙

```
// Declare shared memory vector used to check
// validity of all resource requests by a process
__shared__ sArrCheck[J]

// CUDA Indexing Variables - Shorthand notation for threadIdx.x
$tid \leftarrow threadIdx.x$

// Initialize shared memory to 0 and synchronize
// to prepare for resource request validity check
$sArrCheck[tid] \leftarrow 0$
__syncthreads()

// Perform a check to determine if request is valid
if $process\ i's\ request\ for\ resource\ j \leq Need[p_i \times J + tid]$ then
    $sArrCheck[tid] \leftarrow 1$
end if

// Reduce sArrCheck[ ]
// If sum is equal to J, then $p_i$'s request for all resources are valid
for $s = 1$; $s < blockDim.x$; $s = s \times 2$ do
    $index \leftarrow 2 \times s \times tid$
    if $index < blockDim.x$ then
        $sArrCheck[index] += sArrCheck[index + s]$
    end if
    __syncthreads()
end for

// Have thread 0 write-back the validity status to cont variable
if $tid == 0$ then
    if $sArrCheck[0] == J$ then
        $cont \leftarrow 1$
    else
        $cont \leftarrow 0$
    end if
end if
```

To start Algorithm 24, a shared vector *sArrCheck* of length J is declared in line 3 that will be used in the kernel to check the validity of all resource requests by the process (say p_i). Line 6 follows by declaring the *tid* variable which is assigned all *threadIdx.x* variables. Then in lines 10-11, the shared memory vector *sArrCheck* is initialized to 0 and all threads synchronize on the *__syncthreads()* intrinsic function. The *sArrCheck* vector is initialized to 0 because in the upcoming conditional check, if the process' request for resource q_{tid} is valid, then *sArrCheck*[*tid*] is set to 1. This usage will be made apparent in the discussion that follows.

Starting in line 14, each resource (q_j) amount requested by process p_i is compared against p_i's current need of each resource in the Need[p_i][q_j]. If the request amount is less than or equal to the current need[ ] amount for every resource q_j, then the *sArrCheck*[q_j] value is updated to contain a 1 in line 15. After all resources have been checked in parallel, then the *sArrCheck* vector is reduced in order to determine if all resource amounts requested were valid. Lines 20-26 perform the Add reduction by summing all values in the *sArrCheck* vector into the *sArrCheck*[0] location. Line 29 then forces a single thread (with threadIdx.x equal to 0) write the validity status of the request to the *cont* variable. If the result of the Add reduction was equal to J (line 30), then all resource amounts requested were valid and the *cont* variable is written with a 1 in line 31. On the other hand, if the reduction did not yield the value J, the request was invalid and the *cont* variable is written with a 0 in line 33. After returning to the CPU, GPU-PBA determines if the *resource request* event should continue to the next phase in computation by checking the status of the *cont* variable. If *cont* equals 1 (or *true*) then GPU-PBA advances to the *Check Resource Availability* phase. Otherwise, the resource request is denied.

Checking Resource Availability

Assuming the validity check of the resource request was valid (and the *cont* variable was 1 or *true*), then GPU-PBA initiates the next phase of computation which checks resource availability for the request. For this phase of computation, the *Check_Resource_Availability* kernel is invoked. The pseudo-code for this kernel is provided in Algorithm 25.

Algorithm 25 Check_Resource_Availability⋘1,J⋙

```
// Declare shared memory vector used to check
// availability of all resource requests by a process
__shared__ sArrCheck[J]

// CUDA Indexing Variables - Shorthand notation for threadIdx.x
$tid \leftarrow threadIdx.x$

// Initialize shared memory to 0 and synchronize
// to prepare for resource availability check
$sArrCheck[tid] \leftarrow 0$
__syncthreads()

// Perform check to determine if resources are available
if $process\ i's\ request\ for\ resource\ j \leq available[tid]$ then
    $sArrCheck[tid] \leftarrow 1$
end if

// Reduce sArrCheck[ ]
// If sum is equal to J, then all required resources are available
for $s = 1; s < blockDim.x; s = s \times 2$ do
    $index \leftarrow 2 \times s \times tid$
    if $index < blockDim.x$ then
        $sArrCheck[index] += sArrCheck[index + s]$
    end if
    __syncthreads()
end for

// Have thread 0 write-back whether or not resources are available
if $tid == 0$ then
    if $sArrCheck[0] == J$ then
        $cont \leftarrow 1$
    else
        $cont \leftarrow 0$
    end if
end if
```

To start the *Check_Resource_Availability* kernel, a shared vector *sArrCheck* is created of length J in line 3. This vector serves a similar purpose to the shared vector in the *Check_Valid_Request* kernel, except in this kernel it is used to check the availability of all requested resources by a process (say p_i). Line 6 follows by creating the *tid* variable, which holds the thread identifier (*threadIdx.x*) for each thread. Lines 10-11 then initialize the shared vector *sArrCheck* elements to 0 and synchronize all threads by the *__syncthreads()* function. Similar to the *Check_Valid_Request* kernel, if the request for a particular resource q_{tid} can be satisfied, then *sArrCheck*[*tid*] is set to 1. This usage will also be made apparent in the upcoming discussion.

Starting in line 14, the kernel checks the amount of each resource (q_{tid}) requested against the available units of that variable in the available[q_{tid}] matrix. If the requested amount of resource q_{tid} is less than or equal to the available amount, then the *sArrCheck[tid]* element is set to 1 in line 15. Following this availability check, lines 20-26 perform an Add reduction on the *sArrCheck* vector and store the result in *sArrCheck[0]*. Line 29 then forces the thread with *threadIdx.x* equal to 0 to take over and write the availability check status back to the *cont* variable. If the reduction of *sArrCheck* was equal to J, then this implies that all requested resources are available and the *resource request* event is still valid. As a result, the *cont* variable is set to 1 (or true) in line 31. If the reduced vector value is not equal to J, it implies that one or more resources requested were not available and then *cont* variable is set to 0 in line 33. After returning to the CPU, GPU-PBA checks the *cont* variable. If the *cont* variable is equal to 1 (or true), GPU-PBA continues to the next phase of computation: *Preparing for the H-Safety Check*. Otherwise, the resource request is denied.

Preparing for the H-Safety Check

If the *resource request* event is valid and all resources are available, then GPU-PBA needs to prepare for the H-Safety check. To do so, the algorithm temporarily grants all requested resources to the requesting process and initializes the work[ ] matrix. The *Temporary_Allocate_Request* kernel fulfills this function and the pseudo-code is provided in Algorithm 26.

Algorithm 26 Temporary_Allocate_Request⋘1,J⋙

```
1:  // Declare shared memory vector
2:  // used to hold resource request amounts
3:  __shared__ sRequest[J]
4:
5:  // CUDA Index Variables - Shorthand for threadIdx.x and Global Index
6:  $tid \leftarrow threadIdx.x$
7:  $idx \leftarrow p_i \times J + tid$
8:
9:  // Populate shared vector with requested resource amounts and synchronize
10: $sRequest[tid] \leftarrow requestedResources[tid]$
11: __syncthreads()
12:
13: // Pretend to allocate the requested resources
14: $available[tid] = available[tid] - sRequest[tid]$
15: $allocation[idx] = allocation[idx] + sRequest[tid]$
16: $need[idx] = maximum[idx] - allocation[idx]$
17: __syncthreads()
18:
19: // Initialize the Work array
20: $work[tid] = available[tid]$
```

To start the *Temporary_Allocate_Request* kernel, in Line 3 a shared memory vector is created of length J called *sRequest* that is used to temporarily hold the amounts of all resources requested by a process (say p_i). Lines 6-7 generate all indexing variables used throughout the kernel. Line 6 follows by assigning the *tid* variable, the identifier found in *threadIdx.x*. Then, line 7 creates the *idx* variable which is used to address the 2-D matrices in this kernel (i.e., allocation[ ], need[ ], and maximum[ ]). The value assigned to *idx* is comprised of the product of the process identifier and the width of the matrix row (i.e., J) and added to this is the thread identifier variable *tid*. Lines 10-11 initialize the *sRequest* vector to contain the amounts of each requested resource by process p_i (i.e., $sRequest[q_j]$ = requested amount of resource q_j) and then synchronize all threads on the *__syncthreads()* function.

Starting line 14, the kernel begins the temporary allocation process. Line 14 removes the requested resource amounts of each resource from the availability pool. Line 15 adds the requested resource amounts to the allocation[ ] vector for the requesting process. Then, in line 16, the requesting process' need[ ] vector is updated by subtracting its current number of allocated resources from the process' maximum claim for each resource. After these updates are completed, all threads synchronize on the *syncthreads()* function in line 17. Last, the work[ ] vector is initialized with the values in the available[ ] vector in preparation for the upcoming safety check.

Performing the H-Safety Check

When the CPU initiates iterations of the safety check, the *Perform_Safety_Check* kernel is invoked. The pseudo-code for this kernel is provided in Algorithm 27. In this kernel, every block performs computations for a particular process ($p_{blockIdx.x}$). Each thread in a block will perform computations involving a particular resource that a process needs or currently holds ($q_{threadIdx.x}$). Thus, all processes and resources are checked in parallel.

To start the *Perform_Safety_Check* kernel, line 3 declares a shared vector *resource_check* of length J that holds the status of each resource checked between a process' need[ ] and work[ ] vectors. Lines 7-8 then create the variables *tid* and *bid* and initialize them with the *threadIdx.x* and *blockIdx.x* identifiers, respectively. Lines 11-12 follow by initializing the *resource_check* vector with 1's and synchronizing all threads at that point. In line 14, the *finish* vector is checked to determine if any process p_{bid} has been deemed able to finish. If it has, the block pertaining to that process goes idle and its computation is complete. However, on this first iteration, at least, no processes have yet been deemed able to finish. Therefore, in line 16, every process' resources in the need[ ] matrix are checked against values in the work[ ] matrix. If the process' need[ ] value for any resource is greater than the value in work[ ], then the process will not be able to finish in this iteration, so the

Algorithm 27 Perform_Safety_Check⋘I,J⋙(need, work, finish, atf)

```
// Declare shared memory vector to hold status of
// check between a process' need[ ] and work[ ] vectors
__shared__ resource_check[J]

// CUDA Index Variables
// Each thread handles a process' resources and each block handles a process
$tid = threadIdx.x$
$bid = blockIdx.x$

// Initialize shared vector to contain all 1's and synchronize
$resource\_check[tid] = 1$
__syncthreads()

// If process denoted by bid has already finished, skip
// Otherwise, determine if the process may finish
if $finish[bid] == 0$ then
    // Check if the process is able to finish
    if $need[bid \times J + tid] > work[tid]$ then
        $resource\_check[tid] \leftarrow 0$
    end if
    __syncthreads()

    // Reduce resource_check[ ]
    // If sum is equal to J, then process can finish
    for $s = blockDim.x/2$; $s > 0$; $s \gg= 1$ do
        if $tid < s$ then
            $resource\_check[tid] += resource\_check[tid + s]$
        end if
        __syncthreads()
    end for

    // Each block - or process - checks to see if it was able to finish
    // If so, assert its completion status in finish[ ] vector, and
    // also assert its completion status in atf[ ] vector.
    // Reminder: atf[ ] is used to keep track of processes that need to add their resources back to work pool
    if $tid == 0$ then
        if $resource\_check[0] == J$ then
            $finish[bid] \leftarrow 1$
            $atf[bid] \leftarrow 1$
        end if
    end if
end if
```

resource_check[ ] value pertaining to a particular process' resource is set to 0. After checking all resources for each process, the *resource_check* vector is reduced via the Add reduction in lines 25-30. After the reduction, a single thread per block takes control by means of the conditional in line 36. Then in line 37, if the result of the *resource_check* reduction is equal to J, this implies that process p_{bid} is able to finish. Therefore, that process' index into the *finish* vector is set to 1 in line 38. Then, in line 39, the same index into the *atf* vector is set to 1 as well.

Looking back at Algorithm 23, after the *Perform_Safety_Check* kernel is complete, the CPU copies the *atf* vector from the GPU to the CPU. Then in lines 8-14 of Algorithm 23, a sequential scan is performed on the *atf* vector. If an index of the *atf* vector (say index i) is equal to 1, that process is able to finish (checked in line 9). Then in line 10, the *total_atf* variable is incremented for every process that has been deemed able to finish. The process that is able to finish needs to add its allocated resources back to the work pool and does so by invoking the *Add_Finished_To_Work* kernel in line 12. When invoking this kernel, GPU-PBA passes it the necessary matrices along with the identifier (i) of the process that was able to finish. Pseudo-code for this kernel is provided in Algorithm 28.

Algorithm 28 Add_Finished_To_Work⋘1,J⋙

```
1: // CUDA Index Variables
2: $tid \leftarrow threadIdx.x$
3:
4: // For processes able to finish, add their allocated resources to work
5: $work[tid] += allocation[p_i \times J + tid]$
```

The *Add_Finished_To_Work* kernel starts by creating the *tid* variable and setting it equal to the value of *threadIdx.x* in line 2. Then in line 5, all resources currently allocated to process i (i.e., allocation[$p_i \times J + tid$]) are added back to the associated resource amounts in the work[ ] matrix (i.e., work[tid]). Looking back at Algorithm 23, after the *Add_Finished_To_Work* kernel is finished executing, the value of the *total_atf* variable is checked in line 17. If *total_atf* is equal to I, then all processes are able to finish (meaning an H-Safe sequence exists) and the safety check is finished (execution of the for-loop stops via the break statement in line 18). Otherwise, the CPU initiates additional iterations of the safety check after clearing the *atf* vector in line 21.

Handling Results of H-Safety Check

After iterations of the H-Safety check have completed, GPU-PBA determines if an H-Safe sequence has been found. Looking at Algorithm 22, after the safety check has been performed, line 19 checks if *total_atf* is not equal to *I*. If the condition is satisfied, it implies that all processes were not able to finish and that no H-Safe sequence exists. As a result, the resource request made by p_i is denied. Thus, all of the prior temporary allocations must be reversed. GPU-PBA then invokes the *RollBack_Allocations* kernel in line 20. The pseudo-code for this kernel is provided in Algorithm 29.

Algorithm 29 RollBack_Allocations⋘1,J⋙

```
// Declare shared memory vector to hold the amounts
// of each resource that a process (say $p_i$) previously requested
__shared__ sRequest[J]

// CUDA Indexing Variables
// process holds the process id ($p_i$) that is releasing resources
// tid holds the value corresponding to each threadIdx.x value and
// idx acts as an index into the 2-D matrices (allocation, need, maximum)
$process \leftarrow p_i$
$tid \leftarrow threadIdx.x$
$idx \leftarrow process \times J + tid$

// Populate shared vector with prior resource request amounts
$sRequest[tid] = requestedResources[tid]$
__syncthreads()

// Deallocate all previously allocated resources
$available[tid] = available[tid] + sRequest[tid]$
$allocation[idx] = allocation[idx] - sRequest[tid]$
$need[idx] = maximum[idx] + allocation[idx]$
```

The *RollBack_Allocations* kernel starts by declaring a shared memory vector of size J called *sRequest* in line 3 that holds the amounts of each resource that a process (say p_i) previously requested. Then in lines 9-11, indexing variables are created. Line 9 creates a process variable and assigns it the identifying value of process p_i. Line 10 creates the variable *tid* and assigns it the value of *threadIdx.x* for each thread in the kernel. Then in line 11, a global index into the 2-D matrices (allocation[ ], need[ ], and maximum[ ]) is created called *idx*. The *idx* variable is formed by multiplying the row in the matrix (row p_i) by the width of the row (width of$J$) and adding the *tid* offset into the result.

Starting in line 14, the *sRequest* vector is populated with the resource request amounts that the process p_i had previously requested. This is followed by a synchronization of all threads in line 15. Line 18 then adds all the previously requested resources back to the availability pool (available[ ] matrix). Followed by the temporary allocations being removed from the allocation[ ] matrix in line 19. Then in line 20, the requesting process' need[ ] matrix values are updated to reflect the removal of allocated resources. After all of these parallel computations have completed, the rollback of all prior temporary allocations has been completed. Now that the handling of a *resource request* event has been covered, Section 5.3.3 covers how GPU-PBA handles the *resource release* event.

Summary of the Resource Request Event Handling

Provided here is a summary of the computations GPU-PBA performs in order to satisfy a *resource request* event. During a *resource request* event, a process may request multiple instances of multiple resources at a time. When a process (say p_i) makes a request, GPU-PBA must check if the requested amount of each resource are less than or equal to process p_i's maximum claim (or less than what is currently in need[i][ ]) [14]. This step is denoted as the validity check.

If all resource requests are less than or equal to p_i's maximum claim, then the request is valid and the algorithm continues to the next phase. Otherwise, the request is invalid and process p_i is denied its request.

Assuming p_i's request was valid, GPU-PBA needs to check if the requested resources are available [14]. GPU-PBA compares p_i's requested resource amounts with the corresponding amounts in the available[ ] vector. If the requested resource amounts are less than or equal to the resource amounts in the available[ ] vector, then the resources are available to satisfy the request. This phase is called the *checking resource availability* phase. If the resources are available to satisfy the request, GPU-PBA continues to the next phase of execution. Otherwise, the request cannot be satisfied and the request made by process p_i is denied.

The next phase of execution is denoted as the *preparing for H-safety check* phase. During this phase, GPU-PBA temporarily grants all the requested resources to process p_i. This is done in three steps: 1) removing requested resources from the available[ ] vector, 2) adding the requested resources to p_i's allocated resources in the allocation[i][ ] vector, and 3) removing the allocated resource amounts from process p_i's maximum claim (or adjusting need[i][ ] accordingly) [14]. After these steps are completed, the work[ ] vector is populated with the values currently contained in the Available[ ] vector. The work[ ] vector is then used to search for processes that are able to finish by acquiring resources that are currently available (in the available[ ] vector) or that will become available during the execution of an H-Safe sequence [14]. The next phase of execution is performing the H-Safety check to determine if an H-safe sequence exists.

The H-Safety check phase attempts to find all processes that are able to finish in order to determine if an H-Safe sequence exists. It accomplishes this by comparing each process' need[ ] vector with the resource amounts available in the work[ ] vector. If a process is deemed able to finish, its allocated resources are added back to the work[ ] vector to be claimed (potentially) by other processes in the system. If following processes are able to finish by claiming the resources that were added back to the work[ ] vector, they are deemed able to finish by setting their corresponding entry in the finish[ ] vector to true (or 1). The H-Safety check computations are repeated for up to I iterations until either all processes are able to finish (indicating that an H-Safe sequence exists) or not all processes are able to finish (indicating that no H-Safe sequence exists) [14]. Assuming that all processes are able to finish and that an H-Safe sequence exists, GPU-PBA anchors all of the temporary allocations made previously. If the converse is true, the pretended allocations are rolled back and the resource request event initiated by process p_i is denied.

5.3.3 Handling a Resource Release

The way GPU-PBA handles a *resource release* event is fairly straightforward. Since this is not a deadlock detection algorithm, but a deadlock avoidance algorithm, then there is no update of resource reachability in a RAG or anything of the sort. So, if a process (say p_i) releases a set of resources, GPU-PBA invokes the *Release_Resources* kernel with a single block containing J threads per block. The pseudo-code for this kernel is provided in Algorithm 30.

To start the *Release_Resources* kernel, a shared vector of size J is declared in line 2 called *sResources*. Lines 5-7 declare the indexing variables used for the kernel. Line 5 assigns the process id (p_i) to the variable *process*. This is followed by the *tid* variable being populated with the values of *threadIdx.x* in line 6. Then in line 7, a 2-D global index is declared called *idx*. The *idx* variable is populated by multiplying the row in the desired 2-D matrix (row *process*) by the width of the row (width of J) and adding the *tid* offset. Lines 10-11 then populate the shared vector *sResources* with the amounts of each resource to be released.

Then in line 14, the released resources are added back to the availability pool by adding the values in the *sResources* vector to the corresponding values in the *available* vector. Line 17 follows by adding the released resource amounts back to process p_i's need pool (via the *need*[ ] matrix). Lastly, in line 20, the resources are finally removed

Algorithm 30 Release_Resources≪1,J≫

```
// Declare shared memory vector
__shared__ sResources[J]

// CUDA Indexing Variables
$process \leftarrow p_i$
$tid \leftarrow threadIdx.x$
$idx \leftarrow process \times J + tid$

// Populate shared vector with released resource amounts
$sResources[tid] \leftarrow releasedResources[tid]$
__syncthreads()

// Add the released resources back to the availability pool
$available[tid] += sResources[tid]$

// Add these released resources back to the processes need pool
$need[idx] += sResources[tid]$

// Remove allocated resources from allocation matrix
$allocation[idx] -= sResources[tid]$
```

from process p_i's allocated resources pool by subtracting the *sResources* values from the corresponding values in p_i's row in the allocation[ ] matrix. After completing the resource release event, GPU-PBA is free to handle more incoming *resource request* or *release* events.

To summarize GPU-PBA's handling of the *resource release* event, the first computation performed is to add the released resources back to the availability pool (available[ ]). Since process p_i released resources, its associated need[i][ ] resource amounts are increased to reflect p_i's potential need for additional resources in the future to complete its tasks. Finally, the released resources are removed from process p_i's allocated resources in allocation[i][ ].

5.4 Experimentation and Results

A serial version of GPU-PBA was first implemented using the C language referred to as CPU-PBA. This version, like the previous CPU implementations of deadlock detection algorithms, utilizes no parallel programming primitives (such as OpenMP) in order to fully demonstrate the capabilities of parallel processing on GPU.

All experiments were performed on an Intel ® Core i7 CPU @ 2.8 GHz with 12 GB RAM. The CUDA GPU-PBA implementation was tested on three different GPUs: GTX 670, Tesla C2050, and Tesla K20c. The GTX 670 has 7 SMXs (1344 CUDA Cores) with 2 GB Global Memory, the Tesla C2050 has 14 SMs (448 CUDA Cores) with 3 GB Global Memory, and the Tesla K20c has 13 SMXs (2496 CUDA Cores) with 5 GB Global Memory.

To verify the correctness of our algorithm, both CPU-PBA and GPU-PBA were tested against a series of different resource events. Once results were verified as correct, more processes and resources were added to the resource requests (thus, adding additional computations for calculations of PBA). What was found was that the CPU and GPU were almost identical in terms of run-time when it came to events that fell into the best-case scenario of finding the H-Safe sequence. So, to truly see how GPU-PBA fairs against CPU-PBA, tests were performed on the worst case scenarios where all I iterations of the safety check must be performed. In other words, each iteration of the H-Safety check yields a single process that is able to finish.

To test GPU-PBA's run-time behavior, a several tests were run with varying numbers of resources and increasing amounts of processes. This test should show how CPU-PBA and GPU-PBA handle higher levels of resource contention where increasing numbers of processes are competing for lesser resources. Tables 5.2, 5.3, and 5.4 show the run-times of CPU-PBA versus GPU-PBA with 256, 512, and 1024 resources, respectively. For each of these tables, the resource amount was kept constant while process counts were increased. Following the result tables, Figures 5.2, 5.3, and

5.4 depicts the speedups GPU-PBA achieves over CPU-PBA for the sets with 256, 512, and 1024 resources, respectively.

Table 5.2: Run-Time/Speedup of CPU-PBA and GPU-PBA (256 Resources)

Processes×Resources	CPU-PBA	GTX 670	Tesla C2050	Tesla K20c
512×256	0.03/0x	0.02/0.65x	0.03/0.10x	0.02/0.65x
1024×256	0.13/0x	0.06/1.17x	0.08/0.63x	0.05/1.60x
2048×256	0.51/0x	0.21/1.43x	0.25/1.04x	0.16/2.19x
4096×256	2.05/0x	0.67/2.06x	0.87/1.36x	0.55/2.73x
8192×256	8.34/0x	2.37/2.52x	3.10/1.69x	1.99/3.19x

Table 5.3: Run-Time/Speedup of CPU-PBA and GPU-PBA (512 Resources)

Processes×Resources	CPU-PBA	GTX 670	Tesla C2050	Tesla K20c
512×512	0.05/0x	0.03/0.67x	0.04/0.25x	0.02/1.50x
1024×512	0.20/0x	0.12/0.63x	0.13/0.51x	0.09/1.18x
2048×512	0.78/0x	0.39/1.00x	0.47/0.66x	0.30/1.60x
4096×512	3.26/0x	1.37/1.38x	1.70/0.92x	1.13/1.88x
8192×512	13.60/0x	5.19/1.62x	6.40/1.13x	4.30/2.16x

Table 5.4: Run-Time/Speedup of CPU-PBA and GPU-PBA (1024 Resources)

Processes×Resources	CPU-PBA	GTX 670	Tesla C2050	Tesla K20c
512×1024	0.10/0x	0.06/0.67x	0.08/0.25x	0.05/1.00x
1024×1024	0.39/0x	0.23/0.70x	0.31/0.26x	0.18/1.17x
2048×1024	1.59/0x	0.88/0.81x	1.19/0.34x	0.68/1.34x
4096×1024	6.62/0x	3.25/1.04x	4.60/0.44x	2.63/1.52x
8192×1024	26.62/0x	12.69/1.10x	17.90/0.49x	10.30/1.58x

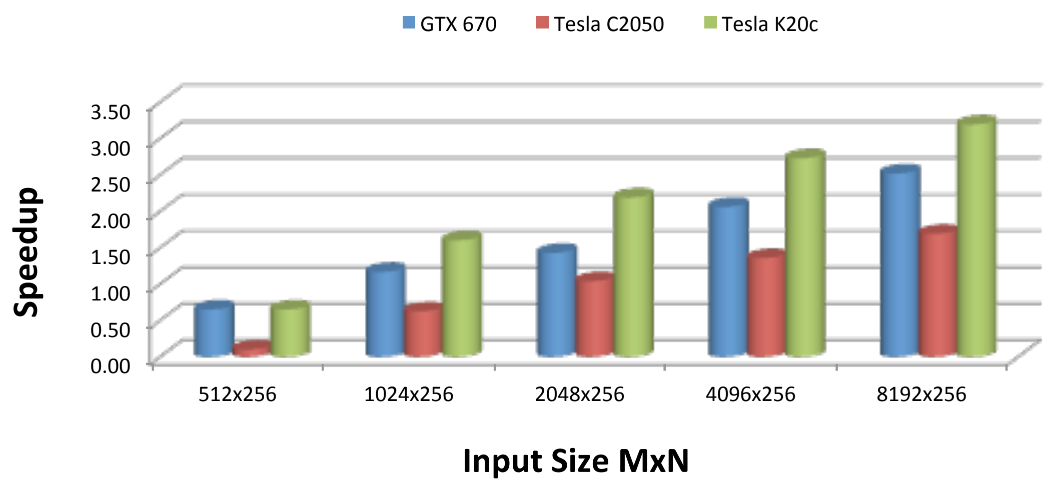


Figure 5.2: GPU-PBA Speedup w/ 256 Resources

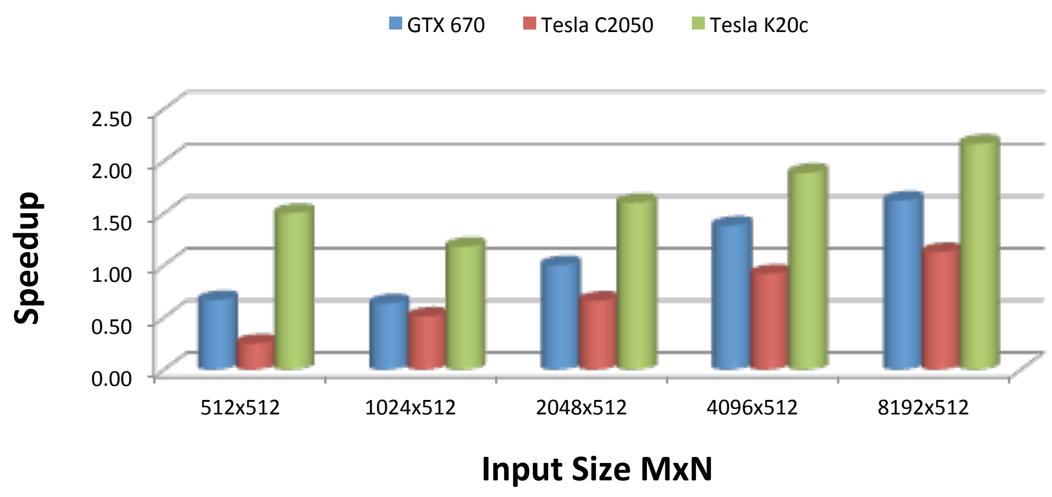


Figure 5.3: GPU-PBA Speedup w/ 512 Resources

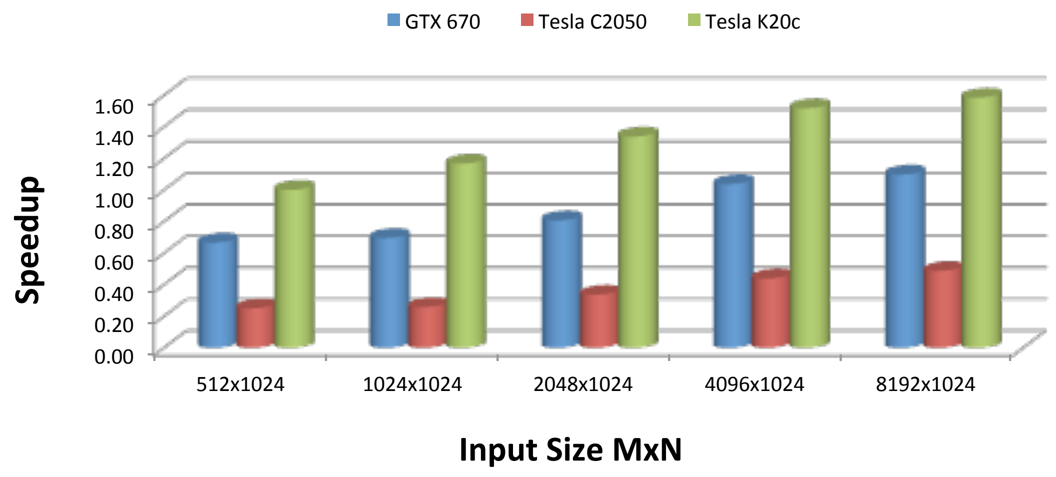


Figure 5.4: GPU-PBA Speedup w/ 1024 Resources

5.5 Conclusion

A GPU-based approach to deadlock avoidance has been implemented and successfully tested. Speedups achieved by GPU-PBA over CPU-PBA are in the range of 0.10-3.2X. There are several hypothesized reasons for the speedup difference between this algorithm and the two prior algorithms. The reasons and potential solutions will be discussed in the Future Work section of this thesis.

6 SUMMARY

6.1 Thesis Conclusions

This thesis has proposed and developed three algorithms, namely a single-unit deadlock detection algorithm (GPU-OSDDA), a multi-unit deadlock detection algorith (GPU-LMDDA), and a deadlock avoidance algorithm (GPU-PBA) on the GPU platform. The goal of providing deadlock detection/avoidance algorithms as an *interactive service* to the CPU has been accomplished. By modeling prior hardware deadlock detection/avoidance algorithms for use on the GPU, utilizing bit-packing methods to store algorithm matrices, and solving algorithm computations with bit-wise operations, the three algorithms developed have, in general, provided speedups over two orders of magnitude higher than CPU implementations. The following provides lists of the contributions brought by each of the aforementioned algorithms.

GPU-OSDDA

1. First single-unit deadlock detection algorithm available for the GPU platform
2. Keeps track of all resource allocation events on the GPU
3. Implements matrix storage and algorithm computations on bit-vectors
4. Has the ability to handle large numbers of processes and resources
5. Required memory space substantially reduced with current implementation
6. Limited interaction with the CPU, allowing for an *interactive service* behavior
7. Achieved speedups over two orders of magnitude higher than CPU version

GPU-LMDDA

1. First multi-unit deadlock detection algorithm available for the GPU platform
2. Keeps track of all resource allocation events on the GPU
3. Implements matrix storage and algorithm computations on bit-vectors
4. Has the ability to handle a set size of 1024x1024 (processes x resources)
5. Required memory space substantially reduced with current implementation
6. Highly parallelized reachability computation which could be used in different applications
7. Limited interaction with the CPU, allowing for an *interactive service* behavior
8. Achieved speedups over two orders of magnitude higher than CPU version

GPU-PBA

1. First parallelized deadlock avoidance algorithm available for the GPU platform
2. Provided speedups over CPU implementation

6.2 Future Work

The three proposed algorithms in this thesis have laid the groundwork for future research regarding parallelized deadlock detection and avoidance algorithms on the GPU platform. The single-unit deadlock detection algorithm (GPU-OSDDA) was able to achieve substantial speedups over the CPU with no process or resource amount limitations (with the exception being the available memory space on a GPU). A potential extension to this algorithm could be to implement a sparse-matrix storage mechanism over-top of the bit-vector technique currently being utilized. While this may not eliminate all nodes of disinterest for computations (as 32 sequential process or resource amounts would have to be 0), in RAGs with very sparse elements, substantial speedups could be achieved over the current implementation if the sparse locations of a RAG were removed from consideration during computation of GPU-OSDDA.

The multi-unit deadlock detection algorithm (GPU-LMDDA), while able to achieve impressive speedups over the CPU, has a caveat being that its performance degrades as the process and resource amounts approach 1024. This is due to scheduling the maximum number of threads per block (1024) during the initialization and computation phases of the reachability computation. A future addition would be to expand the algorithm's capability by reworking the computations involved to accept a lower number of threads per block (preferably 256), which then causes computations to be broken apart and handled by more blocks. As a result, this would allow for a much greater number of allowable process and resource amounts the algorithm may handle and greatly increase speed since more blocks may be running concurrently on an SM(X). Implementing a sparse-matrix storage mechanism over-top of the bit-vector technique used in GPU-LMDDA may also advocate greater speedups, similar to what was suggested for GPU-OSDDA.

GPU-PBA, the deadlock avoidance algorithm proposed in this thesis, presented concerns about being able to significantly accelerate computations of deadlock avoidance on GPUs. There are several primary limitations, the first being the synchronization problem. During the H-Safety check computation, the comparisons performed for all processes are done in parallel. The issue is that there are no synchronization primitives between blocks in the CUDA framework. This implies that for each iteration of the H-Safety check, the algorithm must return to the CPU to synchronize all blocks. When re-entering the H-Safety check kernel, all matrix elements of interest must be reread from global memory, all the while incurring kernel call overhead for each iteration. This alone greatly increases the run-time of the algorithm. In contrast, the CPU performs all of these comparisons in serial and is able to cache most (if not all) values for future iterations of the H-Safety check. Additionally, the CPU does not incur a kernel call overhead as GPU-PBA does. These factors, as well as the CPU's increased clock speeds over the GPU, lead to the limited speedups that GPU-PBA was able to achieve. If the CUDA framework provided a cross block synchronization mechanism (or if one could be implemented efficiently), then matrix elements could be cached, thus eliminating kernel call overhead and allowing for cache utilization.

LIST OF REFERENCES

LIST OF REFERENCES

[1] E. W. Dijkstra, "The origin of concurrent programming," ch. Cooperating sequential processes, pp. 65–138, New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[2] A. N. Habermann, "Prevention of system deadlocks," *Communications of the ACM*, vol. 12, pp. 373–377, July 1969.

[3] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, pp. 179–196, Sept. 1972.

[4] T. F. Leibfried, "A deadlock detection and recovery algorithm using the formalism of a directed graph matrix," *SIGOPS Operating Systems Review*, vol. 23, pp. 45–55, Apr. 1989.

[5] J. K. Kim and K. Koh, "A O(1) time deadlock detection scheme in a single unit and single request multiprocessor system," in *TENCON '91.1991 IEEE Region 10 International Conference on EC3-Energy, Computer, Communication and Control Systems*, vol. 2, pp. 219–223, Aug.

[6] A. Shoshani, *Detection, Prevention and Recovery from Deadlocks in Multiprocess Multiple Resource Systems.* Reports, Princeton University, 1969.

[7] J. G. Kim, "An algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems," in *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, PAS '97, (Washington, DC, USA), pp. 233–, IEEE Computer Society, 1997.

[8] P. Shiu, Y. Tan, and V. J. Mooney, "A novel parallel deadlock detection algorithm and architecture," in *Hardware/Software Codesign, CODES 2001. Proceedings of the Ninth International Symposium on*, pp. 73–78, 2001.

[9] X. Xiao and J. Lee, "A novel parallel deadlock detection algorithm and hardware for multiprocessor system-on-a-chip," *Computer Architecture Letters*, vol. 6, no. 2, pp. 41–44, Feb.

[10] X. Xiao and J. Lee, "A true O(1) parallel deadlock detection algorithm for single-unit resource systems and its hardware implementation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 1, pp. 4–19, Jan.

[11] X. Xiao and J. Lee, "A novel O(1) parallel deadlock detection algorithm and architecture for multi-unit resource systems," in *Computer Design, ICCD 2007. 25th International Conference on*, pp. 480–487, Oct. 2007.

[12] X. Xiao and J. Lee, "A parallel multi-unit resource deadlock detection algorithm with O(log2(min(m,n))) overall run-time complexity," *Elsevier Journal of Parallel and Distributed Computing*, vol. 71, pp. 938–954, July 2011.

[13] J. Lee, *Hardware/software deadlock avoidance for multiprocessor multiresource system-on-a-chip.* PhD thesis, School of ECE, Georgia Institute of Technology, Atlanta, GA, USA, Nov. 2004.

[14] J. Lee and V. J. Mooney, "A novel O(n) parallel banker's algorithm for system-on-a-chip," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 12, pp. 1377–1389, Dec. 2006.

[15] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Computing Surveys*, vol. 28, pp. 67–70, Mar. 1996.

[16] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* Morgan Kaufmann - Elsevier, 1 ed., 2012.

[17] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. Accessed: 2013-03-22.

[18] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler." http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. Accessed: 2013-03-22.

[19] NVIDIA Corporation, "CUDA C Best Practices Guide." http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, October 2012. Accessed: 2012-09-25.

[20] NVIDIA Corporation, "CUDA C Programming Guide." http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, October 2012. Accessed: 2012-09-25.

[21] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference*, 2010.

[22] H. S. Warren, *Hacker's Delight.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[23] M. Harris, "Optimizing Parallel Reduction in CUDA." http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction/pdf, 2009. Accessed: 2013-02-12.