[[1]](#footnote-1)

CUDA Implementation of Banker’s Algorithm for Deadlock Avoidance

Enes Muvahhid ŞAHİN

*Abstract*—Deadlock has always been a critical problem to be addressed carefully in multi-process & multi-resource systems. It severely affects programs, their working principles, and performances. Hence, it must be handled properly, that is, deadlock must be avoided, detected, or recovered. Banker’s Algorithm [1], proposed by Dijkstra, attacks deadlock avoidance problem, and proposes a method for detecting potential deadlocks in a multi-process & multi-resource system in order to avoid them. In this paper, we propose an efficient GPU implementation of Banker’s Algorithm. Our approach utilizes NVIDIA’s parallel programming platform, CUDA. We also make use of dynamic parallelism which enables GPU to generate works for itself without explicit generation of kernels by the CPU [2]. Depending on the input contents and sizes, our CUDA implementation of Banker’s Algorithm can achieve 20x-30x speed improvement compared to sequential/partially-parallel CPU implementations.

*Index Terms*—Banker’s Algorithm, CUDA, Deadlock Avoidance, GPU Programming, Resource Allocation

# Motivation & Significance

D

eadlock is a well-known problem that appears in multi-process & multi-resource systems. Basically, Deadlock is a situation in which some processes in a system wait on some resources such that none of the processes can terminate since necessary resources are held by some other processes. It is very crucial to handle deadlock situations properly because deadlock may lead programs or systems to unexpected or undefined states. For example, in a mission-critical military application, a deadlock appearing in some part of the system may result in catastrophic consequences. There have been many attempts to handle deadlock situations such as deadlock avoidance [1], deadlock detection [3], recovery from deadlock [4]. Each approach attacks to a different aspect of the deadlock problem.

Banker’s Algorithm, proposed by Edsger Dijkstra [1], is a resource allocation and deadlock avoidance algorithm which enforces a resource allocation policy. With this policy, resource allocation requests that may lead to deadlocks are anticipated beforehand and restrained so that deadlock situations never occur.

Banker’s Algorithm is a very important and effective way of dealing with deadlocks because it makes sure deadlock never occurs in the first place in a multi-process & multi-resource system. It differs from deadlock detection and recovery from deadlock algorithms since it proposes a resource allocation mechanism and guarantees deadlock will not happen.

Number of processes and shared resources in today’s computers continue to grow as processing capabilities of the hardware increases. It becomes very important to handle large number of processes and resources without causing deadlocks. Banker’s Algorithm solves this problem. However, this algorithm’s performance depends on number of processes and number of resource types. As these two quantities grow, it becomes very time consuming to apply the algorithm using standard, sequential, CPU-based implementations. Our proposed approach tackles this issue and provides an effective and super-fast way of applying Banker’s Algorithm even in systems with large number of processes and/or resources.

In Section II, we describe the standard Banker’s Algorithm and introduce necessary data structures and basic algorithmic steps. In Section III, we discuss previous approaches that implements Banker’s Algorithm and we state their shortcomings. In Section IV, we explain our CUDA based Banker’s Algorithm in detail and discuss the decisions we took while developing and implementing the algorithmic solution. In Section V, we provide experimental results and comparisons of different implementations. In Section VI, we make some discussions about our algorithm and implementation, state its powerful and weak aspects, and provide some ideas about the future work.

# Problem Statement

Banker’s Algorithm determines whether a requested allocation may lead to deadlock or not in a multi-process & multi-resource system. It first checks whether the requested allocation is valid or not. Then, if the request is feasible, it investigates whether all processes can finish if allocation request is served. Namely, it checks if the allocation yields to a safe state (deadlock will not happen) or to an unsafe state (allocation might cause deadlock).

For this purpose, algorithm makes use of following data structures (m: number of resource types, n: number of processes):

*Available[m]:* One dimensional array of size m. Indicates the number of available resources of each type.

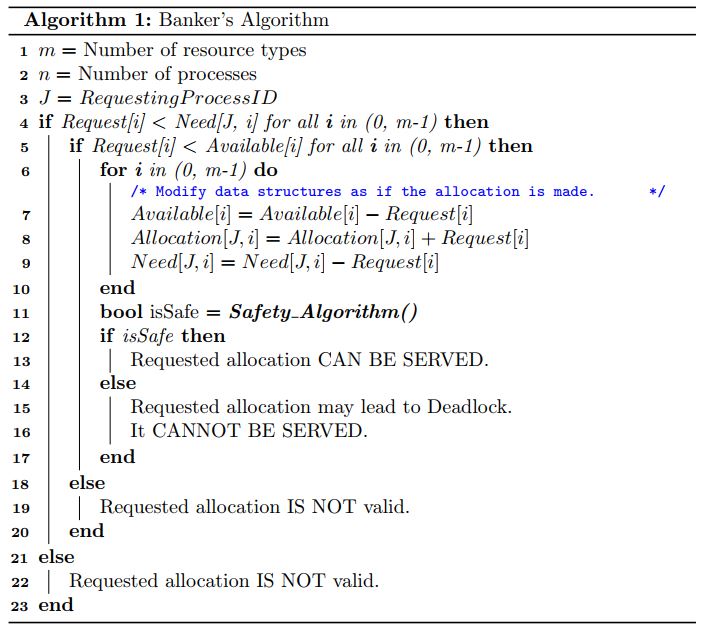
*Max[n, m]:* Two dimensional array of size n x m. Defines the maximum demand of each process from each resource type.

*Allocation[n, m]:* Two dimensional array of size n x m. Defines the number of resources of each type currently allocated to each process.

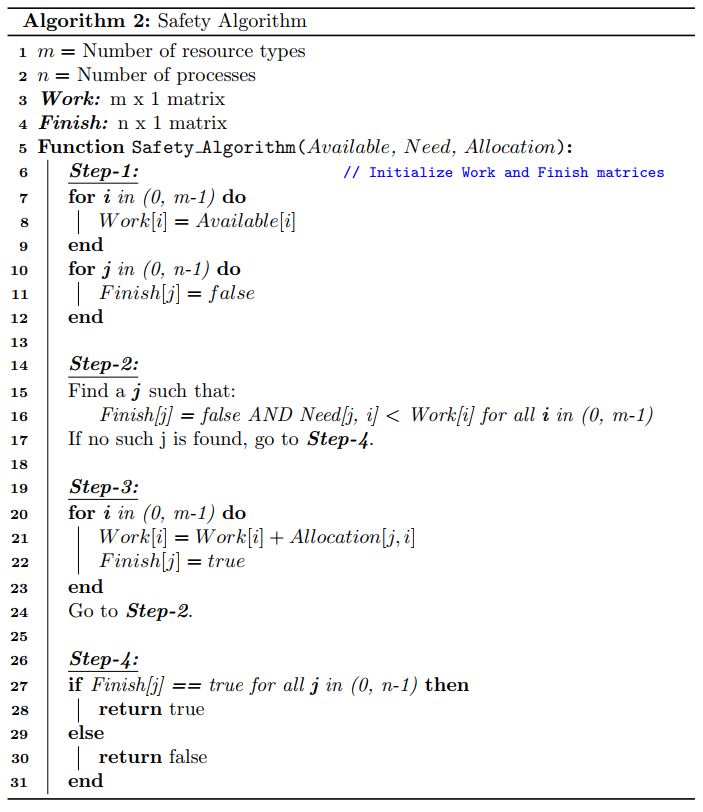
*Need[n, m]:* Two dimensional array of size n x m. Indicates remaining need of each process, of each resource type.

*Request[m]:* One dimensional array of size m. It indicates requested allocation amount of each resource type for a given process represented by *requestingProcessId*.

When a process with ID – *requestingProcessId = J,* makes an allocation request where requested amount for each resource type is represented by the *Request* vector, algorithm works as follows:



Detailed flow of Safety Algorithm is as follows:



We check whether the resulting state after modification is safe or not using the **Safety Algorithm** above (A safe state is a state for which there is an allocation sequence among processes which results in successful termination of all processes).

Before the safety algorithm, we modified the matrices as if the requested allocation were made.

Then, in the safety state, we investigate states of the matrices, and seek for processes that can terminate its job using the available resources (Work vector). When such a process is found, we claim that it is able to terminate, we mark it as finished (Finish[j] = true), and we transfer resources allocated to that process to the available resources (Work = Work + Allocation).

With the new amount of available resources, we again seek for unfinished and able to finish process in step 2.

It goes on like this until we cannot find any further processes that can terminate.

Finally, we check if all processes were able to finish (Finish[j] = true for all j). If yes, then the system is in a safe state, if not, then the system is in an unsafe state and may lead to deadlocks.

If the matrices after modification leads to a safe state, allocation can be made. Otherwise requested allocation must wait.

# Prior Work & Limitations

Banker’s Algorithm is a well-known algorithm that is being taught in OS courses for years. There are many implementations available on the GitHub [5]. All of them are implemented on CPU and in a sequential manner. They perform elementwise comparison/addition/subtraction operations over all elements of the data structures mentioned in Section II. Moreover, they do not utilize any parallelism in Safety Algorithm as well.

As we will discuss in the Section V, CPU implementations fail to perform well when number of processes and resources are large.

Since computational power of computers increases each day, number of processes and resources in computer systems also grow up. In such realistic scenarios, data structures mentioned in Section II would be huge and processing them sequentially would be very time consuming. So, available implementations are not very applicable to real-world systems with many processes and many different resources.

# Our Algorithm and Implementation

We divide the described Banker’s Algorithm in the previous section into 3 states and discuss those states separately. These are Validation, Modification and Safety States. Overall block diagram of the algorithm is given in Figure 1. Detailed schematics of Modification and Safety States are also given in Figure 2, and Figure 3 respectively.

***Notation:***

**X(i),** a vector of size 1 x m, represents

* ith row vector of X, if X is an n x m matrix
* Vector itself, if X is a 1 x m vector.

**Requesti ≤ Need(i**) means that each element of Requesti vector is less than or equal to corresponding element of ith row of Need matrix where Requesti is the requested allocation vector for process with requestingProcessId = i.

**X(i) = X(i) + Requesti** means that each element of Requesti is added to each corresponding element of ith row of X if X is a metin, harita içeren bir resim

Açıklama otomatik olarak oluşturuldumatrix, or added to each corresponding element of X if X is a vector with same dimension as Requesti.

Figure : Overall block diagram of the algorithm

Other notations are similar to these three notations.

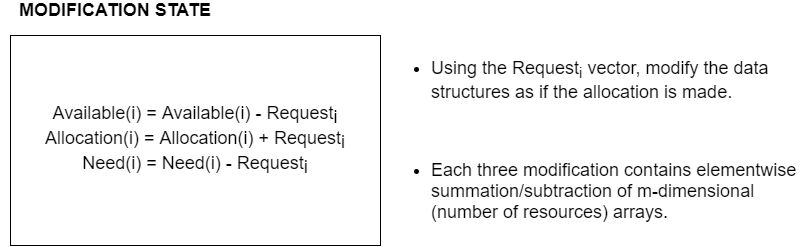


Figure 2: Block diagram of modification state

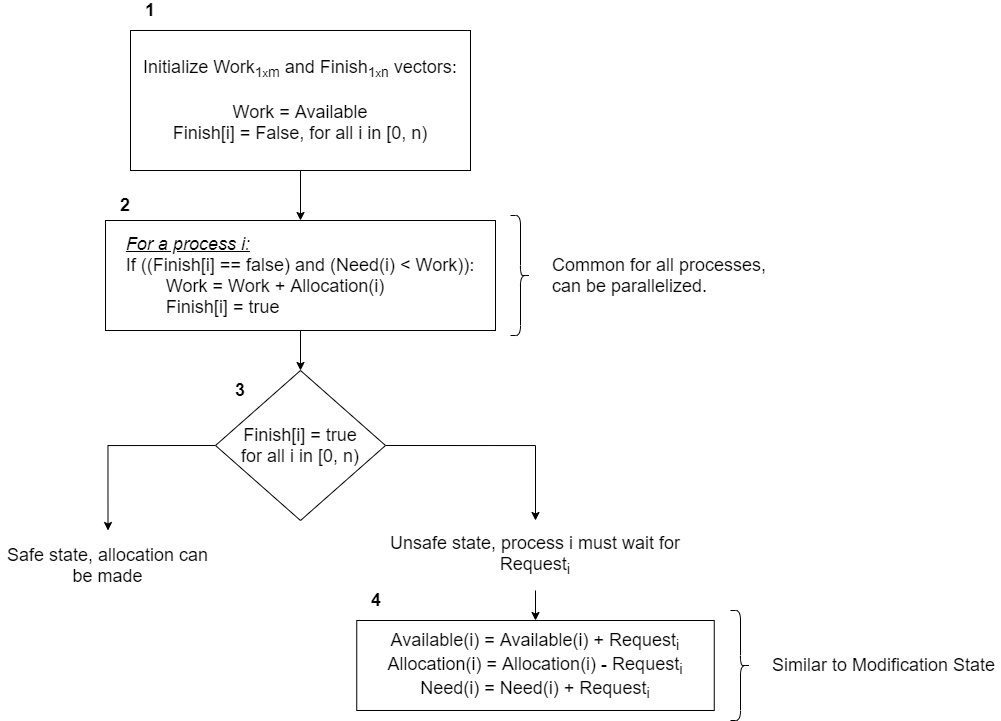


Figure 3: Block diagram of Safety State

We have implemented 3 versions of Banker’s Algorithm:

* Single-threaded CPU version,
* Multi-threaded, partially parallel CPU version,
* GPU version with CUDA.

These 3 versions will be explained separately.

In all implementations, we start with initializing host vectors/arrays from text files. Details about the initialization will be discussed in Section V.

## SINGLE-THREADED CPU IMPLEMENTATION

In the single-threaded CPU implementation, all elementwise operations are performed by looping through each element of matrices.

### Validation State

In the Validation State of this implementation, we iterate through all elements of *Request* vector and compare it with corresponding elements of *Available(i)* and *Need(i)*. If any element of *Request* vector is greater than corresponding elements of *Available* or Need vectors, that means requested allocation IS NOT valid. So, request cannot be served and algorithm finishes.

If each element of *Request* vector is less than or equal to that of *Available* and *Need* vectors, than request is valid, and we can proceed to Modification State.

### Modification State

In the Modification State of this implementation, **in a for loop of size m**, number of resource types, we apply following operations to each element of *Available, Allocation* and *Need* matrices:

* *Available = Available – Request*
* *Allocation = Allocation + Request*
* *Need = Need - Request*

By this way, we modify matrices as if the allocation is served and *process i* is granted with the resources in *Requesti* vector. Then, we continue with Safety State in order to check whether the new states of matrices can cause a deadlock or not.

### Safety State

In the Safety State of this implementation, we first initialize *Finish* and *Work* vectors as described in *Algorithm 2*, in Section II.

After initialization, in an infinite *while* loop:

* We try to find a *process i* for which *Finish[i] = false*. If we find such process, we check whether *Need(i) < Work* for that process. If it is, we transfer allocated resources of that process to the *Work* vector. If not, we seek for another *process i* as described above.
* We break the loop when we cannot find a *process i* for which *Finish[i] = false* AND *Need(i) < Work.*

Finally, we check whether all elements of *Finish* vector are true, i.e. whether all process can terminate successfully given the new states of matrices. If that is the case, we are in a safe state, and requested allocation IS servable.

However, if there is at least one *non-true* entry in the *Finish* vector, then the system is in an unsafe state. Requested allocation may lead to deadlock. Hence, request IS NOT servable.

## MULTI-THREADED CPU IMPLEMENTATION

Multi-threaded CPU implementation is not fully parallel. During validation and modification states, operations on different matrices are performed on different threads in parallel. Operations inside each thread, i.e. operations on thread-specific matrices, are again performed sequentially by looping through each element of matrices.

Safety State implementations of multi-threaded and single-threaded versions are completely the same.

There are several reasons behind not implementing a fully parallel CPU version where elementwise operations and Safety Algorithm are also parallelized:

* As discussed in Section III, available implementations do not perform well for large matrices. This issue is what we try to address. So, our aim is boosting the performance of Banker’s Algorithm even for systems with large number of processes and resources. However, this is not feasible in CPU because creating large number of CPU threads are expensive to create.
* Moreover, number of CPU cores are very limited and creating that many threads on these cores degrades the performance drastically.

### Validation State

In the Validation State, our aim is to compare *Request* vector with *Available* and *Need(i)* vectors to see if all elements of *Request* vector are less than or equal to that of other two vectors.

For this purpose, we create two threads, one for comparing *Request* with *Available* and one for comparing *Request* with *Need(i)*.

Inside each thread, we iterate through *Request* vector and the other vector, make an elementwise comparison, and modify a flag if there is at least one *invalid* element in *Request* vector that is greater than corresponding element of the other vector (flags are called *invalidResourceIdxNeed* and *invalidResourceIdxAvailable* in the code and they are set to index of the *invalid* element or -1 if all elements are valid.).

If the flags of both threads are set to -1, i.e. both threads state that request is valid, we move on to the Modification State. Otherwise, requested allocation IS NOT valid.

### Modification State

In the Modification State, we create 3 threads in order to modify each of the three vectors, *Available, Allocation,* and *Need*. Inside each thread, we perform following elementwise operations on the corresponding vector:

*Thread modifyAvailable: Available = Available – Request*

*Thread* *modifyAllocation: Allocation = Allocation + Request*

*Thread modifyNeed: Need = Need - Request*

### Safety State

Safety State for this implementation is the same as single-threaded implementation. Modified *Available, Allocation,* and *Need* matrices are passed to the Safety State and the same implementation is used.

Note: In CPU implementations we heavily used C++ STL which provides efficient and easy-to-use data structures and algorithms.

## GPU IMPLEMENTATION WITH CUDA

As stated earlier, our main aim is to provide a fast implementation of Banker’s Algorithm for systems with large number of processes and resources.

In today’s computers there are hundreds of processes and thousands of resources that are being used by these processes. We we design our algorithm accordingly, in order to fulfil necessities of such systems.

We define our kernel grid dimensions so that 65535 \* 1024 (max number of blocks per one grid dimension \* max number of threads per block) processes are supported.

Moreover, we support up to 1024 processes. This number seems enough for today’s computers, but we will discuss the reason behind limiting number of processes to 1024 later.

Now we will describe our GPU implementation in detail.

As in all CUDA applications, we start with memory allocations on host and device memory. We allocate memory for host and device matrices according to their sizes which are described in Section II.

Then, we need to copy initialized host matrices to device matrices. For this purpose, we create 4 CUDA streams. We fill device matrices (*Available, Max, Allocation, Need, Request*) using created 4 streams plus default stream 0. Using these streams, we perform *cudaMemcpyAsync* in order to make use of maximum number of DMA engines.

After device matrices are initialized by copying contents from host matrices, we start implementing Banker’s Algorithm.

### Validation State

In the validation state, we need to check followings:

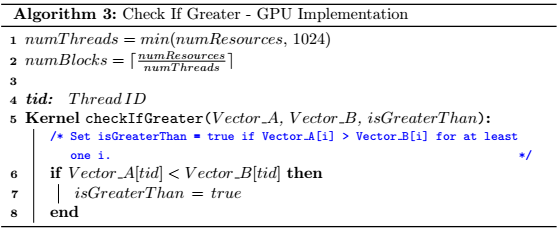
* *Requesti < Need(i)*
* *Requesti < Available(i)*

These vectors/row-vectors are of size *1 x numResources*. In order to compare them, we define a kernel, *checkIfGreater*. Threads of this kernel compares two vectors elementwise using thread IDs, then set a flag in the global memory, *isGreaterThan*, if corresponding element of *Vector\_A* is greater than *Vector\_B*.

In order to parallelize comparison of *Requesti* vector with *Need(i)* and *Available(i)* we launch two kernels in different streams, one for *Need*, one for *Available*.

When both kernels finish, we check *isGreaterThan* flags of two kernels. If at least one of them is true, then, requested allocation IS NOT valid. If both of them are false, then all elements of *Requesti* vector are less than or equal to that of *Need(i)* and *Available(i)*, and requested allocation IS valid.

Kernel configuration and Algorithmic flow of *checkIfGreater* kernel is given below:



We configure kernels so that number of blocks is minimum by setting maximum possible threads per block. This is done in order to support systems with large number of resource types.

All matrices used in this state are stored in the global memory. This is OK because we reach each element of global memory exactly once. So, storing them in shared memory would require same amount of global memory access plus extra shared memory accesses.

Furthermore, we store *isGreaterThan* flags in global memory. All threads launched for a kernel modifies the same global memory location. This does not create a race condition since only writing operation is performed with all threads writing the same value.

If requested allocation is valid, we move on to Modification State.

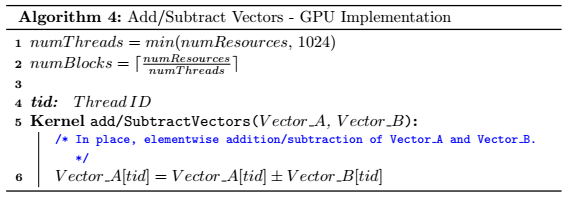
### Modification State

Modification State is pretty similar to Validation State in terms of kernel configurations. In this state we need to perform following elementwise modifications:

* *Available = Available – Request*
* *Allocation = Allocation + Request*
* *Need = Need - Request*

Each of these vectors/row-vectors are of size *1 x numResources*. In order to modify them in parallel, we launch 3 kernels in different streams with the same configuration as in *checkIfGreater* kernel. Each thread modifies exactly one element of corresponding matrices.

Launched kernel functions are called *subtractVectors,* and *addVectors* in the code, and they are identical to each other except that operations they perform. Pseudocode for these kernels are given below:



### Safety State

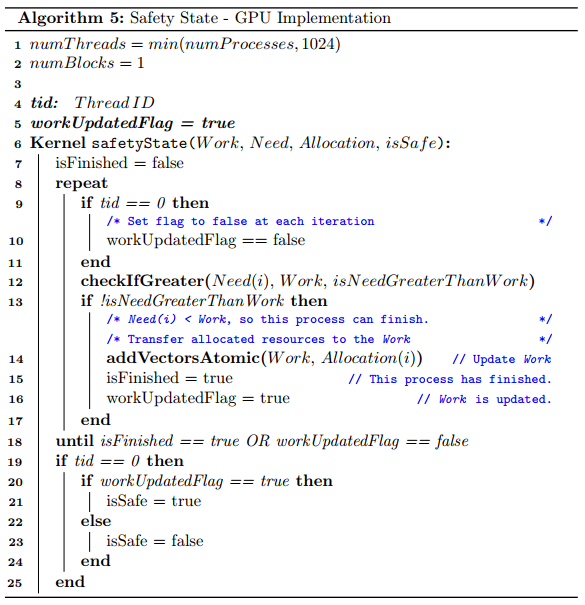
Flow char of Safety State is given in Figure 3.

We start with allocating memory for *Work* and *Finish* matrices. Then, we perform *cudaMemcpyAsync* operation to copy *Available* matrix to *Work* vector. In a different stream, we perform *cudaMemsetAsync* operation to initialize *Finish* matrix with all 0’s (false).

After initialization we create a kernel, *safetyState*, with 1 block and *numProcesses* threads per-block. Note that, we implemented our algorithm so that *numProcesses* must be less than 1024. The reason behind this will be discussed in Section VI.

We store a flag in **shared memory** called ***workUpdatedFlag*** that is common to all threads (processes).

Overall workflow of remaining part of the Safety State is given below:



Above pseudocode summarizes our implementation of the Safety State. Inside *safetyState* kernel, we configure 2 more kernels:

* *checkIfGreater* in order to check *Need(i) < Work*
* *addVectorsAtomic* in order to transfer allocated resources of process i to the *Work* vector. This kernel performs elementwise **atomic** additions of *Allocation(i)* and *Work* vectors. Atomic addition is necessary because more than one processes can modify the *Work* vector. With atomic addition, we prevent race conditions.

Configuring new kernels inside GPU kernels is possible with a concept called **Dynamic Parallelism [2].** Dynamic Parallelism is available in **CUDA 5.0** and afterwards. It also requires NVIDIA GPUs with **compute capability 3.5 or higher.**

When all processes break out of the loop, thread 0 checks status of the *workUpdatedFlag.* ***We can determine whether the state is safe or not using this flag.***

Scenarios available for the status of this flag is as follows:

**1 -** NONE OF THE PROCESSES IS ABLE TO FINISH AT FIRST ITERATION, flag is set to false. At the 2nd iteration, loop is broken. flag == false

**2 -** SOME OF THE PROCESSES FINISHED EARLIER. AT SOME POINT NONE OF THE PROCESSES IS ABLE TO FINISH. So, none of the processes is able to set the flag to true. At the next iteration, loop is broken. flag == false

**3 -** ALL OF THE PROCESSES ARE ABLE TO FINISH. At the last iteration, last process finished and set flag = true. At the next iteration, none of the processes has isFinished == false since all of them are finished. Loop is exited and flag == true.

So, if the state is safe, i.e. all processes are able to finish (Scenario - 3), then flag == true. Otherwise, if the state is unsafe, flag == false.

Detailed explanation of the above pseudocode, *addVectorsAtomic* kernel, and dynamic parallelism are given in the Appendix A.

There are several reasons we preferred this kind of implementation:

First of all, we store *Dynamic Parallelism* is used in order to perform elementwise operations in the Safety State such as:

* Checking *Need(i) < Work*
* *Work = Work + Allocation(i)*

in parallel. With the help of Dynamic Parallelism, each process (thread) can launch new kernels in order to perform above operations in parallel. Launched kernels have one block and *numResources* threads since above operations are performed on vectors with size *1 x numResources*.

Secondly, we store *workUpdatedFlag* in the **shared memory** because all processes(threads) needs to modify this flag when they made a change to the *Work* vector. By this way, all processes are notified when there is a change to the *Work* vector. Since this is a shared resource, race conditions can occur. In order to prevent this, we use *\_\_syncthreads()* after writing and before reading the flag.

Thirdly, we store *Work* vector in the global memory. One can think that storing it in the shared memory is more feasible since all threads share the same vector. However, *Work* vector is of size *1 x numResources.* In order to perform elementwise operations on this vector in parallel, we need to create *numResources* threads per process (thread). That makes *numProcesses x numResources* threads. However, shared memory is only accessible for threads within the same block, i.e. maximum of 1024 threads can access this memory. So, elementwise operations are not parallelizable if *Work* vector is stored in the shared memory.

Finally, number of processes in Banker’s Algorithm are limited to 1024 for our implementation. The reason behind this is, first of all, 1024 processes is a large number even in today’s computers. So, it fulfils needs of the today’s systems. Moreover, when *numProcesses* is less than 1024, we can create a thread for each process and each thread belongs to a single block. This make it super easy to perform synchronization between different threads using *\_\_syncthreads().* This kind of synchronization makes it easy to notify other threads when there is an update made to the *Work* vector, while preventing race conditions. If we had more than 1024 processes (threads), using shared memory would not be possible. For synchronization, we would require a different inter-block synchronization mechanism such as using global memory flags together with atomic operations.

# Conclusion

## A conclusion section is not required. Although a conclusion may review the main points of the paper, do not replicate the abstract as the conclusion. A conclusion might elaborate on the importance of the work or suggest applications and extensions.

Appendix

## Safety State for GPU Implementation

In this section we will discuss Safety Algorithm for GPU implementation in more detail.

We configure *safetyState* kernel so that there is exactly one block with *numProcesses* threads.

In each thread, we keep a boolean called *isFinished* which corresponds to ith entry of the *Finish* vector. When a process “*is able to terminate*” it sets *isFinished* variable to true. There is another flag stored in the shared memory called *workUpdatedFlag.* If this flag is true, it means that there has been an update to the *Work* vector. This flag is set to true by the Thread-0 initially.

After initializations, each thread runs inside a while loop. There are 2 ways for a thread ***i*** to exit this loop:

1. That thread is able to finish, i.e. its *isFinished* flag was false and *Need(i) < Available*. In this case, *isFinished* is set to true and in the next iteration condition of the while loop is violated and thread ***i*** exits the loop.
2. All threads are synchronized at the beginning of the loop. Then, they check whether *workUpdatedFlag* is true or false. If it is false, that means that in the previous iteration of the loop, none of the processes (threads) were able to terminate (could not satisfy *isFinished = false* AND *Need(i) < Available*). So, none of the threads made a change to the *Work* vector. Hence, algorithm will not be able to proceed further in this iteration as well. So, when *workUpdatedFlag* is false all remaining threads exit the loop. This indicates that the system is in an unsafe state and requested allocation IS NOT servable.

Now let us consider what happens when threads actually run through the loop.

In the beginning of the loop all threads are synchronized. Then, Thread-0 sets *workUpdatedFlag* to false. By this way, if there is no update made to the *Work* vector in the current iteration, flag will remain false and loop will be broken in the next iteration as stated above.

Then, we continue with *Need(i) < Work* comparison. Since these two vectors are of size *1 x numResources* we configured a *checkIfGreater* kernel. Grid and block sizes are the same as described in Algorithm 3. In this kernel, two vectors are compared elementwise in parallel, and a flag is set if there is at least one element of *Need(i)* that is greater than corresponding element of *Work.*

If this *isNeedGreaterThanWork* flag is set, that thread cannot terminate given the current state of the *Work* vector and has to wait for an update to the *Work* vector by the other threads.

If the flag is not set to true, that thread satisfies both *isFinished = false AND Need(i) < Available*. So, that thread can terminate. Hence, we need to add resources allocated to that process to the *Work* vector, i.e. we need to perform *Work = Work + Allocation(i)*.

In this case, we will again do an elementwise summation of two *1 x numResources* vectors. In order to parallelize this operation, we create a kernel called *addVectorsAtomic* which is pretty similar to *addVectors* kernel in Algorithm 4. But this time, since *Work* vector can be modified by numerous threads, we perform an atomic add operation in order to prevent race conditions. This guarantees that all threads that are able to terminate in an iteration will transfer their allocated resources to the *Work* vector successfully. We set *isFinished* flag of terminated processes to true. We also set *workUpdatedFlag* to true in order to notify other unfinished threads that there has been an update to the *Work* vector.

We perform a *\_\_syncthreads()* operation and start with the next iteration.

As discussed in the IV-C-3, when while loop is terminated for all threads, value of the *workUpdatedFlag* tells us whether the system is in a safe state or not. This information is passed to the host by setting *isSafe* device flag and copying it back to host.

Acknowledgment

The preferred spelling of the word “acknowledgment” in American English is without an “e” after the “g.” Use the singular heading even if you have many acknowledgments. Avoid expressions such as “One of us (S.B.A.) would like to thank ... .” Instead, write “F. A. Author thanks ... .” In most cases, sponsor and financial support acknowledgments are placed in the unnumbered footnote on the first page, not here.

References and Footnotes

## References

References need not be cited in text. When they are, they appear on the line, in square brackets, inside the punctuation. Multiple references are each numbered with separate brackets. When citing a section in a book, please give the relevant page numbers. In text, refer simply to the reference number. Do not use “Ref.” or “reference” except at the beginning of a sentence: “Reference [3] shows ... .” Please do not use automatic endnotes in *Word*, rather, type the reference list at the end of the paper using the “References” style.

Reference numbers are set flush left and form a column of their own, hanging out beyond the body of the reference. The reference numbers are on the line, enclosed in square brackets. In all references, the given name of the author or editor is abbreviated to the initial only and precedes the last name. Use them all; use *et al*. only if names are not given. Use commas around Jr., Sr., and III in names. Abbreviate conference titles. When citing IEEE transactions, provide the issue number, page range, volume number, year, and/or month if available. When referencing a patent, provide the day and the month of issue, or application. References may not include all information; please obtain and include relevant information. Do not combine references. There must be only one reference with each number. If there is a URL included with the print reference, it can be included at the end of the reference.

Other than books, capitalize only the first word in a paper title, except for proper nouns and element symbols. For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation See the end of this document for formats and examples of common references. For a complete discussion of references and their formats, see the IEEE style manual at [www.ieee.org/authortools](http://www.ieee.org/authortools).

## Footnotes

Number footnotes separately in superscripts (Insert | Footnote).[[2]](#footnote-2) Place the actual footnote at the bottom of the column in which it is cited; do not put footnotes in the reference list (endnotes). Use letters for table footnotes (see Table I).

# Submitting Your Paper for Review

## Review Stage Using Word 6.0 or Higher

If you want to submit your file with one column electronically, please do the following:

--First, click on the View menu and choose Print Layout.

--Second, place your cursor in the first paragraph. Go to the Format menu, choose Columns, choose one column Layout, and choose “apply to whole document” from the dropdown menu.

--Third, click and drag the right margin bar to just over 4 inches in width.

The graphics will stay in the “second” column, but you can drag them to the first column. Make the graphic wider to push out any text that may try to fill in next to the graphic.

## Final Stage Using Word 6.0

When you submit your final version (after your paper has been accepted), print it in two-column format, including figures and tables. You must also send your final manuscript on a disk, via e-mail, or through a Web manuscript submission system as directed by the society contact. You may use *Zip* for large files, or compress files using *Compress, Pkzip, Stuffit,* or *Gzip.*

Also, send a sheet of paper or PDF with complete contact information for all authors. Include full mailing addresses, telephone numbers, fax numbers, and e-mail addresses. This information will be used to send each author a complimentary copy of the journal in which the paper appears. In addition, designate one author as the “corresponding author.” This is the author to whom proofs of the paper will be sent. Proofs are sent to the corresponding author only.

## Review Stage Using ScholarOne® Manuscripts

Contributions to the Transactions, Journals, and Letters may be submitted electronically on IEEE’s on-line manuscript submission and peer-review system, ScholarOne® Manuscripts. You can get a listing of the publications that participate in ScholarOneat

<http://www.ieee.org/publications_standards/publications/authors/authors_submission.html> First check if you have an existing account. If there is none, please create a new account. After logging in, go to your Author Center and click “Submit First Draft of a New Manuscript.”

Along with other information, you will be asked to select the subject from a pull-down list. Depending on the journal, there are various steps to the submission process; you must complete all steps for a complete submission. At the end of each step you must click “Save and Continue”; just uploading the paper is not sufficient. After the last step, you should see a confirmation that the submission is complete. You should also receive an e-mail confirmation. For inquiries regarding the submission of your paper on ScholarOne Manuscripts, please contact oprs-support@ieee.org or call +1 732 465 5861.

ScholarOne Manuscripts will accept files for review in various formats. Please check the guidelines of the specific journal for which you plan to submit.

You will be asked to file an electronic copyright form immediately upon completing the submission process (authors are responsible for obtaining any security clearances). Failure to submit the electronic copyright could result in publishing delays later. You will also have the opportunity to designate your article as “open access” if you agree to pay the IEEE open access fee.

## Final Stage Using ScholarOne Manuscripts

Upon acceptance, you will receive an email with specific instructions regarding the submission of your final files. To avoid any delays in publication, please be sure to follow these instructions. Most journals require that final submissions be uploaded through ScholarOne Manuscripts, although some may still accept final submissions via email. Final submissions should include source files of your accepted manuscript, high quality graphic files, and a formatted pdf file. If you have any questions regarding the final submission process, please contact the administrative contact for the journal.

In addition to this, upload a file with complete contact information for all authors. Include full mailing addresses, telephone numbers, fax numbers, and e-mail addresses. Designate the author who submitted the manuscript on ScholarOne Manuscripts as the “corresponding author.” This is the only author to whom proofs of the paper will be sent.

## Copyright Form

Authors must submit an electronic IEEE Copyright Form (eCF) upon submitting their final manuscript files.  You can access the eCF system through your manuscript submission system or through the Author Gateway. You are responsible for obtaining any necessary approvals and/or security clearances. For additional information on intellectual property rights, visit the IEEE Intellectual Property Rights department web page at

<http://www.ieee.org/publications_standards/publications/rights/index.html>.

# IEEE Publishing Policy

The general IEEE policy requires that authors should only submit original work that has neither appeared elsewhere for publication, nor is under review for another refereed publication. The submitting author must disclose all prior publication(s) and current submissions when submitting a manuscript. Do not publish “preliminary” data or results. The submitting author is responsible for obtaining agreement of all coauthors and any consent required from employers or sponsors before submitting an article. The IEEE Transactions and Journals Department strongly discourages courtesy authorship; it is the obligation of the authors to cite only relevant prior work.

The IEEE Transactions and Journals Department does not publish conference records or proceedings, but can publish articles related to conferences that have undergone rigorous peer review. Minimally, two reviews are required for every article submitted for peer review.

# Publication Principles

The two types of contents of that are published are; 1) peer-reviewed and 2) archival. The Transactions and Journals Department publishes scholarly articles of archival value as well as tutorial expositions and critical reviews of classical subjects and topics of current interest.

Authors should consider the following points:

1. Technical papers submitted for publication must advance the state of knowledge and must cite relevant prior work.
2. The length of a submitted paper should be commensurate with the importance, or appropriate to the complexity, of the work. For example, an obvious extension of previously published work might not be appropriate for publication or might be adequately treated in just a few pages.
3. Authors must convince both peer reviewers and the editors of the scientific and technical merit of a paper; the standards of proof are higher when extraordinary or unexpected results are reported.
4. Because replication is required for scientific progress, papers submitted for publication must provide sufficient information to allow readers to perform similar experiments or calculations and use the reported results. Although not everything need be disclosed, a paper must contain new, useable, and fully described information. For example, a specimen’s chemical composition need not be reported if the main purpose of a paper is to introduce a new measurement technique. Authors should expect to be challenged by reviewers if the results are not supported by adequate data and critical details.
5. Papers that describe ongoing work or announce the latest technical achievement, which are suitable for presentation at a professional conference, may not be appropriate for publication.

References

*Basic format for books:*

J. K. Author, “Title of chapter in the book,” in *Title of His Published Book, x*th ed. City of Publisher, (only U.S. State), Country: Abbrev. of Publisher, year, ch. *x*, sec. *x*, pp. *xxx–xxx.*

*Examples:*

1. G. O. Young, “Synthetic structure of industrial plastics,” in *Plastics,* 2nd ed., vol. 3, J. Peters, Ed. New York, NY, USA: McGraw-Hill, 1964, pp. 15–64.
2. W.-K. Chen, *Linear Networks and Systems.* Belmont, CA, USA: Wadsworth, 1993, pp. 123–135.

*Basic format for periodicals:*

J. K. Author, “Name of paper,” *Abbrev. Title of Periodical*, vol. *x, no*. *x,* pp*. xxx-xxx,* Abbrev. Month, year, DOI. 10.1109.*XXX*.123456.

*Examples:*

1. J. U. Duncombe, “Infrared navigation—Part I: An assessment of feasibility,” *IEEE Trans. Electron Devices*, vol. ED-11, no. 1, pp. 34–39, Jan. 1959, 10.1109/TED.2016.2628402.
2. E. P. Wigner, “Theory of traveling-wave optical laser,”   
   *Phys. Rev*.,   
   vol. 134, pp. A635–A646, Dec. 1965.
3. E. H. Miller, “A note on reflector arrays,” *IEEE Trans. Antennas Propagat*., to be published.

*Basic format for reports:*

J. K. Author, “Title of report,” Abbrev. Name of Co., City of Co., Abbrev. State, Country, Rep. *xxx*, year.

*Examples:*

1. E. E. Reber, R. L. Michell, and C. J. Carter, “Oxygen absorption in the earth’s atmosphere,” Aerospace Corp., Los Angeles, CA, USA, Tech. Rep. TR-0200 (4230-46)-3, Nov. 1988.
2. J. H. Davis and J. R. Cogdell, “Calibration program for the 16-foot antenna,” Elect. Eng. Res. Lab., Univ. Texas, Austin, TX, USA, Tech. Memo. NGL-006-69-3, Nov. 15, 1987.

*Basic format for handbooks:*

*Name of Manual/Handbook, x* ed., Abbrev. Name of Co., City of Co., Abbrev. State, Country, year, pp. *xxx-xxx.*

*Examples:*

1. *Transmission Systems for Communications*, 3rd ed., Western Electric Co., Winston-Salem, NC, USA, 1985, pp. 44–60.
2. *Motorola Semiconductor Data Manual*, Motorola Semiconductor Products Inc., Phoenix, AZ, USA, 1989.

*Basic format for books (when available online):*

J. K. Author, “Title of chapter in the book,” in *Title of Published Book*, *x*th ed. City of Publisher, State, Country: Abbrev. of Publisher, year, ch. *x*, sec. *x*, pp. *xxx–xxx*. [Online]. Available: http://www.web.com

*Examples:*

1. G. O. Young, “Synthetic structure of industrial plastics,” in Plastics, vol. 3, Polymers of Hexadromicon, J. Peters, Ed., 2nd ed. New York, NY, USA: McGraw-Hill, 1964, pp. 15-64. [Online]. Available: http://www.bookref.com.
2. *The Founders’ Constitution*, Philip B. Kurland and Ralph Lerner, eds., Chicago, IL, USA: Univ. Chicago Press, 1987. [Online]. Available: http://press-pubs.uchicago.edu/founders/
3. The Terahertz Wave eBook. ZOmega Terahertz Corp., 2014. [Online]. Available: http://dl.z-thz.com/eBook/zomega\_ebook\_pdf\_1206\_sr.pdf. Accessed on: May 19, 2014.
4. Philip B. Kurland and Ralph Lerner, eds., *The Founders’ Constitution.* Chicago, IL, USA: Univ. of Chicago Press, 1987, Accessed on: Feb. 28, 2010, [Online] Available: http://press-pubs.uchicago.edu/founders/

*Basic format for journals (when available online):*

J. K. Author, “Name of paper,” *Abbrev. Title of Periodical*, vol. *x*, no. *x*, pp. *xxx-xxx*, Abbrev. Month, year. Accessed on: Month, Day, year, DOI: 10.1109.*XXX*.123456, [Online].

*Examples:*

1. J. S. Turner, “New directions in communications,” *IEEE J. Sel. Areas Commun*., vol. 13, no. 1, pp. 11-23, Jan. 1995.
2. W. P. Risk, G. S. Kino, and H. J. Shaw, “Fiber-optic frequency shifter using a surface acoustic wave incident at an oblique angle,” *Opt. Lett.*, vol. 11, no. 2, pp. 115–117, Feb. 1986.
3. P. Kopyt *et al., “*Electric properties of graphene-based conductive layers from DC up to terahertz range,” *IEEE THz Sci. Technol.,* to be published. DOI: 10.1109/TTHZ.2016.2544142.

*Basic format for papers presented at conferences (when available online):*

J.K. Author. (year, month). Title. presented at abbrev. conference title. [Type of Medium]. Available: site/path/file

*Example:*

1. PROCESS Corporation, Boston, MA, USA. Intranets: Internet technologies deployed behind the firewall for corporate productivity. Presented at INET96 Annual Meeting. [Online]. Available: http://home.process.com/Intranets/wp2.htp

*Basic format for reports and handbooks (when available online):*

J. K. Author. “Title of report,” Company. City, State, Country. Rep. no., (optional: vol./issue), Date. [Online] Available: site/path/file

*Examples:*

1. R. J. Hijmans and J. van Etten, “Raster: Geographic analysis and modeling with raster data,” R Package Version 2.0-12, Jan. 12, 2012. [Online]. Available: http://CRAN.R-project.org/package=raster
2. Teralyzer. Lytera UG, Kirchhain, Germany [Online]. Available: http://www.lytera.de/Terahertz\_THz\_Spectroscopy.php?id=home, Accessed on: Jun. 5, 2014

*Basic format for computer programs and electronic documents (when available online):*

Legislative body. Number of Congress, Session. (year, month day). *Number of bill or resolution*, *Title*. [Type of medium]. Available: site/path/file

***NOTE:*** ISO recommends that capitalization follow the accepted practice for the language or script in which the information is given.

*Example:*

1. U.S. House. 102nd Congress, 1st Session. (1991, Jan. 11). *H. Con. Res. 1, Sense of the Congress on Approval of Military Action*. [Online]. Available: LEXIS Library: GENFED File: BILLS

*Basic format for patents (when available online):*

Name of the invention, by inventor’s name. (year, month day). Patent Number[Type of medium]. Available: site/path/file

*Example:*

1. Musical toothbrush with mirror, by L.M.R. Brooks. (1992, May 19). Patent D 326 189

[Online]. Available: NEXIS Library: LEXPAT File: DES

*Basic format for conference proceedings (published):*

J. K. Author, “Title of paper,” in *Abbreviated Name of Conf.*, City of Conf., Abbrev. State (if given), Country, year, pp. *xxxxxx.*

*Example:*

1. D. B. Payne and J. R. Stern, “Wavelength-switched pas- sively coupled single-mode optical network,” in *Proc. IOOC-ECOC,* Boston, MA, USA,1985,   
   pp. 585–590.

*Example for papers presented at conferences (unpublished):*

1. D. Ebehard and E. Voges, “Digital single sideband detection for interferometric sensors,” presented at the *2nd Int. Conf. Optical Fiber Sensors,* Stuttgart, Germany, Jan. 2-5, 1984.

*Basic format for patents:*

J. K. Author, “Title of patent,” U.S. Patent *x xxx xxx*, Abbrev. Month, day, year.

*Example:*

1. G. Brandli and M. Dick, “Alternating current fed power supply,” U.S. Patent 4 084 217, Nov. 4, 1978.

*Basic format**for theses (M.S.) and dissertations (Ph.D.):*

a) J. K. Author, “Title of thesis,” M.S. thesis, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

b) J. K. Author, “Title of dissertation,” Ph.D. dissertation, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

*Examples:*

1. J. O. Williams, “Narrow-band analyzer,” Ph.D. dissertation, Dept. Elect. Eng., Harvard Univ., Cambridge, MA, USA, 1993.
2. N. Kawasaki, “Parametric study of thermal and chemical nonequilibrium nozzle flow,” M.S. thesis, Dept. Electron. Eng., Osaka Univ., Osaka, Japan, 1993.

*Basic format for the most common types of unpublished references:*

a) J. K. Author, private communication, Abbrev. Month, year.

b) J. K. Author, “Title of paper,” unpublished.

c) J. K. Author, “Title of paper,” to be published.

*Examples:*

1. A. Harrison, private communication, May 1995.
2. B. Smith, “An approach to graphs of linear forms,” unpublished.
3. A. Brahms, “Representation error for real numbers in binary computer arithmetic,” IEEE Computer Group Repository, Paper R-67-85.

*Basic formats for standards:*

a) *Title of Standard*, Standard number, date.

b) *Title of Standard*, Standard number, Corporate author, location, date.

*Examples:*

1. IEEE Criteria for Class IE Electric Systems, IEEE Standard 308, 1969.
2. Letter Symbols for Quantities, ANSI Standard Y10.5-1968.

*Article number in reference examples:*

1. R. Fardel, M. Nagel, F. Nuesch, T. Lippert, and A. Wokaun, “Fabrication of organic light emitting diode pixels by laser-assisted forward transfer,” *Appl. Phys. Lett.*, vol. 91, no. 6, Aug. 2007, Art. no. 061103.
2. J. Zhang and N. Tansu, “Optical gain and laser characteristics of InGaN quantum wells on ternary InGaN substrates,” *IEEE Photon. J.*, vol. 5, no. 2, Apr. 2013, Art. no. 2600111

*Example when using et al.:*

1. S. Azodolmolky *et al.*, Experimental demonstration of an impairment aware network planning and operation tool for transparent/translucent optical networks,” *J. Lightw. Technol.*, vol. 29, no. 4, pp. 439–448, Sep. 2011.

**First A. Author** (M’76–SM’81–F’87) and all authors may include biographies. Biographies are often not included in conference-related papers. This author became a Member (M) of IEEE in 1976, a Senior Member (SM) in 1981, and a Fellow (F) in 1987. The first paragraph may contain a place and/or date of birth (list place, then date). Next, the author’s educational background is listed. The degrees should be listed with type of degree in what field, which institution, city, state, and country, and year the degree was earned. The author’s major field of study should be lower-cased.

The second paragraph uses the pronoun of the person (he or she) and not the author’s last name. It lists military and work experience, including summer and fellowship jobs. Job titles are capitalized. The current job must have a location; previous positions may be listed without one. Information concerning previous publications may be included. Try not to list more than three books or published articles. The format for listing publishers of a book within the biography is: title of book (publisher name, year) similar to a reference. Current and previous research interests end the paragraph.

The third paragraph begins with the author’s title and last name (e.g., Dr. Smith, Prof. Jones, Mr. Kajor, Ms. Hunter). List any memberships in professional societies other than the IEEE. Finally, list any awards and work for IEEE committees and publications. If a photograph is provided, it should be of good quality, and professional-looking. Following are two examples of an author’s biography.

**Second B. Author** was born in Greenwich Village, New York, NY, USA in 1977. He received the B.S. and M.S. degrees in aerospace engineering from the University of Virginia, Charlottesville, in 2001 and the Ph.D. degree in mechanical engineering from Drexel University, Philadelphia, PA, in 2008.

From 2001 to 2004, he was a Research Assistant with the Princeton Plasma Physics Laboratory. Since 2009, he has been an Assistant Professor with the Mechanical Engineering Department, Texas A&M University, College Station. He is the author of three books, more than 150 articles, and more than 70 inventions. His research interests include high-pressure and high-density nonthermal plasma discharge processes and applications, microscale plasma discharges, discharges in liquids, spectroscopic diagnostics, plasma propulsion, and innovation plasma applications. He is an Associate Editor of the journal *Earth*, *Moon*, *Planets*, and holds two patents.

Dr. Author was a recipient of the International Association of Geomagnetism and Aeronomy Young Scientist Award for Excellence in 2008, and the IEEE Electromagnetic Compatibility Society Best Symposium Paper Award in 2011.

**Third C. Author, Jr.** (M’87) received the B.S. degree in mechanical engineering from National Chung Cheng University, Chiayi, Taiwan, in 2004 and the M.S. degree in mechanical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2006. He is currently pursuing the Ph.D. degree in mechanical engineering at Texas A&M University, College Station, TX, USA.

From 2008 to 2009, he was a Research Assistant with the Institute of Physics, Academia Sinica, Tapei, Taiwan. His research interest includes the development of surface processing and biological/medical treatment techniques using nonthermal atmospheric pressure plasmas, fundamental study of plasma sources, and fabrication of micro- or nanostructured surfaces.

Mr. Author’s awards and honors include the Frew Fellowship (Australian Academy of Science), the I. I. Rabi Prize (APS), the European Frequency and Time Forum Award, the Carl Zeiss Research Award, the William F. Meggers Award and the Adolph Lomb Medal (OSA).

1. [↑](#footnote-ref-1)
2. It is recommended that footnotes be avoided (except for the unnumbered footnote with the receipt date on the first page). Instead, try to integrate the footnote information into the text. [↑](#footnote-ref-2)