CUDA Implementation of Banker's Algorithm for Deadlock Avoidance

Enes Muvahhid ŞAHİN, Middle East Technical University, Electrical and Electronics Engineering
enes.sahin@metu.edu.tr

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-32x speed improvement compared to sequential/partially-parallel CPU implementations.**

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

Note: Please refer to README.md for guidelines about execution of the provided codes.

I. Motivation & Significance

DEADLOCK 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. We also explain our sequential and partially-parallel CPU implementations in this section. In Section V, we provide experimental results and comparisons of CUDA implementation, sequential CPU implementation and partially parallel CPU implementation. Furthermore, we provide and discuss the profiling results of the GPU implementation. In Section VI, we make some discussions and conclusions about our work, state its powerful and weak aspects, and provide some ideas about the future work.

II. 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.

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

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:

```
Algorithm 1: Banker's Algorithm
m = Number of resource types
n = Number of processes
J = RequestingProcessID
if Request[i] < Need[J, i] for all i in (0, m-1) then
    if Request[i] < Available[i] for all i in (0, m-1) then
        for i in (0, m-1) do
            /* Modify data structures as if the allocation is made.      */
            Available[i] = Available[i] − Request[i]
            Allocation[J, i] = Allocation[J, i] + Request[i]
            Need[J, i] = Need[J, i] − Request[i]
        end
        bool isSafe = Safety_Algorithm()
        if isSafe then
            Requested allocation CAN BE SERVED.
        else
            Requested allocation may lead to Deadlock.
            It CANNOT BE SERVED.
        end
    else
        Requested allocation IS NOT valid.
    end
else
    Requested allocation IS NOT valid.
end
```

Detailed flow of Safety Algorithm is as follows:

```
Algorithm 2: Safety Algorithm
m = Number of resource types
n = Number of processes
Work: m x 1 matrix
Finish: n x 1 matrix
Function Safety_Algorithm(Available, Need, Allocation):
    Step-1:                         // Initialize Work and Finish matrices
    for i in (0, m-1) do
        Work[i] = Available[i]
    end
    for j in (0, n-1) do
        Finish[j] = false
    end

    Step-2:
    Find a j such that:
        Finish[j] = false AND Need[j, i] < Work[i] for all i in (0, m-1)
    If no such j is found, go to Step-4.

    Step-3:
    for i in (0, m-1) do
        Work[i] = Work[i] + Allocation[j, i]
        Finish[j] = true
    end
    Go to Step-2.

    Step-4:
    if Finish[j] == true for all j in (0, n-1) then
        return true
    else
        return false
    end
```

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 modify 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 finish.

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.

III. 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.

IV. 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.

<u>***Notation:***</u>

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

- i^{th} row vector of X, if X is an n x m matrix
- Vector itself, if X is a 1 x m vector.

Request$_i$ ≤ Need(i) means that each element of *Request$_i$*

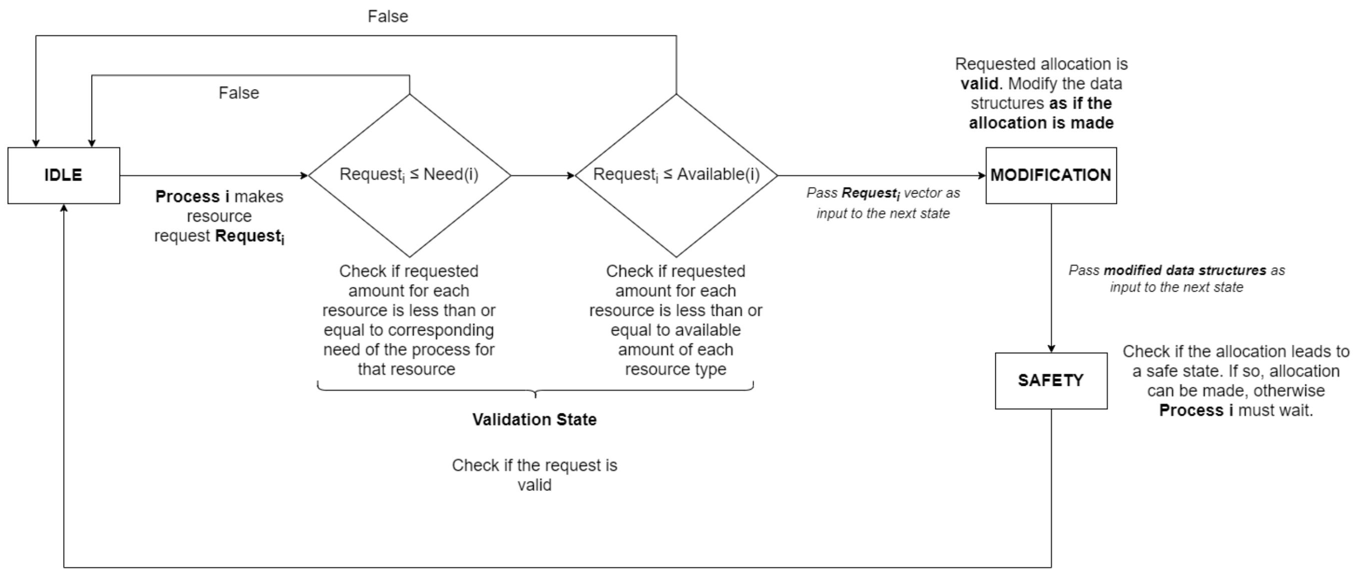

False
False
Requested allocation is valid. Modify the data structures as if the allocation is made
IDLE
Process i makes resource request Request_i
Request_i ≤ Need(i)
Request_i ≤ Available(i)
MODIFICATION
Pass Request_i vector as input to the next state
Check if requested amount for each resource is less than or equal to corresponding need of the process for that resource
Check if requested amount for each resource is less than or equal to available amount of each resource type
Pass modified data structures as input to the next state
SAFETY
Check if the allocation leads to a safe state. If so, allocation can be made, otherwise Process i must wait.
Validation State
Check if the request is valid

Figure 1: Overall block diagram of the algorithm

vector is less than or equal to corresponding element of ith row of *Need* matrix where *Request$_i$* is the requested allocation vector for process with *requestingProcessId = i.*

X(i) = X(i) + Request$_i$ means that each element of *Request$_i$* is added to each corresponding element of ith row of X if X is a matrix, or added to each corresponding element of X if X is a vector with same dimension as *Request$_i$*.

Other notations are similar to these three notations.

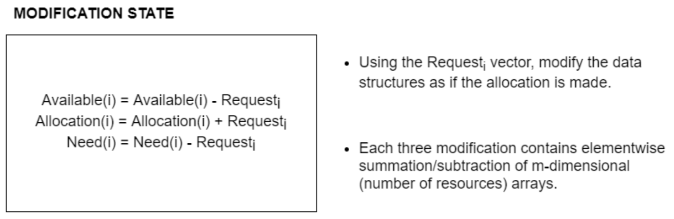

MODIFICATION STATE
Available(i) = Available(i) - Request_i
Allocation(i) = Allocation(i) + Request_i
Need(i) = Need(i) - Request_i
Using the Request_i vector, modify the data structures as if the allocation is made.
Each three modification contains elementwise summation/subtraction of m-dimensional (number of resources) arrays.

Figure 2: Block diagram of modification state

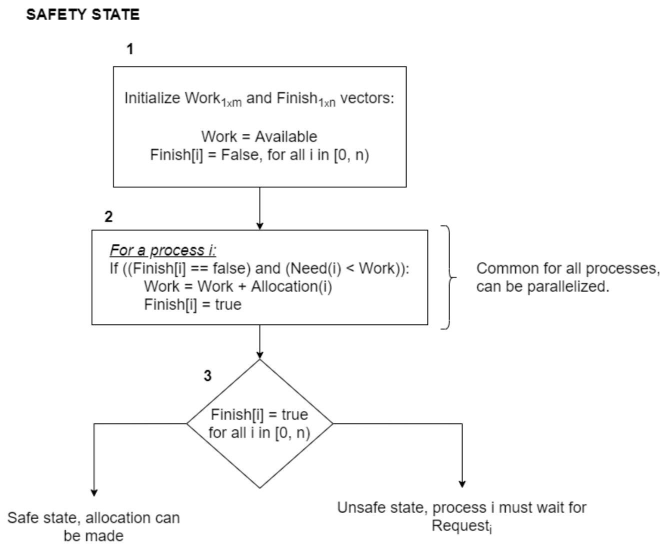

SAFETY STATE
1
Initialize Work_1xm and Finish_1xn vectors:
Work = Available
Finish[i] = False, for all i in [0, n)
2
For a process i:
If ((Finish[i] == false) and (Need(i) < Work)):
Work = Work + Allocation(i)
Finish[i] = true
Common for all processes, can be parallelized.
3
Finish[i] = true for all i in [0, n)
Safe state, allocation can be made
Unsafe state, process i must wait for Request_i

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.

Note: **Max** matrix is not used throughout the algorithm, but it would be probably used by the operating system in a real-life application in order to store each process' maximum needs of each resource type. So, in all implementations, **Max** matrices are initialized, but not used.

A. SINGLE-THREADED CPU IMPLEMENTATION

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

1) 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.

2) 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 *Request*$_i$ vector. Then, we continue with Safety State in order to check whether the new states of matrices can cause a deadlock or not.

3) <u>Safety State</u>

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 the 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.

B. 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 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.

1) <u>Validation State</u>

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.

2) <u>Modification State</u>

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:

<u>*Thread modifyAvailable*</u>*: Available = Available – Request*
<u>*Thread modifyAllocation:*</u> *Allocation = Allocation + Request*
<u>*Thread modifyNeed:*</u> *Need = Need - Request*

3) <u>Safety State</u>

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.

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

*C. **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 designed 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.

1) <u>Validation State</u>

In the validation state, we need to check followings:

- *Request*$_i$ *< Need(i)*
- *Request*$_i$ *< 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 *Request$_i$* 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 *Request$_i$* 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:

Algorithm 3: Check If Greater - GPU Implementation

```
$numThreads = min(numResources, 1024)$
$numBlocks = \lceil \frac{numResources}{numThreads} \rceil$

tid: $ThreadID$
Kernel checkIfGreater($Vector\_A$, $Vector\_B$, $isGreaterThan$):
    /* Set isGreaterThan = true if Vector_A[i] > Vector_B[i] for at least one i. */
    if $Vector\_A[tid] < Vector\_B[tid]$ then
        $isGreaterThan = true$
    end
```

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 the same amount of global memory accesses 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 (no reading operation) is performed with all threads writing *the same value (true)*.

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

2) <u>Modification State</u>

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:

Algorithm 4: Add/Subtract Vectors - GPU Implementation

```
$numThreads = min(numResources, 1024)$
$numBlocks = \lceil \frac{numResources}{numThreads} \rceil$

tid: $ThreadID$
Kernel add/SubtractVectors($Vector\_A$, $Vector\_B$):
    /* In place, elementwise addition/subtraction of Vector_A and Vector_B. */
    $Vector\_A[tid] = Vector\_A[tid] \pm Vector\_B[tid]$
```

3) <u>Safety State</u>

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 reasons behind this and other implementation decisions are discussed at the end of this section.

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:

Algorithm 5: Safety State - GPU Implementation

```
$numThreads = min(numProcesses, 1024)$
$numBlocks = 1$

tid: $ThreadID$
workUpdatedFlag = true
Kernel safetyState($Work$, $Need$, $Allocation$, $isSafe$):
    isFinished = false
    repeat
        /* Set flag to false at each iteration */
        workUpdatedFlag == false
        checkIfGreater($Need(i)$, $Work$, $isNeedGreaterThanWork$)
        if $!isNeedGreaterThanWork$ then
            /* Need(i) < Work, so this process can finish. */
            /* Transfer allocated resources to the Work */
            addVectorsAtomic($Work$, $Allocation(i)$)   // Update Work
            isFinished = true                        // This process has finished.
            workUpdatedFlag = true                   // Work is updated.
        end
    until $isFinished == true$ OR $workUpdatedFlag == false$
    if $tid == 0$ then
        if $workUpdatedFlag == true$ then
            isSafe = true
        else
            isSafe = false
        end
    end
```

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.***
3 possible scenarios for the status of this flag is as follows:

DATASET	numProcesses	numResources	Test Cases
Dataset 1	1024	1024	1. Validation step fails in the first element of the *Request* vector 2. Validation step fails in the middle element of the *Request* vector 3. Validation step fails in the last element of the *Request* vector 4. Validation is successful, Safety step fails. 5. Validation and Safety steps are successful, allocation can be made.
Dataset 2	256	256	1. Validation step fails in the middle element of the *Request* vector. 2. Validation is successful, Safety step fails. 3. Validation and Safety steps are successful, allocation can be made.
Dataset 3	1024	10240	1. Validation step fails in the middle element of the *Request* vector. 2. Validation is successful, Safety step fails. 3. Validation and Safety steps are successful, allocation can be made.
Dataset 4	128	1024	1. Validation step fails in the middle element of the *Request* vector. 2. Validation is successful, Safety step fails. 3. Validation and Safety steps are successful, allocation can be made.
Dataset 5	1024	128	1. Validation step fails in the middle element of the *Request* vector. 2. Validation is successful, Safety step fails. 3. Validation and Safety steps are successful, allocation can be made.
Simple Example	5	3	1. Validation and Safety steps are successful, allocation can be made.

Table 1: Datasets used in the experiments, their properties and test cases

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, *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 (CUDA thread) can launch new kernels in order to perform above operations in parallel. Launched kernels have one block and *numResources* sub-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* sub-threads per process (CUDA 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.

Having discussed our implementations and the reasons behind choosing certain paths in those implementations, we provide extensive experimental results and comparisons of these implementations in Section V.

V. Experimental Results

Our main goal while implementing Banker's Algorithm in CUDA was to provide a feasible solution for systems with large number of processes and resource types. In this section we will benchmark all of the discussed 3 implementations, which are GPU implementation, sequential CPU implementation and partially-parallel CPU implementation under varying input sizes (number of processes and number of resources).

In order to generate large number of data structures, we created a python script (*input_generator.py*) to randomly generate elements of matrices.

There are 5 different datasets that we utilize during the experiments in order to benchmark different aspects of the algorithm and implementations. Dataset descriptions and test cases performed on those datasets are given in **Table 1**. There are different number of test cases. For each test case, generated data structures are separate. So, test cases are like sub-datasets. For example, in Dataset 1 – Test Case 1, Validation step will fail due to first element of the *Request* vector being larger than first element of *Available* vector.

In the first dataset and its test cases, our aim is to investigate how values of the matrices affect the performances of different implementations. For instance, in Test Cases 1-3, we basically benchmark failure of Validation State given that failure is due to the element in the beginning (1), middle (2), end (3) of the *Request* vector. For sequential implementations, this may affect performance.
In the remaining test cases of Dataset 1, we basically benchmark how fast an implementation detects an unsafe state (4), and how fast it determines a state is safe (5).

Test cases for the remaining datasets are basically the same as Test Case – 2, Test Case – 4, and Test Case – 5 of Dataset 1. The purpose of these datasets is to compare how number of processes and number of resources affect the performances of different implementations.

In Dataset 2, we simulate a system with relatively small number of resources and processes.

In Dataset 3, we simulate a system with maximum possible processes allowed in CUDA implementation (1024) and a huge number of resources.

In Dataset 4 and 5 we try to analyze whether the performance of the Banker's Algorithm and different implementations are bounded by number of processes or number of resources given that $numProcesses \times numResources$ product is constant.

Simple Example dataset is taken from [6], where step by step traversal of the Banker's Algorithm is also provided in the website. This dataset is very small, there are 5 process and 3 resource types. Also provided request leads to a safe state and it is feasible. Note that provided safe sequence in [6] may differ from our implementations since in the Safety State we consider processes from the beginning (*processId = 0*) at each iteration. On the contrary, in [6] they consider processes till the end (*processId = 4*), then they start from the beginning. Moreover, safe sequence between GPU and CPU implementations may also differ since in GPU implementation Safety State is parallelized. It is undetermined which process will finish first, which will finish second, etc. But the result of the Banker's Algorithm, whether a request is feasible or not, is the same for all implementations.

More detailed explanation about how we created the datasets is given in the **Appendix B**.

Our test environment during these experiments are given in Table 2.

GPU	GTX 1060
CPU	INTEL I7-6700 HQ
OS	WINDOWS 10
CUDA VERSION	10.1
IDE	VISUAL STUDIO 2019 (built-in compiler)

Table 2: Experiment Environment

We have run all 3 implementations in all test cases of all datasets. We measured the execution time for each implementation and dataset (test case). <u>Measurements are performed as follows:</u>

For the CPU implementations, we measured the time from the beginning of the Validation State until the end of the algorithm where decision of whether the allocation is feasible or not is made.

For the GPU implementation, we measured the time from the beginning of device memory allocations until the end of the algorithm where decision of whether the allocation is feasible or not is made by transferring the results from device to host (returning allocated memory is also included).

To measure the execution time, we used *QueryPerformanceCounter* for high precision timing of CPU. In the GPU implementation, we measured the time using this counter from the beginning of first CUDA call until the end of the program. This is OK because we first copy results from device to host, perform *cudaDeviceSynchronize()*, then call *cudaFree()* in order to return allocated memory to the system. After that we measure the end time. Since *cudaFree()* is synchronous, measured quantity represents the whole execution time of the program.

For each dataset (test case) we run the algorithm 10 times in a loop and took the average execution time. In all measurements, execution time includes memory allocation and deallocation steps. Namely, we allocate and deallocate memory at each iteration as if the program starts executing from scratch.

We do not include reading time of the matrices from the text files in measurements. We also do not re-read the matrices from the disk at each iteration. They are loaded just once to the host memory. GPU implementation allocates space on the device memory and copies those matrices from host to device at each iteration.

Experiment results are given in Table 3.

<u>Note:</u> For the GPU implementation, first execution time takes too long. Executions afterwards do not take that much. This is due to loading/creation of libraries/contexts in the first execution of the program. This affects average execution time severely. Given measurements for the GPU implementation in Table 3 represents average execution time where first execution time is not taken into account. The values in the parenthesis next to them represents the overall average execution time where first execution is also considered. Since in a real-life scenario Banker's Algorithm would be an always-active program, effect of the first execution would be negligible. So, it is better to ignore the first execution while performing benchmarks.

<u>Following conclusions can be made from the experimental results provided in Table 3:</u>

D1_TC1, D1_TC2, D1_TC3 results show that Validation step isn't severely affected by the position of the invalid element in the request vector. We can also see that GPU implementation is **2x** faster than the CPU implementations.
D1_TC4 and D1_TC5 represents the datasets where Safety Step fails for just one process and Safety step is successful,

	D1_TC1 (ms)	D1_TC2 (ms)	D1_TC3 (ms)	D1_TC4 (ms)	D1_TC5 (ms)	D2_TC1 (ms)	D2_TC2 (ms)	D2_TC3 (ms)	D3_TC1 (ms)
GPU	6.428 (52.433)	6.224 (29.938)	7.391 (28.356)	29.066 (52.411)	29.173 (52.490)	3.302 (25.908)	7.957 (33.544)	7.985 (33.307)	35.967 (59.901)
CPU (single-threaded)	12.546	11.679	12.62	639.294	652.045	4.974	48.316	47.951	80.986
CPU (multi-threaded)	15.445	15.064	14.964	652.848	659.702	7.174	56.085	55.467	84.604

	D3_TC2 (ms)	D3_TC3 (ms)	D4_TC1 (ms)	D4_TC2 (ms)	D4_TC3 (ms)	D5_TC1 (ms)	D5_TC2 (ms)	D5_TC3 (ms)	Simple Example (ms)
GPU	57.589 (81.537)	58.951 (100.21)	2.383 (24.368)	6.350 (26.887)	6.333 (28.821)	2.176 (23.985)	18.046 (39.785)	17.896 (37.733)	4.508 (23.238)
CPU (single-threaded)	1524.6	1593.9	5.12169	23.610	24.608	5.163	561.597	563.341	1.318
CPU (multi-threaded)	1573.7	1563.6	7.75669	33.624	33.506	7.935	563.01	572.447	6.904

Table 3: Experimental results of all 3 implementations for all datasets. DX_TCY represents Dataset X - Test Case Y. Numbers for the GPU implementation indicates average execution time where first execution is not taken into account whereas numbers in parentheses indicates average execution time where first execution is also taken into account. All values are in milliseconds.

respectively. Results for two datasets are similar and the reason is explained in Appendix B. For these two datasets we can see that GPU implementation if superior to CPU implementations, **22x** faster. Dataset 1 is a moderate dataset with 1024 processes and 1024 resource types.

Dataset 2 is a simpler dataset than Dataset 1 and contains 256 processes and 256 resource types. In D2_TC1 results, we can see that the execution time gap between CPU and GPU implementations are reduced for the Validation step failure case. This is expected because we process a smaller number of elements (128) and CPU can terminate quickly.

D2_TC3 results show that even for systems with small number of processes and resource types, GPU implementation surpasses CPU implementations. CUDA implementation is **7x** faster than single-threaded CPU implementation and **8x** faster than multi-threaded CPU implementation.

In Dataset 3, we tried to benchmark different implementations under a huge dataset where number of processes are at maximum (1024) and number of resource types is 10 times larger than this number, 10240. D3_TC1 results are similar to previous Validation step failure test cases in terms of the execution time gap between GPU and CPU implementations.

The real performance difference between CPU and CUDA implementations are clearly visible in D3_TC3 where a completely feasible request is simulated in this large dataset. For GPU implementation, D1_TC5 took 29 milliseconds. In this test case, there were 1024 different resource types. In D3_TC3 test case, this number is 10 times larger, 10240. However, execution time is just doubled, **58 milliseconds**. When we look at CPU implementations, it took more than **1.5** seconds to process this large dataset. This shows that, in such large systems with huge number of resources, GPU implementation is more than **25x** faster than CPU implementations.

In Dataset 4 and Dataset 5, we tried to analyze whether performance of Banker's Algorithm is more sensitive to number of processes or number of resource types for different implementations. Dataset 4 contains 128 processes and 1024 resource types whereas Dataset 5 contains 1024 processes and 128 resource types.

D4_TC1 and D5_TC1 indicates that validation step is not affected too much from this change. This is expected because we'd observed that Validation step is not heavily affected by the positions of the invalid element during benchmarking with D1_TC1, D1_TC2, D1_TC3.

The actual impact of change in number of processes versus number of resource types is visible in D4_TC3 and D5_TC3. We can see that when we increase number of processes and decrease number of resources in Dataset 5, while keeping product of them constant, execution time is **tripled**. This is expected because there are atomic add operations in the Safety State. Furthermore, inter-thread dependencies take much more time when number of processes (CUDA threads) is larger. But even execution time is tripled, it is still much faster than CPU implementations.

On the contrary, CPU implementations are affected severely when we switch from Dataset 4 to Dataset 5. D4_TC3 and D5_TC3 results show that CPU implementations are **20-23** times slower in Dataset 5, compared to Dataset 4. Moreover D5_TC3 simulations show that, GPU implementation is **32x** faster than CPU implementations.

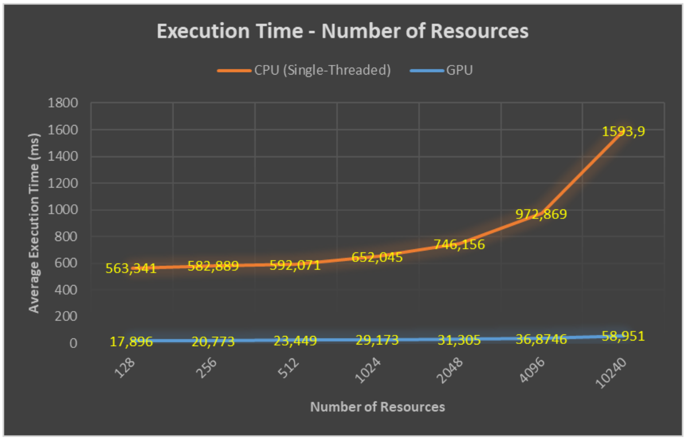


Figure 4: Average Execution Time (ms) vs Number of Resources for GPU and CPU (Single-Threaded) implementations

Finally, in Simple Example dataset, in which there are 5 processes and 3 resource types, single-threaded CPU implementation is faster than CUDA implementation. This dataset is not very realistic, and it is only benchmarked for the sake of completeness. In such a small dataset, overhead of device memory allocations, deallocations, and inter-thread dependencies cause GPU implementation to be slower than single-threaded CPU implementation. Because, it is very cost-free to process such a small dataset sequentially in for loops.

One final thing to note, among CPU implementations, single-threaded CPU implementation is almost always faster than multi-threaded CPU implementation. Only in D3_TC3 we can see that multi-threaded implementation surpassed single-

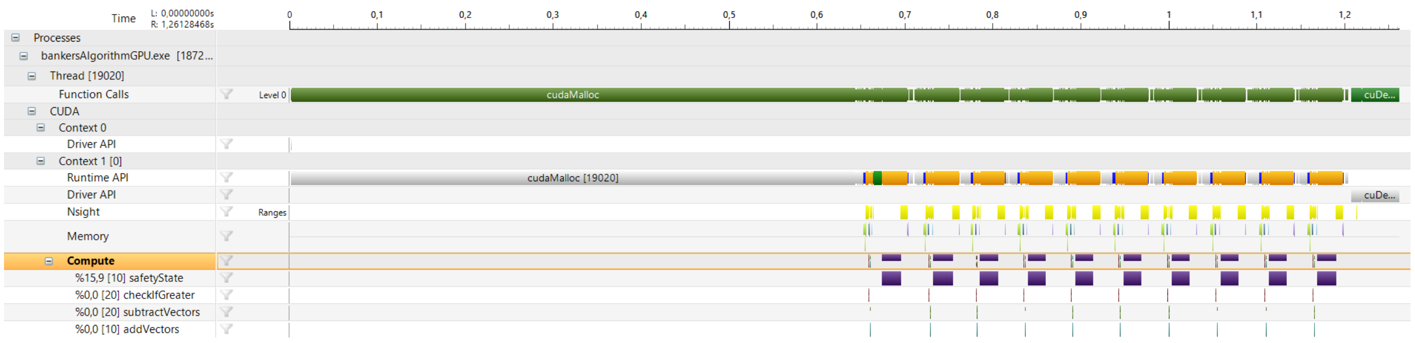
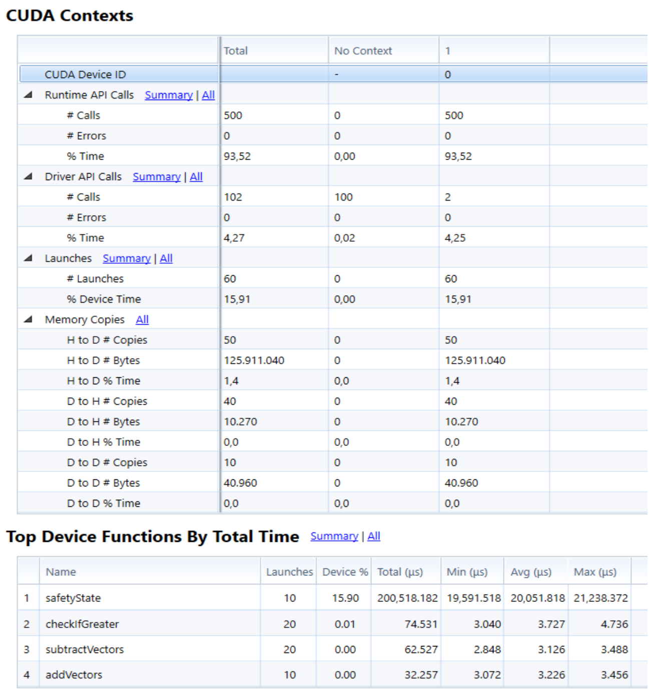


Figure 5: NSIGHT Profiling Timeline for the execution of GPU implementation on D1_TC5 dataset

threaded implementation, but by a very small margin. The reason for single-threaded version being faster than multi-threaded version is bottleneck of the algorithm is Safety State. In multi-threaded version, we parallelized only Validation and Modification states. Thread creation overhead for these two states is larger than performance gain. So, it is not feasible to implement CPU version multi-threaded unless thread creation overheads are lowered and many number of threads can be created and executed without additional overheads (which is not possible currently, since number of cores in CPUs are very few compared to GPUs.)

We also provided a comparison graph in Figure 4. In this experiment, we kept number of processes at max, 1024. Then, we measured execution time of GPU and single-threaded CPU implementations for different number of resource types. All measurements are done with datasets where requested allocation is valid and feasible. The graph shows how quickly the difference between execution times of GPU and CPU implementations grow as we increase number of resource types.

We have provided NSIGHT Profiling Timeline results in Figure 5 for the execution of Dataset 1, Test Case 5. CUDA Summary for this experiment is also given in Figure 6. Execution is done 10 times in a loop as previously explained. As seen from the timeline, most of the time spent in GPU is caused by *cudaMalloc()* calls. In a real-life application, this allocation can be done once, and system can run forever if maximum number of processes and resources can be fixed to a number. By this way, allocation overheads can be avoided.
Purple sections indicate execution of Safety State kernel. This kernel is the most time-consuming kernel among all the others. It constitutes 15.9% of the overall execution time on GPU. This is expected because Safety State is the most computationally expensive part of the algorithm since it contains inter-thread communications and unavoidable loops.



CUDA Contexts

	Total	No Context	1
CUDA Device ID		-	0
Runtime API Calls Summary | All			
# Calls	500	0	500
# Errors	0	0	0
% Time	93,52	0,00	93,52
Driver API Calls Summary | All			
# Calls	102	100	2
# Errors	0	0	0
% Time	4,27	0,02	4,25
Launches Summary | All			
# Launches	60	0	60
% Device Time	15,91	0,00	15,91
Memory Copies All			
H to D # Copies	50	0	50
H to D # Bytes	125.911.040	0	125.911.040
H to D % Time	1,4	0,0	1,4
D to H # Copies	40	0	40
D to H # Bytes	10.270	0	10.270
D to H % Time	0,0	0,0	0,0
D to D # Copies	10	0	10
D to D # Bytes	40.960	0	40.960
D to D % Time	0,0	0,0	0,0

Top Device Functions By Total Time Summary | All

	Name	Launches	Device %	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
1	safetyState	10	15.90	200,518.182	19,591.518	20,051.818	21,238.372
2	checkIfGreater	20	0.01	74.531	3.040	3.727	4.736
3	subtractVectors	20	0.00	62.527	2.848	3.126	3.488
4	addVectors	10	0.00	32.257	3.072	3.226	3.456

Figure 6: CUDA Summary for the execution of GPU implementation on D1_TC5 dataset

VI. Discussions, Conclusion and Future Work

In this work, I aimed to provide an applicable implementation of Banker's Algorithm for today's computers with large number of processes and resources. For this purpose, we utilized NVIDIA's parallel programming environment, CUDA. I tried to apply the concepts that I learned during the lectures to a problem that is not attacked by anyone using CUDA. I have used concepts such as multiple kernels, atomic operations, shared memory, and inter-thread communication. Furthermore, I made some research about how to implement the algorithm in an efficient way. This research has led me to the concept of **Dynamic Parallelism**, a topic which we did not cover during the lectures. Using the prior knowledge I gained during lectures, I was able to successfully integrate Dynamic Parallelism concept to my project. In the end, I have implemented Banker's Algorithm in CUDA, performed experiments on CUDA and CPU implementations, and obtained 20-32x performance improvements compared to CPU implementations.

Of course, there are limitations of my CUDA implementation. As discussed previously, my implementation restricts number of processes in the system to 1024. Even though this seems enough for today's systems, technology grows fast and this number may be insufficient in the near future. As a future work, this implementation can be generalized to systems with more than 1024 processes. Moreover, there are also some rooms for performance improvements. Inter-thread communication in the Safety State is the bottleneck of the algorithm apart from the memory allocations. Knowing the number of processes and resources in the system priorly, a more naïve implementation may be designed in order to minimize dependencies between different CUDA threads.

All in all, this has been very beneficial work for me. I had a chance to practice the concepts I learned during lectures. Moreover, I have learned some new concepts such as Dynamic Parallelism. In the end, I have implemented Banker's Algorithm in CUDA and achieved superior performance improvements compared to CPU implementations.

References

[1] Dijkstra, E. W. "Cooperating sequential processes," Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[2] Jones, S. "Introduction to Dynamic Parallelism", GTC2012, 2012. [Online]. Available: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0338-GTC2012-CUDA-Programming-Model.pdf

[3] Chandy, K. M.; Misra, J.; Haas, L. M. "Distributed deadlock detection", ACM Transactions on Computer Systems, 1983.

[4] Dimitoglou, G. "Deadlocks and methods for their detection, prevention and recovery in modern operating systems", ACM SIGOPS Operating Systems Review, 1998.

[5] https://github.com/topics/bankers-algorithm

[6] "Banker's Algorithm in Operating System", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/

[7] Adinets, A. "Adaptive Parallel Computation with CUDA Dynamic Parallelism", NVIDIA Developer Blog, 2014. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/

Appendix

A. Safety State for GPU Implementation

In this section we will discuss Safety Algorithm for GPU implementation in more detail. Please refer to Algorithm 5 in Section IV-C for general workflow of Safety State.

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, all threads set *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. Note that setting this *workUpdateFlag* by all threads causes bank conflicts since all threads write to this shared memory location at the same time. But this is unavoidable since threads cannot know which other threads have finished which have not. Passing this information between threads would require additional inter-thread communication and this would again slow down the overall operation.

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.

Some Notes About Dynamic Parallelism:

Dynamic Parallelism is a concept that is introduced in **CUDA 5.0 and it requires Compute Capability 3.5 or higher (sm_35).** Dynamic Parallelism basically allows launching kernels from threads running on the device. Threads can create kernels with different number of threads. Dynamic parallelism is useful for problems where there exists a nested parallelism [7]. Actually, Safety State of Banker's Algorithm also contains a nested parallelism. If we look at Algorithm 5 in Section IV-C-3, repeat loop is parallelized for all processes (CUDA threads) in the multi-process & multi-resource system. Moreover, in each parallel branch, we check if *Need(i) < Work* is satisfied or not. Since these vectors are of size *1 x numResources*, we can compare them in parallel. This step causes a nested parallelism, and this is where Dynamic Parallelism comes in handy. Thanks to Dynamic Parallelism, by launching new kernels (*checkIfGreater*) in each parallel branch, we make the aforementioned comparison in parallel on GPU.

B. Creation of Datasets

We created all datasets using the *input_generator.py* python script. For all datasets, we set maximum number of resources from any resource type to 20. There is no specific reason for this selection, and it does not affect performances of the algorithms. All datasets use *requestingProcessId = 0* for simplicity. Again, this does not affect the performance and working principles of the implementations. For each test case of each dataset, we generate text (.txt) files for each of the *Max, Allocation, Need, Available, and Request* matrices. Moreover, we generate an *info.txt* file for each test case which stores necessary information about the generated data that will be parsed by the CPU and GPU implementations while loading matrices from disk to the memory.

In order to generate datasets for the test cases where Validation step fails, we simply incremented Request vector's corresponding element by 1 in order to generate an invalid request (lines 60-67 in *input_generator.py*).

In order to generate datasets for the test cases where Safety step is successful and allocation is feasible, we simply generated *Available* matrix so that its values are between *maxResourceAmount* and *2 x maxResourceAmount* (line 55 in the script). By this way, we make sure that *Available* number of resources more than any processes needs, and system can finish without a deadlock.

In order to generate datasets for the test cases where Safety step fails, we simply took the dataset generated for the test case where Safety step is successful. Then, we changed the first element of the *processId = 1* to a large number, 1000000. This operation causes all processes other than *processId = 1* to finish. Then, in the end, *processId = 1* cannot terminate since necessary resources are not available. We say system is in an unsafe state. These test cases are pretty similar to the test cases where Safety step is successful in terms of execution time because in both datasets, we traverse all processes until deciding whether the state is safe or not. This is because we generate UNSAFE dataset from SAFE dataset by modifying only one element as described above.