KOÇ UNIVERSITY

parCorelab

Başarım 2017

MAM: A Memory Allocation Manager for GPUs

Can Aknesil, Didem Unat

Computer Science and Engineering

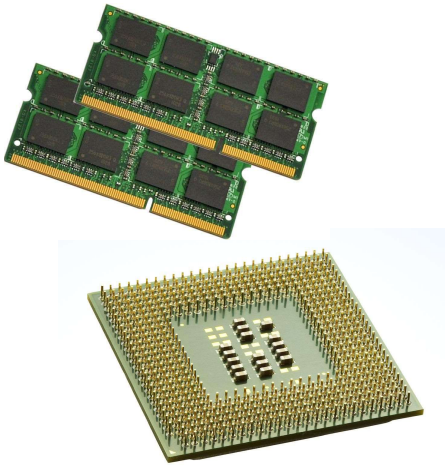
Koç University, Istanbul, Turkey

OUTLINE

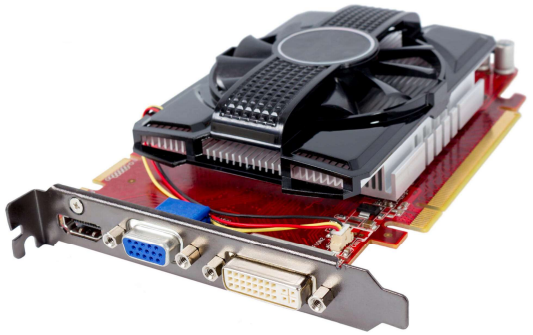
- Background
- Motivation
- What is MAM and how to use it?
- How MAM works?
- Performance results of MAM
- Extensibility of MAM

BACKGROUND: HOST VS DEVICE

- **Host:** The CPU and the host memory (the memory that CPU uses).
- **Device:** The GPU and the device memory (the memory that GPU uses).



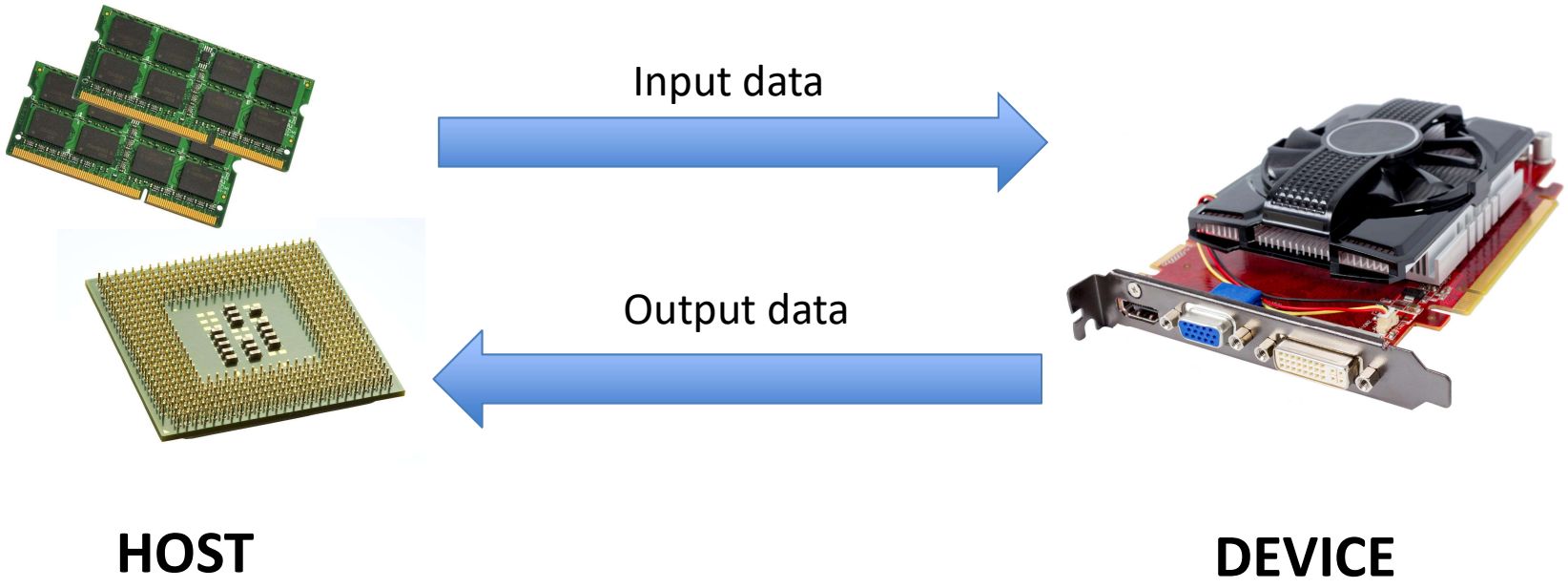
HOST



DEVICE

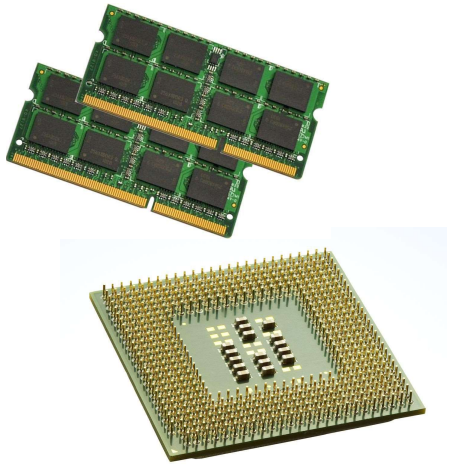
BACKGROUND: GPU USAGE

1. Input data is copied from the host memory to the device memory.
2. The GPU computes the output.
3. Output data is copied back to the host memory.

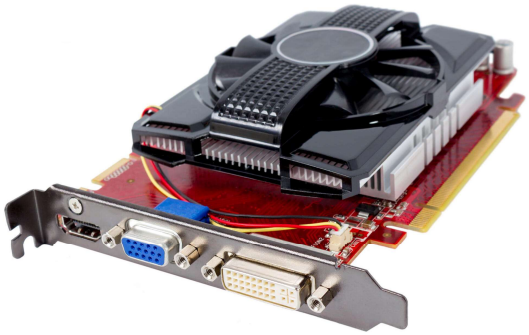


BACKGROUND: HOST & DEVICE MEMORY ALLOCATION

- Host memory allocation is performed via `malloc()` call (with C, C++).
- Device memory allocation is performed via `cudaMalloc()` call (with CUDA).



HOST

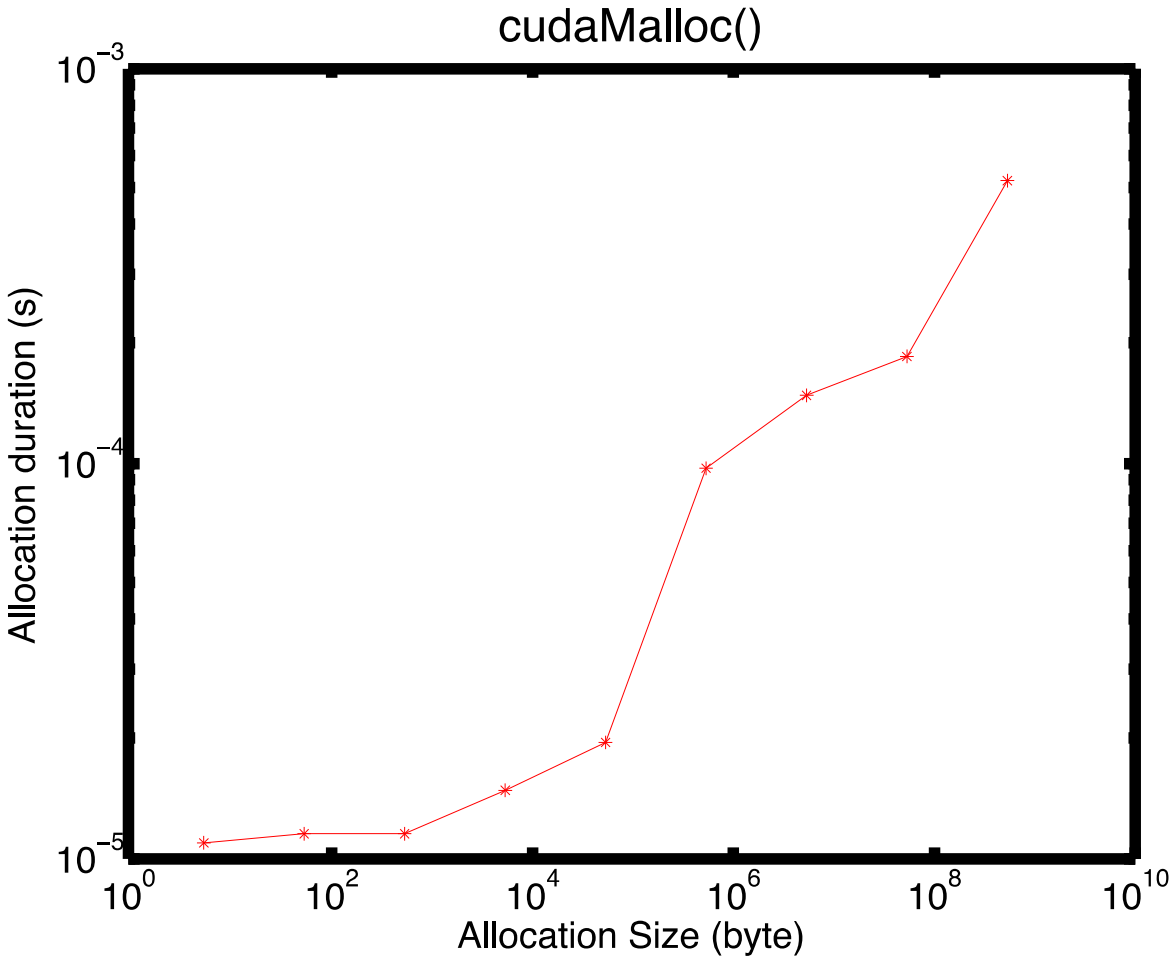


DEVICE

MOTIVATION

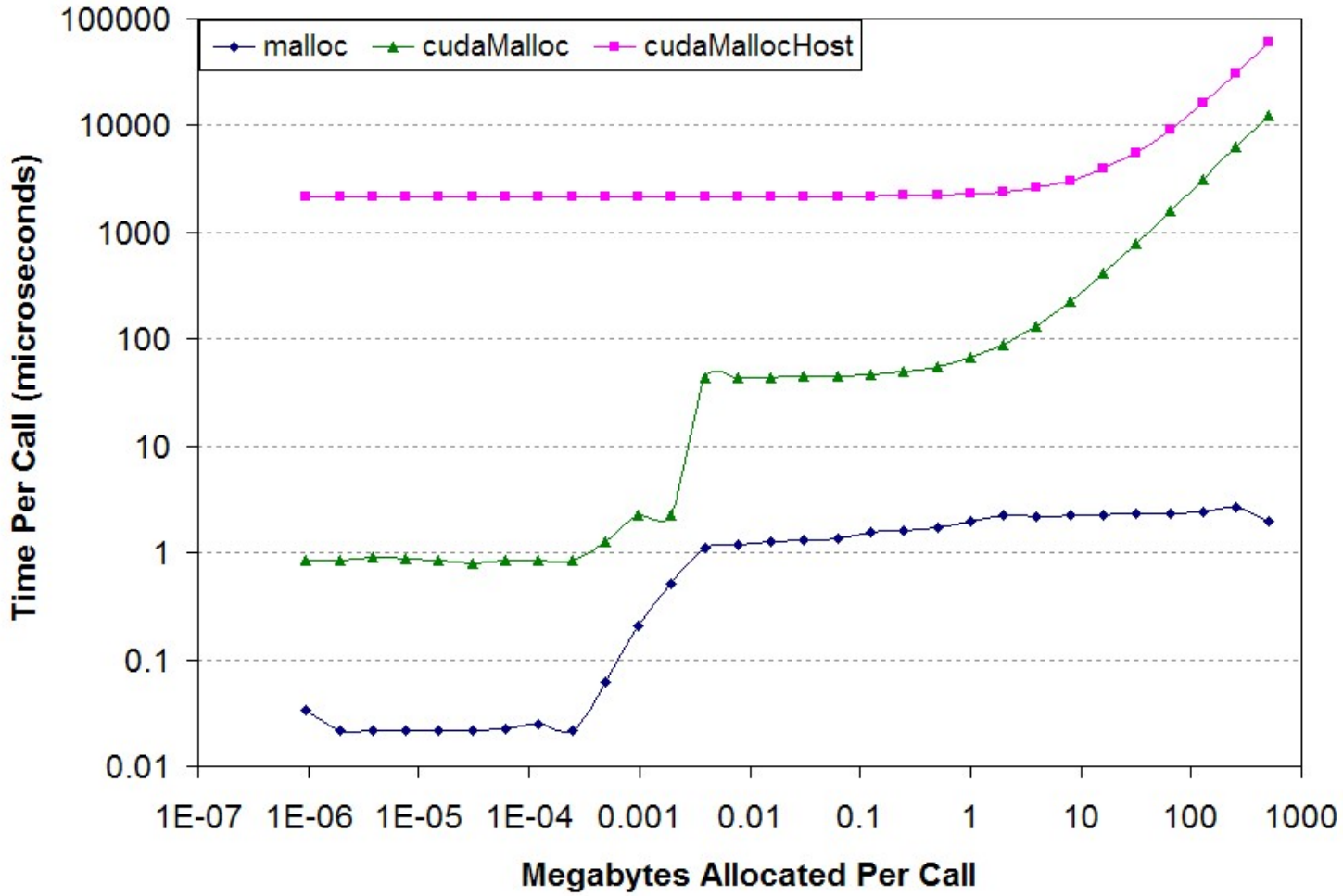
- `cudaMalloc()` is slow especially when allocation size is large.
- Applications requiring repetitive allocations may reduce the performance.
- We develop a memory management library primarily for GPUs.

MOTIVATION



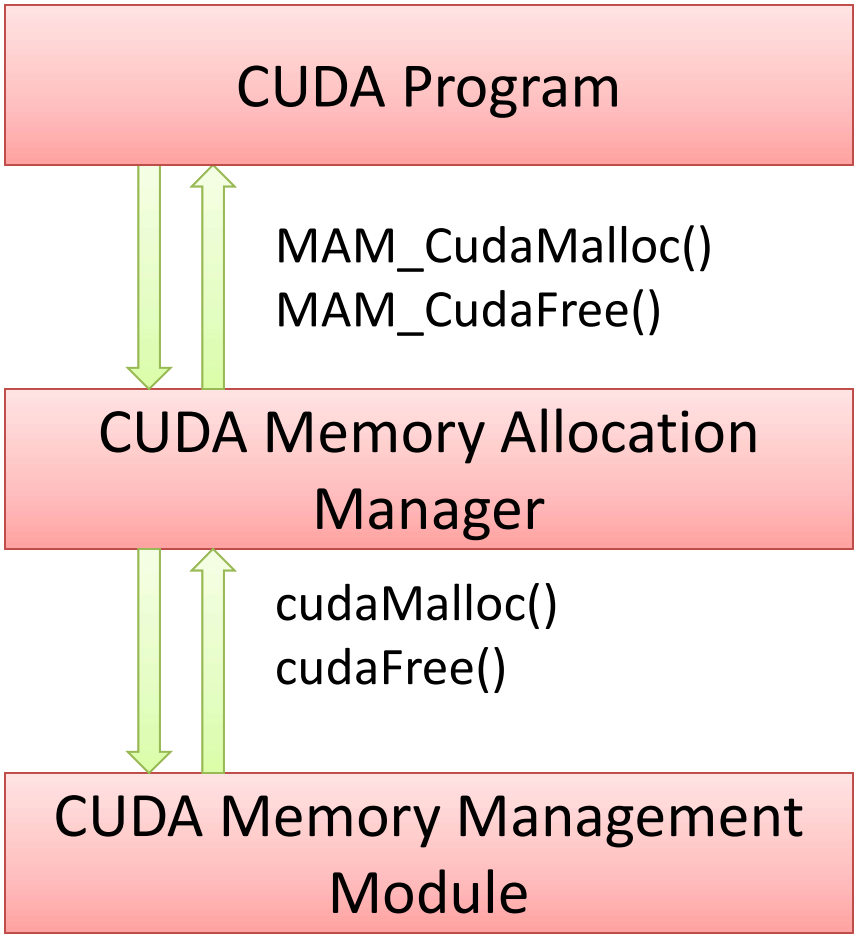
Kernel: Linux 2.6.32-431.11.2.el6.x86_64,
GPU: Tesla K20m, NVCC: 7.0, V7.0.27

MOTIVATION



From:
https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html

WHAT IS MAM?



- MAM is a C library that provides an abstraction layer between the programmer and the memory management module of CUDA environment.
- MAM removes the overhead of memory allocation.
- MAM is thread-safe.

HOW TO USE MAM (general)

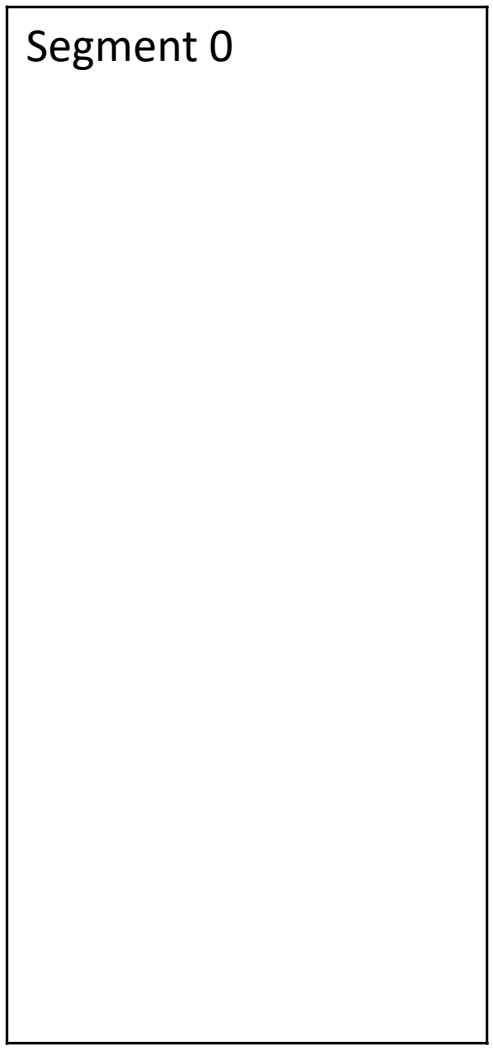
REGULAR

```
cudaMalloc(&ptr, size);
...
cudaFree(ptr);
```

WITH MAM

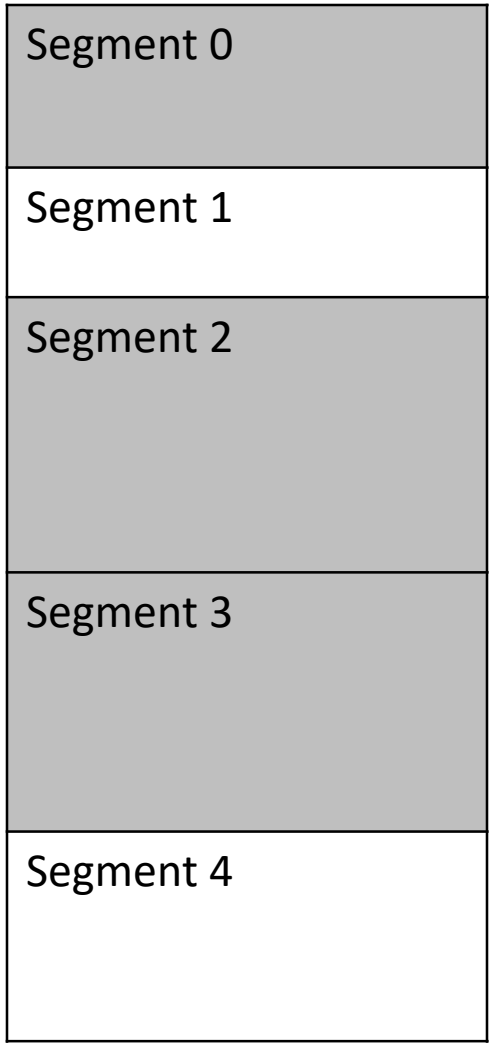
```
MAM_Create();

MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);

MAM_Destroy();
```

HOW MAM WORKS?



- When *MAM_Create()* is called, a large continuous *chunk* of memory is allocated on the device.
- The size of the chunk is expected to be greater than the total size of the memory that will be allocated by the CUDA program at a time instance.

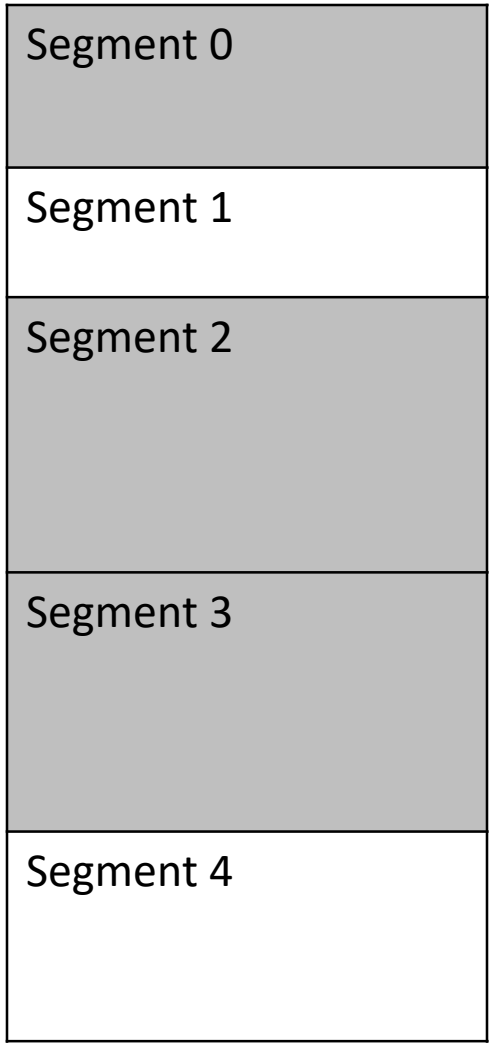
HOW MAM WORKS?



- During allocation and deallocation, MAM manipulates *segments* of the *chunk* by creating and destroying them.
- Each *segment* is nothing but a small sized structure stored in the host memory.

Blue: Segments being used
White: Empty segments

HOW MAM WORKS?



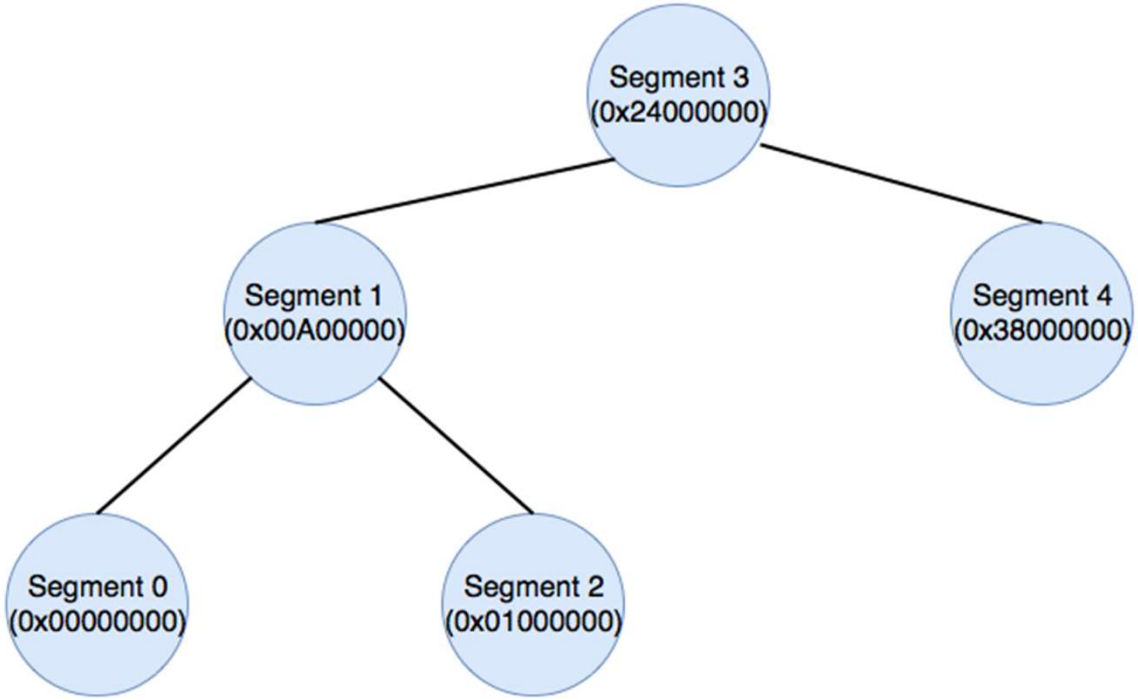

```
struct segment {
   void *basePtr;
   size_t size;
   char isEmpty;

   /* data related to data
   structures */
   ...
};
```

DATA STRUCTURES IN MAM

Segment 0 (base = 0x00000000)
Segment 1 (base = 0x00A00000)
Segment 2 (base = 0x01000000)
Segment 3 (base = 0x24000000)
Segment 4 (base = 0x38000000)

POINTER-TREE:

- Stores empty segments.
- Sorted according to their base pointer.
- Red-Black Tree is used.

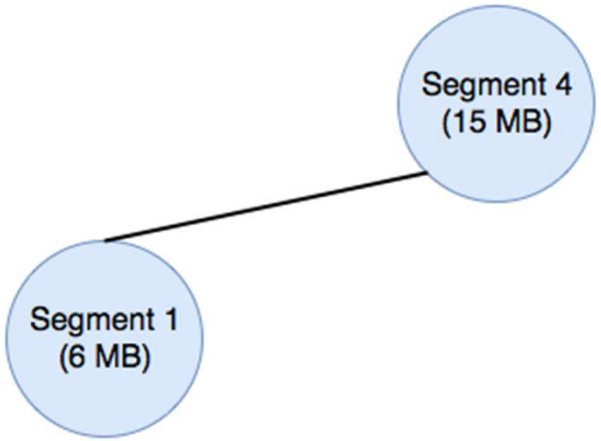


DATA STRUCTURES IN MAM

Segment 0 (size = 10 MB)
Segment 1 (size = 6 MB)
Segment 2 (size = 20 MB)
Segment 3 (size = 20 MB)
Segment 4 (size = 15 MB)

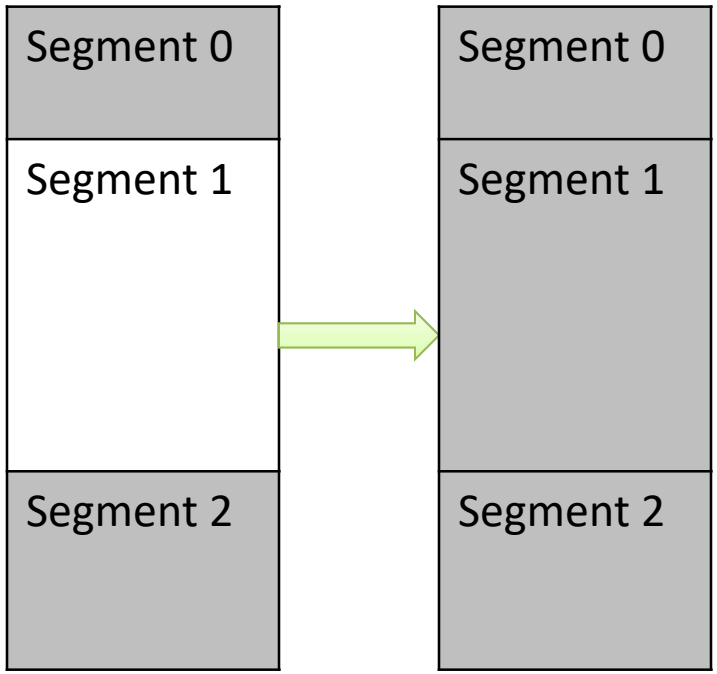
EMPTY-SEGMENT-TREE-DICTIONARY:

- Stores empty segments.
- Sorted according to their sizes
- Red-Black Tree is used.

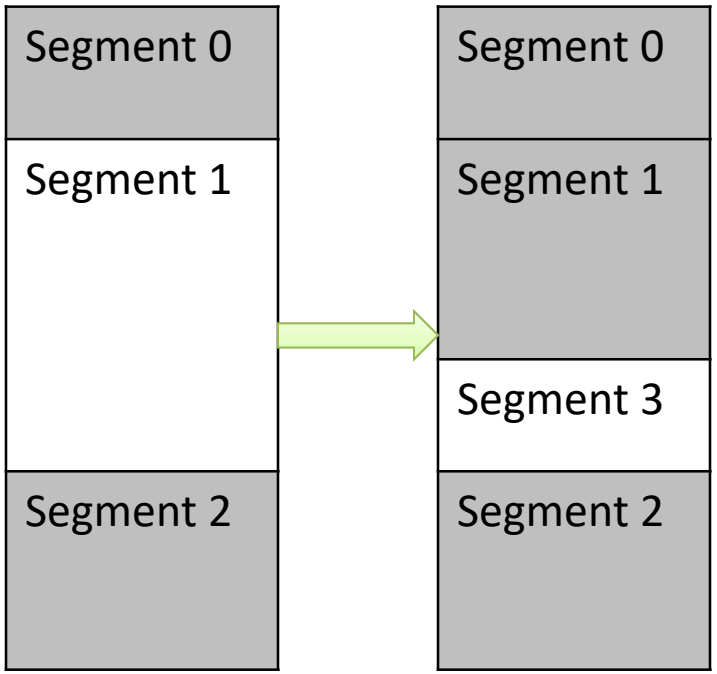


HOW MAM WORKS? (ALLOCATION)

IF DESIRED SEGMENT SIZE PERFECTLY FITS TO AN EMPTY SEGMENT



IF DESIRED SEGMENT SIZE DOESN'T PERFECTLY FIT TO ANY EMPTY SEGMENT, BEST FITTING EMPTY SEGMENT IS CHOSEN



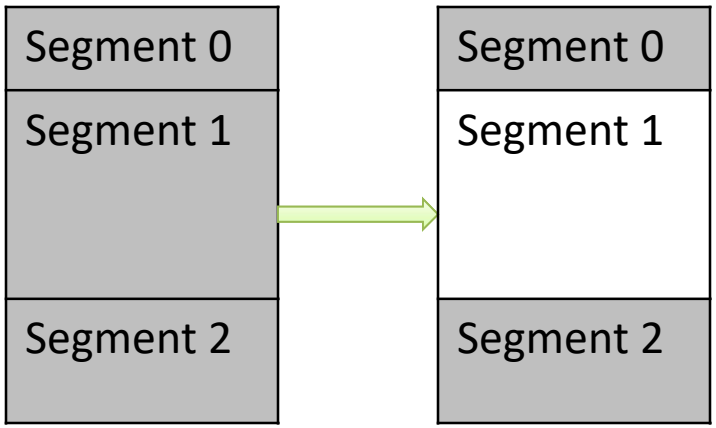
HOW MAM WORKS? (ALLOCATION)

```
Find a best-fitting empty segment from tree-dictionary;
//O(log(n))
Mark the segment as filled; //O(1)
If perfectly fits //O(1)
   Remove it from tree-dictionary; //O(log(n))
Else
   Resize it; //O(1)
   Remove it from tree-dictionary; //O(log(n))
   Create a new empty segment; //O(1)
   Insert it in pointer-tree and tree-dictionary;
   //O(log(n))
Return base pointer of filled segment; //O(1)
```

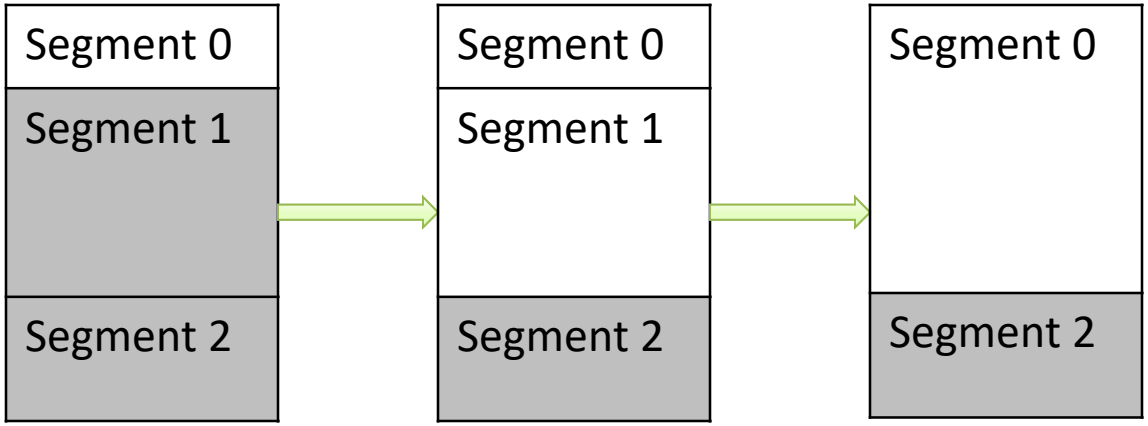
- Time complexity: O(log(n))
- Space complexity: O(n)
- n = number of segments

HOW MAM WORKS? (DEALLOCATION)

IF NEIGBOURS ARE BEING USED



IF NEIGBOURS AREN'T BEING USED



HOW MAM WORKS? (DEALLOCATION)

```
Find the segment with pointer-tree; //O(log(n))
Mart it as empty; //O(1)
Get previous and next segments; //O(log(n))
If previous segment is empty //O(1)
   Free the emptied segment; //O(1)
   Remove it from pointer-tree and tree-dictionary;
   //O(log(n))
   Resize previous segment; //O(1)
   Replace it in tree-dictionary; //O(log(n))
If next segment is empty //O(1)
   Free the next segment //O(1)
   Remove it from pointer-tree and tree-dictionary;
   //O(log(n))
   Resize previous (or emptied) segment; //O(1)
   Replace it in tree-dictionary; //O(log(n))
```

- Time complexity: O(log(n))
- Space complexity: O(n)
- n = number of segments

PERFORMANCE ANALYSIS OF MAM

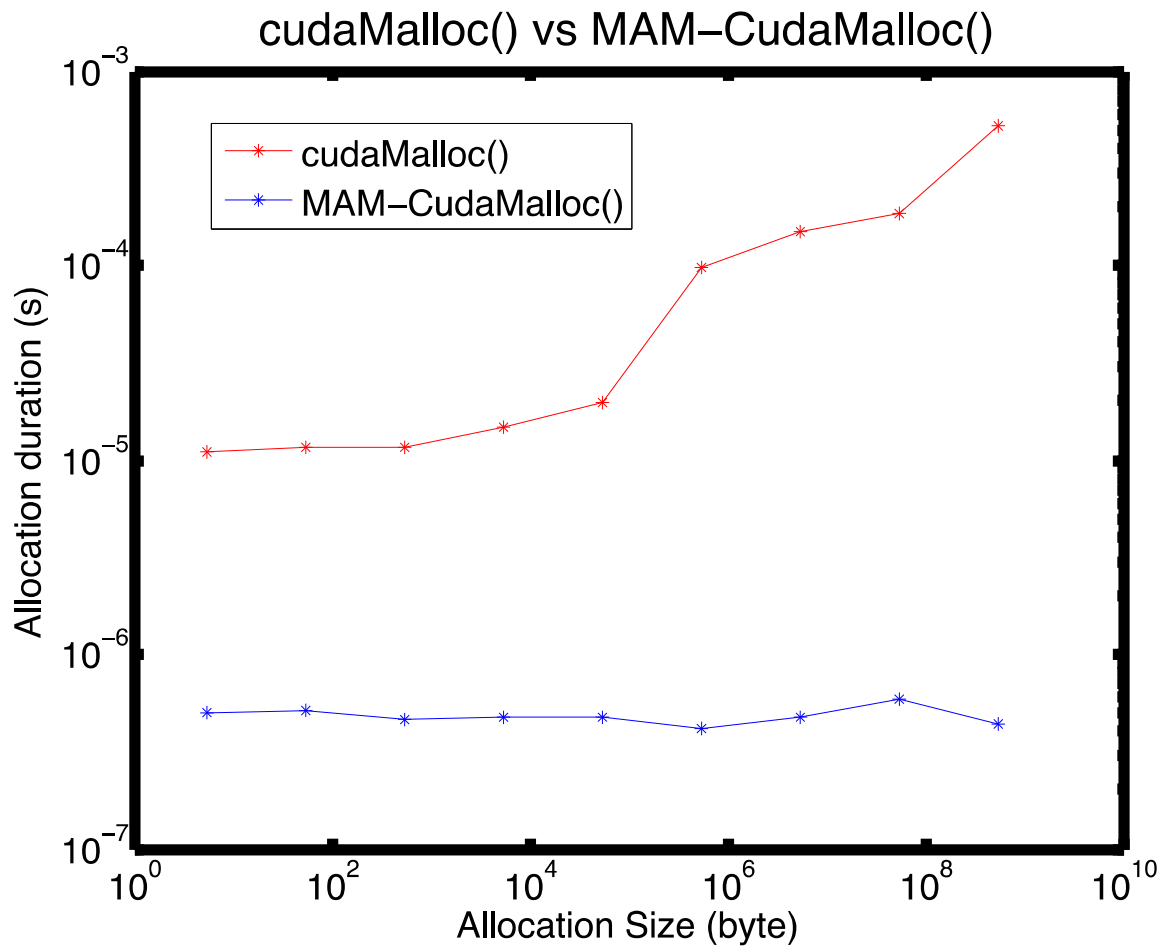
We measured the performance of MAM in two ways:

1. In terms of the allocation size:
   - Created an histogram by measuring time elapsed of several allocations of random size until the chunk is nearly full.
2. In terms of the number of previously allocated segments:
   - We measured the time elapsed during the first allocation after allocated variable number of segments.

Note: Both of these measurements exclude the creation overhead of MAM environment.

PERFORMANCE RESULTS OF MAM

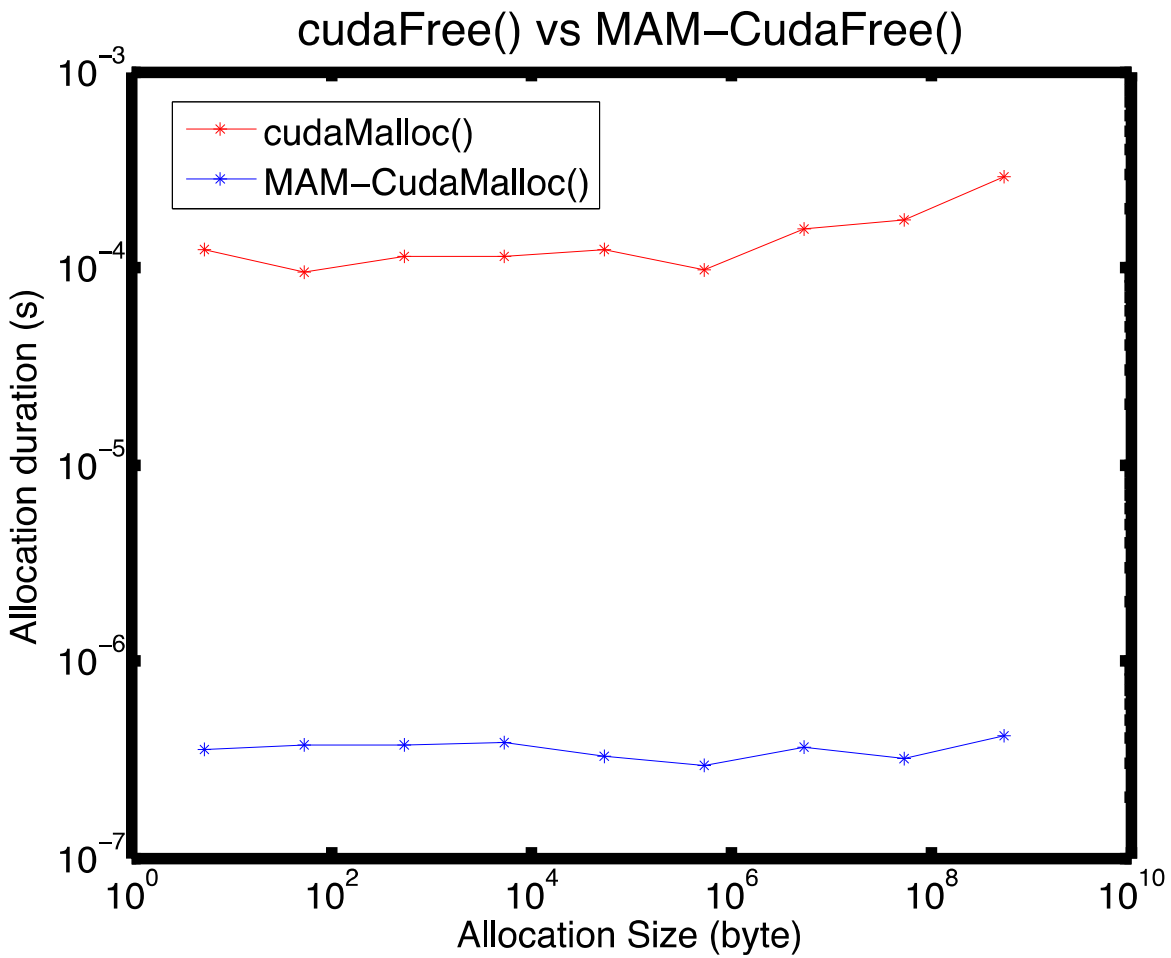
ACCORDING TO ALLOCATION SIZE



Kernel: Linux 2.6.32-431.11.2.el6.x86_64,
GPU: Tesla K20m, NVCC: 7.0, V7.0.27

PERFORMANCE RESULTS OF MAM

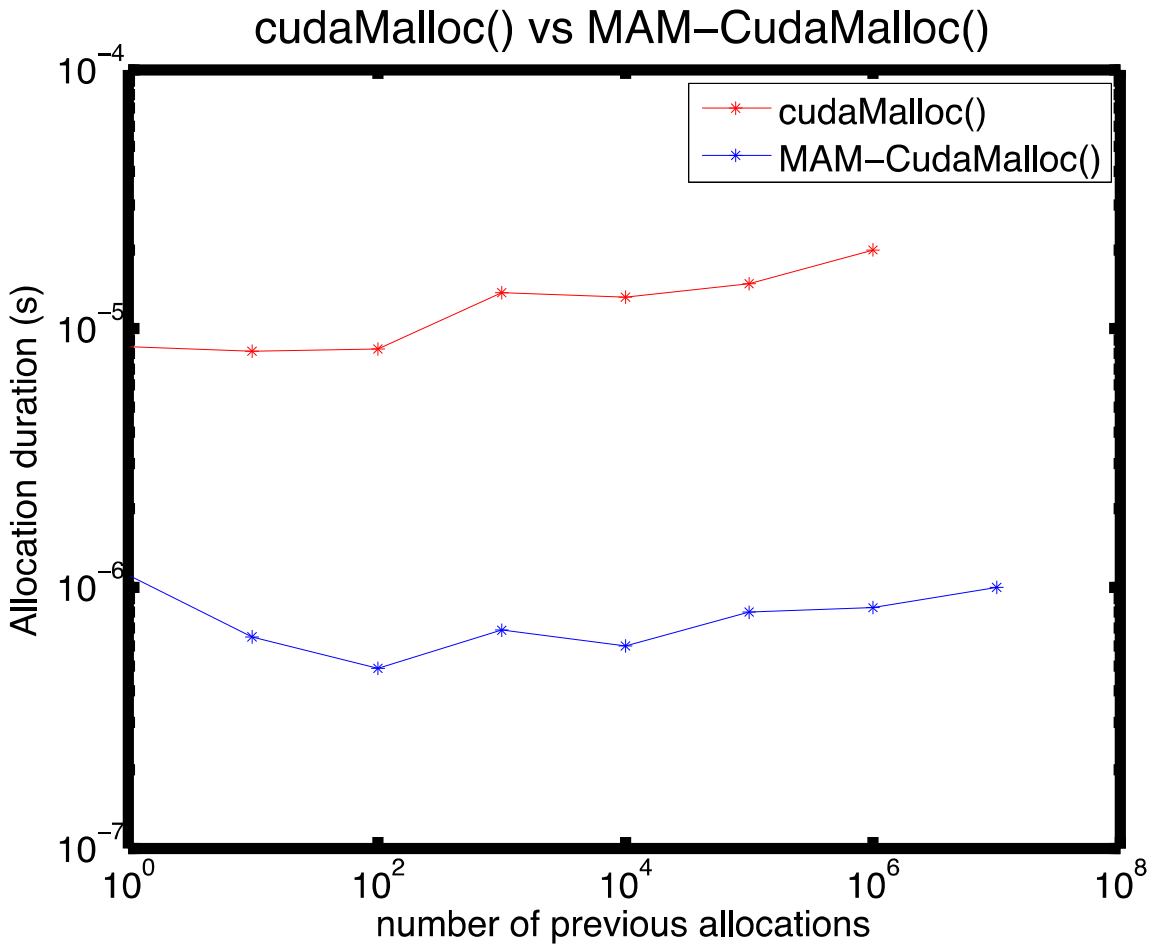
ACCORDING TO ALLOCATION SIZE



Kernel: Linux 2.6.32-431.11.2.el6.x86_64,
GPU: Tesla K20m, NVCC: 7.0, V7.0.27

PERFORMANCE RESULTS OF MAM

ACCORDING TO THE NUMBER OF PREVIOUSLY ALLOCATED SEGMENTS



Kernel: Linux 2.6.32-431.11.2.el6.x86_64,
GPU: Tesla K20m, NVCC: 7.0, V7.0.27

HOW TO USE MAM (alternatives)

SPECIFYING THE CHUNK SIZE DURING THE CREATION

```
MAM_Create(maxSize);

MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);

MAM_Destroy();
```

HOW TO USE MAM (alternatives)

WITHOUT SPECIFYING THE CHUNK SIZE DURING THE CREATION
(The largest possible chunk allocation is done automatically)

```
MAM_Create_auto();

MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);

MAM_Destroy();
```

HOW TO USE MAM (alternatives)

WITHOUT EXPLICIT CREATION

During the first call to *MAM_CudaMalloc()*, the largest possible chunk allocation is done automatically. (Lazy creation)

When all segments are deallocated, MAM destroys itself automatically.

```
MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);
```

EXTENSIBILITY OF MAM

- This presentation only covers performance comparison with CUDA memory management (`cudaMalloc()` and `cudaFree()`).
- MAM is completely applicable to any situation where a contiguous allocation occurs. For example, pined memory allocation and host memory allocation.
- MAM will work exactly the same way with any of these situations since it does not depend on allocation after it is created.

REFERENCES

[1] “CUDA Memory Management Overhead,” *CUDA Memory Management Overhead. [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_management_overhead.html. [Accessed: 14-Oct-2016].*

THANK YOU FOR LISTENING