King Saud University

Course Code:	CSC 453
Course Title:	Parallel Computing
Semester:	
Exercises Cover Sheet:	**Tutorial on Dynamic Parallelism**

Student Name:	
Student ID:	
Student Section No.	

Question 1

1. Let's consider that we would like to apply the divide and conquer parallel programming model to calculate the sum of N integers as follows: $\sum_{i=0}^{2^n-1} x_i = \sum_{i=0}^{2^{n-1}-1} x_i + \sum_{i=2^{n-1}}^{2^n-1} x_i$

   Let's consider the following kernel:

   __global__ void **sum_Kernel(**int *** data,** int *** res,** int **parentStartingIndex,**
   int **parentNbElements);**

   This kernel calculates: $res = \sum_{i=parentStartingIndex}^{parentStaringIndex+parentNbElements-1} data[i]$

   The kernel is launched by the main program as follows:

   sum_Kernel<<<1,2>>>(data, res, 0, 16);

   This will launch a grid composed of 1 block of 2 threads. Every thread will calculate the sum of ***N/2*** elements as follows:

   - Thread ***T₀*** will calculate: $\sum_{i=0}^{\frac{N}{2}-1} data[i]$
   - Thread ***T₁*** will calculate: $\sum_{i=\frac{N}{2}}^{N-1} data[i]$

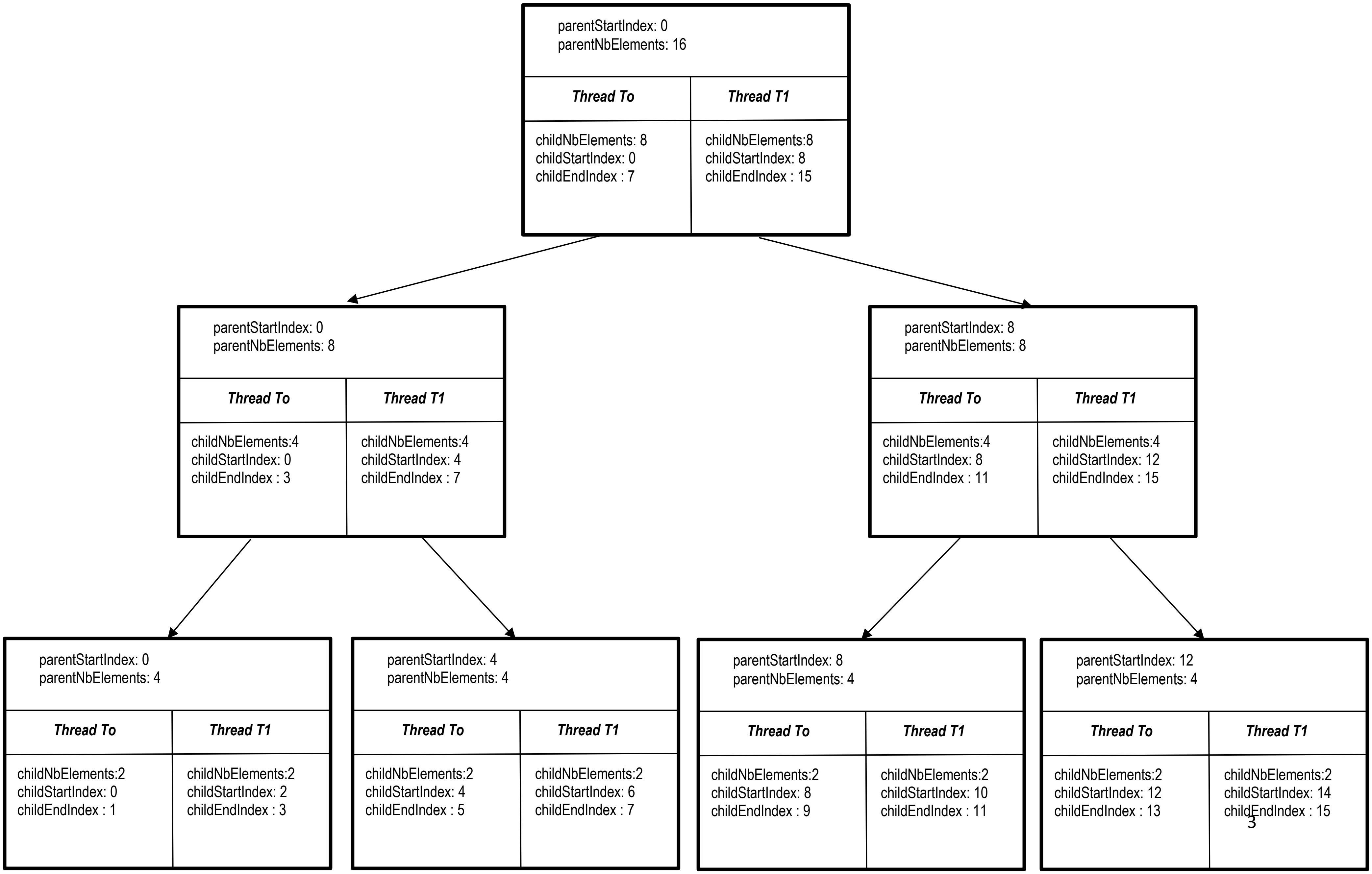
Every thread will decompose its corresponding problem (sum of ***N/2*** elements) into 2 sub-problems of size ***N/4***. This decomposition will continue recursively until we reach the base case where no more decompositions are possible. So every thread calls a grid composed of 1 block of 2 child threads. This will spread evenly the work load of the parent thread (***parentNbElements***) over its 2 child threads. The first child thread will process the first half of the parent data, while the second thread will process the second half of the parent data. Thus, the child work load (***childNbElements***) is calculated as follows: $childNbElements = parentNbElements\ /\ 2$.

Every child thread **T$_i$** will calculate the values of the following parameters:

	Thread T$_0$	**Thread T$_1$**
A$_i$	0	1
childNbElements	$parentNbElements\ /\ 2$	
childStartIndex	$parentStartIndex + A_i \times childNbElement$	
childEndIndex	$childStartIndex + childNbElement - 1$	

As such, a child thread **T$_i$** will calculate: $\sum_{i=childStartIndex}^{childEndIndex} data[i]$

a. Fill the following figure that represents the execution plan of the kernel described above to calculate the sum of 16 integers:



parentStartIndex: 0
parentNbElements: 16

Thread To	*Thread T1*
childNbElements: 8 childStartIndex: 0 childEndIndex : 7	childNbElements:8 childStartIndex: 8 childEndIndex : 15

parentStartIndex: 0
parentNbElements: 8

Thread To	*Thread T1*
childNbElements:4 childStartIndex: 0 childEndIndex : 3	childNbElements:4 childStartIndex: 4 childEndIndex : 7

parentStartIndex: 8
parentNbElements: 8

Thread To	*Thread T1*
childNbElements:4 childStartIndex: 8 childEndIndex : 11	childNbElements:4 childStartIndex: 12 childEndIndex : 15

parentStartIndex: 0
parentNbElements: 4

Thread To	*Thread T1*
childNbElements:2 childStartIndex: 0 childEndIndex : 1	childNbElements:2 childStartIndex: 2 childEndIndex : 3

parentStartIndex: 4
parentNbElements: 4

Thread To	*Thread T1*
childNbElements:2 childStartIndex: 4 childEndIndex : 5	childNbElements:2 childStartIndex: 6 childEndIndex : 7

parentStartIndex: 8
parentNbElements: 4

Thread To	*Thread T1*
childNbElements:2 childStartIndex: 8 childEndIndex : 9	childNbElements:2 childStartIndex: 10 childEndIndex : 11

parentStartIndex: 12
parentNbElements: 4

Thread To	*Thread T1*
childNbElements:2 childStartIndex: 12 childEndIndex : 13	childNbElements:2 childStartIndex: 14 childEndIndex : 15

b. Describe the base case and give its corresponding code.

c. Give the fragment of code that calculates Ai for every thread Ti.

Ai = threadIdx.x

d. Give an implementation of the kernel.

```
__global__ void sum_Kernel(        int * data, int * res, int parentStartingIndex, int parentNbElements) {
        int Ai = threadIdx.x;
        int childNbElements = parentNbElements / 2;
        int childStartIndex = parentStartingIndex + Ai * childNbElements;
        int childEndIndex = childStartIndex + childNbElements -1;
        int partialResult = 0;
        if (childNbElements == 2) { // The base case
                partialResult = data[childStartIndex] + data[childEndIndex];
        }
        else {
                sum_Kernel<<<1,2>>>(data, &partialResult, childStartIndex, childNbElements);
                cudaDeviceSynchronize();
        }
        atomicAdd(res, partialResult)
}
```