```
for i = 1 to N
    For j = 1 to M
        doSomething(i, j)
```

يتحل حل مثالي بدون داينمك

1D blocks 1D threads

```
i = bidx.x
j = tidx.x
doSomething(i, j)
```

حالة عندنا أكثر من بلوك 1D وثريد وكلها

N*M Blocks 1 thread

```
i = bidx.y
j = bidx.x
doSomthing(i, j)
```

حالة عندنا 2D بلوك وثريد واحد فقط

1 Block N*M threads

```
i = tidx.y
j = tidx.x
doSomething(i, j)
```

حالة عندنا 2D ثريدز وبلوك واحد فقط

N*M blocks N*M threads

```
i = bidx.x * bDim.x + tidx.x
j = bidx.y * bDim.y + tidx.y
doSomething(i, j)
```

حالة عندنا 2D بلوك و2D ثريدز

```
for i=0 to N
  for j=0 to X[i]
    doSomthing(i,j);
```

الحل المثالي لازم يكون بالداينمك عشان مايكون فيه آيدل ثريدز

~~i~~

Parallel Solution

```
__global__ void callKernal(...){
int i = bidx.x;
int j = tidx.x;
  if(j < X[i]){
     dosomething(i,j);
  }
}
void main(){
callKernal<<<N, max(x)>>>(X,...);
```

Bad Solition because we will have idle threads

Dynamic Solution

```
__global__ void chiledKernal(int i){
 int j = tidx.x;
dosomething(i,j);
}

__global__ void Kernal(...){
int i = tidx.x;
childKernal<<<1, X[i]>>>(i);
 cudaDeviceSyncronise();
}

void main()
{
kernal<<<1,N>>>(X..);
}
```

Perfect Solution we don't have ~~any idle threads~~

Question

Let's consider the following parallel code:

```
__global__ void Kernel_A(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_C<<< 1, 256 >>>(data);
        Kernel_D<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
__global__ void Kernel_C(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_E<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
void host_launch(int *data) {
      kernel_A<<< 1, 256 >>>(data);
      kernel_B<<< 1, 256 >>>(data);
      cudaDeviceSynchronize();
}
```

1. Give and explain the order of execution of the given parallel nested kernels.
2. Explain the role of the `__syncthreads()` statements.
3. Explain the role of the `cudaDeviceSynchronize()` statements.

1-ACEDB

__syncthreads() Allows threads of the same block to wait for each other

cudaDeviceSynchronize() the parent thread waits for its child threads to finish,
In HOST the parent waits for its last called kernel

D&C

```
__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)
}
```

Quadtree

```
__global__ void Quadtree_Kernel(int * data, int R, int C, int W, int level) {
    int X, Y;
    X = threadIdx.x / 2;
    Y = threadIdx.x % 2;

    int Ri, Ci, Wi;
    Wi = W / 2;
    Ri = R + X * Wi;
    Ci = C + Y * Wi;

    if (level == 10) {
        do_baseCase(data,Ri, Ci, Wi);
    } else {
        Quadtree_Kernel<<<1, 4>>>(data, Ri, Ci, Wi, level+1);
        cudaDeviceSynchronize();
    }

}
```

Odd Even Sort

a. How many iterations are required to sort an array of size N. ceiling(N/2)

b. How many steps are performed per iteration and describe every one of them.

b) Two steps, Odd step and Even step.
In odd step we compare the element of every odd index with the next cell, if they are out of order we swap
In even step – – – – – – – – – – – – – – – – even index –

d. Which threads will be involved in every ***step i*** in case the algorithm is performed in parallel. Don't forget to specify, for every thread, the index of the cells it will process.

In Odd step: Thread Ti | i is an odd number and will process data[i], data[i+1]
In Even step: Thread Ti | i is an even number and process data[i], data[i+1]

Bitonic sort

2. Give the number of steps that are required to sort elements of an array of size N. $\log_2(N)$

3. Give the number of stages that are required in a given step ***i***. Stages = i

4. Give the size of bitonic sequences in a given stage ***j*** of a step ***i***. 2^{i-j+1}

5. Give the condition that should satisfy a thread to participate in the processing of bitonic sequences of a stage ***j*** of a step ***i***. index $\%$ $2^{i-j+1} < 2^{i-j}$

6. Give the condition that should satisfy a thread that participates in the processing of sequences of a stage ***j*** of a step ***i*** to sort its corresponding bitonic-sequence ascendingly.