Introduction to Parallel Programming Using OpenCL and C++

Ivan Matic

1. Introduction

1.1. **GPGPU.** The abbreviation GPGPU refers to *general purpose programming on graphics processing units*. The graphic cards of modern computers can be used to do serious scientific calculations. They contain thousands of computing cores (that we will call *processing elements*), and as such they are ideal for parallel programming. The processing elements can be thought of as *small CPUs* that are numerous, but not as advanced as the central processors. High end CPUs can have speeds in range of 4GHz, but a quad-core system will have only four of these units. On the other hand, the graphic card can have 4000 processing elements each of which runs at 1GHz. In addition, not all operations are permitted and not all data structures are available to the processing elements of a typical graphic card.

1.2. **Class GraphicCard.** The interaction between GPU and the host platform is fairly complex for the beginners. This tutorial introduces the class GraphicCard written in C++ which does most of the work associated with memory management on GPU. The usage of this class removes the necessity of understanding completely the mechanisms on how *platforms*, *contexts*, *devices*, *kernels*, *workgroups*, and *memory objects* are related to each other.

Needless to say, the usage of this class is limited to building the basic codes only. It is unlikely that the class will be suitable for projects that need to harness the full power of parallel programming. The comprehensive coverage of OpenCL is provided in [1, 3].

2. CPU, GPU, RAM, and device memory

While introducing the necessary terminology we will frequently refer to the first problem we have the intention to solve:

Problem 2.1. *Read the sequence* $\overrightarrow{a} = (a[0], a[1], \ldots, a[n-1])$ *from a file on the hard disk and update it to obtain* $a[k] := 2 * a[k]$ *for each* $k \in \{0, 1, \ldots, n-1\}$.

The main idea in solving this problem is to assign each term of the sequence $\overrightarrow{a}$ to a specific processing element on the graphic card. This way one processing element will perform the operation $a[0] := 2 * a[0]$. At the same time another element will perform $a[1] := 2 * a[1]$, and so on.

The programmer may always assume that the number of processing elements is as large as necessary. When the shortage of resources occurs, a single processing element gets assigned to multiple tasks that will not be performed in parallel. The programmer does not have to be aware of that.

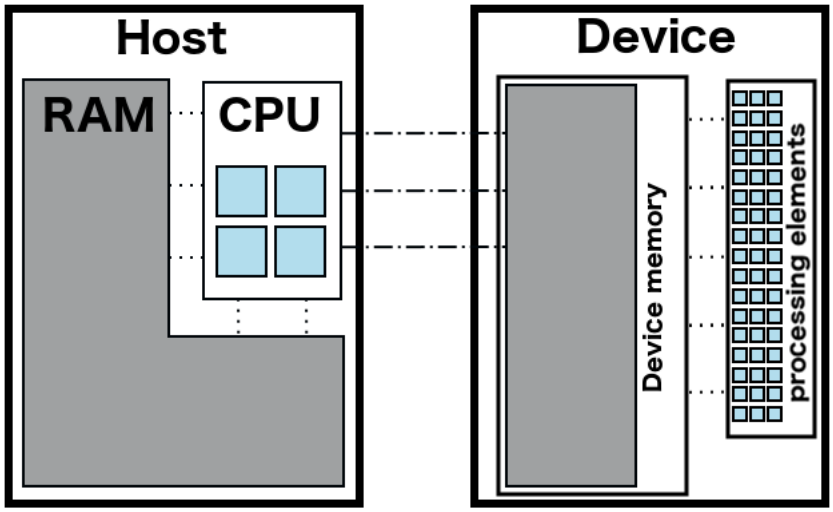

FIGURE 1. The relationship between the host and the device.

2.1. **Organization of the hardware.** For the purposes of this tutorial the computer hardware can be thought to consist of the following two units:

(1) *Host* – a traditional computing system that consists of CPU and RAM memory. Programs that are executed on CPUs will be written in C++.
(2) *Device* – a graphic card that has GPU consisting of thousands of processing elements, and its own memory that we will call *device memory*. Programs that can be run on processing elements are called *kernels* and are written in OpenCL.

Processing elements have access to the device memory only. Specifically, they cannot access the RAM memory on the host. On the other hand, the CPU can access its RAM memory, and can perform basic copying of elements from RAM to device memory and vice versa. However, one should always keep in mind that communication between CPU and device memory is not as fast as the communication between the processing elements and the device memory.

Traditional programs written in C++ organize complicated data structures within the RAM memory. In contrast, sequences are the only data structures that the class GraphicCard permits on the device memory.

2.2. **Organization of the program.** Each program that uses graphic cards has the following main components: *Kernels*, which are run on GPU processing elements and are written in OpenCL; and *Host*, which is run on CPU and is written in C++.

Solving our first problem consists of the following tasks:

(1) Reading the sequence $\overrightarrow{a}$ from a file. This task must be performed by host. The processing elements do not have the ability to access the hard disk on which the file is located.
(2) Copying the sequence $\overrightarrow{a}$ from RAM to device memory. This task has to be performed by host as well.
(3) Executing the kernel. The host will specify how many processing elements will be deployed in executing the kernel, and will synchronize the execution. The processing elements work in parallel on the same kernel code and have access to the same sequence on the device memory. The only differentiating factor between the processing elements is their ID number. Each processing element will receive a different ID and our kernel program will be designed to use this ID as an index of the sequence. This way the processing element with ID 27 will perform the task $a[27] = 2 * a[27]$, while the one with the ID 17 will perform the task $a[17] = 2 * a[17]$.
(4) Reading the result from the device memory to the host memory. This operation will be performed by the host.

3. Reading sequences from files

Our goal is to get to OpenCL programming as soon as possible and in order to do this we will use a pre-made program that can read the sequence $\overrightarrow{a}$ from the file `input00.txt` and store it in the RAM memory. The code in

`generatingSequenceFromFile.cpp`

contains two functions: `readSequenceFromFile` and `printToFile`.

Please go over the simple code from the file `example00.cpp`. It explains how the sequence is read from the input file, stored in the memory and the written to the hard-disk using `printToFile`.

The input file `input00.txt` contains the sequence that is to be read by the program. Integers (and minus signs) are treated as input, while everything else is ignored. The first number in the file is the length n of the sequence. The remaining numbers are the elements, until the number -9 is reached. This number -9 is not included in the sequence, but all the remaining terms (and there has to be a total of n terms) are generated at random.

4. Multiplying each term of sequence by 2

In this section we will develop our first program that solves Problem 2.1. We will multiply each term of the sequence $\overrightarrow{a}$ by 2.

4.1. **Designing the kernel.**

Our intention is for each processing element to work on one term of the sequence. `get_global_id(0)` is an OpenCL command that provides the processing element with the information on which ID number is

assigned to it. Once this information is obtained the task is obvious, and we may summarize this with the following code:

```
int index = get_global_id(0);
a[index]=2*a[index];
```

There is one one unpleasant surprise coming from the design of the GPU hardware. The host program (run on CPU) cannot request any number of processing elements. The architecture of GPU groups these elements and the members of each group have to be invoked together. Typically, NVIDIA hardware has groups of size 32 while AMD has groups of size 64. These particular numbers are something that is supposed to be ignored by those wishing to write elegant programs. However, it is dangerous to forget the fact that the host is almost always going to receive more processing elements than it has asked for.

In particular, the previous two lines of code may result in a disaster: If the sequence $\overrightarrow{a}$ has 57 terms and the host asks for the kernel to be run on 57 processing elements, the host will actually receive 64 of processing elements. One unfortunate processing element will receive the id 60 and consequently will try to access $a[60]$, which is not a memory that should be accessed. One way to prevent the described difficulty is to supply each processing element with the information on the length of the sequence, and the code becomes:

```
int index = get_global_id(0);
if(index<lengthOfTheSequence){
     a[index]=2*a[index];
}
```

We have now finished designing the algorithm that will be run on device. It remains to create the file `parallel01.cl` with the following code:

```
__kernel void eachTermGetsDoubled(__global int* a,
                                  __global int* n)
{
    int lengthOfTheSequence=*n;
    int index = get_global_id(0);
    if(index<lengthOfTheSequence){
        a[index]=a[index]*2;
    }
}
```

The word `__kernel` must be used before the declaration of kernel. The kernel must not return anything because there is no mechanism for CPU to accept parallel

returns of thousands of kernels. Thus, the function type must be `void`. The name we have chosen for our kernel is `eachTermGetsDoubled`.

Then we list our arguments. The arguments are data structures that are located on the device memory. They must be sequences. In C++ sequences and pointers are related, and sequence `a` is a variable of type `int*`. The length of `a` has to be supplied to the processing elements and this task is accomplished by providing an additional argument to the kernel. Since this argument must be a sequence, we are forced to have `n` of type `int*`. We will not go into details on what does `__global` mean.

4.2. **Designing the program on the host.** The host first reads the input sequence from the file `input01.txt` in the way that was described in Section 3. The sequence will be stored in `aRAM`. The host will also learn the length of the sequence and this information will be kept by the variable `nRAM`. The host will then create an object `myCard` that will handle the communication with the graphic card and the synchronizations between processing elements. This is achieved with the command

```
GraphicCard myCard("parallel01.cl");
```

Immediately after its creation the object `myCard` allows us to create the memory on the device in which the sequence `a` will be stored.

```
myCard.writeDeviceMemory("a",aRAM,nRAM);
```

The function `writeDeviceMemory` creates a sequence on the device. The first argument specifies the name by which this sequence will be referred to in future. The kernel can now use this name in its arguments to access the sequence. The second argument is the sequence in RAM whose content will be copied to the device, and the third argument specifies the length of the sequence on the device.

The device memory also needs to contain the information on how long the sequence $\overrightarrow{a}$ is, which is stored in `nRAM`. In order to move this information to a place accessible to processing elements we will use the function `writeDeviceMemory` again. The first argument will be `"n"` since our kernel expects this name to be used for the length of the sequence. The second argument must be some kind of a sequence. The problem is that `nRAM` is not a sequence. It is an integer. However, `&nRAM` is a pointer to integer, which in C++ is the same as sequence of length 1. The last, third argument to `writeDeviceMemory` is the length of the sequence `&nRAM` and this length is 1. In summary, we will be calling the function by

```
myCard.writeDeviceMemory("n",&nRAM,1);
```

The kernel can be now executed using `nRAM` processing elements. This is accomplished with the command:

```
myCard.executeKernel("eachTermGetsDoubled",nRAM);
```

The execution will modify the sequence `a`. Each of its terms will be multiplied by 2, and the result will be stored in `a` again. However, the sequence `a` is located on the device, not in the RAM memory. The retrieval is obtained using the following code

```
int* resultRAM;
resultRAM=new int[nRAM];
myCard.readDeviceMemory("a",resultRAM,nRAM);
```

After these lines of code, the sequence `resultRAM` contains the desired outcome. We can summarize our code with the following listing

```
#include <iostream>
#include <fstream>
#include "GraphicCardOpenCL.cpp"
#include "generatingSequenceFromFile.cpp"
int main(){
    int theNumberOfElementsToOutput=30;
    int *aRAM;
    int nRAM;
    readSequenceFromFile("input01.txt", &aRAM, &nRAM );

    GraphicCard myCard("parallel01.cl");
    myCard.writeDeviceMemory("a",aRAM,nRAM);
    myCard.writeDeviceMemory("n",&nRAM,1);

    myCard.executeKernel("eachTermGetsDoubled",nRAM);

    int * resultRAM;
    resultRAM=new int[nRAM];
    myCard.readDeviceMemory("a",resultRAM,nRAM);

    std::ofstream fOutput;
    fOutput.open("output01.txt");
    int mLen=nRAM;
    if(mLen>theNumberOfElementsToOutput){
       mLen=theNumberOfElementsToOutput;
    }
    for(int i=0;i<mLen;i++){
        fOutput<<resultRAM[i]<<" ";
    }
```

```
    fOutput<<std::endl;

    fOutput.close();
    delete [] aRAM; delete [] resultRAM;
    return 1;
}
```

Provided that you have already positioned yourself in the appropriate folder within the terminal, the compilation of the program is done with the command:

```
c++ -framework opencl -o myprogram01 example01.cpp
```

The file `myprogram01` contains the executable binary and on OSX system can be called by typing

```
./myprogram01
```

5. Memory managment

In this section we will consider a more advanced problem that will illustrate one famous trap that occurs in designing parallel algorithms. The difficulty is referred to as *memory race* and it appears when one processing element is attempting to write to a memory location that the other processing element is reading from. Let us consider the following problem:

Problem 5.1. *The sequence $\overrightarrow{a} = (a[0], a[1], \ldots, a[n-1])$ is stored in a file on the hard disk. The operation X is defined on sequences in such a way that it takes a sequence $\overrightarrow{s}$ of length n as an input and produces an entire sequence $\overrightarrow{t} = X\left(\overrightarrow{s}\right)$ of length n as an output. The terms of the sequence $\overrightarrow{t}$ are defined as $t[k] = s[k] + 2 * s[k+1]$ if $k < n-1$, and $t[n-1] = s[n-1]$. The operation X can be defined using precise mathematical language:*

$$X\left(\overrightarrow{s}\right)[k] = \begin{cases} s[k] + 2 * s[k+1], & \text{if } k < n-1 \\ s[k], & \text{if } k = n-1. \end{cases}$$

Another operation Y is also defined on sequences in the following way:

$$Y\left(\overrightarrow{s}\right)[k] = \begin{cases} s[k] + 3 * s[k-1], & \text{if } k > 0 \\ s[k], & \text{if } k = 0. \end{cases}$$

The sequence $w[0], w[1], \ldots, w[m-1]$ is entered through standard input.

The goal of the program is to replace the sequence $\overrightarrow{a}$ with a new sequence in each of m consecutive iterations. In the first iteration, if $w[0] = 0$ then $\overrightarrow{a}$ should be replaced by $X(\overrightarrow{a})$. However, if $w[0] = 1$, then $\overrightarrow{a}$ should be replaced by $Y(\overrightarrow{a})$. In the j-th iteration (for $j \in \{0, 1, \ldots, m-1\}$), the sequence $\overrightarrow{a}$ should be replaced

by $X(\overrightarrow{d})$ if $w[j+1]=0$, or by $Y(\overrightarrow{d})$ if $w[j+1]=1$. The output of the program should be the final state of $\overrightarrow{d}$.

To make sure that the problem is correctly understood, let us consider the sequence $\overrightarrow{d}$ of length 5 with the terms $(-2,3,0,1,1)$. Let us assume that there will be $m=3$ iterations and that $w[0]=0$, $w[1]=1$, and $w[2]=1$. Then in the first iteration, the value $w[0]=0$ is used to determine that the sequence $\overrightarrow{d}$ should be replaced by the sequence

$$\begin{aligned} X(\overrightarrow{d}) &= X(-2,3,0,1,1) \\ &= (a[0]+2*a[1], a[1]+2*a[2], a[2]+2*a[3], a[3]+2*a[4], a[4]) \\ &= (4,3,2,3,1)\,. \end{aligned}$$

In the next iteration we will use that $w[1]=1$ and replace $\overrightarrow{d}$ with

$$Y(\overrightarrow{d}) = Y(4,3,2,3,1) = (4,15,11,9,10)\,.$$

In the final iteration we will replace the new $\overrightarrow{d}$ with $Y(\overrightarrow{d})$ because $w[2]=1$. The final outcome of the described sequence of operations should be the sequence $\overrightarrow{d}$ that contains the terms $Y(\overrightarrow{d}) = Y(4,15,11,9,10) = (4,27,56,42,37)$.

5.1. **The design of the algorithm.** For solving this problem we will introduce two kernels: one for operation X and another for operation Y. Then we will run a for loop on the host system and depending on whether the corresponding $w[i]$ is 0 or 1 we will call either the kernel X or the kernel Y.

5.2. **An undesirable kernel that would result in a memory race.** We will first implement a naive approach to building the kernel for the operation X. If we try to recycle our old code and modify it slightly we would end up with something like:

```
__kernel void X(__global int* a, __global int* n)
{
    int index = get_global_id(0);
    if(index<*n){
        if(index< (*n)-1){
           a[index]=a[index]+2*a[index+1];
        }
    }
}
```

We will now consider the effects of applying the previous kernel to the sequence $\overrightarrow{d} = (-2,3,0,1,1)$. Let us analyze the behaviors of the processing elements that get assigned the IDs 1 and 2. We will label these two processing elements with $PE1$ and $PE2$.

$\overrightarrow{d}$	-2	3	0	1	1
$X(\overrightarrow{d})$		$PE1$	$PE2$		

The processing element $PE1$ is doing the operation $a[1] := a[1] + 2a[2]$, and the correct result that $PE1$ should obtain is $3 + 2 \cdot 0 = 3$. In the same kernel execution, the processing element $PE2$ is calculating $a[2] := a[2] + 2a[3]$. If by any chance the element $PE2$ finishes the calculation before the element $PE1$ has started, then the value stored in $a[2]$ will no longer be 0, but 2 instead. The evaluation that $PE1$ performs will end in tragedy. The value of $a[1]$ will be $3 + 2 \cdot 2 = 7$, instead of 3.

One may ask "Is this really a problem" – the processing elements are working simultaneously. Well, this simultaneous work is not guaranteed. They may be working simultaneously, but there are many situations in which that would not be the case. For example, if the number of processing elements that the host requests is larger than the number of available elements, than some elements will get several tasks assigned to them. The order in which they choose to execute the tasks may not be the one that the programmer had in mind.

5.3. **memory race avoidance.** We have to make sure that the processing elements are not writing to the locations from which some other processing elements could be reading. In this particular example this could be avoided by using another sequence $\overrightarrow{b}$ to which the processing elements will write while executing the kernel X. The updated kernel X becomes:

```
__kernel void X(__global int* a, __global int* b,
                __global int* n)
{
    int index = get_global_id(0);
    if(index<*n){
        if(index< (*n)-1){
           b[index]=a[index]+2*a[index+1];
        }
        else{
           b[index]=a[index];
        }
    }
}
```

In a similar way we construct the kernel Y.

```
__kernel void Y(__global int* a, __global int* b,
                __global int* n)
{
    int index = get_global_id(0);
    if(index<*n){
        if(index>0){
           b[index]=a[index]+3*a[index-1];
```

```
        }
        else{
           b[index]=a[index];
        }
    }
}
```

These modifications in the kernel will have one consequence that would require a few more lines of code. The sequence $\overrightarrow{b}$ will have the values that we intended to assign to $\overrightarrow{a}$. One remedy for this small trouble is to add one more kernel that copies the sequence $\overrightarrow{b}$ to sequence $\overrightarrow{a}$. This new kernel will be called after each execution of X or Y.

```
__kernel void copyBtoA(__global int* a,
                       __global int* b,
                       __global int* n)
{   int index = get_global_id(0);
    if(index<*n){
        a[index]=b[index];
    }
}
```

If the previous three kernels are saved in the file `parallel02.cl` we can write the following program in `example02.cpp`:

```
#include <iostream>
#include <fstream>
#include "GraphicCardOpenCL.cpp"
#include "generatingSequenceFromFile.cpp"
int main(){
    std::ofstream fOutput;
    fOutput.open("output02.txt");
    int *aR; int *wR;
    myint nRAM; myint m;

    std::cout << "How long is the array w?";
    std::cout<< std::endl;
    std::cin >> m;
    std::cout<< "Enter array of "<<m;
    std::cout<<" terms."<<std::endl;
    wR = new int[m];
    for (int j=0; j<m; j++) {
```

```
        std::cout<<"Enter the "<<j<<"th element. ";
        std::cin >> wR[j];
    }

    readSequenceFromFile("input02.txt",&aR,&nRAM );

    GraphicCard myCard("parallel02.cl");
    myCard.writeDeviceMemory("a",aR,nRAM);
    myCard.writeDeviceMemory("b",aR,nRAM);
    myCard.writeDeviceMemory("n",&nRAM,1);

    for (int k=0; k < m; k++) {
        if (wR[k] == 0) {
            myCard.executeKernel("X",nRAM);
            myCard.executeKernel("copyBtoA",nRAM);
        } else {
            myCard.executeKernel("Y",nRAM);
            myCard.executeKernel("copyBtoA",nRAM);
        }
    }

    int * resultRAM; resultRAM=new int[nRAM];
    myCard.readDeviceMemory("a",resultRAM,nRAM);

    myint mLen=nRAM;
    if(mLen>30){mLen=30;}
    for(int i=0;i<mLen;i++){
        fOutput<<resultRAM[i]<<" ";
    }
    fOutput<<std::endl;
    fOutput.close();
    delete [] aR;
    delete [] resultRAM;
    return 1;
}
```

5.4. **A more efficient avoidance of memory race.** There is a room for improvement in our previous code. The execution of the kernel `copyBtoA` could be avoided if we could find a way to reverse the roles of the sequences $\overrightarrow{a}$ and $\overrightarrow{b}$ in the kernels `X` and `Y`.

The magic can be obtained by applying the method `setKernelArguments` to the object `myCard`. In our previous programming this method was not used and we did not explicitly set the kernel arguments. Instead, the kernel arguments were set implicitly for each of our kernels. The kernel `X` was at the device memory and searched for parameters that are called `"a"`, `"b"`, and `"n"`. Since the sequences with these exact names were available in the device memory, the kernel execution was successful. Otherwise, the execution would crash.

We will now create the file `parallel03.cl` to contain the exact same kernels *X* and *Y* as the previous one `parallel02.cl` did. However, `parallel03.cl` does not need to contain the kernel `copyBtoA`.

The host program that will be written in `example03.cpp` will contain the code that explicitly sets the arguments to the kernel. To illustrate the usage of `setKernelArguments`, we will slightly change the lines of the code that write the sequences to the device memory.

```
myCard.writeDeviceMemory("aDM",aR,nRAM);
myCard.writeDeviceMemory("bDM",aR,nRAM);
myCard.writeDeviceMemory("nDM",&nRAM,1);
```

The previous three lines create three sequences in the device memory. The first sequence is called `aDM`, the second is `bDM`, and the third is an one-element sequence whose only element, named `nDM` contains the length of the previous two. The sequences `aDM` and `bDM` initially contain the copy of `aR` each.

If we now try to execute the kernels `X` and `Y`, the executions would crash. The kernels will search for the arguments `"a"` and `"b"` on the device memory but will find nothing. The kernels have to be informed that their first argument is called `"aDM"` and not `"a"`. We first create the sequence of strings `"aDM"`, `"bDM"`, and `"nDM"`. This is accomplished by

```
std::string* parameters;
parameters=new std::string[3];
parameters[0]="aDM";
parameters[1]="bDM";
parameters[2]="nDM";
```

Now before each execution of the kernel *X*, we use the command

```
myCard.setKernelArguments("X",parameters,3);
```

This command informs the kernel `X` that its three arguments are `parameters[0]`, `parameters[1]`, and `parameters[2]`. The implementation of kernel `X` is allowed to refer these arguments by labels `"a"`, `"b"`, and `"n"`.

Similarly, before each execution of `Y`, we use an analogous command in which `"X"` is replaced by `"Y"`. After each kernel execution we exchange the contents of `parameters[0]` and `parameters[1]`. This last step achieves the "flipping of arguments" effect. The listing of the file `example03.cpp` is

```
#include <iostream>
#include <fstream>
#include "GraphicCardOpenCL.cpp"
#include "generatingSequenceFromFile.cpp"
int main(){
    std::ofstream fOutput;
    fOutput.open("output03.txt");
    int *aR; int *wR;
    myint nRAM;
    myint m;
    std::cout << "How long is the array w?";
    std::cout << std::endl;
    std::cin >> m;
    std::cout<<"Enter array of "<<m;
    std::cout<<" terms"<<std::endl;
    wR = new int[m];
    for (int j=0; j<m; j++) {
        std::cout<<"Enter the "<<j<<"th element. ";
        std::cin >> wR[j];
    }
    readSequenceFromFile("input03.txt",&aR,&nRAM );
    GraphicCard myCard("parallel03.cl");
    myCard.writeDeviceMemory("aDM",aR,nRAM);
    myCard.writeDeviceMemory("bDM",aR,nRAM);
    myCard.writeDeviceMemory("nDM",&nRAM,1);

    std::string* parameters;
    parameters=new std::string[3];
    parameters[0]="aDM";
    parameters[1]="bDM";
    parameters[2]="nDM";
    std::string helpS;

    for (int k=0; k < m; k++) {
        if (wR[k] == 0) {
            myCard.setKernelArguments("X",
                                parameters,3);
```

```
            myCard.executeKernel("X",nRAM);
        }
        else {
            myCard.setKernelArguments("Y",
                                parameters,3);
            myCard.executeKernel("Y",nRAM);
        }
        helpS=parameters[0];
        parameters[0]=parameters[1];
        parameters[1]=helpS;
    }
    int * resultRAM;
    resultRAM=new int[nRAM];
    myCard.readDeviceMemory(parameters[0],
              resultRAM,nRAM);
    myint mLen=nRAM;
    if(mLen>30){mLen=30;}
    for(int i=0;i<mLen;i++){
        fOutput<<resultRAM[i]<<" ";
    }
    fOutput<<std::endl;
    fOutput.close();
    delete [] aR;
    delete [] resultRAM;
    delete [] parameters;
    return 1;
}
```

5.5. **Problems.**

Problem 5.2. *For a given sequence $\overrightarrow{a}$ of length n (that is located for example, in* `input.txt`*), and a given integer j, determine the sequence that is obtained when elements of $\overrightarrow{a}$ are shifted to the right by j locations. This way the following actions happen*

- *The element $a[0]$ is moved to the position j;*
- *The element $a[1]$ is moved to $j+1$,*
- *...,*
- *The element $a[n-j-1]$ is moved to $n-1$;*
- *The element $a[n-j]$ is moved to 0,*
- *...,*
- *The element $a[n-1]$ is moved to $j-1$.*

For example, if $\overrightarrow{a} = (1,2,3,4,5,6,7,8)$ *and* $j = 3$ *the new sequence is* (6, 7, 8, 1, 2, 3, 4, 5).

Problem 5.3. *Given a sequence* $\overrightarrow{a}$ *of length* n^2+n*, let us define the matrix A and the vector* $\overrightarrow{x}$ *in the following way:*

$$A = \begin{bmatrix} a[0] & a[1] & a[2] & \cdots & a[n-1] \\ a[n] & a[n+1] & a[n+2] & \cdots & a[2n] \\ & & & \vdots & \\ a[n^2-n] & a[n^2-n+1] & a[n^2-n+2] & \cdots & a[n^2-1] \end{bmatrix}, \quad \textit{and}$$

$$\overrightarrow{x} = \begin{bmatrix} a[n^2] \\ a[n^2+1] \\ a[n^2+2] \\ \vdots \\ a[n^2+n-1] \end{bmatrix}.$$

Calculate the product $A\overrightarrow{x}$ *of the matrix A and the vector* $\overrightarrow{x}$*. The product of the matrix and the vector is defined in the following way:*

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ & & \vdots & \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} a_{1i}p_i \\ \sum_{i=1}^{k} a_{2i}p_i \\ \vdots \\ \sum_{i=1}^{k} a_{2k}p_k \end{bmatrix}.$$

Problem 5.4. *Write a program that reads the sequence* $\overrightarrow{a}$ *of length* n^2 *from the hard disk, creates the matrix A in the same way as in Problem 5.3, and calculates* A^2*.*

Problem 5.5. *Write a program that reads the sequence* $\overrightarrow{a}$ *of length* n^2 *from the hard disk and calculates* A^k*, where A is the matrix defined as in the previous problem.*

6. Parallel inspection of a sequence

Let us consider the following problem.

Problem 6.1. *Assume that the sequence* $\overrightarrow{a}$ *for length n is located in the device memory. Determine whether each element of the sequence is divisible by 5. If all numbers are divisible by* 5*, the program should return* 1*. If at least one of the numbers is not divisible by* 5*, the program should return* 0*.*

The described problem has an input that is a sequence. However, the output is a binary number. The most obvious approach to building a parallel algorithm is to assign a processing element to each term of the sequence. We may assume that the device memory contains the sequence `a`, one-element sequence `n` that represents the length of `a`, and an one-element sequence `result`. Each processing element

checks whether the corresponding term is divisible by 5, and writes 1 or 0 in the variable `result`. Since all processing elements are writing in the same memory location, the results are not predictable. This is a classical example of the memory race. One of the outcomes could be that the hardware turns the requests from parallel into serial.

Therefore we have to make a trade-off. Instead of writing in only one memory location, after the inspection, each processing element should write the result 0 or 1 in a sequence $\overrightarrow{b}$ on the device memory. Then we have to make another kernel that calculates the product of elements of the sequence $\overrightarrow{b}$. This calculation can be performed in parallel.

Problem 6.2. *There are n boxes placed around the circle. They are labeled as* 0, 1, ..., $n-1$. *The sequence* $\overrightarrow{r}$ *contains the number of cats in each of the boxes. The number* $r[j]$ *is the population of the box* j. *Every day, the following happens in each of the boxes. For each* j, *one cat from the box* j *may jump to the box immediately to the right. The box immediately to the right is labeled by* $j+1$ *if* $j < n-1$, *or by* 0 *in the case that* $j = n-1$. *The jump of one cat happens, if the night before there are at least three more cats in box* j *than in the box* $j+1$. *More formally, if* $r[j] \geq r[j+1]+3$, *then one cat jumps from* j *to* $j+1$. *Our convention is that* $(n-1)+1$ *is assumed to be* 0.

Write a program that determines the number of days it will take for migrations to stop.

7. Parallel random number generator

We will explain here a matrix method for improving the traditional random number generators. The methods can be generalized to multi-dimensional matrices.

The algorithm takes as an input an order of $O(N)$ random numbers created by a traditional sequential random number generator and produces the output consisting of $O(N^2)$ random numbers. The generalizations to higher dimensions result in higher orders of magnitudes.

In each iteration, a standard random number generator is used to produce:

- Two sequences $a[0]$, $a[1]$, ..., $a[N-1]$ and $b[0]$, $b[1]$, ..., $b[N-1]$ of integers from the interval $[0, 2^r - 1]$.
- Two sequence $\alpha[0]$, $\alpha[1]$, ..., $\alpha[N-1]$ and $\beta[0]$, $\beta[1]$, ..., $\beta[N-1]$ of permutations of the set $\{0, 1, 2, \ldots, r-1\}$.

The input includes $2N + 2rN$ numbers generated using a traditional method like Mersenne twister [2]. The output will be N^2 numbers $M_{i,j}$ defined as

$$M_{i,j} = \alpha_i\left((a|_p b)[j]\right) + \beta_j\left((b|_p a)[i]\right),$$

where $x|_p y$ is the number obtained by taking the first p binary digits of x and last $r-p$ binary digits of y.

This random number generator can be called by the method `runRN(N,r)` applied to a graphic card object. The resulting sequence will appear on device memory under the name `"rNumGCCL"`. The subsequent runs of the algorithm can be faster than the first one as the permutations need not to be generated every time. Only the sequences $\overrightarrow{a}$ and $\overrightarrow{b}$ will be created for each application of the algorithm.

Here is a sample code that will run the random number generating algorithm.

```
GraphicCard gCard("parallel03.cl",forceCPU);
int N=1000;
int r=15;
gCard.runRN(N,r);
int *result;
int resultSize=N*N;
result =new int[resultSize];
gCard.readDeviceMemory("rNumGCCL",result,resultSize);
```

REFERENCES

[1] Apple. OpenCL programming guide for Mac. *Apple* 2013.
[2] M. Matsumoto. www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.
[3] A. Munshi, B. Gaster, T.G. Mattson, J. Fung, D. Ginsburg. OpenCL programming guide. *Addison-Wesley* 2012.