# Advanced programming paradigms

## PECL 1

Gabriele Colapinto

# Implementation of the program

## Summary

# Overview

The program starts with the declaration of global variables and enumerations to make the software more readable and maintainable. Then there are the kernels which implement the functions of the program using the video card.

The kernels are the following:

| | |
|---|---|
| matrix_initializer | Kernel to set the matrices in their initial status. |
| initCurand | Kernel to initialize the random number generator of the video card. |
| shieldGenerator | Kernel to generate the shield in the fourth row of the matrix. |
| addAliens | Kernel to randomly generate a row of aliens in the last row of the matrix. |
| manualPlayerMovement | Kernel which makes the ship of the player move in row 0 in manual mode. |
| automaticPlayerMovement | Kernel which makes the ship of the player move in row 0 in automatic mode. |
| advance | Kernel to make the aliens advance in the matrix. |
| collateralEffectsActivator | Kernel to activate the collateral effects of the matrix |
| cruiserCollateralEffect | Kernel to implement the collateral effect of the cruiser. |
| destroyerCollateralEffect | Kernel to implement the collateral effect of the destroyer. |
| reconversion | Kernel to reconvert the aliens in the matrix |

In the host there are two functions: main and print_matrix. The main function is the driver code of the program, and the other function is a function to print the matrix we need for convenience.

Now let us expose the functioning of the program starting from the kernels.

## matrix_initializer

This kernel sets the matrices used by the program to their initial status. The program uses two matrices: the stage matrix and the support matrix.
The stage matrix is a matrix of characters which contains the stage of the game and the support matrix is a matrix of integers which is used by the advance kernel and the reconversion kernel.

In the initial status of the stage matrix all cells are set to empty except for the row 0 in which it puts the player in its initial position.

In the initial status of the support matrix all the cells are set to 0.

## initCurand

This kernel initializes the random number generator of the video card using the time in which this kernel is called as a seed for the random function.

This kernel is crucial for the functioning of the program because it allows to implement the probabilities starting from the generation of a random number using a uniform distribution.

The probabilities in the program are implemented by generating a random number and classifying it. For instance, the shield has a generation probability of 15%. To implement this mechanism, we generate a random number from 1 to 100 and check if it is lesser or equal to 15.

Likewise, to generate the aliens we generate a random number from 1 to 100 and check in which range it falls. The range to generate the aliens are defined in an enumeration at the beginning of the program.

## shieldGenerator

This kernel generates the shield in the fourth row of the matrix. To do everything in parallel we need to consider what are the possible conditions that could generate an invalid shield i.e. a shield made of four consecutive blocks.

We can have 4 consecutive shield blocks if:

A) The cell has 3 blocks at its right and they are occupied by a shield

B) The cell has 1 block at its left, 2 at its right and they are occupied by a shield

C) The cell has 2 blocks at its left, 1 at its right and they are occupied by a shield

D) The cell has 3 blocks at its left and they are occupied by a shield

Each thread computes these conditions and try to generate a shield with a probability of 15% if none of them are verified.

## addAliens

This function generates a row of aliens in the last row of the matrix.

To generate the aliens, we make the thread responsible of each cell generate its own alien by generating a random number and classifying it according to the following table:

| Number range | Ship type |
|---|---|
| 1 - 40 | Alien |
| 41 - 65 | Cloud |
| 66 - 80 | Cephalopod |
| 81 – 85 | Destroyer |
| 86 – 98 | Cruiser |
| 99 – 100 | Commander |

The generation of the aliens always involves the last row entirely, there cannot be empty cells.

## manualPlayerMovement

This kernel allows the player to move in the row 0 of the matrix in manual mode.

Considering that the movement depends on the input of the user we need to first check if the movement is possible. The movement is possible if the destination of the player is inside the matrix.

The movements which are not allowed are:

A) player_position == 0 && direction == LEFT

B) player_position == numColumnas - 1 && direction == RIGHT

If the movement is possible, we need to store the fact that the player has moved and check if the player hits an alien while moving. If so, we need to set to true the variable which detects the movement damage.

We need to store this information because if the player does not move the screen does not get refreshed and the movement damage is one of the possible causes of damage to the player.

## automaticPlayerMovement

This kernel allows the player to move in the row 0 of the matrix in automatic mode.

In this case the movement does not depend on the input of the user and we can implement a system that makes the ship move every time. The kernel checks if the player is in one of the corners of the matrix and, if so, makes the ship moves towards the center of the matrix or else it randomly generates a direction.

After generating the direction, this kernel moves the ship and detects the movement damage.

## advance

This kernel makes the ships advance in the matrix and implements other functions of the program in some specific rows. Furthermore, it stores the positions of possible collateral effects in the support matrix.

The advance kernel uses the support matrix to store the positions of possible collateral effects which can have the following values:

1 = Destroyer

2 = Cruiser

### Row 0:

In the first row of the matrix the aliens hit the Earth. We increase the score of the turn, increase the lives given to the player in case an alien is a commander, and we store the positions of possible collateral effects in the support matrix.

Then we check if the cell above the player contains an alien and, in such case, we store the fact that the player has received advance damage. If a ship collides with the player that ship disappears without triggering collateral effects.

After making all these computations, we can delete the aliens in this row.

### Row below the shield:

We need to check if the cell in the shield row is not a shield and, if so, we can move it down.

### Shield row:

In this case we need to check if the considered cell contains a shield or not. If it does not contain a shield, we can make the aliens advance but if it does, we need to distinguish the cases in which the ship can destroy the shield or not and consider if it can activate a collateral effect or not.

In particular:

- The cruiser has a collateral effect and it can destroy the shield.
- The destroyer cannot destroy the shield but it has a collateral effect.
- The commander can destroy the shield but it does not have a collateral effect.

Anyways, when an alien ship collides with the wall the ship gets deleted.

## collateralEffectsActivator

This kernel scans the support matrix to look for cells containing the center of possible collateral effects. Considering that collateral effects can be activated either in the row 0 or in the shield row, this kernel analyzes only these two rows.

Once a collateral effect is detected, it is activated and then its position in the support matrix is put back to 0.

To activate the collateral effects, this kernel launches other kernels: one for the cruiser and one for the destroyer. To be able to do so we need to change the settings of the project, make the code relocatable and add the library “cudadevrt.lib” to the additional dependencies.

The kernel that activates the collateral effect of the cruiser requires the direction of the explosion as an input parameter. To generate the direction of the explosion we first generate a random number between 1 and 100 then we calculate its carry by 2. By convention, we decide to use 0 to destroy the row and 1 to destroy the column.

## cruiserCollateralEffect

The collateral effect of the cruiser either sets the row or the column to empty and if the player is inside the effect, it receives collateral damage.

## destroyerCollateralEffect

This kernel applies the explosion to the stage matrix like if it was a mask. If a cell which does not contain the player is inside the explosion it is set to empty. If the player is involved in the explosion, it receives collateral damage.

## reconversion

The reconversion of the aliens is made in two steps:

1. Scanning the stage matrix to search for aliens eligible for reconversion.
2. Applying the actual reconversion of the aliens.

There are 3 types of reconversions:

1. From alien to cloud.
2. From cloud to cephalopod.
3. The commander generates clouds in its surroundings.

We first analyze the stage matrix to check if the conversions are possible and if they are we change the support matrix.

The cells of the support matrix are filled as follows:

- We put 1 if we want to convert that cell into a cloud.
- We put 2 if we want to convert that cell into a cephalopod.
- We put 3 if we want to fill that cell with a cloud.

Considering that the scanning is done in parallel, we need to make sure that all the threads have finished analyzing the matrix before proceeding with the actual reconversion. So, after scanning the matrix we need to wait for all the threads to finish and then we can proceed.

While reconverting the stage matrix we also need to change to 0 the values of the eligibility matrix.

## main

The main function contains the driver code of the program.

We first ask the user to insert the dimensions of the matrix and the game mode, then we perform all the operations which are necessary for the functioning of the game.

After these operations we can start using the kernels. We first initialize the matrix, add the shield and the first row of aliens, then we start the iterations. During the iterations we implement the sequence of actions written in the assignment and we account for the score, the lives and the damage.

The specification does not specify how the player receives damage so we could implement the damage system as we preferred. The damage system of the game is made in such a way that the player can only receive one damage per iteration. This is to be fair because there could be many simultaneous situations that make the player receive damage. For instance, the player moves onto an alien ship, then another alien ship advances against them and in a neighboring cell there is a destroyer which explodes and hits the player. In this case the player would lose three lives in only one turn, and it is unfair.

The damage system is implemented as follows:

```
if (h_movement_damage || h_advance_damage || h_destroyer_collateral_damage
||h_cruiser_collateral_damage) {
        lives--;
}
```

Between two consecutive iterations we use the Windows command “cls” to clear the screen so that the user does not see all the matrices of the current game one after the other.

At the end of the main function, we free all the allocated memory both in the host and in the device.

# Basic implementation

In the basic implementation of the project all the kernels are made to be used in one block. Of course, by doing so we need to consider that the number of threads we can use in the kernels is constrained by the number of threads a block of the video card can handle. For this reason, the number of rows is limited to 50 and the number of columns to 20.

The kernels that affect a single row of the matrix are:

- shieldGenerator
- addAliens
- manualPlayerMovement
- automaticPlayerMovement

These kernels are launched using a number of threads equal to the number of columns of the matrix and each thread handles a cell.

The kernels that affect the whole matrix are:

- matrixInitializer
- advance

These two kernels are launched using a number of threads equal to the number o columns of the matrix and go through the matrix line by line in a for loop.

The reason why we use a for loop is not only the simplicity of the implementation but also the fact that making the aliens advance using only one matrix requires going through the matrix starting from the bottom row and going upwards. If we do not do it this way, we lose information because the row on top starts overwriting the rows beneath it.

The special kernels are:

- initCurand
- reconversion
- collateralEffectsActivator
- cruiserCollateralEffect
- destroyerCollateralEffect

These kernels are special because they have something different from the others.

The kernel “initCurand” needs to be launched using a bidimensional block of threads. The dimensions of the block are (numFilas, numColumnas), this way there is a random number generator for each cell of the matrix.

The kernel “reconversion” needs to be used using the same threads of “initCurand” because the commanders could be anywhere in the matrix, and we need to make sure that they can activate their special effect regardless of their position.

The kernel “collateralEffectsActivator” is launched using a number of threads equal to the number of columns in the matrix. It is special because the collateral effects can only occur in the bottom row and the shield row of the matrix. For this reason, this kernel checks only these two rows instead of the whole matrix. When this kernel detects a collateral effect, it launches the kernel related to that effect.

The collateral effect of the cruiser is applied to the matrix by the kernel “cruiserCollateralEffect”. This kernel is launched using a number of threads equal to the biggest dimension of the matrix. The assignment says

that, in general, the height of the matrix is bigger than its width but the user could insert a number of columns greater than the number of row so we need to make sure that this kernel can affect the whole desired dimension.

The collateral effect of the destroyer is applied to the matrix by the kernel “destroyerCollateralEffect”. This kernel is launched only with the necessary number of threads which are organized in a bidimensional block of dimensions (explosionSide, explosionSide).

# Intermediate implementation

In this version of the project the kernels which affect only one row are not changed and the kernels which affect more than one row do it in parallel. We achieve this parallelism referencing the rows of the matrix using the blocks and referencing the cells using the threads.

This parallelism implies that we do not need to use the for loops to iterate over the rows of the matrix and we can do everything in parallel.

The kernels in which we just removed the for loops without significantly change the code are the following:

- matrixInitializer
- collateralEffectsActivator
- reconversion

The only kernel which was significantly changed to be more efficient is “advance”.

## advance

This kernel too does not use a for loop anymore to access all the cells of the matrix, but the significant change is the fact that it uses a result matrix. The result matrix is a matrix of characters of the same size of the stage matrix which is filled in the first part of the kernel with the result of the advancement. After this matrix is filled, we wait for all the threads to finish the operation and we copy its content into the stage matrix.

The result matrix was introduced to solve a problem related to the parallelism of the kernel. The advancement of the ships must be done necessarily one row at a time. If we use only one matrix and do not know which cell is processed at a given time, we might lose data because a row overwrites another one below it. But if we refer mainly to the result matrix and we use the kernel to calculate the content of this matrix we do not lose data anymore.

# Advanced implementation

In this version of the program kernels use the shared memory.

The kernels which affect a single row or column use a temporary array of characters in the shared memory, perform the computation and write the result in the global memory. Considering that the size of a row or a column is not given at compile time we need to allocate the shared memory used by the kernels dynamically. We can do it by providing the size of the array to the kernel as a third parameter, after the sizes of the grid and the sizes of the block.
For example:

```
const size_t ROW_SIZE = sizeof(char) * numColumnas;

shieldGenerator <<<1, numColumnas, ROW_SIZE >>>(...);
```

Using this technique requires the programmer to tell the graphic card that the size of the array in the shared memory needs to be retrieved from the parameters used to launch the kernel. To do so we use the instruction “extern”:

```
// Temporary row of characters in the shared memory

extern __shared__ char tempRow[];
```

The kernels which affect multiple rows, instead, use tiles containing each one a piece of the stage matrix.

The kernel that initializes the matrix does not have the problem of having to access elements which are not inside the tile, but other kernels do. In this case we perform a boundary check to assess if we can use an element inside the tile or need to access the global memory.

## advance

The advance algorithm does not use the result matrix in the global memory anymore. Instead, it uses result tiles and stage tiles. Before we make the assignments of the values, we perform a boundary check on the rows. If they are within the tile, we can access the shared memory but, if they are outside of it, we need to retrieve the value from the global memory. After we compute the result tiles, we wait for all the threads to finish the operations and we put the results in the stage matrix.

## reconversion

In this kernel we use tiles in the shared memory only to scan the stage matrix and compute the support matrix. Unfortunately, we cannot use a tiled algorithm to form the result matrix because threads need to access the cells neighboring the one under test and this may cause a thread to try to access another tile which is in the shared memory of another block, and this is impossible.

So, in conclusion, we use a tiled algorithm to compute the stage matrix, we wait for all the threads to finish the computation and we proceed with changing the actual stage matrix in the global memory.

Considering that the reconversion affects the cells on the top, left, right and bottom of the cell under test, it is convenient to use the following variables:

```
int pos1 = (row - 1) * numColumnas + col;
int pos2 = row * numColumnas + col - 1;
int pos3 = (row + 1) * numColumnas + col;
int pos4 = row * numColumnas + col + 1;

char pos1_char, pos2_char, pos3_char, pos4_char;
```

Every time the algorithm detects a ship which can be potentially reconverted, we assign the values to the character variables as follows:

```
pos1_char = ty - 1 >= 0 ? stageSubmatrix[ty - 1][tx] : dev_Stage_Matrix[pos1];

pos2_char = tx - 1 >= 0 ? stageSubmatrix[ty][tx - 1] : dev_Stage_Matrix[pos2];

pos3_char = ty + 1 < TILE ? stageSubmatrix[ty + 1][tx] : dev_Stage_Matrix[pos3];

pos4_char = tx + 1 < TILE ? stageSubmatrix[ty][tx + 1] : dev_Stage_Matrix[pos4];
```

This way, in a single line we perform both the boundary check and the assignment.