Mapeo de Arnold usando la librería de computación basada en grafos Taskflow

Saul Andersson Rojas Coila
Departamento de Ciencia de la Computación
Universidad Católica San Pablo
Perú, Arequipa-Arequipa
Correo electrónico: saul.rojas@ucsp.edu.pe
18 de Mayo, 2022

***Resumen*—El mapeo de Arnold (también conocido como gato) es un método de transformación de imágenes utilizado en varias áreas de la computación entre ellas la encriptación, en este escrito se mostrará una implementación del algoritmo basándonos en la esquematización de tareas basadas en grafos con la librería de C++ Taskflow, el algoritmo se implementará en el paradigma de la computación heterogénea, básicamente la transformada se realizará en GPU y el resto de tareas auxiliares en la CPU. El algoritmo es utilizado como ejemplo para ver el rendimiento de la librería y aplicar los conocimientos básicos de los tipos de tareas que se pueden implementar con Taskflow. El código fuente lo puede encontrar en el siguiente enlace a github: https://github.com/webtaken/ArnoldCatMap-Taskflow.git**

1. Introducción

El mapeo de Arnold o más conocida como *Arnold's Cat Map* en honor al matemático Vladimir I. Arnold, es una transformación lineal empleada para la expansión y compresión continua de una imagen de dimensión NxN, la cual después de aplicarsele la transformación es reconstruida después de cierta cantidad de iteraciones. Esta transformación es de especial interés para las personas que realizan investigación en los sistemas caóticos y ecuaciones diferenciales porque presenta las mismas propiedades que muchos otros sistemas caóticos, entre ellas la ya mencionada reconstrucción de una imagen después de cierta cantidad de iteraciones, por esta característica es que se dice a esta transformación periódica después de n iteraciones.
En el presente documento se propondrá una implementación de esta transformación haciendo uso de la computación basada en grafos con la librería Taskflow.

2. Transformación de Arnold

En la imagen 1 se muestra la visualización de la transformación sobre una imagen dada. Para generar las imágenes resultantes de la transformación esta es aplicada sobre cada uno de los *pixeles* de una imagen con la siguiente fórmula.

$$\Gamma \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \text{modN}$$

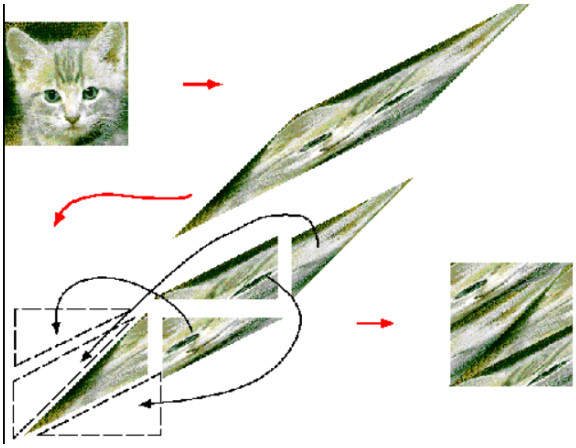

Figura 1. Ejemplo visual de la transformación de Arnold sobre una imagen [2].

Aquí el operador módulo es lo que hace que la transformación sea periódica después de cierta cantidad de iteraciones, p.ej. en el siguiente enlace Transformación de Arnold se ve una animación de la transformación sobre una imagen, primero a la iteración 2 se ve que la imagen se deforma de una forma irreconocible pero después de 15 iteraciones se regresa a la imagen original.

3. Taskflow

Taskflow [1] es una librería para la computación basada en grafos, la librería presenta varios tipos de tareas: estáticas, dinámicas, componibles, condicionales y *cudaflow*. Cada uno de este tipo de tareas realiza una tarea distinta, las estáticas son las más comunes y ejecutan tareas secuenciales en distintos nodos, las tareas dinámicas ejecutan un conjunto de subtareas dentro de estas para una mejor abstracción de procesos, las tareas componibles son aquellas que buscan unir modularmente diferentes tareas cuya fuente está dispersa en varios documentos de código fuente, las condicionales son aquellas que permiten la simulación de bifurcación en el recorrido de un grafo de tareas, estas pueden ser determinísticas o no determinísticas, y por último las *cudaflow* son la integración de este paradigma de programación con

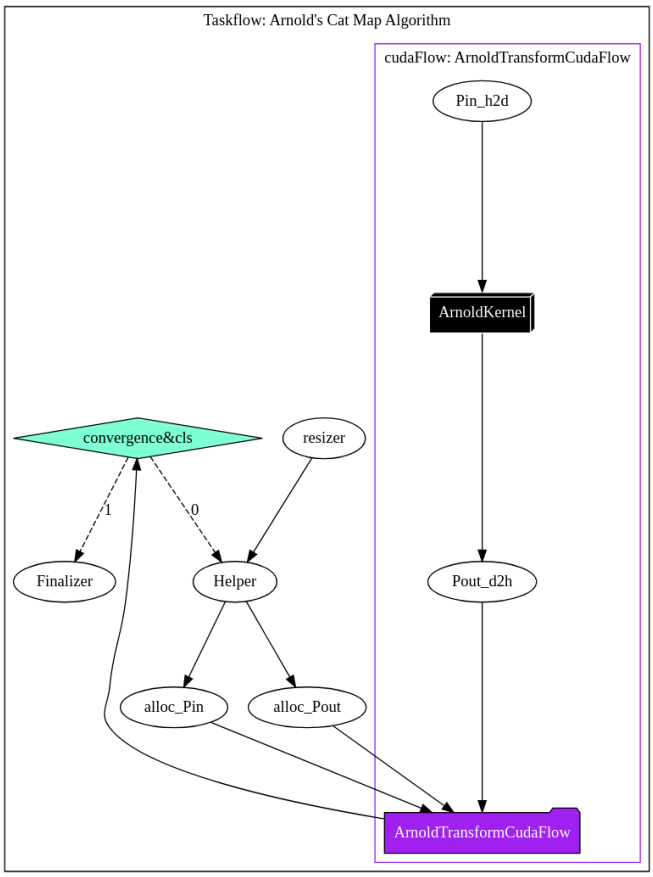

Figura 2. Flujo de tareas para la transformada de Arnold en Taskflow.

CUDA para la computación heterogénea.
En el proyecto los tipos de tareas en la librería esenciales serán las tareas estáticas, condicionales y *cudaflow*.

4. Descripción del proyecto

El proyecto busca la implementación de la transformación de Arnold haciendo uso de taskflow, el objetivo es contar la cantidad de veces que se realiza esta transformación, además de guardar las imágenes resultantes de cada una de las iteraciones de la transformación. En la imagen 4 se muestra el grafo de tareas correspondiente a la transformación de Arnold a implementarse en Taskflow.

Empezamos con la tarea de `resizer` la cual tomara una imagen a la cual se le aplicará la transformación, en esta primera etapa adaptaremos la imagen para que esta posea las dimensiones NxN necesarias para el algoritmo, después tenemos la función `Helper` que es una tarea puente en cada iteración, luego se llama a las funciones `alloc_Pin` y `alloc_Pout` las cuales alojarán en GPU la memoria para la imágen de entrada y de salida al momento de aplicar la transformada.
Hasta este punto solo hemos utilizado tareas estáticas, luego

Figura 3. Ilustración del proceso adaptativo para la resolución de una imagen. A y B Vertical, C y D horizontal.

se continúa con un conjunto de tareas de tipo *cudaflow*, con la función `Pin_h2d` básicamente copiamos del *host* al *device* la imagen de entrada, luego se procede con la función kernel `ArnoldKenel` la cual es la que realizará dicha transformación, una vez terminada la tarea se pasa a copiar el resultado del *device* al *host* para terminar el flujo *cudaflow*.
Por último se pasa a la tarea condicional `convergence&cls` que nos indica si la imagen resultante tras la transformación es igual a la original, esta función retornará 1 si ambas son las mismas con lo cual pasaremos a la tarea `Finalizer`, y 0 en caso contrario con lo cual regresamos a la tarea `Helper` para repetir el proceso hasta que se llegue a una convergencia final. Por último en la función `Finalizer` termina de liberar los recursos del programa y muestra las iteraciones totales.

4.1. Consideraciones para la implementación

La transformada de Arnold solo converge con imágenes de dimensiones NxN por ello antes de reservar memoria y enviar datos a la GPU se tiene que redimensionar la imagen en caso de que esta no contenga dicha característica, la adaptación simplemente rellenará de *pixeles* negros la imagen, en la imagen 3 se ve una ilustración del proceso. Este proceso adaptativo se ubicará en la función `resizer`, antes de trabajar con la GPU.

La versión de CUDA toolkit que se usa es la 11.7, con estándar C++17, todo ello para hacer uso de las funciones

`lambda` que son muy útiles a la hora de trabajar con Taskflow.
En cuanto a la dimensión de los bloques de hilos estos serán de dimensión 16x16x1 que operarán sobre toda la imagen hilo a hilo ejecutando la transformación.

5. Tareas empleadas en el algoritmo

En esta sección se explicarán los detalles en la implementación de cada tarea que aparece en el algoritmo, en la sección 4 se explicó brevemente lo que cada tarea realiza.

5.1. Tarea `resizer`

En la tarea `resizer` 1 se encarga de redimensionar la imagen dada para que sea cuadrada, en el código de la tarea la variable N es el máximo entre el ancho y alto de la imagen, los canales de la imagen son 3 RGB, tenemos tres posibiliadades que el ancho sea más grande que el alto, que el alto sea más grande que el ancho y que ambo sean iguales, en todos los casos excepto el tercero se debe redimenionar la imagen basándonos en esa característica.

```
    tf::Task resizer = taskflow.emplace([&]()
    {
        // la imagen adaptada es de NxNxChannels
        // inicializamos las variables h_Pin y
    h_Pout
        h_Pin = new unsigned char[N * N *
    img_props.desired_no_channels];
        h_Pout = new unsigned char[N * N *
    img_props.desired_no_channels];

        if(img_props.width > img_props.height)
        {
            // redimensionamos la imagen en base a
     su ancho
        }
        else // width <= height
        {
            // redimensionamos la imagen en base a
     su alto
        }

        // escribimos la imagen resultante de la
    adaptaci n en la misma carpeta
        stbi_write_jpg(transformed_image_name.
    c_str(), img_props.width, img_props.height,
        img_props.no_channels, h_Pin, 100);
    }
```

Listing 1. Tarea para la adaptación de la imagen

5.2. Tareas `Helper`, `alloc_Pin` y `alloc_Pout`

Como se mencionó la tarea `Helper` es una tarea puente, que no realiza nada, esto también se realiza de este modo ya que si conectáramos directamente de alguna manera la tarea `resizer` con la tarea `convergence&cls` se tendría una condición de carrera a nivel de tareas. Luego vienen las tareas `alloc_Pin` y `alloc_Pout` que reservan espacio en la GPU para las imágenes de entrada, ver el código 2, esto se realiza una sola vez, ya que se iterará varias veces hasta obtener la convergencia y se tendría un gran retraso si reservamos memoria y la eliminamos de la GPU varias veces, por ello se declaran dos variables `init1` e `init2` que son banderillas utilizadas para que solo se reserve memoria una sola vez.

```
    // una tarea puente
    tf::Task helper = taskflow.emplace(
        [&](){}).name("Helper");

    auto [alloc\_Pin, alloc\_Pout] = taskflow.
    emplace(
        [&]()
        {
            if (init1)
            {
                cudaMalloc(&d_Pin, N * N *
    img_props.desired_no_channels * sizeof(
    unsigned char));
                init1 = false;
            }
        },
        [&]()
        {
            if(init2) {
                cudaMalloc(&d_Pout, N * N *
    img_props.desired_no_channels * sizeof(
    unsigned char));
                init2 = false;
            }
        }
    );
```

Listing 2. Tareas de alojamiento y helper

5.3. Tarea `ArnoldTransformCudaFlow`

En esta tarea de tipo cudaflow, ver el código 3, se realizan las tareas para el flujo de trabajo en GPU, se compone de las subtareas `Pin_h2d`, `Pout_d2h` y `ArnoldKernel`.

```
tf::Task arnoldflow = taskflow.emplace([&](tf::
    cudaFlow &cf) {
    // transfiriendo datos de la imagen adaptada (
    host) a la memoria en el dispositivo
    tf::cudaTask Pin_h2d = cf.copy(d_Pin, h_Pin, N
     * N * img_props.desired_no_channels).name("
    Pin_h2d");
    // transfiriendo el resultado del dispositivo
    al host
    tf::cudaTask Pout_d2h = cf.copy(h_Pout, d_Pout
    , N * N * img_props.desired_no_channels).name(
    "Pout_d2h");

    // dimensiones del grid
    dim3 dimGrid(ceil(N / 16.0f), ceil(N / 16.0f),
     1);
    // dimensiones de los bloques
    dim3 dimBlock(16, 16, 1);

    // launch ArnoldKernel<<<dimGrid, dimBlock>>>(
    d_Pin, d_Pout, N, 3)
    tf::cudaTask ArnoldKernel = cf.kernel(
        dimGrid, dimBlock, 0,
    ArnoldTransformKernel, d_Pin, d_Pout, N,
        img_props.desired_no_channels).name("
    ArnoldKernel");

    // construimos el flujo de trabajo
```

```
    ArnoldKernel.succeed(Pin_h2d).precede(Pout_d2h
    );
}).name("ArnoldTransformCudaFlow");
```

Listing 3. Cudaflow de Arnold

La subtarea `Pin_h2d` se encarga de copiar los datos de la imagen en la CPU a la GPU, `h_Pin` a `d_Pin`. En el otro caso la subtarea `Pout_d2h` se encarga de transferir el resultado después de aplicado el *kernel* de la transformada hacia la imagen de salida en la CPU, de `d_Pout` hacia `h_Pout`. Por último la llamada a la función *kernel* `ArnoldTransformKernel`, ver el código 4, se realiza en la subtarea `ArnoldKernel` del cudaflow.
Como se especificó la dimensión de los bloques serán de 16x16x1, ver la línea 8 y 10 del código 3.

```
// Arnold's Cat Map Kernel
__global__ void ArnoldTransformKernel(unsigned
    char* Pin, unsigned char* Pout, int N, int
    channels)
{
    int Col = threadIdx.x + blockDim.x * blockIdx.
    x;
    int Row = threadIdx.y + blockDim.y * blockIdx.
    y;

    if(Col < N && Row < N){
        // Ver el siguiente enlace para ver la
    formula de la transformada de Arnold
        // http://fibonacci.math.uri.edu/~kulenm/
    diffeqaturi/victor442/index.html
        int newCol = (Col + Row) % N;
        int newRow = (Col + 2*Row) % N;

        int offset = (Row * N + Col) * channels;
        int newOffset = (newRow * N + newCol) *
    channels;

        // Valores RGB
        Pout[newOffset] = Pin[offset]; // R
        Pout[newOffset + 1] = Pin[offset + 1]; //
    G
        Pout[newOffset + 2] = Pin[offset + 2]; //
    B
    }
}
```

Listing 4. Kernel de la transformada de Arnold

5.4. Tarea `convergence&cls`

Esta tarea 5 se encarga de verificar que la imagen resultante de la N-ésima iteración de la transformada de Arnold sea igual a la imagen original, en caso de que sean iguales se debe

```
tf::Task convergence_checker = taskflow.emplace
    ([&]() {
    // se llego al tope de las iteraciones
    if(iter_transform >= max_iters){
        // Antes liberamos la memoria reservada en
     la GPU y CPU
        return 1;
    }

    // compararemos la imagen original img con el
    resultado de la transformaci n h_Pout
    for (int j = 0, y = 0; j < img_props.height; j
    ++, y++)
    {
        for (int i = 0; i < img_props.width; i++)
        {
            int offset_orig = (j * img_props.width
     + i) * img_props.desired_no_channels;
            int offset_transform = (y * N + i) *
    img_props.desired_no_channels;
            // verificamos que ambas imagenes sean
     iguales
            if (h_Pout[offset_transform] != img[
    offset_orig])
            {
                // copiamos los datos de h_Pout
    hacia h_Pin para la siguiente iteracion
                memcpy(h_Pin, h_Pout, N * N *
    img_props.desired_no_channels * sizeof(
    unsigned char));

                // antes guardaremos la imagen de
    la iteracion N esima del proceso de la
    transformada
                unsigned char *tmp_img = new
    unsigned char[img_props.width * img_props.
    height

          * img_props.desired_no_channels];

                // rellenamos tmp_img con la
    imagen resultante

                std::string transformed_image_name
     = img_filename + "_arnold_iter_"

          + std::to_string(iter_transform) +
    img_extention;
                iter_transform++; // incrementamos
     la iteracion
                // escribimos la imagen resultante
     en la misma carpeta
                stbi_write_jpg(
    transformed_image_name.c_str(), img_props.
    width, img_props.height,
                                        img_props.
    no_channels, tmp_img, 100);

                    // siempre debemos liberar la
    memoria
                    delete [] tmp_img;
                    // volvemos a la primera
    funcion
                    return 0;
                }
            }
        }

        // copiamos los datos de h_Pout hacia
    h_Pin para copiar el resultado final
        memcpy(h_Pin, h_Pout, N * N * img_props.
    desired_no_channels * sizeof(unsigned char));
        // antes guardaremos la ultima imagen de
    la iteracion N esima del proceso de la
    transformada de Arnold
        unsigned char *tmp_img = new unsigned char
    [img_props.width * img_props.height

          * img_props.desired_no_channels];

        std::string transformed_image_name =
    img_filename + "_arnold_iter_"
```

```
                                    + std::
    to_string(iter_transform) + img_extention;
        // escribimos la imagen resultante en la
    misma carpeta
        stbi_write_jpg(transformed_image_name.
    c_str(), img_props.width, img_props.height,
                                    img_props.
    no_channels, tmp_img, 100);
        // En este caso se ha llegado a la imagen
    original, entonces pasamos a la tarea final
        // Antes liberamos la memoria reservada en
     la GPU y CPU
        return 1;
    }).name("convergence&cls");
```

Listing 5. Verificador de convergencia

5.5. Tarea `Finalizer`

Esta tarea se encarga simplemente de mostrar la cantidad de iteraciones que tomó para llegar a la imagen original, ver el código 6.

```
tf::Task finalizer = taskflow.emplace(
    [&]()
    {
        std::cout << "Arnold Transformation ended\
    n";
        std::cout << "Image path: " << img_path <<
     "\n";
        std::cout << "With " << iter_transform <<
    " iterations\n";
        // Terminamos las tareas liberando los
    recursos empleados
        // siempre debemos liberar la memoria
        stbi_image_free(img);

        // liberamos la memoria de ambas imagenes
    en CPU
        if (h_Pin != nullptr)
        {
            delete[] h_Pin;
        }
        if (h_Pout != nullptr)
        {
            delete[] h_Pout;
        }
    }).name("Finalizer");
```

Listing 6. Tarea Final

5.6. Uniendo todo

Por último se muestra 7 la construcción del grafo usando las funciones de enlazado de grafos de Taskflow.

```
// Ahora construimos el grafo de tareas
resizer.precede(helper);
helper.precede(alloc_Pin, alloc_Pout);
arnoldflow.succeed(alloc_Pin, alloc_Pout);
arnoldflow.precede(convergence_checker);
convergence_checker.precede(helper, finalizer);
```

Listing 7. Unificando tareas

```
saul@saul-ubuntu:~/Desktop/UCSP/Paralelos/Laboratorios/CUDA$ ./ArnoldTransform i
mgs/test1/Lenna.jpg
Loaded image characteristics:
width: 512px
height: 512px
original N channels: 3
loaded with N channels: 3
Arnold Transformation ended
Image path: imgs/test1/Lenna.jpg
With 384 iterations
```

Figura 4. Resultado de la ejecución.

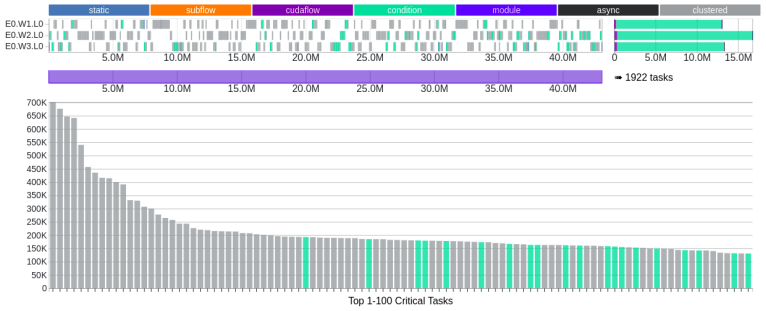

Figura 5. *Profile* del algoritmo de Arnold.

6. Resultados

A continuación se muestran los resultados de la ejecución del algoritmo sobre una imagen de 512x512 pixeles, después de 384 iteraciones se regresa a la imagen original.

Taskflow también viene con un analizador o *profiler* que permite el análisis del impacto de las tareas sobre un grafo de tareas, se puede comprobar el impacto de cada tarea y su tipo al momento de ser ejecutado el programa, en la imagen 5 se muestra el resultado para el algoritmo sobre la imagen de 512x512 pixeles, vemos que el mayor impacto se lo llevan las tareas clusterizadas.

7. Conclusión

Taskflow es una librería muy útil para el modelado de tareas basadas en grafos, en este documento se implementó el algoritmo de Arnold aplicado en muchas áreas de la computación entre ellas la encriptación, la abstracción de todo el proceso basado en un grafo es bastante sencillo con taskflow las funciones de enlazado son bastante explícitas además que pueden ser fácilmente modularizadas en diferentes archivos, la única desventaja es que la librería no posee la característica de devolver menssajes para la corrección de errores como la condición de carrera en tareas o la sugerencia de mejores modelos a la hora de crear lo grafos, por último el uso de un estándar reciente como lo es C++ 2017 le brinda a la librería la ventaja de un mantenimiento sostenible en el tiempo.

Referencias

[1] T. -W. Huang, D. -L. Lin, C. -X. Lin and Y. Lin, ”Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,ïn

IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 6, pp. 1303-1320, 1 June 2022, doi: 10.1109/TPDS.2021.3104255.

[2] Arnold's Cat Map. [Online]. Available: http://fibonacci.math.uri.edu/ kulenm/diffeqaturi/victor442/index.html. [Accessed: 22-Jun-2022].