TESIS MAESTRÍA EN FÍSICA

FRENTES DE ONDA EN ECUACIONES DE REACCIÓN-DIFUSIÓN-CONVECCIÓN SOBRE MEDIOS HETEROGÉNEOS

Lic. Renzo Zagarra Saez
Maestrando

Dr. Alejandro Kolton
Director

Miembros del Jurado
Dr. Ezequiel Ferrero

24 de Octubre de 2022

Teoría de la Materia Condensada – Centro Atómico Bariloche

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

A mis padres, Francisco y Liliana,
por su apoyo incondicional.
A mis hermanos, Franco y Lucas,
por hacer de cada momento una sonrisa.
A mi novia, Sol,
por acompañarme y entenderme como nadie.

Índice de contenidos

Índice de símbolos

$S(x, y, t)$ Fracción de susceptibles en la posición (x, y) en el instante t.
$I(x, y, t)$ Fracción de infectados en la posición (x, y) en el instante t.
$R(x, y, t)$ Fracción de recuperados en la posición (x, y) en el instante t.
β Tasa de transmisión.
γ Tasa de recuperación.
S_0 Distribución inicial de la fracción de susceptibles.
S_c Fracción de susceptibles crítica.
R_0 Coeficiente de reproducción.
D_x Coeficiente de dispersión de x.
$u(y, t)$ Campo de desplazamiento del frente de onda.
$u_{cm}(t)$ Centro de masa del frente de onda.
I_{max} Amplitud media del frente de onda.
c Velocidad media del frente de onda.
$w(t)$ Rugosidad del frente de onda.
$S(q)$ Factor de estructura del campo de desplazamiento del frente de onda.
$f_I(t)$ Perfil centrado del frente de onda.
$\beta_{\mathbf{r}}$ Distribución espacial de la tasa de transmisión.
H Medio homogéneo
DA Heterogeneidad dicotómica-aleatoria.
S Heterogeneidad suavizada.
DC Heterogeneidad dicotómica-correlacionada.

Índice de figuras

Índice de tablas

Resumen

Se estudió la propagación de frentes de onda gobernados por ecuaciones de reacción-difusión en el marco del modelo SIR espacial. Dichos frentes de onda podrían utilizarse para caracterizar frentes de infección en una problemática epidemiológica o bien orientarse a una problemática completamente diferente como lo son los frentes de incendios. Se definió una metodología estadística para la caracterización de los frentes de onda a partir de la cual se obtuvieron resultados cuantitativos respecto de la velocidad, la amplitud media, las propiedades geométricas e incluso la nocividad de los frentes sobre diferentes medios isotrópicos. En particular, se exploraron medios homogéneos, desordenados y correlacionados a partir de lo cual pudo describirse cuantitativamente el efecto que tenía cada uno de ellos sobre las características del frente de onda.

Se realizaron simulaciones numéricas masivas para resolver el sistema de ecuaciones de reacción-difusión involucrado en la dinámica. Estas se implementaron de manera eficiente utilizando computación acelerada a través de programación en paralelo sobre procesadores gráficos. De esta manera fue posible obtener resultados sobre sistemas a gran escala en tiempos razonables.

Palabras clave: SISTEMAS COMPLEJOS, MEDIOS DESORDENADOS, ECUACIONES DE DIFUSIÓN, MODELO SIR

Abstract

The propagation of wave fronts governed by reaction-diffusion equations were studied within the framework of the spatial SIR model. These wave fronts could be used to characterize infection fronts in an epidemiological problem or be oriented to a completely different problem such as fire fronts. A statistical methodology was defined for the characterization of the wave fronts from which quantitative results were obtained regarding the speed, the mean amplitude, the geometric properties and even the harmfulness of the fronts on different isotropic media. In particular, homogeneous, disordered and correlated media were explored, from which it was possible to quantitatively describe the effect that each of them had on the characteristics of the wavefront.

Massive numerical simulations were performed to solve the system of reaction-diffusion equations involved in the dynamics. These were efficiently implemented using accelerated computing through parallel programming on graphics processors. In this way it was possible to obtain results on large-scale systems in reasonable times.

Keywords: DYNAMIC SYSTEMS, DISORDERED MEDIA, DIFFUSION EQUATIONS, SIR MODEL

Capítulo 1

Modelo SIR

1.1. Modelo SIR

Capítulo 2

Reseña numérica y computacional

La idea de este capítulo es dejar documentación que muestre con claridad el trabajo de investigación y elaboración técnico, a nivel computacional, realizado para llevar a cabo este proyecto de tesis. Adicionalmente, está en mi intención ser lo más genérico, claro y acotado posible, para que sea de utilidad a cualquier otra persona que esté interesada en resolver ecuaciones diferenciales parciales haciendo uso de programación en paralelo. En particular, usando *CUDA* a través de las bondades que ofrece *Python* mediante la librería *CuPy*.

Este capítulo cuenta con un material complementario en formato de *Jupyter Notebook* en *Google Colab* al cual puede acceder desde aquí. *Google Colab* da acceso gratuito a GPUs, lo cual está fantástico para aprender a usarlas, aunque evidentemente es con tiempo limitado. En este *Jupyter Notebook* esencialmente encontrará todo el código prensentado aquí y un poco más, para que pueda interactuar y hacer las modificaciones que quiera.

A continuación dejamos constancia del *software* y *hardware* utilizado en la ejecución de los códigos de este capítulo.

- Python: 3.9.7
- CUDA Version: 11.6
- CPU: AMD Ryzen 9 5900HX
- GPU: NVIDIA GeForce RTX 3060 Laptop
- Memoria de GPU: 6GB
- CUDA cores: 3840
- RAM: 32GB

2.1. Diferencias finitas

El objetivo es resolver numéricamente ecuaciones diferenciales parciales de la manera más simple y eficiente posible. Para la física este tipo de ecuaciones son de gran interés ya que se usan ampliamente para modelar todo tipo de fenómenos.

Para reducir la complejidad del problema, acotamos la discusión mostrando en detalle el proceso de resolución de un sistema de dos ecuaciones de reacción-difusión con dos dimensiones espaciales (x, y) en cierta región $\Omega \in \mathbb{R}^2$. Esto es conveniente porque este problema corresponde con el tipo de sistemas usados en este trabajo. Explícitamente, queremos resolver,

$$\begin{aligned} \partial_t u &= f_u(u, v) + D_u \nabla^2 u \\ \partial_t v &= f_v(u, v) + D_v \nabla^2 v, \end{aligned} \tag{2.1}$$

donde u y v son las variables dinámicas que dependen de las variables (t, x, y), f_u y f_v son funciones suaves, mientras que D_u y D_v son los coeficientes de difusión de u y v respectivamente. Estas ecuaciones deben resolverse teniendo en cuenta determinadas condiciones iniciales y de contorno para el problema en cuestión, podemos expresarlas genéricamente de la siguiente manera,

$$\begin{array}{llll} u(t=0,x,y)=g_u(x,y) & \forall x,y\in\Omega & | & u(t,x,y)=h_u(t,x,y) \quad \forall t\in\mathbb{R};\ \forall x,y\in\delta\Omega \\ v(t=0,x,y)=g_v(x,y) & \forall x,y\in\Omega & | & v(t,x,y)=h_v(t,x,y) \quad \forall t\in\mathbb{R};\ \forall x,y\in\delta\Omega. \end{array}$$

Donde las funciones g_u y g_v denotan las condiciones iniciales del sistema, las funciones h_u y h_v las condiciones de contorno y $\delta\Omega$ es el borde de Ω.

De esta manera queda completamente definido el problema y procedemos al armado del esquema numérico necesario para resolverlo. Por simplicidad consideramos que Ω es una región rectangular del plano donde $x, y \in [0, L_x]\times[0, L_y]$. Segmentamos este espacio en $N_x \times N_y$ cuadrantes de dimensiones $d_x = L_x/N_x$ y $d_y = L_y/N_y$ y denommina-mos u_{ij} y v_{ij} a los valores de las funciones u y v a tiempo t en el cuadrante (i, j) corespondiente a la región $[jd_x, (j+1)d_x) \times [id_y, (i+1)d_y)$, con $i = 0, 1, ..., N_y - 1$ y $j = 0, 1, ..., N_x - 1$.

A continuación, la idea consiste en reducir el sistema de ecuaciones diferenciales parciales 2.1 en un sistema de ecuaciones diferenciales ordinarias, donde cada ecuación describe la evolución de las variables dinámicas en un cuadrante distinto. Para ello necesitamos llevar los laplacianos de las ecuaciones al nuevo esquema espacial discretizado. Lo hacemos usando la siguiente aproximación por diferencias finitas para los laplacianos,

$$\begin{aligned} (\nabla^2 u)_{ij} &= \frac{1}{d^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) \\ (\nabla^2 v)_{ij} &= \frac{1}{d^2}(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{ij}) \end{aligned} \tag{2.2}$$

donde usamos que $d = d_x = d_y$. Utilizando la notación dada para la discretización espacial, tenemos el siguiente sistema de ecuaciones diferenciales ordinarias,

$$\begin{aligned} \frac{\mathrm{d}u_{ij}}{\mathrm{d}t} &= f_u(u_{ij}, v_{ij}) + D_u(\nabla^2 u)_{ij} \\ \frac{\mathrm{d}v_{ij}}{\mathrm{d}t} &= f_v(u_{ij}, v_{ij}) + D_v(\nabla^2 v)_{ij}. \end{aligned} \tag{2.3}$$

Finalmente, discretizamos el espacio temporal con un intervalo dt y aproximamos la derivada temporal a primer orden. Usando $n \in \mathbb{N}$ como ínidce temporal, notamos u^n_{ij} y v^n_{ij} como el valor de las funciones u y v en el instante $t = n*dt$ sobre el cuadrante (i, j). De esta manera obtenemos el siguiente esquema explícito de Euler para la resolución numérica del sistema 2.1,

$$\begin{aligned} u^{n+1}_{ij} &= u^n_{ij} + dt\ \left(f_u(u^n_{ij}, v^n_{ij}) + D_u(\nabla^2 u)^n_{ij}\right) \\ v^{n+1}_{ij} &= v^n_{ij} + dt\ \left(f_v(u^n_{ij}, v^n_{ij}) + D_v(\nabla^2 v)^n_{ij}\right), \end{aligned} \tag{2.4}$$

a partir del cual podemos iterar para obtener la solución. En la siguiente sección se decribe cómo hacer esto usando *Python* con diferentes niveles de eficiencia.

2.2. Implementaciones en *Python*

En esta sección se describen brevemente 4 implementaciones distintas para la resolución del esquema numérico 2.4 hayado en la sección anterior. Comenzamos por la más ineficiente, que correponde con el uso de la librería *NumPy*

en un esquema serial (mucho peor aún es *Python* nativo, pero vamos a saltearnos este caso), hasta llegar a la más eficiente obtenida, correspondiente a la utilización de *CUDA* a través de la librería *CuPy*.

La razón para hacer esto reside nuevamente en la idea de dejar documentación que pueda llegar a ser de utilidad para alguien más lidiando con problemas de índole numérico donde la eficiencia de la resolución es relevante. Además, antes de llegar a la versión final, donde usamos *CUDA* y consecuentemente es necesario un procesador gráfico (GPU) compatible con *CUDA*, se muestran implementaciones que son notablemente eficientes sin necesidad de un GPU, y que pueden ser usadas por una computadora más modesta.

2.2.1. Implementación con *NumPy*

NumPy es una de las librerias fundamentales para hacer computación científica en *Python*. Ofrece los invaluables *NumPy Arrays*, que están diseñados específicamente para ser lo más eficiente posible en la manipulación numérica sin perder la simplicidad y elegancia de *Python*. Utilizando *NumPy* se obtienen mejores resultados que usando *Python* nativo porque integra código de *C,C++ y Fortran* dentro de *Python*, adicionalmente la mayoría de los métodos implementados para la operación con *NumPy arrays* están paralelizados.

El código 2.1 muestra la implementación propuesta usando *NumPy*. Nótese que hay más líneas de comentarios que de código. Esencialemente, primero definimos una función que llamamos *laplacian*, que toma un *array* 2D y devuelve su laplaciano, para ello usamos la función *roll* de *NumPy* que desplaza un *array* sobre un eje dado cuanto queramos y con condiciones periódicas. Luego, sumamos y restamos estas matrices desplazadas según 2.2 y obtenemos el laplaciano. Finalmente, definimos la función *cpu_ numpy_ solver*, que toma por argumentos las condiciones iniciales del problema, las funciones de reacción f_u y f_v, los coeficientes de difusión, el intervalo de integración temporal dt y espacial d y la cantidad de iteraciones it.

```python
import numpy as np

def laplacian(X):
  '''
  Take the Laplacian of a 2D array with periodic boundary conditions.
  '''
  return np.roll(X,1,axis = 0) + np.roll(X,-1,axis = 0) + np.roll(X,1,axis = 1) + np.roll(X,-1,axis = 1) - 4*X

def cpu_numpy_solver(u, v, fu, fv, Ds, dt = .01, d = 1, it = 1000):
  '''
  Solve a 2D reaction-diffusion equation for two dynamical variables with periodic boundary conditions using NumPy.

  u: Intial conditions for the first dynamical variable. 2d NumPy array.
  v: Intial conditions for the second dynamical variable. 2d NumPy array.
  fu: Reaction term for the first dynamical variable. Function.
  fv: Reaction term for the second dynamical variable. Function.
  Ds: Diffusion coefficients for the first and second dynamical variables. List or array of length 2.
  dt: Time step. Default value is 0.01.
  d: Space step. Default value is 1.
  it: Number of iterations. Default value is 1000.
  '''

  Du,Dv = Ds
  for _ in range(it):
      u = u + dt*(fu(u,v) + Du*laplacian(u)/d**2)
      v = v + dt*(fv(u,v) + Dv*laplacian(v)/d**2)

  return u,v
```

Código 2.1: Implementación con NumPy.

```python
def fu(u,v,beta):
  return -beta*u*v

def fv(u,v,beta,gamma):
  return beta*u*v - gamma*v

L = 1024
beta = 1
```

```
gamma = .2
u = np.ones((L,L))
v = np.zeros((L,L))
u[:,0] = 0
v[:,0] = 1
Ds = [0,1]

uf,vf = cpu_numpy_solver(u,v,fu,fv,Ds)
```

Código 2.2: Ejemplo de uso de la implementación con *NumPy*.

En el código 2.2 se muestra un ejemplo de uso utilizando las siguientes funciones de reacción,

$$\begin{aligned} f_u(u,v) &= -\beta uv, \\ f_v(u,v) &= \beta uv - \gamma v, \end{aligned} \tag{2.5}$$

donde β y γ son constantes.

Finalmente, a continuación se muestra la velocidad de la resolución con un sistema de 1024×1024 utilizando todos los parámetros y funciones del ejemplo 2.2. Para una justa comparación con las demás implementaciones, utilizaremos exactamente los mismos parámetros y funciones.

```
  %timeit cpu_numpy_solver(u,v,fu,fv,Ds)
  1min +- 1.53 s per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

2.2.2. Implementación serial con *Numba*

Otra de las librerías fundamentales para cálculo numérico en *Python* es *Numba*. *Numba* toma código nativo de *Python* y genera código de máquina optimizado usando *LLVM compiler infrastructure*. También es capaz de procesar *NumPy* con una gran cantidad de sus métodos.[1]

Lo mejor de todo esto es que *Numba* lo hace de manera completamente autónoma, solo hay que decirle que lo haga. Para ello utilizamos el decorador *@njit*, que indica a *Numba* que la función en cuestión debe ser procesada. El código 2.3 muestra cómo sería la implementaciones en este caso. Se define la función *cpu_numba_solver* que toma los mismos argumentos que la función *cpu_numpy_solver* vista anteriormente. Dentro de la función iteramos temporalmente y para calcular los laplacianos y términos de reacción en cada cuadrante recorremos las matrices con un ciclo *for*. Usualmente es posible tomar las funciones o implementaciones realizadas con *Numpy* y decorarlas con *@njit* para obtener el resultado deseado. Sin embargo, en este caso no podemos hacerlo con las funciones del código 2.1 porque *Numba* no soporta la función *roll* de *NumPy*. Por lo cual estamos forzados a calcular el laplaciano de una manera más explicita para que *Numba* lo entienda.

```
from numba import njit
import numpy as np

@njit()
def cpu_numba_solver(u, v, fu, fv, Ds, dt=.01, d = 1, it = 1000):
  '''
  Solve a 2D reaction-diffusion equation for two dynamical variables with periodic boundary conditions using Numba
  in a serial implementation.

  u: Intial conditions for the first dynamical variable. 2d NumPy array of shape (Ly,Lx).
  v: Intial conditions for the second dynamical variable. 2d NumPy array of shape (Ly,Lx).
  fu: Reaction term for the first dynamical variable. Function.
  fv: Reaction term for the second dynamical variable. Function.
  Ds: Diffusion coefficients for the first and second dynamical variables. List or array of length 2.
  dt: Time step. Default value is 0.01.
  d: Space step. Default value is 1.
  it: Number of iterations. Default value is 1000.
  '''

  Ly,Lx = u.shape
```

[1] Para ver más detalles consultar la documentación de *Numba*.

```
  u = u.reshape(Ly*Lx)
  v = v.reshape(Ly*Lx)
  Fu = np.zeros_like(u)
  Fv = np.zeros_like(v)
  Du,Dv = Ds
  for _ in range(it):
    for i in range(Lx*Ly):
      x = i % Lx
      y = i // Lx
      Lu = (u[(x+1)%Lx + Lx*y] + u[(x-1+Lx)%Lx+Lx*y] + u[x + Lx*((y+1)%Ly)] + u[x + Lx*((y-1+Ly)%Ly)] - 4*u[i])/d**2
      Lv = (v[(x+1)%Lx + Lx*y] + v[(x-1+Lx)%Lx+Lx*y] + v[x + Lx*((y+1)%Ly)] + v[x + Lx*((y-1+Ly)%Ly)] - 4*v[i])/d**2
      Fu[i] = fu(u[i],v[i]) + Du*Lu
      Fv[i] = fv(u[i],v[i]) + Dv*Lv
    u = u + dt*Fu
    v = v + dt*Fv
  return u.reshape(Ly,Lx),v.reshape(Ly,Lx)
```

Código 2.3: Implementación serial con *Numba*.

En cuanto al ejemplo de uso, sería idéntico al mostrado en 2.2, con la única salvedad de que las funciones f_u y f_v también deben estar decoradas con *@njit*. A continuación se muestra el tiempo de resolución para sistemas de 1024×1024 en esta versión. Resulta cerca de 3 veces más rápido que 2.1.

```
%timeit cpu_numba_solver(u,v,fu,fv,Ds)
18.1 s +- 564 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

2.2.3. Implementación paralela con *Numba*

En esta ocasión la idea es utilizar *Numba* aprovechando que una parte del algoritmo se puede realizar en paralelo. Esto simplemente quiere decir que hay un conjunto de operaciones que puede realizarse en simultáneo en lugar de una por una. Este es el caso para el ciclo *for* que se utiliza para calcular los laplacianos y términos de reacción en cada cuadrante. Es decir, no es necesario calcular el valor correspondiente al cuadrante (i, j) para luego calcular el del cuadrante (i', j'), se pueden calcular simultáneamente, en paralelo.

Lo mejor del caso, nuevamente, es que podemos indicarle de una manera muy sencilla a *Numba* que determinado *for* es paralelizable y listo, *Numba* se encargará de darle las instrucciones en paralelo al procesador. Para ello, todo lo que tenemos que hacer es importar la función *prange* de *Numba* que sirve para indicarle a *Numba* que el ciclo *for* es paralelizable y pasar la opción *parallel* = *True* al decorador *@njit*. Por completitud mostramos el código de la función *cpu_ numba_ parallel_ solver* en 2.4, pero las únicas diferencias con la versión serial 2.3 son las indicadas aquí.

```
from numba import njit,prange
import numpy as np

@njit(parallel = True)
def cpu_numba_parallel_solver(u, v, fu, fv, Ds, dt=.01, d = 1, it = 1000):
  '''
  Solve a 2D reaction-diffusion equation for two dynamical variables with periodic boundary conditions using Numba
  with a parallel implementation.

  u: Intial conditions for the first dynamical variable. 2d NumPy array of shape (Ly,Lx).
  v: Intial conditions for the second dynamical variable. 2d NumPy array of shape (Ly,Lx).
  fu: Reaction term for the first dynamical variable. Function.
  fv: Reaction term for the second dynamical variable. Function.
  Ds: Diffusion coefficients for the first and second dynamical variables. List or array of length 2.
  dt: Time step. Default value is 0.01.
  d: Space step. Default value is 1.
  it: Number of iterations. Default value is 1000.
  '''

  Ly,Lx = u.shape
  u = u.reshape(Ly*Lx)
  v = v.reshape(Ly*Lx)
  Fu = np.zeros_like(u)
  Fv = np.zeros_like(v)
  Du,Dv = Ds
```

```
26
27  for _ in range(it):
28    for i in prange(Lx*Ly):
29      x = i % Lx
30      y = i // Lx
31      L_u = (u[(x+1)%Lx + Lx*y] + u[(x-1+Lx)%Lx+Lx*y] + u[x + Lx*((y+1)%Ly)] + u[x + Lx*((y-1+Ly)%Ly)] - 4*u[i])/d**2
32      L_v = (v[(x+1)%Lx + Lx*y] + v[(x-1+Lx)%Lx+Lx*y] + v[x + Lx*((y+1)%Ly)] + v[x + Lx*((y-1+Ly)%Ly)] - 4*v[i])/d**2
33      Fu[i] = fu(u[i],v[i]) + Du*L_u
34      Fv[i] = fv(u[i],v[i]) + Dv*L_v
35    u = u + dt*Fu
36    v = v + dt*Fv
37  return u.reshape(Ly,Lx),v.reshape(Ly,Lx)
```

Código 2.4: Implementación paralela con *Numba*.

A continuación mostramos el rendimiento obtenido con esta nueva versión, observamos que es cerca de 3 veces más rápido que la versión serial 2.3 y 10 veces más rápido que la versión de *NumPy* 2.1. Esto muestra una primera aproximación al poder de la programación en paralelo y cómo es posible implementarla sin la necesidad de una GPU que naturalemente provee una arquitectura diseñada específicamente para hacer operaciones en paralelo de manera masiva.

```
%timeit cpu_numba_parallel_solver(u,v,fu,fv,Ds)
6.31 s +- 439 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

2.2.4. Implementación con *CuPy*

CuPy es una librería de código abierto desarrollada para facilitar el acceso a las GPU compatibles con *CUDA*[2] en *Python*. *Cupy* ofrece prácticamente todos los métodos de *NumPy* y se encarga de lidiar de forma autónoma y eficiente con el problema de pasar las instrucciones a la GPU. Esta característica por sí sola ya es sorprendente, dado que ofrece la posibilidad de explotar las capacidades de la GPU sin saber nada de CUDA. Ello quiere decir, que en muchos casos basta escrbir el código tal como lo haríamos con *NumPy* pero usando *CuPy*, y eso bastaría para tener una mejora decisiva en la eficiencia. De hecho, hagamos la prueba, si tomamos el código 2.1 y lo único que hacemos es cambiar *NumPy* por *CuPy*, el resultado obtenido es el siguiente.

```
%timeit gpu_simple_cupy_solver(u,v,fu,fv,Ds)
3.12 s +- 3.51 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Esto es cerca de 19 veces más rápido que la versión de *NumPy* y además la más rápida hasta ahora, lograda con un esfuerzo mínimo. Cabe recordar que el tamaño del sistema que estamos resolviendo, es de 1024×1024 en todos los casos, este es un terreno favorable para el uso de GPU, dado que es un sistema lo suficientemente grande como para que la capacidad de paralelización másiva de la GPU marque la diferencia. Esto es de carácter elemental en lo que respecta al uso de procesadores gráficos para cálculo numérico, sin embargo no está de más recordarlo. No siempre es preferible usar la GPU, hay que ponderar el carácter del algoritmo a utilizar y la magnitud de operaciones susceptibles de ser paralelizadas. Si corremos los mismos códigos con un sistema de 100×100, las cosas cambian rotundamente.

```
#Sistemas de 100x100
#Numpy
%timeit cpu_numpy_solver(u,v,fu,fv,Ds)
163 ms +- 502 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
#Cupy
%timeit gpu_simple_cupy_solver(u,v,fu,fv,Ds)
1.07 s +- 3.91 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Ahora bien, volviendo a sistemas grandes, podría decirse que una mejora en un factor 19 es sorprendente, sin embargo esto es así solo si comparamos con la implementación de *NumPy*, pero si miramos la mejora respecto de la implementación con *Numba* en paralelo, el factor de mejora es cerca de 2. En este caso alguien podría decir, con razón, que la utilización de la *GPU* no está justificada, dado que la mejora en eficiencia no vale el costo que implica el acceso a la GPU.

[2]Está en fase experimental la posibilidad de usar GPUs con ROCm.

Para sortear esta razonable objeción, solo debemos profundizar un poco más en las herramientas que nos ofrece *CuPy*. Como decíamos al principio, el hecho de que *CuPy* ofrezca la posibilidad de acceder a la computación en GPU de una manera extremadamente sencilla y con buenos resultados, es sorprendente. Sin embargo, como es habitual, esa sencillez no viene gratis, paga un peaje a la potencia de cómputo real que puede ofrecer una GPU.

En lo que sigue se muestra cómo es posible sacar más provecho de la GPU, sin salirnos del entorno que ofrece *CuPy*. La razón por la que estamos desperdiciando potencia de cómputo al resolver el problema reemplazando *CuPy* por *NumPy* es porque estamos lanzando demasiados *kernels* de manera de innecesaria. Es radicalmente más eficiente si conseguimos juntar todas las operaciones necesarias en una menor cantidad de *kernels*. Por cierto, se le dice *kernel* a una función que se ejecuta en la GPU, comúnmente esta función representan operaciones elementales que se realizan en paralelo en la GPU. Para entender esto, vamos a ver un ejemplo sencillo antes de atacar el problema original.

Supongamos que queremos multiplicar las matrices A y B, elemento a elemento, y luego sumar el resultado a la matriz C. Utilizando *CuPy arrays* la función en cuestión y el resultado obtenido es el siguiente.

```
import cupy as cp
def mul_add(A,B,C):
    return A*B + C

L = 1024
A = cp.ones((L,L))
B = cp.ones((L,L))
C = cp.ones((L,L))
mul_add(A,B,C)
%timeit mul_add(A,B,C)
215 us +- 6.81 us per loop (mean +- std. dev. of 7 runs, 1,000 loops each)
```

Ahora bien, podemos hacerlo mejor, el problema con la función anterior es que está utilizando dos *kernels* en lugar de uno, uno para multiplicar y el otro para sumar. Sin embargo sería mejor si pudieramos usar un solo *kernel* que hiciera las dos cosas a la vez. Para ello, *CuPy* ofrece la posibilidad de escribir *kernels* personalizados, una de las maneras de hacerlo, es utilizando la función *cupy.ElementwiseKernel*. A continuación se muestra cómo quedaría la función y su resultado utilizando esta alternativa.

```
import cupy as cp
mul_add_kernel = cp.ElementwiseKernel(
  'float64 A, float64 B, float64 C', 'float64 out',
  'out = A*B + C', 'mul_add')

L = 1024
A = cp.ones((L,L))
B = cp.ones((L,L))
C = cp.ones((L,L))
mul_add_kernel(A,B,C)
%timeit mul_add_kernel(A,B,C)
136 us +- 273 ns per loop (mean +- std. dev. of 7 runs, 10,000 loops each)
```

Vemos que la velocidad aumentó apreciablemente aún en un ejemplo tan sencillo como este, si aplicamos esta misma idea a algoritmos más complejos tenemos la posibilidad de mejorar la eficiencia sorprendentemente. No voy a entrar en los detalles de utilización de la función *cupy.ElementwiseKernel*, para más detalles recomiendo la siguiente documentación.

Ahora procedemos finalmente con la propuesta para la resolución del problema original usando esta nueva idea de fusionar *kernels* (Código 2.5). Para ello lo que haremos será concentrar en un único *kernel* los cálculos necesarios para evaluar las funciones de reacción y los laplacianos. Nuevamente, no me dentendré a explicar los detalles de la implementación, pero espero que el código sea lo suficientemente claro como para motivar el interés del lector.

```
import cupy as cp

forces = cp.ElementwiseKernel(
  'raw float64 u, raw float64 v, float64 beta, float64 gamma, float64 Du, float64 Dv, uint32 Lx, uint32 Ly',
  'float64 Fu, float64 Fv',
  '''
  int x = i % Lx;
  int y = (int) i / Lx;
  Fu = -beta*u[i]*v[i] + Du*(u[(x+1)%Lx+Lx*y] + u[(x-1+Lx)%Lx+Lx*y] + u[x+Lx*((y+1)%Ly)] + u[x+Lx*((y-1+Ly)%Ly)]
```

```
        - 4*u[i]);
  Fv = beta*u[i]*v[i] - gamma*v[i] + Dv*(v[(x+1)%Lx + Lx*y] + v[(x-1+Lx)%Lx+Lx*y] + v[x + Lx*((y+1)%Ly)]
        + v[x + Lx*((y-1+Ly)%Ly)] - 4*v[i]);
  ''',
  'forces')

euler = cp.ElementwiseKernel(
  'float64 Fu, float64 Fv, float64 dt','float64 u, float64 v',
  '''
  u = u + dt*Fu;
  v = v + dt*Fv;
  ''',
  'euler')

def gpu_cupy_solver(u, v, Ds, beta = 1.,gamma = .2, dt = .01, d = 1, it = 1000):
  '''
  Solve a 2D reaction-diffusion equation for two dynamical variables with periodic boundary conditions using CuPy.

  u: Intial conditions for the first dynamical variable. 2d CuPy array of shape (Ly,Lx).
  v: Intial conditions for the second dynamical variable. 2d CuPy array of shape (Ly,Lx).
  Ds: Diffusion coefficients for the first and second dynamical variables. List or array of length 2.
  dt: Time step. Default value is 0.01.
  d: Space step. Default value is 1.
  it: Number of iterations. Default value is 1000.
  '''
  Ly,Lx = u.shape
  Du,Dv = Ds
  Fu = cp.zeros_like(u)
  Fv = cp.zeros_like(v)

  for _ in range(it):
    forces(u,v,beta,gamma,Du,Dv,Lx,Ly,Fu,Fv)
    euler(Fu,Fv,dt,u,v)
  return u,v
```

Código 2.5: Implementación con *CuPy*.

Este es el tipo de implementación utilizada en este proyecto de tesis, el resultado obtenido en esta ocasión es el siguiente,

```
  %timeit gpu_cupy_solver(u,v,Ds)
  511 ms +- 12.3 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

esto es, mejor en un factor 6 que la versión estándar de *CuPy*, 12 veces mejor que la implementación con *Numba* en paralelo (la mejor sin usar GPU), y 117 veces más rápido que la versión con *NumPy*. Ahora sí, vemos que la diferencia entre usar una GPU y no usarla es, por lo menos, en un factor de 12, la diferencia entre una simulación de un 1 día y una de 12 días.

Por último, quisiera terminar este capítulo con algunos comentarios. Por un lado, recordar que todas las comparaciones de velocidad en las diferentes implementaciones se realizaron con los mismos parámetros, y fundamentalmente con sistemas relativamente grandes, 1024×1024, donde la GPU se ve favorecida sobre la CPU. Diferente es el caso para sistemas chicos, por completitud en este sentido, en la figura 2.1 mostramos los resultados de velocidad de las distintas implementaciones para distintos tamaños de sistemas. Por otro lado, la mayoría de los resultados presentados en este proyecto fueron obtenidos con sistemas de $2^{16} \times 2^{11}$, en un sistema de dos ecuaciones de reacción-difusión como el discutido aquí, esto implica cuatro matrices con 2^{27} elementos cada una, considerando que además utilizamos un formato en coma flotante de doble precisión, cada matriz ocupa alrededor de $1GB$ de memoria. El tiempo que llevaría resolver sistemas de este tamaño sin GPU sería completamente inviable.

Finalmente, es posible que quienes estén más interiorizados con *CUDA* tal vez estén sorprendidos por el carácter superficial con el que usamos la GPU aquí. Es decir, cuando escribimos el *kernel* no lo escribimos en *CUDA C* o *CUDA C++*, es algo similar, pero es distinto, lo cual puede confundir un poco. Además, en ningún momento indicamos cantidad de hilos por bloque y cantidad de bloques en la grilla. Esto es nuevamente, porque *CuPy* intenta simplificar todo lo posible el uso de la GPU, para que sea accesible a usarios sin conocimientos de *CUDA*. Sin embargo, *CuPy* ofrece también todo el acceso al detalle, tal como podría hacerse con *CUDA C* por ejemplo, y admite la escritura

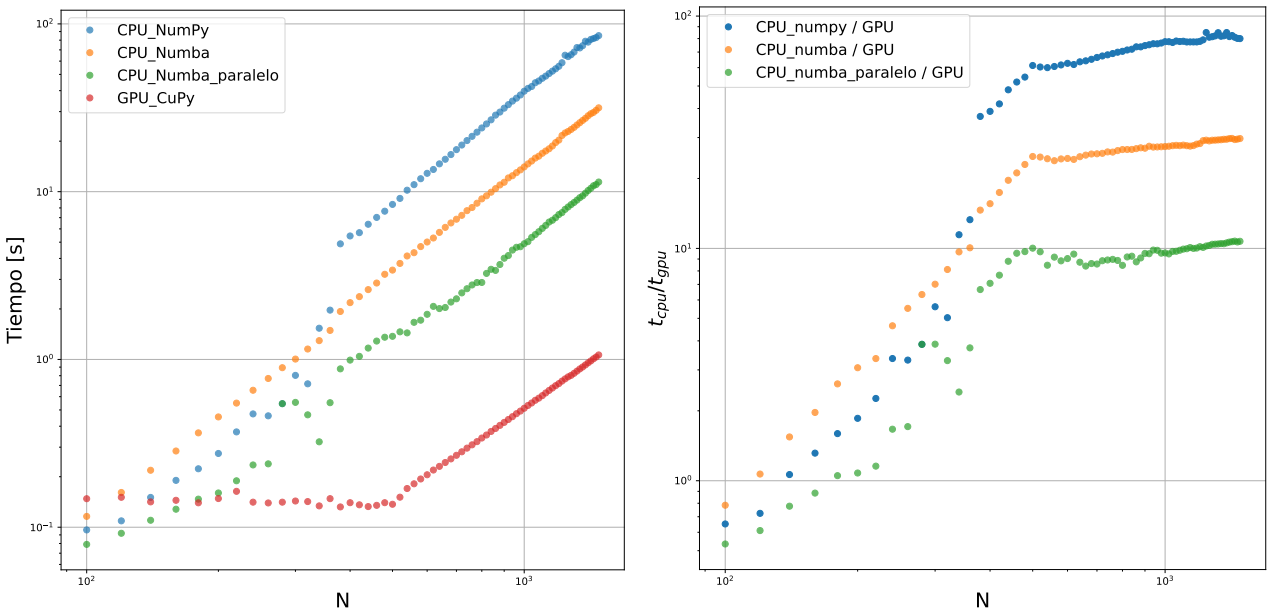

Figura 2.1: bla

de *kernels* directamente en *CUDA C* a través de la función *cupy.RawKernel*. En el material complementario de este capítulo, dejo la implementación usando esta función que da acceso completo a *CUDA*. No la agrego acá porque entiendo que no aporta demasiado, adicionalemente, no pude obtener ninguna mejora notable respecto al *kernel* escrito con *cupy.ElementwiseKernel*.

Apéndice A

Metodología numérica

El sistema de ecuaciones dado por

$$\frac{\partial S}{\partial t} = -\beta_{\mathbf{r}} SI + D_S \nabla^2 S,$$

$$\frac{\partial I}{\partial t} = \beta_{\mathbf{r}} SI - \gamma I + D_I \nabla^2 I,$$

se resolvió numéricamente utilizando el siguiente esquema explícito de Euler:

$$S_{ij}^{n+1} = S_{ij}^n + \Delta t \left(-\beta_{ij} S_{ij}^n I_{ij}^n + \frac{D_S}{d^2} (S_{i+1j}^n + S_{i-1j}^n + S_{ij+1}^n + S_{ij-1}^n - 4S_{ij}^n) \right)$$

$$I_{ij}^{n+1} = I_{ij}^n + \Delta t \left(\beta_{ij} S_{ij}^n I_{ij}^n - \gamma I_{ij}^n + \frac{D_I}{d^2} (I_{i+1j}^n + I_{i-1j}^n + I_{ij+1}^n + I_{ij-1}^n - 4I_{ij}^n) \right)$$

donde los índices i y j corresponden a la discretización de las coordenadas espaciales y el índice n corresponde a la discretización temporal. Se utilizaron de manera general los siguientes parámetros $\Delta t = 0.1$, $d = 1$, $D_I = 1$ y $D_S = 0$. Los demás parámetros involucrados se indican según corresponde en el texto.

Este esquema se implementó por computación en paralelo utilizando procesadores gráficos. De esta manera fue posible resolver de manera eficiente cientos de sistemas sobre grillas de 1024×1024. En particular se utilizó la librería de `Python` para computación en paralelo llamada `CuPy`, que es un equivalente de la librería `NumPy` pero integrada con `CUDA`.

A continuación se muestra la función central del esquema numérico que permite recorrer la grilla del sistema en paralelo y calcular las derivadas del sistema en cada nodo utilizando esta librería, se representan a S e I como X e Y respectivamente:

```
forces = cp.ElementwiseKernel(
'raw float64 X, raw float64 Y, raw float64 params,raw float64 beta,raw float64 gamma, int16 L' ,
'float64 fX,float64 fY',
'''
double N = params[0]; double nu = params[1]; double mu = params[2]; double Dx = params[3];
double Dy = params[4];

int x = i % L;
int y = (int) i/L;

fX =  nu - beta[i]*X[i]*Y[i]/N - mu*X[i] +
    Dx*(X[(x+1)%L + L*y] + X[(x-1+L)%L+L*y] + X[x + L*((y+1)%L)] + X[x + L*((y-1+L)%L)] - 4*X[i]);

fY = beta[i]*X[i]*Y[i]/N - (gamma[i]+mu)*Y[i] +
    Dy*(Y[(x+1)%L + L*y] + Y[(x-1+L)%L+L*y] + Y[x + L*((y+1)%L)] + Y[x + L*((y-1+L)%L)] - 4*Y[i]);
''',
'forces')
```

y el paso de Euler se lee simplemente como:

```
...
forces(X,Y,params,beta,gamma,Lx,fX,fY)
X = X + tstep*fX
Y = Y + tstep*fY
...
```

Para dar una idea de la aceleración dada por la programación en paralelo, en la figura A.1 se muestra el tiempo necesario para resolver un sistema de $N \times N$ sitios y 1000 pasos de Euler con procesadores gráficos (GPU) y con procesadores convencionales (CPU). La diferencia es notable, por ejemplo, para un sistema de 1024×1024, el tiempo necesario para resolverlo con CPU es aproximadamente de 2.7 horas, mientras que con GPU se resuelve en 1 segundo. Se muestra también un ajuste para ambas curvas del tipo $t \propto N^a$ con $a \approx 2$ para ambos. De todo esto resulta clara la necesidad de trabajar con computación en paralelo para obtener los resultados desarrollados en este trabajo en un tiempo razonable. Para ver más detalles acerca del código y las herramientas computacionales implementadas puede acceder al siguiente *Notebook* de *Google Colab*.

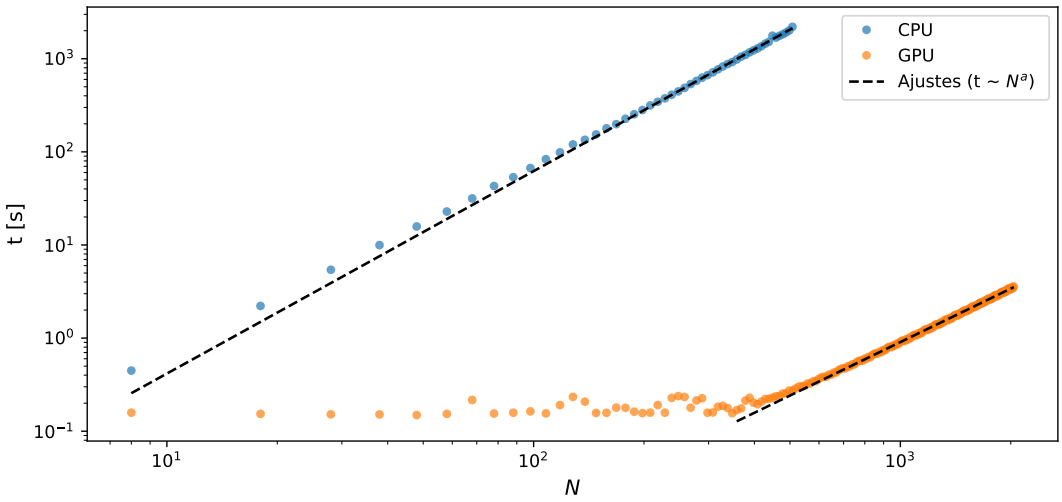

Figura A.1: Tiempo de resolución de un sistema de $N \times N$ sitios y 1000 pasos de Euler con procesadores gráficos (GPU) y con procesadores convencionales (CPU). Se muestran también los ajustes de tipo $t \propto N^a$ con $a \approx 2$ para ambos.

Bibliografía

Agradecimientos

Agradezco al Instituto Balseiro, a la Universidad Nacional de Cuyo y a la Comisión Nacional de Energía Atómica por hacer posible mis estudios universitarios. Agradezco enteramente a todo el personal de estas instituciones que hacen posible, con gran esfuerzo y a pesar todas las complicaciones dadas por la pandemia, que educación universitaria de altísimo nivel, pública y gratuita llegue a todo el país. Particular agradecimiento para mi director de tesis, Dr. Alejandro Kolton, por su gran apoyo en el desarrollo de este trabajo y buena predisposición para todo. Agradezco también a mis compañeros de camada Amir Zablotsky, Pedro *El Bata* Llauradó, Esteban *El Leches* Acerbo, Marco Madile, Ezequiel Saidman, Martín Famá y Francisco *El Bbto* Gymnich por estar siempre presentes y ayudar a liberar las tensiones de la vida universitaria.