Escuela Técnica Superior de Ingeniería Informática

Trabajo Fin de Grado

Grado en I.I. en Tecnologías Informáticas

Simulación Acelerada por GPU de Máquinas de Virus

Realizado por

Sergio Velázquez García

Dirigido por

D. Miguel Ángel Martínez del Amor

D. David Orellana Martín

5 de julio de 2023

Índice

Resumen

Las GPU reinan como dispositivos de programación heterogénea paralela y la computación de propósito general, y dentro de estos campos, CUDA se mantiene como principal paradigma para la programación en GPU. Se plasmará los paradigmas de CUDA y los pros y contras de GPUs frente a problemas computacionales y, partiendo de un simulador de un sistema bio-inspirado como es la máquina de virus, se construirá unos simuladores ad-hoc paralelos de distintos ejemplos del sistema anterior, como probabilísticos o multi-instrucción. Seguidamente, evaluaremos Numba como plataforma para conseguir todo lo anterior, calculando tiempos promedios de ejecución tanto en CPU como en GPU.

1 Introducción

Hoy en día, toda CPU es acompañada de un coprocesador, este microprocesador es utilizado como soporte de la CPU como, por ejemplo, el procesamiento gráfico. Dicho coprocesador es llamado **procesador gráfico** o **GPU** (del inglés *graphics processing unit*) y su función principal, valga la redundancia, es trabajar con gran parte de las funciones gráficas. La razón de esto radica en la arquitectura común entre las GPUs, que veremos más adelante, además de qué se diferencian cada unidad de procesamiento.

Pero la verdad es, las GPUs pueden ser utilizadas para más que procesamiento de gráficos y este trabajo va a pivotar alrededor de esta idea [DKK21]. Aprovecharemos la potencia que nos ofrece su arquitectura para acelerar un simulador de un sistema computacional llamado **Máquina de Virus** [VCPJC+15] basado en replicar el comportamiento de virus sobre huéspedes. Dichos virus estarán contenidos en huéspedes o *hosts* y serán capaces de ser propagados entre sí mediante canales de transmisión. En sí, el sistema se compone de un grafo bipartito dividido en *hosts* (huéspedes) e *instrucciones* y las conexiones entre ambos. Las instrucciones apuntan a canales que unen los *hosts* y, al ser activadas, abren dichos canales para dejar paso a los virus que tengan los huéspedes. Las máquinas de virus son una línea nueva que cuenta con algunos trabajos ya establecidos como generadores de conjuntos diofánticos [RJVCPJ15], computación de *pairing functions* [RdAOMPJ23] y computación de funciones recursivas parciales [RJVCRNPJ15] en la que se menciona por primera vez el posible futuro de realizar un modelo de máquina de virus paralelo. Se trabaja con el simulador secuencial[1] [OMRdAPJ23] programado en *Python* el cual refleja varios modelos de máquinas de virus — tales como operaciones aritméticas — para trabajar sobre varios simuladores específicos (ad-hoc) que aporten una aceleración con respecto al original [MdAOMPH+19].

La preparación de los simuladores paralelos es construido sobre la base de dicho simulador secuencial genérico de estos modelos bio-inspirados. Al ser el primer simulador paralelo de máquina de virus, se busca marcar un precedente en su desarrollo y estipular las técnicas para acelerar dicho sistema. Se busca además implementar este simulador en Python con el compilador Numba como forma de pro-

[1] Enlace al repositorio: https://github.com/RGNC/virusmachines

gramar con los paradigmas de CUDA sin escribir código C/C++. Se opta de esta forma sobretodo para mantener una cierta facilidad de mantenimiento de código al mantener todo en un mismo lenguaje de programación, además de evitar complicaciones extras como pueden ser las uniones entre Python y C.

1.1. Marco histórico: GPU

Sabiendo que el mayor banco de trabajo de las GPUs son los gráficos, no es una sorpresa que los videojuegos hayan supuesto una necesidad en la evolución de las mismas. En la década de los setenta, la **RAM** (*random-access memory* en inglés) iba siendo cada vez más solicitada para las primeras arcades, así que se empezó a utilizar procesadores gráficos para suplir esta necesidad. [Hag15]

Si avanzamos a la década de los ochenta, encontraremos las primeras GPUs implementadas en computadoras, tales como la *NEC µPD7220*. En esta década ya encontramos soporte para resoluciones de hasta 1024x1024 y computación poligonal 3D.

Y, aunque hayamos estando hablando todo este momento de GPUs, no fue hasta Octubre de 1999 cuando *NVIDIA* introdujo el mismo término — anteriormente se llamaban símplemente tarjetas de vídeo — con la famosa *NVIDIA GeForce256* que compitió con la *Radeon 7500* de *ATI*. [McC10]

Saltando al nuevo milenio, el siguiente paso a la evolución de las GPUs fue la introducción de una *pipeline* programable. En 2001, *NVIDIA* lanzó *Geforce 3*, la nueva generación de GPUs de la desarrolladora, su arquitectura fue capaz de implementar sombreadores de vértices y píxeles programables mediante estas *pipelines*. En 2003, empezaron a aparecer las primeras GPUs con compatibilidad en computación en coma flotante en 32 bits, lo cual empezó a acercar la computación de propósito general al mundo. [Mac03]

El crecimiento en la tecnología de las GPUs en los 2000s fue encomendable, incluso superando teniendo un ratio aún más rápido que la **Ley de Moore** [McC10]. La sexta generación de GPUs de NVIDIA contaba ya con 222 millones de transistores — *GeForce 6800 Ultra* —. [Wik]. De forma paralela, dentro del mundo de la computación empezaron a ver las GPUs como algo más que un procesador de gráficos y texturas y más como una gran herramienta para la *GPGPU* — siglas de *general-purpose computing on graphics processing units* —. Y de ahí, en el año

2006, llega el nuevo modelo de programación **CUDA** por mano de NVIDIA. En nuestro contexto, la llegada de CUDA es bastante importante. A partir de 2006 NVIDIA y en general toda la escena de procesadores gráficos darán un peso extra al GPGPU, haciéndolo más accesible y completo.

Por supuesto, la década de los 2010 estará repleta de avances. Aún siendo NVIDIA y AMD los reyes de la industria en cuanto innovación se trata. A lo largo de la década NVIDIA lanzará las arquitecturas Fermi, Kepler, Maxwell, Pascal y Turing (2010, 2012, 2014, 2016 y 2018 respectivamente).

En esta década el número de transistores osciló entre 585 millones, en los casos más básicos, y 18.600 millones en la gama alta (este último siendo la GeForce RTX Titan de 2018). Es evidente el **crecimiento tecnológico** en solo unos años el cual no se limita a transistores sólo; memoria, procesadores de *streaming*, velocidad del reloj también verán un crecimiento conjunto con cada arquitectura lanzada.

Para explicar en más detalle los logros tecnológicos conseguidos en esta década, podríamos avanzar al año 2017, cuando se introduce por primera vez los *Tensor cores* en la arquitectura Volta de NVIDIA. Estos núcleos se implementan dentro de los multiprocesadores de la propia GPU, en el caso de la arquitectura Volta, cada bloque componía dos *Tensor cores* [MDCL+18]. En particular este avance nos interesa mucho, pues supondría un **avance en el rendimiento de la computación para IA y HPC**, esto es porque dichos núcleos están preparados para hacer operaciones y procesar datos agrupados en tensores — tales como redes neuronales —

En 2018, con la arquitectura Turing, llegó la serie RTX 20 de NVIDIA con un cambio evidente en el nombrado de las famosas GTX de la marca. Este cambio de nombre se debe a la tecnología de **trazado de rayos** y sus núcleos RT que presenta esta serie, una técnica de *renderizado* que simula como se comporta la luz en un entorno 3D. Aunque en general, la serie RTX sigue estando orientada a incrementar la velocidad de accesos aleatorios en memoria y distribución en GPUs. [SGFV19]

Al año siguiente, se lanzó una versión más básica de las famosas RTX, con el nombrado tradicional de NVIDIA: La serie GTX 16 se basaba también en la **arquitectura Turing** como sus hermanas mayores pero de menor costo. Prescinden de los núcleos RT y, por ende, del trazado de rayos.

Desde el punto de vista de NVIDIA, dos arquitecturas gobiernan los escasos tres años que alcanza la década actual: Ampere y Ada Lovelace (2020 y 2022 respectivamente). Ambas arquitecturas se concentran en el desarrollo del trazado de rayos y los *Tensor cores*, incrementando su rendimiento en ambos campos. En cuestión de transistores, entre las dos generaciones rondan **entre los 13.300 millones y los 76.000 millones**.

Aunque tenemos un presente lleno de grandes avances y que parecen que van a más y más, en el futuro no podremos esperar a ver un crecimiento constante. Existe un modelo tecno-económico llamado La **Ley de Moore** que establece que la cantidad de transistores que se pueden colocar en un microprocesador se duplica cada dos años, esta cantidad es alcanzada debido a que los transistores son cada vez más pequeños. Sin embargo, cuanto los transistores más se acercan a una escala atómica, su coste se dispara, vaticinando el fin de la Ley de Moore para 2025, después de 50 años de que se postulara. [Sha20]

Para el futuro, se pone la vista a la tecnología de semiconductores, pero para que sea una alternativa viable — tanto económica, tecnológica y energética —, la transición podría durar décadas [Sha20]. Pero como todo, el mañana lo dirá.

1.2. El Estado del Arte: Computación en GPU

En el anterior apartado se mencionó brevemente el término GPGPU, relacionándolo con el nacimiento de CUDA en 2006. GPGPU o computación de propósito general en unidades de procesamiento gráfico, es el término que se utiliza para designar las tareas que normalmente se asignarían a ser procesadas por la CPU pero, en vez de eso, **se ejecutan en GPU**. Todo con el propósito de llegar a unos tiempos más eficientes.

Por ello, la primera tarea es ver si el problema en cuestión puede aprovechar todo el potencial de la GPU, porque de no ser así, el tiempo de ejecución **no variará mucho de una CPU** — o en algunos casos, incluso será peor—.

Con todas las cualidades que puede venir de usar GPGPU, también vienen inconvenientes. En su inicio, el resultado de paralelizar venía con el coste de no manejar operandos de enteros, desplazamientos de bits o aritmética de precisión doble [GPKB12]. Al final, cada vez más programadores buscan hacer su código más orientado al GPGPU, lanzándose a explorar cómo funciona el paralelismo de datos

en el entorno tan especial como es la GPU.

Esto se realiza acelerando solo las partes del código que son paralelizables. Por norma general, no todo el código se acelera con la GPU, si no que se le asocia solo la **carga de trabajo más intensiva**. [NVI]

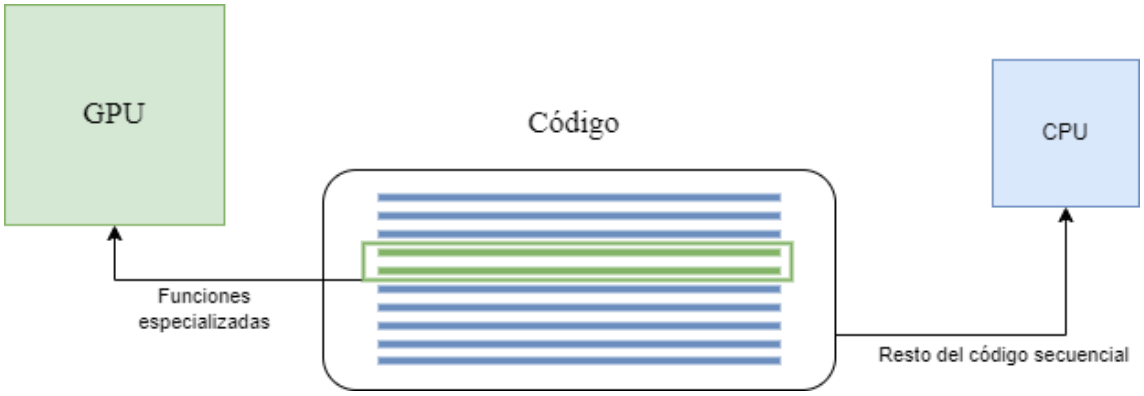

Figura 1: Cómo funciona la aceleración en GPU

A lo largo de los años, han surgido múltiples SDKs y APIs que han servido como llave al GPGPU. NVIDIA CUDA, ATI Stream SDK, OpenCL, SYCL y PGI Accelerator no son más que algunos ejemplos de algunos. NVIDIA CUDA reina como el método más utilizado y demandado por su grandes aplicaciones en machine learning [Pat23] así como en GPGPU [GPKB12].

1.3. Objetivos

Contaremos como objetivos para este trabajo la implementación de buenas prácticas de CUDA sobre el primer simulador paralelo de máquina de virus con el fin de decrementar su tiempo de ejecución. Más concretamente, estos son los preceptos que se quieren conseguir:

- Análisis del funcionamiento de la máquina de virus estándar, además de su modelo probabilístico y *softparallel*.
- Llegar a una mayor comprensión en GPUs y GPGPU.
- Implementar varios simulador paralelo ad-hoc de máquinas de virus
- Brindar un tiempo de ejecución menor con el uso de GPU
- Evaluar la plataforma Python Numba como alternativa a CUDA-C

1.4. Estructura del trabajo

La estructura que se estudiará en este trabajo sigue el siguiente esquema:

- **Computación paralela y CUDA**. Estudio del funcionamiento básico de una GPU y cómo se relaciona con el modelo de programación CUDA además de su arquitectura general. Se introduce el compilador JIT Numba como herramienta para escribir en CUDA y sus diferencias con C++. Finalmente, se introduce brevemente el compilador JIT.

- **Definición y estudio de la máquina de virus**. Se darán definiciones formales de tres tipos de máquinas de virus: estándar, probabilística y softparallel junto a la representación de un modelo para cada una.

- **El simulador secuencial**. Se estudiará y expondrá el pseudocódigo y funcionamiento de las tres versiones del simulador secuencial para cada máquina de virus antes explicada.

- **El simulador paralelo**. Siguiendo todo lo aprendido, se plasma la implementación de múltiples simulaciones en paralelo con Numba CUDA del modelo basicprobabilistic, además de soluciones en Python puro y JIT. Seguidamente, también se expone una implementación de un simulador paralelo para el modelo spsuma.

- **Resultados de las simulaciones**. Se plasma todos los resultados de los simuladores descritos anteriormente además de unas comparaciones de tiempos entre CPU y GPU. Se analizará dichas comparaciones explicando sus comportamientos y curiosidades.

- **Conclusiones**. Se dará broche final al trabajo, dando un resumen de todo lo logrado y trabajo además de algunas reflexiones sobre CUDA-Numba y el futuro del simulador paralelo de las máquinas de virus.

2 Computación paralela: CUDA y tú

Ya sea un simple programador queriendo acelerar su aplicación, un ingeniero de una multinacional buscando cómo entrenar su IA más eficientemente o un estudiante queriendo explorar el mundo más allá de la programación secuencial, NVIDIA-CUDA sirve como llave al uso del GPGPU y la computación paralela o acelerada. En este apartado explicaremos su modelo de programación, arquitectura y peculiaridades que le hacen una herramienta de paralelismo muy potente. Para evitar confusiones con la guía oficial de CUDA [NVF23a], todos los términos reminiscentes a su modelo o arquitectura **permanecerán en inglés**.

2.1. Entender el medio: GPU

Una característica fundamental que hace las GPUs únicas entre todos los coprocesadores — y procesadores principales — es su gran capacidad de **paralelización**. Pero, ¿por qué nos interesa esto? ¿Y cómo lo consigue?

1. Ya sea para mostrar gráficos o para **trabajar con una gran cantidad de datos**, crear procesos paralelos mejora la eficiencia del problema conjunto.

2. Una GPU contiene cientos o miles de **pequeños procesadores**, cada uno de esos procesadores puede crear un *thread* de procesamiento, con lo que su potencial para procesar algoritmos paralelos es muy superior al de una CPU.

Concentrémonos por ahora en el segundo punto.
Cada una de las unidades de procesos — a veces llamadas *Unified Shaders* o *Stream Processors* — es capaz de ejecutar operaciones muy sencillas, por lo que son más baratas que las que componen una CPU. Sin embargo, y como se ha dicho anteriormente, de lo que carece en potencia de procesamiento, lo compensa agrupándose en grandes números. Estos núcleos, a su vez, se agrupan en multiprocesadores los cuales se ubican en la misma GPU, de modo que un multiprocesador de una GPU puede albergar k núcleos (ver figura [2]).

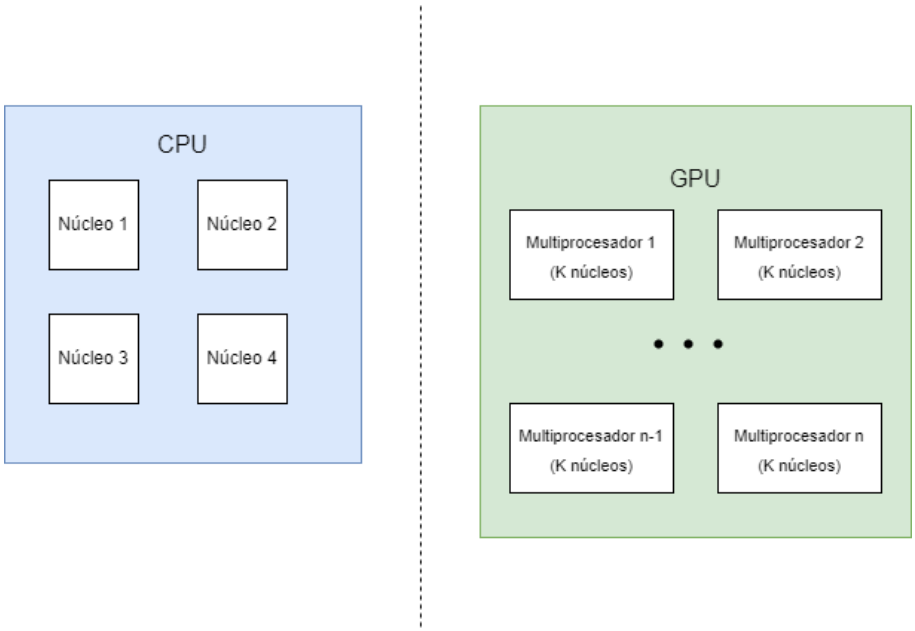

Figura 2: Comparación entre CPU y GPU

Además de todo esto, toda GPU cuenta con una RAM dedicada y puertos de entrada y salida. Como veremos más adelante, es importante almacenar toda la información que podamos en la misma GPU, ya que queremos coger los datos que nos brinde la CPU para computarlos y luego devolvérselos.
Para comprender mejor la capacidad de la GPU de procesar datos paralelamente frente a CPU es necesario introducir los siguientes términos:

- **Latencia**: Duración de una operación desde su «nacimiento» hasta su «muerte». Expresado en microsegundos.
- **Tasa de transferencia**: Número de operaciones procesadas por unidad de tiempo. Expresada en *gigaflops*.
- **Ancho de banda**: La cantidad de datos procesados por unidad de tiempo. Expresado en *megabytes* por segundo o *gigabytes* por segundo.

Como hemos visto en la figura [2], nuestra unidad de procesamiento gráfico está compuesta de múltiples multiprocesadores, este diseño bebe del deseo de **maximizar la tasa de transferencia** mediante el manejo de un gran número de operaciones simultáneas. Esto último se verá en más detalle en la arquitectura SIMT de CUDA.

2.2. Modelo de Programación

CUDA es una plataforma de programación heterogénea paralela la cual parte de una abstracción que ayuda al programador a desgranar el problema principal en distintos **sub-programas** los cuales se pueden resolver independientemente por paralelo mediante hilos de ejecución de instrucciones que se ejecutan simultáneamente en paralelo con respecto a otros hilos o *threads*.
En sí, CUDA utiliza directivas de compilador, APIs (interfaz de programación de aplicaciones), extensión de lenguajes de programación y librerías para brindar un acceso a la GPU programable.
Nuestro modelo de programación cuenta con toda una arquitectura y funcionamiento propio que se deberá estudiar antes de pensar en analizar o bordar el problema.

2.2.1. Funciones en CUDA

CUDA ofrece al programador unas funciones llamadas *kernels* que se ejecutan asíncronamente por defecto, los lanzamientos ponen en cola su ejecución en el dispositivo y luego vuelven inmediatamente. **No devuelven un valor**, todo resultado tiene que estar escrito en un array que recibe el kernel como parámetro. Cuando se ejecuta, la función es ejecutada a la vez en cada *thread* — dependiendo en jerarquía de *threads* definida anteriormente —. Cada *thread* que ejecuta un *kernel* tiene un único ID que es accesible dentro del mismo *kernel* mediante **variables integradas**.
En paralelo con los *kernels*, tenemos las *device functions* las cuales se definen y llaman en el código CUDA y se ejecutan en la GPU, en contraposición a las funciones *hosts* que se definen y llaman en la CPU. Se utilizan para aprovechar los datos que ya se encuentran en GPU, sin necesidad de moverse entre *host* y *device*.

2.2.2. Jerarquía de Threads

Por si no se ha notado hasta ahora — y se promete que se seguirá nombrando — los *threads* son una parte fundamental en la GPGPU. Particularmente en CUDA, tenemos una variable integrada en forma de un vector de tres componentes llamado **threadIdx**. Esta variable proporciona una forma muy natural de convocar elementos como vectores, matrices o volúmenes. Teniendo esto en mente, ¿cómo

funciona?

Nuestra variable integrada proporciona a cada *thread* con un índice propio, el cual puede ser usado para realizar una operación cualquiera. La tri-dimensión se requirirá para problemas de la misma envergadura, es decir, para operaciones con matrices de dos dimensiones se usaría la componente x y la componente y de **threadIdx**. El valor en cada *thread* es único para cada *thread* y *thread block*, esto permite que se pueda operar en un determinado elemento basado en una posición única dentro del bloque.

Claro está, hay un número máximo de *threads* por bloque. La teoría detrás de esto reside en que cada *thread* tiene que residir en el mismo núcleo SM y compartir la misma memoria del mismo núcleo. Hoy en día, una GPU podría llegar a contener hasta 1024 *threads* por bloque.

Sin embargo, un *kernel* **puede ser ejecutado por múltiples *thread blocks*** — siempre con mismas dimensiones —, así que el número total de *threads* es igual a:

$$\Big\{\text{NTB} \times \text{NB}\Big\}$$

Siendo:

- NTB: Número de *threads* por bloque
- NB: Número de bloques

Los bloques pueden estar formados por una, dos o tres dimensiones de *grids*. El número de bloques en un *grid* es normalmente dado por el tamaño de los datos que se quieren procesar. Mediante la variable integrada **blockIdx**, un kernel puede acceder al índice de un bloque dentro de un *grid*. Como el nombrado sugiere, tiene el mismo tratamiento tri-dimensional que tiene **threadIdx**.

Finalmente tenemos todo el esquema de la jerarquía de *threads* (ver Figura 4). Una caja grande (*grid*) que contiene n cajas más pequeñas (*thread block*) que a su vez contienen k objetos por cada caja anterior (*thread*).

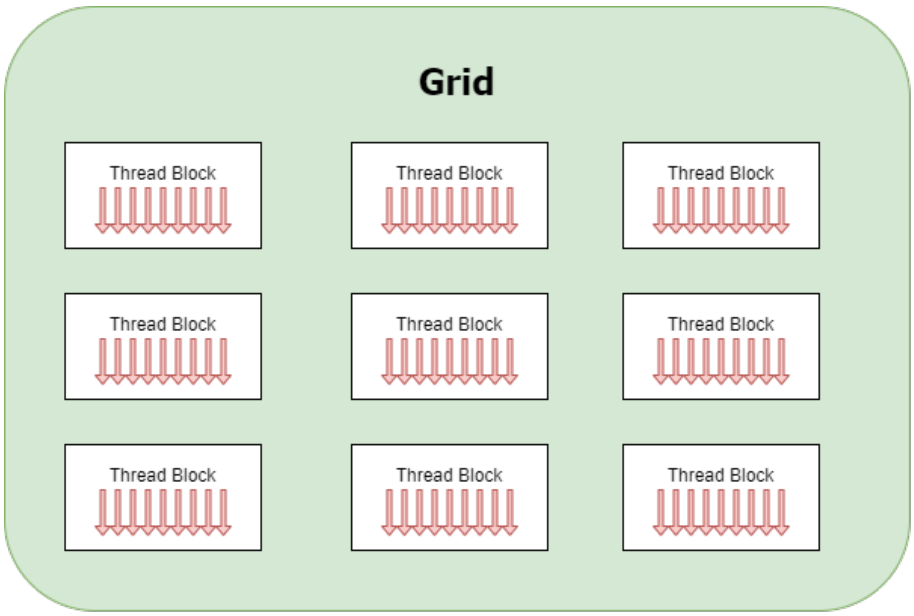

Figura 3: Representación gráfica de la jerarquía de *threads* (creación propia)

Con la entrada en escena de la **Capacidad Computacional 9.0** de NVIDIA, CUDA introduce un nivel más de jerarquía llamada *thread block clusters* que son, básicamente, bloques de bloques. Así, de igual manera que un bloque agrupaba a múltiples *threads* o un *grid* a varios bloques, en este nivel de jerarquía son los *clusters* quienes contienen varios bloques — siendo ahora el *grid* quien contiene ahora los *clusters* —.

2.2.3. Jerarquía de Memoria

Cuando la GPU procesa un conjunto de datos, lo hace en su propia memoria, es decir; GPU y CPU tienen distintas memorias y por lo general, este conjunto de datos tiene que ir «bailando» de memoria en memoria. Para que la GPU tenga los datos a computar, estos datos tienen que ser movidos de CPU a GPU y cuando el cálculo ha terminado, los datos tendrán que volver al *host*.

Para el programador, la GPU ofrece distintos niveles de memoria, cada uno con sus distintivas diferencias en cuanto lectura y escritura se refiere. Cada uno de estos niveles se puede relacionar con la pasada jerarquía de *threads* debido a su intrínseca relación. (ver Tabla [1])

Tenemos pues, nuestra memoria global — y caché — que se utiliza como puerto entre el *device* y el *host*, esta memoria se comunica directamente con cada *thread*. A su vez, cada *thread* tiene sus propios registros y memoria local y una agrupación de *threads* o *thread block* que tiene su propia memoria compartida. Mientras que

	Alcance	Acceso	Velocidad
Memoria local	Thread	RW	Lenta
Registros	Thread	RW	Rápida
Memoria Caché	Compartida por todos los SM	RW	Rápida*
Memoria compartida	Thread Block	RW	Rápida
Memoria constante	Threads y CPU	R	Lenta
Memoria de textura	Threads	R	Lenta
Memoria global	Threads y CPU	RW	Lenta

*L1 es más rápida que L2, pero la última es compartida por todos los SM

Tabla 1: Tabla de jerarquía de memorias en CUDA

la memoria global hace de suelo para todos los *kernels* albergando todo *input* o *output* necesario para su ejecución, todo lo demás encapsula cada multiprocesador. (ver Figura [4])

2.2.4. Programación Heterogénea

Durante todo este tiempo hemos estado nombrando a la práctica de utilizar exclusivamente la potencia del hardware de la GPU para hacer un cálculo especializado. Esta práctica de utilizar diferentes arquitecturas a la hora de procesar datos se denomina **programación heterogénea**. Pero, ¿qué significa realmente y qué diferencia tiene con su antónima?

Primeramente, la **programación homogénea** es el tipo de programación que conocemos habitualmente, tenemos nuestro procesador — o procesadores — los cuales ejecutan instrucciones una detrás de otra. Sabiendo esto, cabe imaginar que su contrapartida heterogénea es la responsable de la paralelización.

La programación heterogénea es el uso del hardware especializado para el realizado de una tarea. Ejemplos de esto incluyen aceleradores criptográficos, matrices de puertas lógicas programable en campo (FPGA en inglés), procesadores de señal digital (DSPs en inglés), unidades de procesamiento tensorial y, claro está, unidades de procesamiento gráfico (GPU).

2.3. Arquitectura SIMT

La arquitectura SIMT — del inglés *Single Instruction Multiple Thread*— se encuentra al uso en las GPUs modernas de principales fabricantes, como NVIDIA o AMD. Esta arquitectura se basa en el concepto de dividir una tarea en múltiples *thread*s o *threads* y procesarlos de manera simultánea.

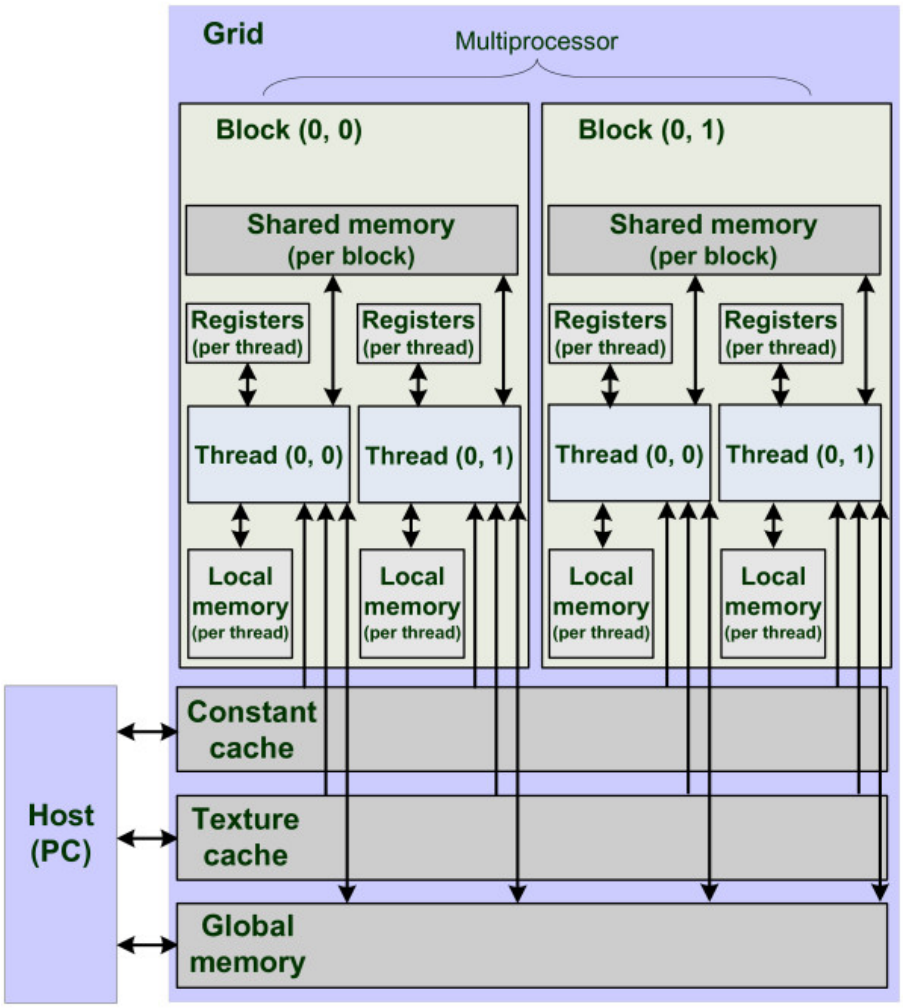

Figura 4: Representación gráfica de la jerarquía de memoria [HKAA11]

Los multiprocesadores crean, gestionan y ejecutan *threads* en grupo de 32 *threads* paralelos llamados, en la arquitectura CUDA, *warps* —así mismo, AMD los llama *wavefronts*—. Cada *thread* individual que compone un *warp* comienza en la misma dirección del programa, pero cada uno tiene su propia dirección de instrucciones y estado de registro y por lo tanto son libres de ramificarse y ejecutarse de forma independiente.

De esta manera, tenemos un conjunto de *thread*s que se agrupa en bloques — o *warps* como hemos visto —, los cuales se asignan a un multiprocesador en la GPU. Cada uno de esos multiprocesadores tiene varios **núcleos de procesamiento** o SM (*Streaming Multiprocessor*) que a su vez pueden procesar múltiples *thread*s simultáneamente. Cada SM tiene su propia memoria compartida, que se utiliza para compartir datos entre los *thread*s que se ejecutan en el SM.

Cuando un bloque de *thread*s se asigna a un SM, cada *thread* procesa una porción del trabajo total en paralelo con los demás *thread*s del mismo bloque. En la arquitectura SIMT, todos los *thread*s del bloque ejecutan la misma instrucción al mismo tiempo, lo que significa que la unidad de control del SM solo necesita decodificar y ejecutar una única instrucción por ciclo de reloj, mejorando la eficiencia de procesamiento.

Además, la arquitectura SIMT utiliza la técnica de **ejecución especulativa**, que permite que los *thread*s continúen ejecutando instrucciones aunque se encuentren en espera de que se completen otras instrucciones. Esto ayuda a reducir los tiempos de espera y mejorar la utilización de los recursos de procesamiento.

2.4. Multithreading

En CUDA, el *multithreading* es una técnica usada para mejorar el rendimiento de las aplicaciones mediante una división de la misma ejecución entre múltiples *threads* (o *thread*s) que se ejecutan al mismo tiempo.

Más detenidamente, en el marco contextual de la ejecución, por cada *warp* procesado por un SM es mantenido *on-chip* durante toda la vida del *warp*. Esto desemboca a un coste nulo a la hora de cambiar entre una ejecución y otra, usando un *warp scheduler* para seleccionar el *warp* que tiene *threads* listos para ejecutar la siguiente instrucción.

El número de bloques y *warps* que pueden residir y ser procesados en un SM dado

un *kernel* depende de la cantidad de registros y memoria compartida usada por el mismo *kernel* y del número de registros y memoria compartida que hay disponible en el SM. Naturalmente, existe un número máximo de bloques o *warps* que están siendo ejecutados simultáneamente — también llamados bloques o *warps* residentes —. Este límite viene dado en función de la **Capacidad Computacional** de la GPU. Si no hay suficientes registros o memoria compartida disponible por SM para procesar al menos un bloque, el kernel no se ejecutará.
El número total de *warps* en un bloque se define por:

$$\left\lceil \frac{T}{Wsize} \right\rceil$$

Tener en cuenta:

- T representa el número de *thread*s por bloque
- *Wsize* es el tamaño del *warp*, el cual equivale a 32.
- $\lceil \frac{x}{y} \rceil$ simboliza el techo de la fracción, es decir equivale al resultado de $\frac{x}{y}$ redondeado hacia arriba.

2.5. Pautas de Rendimiento en CUDA

Nuestro objetivo final es un buen rendimiento del sistema que queremos apoyar. Para ello, CUDA ofrece una serie de pautas — algunas de prioridad alta o baja — para lograr, en el mayor de los casos, un tiempo de ejecución menor. A continuación, se definirán todas las prácticas escogidas para programar sobre el código del simulador. La propia guía de programación insta cuatro prácticas fundamentales: [NVF23b]

1. **Maximizar la ejecución paralela**. Una aplicación que logra esto debería estar estructurada de forma que se exponga toda la paralelización que se pueda para mantener la mayor parte de la GPU ocupada el mayor tiempo posible. Para ello, se necesita lanzar un mínimo de orden de miles de hilos para mantener la GPU ocupada.

2. **Optimizar el uso de memoria para alcanzar máximo rendimiento en memoria.** Se insta minimizar el flujo de datos con poco ancho de banda,

como por ejemplo aquellos entre CPU y GPU. También se recomienda minimizar la transmisión de datos entre memoria global y GPU mediante el uso de memoria *on-chip* — caché y memoria compartida—.

3. **Optimizar el uso de instrucciones para alcanzar máximo rendimiento en instrucciones.** Minimizar el uso de instrucciones aritméticas de poco rendimiento, minimizar la divergencia entre instrucciones y *warps*, reducir el número de instrucciones mediante un uso inteligente de la sincronización de hilos y maximizar la ejecución de instrucciones sobre un dato ya cargado en memoria — aunque se podría considerar que esta práctica baila entre el anterior punto y este— son ejemplos propios de este punto.

4. **Minimizar la hiperpaginación de memoria.** Asignar y liberar memoria de forma constante puede acabar empeorando el rendimiento de nuestro código.

2.6. CUDA en Python: Numba

Originalmente, NVIDIA introdujo CUDA como su plataforma de computación paralela de propósito general con un entorno de *software* que permitía a los desarrolladores usar C/C++ como lenguaje de programación. Hay alternativas a dicho lenguaje, como por ejemplo Java o Python, los cuales son implementados mediante *wrappers*. Nosotros nos enfocaremos en Python para el desarrollo de este trabajo y, para ello, Numba está basado directamente en como CUDA se implementa en *C++*.

Numba intenta optimizar el código combinando el uso de *decorators* e información intrínseca de las variables de entrada de una función. Siguiendo el modelo de programación de CUDA, Numba compila el código dado usando un compilador LLVM (*Low Level Virtual Machine*) para generar un código máquina que pueda ser familiar para la GPU. Por ejemplo, los *kernels* tienen acceso directo a *arrays* de *NumPy* los cuales son transferidos de CPU a GPU automáticamente — y viceversa —. Así pues, todo código escrito en Numba-CUDA es ejecutado en paralelo por defecto. [Ana]

Tipos de datos como arrays de NumPy, *integers* o *floats* serán nuestra predilección. Esto no sorprende a estas alturas, como hemos visto antes, la GPU trabaja

muy bien con datos de bajo nivel así que Numba ofrece un soporte a NumPy aún con una limitación en algunas de sus APIs.
Pero en todas sus ventajas, también tiene sus limitaciones: Al ser una herramienta de compilado *Just In Time* (JIT), Numba no puede operar con distintos tipos de datos. Seguidamente, como nuestro código se va a ejecutar en GPU, datos como clases, objetos u otros tipos de alto nivel no se pueden compilar. Y para terminar, Numba no implementa todas las características de CUDA: Paralelismo dinámico y memoria de texturas no se encuentran implementadas al día de hoy [Ana].

2.6.1. CUDA-C vs CUDA-Numba

Numba ofrece una abstracción del código «original» de CUDA para Python, sin embargo, ¿tiene algún inconveniente? En un primer lugar, podemos echar un vistazo a cómo se estructuran la sintaxis en ambos, ver como Python sigue brillando por su simpleza y legibilidad: La configuración de número de bloques e hilos se agrupa entre corchetes, los atributos de la función no tienen que estar propiamente definidos en ella — al igual que en Python clásico — y con un simple *decorator*, el intérprete ya configura la función englobada como un kernel de CUDA.
Dentro del mismo kernel la estructura es muy parecida entre CUDA-C y CUDA-Numba. Esto es así dado que Numba trata de emular lo máximo posible las prácticas de CUDA — las únicas diferencias que podríamos encontrar son aquellas entre los dos lenguajes bases—. Por ejemplo, para acceder a un threadID, la estructura es prácticamente igual en ambas prácticas (ver línea 4 en los siguientes listings)

Listing 1: Suma vectorial CUDA-C++

```
1  // Inicio de kernel
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4      #Se acceden a los ID desde el ámbito del kernel
5      int i = threadIdx.x;
6      C[i] = A[i] + B[i];
7  }
8
9  int main()
10 {
11     ...
12     // Invocacion del kernel con 1 bloque y  N hilo
13     VecAdd<<<1, N>>>(A, B, C);
14     ...
15 }
```

Listing 2: Suma vectorial CUDA-Numba

```
# Inicio de kernel
@cuda.jit
def VecAdd(A,B,C):
    #En CUDA-Numba los hilos se acceden a través del entorno CUDA que es importado
    i = cuda.threadIdx.x
    C[i] = A[i] + B[i]

#Invocación del kernel con 1 bloque y N hilos.
VecAdd[1,N](A,B,C)
```

Al usar Numba como librería, se evita así usar CPython o Pybind para ligar código de C/C++ a Python. Logramos con esto una aplicación más fácil de mantener al tener todo unificado en un solo lenguaje.

Por otra parte, es bien sabido que C/C++ se mantienen los reyes del rendimiento en cuanto al tiempo de ejecución [NF15] y uno de los objetivos propuestos para este trabajo es conseguir un rendimiento notable.

Sin embargo, CUDA-Numba aguanta el tipo mostrando resultados de rendimiento similares a CUDA-C, pero solo si consideramos el trabajo hecho en GPU. Una vez que medimos las partes en CPU, el infame intérprete de Python ralentiza el cálculo general. [Ode20]

Aún teniendo en cuenta este inconveniente, viene perfecto a la hora de evaluar la plataforma como se propuso en los objetivos del trabajo.

2.6.2. Requisitos para CUDA-Numba

Cada GPU desarrollada por NVIDIA tiene un valor asociado llamado la capacidad computacional que indica las propiedades de cada procesador de dicha GPU: Operaciones atómicas en 64 bit, sincronización de funciones, paralelismo dinámico, núcleos de tensores, etc [NVF23a]

Para poder programar en CUDA-Numba, se necesita una GPU con una capacidad computacional mayor o igual a 5.0 [Ana]. Para la realización de este trabajo, se utilizarán las GPU: Geforce RTX 2080 y 3090, de capacidad computacional 7.5 y 8.6 respectivamente.

2.6.3. Compilado JIT en CPU con Numba

Al principio de la sección, hemos mencionado brevemente que Numba es una herramienta de compilado *JustInTime* (JIT), pero no hemos ahondado en el con-

cepto. JIT es un componente del entorno de ejecución que mora el rendimiento de Python compilando código *bytecode* en código de máquina nativo en el tiempo de compilación en vez de antes de la ejecución en sí.

Para ilustrar la diferencia con la ejecución normal en CPU, un compilador convencional levanta un programa como un fichero binario antes de su primera ejecución. Cuando un código se ejecuta en JIT, traduce a un código intermedio más portable y optimizable (bytecode) (ver Figura [5]). Este código es posteriormente interpretado y ejecutado con el compilador JIT, el cual lo traduce a código máquina nativo. Este código se compila justo cuando se va a ejecutar — de ahí el nombre del compilador —.

La optimización en JIT ocurre cuando se compila el código fuente a bytecode, cosa que ocurre antes del despliegue de la aplicación. Así nos aseguramos que la compilación a código máquina resulte mucho más rápida que ejecutando el código fuente a secas. [Cro]

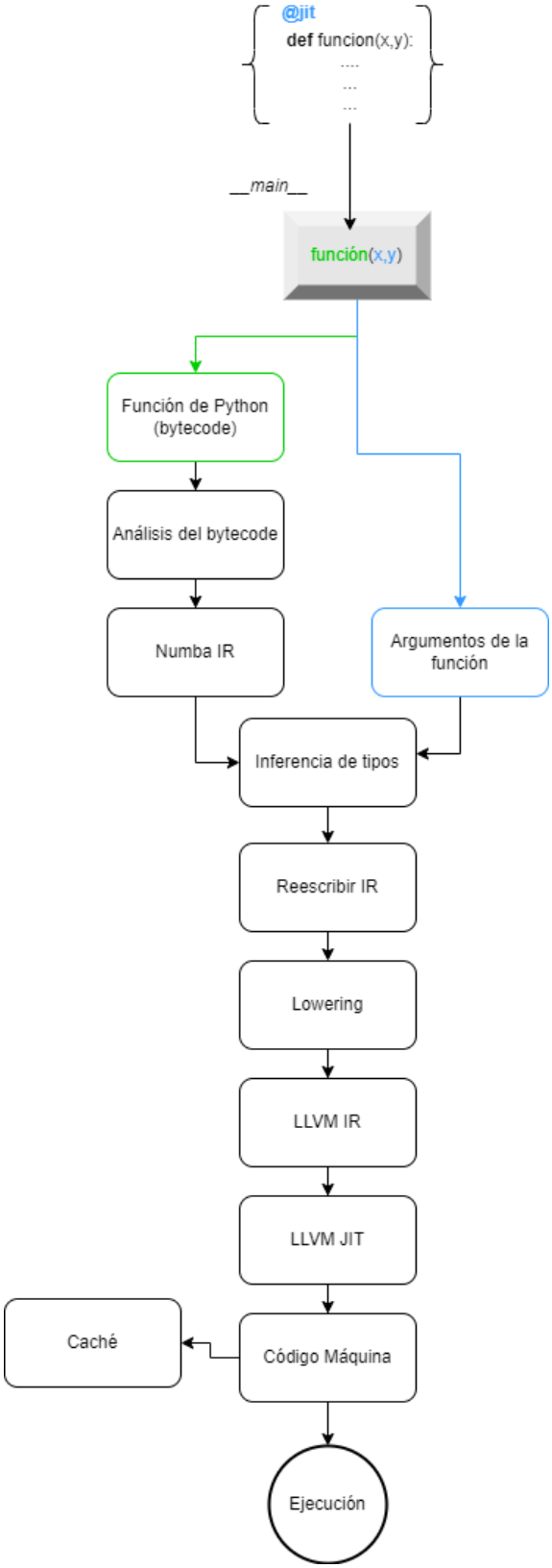

Figura 5: Diagrama de flujo de la ejecución de código en el compilador JIT de Numba.

3 Un paseo por la Máquina de Virus

3.1. Máquina de virus estándar

Un virus se comporta de una forma muy específica: como parásito se replica en otras células para poder reproducirse y es bajo esta base lo que se fundamenta las máquinas de virus. En sí, definimos formalmente una máquina de virus como:

Definición 1 *Una Máquina de Virus de grado* $(p, q), p \geq 1, q \geq 1$ *es una tupla*

$$\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out}),$$

donde:

- $\Gamma = \{v\}$ *es una instancia única, significa que solo un tipo de virus es representado en cada dispositivo;*
- $H = \{h_1, \ldots, h_p\}$ *y* $I = \{i_1, \ldots, i_q\}$ *son conjuntos ordenados tales que* $H \cap I = \emptyset$, $v \notin H \cup I$, *y* $h_{out} \notin I \cup \Gamma$*: cualquier* $h_{out} \in H$ *o* h_{out} *representa el entorno (denotado por* h_0*);*
- $D_H = (H \cup \{h_{out}\}, E_H, w_H)$ *es un grafo dirigido ponderado, donde* $E_H \subseteq H \times (H \cup \{h_{out}\})$, $(h, h) \notin E_H$ *para cada* $h \in H$, *out-degree*$(h_{out}) = 0$ *y* w_H *es un mapeado desde* E_H *hasta* $\mathbb{N} \setminus \{0\}$*;*
- $D_I = (I, E_I, w_I)$ *es un grafo bipartito ponderado, donde* $E_I \subseteq I \times I$, w_I *es un mapeado desde* E_I *hasta* $\mathbb{N} \setminus \{0\}$*. El grado de salida de cada nodo es menos o igual a 2;*
- $G_C = (V_C, E_C)$ *es un grafo bipartito no dirigido, donde* $V_C = I \cup E_H$ *siendo* $\{I, E_H\}$ *la partición asociada a él: cada arista conecta un elemento desde I con, como máximo, un arco desde* E_H*;*
- $n_1, \ldots, n_p$ *que representa el número inicial de virus en la configuración inicial.*
- i_1 *representa la instrucción inicial.*
- h_{out} *representa la región de salida.*

Una *Máquina de Virus* (VM, abreviando) $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ de grado (p, q), puede ser vista como un conjunto ordenado de p *hosts* etiquetados con $h_1, \ldots, h_p$, donde $n_1, \ldots, n_p$ *virus* son contenidos en cada *host*, respectivamente, y como un conjunto ordenado de q *instrucciones* etiquetadas con $i_1, \ldots, i_q$. El símbolo h_{out} representa la *región de salida*: puede ser un *host* en el caso de que $h_{out} \in H$ o h_{out} puede referirse al entorno en el caso que $h_{out} = h_0$. Arcos desde el grafo dirigido D_H representan *canales de transmisión* por los que los virus pueden ser transmitidos desde un *host* h_s (diferente de h_{out}) a otro *host* diferente $h_{s'}$, o el entorno. Si $s' = 0$, virus pueden salir del entorno. En cualquier momento, como mucho una sola instrucción puede ser activada y es entonces cuando el canal $(h_s, h_{s'})$ (arco G_C) con peso $w_{s,s'}$ ligado a él, será *abierto*. Entonces, $w_{s,s'}$ virus serán transmitidos/replicados desde h_s hacia $h_{s'}$. Por defecto, cada canal está *cerrado*.

Arcos desde el grafo dirigido D_I representan *transferencia de instrucciones*, y tienen asociado con ellos un peso. Finalmente, el grafo bipartito no dirigido G_C representa la *red de instrucciones-canales* por el cual una arista $\{i_j, (h_s, h_{s'})\}$ indica una relación entre instrucción i_j y canal $(h_s, h_{s'})$.

Gráficamente, una máquina de virus de grado $(3, 3)$ con 3 hosts y 3 instrucciones puede ser representada como una red heterogénea consistente de tres grafos, ilustrado en la Figura [6]. Cada host es caricaturado como un rectángulo y cada instrucción como un círculo. Cada flecha es un canal de transmisión de virus o un camino de transferencia de instrucciones que une las mismas; en ambos casos, cada flecha es asignada con un entero positivo que representa su peso (el peso 1 no se suele especificar por simpleza). Las relaciones entre instrucciones y canales se representan con líneas de puntos.

Seguidamente, se describe a continuación la **semántica** asociada con el modelo de computación de una máquina de virus. Una *descripción instantánea* o una *configuración* $\mathcal{C}_t$ en un momento t de una máquina de virus es descrita por la tupla $(a_{0,t}, a_{1,t}, \ldots, a_{p,t}, u_t)$ donde $a_{0,t}, a_{1,t}, \ldots, a_{p,t}$ son números naturales, $u_t \in I \cup \{\#\}$, donde $\# \notin H \cup h_0 \cup I$ es un objeto para caracterizar configuraciones interrumpidas. El significado de $\mathcal{C}_t$ es el siguiente: En el momento t el entorno contiene exactamente $a_{0,t}$ virus y el host h_s contiene exactamente $a_{s,t}$ virus, y si $u_t \in I$, entonces la instrucción u_t será activada en el paso $t + 1$ (al contrario, si $u_t = \#$, entonces ninguna instrucción será activada). La configuración inicial del sis-

tema $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, i_{out})$ es $\mathcal{C}_0 = (0, n_1, \ldots, n_p, i_1)$. Una configuración $\mathcal{C}_t = (a_{0,t}, a_{1,t}, \ldots, a_{p,t}, u_t)$ produce una configuración $\mathcal{C}_{t+1} = (a_{0,t+1}, a_{1,t+1}, \ldots, a_{p,t+1}, u_{t+1})$ en un *paso de transmisión* si podemos pasar desde $\mathcal{C}_t$ a $\mathcal{C}_{t+1}$.

3.1.1. Modelo suma

Todas estas reglas y directrices se combinan junto al diseño de unos **modelos** para dar a luz a una máquina de virus que realiza una acción predilecta —como por ejemplo: sumar, multiplicar, encontrar un máximo—. Por ejemplo, la máquina de virus definida en el anterior apartado (ver Figura [6]) tiene el funcionamiento de una simple suma de dos números al que llamaremos modelo suma.
Al iniciarse la máquina, se activaría $i1$, abriendo el canal de transmisión entre *host1* y *host3* permitiendo un virus viajar por dicho canal (ver Figura [7]). Como el modelo no tiene multiplicadores, el virus se resta y se añade en los respectivos *hosts* tal cual. Para iniciar el siguiente paso, la máquina tiene que calcular qué instrucción ejecutar a continuación. Entre las dos aristas dirigidas que salen de $i1$, hay que escoger la de mayor peso — en caso de que haya—. Por lo que la siguiente instrucción sigue siendo $i1$ (ver Figura [8]). El mismo cálculo que antes se repite para *host1* y *host3*.
Una vez llegados a este punto, a *host1* no le quedan virus que poder transmitir, así que $i1$ se ve forzado a cambiar de instrucción (ver Figura [9]). Para $i2$, el cálculo es el mismo que en los anteriores pasos, sin llegar a repetirse así misma — puesto a que *host2* solo albergaba un virus—. Llegados a este punto, $i2$ le pasa la antorcha a $i3$, nuestra instrucción final; que supondría el fin de la computación de este modelo. (ver Figura [10]).

3.2. Máquina de Virus Probabilística

Una *Máquina de Virus Probabilística* (VMP, abreviando) sigue muchos de los esquemas y formalidades que la máquina de virus estándar. Se presenta los siguientes cambios (resaltados en negrita):
$\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ de grado (p, q), puede ser vista como un conjunto ordenado de p *hosts* etiquetados con $h_1, \ldots, h_p$, donde $n_1, \ldots, n_p$ *virus* son contenidos en cada *host*, respectivamente, y como un conjunto ordenado

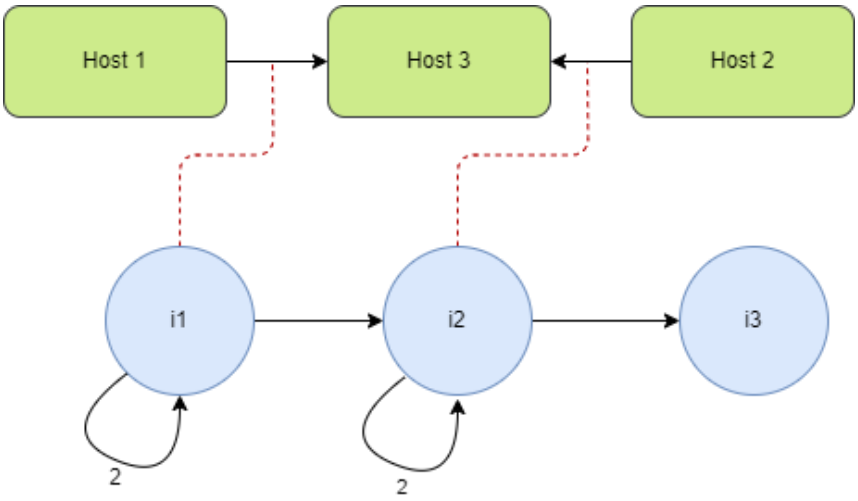

Figura 6: Ejemplo de VM(3,3)

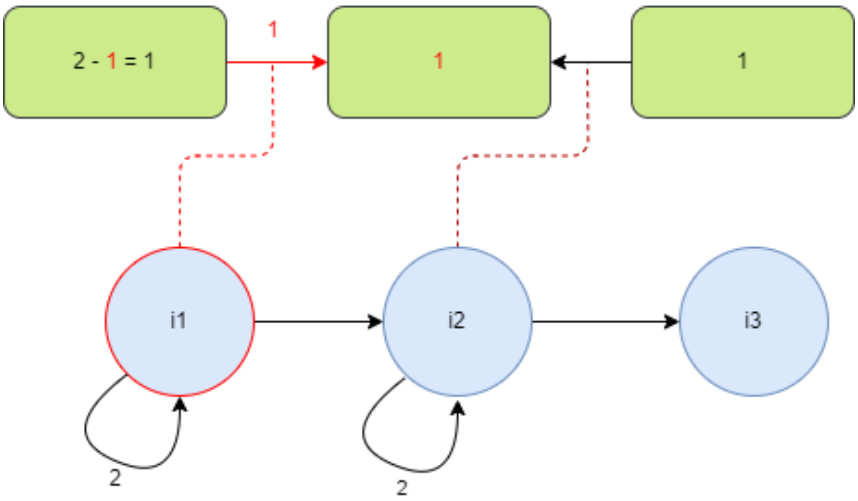

Figura 7: VirusMachine modelo suma. Primera iteración.

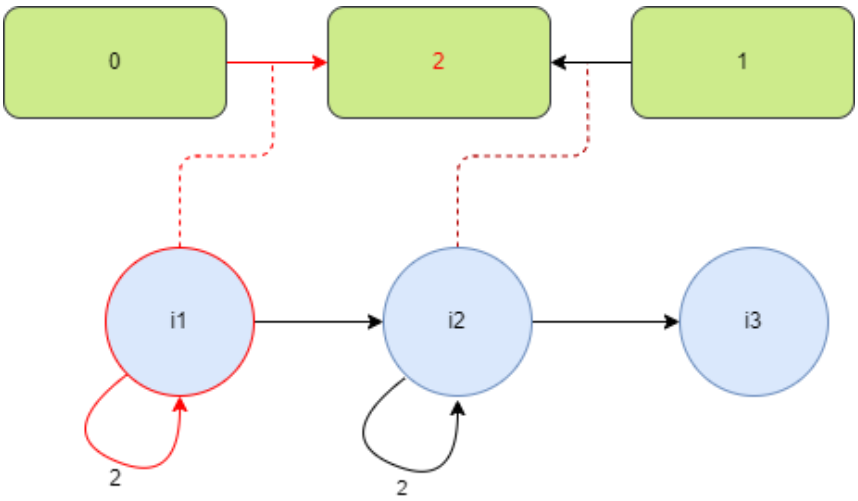

Figura 8: VirusMachine modelo suma. Segunda iteración.

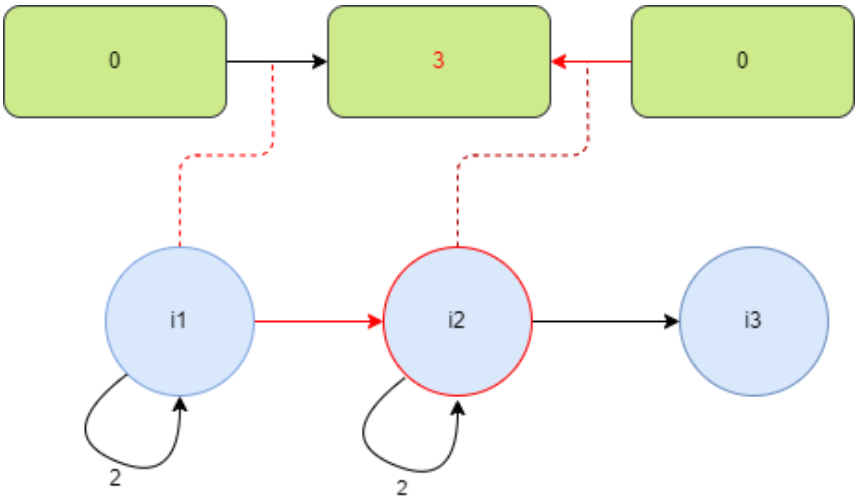

Figura 9: VirusMachine modelo suma. Tercera iteración.

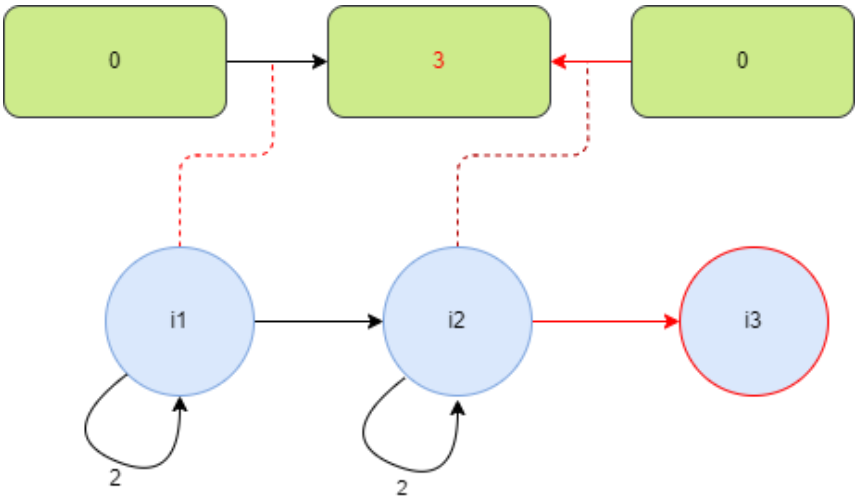

Figura 10: VirusMachine modelo suma — Última iteración.

de q *instrucciones* etiquetadas con $i_1, \ldots, i_q$. El símbolo h_{out} representa la *región de salida*: puede ser un *host* en el caso de que $h_{out} \in H$ o h_{out} puede referirse al entorno en el caso que $h_{out} = h_0$. Los arcos desde el grafo dirigido D_H representan *canales de transmisión* con peso $w_{s,s'}$ y **probabilidad** $p_{s,s'}$, por los que los virus pueden ser transmitidos desde un *host* h_s (diferente de h_{out}) a otro *host* diferente $h_{s'}$, o el entorno. Si $s' = 0$, virus pueden salir al entorno. En cualquier momento, como mucho una sola instrucción puede ser activada y, en esta nueva versión, **esta instrucción tiene asociada** un *host* (h_s) en vez de una instrucción concreta. De esta manera, un canal con respecto a la probabilidad $p_{s,s'}$ podrá ser *abierto*. Entonces, $w_{s,s'}$ virus serán transmitidos/replicados desde h_s hacia $h_{s'}$. Por defecto, cada canal está *cerrado*.

Arcos desde el grafo dirigido D_I representan *transferencia de instrucciones*, y tienen asociado con ellos un peso. Finalmente, el grafo bipartito no dirigido G_C representa la **red de intrucciones-hosts** por el cual una arista $\{i_j, h_s\}$ indica una relación entre instrucción i_j y el host h_s.

Gráficamente, se sigue el mismo esquema quen el modelo estándar a excepción que ahora la relación entre instrucciones y canales se ha cambiado entre **instrucciones y hosts**. Esta última relación se sigue representando con una línea de puntos.

3.2.1. Modelo Probabilístico Básico

La estructura que tiene este modelo es:

1. *Hosts*: $h1, h2$ y $h3$. $h1$ está conectado a los otros dos mediante canales con probabilidad p. $h1$ empieza la simulación con n virus

2. Instrucciones: $i1$ y $i2$. $i1$ habilita la apertura de los canales de salientes de $h1$ mientras que está conectada así misma y a $i2$ con peso 2 y 1 respectivamente. $i2$ no tiene asociada ningún *host* dado que es una instrucción final.

Alternativamente, se ilustra el mismo modelo en la figura [11]. En una primera iteración, la instrucción $i1$ es activada y se calcula aleatoriamente qué canal de transmisión se abre para que el virus viaje de *host*. Para eso se define una función de distribución acumulada, la cual toma las probabilidades de todos los canales posibles — en nuestro caso, solo dos —, y un número aleatorio y se calcula qué canal de transmisión se abrirá. (representados en las aristas dirigidas de $h1$).

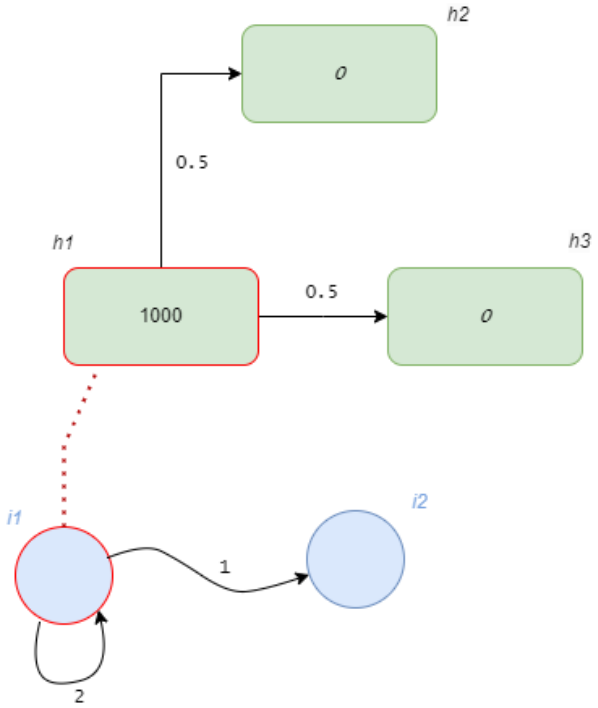

Figura 11: Modelo Probabilístico Básico. Estado inicial

Para este modelo, este mismo cálculo se repite hasta que $h1$ deja de tener virus que transmitir, por lo que $h1$ habrá transmitido n virus a $h2$ y $h3$ con una probabilidad p y $1-p$, respectivamente (ver Figura [12])

3.3. Máquina de Virus SoftParallel

En una primera estancia, en la máquina de virus no se planteaba espacio para el paralelismo. Todos los *hosts* podían abrir como máximo un canal de transmisión y solo una instrucción podía ser actividad en cada paso de la computación. Por eso mismo se baraja un nuevo modelo de sistema de virus, en el cual las instrucciones puedan ser ejecutadas simultáneamente en grupos.

Una *Máquina de Virus SoftParallel* (VMSP, abreviando)

$\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, L, h_{out})$ de grado (p, q), puede ser vista como un conjunto ordenado de p *hosts* etiquetados con $h_1, \ldots, h_p$, donde $n_1, \ldots, n_p$ *virus* son contenidos en cada *host*, respectivamente, y como un conjunto ordenado de q *instrucciones* etiquetadas con $i_1, \ldots, i_q$. El símbolo h_{out} representa la *región de salida*: puede ser un *host* en el caso de que $h_{out} \in H$ o h_{out} puede referirse al entorno en el caso que $h_{out} = h_0$. Arcos desde el grafo dirigido D_H representan *canales de transmisión* por los que los virus pueden ser transmitidos desde un *host* h_s (diferente de h_{out}) a otro *host* diferente $h_{s'}$, o el entorno. Si $s' = 0$, virus pueden salir

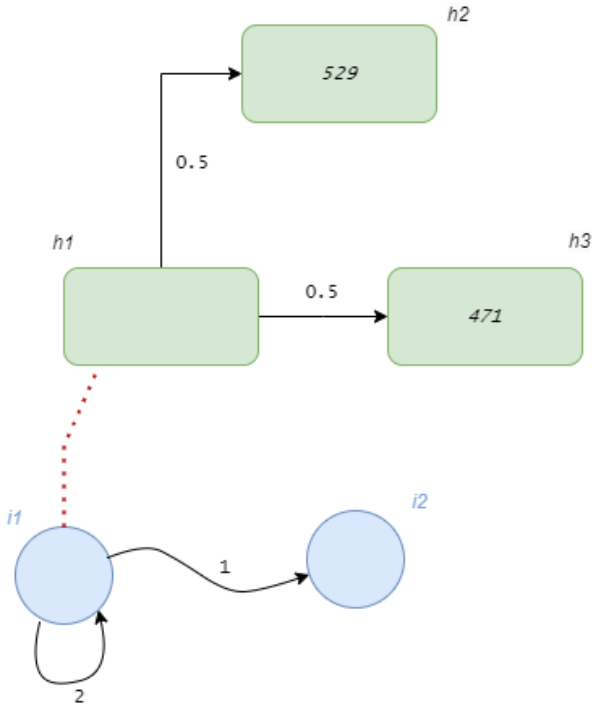

Figura 12: Computación concluida de una máquina de virus probabilística con $n = 1000$

del entorno. En el instante inicial, todas las instrucciones del conjunto L están activadas. En cada paso de computación, cada instrucción abrirá los canales de acuerdo a su funcionamiento en el modelo básico, con la diferencia de poder existir varias instrucciones abriendo canales al mismo tiempo. Si dos instrucciones abren el mismo canal y pasa un virus por este, ambas instrucciones elegirán el camino de mayor peso para elegir la siguiente instrucción. Si una instrucción no tiene siguiente instrucción, en lugar de usar el símbolo # como en el modelo básico, no se elegirá siguiente instrucción. Si el conjunto de instrucciones activas actuales es el vacío, la máquina de virus para. Igual que en el modelo básico, si una instrucción i está activa y está asociada a un canal $(h_s, h_{s'})$ (arco G_C) con peso $w_{s,s'}$ ligado a él, este canal se *abre*. Entonces, un virus podrá ser transmitido/replicado desde h_s hacia $h_{s'}$, llegando $w_{s,s'}$ virus al host s' (en caso de haber algún virus en el host s). Por defecto, cada canal está *cerrado*. Los arcos desde el grafo dirigido D_I representan *transferencia de instrucciones*, y tienen asociado con ellos un peso. Finalmente, el grafo bipartito no dirigido G_C representa la *red de intrucciones-canales* por el cual una arista $\{i_j, (h_s, h_{s'})\}$ indica una relación entre instrucción i_j y canal $(h_s, h_{s'})$.

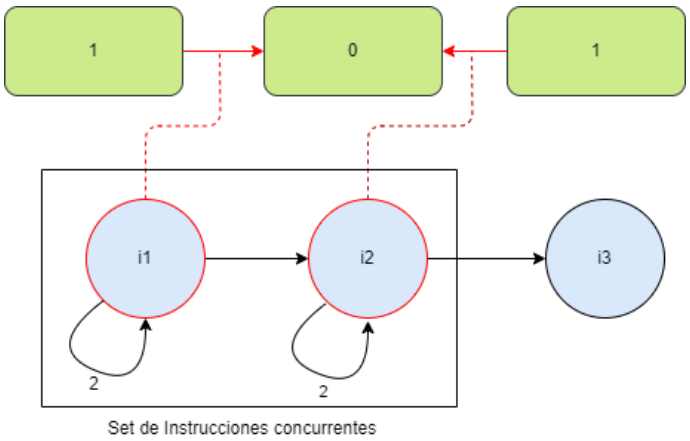

Figura 13: Modelo spsuma con $n = 2$

3.3.1. Modelo Suma SoftParallel

Este modelo es casi igual al modelo suma de la máquina de virus estándar, compartiendo grado, conexiones entre hosts e instrucciones-canales. Solo se añade un conjunto de instrucciones inicial, L, (ver Figura [13]) que contiene las instrucciones que se activarán en el primer paso de la iteración.
Por todo lo demás, funciona igual que el modelo suma.

4 El Simulador de Máquina de Virus

La máquina de virus consta de un simulador desarrollado en *Python* disponible en *GitHub* [Mar23]. De primeras, se planteó el mismo sistema como un sistema secuencial, por lo tanto, este simulador también lo es. Sin embargo, y como vamos a ver más adelante, se plantea un espacio para **una versión paralela**.

4.1. Simulador estándar

El simulador secuencial consta del entramado de varias clases que se apoyan entre sí para dar el funcionamiento final de una máquina de virus. Para dar un entendimiento preciso de cómo funciona el simulador, seguiremos el rastro, paso por paso, desde el llamamiento de un test de una máquina hasta el fin de la simulación. Para empezar, tomaremos un ejemplo sencillo, como es el proceso de ejecución del modelo suma.

Para cada modelo, se define una función $VM(I, H, IC)$ que devuelve un objeto VirusMachine al que I representa una lista de instrucciones, H una lista de *hosts* e IC el grafo de instrucciones y sus pesos. El caso del modelo suma puede ser el más sencillo para orientar su funcionamiento:

Listing 1: Definición del modelo suma

```
#La funcion coge dos atributos enteros que representan
# los numeros a sumar.
def suma(n1,n2):
    # Se crean tres objetos Host, uno de ellos vacío, que funcionará como
    # acumulador para la suma
    h1 = Host(n1)
    h2 = Host(n2)
    env = Host()
    hosts = [h1,h2,env]
    # Se crean tres instrucciones con el host al que apuntan.
    # La tercera instruccion no apunta a nada porque es la final
    i1 = Instruction(h1, env)
    i2 = Instruction(h2, env)
    i3 = Instruction(None, None, None)
    instructions = [i1,i2,i3]
    instruction_connections = [(i1,i1,2),(i1,i2,1),
                               (i2,i2,2),(i2,i3,1)]
    # Llamamiento a la clase VirusMachine para crear la maquina
    vm = VirusMachine(hosts, instructions, instruction_connections)
    return vm
```

Listing 4: Clase Instrucción

```
# La clase Instruction referencia a dos objetos tipo Host y un multiplicador
# que es por defecto 1.
class Instruction(object):
    def __init__(self, origin_host, objective_host, multiplier = 1):
        self.origin_host = origin_host
        self.objective_host = objective_host
        self.multiplier = multiplier
```

Listing 3: Clase Host

```
# En si, la clase Host solo es el numero de virus que tiene el host en
# el momento
class Host(object):
    def __init__(self, viruses = 0):
        self.init_viruses = viruses
        self.viruses = viruses
```

Listing 4: Inicialización de la clase VirusMachine

```
# Inicializacion de la clase VirusMachine con la lista de hosts, instrucciones
# y conexiones. Por defecto el sistema es determinista.
def __init__(self, hosts, instructions, instruction_connections, non_determinism=False):
        self.hosts = hosts
        self.instructions = instructions
        self.instruction_connections = instruction_connections
        self.non_determinism = non_determinism
        self.current_step = 0
        # Calcula la primera instrucción a ser computada a partir de la primera
        # instrucción en la lista de instrucciones
        if len(instructions):
            self.current_instruction = self.instructions[0]
        else:
            self.current_instruction = Nones
```

Una vez instanciado un objeto del tipo *VirusMachine*, ya se puede ejecutar el simulador. Dentro de dicha clase, se encuentra la función *compute()*, la cual llamada itera sobre el objeto hasta completar la simulación del modelo. En sí, la función *compute()* ejecuta *steps* hasta que la condición de parada se encuentra, momento en el cual la simulación termina. La función *step*() comprueba en el momento si se tiene una instrucción viable — es decir, que apunte a un canal con *host* de origen capaz de transmitir virus — y si es cierto, resta un virus del *host* origen y añade tantos virus como el multiplicador del canal de transmisión (como siempre se añade un virus por canal por cada *step* al final el multiplicador es el número de virus que se añaden) entre *host* origen y destino sea. Luego, se calcula la siguiente

instrucción para el siguiente *step* según el peso de las conexiones entre instrucciones — cuanto más peso, más prioridad—.

Al finalizar el *step*, devuelve una tupla (instrucción,string) con el cómputo de la iteración de *step*(). Sabiendo esto, la función *compute*() solo es una iteración de *steps* hasta que finaliza la simulación o lo que es lo mismo, se llega a la instrucción final.

Listing 5: Pseudocódigo de compute VirusMachine

```
Entrada: verbose (int), steps (long)
Salida: (int,string)
begin
    while (step() != None)
        # get_configuracion() es una funcion auxiliar que devuelve un string
        # del estado de la maquina de virus en el step actual
        print(get_configuracion());
    return (current_step,get_configuracion());
end
```

Listing 6: Pseudocódigo de step() VirusMachine

```
Variables: origin_host, current_instruction, objective_host, multiplier, step,
next_instructions
begin
    if (current_instruction.origin_host == True)
        # Calcula las siguientes instrucciones a partir de la actual
        next_instructions = next_instructions(current_instruction)
        if (origin_host.viruses != 0)
            resta_virus(origin_host);
            suma_virus(objective_host, multiplier);
            if next_instructions.size != 0
                if (no_determinista and multiplicador_igual(next_instructions))
                    current_instruction = escoge_aleatorio(next_instructions);
                else
                    current_instruction = next_instructions[0];
        # Caso de que no haya virus en el host origen
        else
            if next_instructions.size != 0
                if (no_determinista and multiplicador_igual(next_instructions))
                    current_instruction = escoge_aleatorio(next_instructions);
                else
                    current_instruction = next_instruction[-1];
        step += 1;
        return configuracion();
    else
        step += 1;
        return None;
end
```

4.2. Simulador probabilístico

El funcionamiento del simulador probabilístico es muy parecido al estándar. Son algunos cambios en clases lo que reflejan su comportamiento y diferencia con la máquina que ya conocemos. Al ejecutar una función *modelo* tenemos una instancia de nuestra clase $ProbabilisticVirusMachine$ hija de la ya conocida $VirusMachine$. Esta vez necesitaremos establecer **conexiones entre hosts** especificando el peso de los canales y la probabilidad de que se abran. Dado que ahora los canales se abren por dicha probabilidad, **las instrucciones ya no apuntarán a los canales, si no a los hosts**.

Listing 7: Función basicprobabilistic

```
def basicprobabilistic(n, p):
    h1 = Host(n)
    h2 = Host()
    h3 = Host()
    env = Host()
    hosts = [h1,h2,h3,env]
    # Definimos conexiones entre host: (hostA,hostB,peso,probabilidad)
    host_connections = [(h1,h2,1,p),(h1,h3,1,1-p)]
    i1 = ProbabilisticInstruction(h1)
    i2 = ProbabilisticInstruction(None)
    instructions = [i1,i2]
    instruction_connections = [(i1,i1,2),(i1,i2,1)]
    vm = ProbabilisticVirusMachine(hosts, host_connections, instructions,
    instruction_connections)
    return vmb
```

El siguiente cambio más sustancial es en la misma función *step*(). Ahora, se tiene que calcular **cuál es el host objetivo** cada paso de la computación (ver Figura 14). Este cálculo viene predeterminado por una probabilidad asociada a cada host. Por lo demás, funciona igual que el simulador estándar.

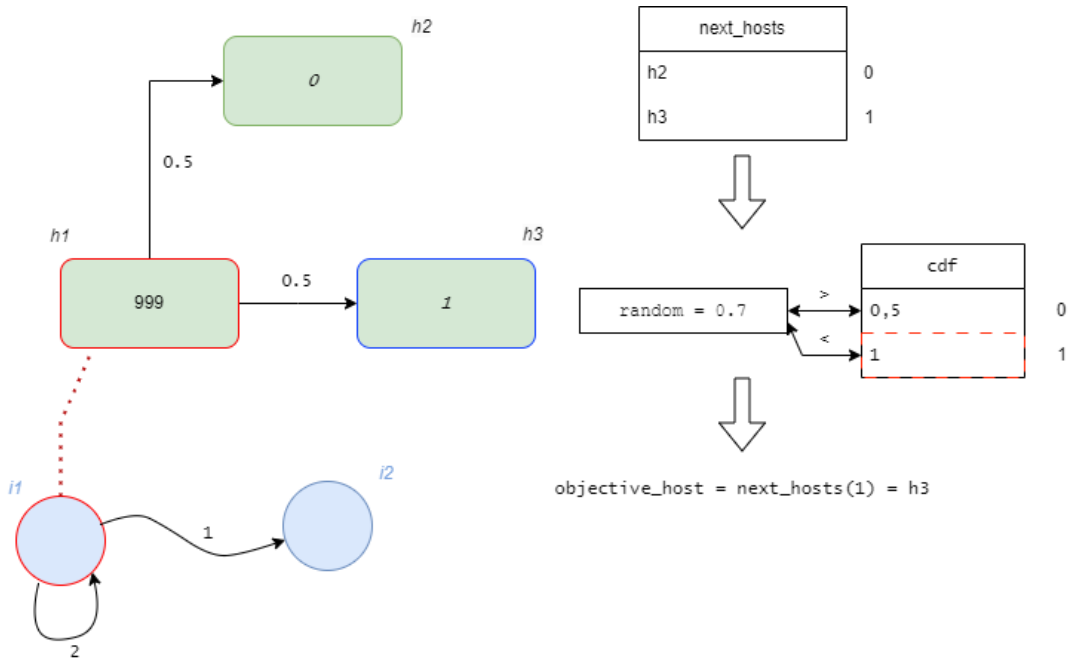

Figura 14: Primera iteración del simulador probabilístico básico

Listing 8: Pseudocódigo step() ProbabilisticVirusMachine

```
Variables: current_instruction, host, next_instructions, step
begin
    if (current_instruction.host == True)
        if host.viruses != 0
            resta_virus(host);
            # Mediante la probabilidad asociada al host, se calcula el canal
            # que se va a abrir.
            objective_hosts, index = calcula_siguiente_host(host);
            objective_host = objective_hosts[index][0];
            multiplier = objective_hosts[index][1];
            suma_virus(objective_host,multiplier);
            if next_instructions.size() != 0
                current_instruction = next_instructions[0][0];
            else
                current_instruction = None;
            step += 1;
        else
            if next_instructions.size() != 0
                current_instruction = next_instructions[-1][0];
            else
                current_instruction = None;
            step += 1;
        return get_configuracion();
    else
        current_instruction = None;
        step += 1;
        return None;
```

4.3. Simulador softparallel

Este nuevo modelo se implementa directamente sobre el simulador original, creando una clase hija a *VirusMachine*, $spVirusMachine$, la cual implementa un nuevo método *step()*. Al disponer de una lista de instrucciones concurrentes, se debe calcular **cuáles son los host orígenes y objetivos** de dicho conjunto de instrucciones, además de qué instrucciones serán prioritarias o no ($instructions - upper$ e $instructions - lower$ en el simulador). Se crea además un diccionario donde se guardará como $[host - origen : (instruccion, indice - host - objetivo)]$ para representar qué canales se van a abrir en el paso de ejecución. La figura [15] representa gráficamente el paso de esta parte del algoritmo.

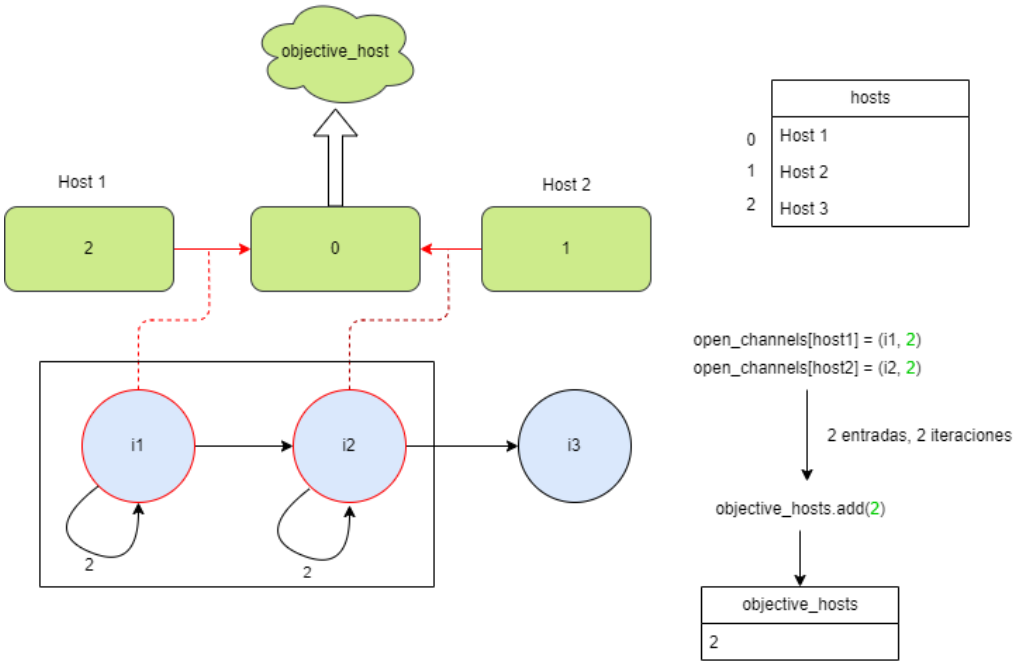

Figura 15: Función *step*(). Se calcula qué canales de transmisión se abrirán a partir del conjunto de instrucciones.

Una vez decididos los *hosts* de origen —o *host*, en singular, en caso de que todas las instrucciones apunten a uno sólo como en el modelo propuesto—, se debe comprobar si el número de canales abiertos de un *host* supera su número de virus. Esto es natural, ya que no podrá transmitir un virus si queda en virus 'negativos'. En caso de que esto ocurra, se decide aleatoriamente qué canales son los que se abren y se guarda la prioridad de las instrucciones ligadas a los canales 'premiados' en $intructions - upper$, mientras que las instrucciones que apuntan a los canales no escogidos se guardan en $instructions - lower$. En el caso contrario en el que no es necesario calcular aleatoriamente los *hosts* objetivos, las instrucciones se

guardan por defecto en $instructions - upper$. En nuestro ejemplo (figura [16]), tenemos claramente el segundo caso, al haber suficientes virus en cada host en relación a canales, se pueden abrir todos estos canales. Las listas de instrucciones sirven como acumuladores para añadir una preferencia a nuestro $next - instructions$ de modo que **los canales que son abiertos son los que más prioridad tienen de volver a ser abiertos.** Una vez que los canales se abren, los virus pasan y se restan y suman en los *hosts* correspondientes y se escogen el nuevo conjunto de instrucciones tal y como se ilustra en la figura [17].

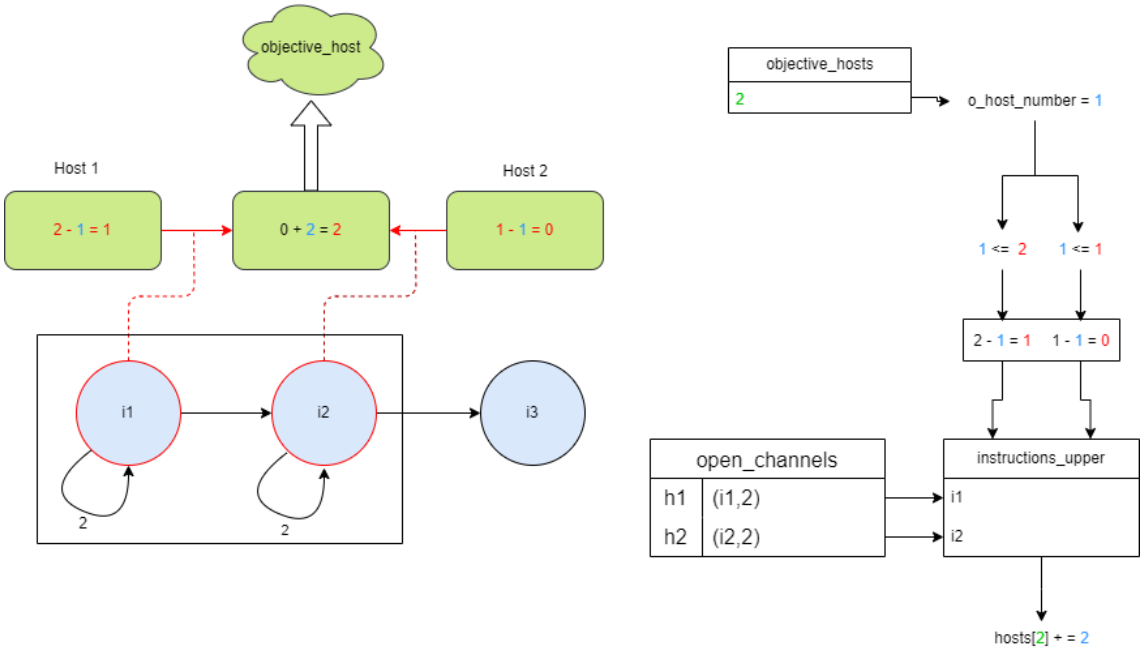

Figura 16: Función *step*(). Se realiza la transmisión de virus para h1 y h2.

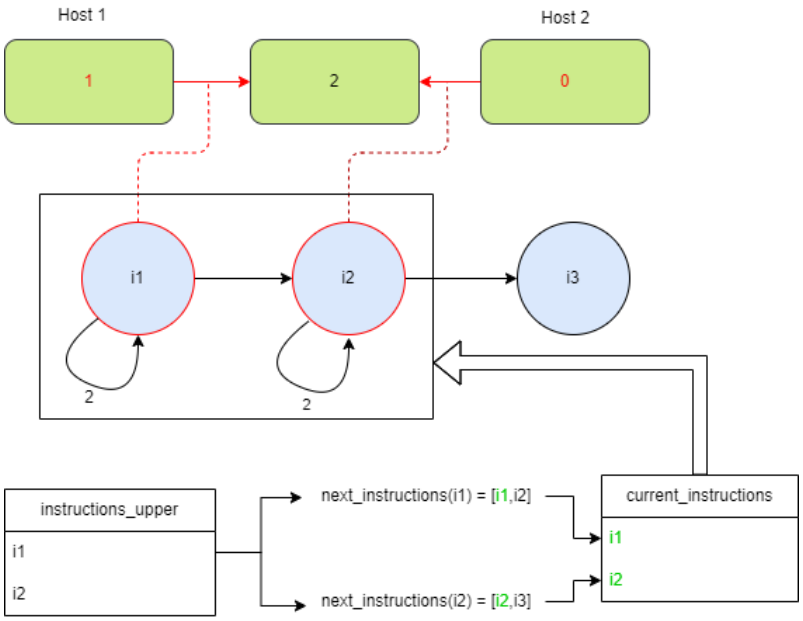

Figura 17: Función *step*(). Se calcula el siguiente conjunto de instrucciones, finalizando el step.

Listing 9: Pseudocódigo de step() spVirusMachine

```
Variables: current_instructions(set), hosts (set), objective_hosts (set),
step(int), open_channels(dict), o_host_number(int), instructions_upper(list),
instructions_lower (list)
begin
    if (quedan_instrucciones)
        for instruccion in current_instructions
            # Calculamos el índice asociado a cada host objetivo según la lista de
            # hosts para posteriormente guardar el par (instruccion, indice) en el map
            ind = calcula_indice(hosts, instruccion.objective_host);
            open_channels[origin_host].add_entry(instruccion,ind);
        for origin_host in open_channels
            for host in open_channels[origin_host]
                objective_hosts.add(host[1]);
                o_host_number += 1;
            if (o_host_number <= origin_host.viruses)
                resta_virus(origin_host,o_host_number);
                seleccionar_objective_host(objective_hosts);
                for elem in open_channels[origin_host]
                    instructions_upper.add(elem[0])
            else
                resta_virus(origin_host,origin_host.viruses);
                seleccionar_objective_host_random(objective_hosts);
                for elem in open_channels[origin_host]
                    # Priorizamos los canales que han sido escogidos
                    if elem[1] in objective_hosts
                        instruction_upper.add(elem[0])
                    else
                        instruction_lower.add(elem[0])
        # Se suman virus en los hosts objetivos correspondientes.
        suma_virus(objective_hosts);
        # Por cada instrucción en el set de instrucciones prioritarias,
        # se agrega la primera instrucción que le preceda
        for instruccion in instructions_upper
            n = next_instructions(instruction);
            if n.size() != 0
                if no_determinista()
                    escoge_instrucciones_random(n);
                else
                    escoge_instruccion_primera(n);
        # Por cada instruccion en el set de instrucciones secundarias,
        # se agrega la ultima instrucción que le preceda
        for instruccion in instructions_lower
            n = next_instructions(instruction);
            if n.size() != 0
                if no_determinista()
                    escoge_instrucciones_random(n);
                else
                    escoge_instrucciones_ultima(n);
        current_instructions = n
        step += 1;
```

```
51          return configuracion()
52      else
53          step += 1;
54          return None
55  end
```

5 Ejecución: Aplicar lo aprendido

Ahora que entendemos cómo funciona CUDA y el sistema de Máquina de Virus, podemos brindar una versión paralela a algunos de sus ejemplos. Ya se ha hecho bastante hincapié en que el procesamiento en GPU se espera que sea mucho más rápido que en la versión secuencial — y sobretodo cuando se quieran hacer simulaciones simultáneas —, sin embargo, esto lleva un coste de pérdida de generalidad. En resumen, casi todas las ventajas que nos brinda la programación orientada a objetos se pierde en este proceso. Por ello se apunta a realizar simuladores específicos para modelos definidos —simulador ad-hoc— y no una versión genérica. Podemos dividir entonces nuestro problema en dos tareas:

1. Ejecutar un gran número de simulaciones independientes. Se darán varios tipos de soluciones para esto, tanto en CPU como en GPU con CUDA.

2. Crear una instancia de una máquina de virus intrínsecamente paralela. Utilizaremos el modelo spsuma para este subproblema.

Todo el código desarrollado en este capítulo, junto al simulador estándar, se encuentra disponible en la plataforma *GitHub*[2] [Gar23]

5.1. Conversión del simulador

Nuestro problema inmediato a la hora de fijar nuestra solución es tener un suelo firme en el que empezar a construir. Como hemos visto antes, para tener el apoyo de Numba necesitamos adaptar nuestro código y evitar toda incompatibilidad. Para ello, se adaptará el simulador con *arrays* debido a su predisposición a ser utilizados de manera óptima por la GPU e incluso para la CPU en el compilador JIT, como vamos a ver en las soluciones específicas para cada subproblema.

Listing 1: Ejemplo básico de inicialización de variables

```
1  variables: int n_virus, int n_sim, n_hosts
2  begin
3      matriz_simulaciones ← array((n_sim,n_hosts))
4      programa(matriz_simulaciones, n_virus)
5  end
```

[2]Enlace: https://github.com/Servelgar/TFG-NUMBA-CUDA

Entiéndase por n_{virus} como el número de virus de cada simulación, n_{sim} como el número de simulaciones que se quieren ejecutar, n_{hosts} el número de *hosts* de la máquina de virus y $matriz - simulaciones$ la adaptación a matriz del modelo a simular. Esto es solo un ejemplo genérico, la realidad es que vamos a necesitar una o varias matrices dependiendo del simulador específico.

5.2. Basicprobabilistic — Ejecución de múltiples simulaciones

Tal y como hemos expuesto al principio del capítulo, nuestro objetivo es realizar múltiples simulaciones independientes del mismo modelo en pos de:

(a) Realizar un benchmark y estresar la GPU para probar su rendimiento

(b) Validar el modelo probabilístico

Para brindar una solución, debemos de realizar una computación de la máquina probabilística por cada simulación que queramos hacer.

5.2.1. Solución en Python puro

Para tener una comparativa clara, necesitaremos representar nuestro problema en el clásico escenario de la CPU y comparar resultados. Se escribirá un código en *Python* para que haga múltiples computaciones de una misma máquina de virus y finalmente hacer la media de los resultados.
Se expone un pseudocódigo que define la solución para este problema:

Listing 2: Esquema test probabilístico CPU (No JIT)

```
1  begin
2  # INICIALIZAMOS VARIABLES
3      i = 0
4      n_vm = 10000 # Numero de simulaciones
5      n = 1000   # Numero de virus
6      p = 0.5    # Probabilidad en abrirse el canal entre Host1 y Host2
7      l_hosts = [(0,0)] // lista inicial
8      vm = basicprobabilistic(n,p) # creamos una instancia de maquina de virus
9  # Iteramos por cada numero de simulacion sobre la maquina de virus
10     while (i < n_vm)
11       (_,lista_compute) = vm.compute()
12       # Guardamos el resultado de los hosts destino
13       host2 = lista_compute[1]
14       host3 = lista_compute[2]
15       # Insertamos en la lista los resultados anteriores
16       # Por lo tanto, hemos guardado los resultados de la simulacion i
```

```
17        l_hosts.insert((host2,host3))
18        i = i + 1
19        vm.restart() # reiniciamos la maquina para la siguiente simulacion
20      end
21       l_hosts.pop(0)
22    end
```

Con esto obtenemos una lista de listas con los valores de los *hosts* 2 y 3 al final de cada computación, siendo la longitud de la lista igual al número de simulaciones. Para plantear la solución a nuestro problema, necesitamos una lista de probabilidades acumuladas que toma las probabilidades de cada simulación supeditadas a la anterior, de forma que la probabilidad i será igual al sumatorio de probabilidades hasta i dividido entre el número de probabilidades por el número de virus por simulación.

Listing 3: CPU probabilidades acumuladas (sin JIT)

```
1   begin
2  # INICIALIZACION DE VARIABLES
3      acum1 = 0
4      acum2 = 0
5      l_res = []
6      i = 0
7      while (i < l_hosts.size())
8  # Por cada par en nuestra lista, sumamos en sendos acumuladores.
9        acum1 += l_res[i][0]
10       acum2 += l_res[i][1]
11 # Las medias acumuladas seran el cociente entre los acumuladores
12 # por el numero de virus multiplicado por el numero de simulacion
13       media1 = acum1 / ((i+1)*n)
14       media2 = acum2 / ((i+1)*n)
15       l_res.insert([media1,media2])
16       i++
17     return l_res
18  end
```

5.2.2. Solución en Numba JIT (CPU)

Este código, sin embargo, no tiene en cuenta el compilador JIT de Numba. Su defecto, como anteriormente hemos expuesto, es que necesita funcionar en tipos de variables "básicas"para llegar a una mejora de rendimiento considerable por lo que no podremos pasarle como variable objetos de clase VirusMachine para iterarlos. Este defecto solo afecta si necesitamos los beneficios de un objeto genérico como son las clases de Python. Ya no podemos utilizar las extensas funciones e instan-

cias que nos brindaba las clases ya implementadas y en su lugar tenemos que simular el funcionamiento de máquina de virus específicas con matrices. Esta misma restricción, como ya se ha inferido, también la comparte con GPU, así que utilizaremos la misma base en ambas partes.

Dicha base comienza con la creación de una matriz, M la cual guardará los resultados de las simulaciones. M tendrá una dimensión (n_{sim},3). El valor M_{i_j} representará el número de virus en la simulación i en el host j. Luego, nos serviremos de un array auxiliar *cdf*, el cual servirá para iterar sobre las probabilidades acumuladas.

Listing 4: CPU basicprobabilistic (JIT)

```
begin
# En la version JIT no hay posible llamada a funciones de alto nivel o instancias
# de objetos. Para ello se trabaja con matrices y operaciones aritmeticas
# simplificando la idea del simulador lo maximo posible sin perder su
# funcionamiento original
@jit
# INICIO DE VARIABLES
  matrix cdf = [p, (p+(1-p))]
  int i = 0
   # Recorremos secuencialmente la matriz por sus filas
   # Cada fila representa una simulacion
  while i < array_sim.shape[0]:
       int j = 0
       # Una vez en la simulacion, hay que computar el modelo probabilistico
       while j < n:
           array_sim[i][0] -= 1 #El host1 pierde un virus
           rnd = random.random()
           index = 0
           # Seleccionamos aleatoriamente que canal se mueve el virus
           # Dependiendo de la probabilidad guardada en cdf
           while rnd > cdf[index]:
               index += 1
           end
           array_sim[i][index+1] += 1 #el host destino recibe un virus
           j += 1
       end
       i += 1
   end
   return array_sim
end
```

Como se puede ya inferir, la diferencia entre este tipo de simulador y el original es extensa. Se insta simplificar el concepto de cada modelo y su computación — ya que dependiendo del modelo, la computación del simulador puede ser distinta,

véase las diferencias entre softparallel y probabilistic—.

5.2.3. Solución en Numba CUDA

Como vamos a ver más adelante, el inconveniente que tiene la CPU con el problema dado es el número tan grande de operaciones concurrentes que tiene que hacer. En un número grande de simulaciones — como 10.000 o incluso 1.000.000 — el tiempo de ejecución puede ser un problema.
Por esto mismo, para la solución en CUDA, vamos a ejecutar tantos hilos como número de simulaciones queramos hacer. Tenemos la ventaja que las simulaciones son independientes entre sí, es decir, no va a haber ningún tipo de convergencia entre hilos durante la computación.
Necesitamos una serie de variables para poder ejecutar nuestro kernel:

1. Matriz de entrada. CUDA Numba soporta las matrices de la librería *numpy*. Las dimensiones de la matriz inicial, M será: ($nsim$,n). Donde $nsim$ es el número de simulaciones y n el número de virus inicial del $host1$. En cada posición de la matriz se calculará, a partir de una probabilidad dada y un número aleatorio, si un virus acaba en un *host* u otro (ver Figura [18]). Se decide abarcar tales dimensiones para conseguir una maximización de hilos por instrucciones en GPU —diferenciando así a la matriz de la solución por JIT, que requería solo una matriz con tres columnas—, además de facilitar la generación de una aleatoriedad independiente entre elementos. Luego se calculará una última matriz, S, con la suma de asignaciones calculadas en M. S tendrá dimensiones ($nsim$,2) —una columna por *host* objetivo—.

2. Bloques e hilos. Pasados como entrada para el kernel, establece la configuración de ejecución del kernel en la GPU. El número de bloques está ligado al de hilos y la variable que se quiere paralelizar, en este caso el número de simulaciones.

3. Estado aleatorio. Numba no da soporte a la función natal de CUDA de generar números aleatorios. En su lugar, utiliza el generador de números aleatorios *xoroshiro128p*, que necesita un estado aleatorio para poder crear dichos números en cada simulación.

La máquina *basicprobabilistic* arranca con un host albergando un número de virus

fijo. Para los ejemplos que se han expuesto y para los que se seguirán exponiendo, ese parámetro tiene valor 1000.

El kernel opera sobre una matriz de dos dimensiones, por lo que es recomendable definir los índices de los hilos en dos dimensiones también ($threadIdx.x$ y $threadIdx.y$ en nuestro kernel, los cuales pertenecen a un $grid2D$). Calculará, virus por virus (o posición por posición), cuál es el *host* objetivo. Para simplificar, se ha asumido que el valor 0 representa el $host2$ y el valor 1 el $host3$. Para calcular esta asignación, se tiene que operar sobre un número aleatorio que **debe ser independiente** de cualquier otro elemento de M. Para ello, se itera de bloque en bloque en ambas dimensiones del $grid$, dando a cada hilo un índice propio para la semilla del número aleatorio.

El coste de proporcionar 'verdadera' aleatoriedad al simulador ralentiza mucho sus cálculos. Por ejemplo, en un primera versión del simulador, se pensó superar este cuello de botella haciendo que cada hilo del bloque calculase su propio número aleatorio proporcionando como semilla su propio ID. Para dimensiones de matriz pequeñas daba 'buenos' resultados — aunque no tan buenos como los que se proporcionan en el siguiente capítulo–, sin embargo, para dimensiones de matriz grandes la probabilidad acumulada decaía drásticamente de la esperada. Esto se debía a que se **repetían varias semillas** en distintas posiciones y, por lo tanto, quebrando la independencia probabilística entre los elementos de la matriz. Al iterar de bloque en bloque — en ambas dimensiones del problema —, nos aseguramos de que cada hilo tiene una semilla única.

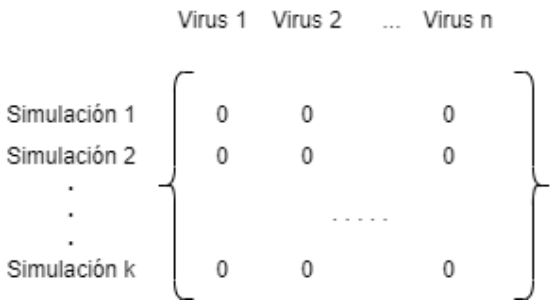

Figura 18: Valor inicial para M.

Listing 5: CUDA kernel compute — basicprobabilistic

```
@cuda.jit
def kernel_compute(array_sims,p, rng_state):
    # Indices para recorrer la matriz por filas y columnas
    x = cuda.threadIdx.x
    y = cuda.threadIdx.y
    columna = cuda.blockDim.y * cuda.blockIdx.y + y
    fila = cuda.blockDim.x * cuda.blockIdx.x + x

     # Tam. del grid
    stridex, stridey = cuda.gridsize(2)
    # Se crea un array local para la probabilidad
    cdf = cuda.local.array(shape=2, dtype=np.float64)
    cdf[0] = p
    cdf[1] = p + (1-p)
    # El generador de numero aleatorios necesita una semilla acorde a la fila
    # y columna para que la aleatoriedad sea fiable. Para ello se calcula idx
    # a partir de los indices y el size del grid y se le pasa a la funcion
    # generadora como semilla.
    for i in range(startx, array_sims.shape[0], stridex):
        for j in range(starty, array_sims.shape[1], stridey):
            # semilla de cada hilo
            idx = i * stridey + j
            rng = xoroshiro128p_uniform_float32(rng_state,idx)
            while (rng > cdf[index]):
                index = 1
            array_sims[i, j] = index
```

Recordemos que en CUDA, los kernel no devuelven ningún valor, si no que sobrescriben una matriz que ha sido otorgada como parámetro. Esa misma matriz reside en la GPU cuando el cálculo se ha terminado (en el caso de M, ver figura 20) por lo que se debe volver a copiar a la CPU (ver Figura [19]).

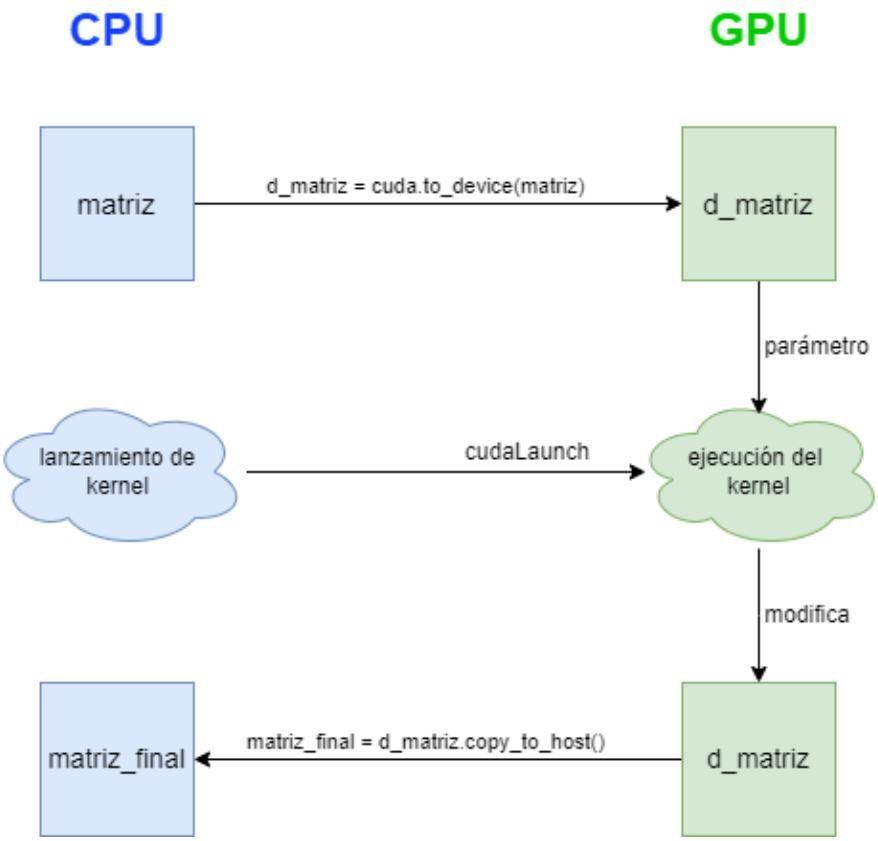

Figura 19: Esquema de movimiento de una matriz

La agrupación de filas de la matriz resultante será la solución de nuestra simulación. La suma de filas se guardará en una matriz que llamaremos S. Esta matriz tendrá tantas filas como simulaciones y sólo dos columnas que representan el *host2* y *host3* — dado que el valor del *host1* es siempre 0 al finalizar la computación— (ver Figura[21]). El problema es relativamente sencillo, según el valor de $M_{i,j}$, se suma en la posición $S_{i,x}$. Por ejemplo, si $M_{1,1} = 1$ se añadirá un virus en $S_{1,0}$. O lo que es lo mismo, se suman unos por filas y el número resultante será el número de virus del *host*3, si calculamos la diferencia con el número de virus totales, tenemos el número de *host*2.

Listing 6: CUDA kernel agrupación — basicprobabilistic

```
# Pero por el momento vamos a aprovecharnos de que en el modelo de
# basicprobabilistic solo hay dos hosts finales.
# Entonces, calculamos los virus del primer host y calculamos la diferencia
@cuda.jit
def k_suma_virus_hosts(array_sims, array_suma):
    fila = cuda.grid(1)
    if fila < A.shape[0]:
        fila_sum = 0
        # Iteramos sobre cada fila y guardamos en el acumulador
        for j in range(A.shape[1]):
```

```
11                 fila_sum += A[fila,j]
12             C[fila,0] = fila_sum
13             C[fila,1] = A.shape[1] - fila_sum
```

Seguidamente, tenemos que inicializar variables y prepararlo todo para la invocación de los *kernels*. Ambas funciones definidas tienen *grids* 2D y 1D respectivamente, por lo que tenemos que adaptar el número de hilos a ello. Puesto que parte de nuestro problema es estresar a la GPU, se definirá el máximo de hilos posible (1024). Con este número, definimos el número de bloques por *grid* como la división entera entre las dimensiones de los *arrays* a calcular entre el número de hilos. En el caso de nuestro kernel $k - suma - virus - hosts$, el número de bloques está supeditado a la dimensión 0 del array $array - agrupacion$, debido a que es la dimensión en la que se mueve la paralelización de la función. Luego, para crear el estado de aleatoriedad que pasamos como atributo a $kernel - compute$ llamamos a la función que nos ofrece Numba $create - xoroshiro128p - states(n, seed)$, donde n es el número de estados aleatorios que queremos —uno por cada posición de la matriz M—. Opcionalmente, podemos seleccionar qué GPU queremos para realizar los cálculos con $cuda.select - device(x)$.

Listing 7: Solución Numba CUDA — Inicialización e invocación de kernels

```
########## VARIABLES #########
n = 1000  # numero de virus
n_sim = 100  # numero de simulaciones
p = 0.5 # prob

######## Preparar los arrays ######
array_sims = np.zeros((n_sim,n), dtype=np.int32)
array_sims_final = np.zeros((n_sim,2), dtype=np.int64)

############ INICIALIZACION DE HILOS, ETC #################
cuda.select_device(1)
threads_per_block = (32,32)
blocks_per_grid_x = math.ceil(array_sims.shape[0] / threads_per_block[0])
blocks_per_grid_y = math.ceil(array_sims.shape[1] / threads_per_block[1])
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)
nTPB = int(np.prod(threads_per_block))
nB = int(np.prod(blocks_per_grid))
rng_states = create_xoroshiro128p_states(int(np.prod(threads_per_block)
                    * np.prod(blocks_per_grid)), seed=1)

#################### INVOCACION  ######################
d_array_sims = cuda.to_device(array_sims)
kernel_computeN[blocks_per_grid, threads_per_block](d_array_sims,p, rng_states)
d_array_final = cuda.to_device(array_sims_final)
s_reduce_sum[nB,nTPB](d_array_sims,d_array_final)
array_sims = d_array_sims.copy_to_host()
array_sims_final = d_array_final.copy_to_host()
```

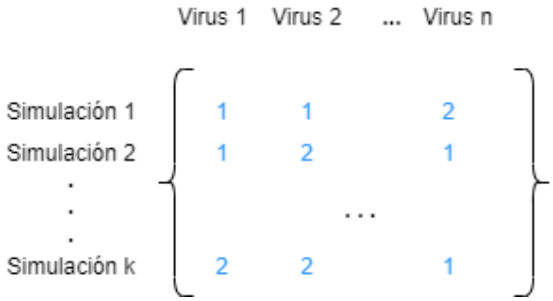

Figura 20: Ejemplo de un valor final para M. Al finalizar la computación, algunas posiciones valen 0 y otros 1.

5.3. SoftParallelSuma

5.3.1. Solución para Numba-CUDA

Entendemos como el modelo *spsuma* al modelo suma básico que ha sido implementado en la clase *spVirusMachine*. Al contrario del basicprobabilistic, ahora

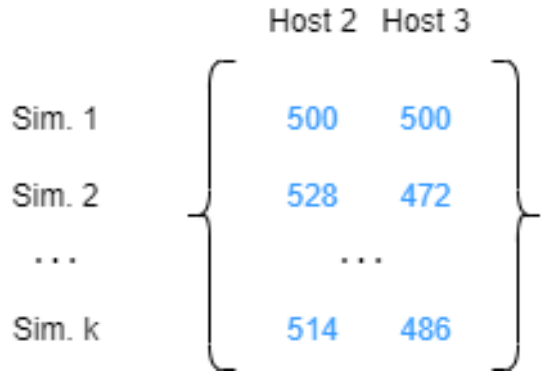

Figura 21: Valor final para S.

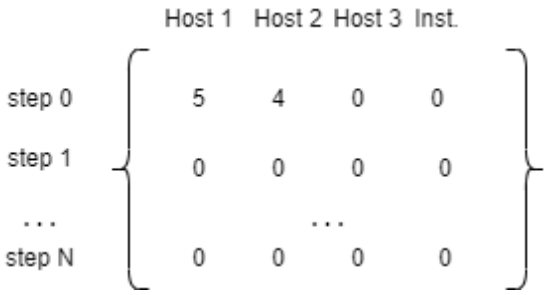

Figura 22: Valor inicial de SP.

nos interesa computar una sola simulación del modelo, reflejando cada paso de la simulación de alguna forma. Se trabajará con una matriz SP con N filas y 4 columnas. N será igual al número de *steps* que queremos ejecutar del sistema y cada columna representa una variable del problema. (ver figura [22]). Se añade el número de instrucciones en esta cuarta columna para llevar cuenta de cuantas instrucciones se activan por cada *step*.

La cuarta columna guarda un índice que indica qué instrucciones o instrucción se ha ejecutado para la computación de dicho step. Para *spsuma* solo hay dos instrucciones como máximo, así que el valor más alto que puede ocupar dicha columna es 2.

La primera dificultad que nos encontramos, es que cada *step* no puede ser computado de forma independiente, ya que el simulador 'tiene que saber' cuándo hay un $host = 0$. En ese mismo caso, el simulador pasa a ser 'mono-instruccional'. Un ejemplo más básico de esto es cuando queremos sumar dos números distintos entre sí: Si sumamos $2 + 1$ con este modelo, en el primer *step* se activarían en paralelo $i1$ e $i2$, abriendo ambos canales emergentes de $h1$ y $h2$ (puede volver a ver la Figura 13 si no recuerda su representación gráfica). Al final de este paso, $h2$ se habrá quedado sin virus que transmitir, por lo tanto $i2$ deja de tener que hacer pasar virus.

Por lo que, virtualmente, el conjunto de instrucciones es como si solo tuviera a $i1$. Para comprobar esto, el kernel tiene que mirar, por cada posición de la columna, si alguno de los anteriores elementos es 0. Luego establece el número de canales que se abren y suma dicho número al host objetivo $h3$.
En sí, el kernel adquiere la filosofía de $SPVirusMachine$ y paraleliza por conjunto de instrucciones, lanzando un hilo por número de instrucciones.

Listing 8: CUDA kernel compute spsuma

```
@cuda.jit
#SP y un array 1D de instrucciones como atributo
def kernel_compute(array_comp, initial_instructions):
    x = cuda.threadIdx.x
    columna = cuda.blockDim.x * cuda.blockIdx.x + x

    # recorrer la matriz a mano, paralelizacion por cada step.
    # Se comprueba por cada step si se computa mas de una instruccion
    for i in range(1, array_comp.shape[0]):
        # En el anterior step no hay ningun host con 0 virus
        if(array_comp[i-1][0] != 0 and array_comp[i-1][1] != 0):
            if (columna < initial_instructions.shape[0]):
                array_comp[i][columna] = array_comp[i-1][columna] - 1
                 # sin pesos, voy a sumar la cantidad de instrucciones que se activen
                array_comp[i][2] = array_comp[i-1][2] + initial_instructions.shape[0]
                array_comp[i][3] = 2
        # En el anterior step los dos hosts tienen 0 virus
        elif(array_comp[i-1][0] == 0 and array_comp[i-1][1] == 0):
            sol = array_comp[i-1][2]
            if (columna < array_comp.shape[0] - i):
                array_comp[columna + i][2] = sol
                array_comp[columna + i][3] = -1
        else:
            # mono instruccion por step:
            if (array_comp[i-1][0] == 0 and array_comp[i-1][1] != 0):
                array_comp[i][1] = array_comp[i-1][1] - 1
                array_comp[i][2] = array_comp[i-1][2] + 1
                array_comp[i][3] = 1

            elif(array_comp[i-1][1] == 0 and array_comp[i-1][0] != 0):
                array_comp[i][0] = array_comp[i-1][0] - 1
                array_comp[i][2] = array_comp[i-1][2] + 1
                array_comp[i][3] = 0
```

Una vez está listo el kernel, debemos de invocarlo de igual manera que lo hicimos para *basicprobabilistic*. Como nos movemos en un $grid1D$ utilizamos una sola dimensión de hilos y bloques.

Listing 9: Invocación kernel SPSUMA

```
#  Creamos variables virus y numero de steps que queremos realizar
n1 = 100
n2 = 100
n_steps = 200
# Inicializamos arrays y damos primeros valores
initial_instructions = np.array([0, 1])
array_comp = np.zeros((n_steps,4), dtype=np.int64)
# La fila 0 representa el step 0
array_comp[0,0] = n1
array_comp[0,1] = n2

# Seleccionamos GPU, hilos y trasladamos arrays a GPU
cuda.select_device(1)
threads_per_block = 1024
blocks_per_grid2 = math.ceil(array_comp.shape[0] / threads_per_block)
d_array_comp = cuda.to_device(array_comp)
d_initial_instructions = cuda.to_device(initial_instructions)
# Invocamos kernel
kernel_compute[blocks_per_grid2, threads_per_block](d_array_comp, d_initial_instructions)
```

5.3.2. Solución para compilador JIT

Debido a las reglas que se suscribe el compilador JIT de Numba, la solución para este apartado es muy parecida para GPU. Solo cambiando el hecho de que no se calcula hilos o bloques por *grid*. De igual manera, no hace falta iterar por el ID de los hilos, ya que todo cálculo se hace de forma secuencial con bucles for.
Se itera por todo el array principal a partir de la segunda fila, donde se representa el primer *step* del simulador. Luego, se comprueba para la anterior fila si se ha llegado a alguna condición de parada o de uso de mono-instrucción. Justo igual que en el simulador paralelo.

Listing 10: Compilador JIT simulador spsuma

```
@njit
def spsuma(array_comp, initial_instructions):
    for i in range(1, array_comp.shape[0]):
        if(array_comp[i-1][0] != 0 and array_comp[i-1][1] != 0):
            for j in range(initial_instructions.shape[0]):
                array_comp[i][j] = array_comp[i-1][j] - 1
                array_comp[i][2] = array_comp[i-1][2] + initial_instructions.shape[0]
                array_comp[i][3] = 2
        elif(array_comp[i-1][0] == 0 and array_comp[i-1][1] == 0):
            sol = array_comp[i-1][2]
            for j in range(array_comp.shape[0] - i):
                array_comp[j + i][2] = sol
```

```
13                  array_comp[j + i][3] = -1
14          else:
15              # mono instruccion por step:
16              if (array_comp[i-1][0] == 0 and array_comp[i-1][1] != 0):
17                  array_comp[i][1] = array_comp[i-1][1] - 1
18                  array_comp[i][2] = array_comp[i-1][2] + 1
19                  array_comp[i][3] = 1
20
21              elif(array_comp[i-1][1] == 0 and array_comp[i-1][0] != 0):
22                  array_comp[i][0] = array_comp[i-1][0] - 1
23                  array_comp[i][2] = array_comp[i-1][2] + 1
24                  array_comp[i][3] = 0
25
26      return array_comp
```

6 Simulaciones y comparaciones: CPU vs GPU

Para las siguientes simulaciones se ha utilizado la NVIDIA RTX 2080 y NVIDIA RTX 3090 dentro del servidor de la facultad Mulhacen además de la última versión del CUDA toolkit 12.1, además del procesador INTEL CORE i7-9700K para todo el código relacionado con CPU (Python puro y JIT)

6.1. basicprobabilistics

Los kernels expuestos anteriormente tienen que ser ejecutados con una buena configuración de hilos y bloques. Dependiendo del problema o de la resolución, el número de hilos utilizado será diferente. Dado el propósito del problema, se utiliza el máximo número de hilos disponible para ambas máquinas ($(32, 32)$ en un total de 1024). Para $seed = 1$ en *create-xoroshiro128p-states*, 100 simulaciones, $n_{virus} = 1000$ y $p = 0{,}5$ el simulador en GPU obtiene los siguientes resultados:

```
basicprobabilistic: 100 simulaciones
virus: 1000
-----------------------------------------

Sim.: 1
HOST 1 | HOST 2 | HOST 3
0 | 498 | 502
++++++++++++++++++++++++++++++++++++++++

Sim.: 2
HOST 1 | HOST 2 | HOST 3
0 | 495 | 505
++++++++++++++++++++++++++++++++++++++++

Sim.: 3
HOST 1 | HOST 2 | HOST 3
0 | 492 | 508
++++++++++++++++++++++++++++++++++++++++

Sim.: 4
HOST 1 | HOST 2 | HOST 3
0 | 544 | 456
```

++++++++++++++++++++++++++++++++++

Sim . : 5

HOST 1 | HOST 2 | HOST 3

0 | 488 | 512

++++++++++++++++++++++++++++++++++

.

.

.

Sim . : 96

0 | 491 | 509

++++++++++++++++++++++++++++++++++

Sim . : 97

0 | 488 | 512

++++++++++++++++++++++++++++++++++

Sim . : 98

0 | 544 | 456

++++++++++++++++++++++++++++++++++

Sim . : 99

0 | 492 | 508

++++++++++++++++++++++++++++++++++

Sim . : 100

0 | 495 | 505

++++++++++++++++++++++++++++++++++

Para ver si el simulador es realmente eficaz, debemos de ver si la probabilidad acumulada coincide con nuestra probabilidad p. Como podemos observar en ambas gráficas [23] [24], la probabilidad en GPU no se normaliza tan pronto como en CPU, pero al terminar las 100 simulaciones obtienen casi el mismo resultado:

GPU

Sim : 100

PROB. ACUM. HOST 2 | PROB. ACUM. HOST 3

50.050505050505045 % | 49.94949494949495 %

++++++++++++++++++++++++++++++++++

CPU

Sim. 100

PROB. ACUM HOST 2 | PROB. ACUM. HOST 3

49.85 % | 50.15 %

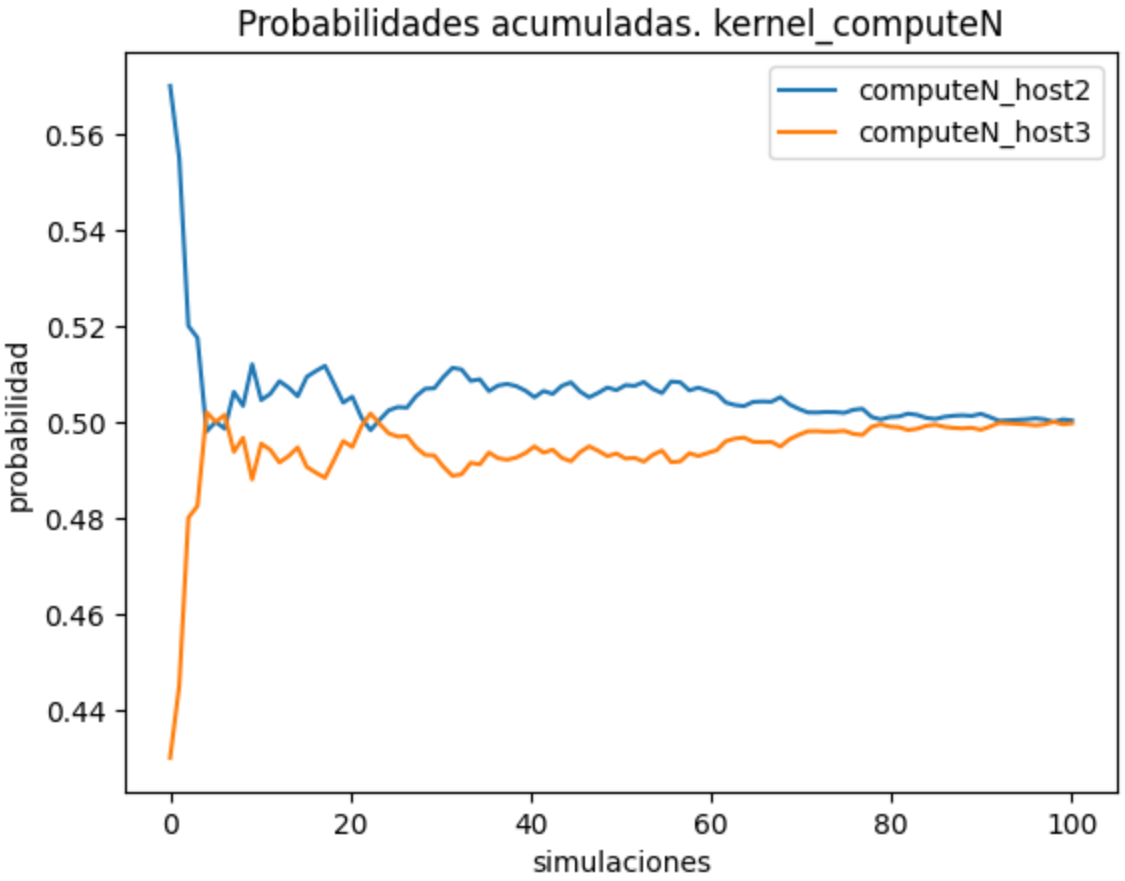

Figura 23: Probabilidades acumuladas GPU — p=0.5

6.1.1. Comparativa de tiempos

Para medir los tiempos, se ha hecho uso del módulo *timeit* de Python, el cual mide el tiempo de pequeños fragmentos de código — como por ejemplo en nuestro caso, la ejecución de un kernel— y cronometra la repetición de este código proporcionando el cálculo de tiempo más preciso posible y reduciendo el impacto de los costos de inicio o apagado en el cálculo de tiempo al ejecutar el código repetidamente. Esto incluye el tiempo de compilación de la primera llamada para el compilador JIT. Seguidamente, se ha proporcionado distintos tamaños de problema para ver cómo evoluciona el tiempo conforme este tamaño aumente. Definimos el tamaño de nuestro problema como la multiplicación del número de simulaciones por el número de virus, o lo que es lo mismo, el número de posiciones en la

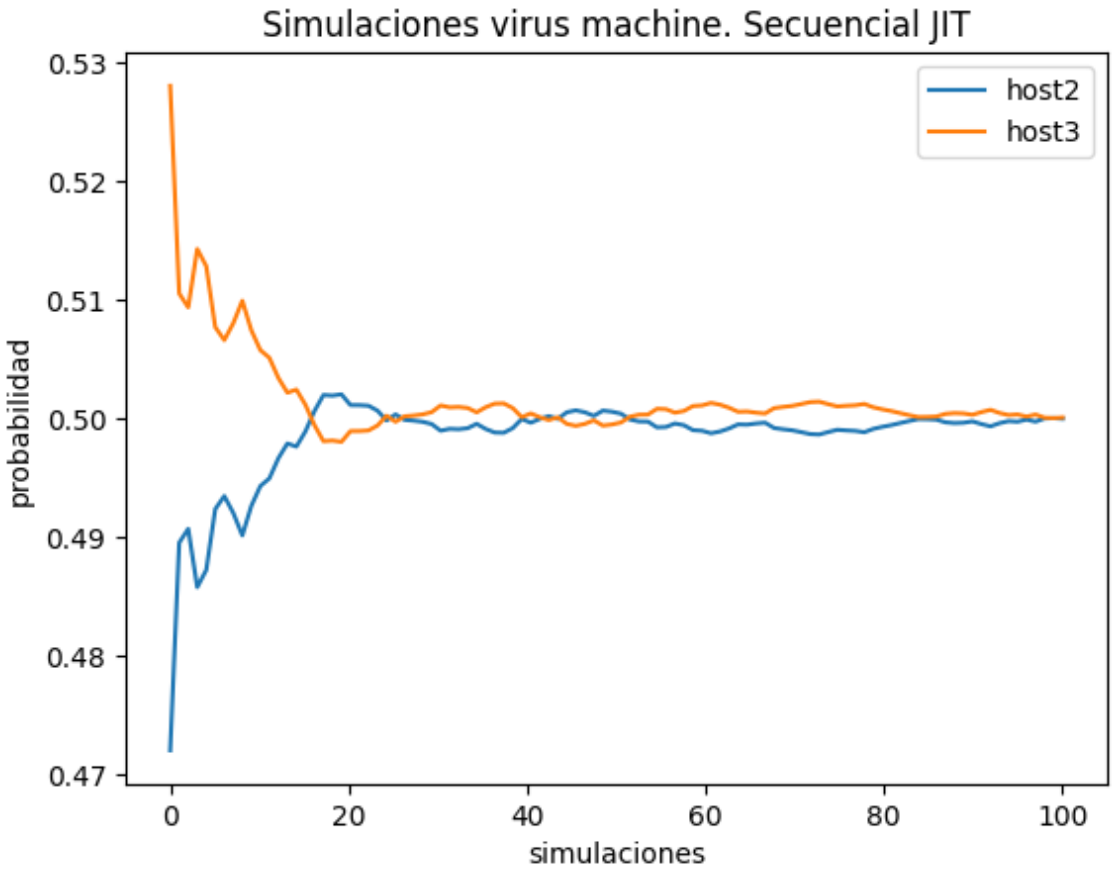

Figura 24: Probabilidades acumuladas CPU — p=0.5

matriz inicial. En la práctica, lanzar muchísimas simulaciones en un problema probabilístico no tiene sentido, sin embargo hay que recordar que nuestra idea aquí es estresar la GPU para ver su rendimiento.

Para empezar, en la figura [25] tenemos una comparativa inicial entre las dos GPU que se han utilizado. Se puede apreciar claramente que el tiempo aumenta de forma des-proporcional con el tamaño del problema a la vez que las dos RTX sirven el mismo patrón —siendo claramente la 3090 quien da mejores tiempos, debido a su arquitectura superior —.

Ya se explicó en el simulador paralelo como el cálculo de un correcto número aleatorio genera un cuello de botella (1 ms vs 20 ms en un tamaño de 100,000,000). Sin embargo, aún teniendo esta 'desventaja', se consigue una aceleración considerable con respecto al simulador secuencial. En la figura [26], en la que se ha usado una escala logarítmica en el eje Y, podemos ver la misma tendencia y curvatura del simulador secuencial pero con tiempos muy superiores. Para un tamaño de problema de 100,000,000 se consigue un **x45.33 de aceleración**, mientras que para un tamaño inferior de 4,000,000 un **x23.81** de aceleración. Aclarar que, como

el compilador JIT de Numba es secuencial por defecto, solo se usa un único núcleo de CPU.

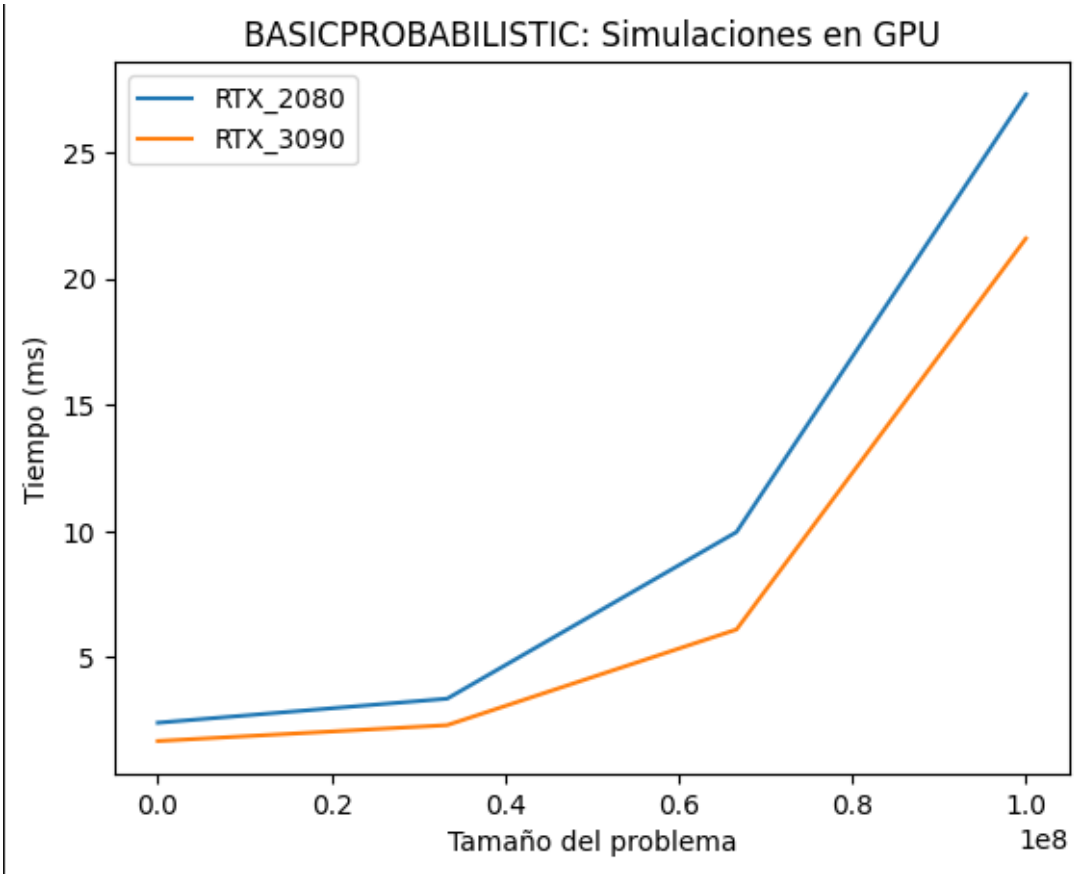

Figura 25: Comparativa de tiempos entre RTX 2080 y RTX 3090

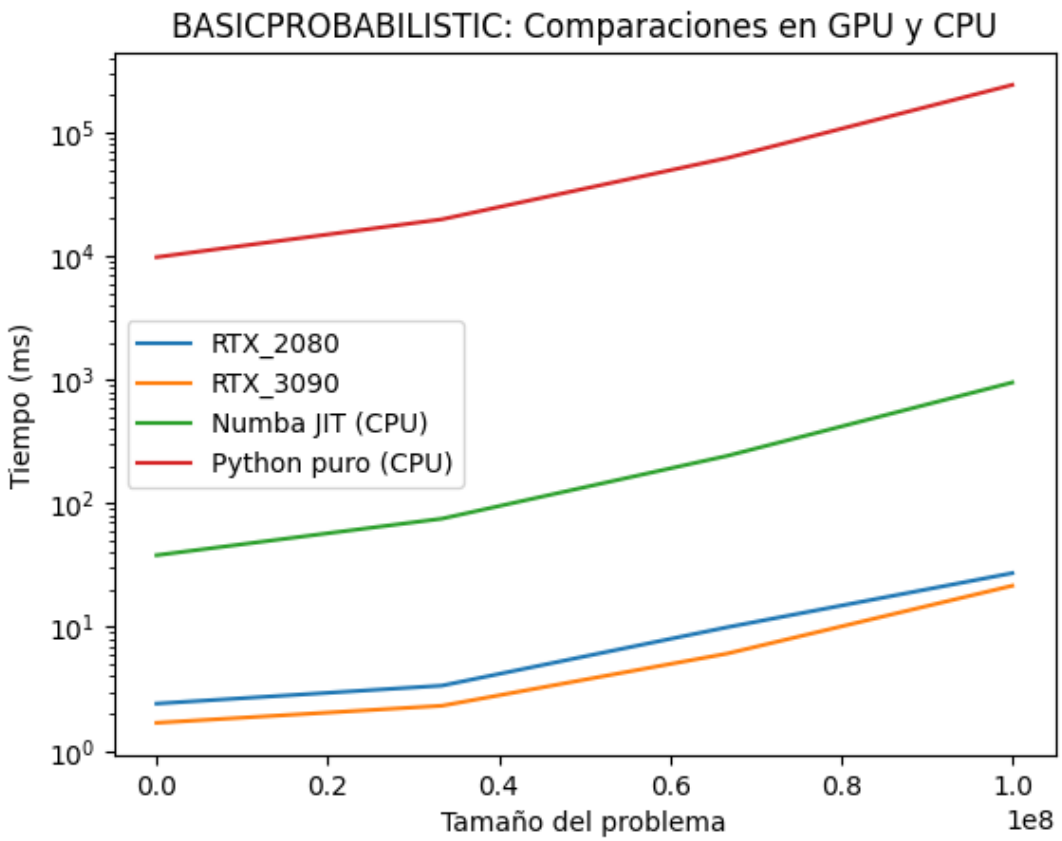

Figura 26: Comparativa de tiempos entre GPU y CPU. Escala logarítmica.

6.2. spsuma

Al ejecutar nuestro simulador paralelo del modelo spsuma, con $n1 = 5$ y $n2 = 4$, y $steps = 5$ obtenemos un resultado satisfactorio al igual que el del simulador secuencial:

```
sofparallel suma: 5 steps
--STEP 0--
n virus HOST 1: 5
n virus HOST 2: 4
Actividades en paralelo: (i1, i2)
--STEP 1--
Actividades en paralelo: (i1,i2)
| HOST 1 | HOST 2 | HOST 3 |
|    4   |    3   |    2   |
--STEP 2--
Actividades en paralelo: (i1,i2)
| HOST 1 | HOST 2 | HOST 3 |
```

```
|     3     |     2     |     4     |
--STEP 3--
Actividades en paralelo: (i1,i2)
| HOST 1 | HOST 2 | HOST 3 |
|     2     |     1     |     6     |
--STEP 4--
Actividades en paralelo: (i1,i2)
| HOST 1 | HOST 2 | HOST 3 |
|     1     |     0     |     8     |
--STEP 5--
Actividad: i1
| HOST 1 | HOST 2 | HOST 3 |
|     0     |     0     |     9     |
```

6.2.1. Comparativa de tiempos

Al igual que para *basicprobabilities*, se ha utilizado el módulo *timeit* para cronometrar los tiempos de todas las funciones relacionadas con *spsuma*. Así mismo, se proporcionan diferentes tipos de tamaño de problema, el cual definimos como la suma de $n1$ y $n2$ — nuestros virus de $h1$ y $h2$—. El simulador paralelo replica una significativa curva (ver figura [27]) en los tiempos de sus simulaciones en la que podemos ver claramente como el tiempo crece de forma no proporcional al tamaño del problema. Para el simulador JIT, el tiempo crece mucho más abruptamente, mientras que el simulador paralelo sigue el mismo promedio que el simulador estándar (ver figura [28]). Para tamaños de problema pequeños, el simulador JIT es indiscutiblemente el simulador que mejores tiempos, esto se debe a la forma tan iterativa que tiene en sí el simulador y el problema. Recordemos que, al tener que calcular cada *step* uno detrás de otro, el simulador se obligaba a tener cierta estructura secuencial que el compilador JIT puede sacar mucho provecho.
Por parte del simulador paralelo, las simulaciones en ambas GPUs tienen tiempos muy parecidos — siempre manteniendo el liderazgo de la 3090 frente a la 2080—. No se ha conseguido un decremento del tiempo con respecto al simulador JIT, pero a partir de un tamaño superior el simulador paralelo le toma hasta un **x9.329 de aceleración** para $n = 100{,}000$. Para el simulador estándar, se consigue un

x6.709 de aceleración con $n = 200$ y hasta **x2.497** de aceleración para $n = 100{,}000$.

La aceleración obtenida es imperceptible para tiempos tan pequeños (recordemos que hablamos de milisegundos o microsegundos en los tamaños más pequeños), además que la aceleración en sí no es tan grande como se puede esperar de la comparación entre CPU y GPU. Se achaca estos resultados a:

1. **Falta de rendimiento del kernel spsuma**. No se ha conseguido una maximización de la ejecución paralela debido a la naturaleza del modelo. Se podría haber ejecutado hilos concurrentes para calcular cada fila por paralelo, pero como ya se ha explicado, esto lleva a divergencia entre hilos debido a que se tiene que comprobar en todo momento cuantas instrucciones hay activas.

2. **Estructura del modelo spsuma**. En sí, el modelo *spsuma* es muy escueto. Solo son tres *hosts* y dos instrucciones que se activan paralelamente. Números tan pequeños hacen que la GPU trabaje sin estresarse demasiado — cosa que queremos que pase—. Como el kernel paraleliza por instrucciones, cuantas más instrucciones, mejor rendimiento tendrá.

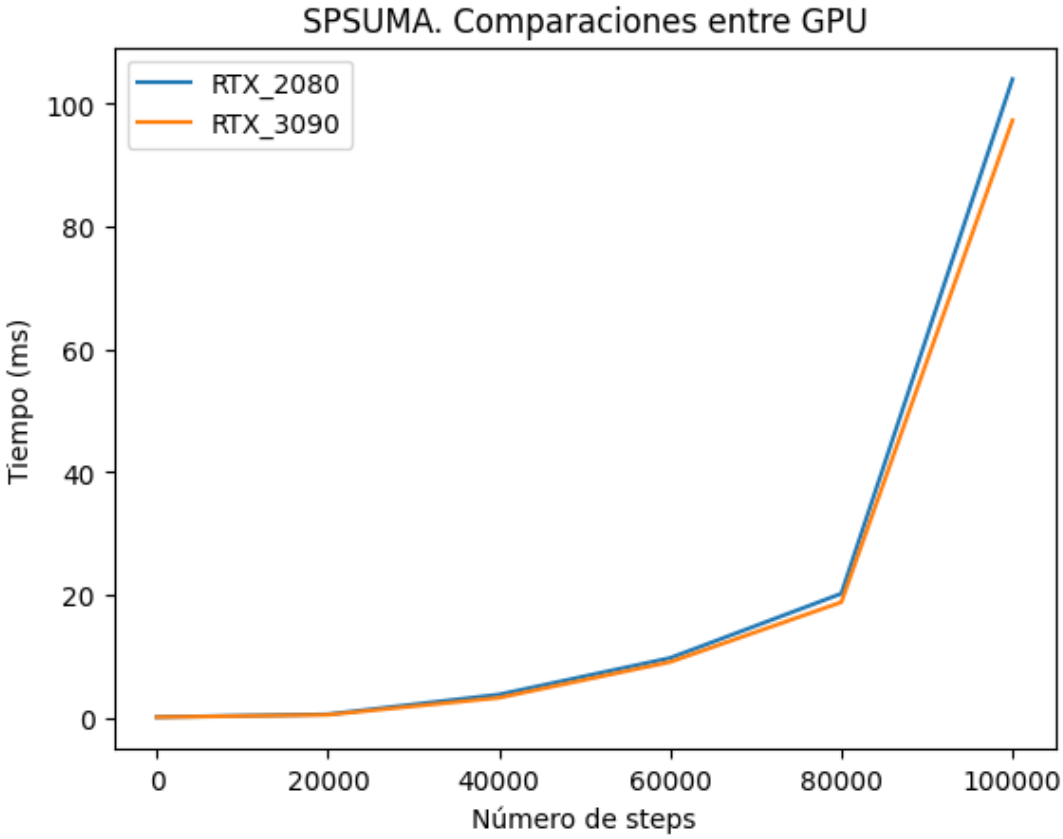

Figura 27: Spsuma. Comparativa de tiempos entre GPUs

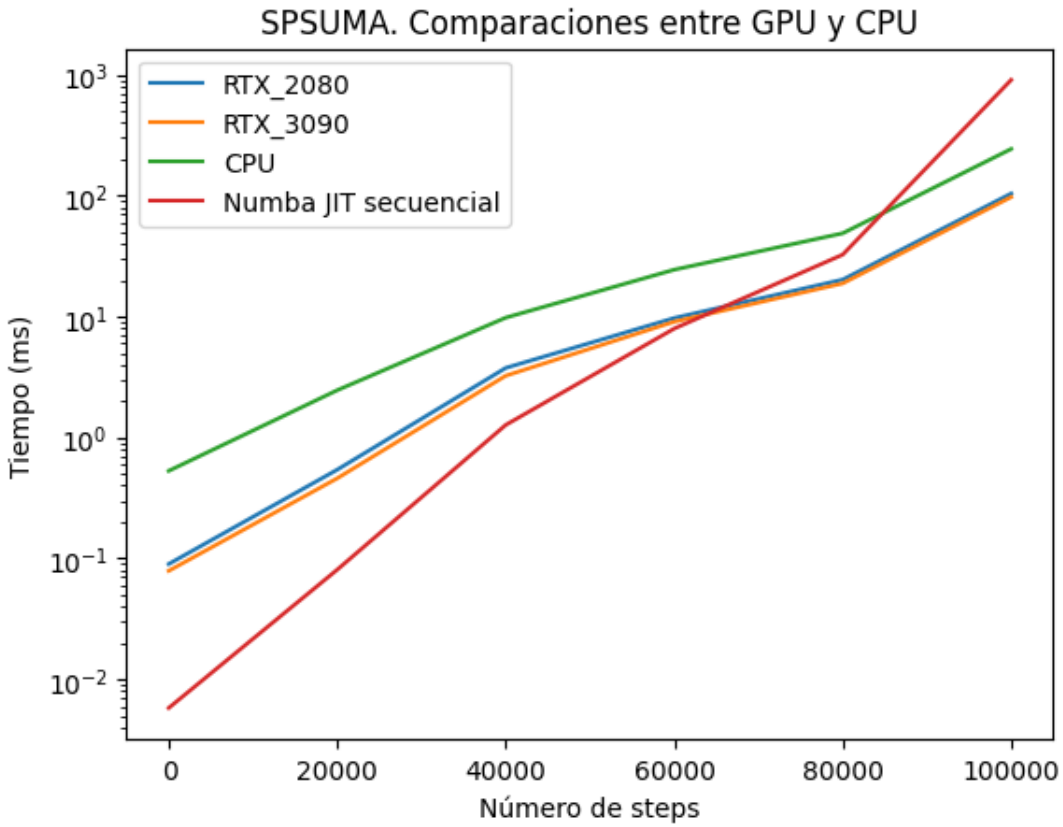

Figura 28: Spsuma. GPU vs CPU. El tiempo de JIT secuencial llega a peores tiempos debido al coste de su compilación en grandes tamaños.

7 Conclusiones

Al principio de este trabajo se propulsaron dos hilos concurrentes de conceptos: Por una parte la programación paralela en GPU con CUDA y por otra la definición de un modelo de computación reciente que es la máquina de virus. Con la puesta en escena del simulador secuencial, se puso un espacio para juntar ambos mundos y se planteó el desarrollo de un simulador paralelo ad-hoc para algunos de los modelos de la máquina de virus.
Por una parte, tenemos el simulador de múltiples computaciones del modelo *basicprobabilistic* el cual pensamos que ha tenido un desarrollo perfecto en su parte paralela, logrando un buen resultado en el cómputo de probabilidades. Esto se debe a que sus requisitos casaban muy bien con las necesidades de operar en una GPU:

- Al ser simulaciones independientes y por el intrínseco funcionamiento del modelo, cada fila y columna de la matriz inicial se podía operar sin riesgos a convergencias o sincronización de hilos, dando pie a una buena oportunidad de estresar la GPU con grandes dimensiones de matriz.

- Aunque el generador de números aleatorios suponía un cuello de botella en el rendimiento del simulador paralelo, sigue proporcionando una aceleración con respecto a las contrapartidas secuenciales.

Seguidamente, se ha conseguido desarrollar el simulador paralelo del modelo *spsuma*. Este simulador es 'equivalente' a su versión estándar debido a que itera todos los *steps* uno a uno y los guarda en un *array*. Sigue siendo un simulador paralelo no genérico, es decir, que su funcionamiento es específico para el modelo *spsuma*. Los resultados obtenidos son tan eficaces como en el simulador estándar, aunque el gran "pero" viene con su rendimiento debido que, al contrario que *basicprobabilistic*, los requisitos del problema no se pudieron adaptar para operar en GPU:

- Cada *step* debía conocer el estado del *step* anterior para saber qué conjunto de instrucciones operar. Esta condición puede hacer converger los hilos si no se manejan adecuadamente.

- Nuestro kernel paraleliza en función del set de instrucciones. Cuantas más instrucciones activas al mismo tiempo, mayor ocupación de la GPU. En sí, el modelo *spsuma* reina por su simpleza en cuanto *hosts* e instrucciones.

En la siguiente sección se dará más a conocer los detalles que se han encontrado en el desarrollo de nuestro simulador paralelo.

7.1. Piedras en el camino

Los retos que se han conocido a la hora del desarrollo de este trabajo no son pocos ni perogrullos. Rescatamos la idea de que este trabajo se asienta en dos campos de conceptos —de nuevo, CUDA y máquina de virus— inicialmente desconocidos al comienzo del trabajo.

El desarrollo en CUDA se apoyó sobretodo en la cantidad de información y talleres que ofrece NVIDIA por internet [NVI15]. Aunque para problemas concretos, se llegaba a quedar corto de referencias debido a nuestro uso de Numba CUDA — C-CUDA es mucho más famoso que Numba-CUDA y, por lo tanto, abarca más en la red—. El desarrollo en sí estuvo centrado en simular máquinas de virus, que sigue siendo una línea de modelos de computación muy nueva y con ejemplos pequeños.

La primera 'gran piedra de nuestro camino' fue adaptar el simulador genérico para que trabajase en GPU. Ya se ha expuesto que la GPU no puede trabajar con código de alto nivel como son las clases o incluso listas o diccionarios, por lo que el desarrollo de cada simulador tenía que tener su propia configuración de matrices y vectores de entrada.

Pero el mayor reto a la hora de simular máquinas de virus paralelas es conseguir la mayor aceleración posible con respecto al simulador estándar. Trabajar con CUDA abre una ventana a la "infinita optimización", término que acuñamos a la ilusión de cambios interminables que puedes hacer en el código para llegar a escasas décimas de aceleración y, a veces, lleva a la pérdida de la eficacia de la función.

Por ejemplo, en un intento de conseguir más aceleración para el simulador paralelo de *spsuma* se llegó a cambiar la estructura de las matrices, cómo operaba el *kernel* e incluso añadir $device functions$ de Numba o diferentes configuraciones de invocación del kernel para llegar a nuestro objetivo. Al final, se perdía el rumbo y el simulador dejaba de ser un simulador fiel a la idea de una máquina de virus —aunque se ejecutase con muchísima más aceleración—. Esto mismo lleva a una

pérdida de visión por parte del desarrollo. Es un objetivo conseguir aceleración con respecto al simulador estándar, pero antes hay que conseguir un simulador fiel a la idea de máquina de virus.
Como ya hemos mencionado, las máquinas de virus es una línea nueva como modelo computacional, pero se dan los primeros pasos para buscar una escalabilidad con GPUs.

7.2. ¿Tiene CUDA Numba cabida en el GPGPU?

Al elegir CUDA como la herramienta de paralelización y modelo de programación para nuestros simuladores paralelos nos abrió dos caminos a seguir: Podíamos escribir dicho simulador en C/C++, lenguaje en el que trabaja CUDA y luego conectar dicho código con Python mediante el uso de librerías externas o probar el módulo Numba para Python y escribir directamente código que se pudiera mantener más fácilmente y así ahorrarnos el uso de susodichas librerías externas.
En el capítulo 2 se mencionó como CUDA Numba no tenía nada que envidiar a C-CUDA en cuanto a rendimiento, aunque si el único propósito de tu problema es conseguir aceleración **quizás lo más razonable sea programar en C/C++**.
Ventajas que podemos señalar a la hora de utilizar CUDA Numba:

1. Sencillez en su uso y sintaxis.
2. Se evita usar uniones C-Python como Pybind o CPython, haciendo el código mucho más fácil de mantener al estar todo en un solo lenguaje
3. Gracias a que se apoya en Python, se puede ejecutar código en Notebooks en los cuales las simulaciones y experimentos son más legibles
4. Rendimiento compite con el análogo C-CUDA
5. Posee un extenso manual de usuario que se complementa a la perfección con el de CUDA.
6. Es 100 % *opensource*

Del mismo modo, se pueden indicar los siguientes inconvenientes de su uso:

1. Por el momento, no adopta todas las funcionalidades de CUDA como, por ejemplo, el lanzamiento dinámico de kernels.

2. Su *troubleshooting* puede llegar a ser confuso de implementar o interpretar.

Como resumen, creemos que CUDA Numba es una excelente herramienta en el uso de GPGPU además de ofrecer una alternativa viable a C-CUDA a aquellos desarrolladores que quieran mantenerse la sencillez y mantenimiento de su código. Al fin al cabo, Python es un lenguaje en alza para el desarrollo de paradigmas como la inteligencia artificial [ZS23], campo que es fácilmente aprovechable con el uso de las GPU y ahí es donde CUDA Numba puede entrar en escena.

7.3. El futuro

Echamos la vista al mañana y nos avista un futuro lleno de posibilidades a realizar. Durante la previa al desarrollo de este trabajo — e incluso a veces durante el mismo— algunas de estas ideas para el futuro se han planteado para ser realizadas. Por motivos de alcance o incompatibilidad entre ellas, se ha escogido lo que tenemos en nuestro presente.

Para empezar, con respecto a lo desarrollado a este mismo trabajo, se podría seguir con la 'tarea infinita' de seguir optimizando los *kernels* definidos para el simulador ad-hoc.

Seguidamente, se podría realizar la misma premisa de este trabajo pero probando CUDA-C para ver el rendimiento de este y de verdad comprobar que es el rey del rendimiento a la vez que conectar el código a Python con el simulador estándar.

Establecer simuladores paralelos ad-hoc está bien para probar conceptos y buscar cómo CUDA es capaz de acelerar código. Pero al iniciar la carrera de simuladores paralelos de máquinas de virus, lo interesante para el futuro sería ver un simulador genérico completamente paralelo. Este simulador se tendría que servir de vectores y matrices de entrada que representen distintos modelos, los cuales tienen que ser computados por unas funciones — o kernels, más bien— genéricas en GPU.

Y finalmente, la versión $softparallel$ de la máquina de virus nos acercaba a una versión intrínsecamente paralela de este modelo de computación. En un futuro se podría explorar en concreto esta rama y, junto el desarrollo de un simulador genérico paralelo, establecer un simulador para esta versión. Quizás con unos modelos más extensos y complejos que den rienda suelta al poder latente de ejecutar múltiples instrucciones por *step*.

Referencias

[Ana] Continuum Analytics. Numba documendation. https://numba.pydata.org/numba-doc/dev/index.html.

[Cro] Louis Croce. Just in time compilation. *Columbia University. URL: http://www. cs. columbia. edu/˜ aho/cs6998/Lectures/14-09-22_ Croce_ JIT. pdf (Data obrashhenija: 10.03. 2021)*.

[DKK21] William J Dally, Stephen W Keckler, and David B Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

[Gar23] Sergio Velázquez García. Repositorio tfg. https://github.com/Servelgar/TFG-NUMBA-CUDA, 2023.

[GPKB12] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[Hag15] James Hague. "why do dedicated game consoles exist?". https://web.archive.org/web/20150504042057/https://prog21.dadgum.com/181.html, 2015.

[HKAA11] Laiq Hasan, Marijn Kentie, and Zaid Al-Ars. Dopa: Gpu-based protein alignment using database and memory access optimizations. *BMC research notes*, 4(1):1–11, 2011.

[Mac03] M. Macedonia. The gpu enters computing's mainstream. *Computer*, 36(10):106–108, 2003.

[Mar23] David Orellana Martín. Repositorio virus machine. https://github.com/RGNC/virusmachines, 2023.

[McC10] Chris McClanahan. History and evolution of gpu architecture. 2010.

[MdAOMPH+19] Miguel Á Martínez-del Amor, David Orellana-Martín, Ignacio Pérez-Hurtado, Luis Valencia-Cabrera, Agustín Riscos-Núñez,

and Mario J Pérez-Jiménez. Design of specific p systems simulators on gpus. In *Membrane Computing: 19th International Conference, CMC 2018, Dresden, Germany, September 4–7, 2018, Revised Selected Papers 19*, pages 202–207. Springer, 2019.

[MDCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[NF15] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[NVF23a] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html, 2023.

[NVF23b] NVIDIA, Péter Vingelmann, and Frank H.P Fitzek. Cuda c++ programming guide, performance guidelines. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#performance-guidelines, 2023.

[NVI] NVIDIA. ¿qué es la computación acelerada por gpu. https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/.

[NVI15] NVIDIA. "nvidia self-paced online courses. fundamentals of accelerated computing with cuda python.". https://courses.nvidia.com/courses/course-v1:DLI+C-AC-02+V1/, 2015.

[Ode20] Lena Oden. Lessons learned from comparing c-cuda and python-numba for gpu-computing. In *2020 28th Euromicro international conference on parallel, distributed and network-based processing (PDP)*, pages 216–223. IEEE, 2020.

[OMRdAPJ23] David Orellana-Martín, Antonio Ramírez-de Arellano, and Mario J Pérez-Jiménez. Simulating and validating virus machines. 2023.

[Pat23] Dylan Patel. How nvidia's cuda monopoly in machine learning is breaking. https://www.semianalysis.com/p/nvidiaopenaitritonpytorch, 2023.

[RdAOMPJ23] Antonio Ramírez-de Arellano, David Orellana-Martín, and Mario J. Pérez-Jiménez. Using virus machines to compute pairing functions. *International Journal of Neural Systems*, 33(05):2350023, 2023. PMID: 36967221.

[RJVCPJ15] Álvaro Romero-Jiménez, Luis Valencia-Cabrera, and Mario J Pérez-Jiménez. Generating diophantine sets by virus machines. In *Bio-Inspired Computing–Theories and Applications: 10th International Conference, BIC-TA 2015 Hefei, China, September 25-28, 2015, Proceedings 10*, pages 331–341. Springer, 2015.

[RJVCRNPJ15] Álvaro Romero-Jiménez, Luis Valencia-Cabrera, Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez. Computing partial recursive functions by virus machines. In Grzegorz Rozenberg, Arto Salomaa, José M. Sempere, and Claudio Zandron, editors, *Membrane Computing*, pages 353–368, Cham, 2015. Springer International Publishing.

[SGFV19] VV Sanzharov, AI Gorbonosov, VA Frolov, and A Voloboy. Examination of the nvidia rtx. In *CEUR Workshop Proceedings*, volume 2485, pages 7–12, 2019.

[Sha20] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[VCPJC+15] Luis Valencia-Cabrera, Mario J Pérez-Jiménez, Xu Chen, Beizhan Wang, and Xiangxiang Zeng. Basic virus machines. In *16th International Conference on Membrane Computing (CMC16)*, pages 323–342, 2015.

[Wik] Wikipedia. Wikipedia. https://en.wikipedia.org/wiki/GeForce_6_series.

[ZS23] Ravshanbek Zulunov and Bakhromjon Soliev. Importance of python language in development of artificial intelligence. *Потомки Аль-Фарғани*, 1(1):7–12, 2023.