Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Servidor con Arquitectura CUDA en Python

Autor: Manuel Redondo Alonso
Tutor/es: José María Gutiérrez Martínez

2017/2018

UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Servidor con Arquitectura CUDA en Python

Autor: Manuel Redondo Alonso
Tutor/es: José María Gutiérrez Martínez

Tribunal:

Presidente: .....

Vocal 1: .....

Vocal 2: .....

Calificación: .....

Fecha: .....

Palabras Clave

Socket, CUDA, Python, Computación Paralela, GPU.

Key Words

Socket, CUDA, Python, Parallel Computing, GPU.

Resumen Corto

Este proyecto da una solución al problema de ejecutar código CUDA en equipos que no cuentan con una arquitectura adecuada para hacerlo, creando un servidor remoto donde cualquier cliente puede conectarse para ejecutar el código como si fuese en su propia máquina. Todo esto se lleva a cabo empleando Python como lenguaje de programación y Sockets como tecnología para crear la arquitectura cliente-servidor.

Short Summary

This project provides a solution to the problem of execute CUDA source code in computers that do not have an adequate architecture to do it, creating a remote server where clients can connect for execute the code like in their own computer. The project is made using Python as a programming language and Sockets as a technology to create the client-server architecture.

Índice Resumido

Índice Detallado

1 Introducción

En la actualidad millones de desarrolladores utilizan una gran variedad de lenguajes de programación, algunos de estos lenguajes requieren de una potencia o arquitectura específicos para su desarrollo y posterior uso.

Un problema concreto, el cual se pretende paliar con este proyecto, es el de desarrollar y ejecutar aplicaciones aceleradas en la GPU en máquinas que no cuentan con la arquitectura adecuada para hacerlo, por ejemplo, en ordenadores que no tienen instalada una tarjeta gráfica. Este problema surge en escuelas y universidades, donde los alumnos tienen que aprender diferentes tecnologías y donde tener un equipo adecuado para utilizar todas ellas cada vez es más complicado. También surgen problemas similares cuando existe una necesidad de trabajar en un equipo donde cada integrante cuenta con arquitecturas, e incluso sistemas operativos, diferentes.

La solución que se suele emplear habitualmente es utilizar algún servicio de Cloud Computing, que cuente con la arquitectura necesaria para ejecutar este tipo de aplicaciones. Aunque esto sea una solución aceptable, estos servicios suelen ser bastante costosos y no siempre son los más adecuados.

Las posibilidades de una arquitectura cliente-servidor propia, es decir, sin subcontratar el servidor a alguna compañía de Cloud Computing, es una buena solución para este problema. Aunque inicialmente el coste es mayor, y quizá no es el adecuado para grandes empresas con muchos usuarios y requerimientos de potencia, en un ámbito universitario como el nuestro sí parece la mejor opción. Permite la centralización del control a los accesos, los recursos y la integridad de los datos gracias a que estos son controlados por el servidor de forma que un programa cliente defectuoso o no autorizado no pueda dañar el sistema. Esta centralización también facilita la tarea de poner al día datos u otros recursos. Facilita la escalabilidad tanto de clientes como de servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores). Al estar distribuidas las funciones y responsabilidades entre varios ordenadores independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio (o se afectarán mínimamente). Esta independencia de los cambios también se conoce como encapsulación. Y por último, pero no por ello menos importante, existen tecnologías suficientemente desarrolladas y diseñadas para el paradigma de C/S que aseguran la seguridad en las transacciones, la amigabilidad de la interfaz, y la facilidad de empleo.

A parte de todo lo explicado anteriormente y especialmente cuando trabajamos en grupo desarrollando aplicaciones aceleradas en la GPU, realizar todo el desarrollo con la misma GPU (la instalada en el servidor) es muy importante, pues para realizar una correcta optimización de cualquier proceso es necesario conocer información como el número de núcleos de la tarjeta gráfica o la estructura de los mismos dentro de la tarjeta. Incluso evita posibles problemas de incompatibilidades, pues todos los integrantes del equipo estarían trabajando con la misma versión de las aplicaciones y con los mismos drivers para los dispositivos.

Personalmente, este proyecto supone una oportunidad de poner en práctica los conocimientos que he aprendido en estos últimos cuatro años de carrera. Concretamente aquellos conocimientos relacionados con la gestión de proyectos, con la computación paralela CUDA y con la arquitectura cliente-servidor. Al igual que permite poner mis conocimientos en práctica, también me ha servido para extraer algunos nuevos, los cuales otorgan una nueva perspectiva de lo que las nuevas tecnologías pueden hacer para cambiar o mejorar el trabajo de miles de desarrolladores en todo el mundo.

2 Objetivo

El objetivo principal de este proyecto de fin de grado consiste en desarrollar una aplicación que permita a los clientes ejecutar código CUDA de manera remota en un servidor remoto con GPU y que puedan visualizar el resultado de esta ejecución en su propia máquina. Al tratarse de una aplicación que va a ser utilizada en el ámbito universitario se va a requerir un rendimiento suficiente para este propósito, pero con un coste asequible.

Se desarrollarán las siguientes funcionalidades:

- Los alumnos deben poder ejecutar el programa cliente directamente en su máquina, sea cual sea su sistema operativo.
- Los clientes deben poder enviar archivos con extensión Python al servidor.
- El servidor debe poder atender y ejecutar las peticiones de varios clientes simultáneamente.
- El servidor debe poder ejecutar programas Python con CUDA y capturar el resultado de la ejecución de dichos programas.
- El servidor debe poder enviar a cada cliente, la salida correspondiente al programa que ha enviado para que el este lo ejecute.
- Los clientes deben poder recibir la salida del programa, que ha enviado previamente el servidor, y mostrarla en el equipo del alumno.

3 Estado del Arte

En este apartado profundizaremos más en el campo de las aplicaciones cliente-servidor, de la computación paralela CUDA y del lenguaje Python. Estos 3 campos son la base principal del posterior desarrollo que haremos para implementar nuestra aplicación. También expondremos una serie de proyectos en los cuales se están aplicando estas tecnologías actualmente, para poder ver realmente el alcance de estas tecnologías.

3.1 La Arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo de diseño de software en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta. Esta idea también se puede aplicar a programas que se ejecutan sobre una sola computadora, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un sólo programa. Los tipos específicos de servidores incluyen los servidores web, los servidores de archivo, los servidores del correo, etc. Mientras que sus propósitos varían de unos servicios a otros, la arquitectura básica seguirá siendo la misma.

La red cliente-servidor es una red de comunicaciones en la cual los clientes están conectados a un servidor, en el que se centralizan los diversos recursos y aplicaciones con que se cuenta; y que los pone a disposición de los clientes cada vez que estos son solicitados. Esto significa que todas las gestiones que se realizan se concentran en el servidor, de manera que en él se disponen los requerimientos provenientes de los clientes que tienen prioridad, los archivos que son de uso público y los que son de uso restringido, los archivos que son de sólo lectura y los que, por el contrario, pueden ser modificados, etc.

3.2 ¿Qué es CUDA?

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.

Gracias a millones de GPU's CUDA vendidas hasta la fecha, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos, la reconstrucción de imágenes de TC, el análisis sísmico o el trazado de rayos, entre otras.

3.3 El Procesamiento Paralelo con CUDA

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU a realizar "coprocesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

En el mercado de consumo, prácticamente todas las aplicaciones de vídeo se han acelerado, o pronto se acelerarán, a través de CUDA, como demuestran diferentes productos de Elemental Technologies, MotionDSP y LoiLo, Inc.

CUDA ha sido recibida con entusiasmo por la comunidad científica. Por ejemplo, se está utilizando para acelerar AMBER, un simulador de dinámica molecular empleado por más de 60.000 investigadores del ámbito académico y farmacéutico de todo el mundo para acelerar el descubrimiento de nuevos medicamentos.

En el mercado financiero, Numerix y CompatibL introdujeron soporte de CUDA para una nueva aplicación de cálculo de riesgo de contraparte y, como resultado, se ha multiplicado por 18 la velocidad de la aplicación. Cerca de 400 instituciones financieras utilizan Numerix en la actualidad.

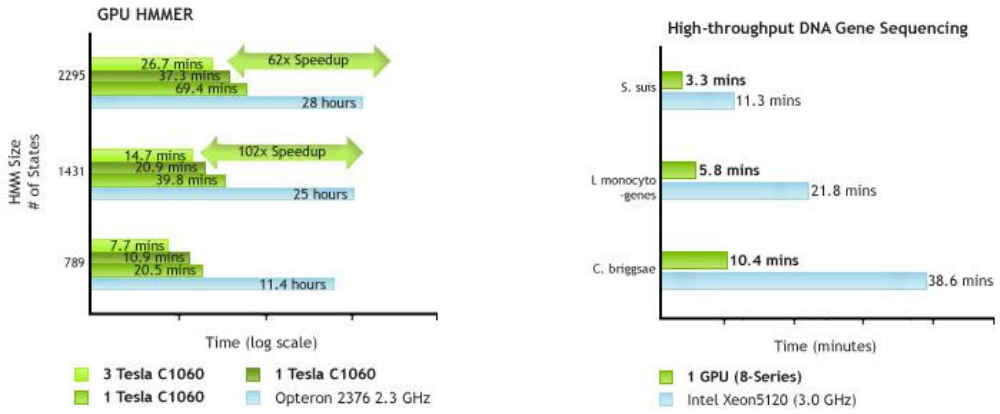
Un buen indicador de la excelente acogida de CUDA es la rápida adopción de la GPU Tesla para aplicaciones de GPU Computing. En la actualidad existen más de 700 clusters de GPUs instalados en compañías Fortune 500 de todo el mundo, lo que incluye empresas como Schlumberger y Chevron en el sector energético o BNP Pariba en el sector bancario.

Por otra parte, la reciente llegada de los últimos sistemas operativos de Microsoft y Apple (Windows 10 y OS X Lion) está convirtiendo el GPU Computing en una tecnología de uso masivo. En estos nuevos sistemas, la GPU no actúa únicamente como procesador gráfico, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

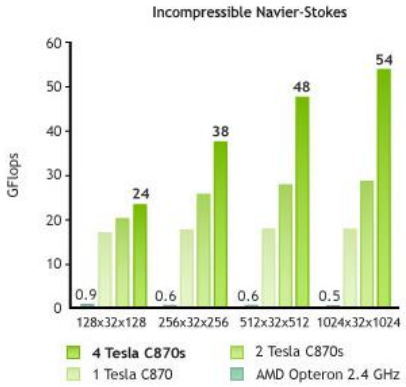
3.4 Beneficios y Aplicaciones del Procesamiento Paralelo usando CUDA

Dadas las características del procesamiento paralelo puede llegar a ser complicado entender la posible utilidad del mismo. Se listan las aplicaciones más prominentes del procesamiento paralelo:

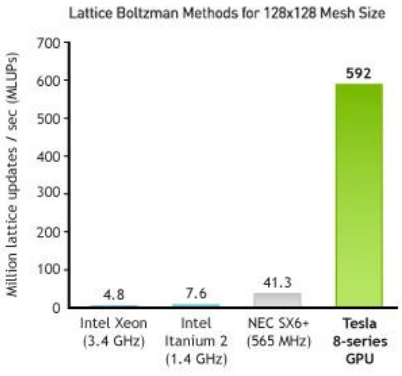
- Bioinformática: Para la secuencia y el acoplamiento de proteínas se requieren procesos informáticos muy intensos, por lo que es muy ventajoso el uso de GPU compatibles con CUDA. En la actualidad ya se utilizan GPU para el trabajo con códigos de bioinformática y ciencias de la vida.



- Dinámica de Fluidos Computacional: Diferentes proyectos basados en los modelos de Navier-Stokes y los métodos de Boltzman han experimentado importantes mejoras de rendimiento mediante el uso de GPUs con CUDA. Estos resultados se muestran a continuación con varios gráficos. Existen otros trabajos sobre modelos climáticos y oceánicos también basados en el uso de la GPU.

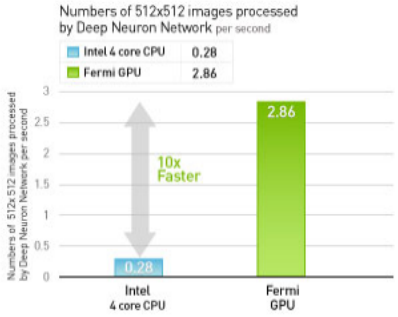


Ecuaciones Navier-Stokes para fluidos incomprensibles
Thibault y Senocak

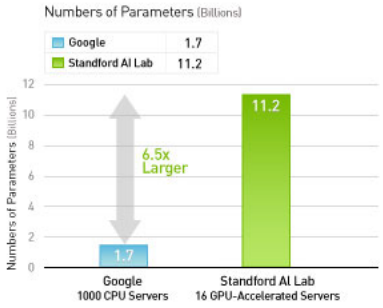


Métodos de Boltzmann en retículo
Tolke y Krafczyk

- Ciencia de los Datos, Analítica y Bases de Datos: Cada vez hay más clientes que recurren a las GPU para analizar cantidades masivas de datos (Big Data) y mejorar así la toma de decisiones de negocio en tiempo real.

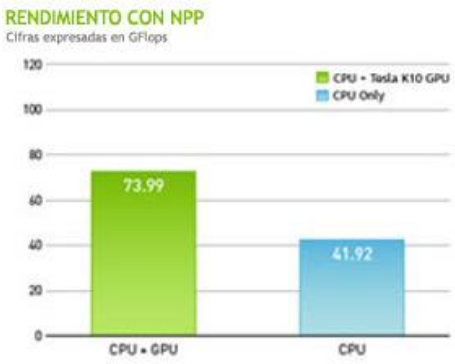


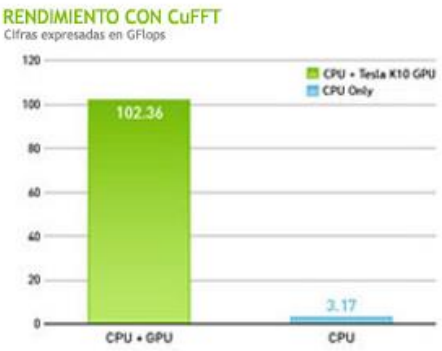
10 VECES MÁS VELOCIDAD DE DETECCIÓN DE LAS IMÁGENES CON LAS REDES NEURONALES
Dr. Dan Ciresan, Laboratorio de IA IDSIA, Suiza



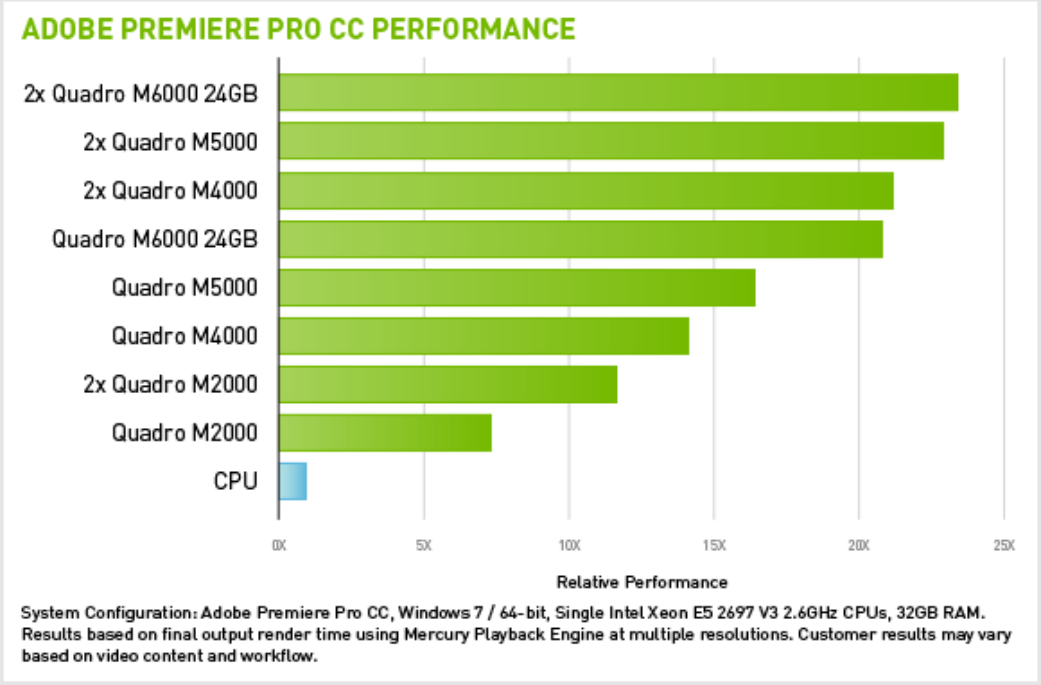
LAS REDES NEURONALES ARTIFICIALES MÁS GRANDES DEL MUNDO SE CREAN CON GPU
Adam Cotes et al, Laboratorio de IA de Sanford, EE.UU. - MÁS INFORMACIÓN

- Defensa e Inteligencia: Los organismos y empresas que trabajan en el sector de la defensa y los servicios de inteligencia necesitan disponer de información precisa y puntual tanto en sus operaciones estratégicas como diarias. Una vez recabados los datos, convertirlos en información de utilidad requiere una cantidad de recursos considerable en términos de personas, sistemas de hardware y software, energía e instalaciones, pero no siempre están disponibles. Las tarjetas gráficas NVIDIA proporcionan una tecnología que "cambia las reglas del juego" e incrementa drásticamente la productividad, al tiempo que reduce los costes, el consumo energético y la necesidad de instalaciones.

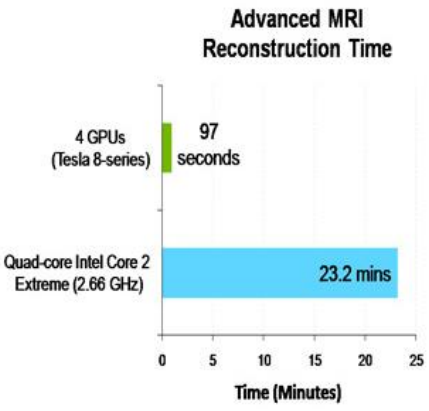




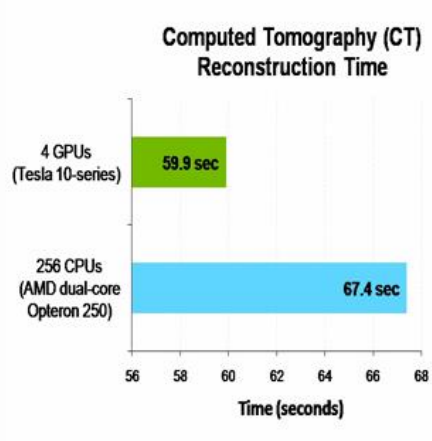
- Medios Audiovisuales y Entretenimiento: Los profesionales del vídeo digital siempre buscan formas más fáciles y rápidas de obtener los mejores resultados. Aplicaciones habituales en este sector como Adobe Premiere Pro, Autodesk 3ds Max y Maya, Assimilate SCRATCH o DaVinci Resolve permiten combinar múltiples capas de efectos digitales que incluyen simulaciones de los tejidos, el cabello o el agua, así como análisis avanzado de las imágenes y codificación de vídeo. Tecnología multi-GPU responde a las necesidades de este sector combinando la potencia de múltiples GPUs para proporcionar al mismo tiempo alto rendimiento gráfico y procesamiento paralelo.



- Imágenes Médicas: La generación de imágenes médicas es una de las primeras aplicaciones para las que se recurrió a la aceleración mediante GPU. El uso de las GPU se ha hecho tan frecuente en dicho campo que existen varios sistemas médicos que se suministran con GPU Tesla de NVIDIA.

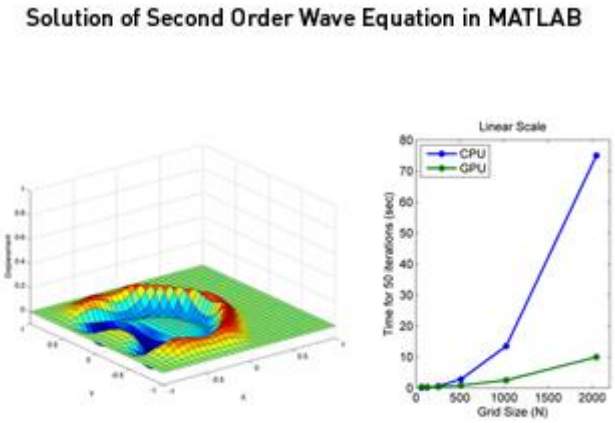


Reconstrucciones de IRM avanzadas aceleradas por GPU
Stone, et al

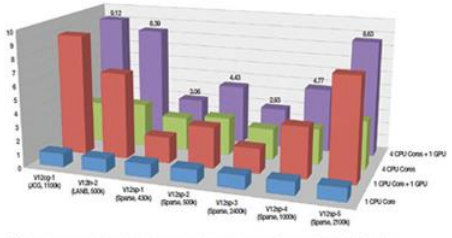


Reconstrucción de tomografía computarizada
Batenburg, Sijbers, et al

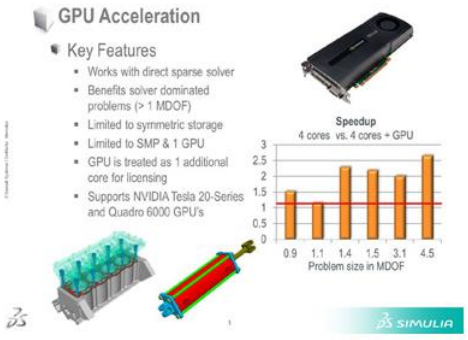
- Análisis Numérico: MATLAB®, Mathematica y LabView se benefician considerablemente del uso de GPUs basadas en CUDA.



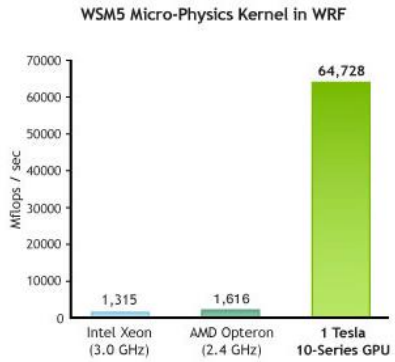
- Mecánica Estructural Computacional: ANSYS, Abaqus, MSC Nastran, IMPETUS Afea y otras aplicaciones de mecánica estructural obtienen importantes ventajas con el uso de las GPUs CUDA.

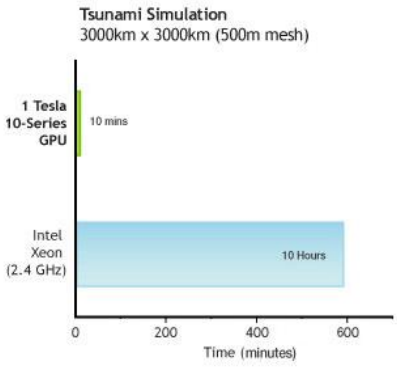


GPU accelerator capability of solver kernel speedups using a prerelease version of ANSYS 13.0



- Modelos Meteorológicos y Climáticos: Las aplicaciones de mecánica de fluidos para el cálculo de modelos climatológicos y oceánicos, como el modelo WRF (Weather Research and Forecasting model), y las simulaciones de maremotos han experimentado avances extraordinarios que permiten acelerar su ejecución y sus niveles de precisión.





3.5 ¿Por qué Python?

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores. Python es un lenguaje de programación multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones.

Una característica importante de Python es la resolución dinámica de nombres; es decir, lo que enlaza un método y un nombre de variable durante la ejecución del programa (también llamado enlace dinámico de métodos). Otro objetivo del diseño del lenguaje es la facilidad de extensión. Se pueden escribir nuevos módulos fácilmente en C o C++. Python puede incluirse en aplicaciones que necesitan una interfaz programable.

Aunque la programación en Python podría considerarse hostil a la programación funcional tradicional del Lisp, existen bastantes analogías entre Python y los lenguajes minimalistas de la familia Lisp como puede ser Scheme.

Sin embargo, el principal motivo para haber elegido este lenguaje y no otro es, que aunque históricamente se ha considerado demasiado lento para la informática de alto rendimiento. Eso ha cambiado con CUDA Python de Continuum Analytics. Con CUDA Python, utilizando el compilador Numba Python, obtienes lo mejor de ambos mundos: desarrollo iterativo rápido con Python combinado con la velocidad de un lenguaje compilado que apunta tanto a CPU como a GPU NVIDIA.

3.6 ¿Por qué Numba Python?

Numba es un compilador de Python de Anaconda que puede compilar el código de Python para su ejecución en GPU compatibles con CUDA o CPU multinúcleo. Dado que Python no es normalmente un lenguaje compilado, podemos cuestionar por qué es necesario un compilador de Python. La justificación está en que ejecutar un código compilado nativo es mucho más rápido que ejecutar código interpretado de manera dinámica. Numba funciona al permitir especificar firmas de tipo para las funciones de Python, lo que permite la compilación en tiempo de ejecución (esto es "Just-in-Time" o compilación JIT). La capacidad de Numba para compilar código dinámicamente significa que no renuncia a la flexibilidad de Python. Este es un gran paso para proporcionar la combinación ideal de programación de alta productividad y computación de alto rendimiento.

Con Numba, ahora es posible escribir funciones estándar de Python y ejecutarlas en una GPU compatible con CUDA. Numba está diseñado para tareas de computación orientadas a arrays, al igual que la biblioteca NumPy ampliamente utilizada . El paralelismo de datos en tareas de computación orientadas a arreglos es una opción natural para aceleradores como GPU. Numba entiende los tipos de matriz NumPy y los usa para generar código compilado eficiente para su ejecución en GPU o CPU multinúcleo. El esfuerzo de programación requerido puede ser tan simple como agregar decorador de funciones para instruir a Numba a compilar para la GPU.

Numba admite la programación de GPU CUDA compilando directamente un subconjunto restringido de código Python en núcleos CUDA y funciones de dispositivo siguiendo el modelo de ejecución CUDA.

Una característica que simplifica significativamente la escritura de núcleos GPU es que Numba hace parecer que el kernel tiene acceso directo a las matrices NumPy. Las

matrices NumPy que se suministran como argumentos al kernel se transfieren automáticamente entre la CPU y la GPU (aunque esto también puede ser un problema).

Numba todavía no implementa la API completa de CUDA, por lo que algunas características no están disponibles. Sin embargo, las funciones que se proporcionan son suficientes para comenzar a experimentar con la escritura de GPU enable kernels. El soporte de CUDA en Numba se está desarrollando activamente, por lo que, con el tiempo, la mayoría de las características deberían estar disponibles.

4 Desarrollo del Proyecto

En este apartado, se explica y detalla el proceso que ha seguido para desarrollar el proyecto. Se analizarán las diferentes decisiones que se han tomado y las soluciones a los problemas que han surgido. Como cualquier proyecto informático, el trabajo de ha dividido en diferentes partes, donde cada una corresponde a una parte de la implementación.

Antes de empezar, vamos a recordar los objetivos a alcanzar que hemos adelantado en el segundo apartado:

- Los alumnos deben poder ejecutar el programa cliente directamente en su máquina, sea cual sea su sistema operativo.
- Los clientes deben poder enviar archivos con extensión Python al servidor.
- El servidor debe poder atender y ejecutar las peticiones de varios clientes simultáneamente.
- El servidor debe poder ejecutar programas Python con CUDA y capturar el resultado de la ejecución de dichos programas.
- El servidor debe poder enviar a cada cliente, la salida correspondiente al programa que ha enviado para que el este lo ejecute.
- Los clientes deben poder recibir la salida del programa, que ha enviado previamente el servidor, y mostrarla en el equipo del alumno.

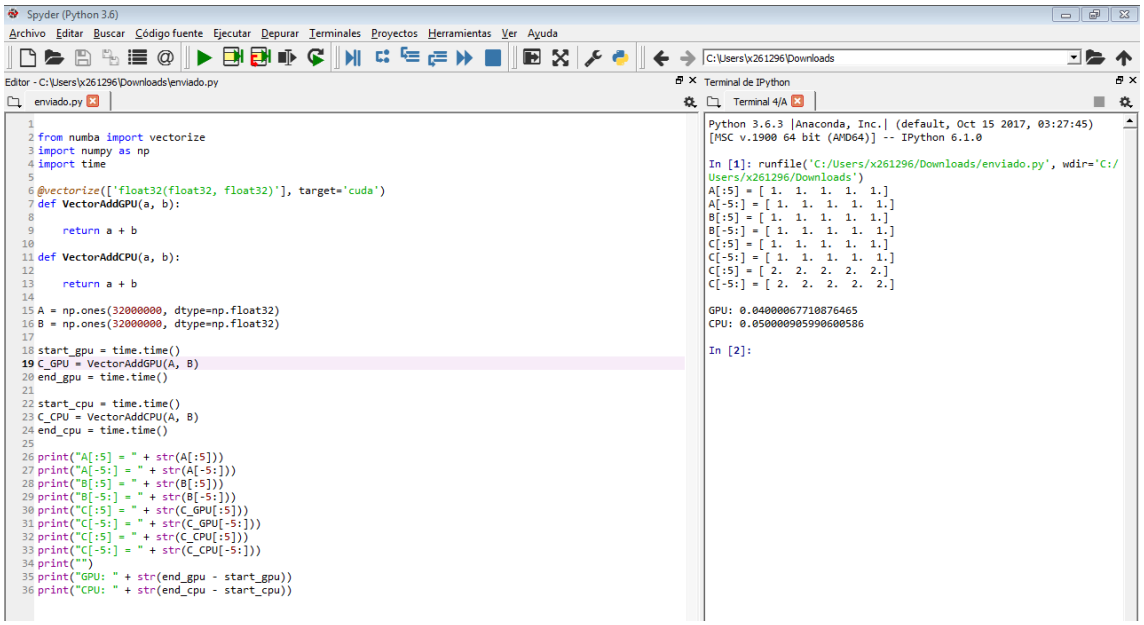
En base a estos objetivos, y como ya hemos comentado, vamos a dividir el desarrollo del proyecto en los siguientes apartados, cada uno con su duración correspondiente:

Preparación del Entorno y Pruebas	**2 semanas**
Análisis de Requisitos de la Aplicación	1 semana
Experimentación con CUDA Python	1 semana
Aplicaciones Distribuidas en Python	**3 semanas**
Aplicaciones Cliente-Servidor en Python	1 semana
Sockets en Python	1 semana
Multithreading en Python	1 semana
Trabajar con Ficheros en Python	**3 semanas**
Transferencia de Archivos a través de Sockets	1 semana
Ejecución de Procesos y Captura del Resultado	2 semanas
Creación de la Aplicación Final	**4 semanas**
Creación de un Archivo Ejecutable en Python	1 semana
Funcionalidad Final del Programa Servidor	2 semanas
Funcionalidad Final de Programa Cliente	1 semana

4.1 Experimentación con CUDA Python

Una vez aclarados los objetivos y fases que vamos a seguir para desarrollar la aplicación, nos surgen dudas de por dónde empezar. Posiblemente por nuestra inexperiencia con Python (lenguaje que no se utiliza a lo largo de la carrera) y con CUDA (utilizado durante solo en cuatrimestre en tercero de carrera). Por este mismo motivo, lo primero que vamos a hacer es familiarizarnos con ambos lenguajes y adquirir conocimientos que nos permitan desarrolla el resto del proyecto de una manera más ágil.

El primer ejemplo que vamos a hacer es una especie de "hola mundo" en CUDA, el cual consiste en realizar la suma de 2 vectores en CPU y en GPU, calculando y comparando el tiempo que se tarda en ejecutar cada una de las opciones.



Como vemos en el principio del programa, este usa el modulo "vectorize" de la biblioteca "numba" y que será la que nos permite realizar la suma de vectores usando CUDA. También se utiliza la biblioteca "numpy", típica en Python para el manejo de arrays y la biblioteca "time" para calcular el tiempo que tarda en ejecutarse cada suma.

Por primera vez podemos observar, de una manera más práctica que teórica, que realizar este tipo de operaciones con la GPU es mucho más rápido que con la CPU. Mientras que la ejecución en CPU ha tardado 0,05 segundos, la ejecución en GPU ha sido solo de 0,04 segundos. Lo cual supone una mejora de rendimiento del 20% y que será más notable cuantas más operaciones de este tipo realicemos en nuestro código.

4.2 Trabajar con Sockets en Python

Como venimos comentando en todo el documento, vamos a crear una aplicación distribuida con un solo servidor y varios clientes. Para este propósito, Python nos brinda la biblioteca “sockets” donde se pone a disposición de los desarrolladores una serie de funciones las cuales sirven para desarrollar clientes y servidores usando la tecnología de los sockets.

4.2.1 Socket en Python [Servidor]

En primer lugar, y como hemos hecho ya previamente en el “hola mundo” lo primero es importar la biblioteca “socket”, la cual nos da acceso a todas las funciones para trabajar con sockets. Una vez hecho esto necesitamos crear el socket servidor, asignarlo a una dirección y puerto concretos, especificar el número de clientes a los que va a escuchar y finalmente aceptar la conexión de los clientes y tener comunicación con ellos. Para cada una de estas tareas la biblioteca “socket” proporciona una función concreta y con diferentes parámetros:

- Función Socket (Familia, Tipo, Protocolo): Crea un nuevo socket usando la familia, tipo y protocolo especificados. La familia debe ser “AF_INET” (por defecto), “AF_INET6” o “AF_UNIX”. El tipo debe ser “SOCK_STREAM” (por defecto), “SOCK_DGRAM” o alguna de las demás constantes “SOCK_” disponibles. Por último, el protocolo es un número que normalmente es 0 y que por tanto puede omitirse con frecuencia. En nuestro caso particular queremos crear un socket TCP/IP y por tanto elegiremos los parámetros (AF_INET, SOCKET_STREAM, 0) respectivamente.

- Función Bind (Dirección): Enlaza un socket creado previamente con la dirección especificada como parámetro. El socket a enlazar no puede tener ningún vínculo activo y la dirección debe ser una tupla formada por una dirección IP y un puerto libre dentro de esta IP. El formato de la dirección IP va a depender de la familia elegido al crear el socket. En nuestro caso la familia elegida es “AF_INET” y los parámetros son ((“127.0.0.1”, 9999)) respectivamente.

- Función Listen (Conexiones): Hace que el socket comience a escuchar peticiones de clientes. El argumento “Conexiones” especifica el número máximo de conexiones en cola que puede atender el servidor (al número debe ser al menos 0). El número máximo de conexiones en cola depende el sistema, pero

normalmente y como nosotros vamos a indicar en nuestro código, elegiremos 5 como este número máximo.

- Función Accept(): Acepta la conexión de un cliente. Para que pueda llamarse a esta función, el socket debe haber llamado a las funciones Bind() y Listen(). El valor que retorna es una tupla (con, address) donde "con" es el objeto socket que representa al cliente conectado y "address" es la dirección asociada al socket al otro lado de la conexión (en este caso el socket cliente del que hablamos).

- Función Recv (Tamaño): Recibe los datos del socket, enviados por el otro extremo de la conexión y retorna este valor como un string. La cantidad máxima de datos que se reciben a la vez depende el parámetro tamaño. Hay que tener un especial cuidado en no sobrepasar el tamaño indicado en la conexión, porque en caso de hacerlo no se recibirá ningún mensaje más de la conexión. En nuestro caso hemos elegido 1024 como tamaño en cada comunicación, pues si nuestro cliente debe enviar más de esa cantidad, dividiremos él envió de paquetes de 1024 para que no falle la conexión.

- Función Send (Cadena): Envía datos al socket, para comunicar con el otro extremo de la conexión. Antes de realizar el envió de datos (parámetro cadena), debemos asegurarnos que el otro extremo, el cual va a recibir los datos, está preparado para recibir la cantidad indicada. En el caso de nuestro programa hemos capado en ambas direcciones el tamaño a 1024.

- Función Close(): Cierra el socket y por tanto todas las operaciones futuras en el objeto socket fallarán. El extremo remoto no recibirá más datos (después de que los datos en cola se vacíen). Los sockets se cierran automáticamente cuando se recolecta basura.

Finalmente, el código que implementa los Sockets en el servidor es el siguiente:

```
1
2  import logging
3  import socket
4
5  logging.basicConfig(filename='log.log',level=logging.DEBUG, format='%(asctime)s -- %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
6
7  socket_servidor = socket.socket()
8  socket_servidor.bind(("localhost", 9999))
9  socket_servidor.listen(128)
10
11 logging.info("SERVIDOR ARRANCADO Y ESPERANDO CLIENTES")
12 print("SERVIDOR ARRANCADO Y ESPERANDO CLIENTES")
13
14 while(1):
15
16     socket_cliente, address = socket_servidor.accept()
17
18     # CODIGO PARA ATENDER LA PETICION DEL CLIENTE
19
20 socket_servidor.close()
21
```

4.2.2 Socket en Python [Cliente]

En primer lugar, y como hemos hecho ya previamente en el "hola mundo" y en el servidor lo primero es importar la biblioteca "socket", la cual nos da acceso a todas las funciones para trabajar con sockets. Una vez hecho esto necesitamos crear el socket cliente, conectarnos con el socket servidor y comunicarnos con el activamente. Al igual que ocurría con el servidor, la biblioteca "socket" proporciona todas las funciones necesarias para llevar esto a cabo (algunas de estas funciones son las mismas que las usadas en el servidor):

- Función Socket (Familia, Tipo, Protocolo): Crea un nuevo sockets usando la familia, tipo y protocolo especificados. La familia debe ser "AF_INET" (por defecto), "AF_INET6" o "AF_UNIX". El tipo debe ser "SOCK_STREAM" (por defecto), "SOCK_DGRAM" o alguna de las demás constantes "SOCK_" disponibles. Por último, el protocolo es un número que normalmente es 0 y que por tanto puede omitirse con frecuencia. En nuestro caso particular queremos crear un socket TCP/IP y por tanto elegiremos los parámetros (AF_INET, SOCKET_STREAM, 0) respectivamente.

- Función Connect (Dirección): Conecta al socket (típicamente el cliente), con un socket que haya realizado un Bind en la dirección a la que nos estamos conectando. La dirección debe ser una tupla formada por una dirección IP y un puerto libre dentro de esta IP. El formato de la dirección IP va a depender de la familia elegido al crear el socket. En nuestro caso la familia elegida es "AF_INET" y los parámetros son (("127.0.0.1", 9999)) respectivamente, ya que esta es la dirección donde hemos puesto a escuchar a nuestro servidor.

- Función Recv (Tamaño): Recibe los datos del socket, enviados por el otro extremo de la conexión y retorna este valor como un string. La cantidad máxima de datos que se reciben a la vez depende el parámetro tamaño. Hay que tener un especial cuidado en no sobrepasar el tamaño indicado en la conexión, porque en caso de hacerlo no se recibirá ningún mensaje más de la conexión. En nuestro caso hemos elegido 1024 como tamaño en cada comunicación, pues si nuestro cliente debe enviar más de esa cantidad, dividiremos él envió de paquetes de 1024 para que no falle la conexión.

- Función Send (Cadena): Envía datos al socket, para comunicar con el otro extremo de la conexión. Antes de realizar el envío de datos (parámetro cadena), debemos asegurarnos que el otro extremo, el cual va a recibir los datos, está

preparado para recibir la cantidad indicada. En el caso de nuestro programa hemos capado en ambas direcciones el tamaño a 1024.

- Función Close (): Cierra el socket y por tanto todas las operaciones futuras en el objeto socket fallarán. El extremo remoto no recibirá más datos (después de que los datos en cola se vacían). Los sockets se cierran automáticamente cuando se recolectan basura.

Después de explicar que hacen y para que vamos a utilizar todas estas funciones, el código que implementa los Sockets en el cliente es el siguiente:

```
1
2 # IMPORTAMOS LA LIBRERIA "SOCKET" PARA TRABAJAR CON SOCKETS
3 import socket
4
5 while(1):
6
7     # ABRIMOS EL SOCKET CLIENTE
8     socket_cliente = socket.socket()
9
10    # CONECTAMOS CON EL SOCKET SERVIDOR
11    socket_cliente.connect(("localhost", 9999))
12
13    # AQUI IRA EL CODIGO PARA ENVIER EL ARCHIVO AL SERVIDOR
14
15    # RECIBIMOS DEL SERVIDOR EL OUTPUT DE EJECUTAR EL FICHERO ENVIADO
16    input_data = socket_cliente.recv(1024)
17
18    # IMPRIMIMOS EL OUTPUT EN LA PANTALLA DEL CLIENTE
19    print("\n" + input_data.decode("utf-8"))
20
21    # CERRAMOS EL SOCKET CLIENTE
22    socket_cliente.close()
23
```

4.3 Multithreading en Python

Una vez desarrollados los dos puntos anteriores, tenemos claro que nuestro proyecto va a seguir una arquitectura cliente-servidor, donde varios clientes se van a conectar a un único servidor. Estas conexiones pueden ser en instantes separados en el tiempo, pero también pueden ser conexiones simultáneas. Si esto ocurre, el servidor según lo tenemos montado ahora mismo no está preparado para soportar esa simultaneidad. Para evitar este problema en Python 3 existe la biblioteca “thread”, que permite acceder a algunas funciones para explicar la tecnología del multithreading.

La biblioteca “thread” permite a los desarrolladores trabajar y gestionar hilos dentro del proceso principal de ejecución. Hay dos formas de especificar la actividad que van a ejecutar los hilos (pasando un objeto al constructor o sobrescribiendo el método “run” en una subclase). Ningún otro método (excepto el constructor) debe ser sobrescrito en una subclase, es decir, solo se deben sobrescribir los métodos init () y run () de esta clase. Nosotros vamos a optar por la primera opción, capturando toda la atención del servidor a los clientes en una sola función, y creando un hilo asociado a esa función y para cada cliente.

Concretamente, las dos funciones que vamos a utilizar para implementar la tecnología threading en el servidor (en realidad son un constructor de la clase “thread” y una función) son las siguientes:

- Clase Thread (Grupo, Objetivo, Nombre, Argumentos): Este constructor devuelve un objeto concreto de la clase “thread” en función de los parámetros que le pasemos, este objeto puede ser ejecutado posteriormente usando la función start (). El “Grupo” debe ser None, ya que debe reservarse para futuras extensiones cuando el ThreadGroup sea implementado en una clase. “Objetivo” es el objeto (o función) que el método run () debe invocar, siendo None el valor predeterminado, lo que significa que no se llama nada. “Nombre” es el nombre del hilo, por defecto, un nombre se construye con la forma "Thread- N" donde N es un número decimal. “Argumentos” es el argumento tuple para la invocación de destino, con valor predeterminado a Null. Si la subclase anula al constructor, debe asegurarse de invocar el constructor de la clase base antes de hacer algo más al hilo.

- Función Start (): Comienza la actividad del hilo y debe llamarse como máximo una vez por objeto de hilo. Hace que el método run () del objeto sea invocado en un hilo de control separado. Este método generará un RuntimeError si se llama más de una vez en el mismo objeto de subproceso.

Solamente con estas dos funciones, y añadiendo el código después de que el servidor acepte la conexión, conseguimos añadir al servidor el paralelismo que necesita:

```
while(1):

    socket_cliente, address = socket_servidor.accept()

    logging.info("CLIENTE CONECTADO DESDE LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
    print("CLIENTE CONECTADO DESDE LA IP %s Y EL PUERTO %s" %(address[0],address[1]))

    hilo = threading.Thread(target=ejecutar, args=(socket_cliente,))
    hilo.start()
```

4.4 Transferencia de Archivos a través de Sockets

Una vez tenemos la arquitectura cliente-servidor terminada y preparada para atender a varios clientes a la vez, vamos a analizar qué tipo de comunicación va a haber entre los clientes y el servidor.

En primer lugar, el cliente va a enviar un archivo con extensión Python al servidor junto con los posibles parámetros que fueran necesarios para la ejecución y posteriormente el servidor le va a responder con el resultado de ejecutar el programa Python que ha recibido. La forma de llevar a cabo esta respuesta por parte del servidor vamos a verla en el siguiente apartado, ahora sin embargo vamos a centrar en como el cliente debe enviar el archivo para que la conexión no se corte y llegue correctamente al servidor.

El usuario que ejecute el programa cliente, debe tener el archivo a enviar en su propia máquina, y debe especificar la ruta local hasta el archivo para que el programa sepa dónde buscarlo. En principio la longitud del archivo no importa (pues como vamos a explicar más adelante vamos a fragmentarlo) mientras que la extensión sea correcta.

Como hemos comentado a la hora de configurar el servidor, el máximo de bytes que puede recibir en cada mensaje son 1024, por tanto a la hora de enviar el archivo debemos fragmentarlo en “paquetes” de ese tamaño, para conseguirlo simplemente debemos abrir el archivo y leerlo dentro de un bucle tantas veces como necesitamos para llegar al fin del fichero. Una vez llegamos al final, y para avisar al servidor que ya hemos terminado, enviamos el carácter 1 (hemos elegido ese carácter porque es imposible que una parte del fichero solo contenga ese carácter). Como podemos en la siguiente imagen:

```
    # ESPERAMOS A QUE EL USUARIO INTRODUZCA LA RUTA DEL FICHERO A ENVIAR
    lista = input(">> %run ").split() # C:\Users\Manue\Dropbox\TFG\PROCESOS\PC_CLIENTE\enviado.py

    # ABRIMOS, ENVIAMOS AL SERVIDOR Y CERRAMOS EL FICHERO

    while(1):

        python_script = open(lista[0], "rb")
        row = python_script.readline()

        while row:

            socket_cliente.send(row)
            row = python_script.readline()

        break

    python_script.close()
    socket_cliente.send(bytes(chr(1), "utf-8"))

    # ENVIAMOS LOS ARGUMENTOS AL SERVIDOR

    if(len(lista) > 1):

        socket_cliente.send(bytes("\n".join(lista[1:]), "utf-8"))

    else:

        socket_cliente.send(bytes(chr(1), "utf-8"))

```

Por su parte, el servidor también tiene que estar preparado para recibir el archivo de esta manera. En primer lugar tiene que crear el archivo (puesto que lo que va a recibir solo es el contenido no el archivo en sí) y para ello debe darle un nombre, para evitar que existan problemas de acceso a un mismo archivo por dos conexiones simultaneas el nombre elegido va a estar formado por la dirección y puerto del cliente que envía el archivo. Una vez creado se va rellenando también en bucle y hasta que llegue al carácter 1 en código ASCII (que marca el fin del archivo), en ese momento cierra el archivo y lo deja preparado para que sea ejecutado en paralelo como vamos a ver en el siguiente punto.

```
    logging.info("RECIBIENDO EL FICHERO DESDE LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
    print("RECIBIENDO EL FICHERO DESDE LA IP %s Y EL PUERTO %s" %(address[0],address[1]))

    python_script = open(str(address[0]) + "_" + str(address[1]) + ".py", "wb")

    while(1):

        try:

            input_data = socket_cliente.recv(1024)

        except socket.error:

            break

        else:

            if input_data[0] == 1:

                break

            else:

                python_script.write(input_data)


    python_script.close()

    input_data = socket_cliente.recv(1024)

    if input_data[0] == 1:

        input_data = ""

```

4.5 Ejecución de Subprocesos y Captura del Resultado

El enfoque recomendado para invocar subprocesos es utilizar la función Run () siempre y cuando las funcionalidades que esta función nos otorga sean válidas para nuestro propósito. Para casos de uso más avanzados, como es el nuestro, se utiliza la interfaz Popen (). Vamos a explicar cómo funciona el constructor de esta interfaz:

- Constructor Popen (args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0): Ejecuta el programa hijo en un nuevo proceso usando la función CreateProcess. Args debe ser una lista de argumentos o una cadena de texto en caso de que solo sea un argumento (el primer argumento representa el programa a ejecutar), en nuestro caso el programa a ejecutar es Python y el segundo argumento es el nombre del script enviado por el cliente (como hemos comentado antes formado por la IP y puerto desde el que se conecta). Bufsize especifica el tamaño aproximado en bytes, aunque si vale 0 indica que la clase es "unbuffered", este último es nuestro caso. Stdin, stdout y stderr especifica cuales van a ser los manejadores en caso de input, outpot y error en el subproceso. En nuestro caso tanto el stdin y el stdout estarán asignados a PIPE, para poder enviar/recoger esta información cuando usemos porteriormente la función Comunicate(). Mientras que el stderr está asignado a STDOUT, para que en caso de que el resultado del programa ejecutado por el subproceso este erróneo, envié como output el mensaje de error con su detalle. El resto de argumentos los vamos a dejar con su valor predeterminado, puesto que no son demasiado útiles para el propósito que buscamos.

Como ya hemos mencionado, después de especificar el objeto de la clase Popen, invocaremos a la función Comunicate () que será la encargada de ejecutar verdaderamente el archivo siguiendo los parámetros que ya hemos indicado y algunos nuevos:

- Función Comunicate (Stdin): Lanza el proceso, interactúa con él y devuelve el resultado (tanto correcto como erróneo). Los parámetros de entrada que pueda necesitar el programa son definidos en Stdin (si son varios se deben incluir como una sola cadena separada por saltos de línea).

Con estas dos utilidades que nos proporciona la librería "subprocess" somos capaces de llevar a cabo la ejecución y captura del resultado de los programas enviamos por los clientes.

```
52    logging.info("ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
53    print("ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
54
55    process = subprocess.Popen(["python", str(address[0]) + "_" + str(address[1]) + ".py"], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
56    out = process.communicate(input=input_data)[0]
57    socket_cliente.send(out)
58    socket_cliente.close()
59
60    logging.info("CERRANDO CONEXION CON LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
61    print("CERRANDO CONEXION CON LA IP %s Y EL PUERTO %s" %(address[0],address[1]))
```

Por último, solo quedaría enviar este resultado al cliente para que este lo muestre en la pantalla del usuario que envío el script.

4.6 Creación de un Archivo Ejecutable en Python

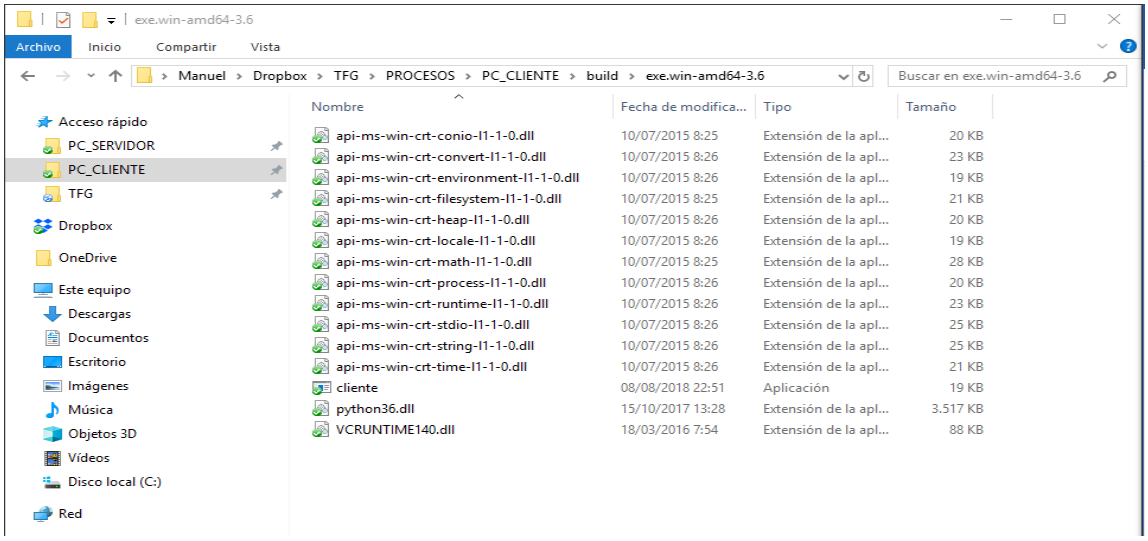
Con el objetivo de que el usuario pueda ejecutar el programa cliente sin necesidad de tener CUDA ni Anaconda instalado, vamos a crear un archivo cliente.exe. Para ellos vamos a seguir los siguientes pasos:

- Primer Paso: Creamos un archivo llamado setup.py en la misma carpeta donde esté el archivo .py que queramos convertir a .exe. El contenido del archivo setup.py debe contener información sobre el nombre del archivo, algún comentario, versión del programa, etc. El código del setup.py concreto que hemos utilizado se puede encontrar en el anexo 4.

- Segundo Paso: Abrir la consola de Windows (símbolo de sistema). En Windows 10 lo más rápido es hacer clic con el botón derecho en el logo de Windows y pichar en “símbolo de sistema”. Una vez que la hemos abierto debemos llegar hasta la carpeta donde tenemos nuestros dos archivos Python. recordemos que para acceder a una carpeta usamos cd “miCarpeta” y para ver que carpetas hay en el directorio donde estemos usamos “dir”.

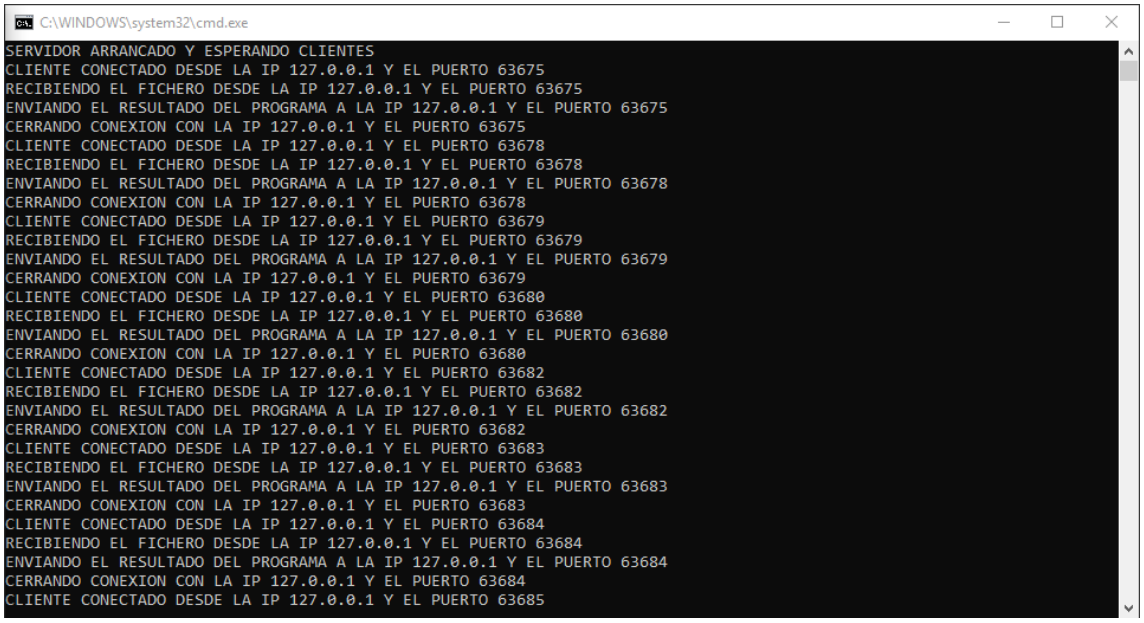
- Tercer Paso: Una vez que ya hemos llegado al directorio objetivo. Introducimos la siguiente sentencia: setup.py py2exe. Nos empezará a salir información en la consola de archivos que está creando. Cuando haya finalizado entonces tendremos nuestro archivo .exe en una carpeta llama "dist". El programa genera 2 carpetas una llamada "dist" y otra llamada "build" pero con el setup.py que hemos usado nos basta con quedarnos sólo con el archivo ejecutable.

n



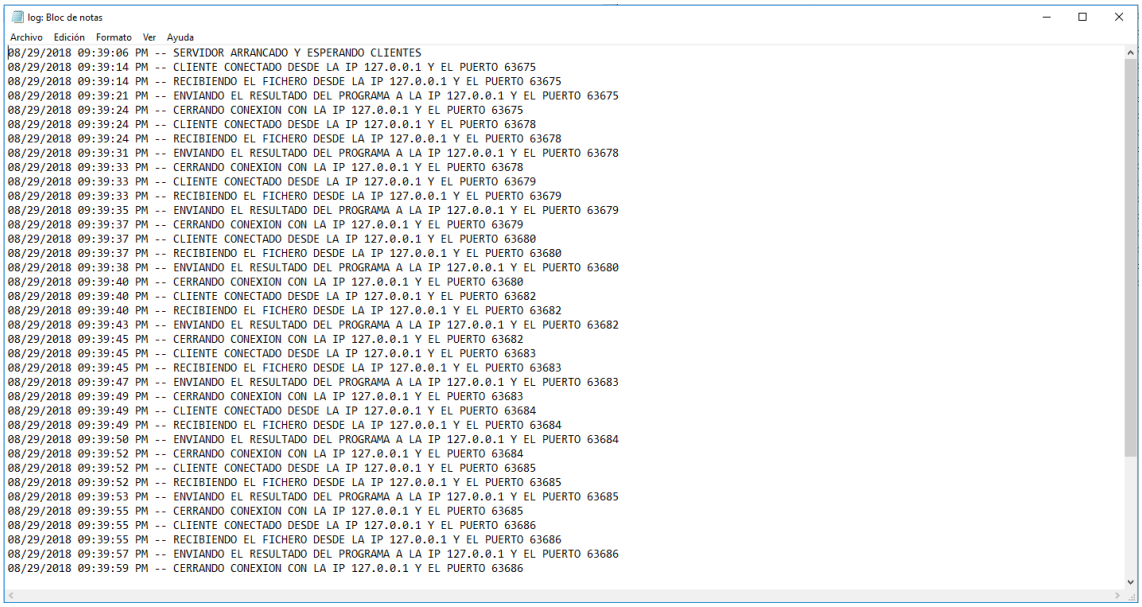
4.7 Funcionalidad Final del Programa Servidor

Para poder comprobar el que va a ser el comportamiento final de la aplicación, lo primero que vamos a hacer es arrancar el servidor y dejarlo a la espera de que los clientes se conecten a él. Para poder llevar un seguimiento del trabajo del servidor, este va mostrando por pantalla mensajes cuando un cliente se conecta, recibe un programa, envía el resultado de la ejecución al cliente y finalmente cuando un cliente se desconecta. Estos mensajes muestran el registro de actividad del servidor como podemos ver en la siguiente imagen:



A demás de mostrar estos mensajes por pantalla, se ha decidido llevar un mejor registro del tráfico del servidor es crear un “logger” el cual guarde en un archivo todos los mensajes que muestra el servidor y la hora a la que se produjo el evento.

Para este propósito, Python pone a nuestra disposición la biblioteca “logging”, la cual define funciones y clases que implementan un sistema flexible de registro de eventos para aplicaciones y bibliotecas. Al final un log es algo muy sencillo que podríamos haber creado nosotros de manera manual, utilizando funciones de Python para el manejo de ficheros. Sin embargo, el beneficio clave de tener la API de registro proporcionada por un módulo de biblioteca estándar, es que todos los módulos de Python pueden participar en el registro, por lo que el registro de la aplicación puede incluir sus propios mensajes integrados con mensajes de módulos externos.

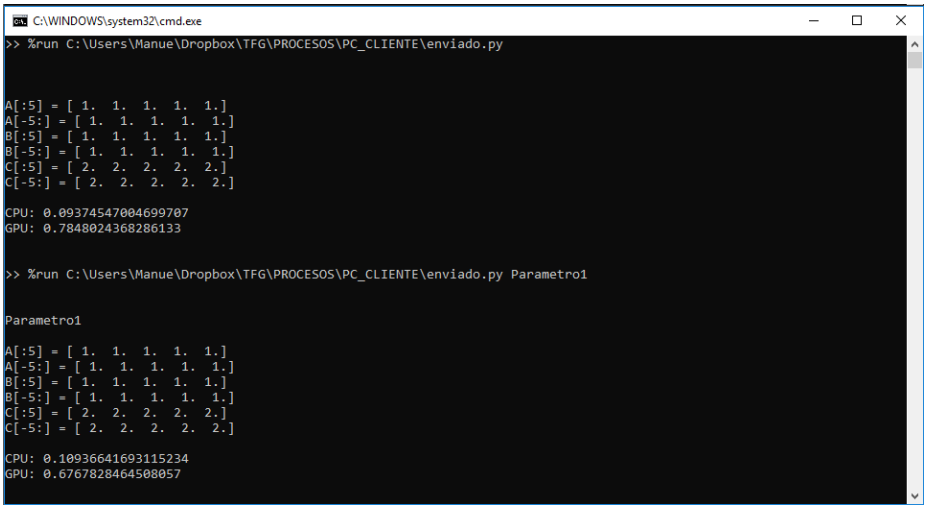

log: Bloc de notas

Archivo Edición Formato Ver Ayuda

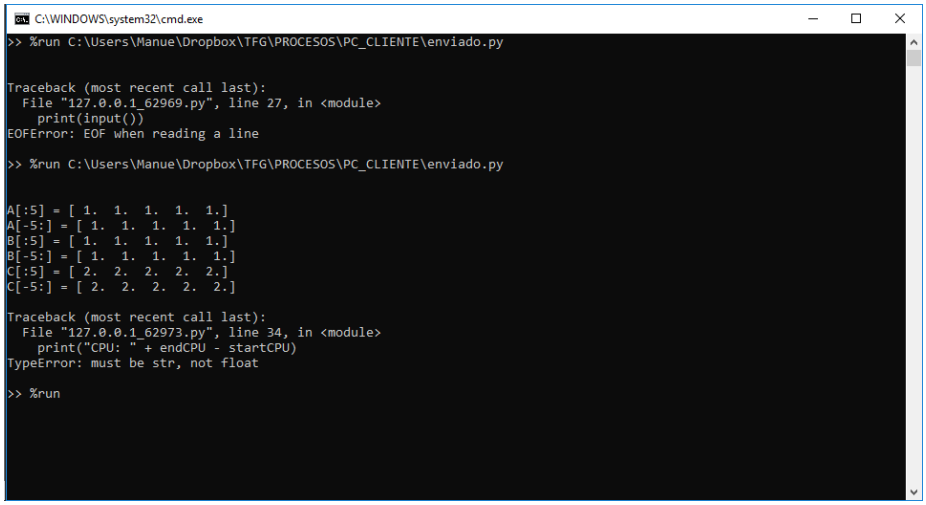
```
08/29/2018 09:39:06 PM -- SERVIDOR ARRANCADO Y ESPERANDO CLIENTES
08/29/2018 09:39:14 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63675
08/29/2018 09:39:14 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63675
08/29/2018 09:39:21 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63675
08/29/2018 09:39:24 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63675
08/29/2018 09:39:24 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63678
08/29/2018 09:39:24 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63678
08/29/2018 09:39:31 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63678
08/29/2018 09:39:33 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63678
08/29/2018 09:39:33 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63679
08/29/2018 09:39:33 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63679
08/29/2018 09:39:35 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63679
08/29/2018 09:39:37 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63679
08/29/2018 09:39:37 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63680
08/29/2018 09:39:37 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63680
08/29/2018 09:39:38 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63680
08/29/2018 09:39:40 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63680
08/29/2018 09:39:40 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63682
08/29/2018 09:39:40 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63682
08/29/2018 09:39:43 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63682
08/29/2018 09:39:45 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63682
08/29/2018 09:39:45 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63683
08/29/2018 09:39:45 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63683
08/29/2018 09:39:47 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63683
08/29/2018 09:39:49 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63683
08/29/2018 09:39:49 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63684
08/29/2018 09:39:49 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63684
08/29/2018 09:39:50 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63684
08/29/2018 09:39:52 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63684
08/29/2018 09:39:52 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63685
08/29/2018 09:39:52 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63685
08/29/2018 09:39:53 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63685
08/29/2018 09:39:55 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63685
08/29/2018 09:39:55 PM -- CLIENTE CONECTADO DESDE LA IP 127.0.0.1 Y EL PUERTO 63686
08/29/2018 09:39:55 PM -- RECIBIENDO EL FICHERO DESDE LA IP 127.0.0.1 Y EL PUERTO 63686
08/29/2018 09:39:57 PM -- ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP 127.0.0.1 Y EL PUERTO 63686
08/29/2018 09:39:59 PM -- CERRANDO CONEXION CON LA IP 127.0.0.1 Y EL PUERTO 63686
```

4.8 Funcionalidad Final de Programa Cliente

Una vez arrancado el servidor podemos empezar a ejecutar tantos clientes como queramos. Simplemente ejecutando el .exe nos aparecerá una terminal con el cursor listo para escribir el path del programa que queremos que el servidor ejecute. Como podemos ver en el siguiente ejemplo es tan fácil como indicar la ruta y los posibles parámetros (en el primer programa no hay inputs mientras que el segundo necesita un parámetro para ejecutarse correctamente):



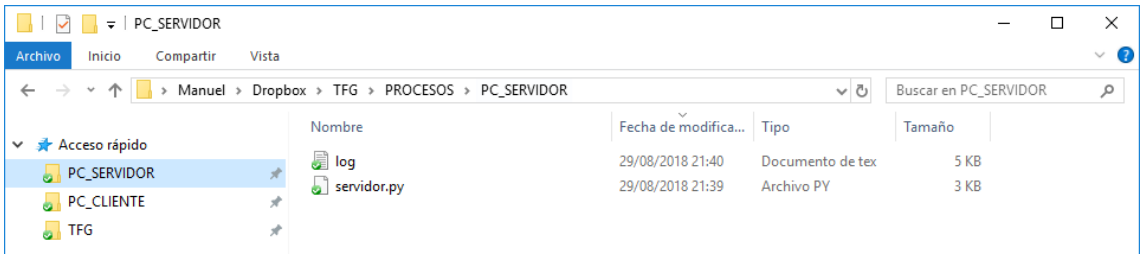
De la misma manera que hemos ejecutado correctamente los dos programas anteriores, puede ocurrir que enviemos al servidor un programa con errores. Los errores pueden ser en el número de parámetros enviados (como en el primer caso de la siguiente imagen) o porque hayamos cometido algún error de programación en alguna parte del código (como ocurre en el segundo ejemplo de la imagen):



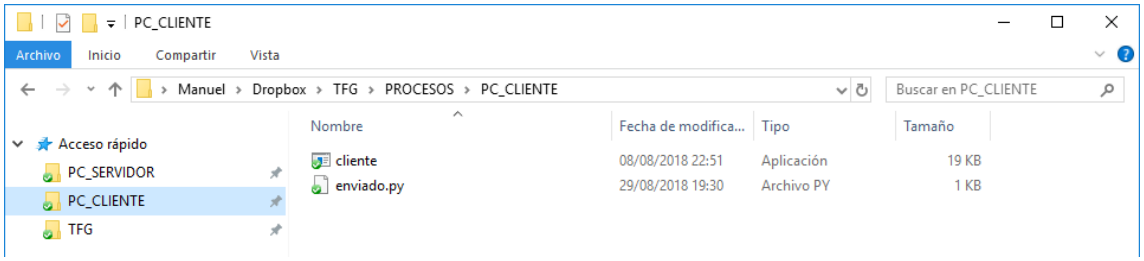
4.9 Funcionalidad Final de la Aplicación

Finalmente, y una vez que tenemos toda la aplicación desarrollada vamos a mostrar paso a paso que es necesario y como debemos poner en marcha el sistema:

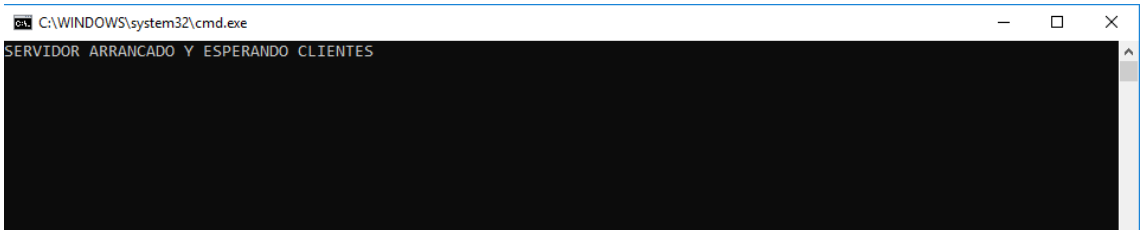
- Paso 1 [Servidor]: Tener el servidor listo para ejecutarse, es decir, haber realizado todas las acciones explicadas en el punto [4.1 Preparación del Entorno de Desarrollo] y tener accesible tanto el script del servidor como el log que usa:



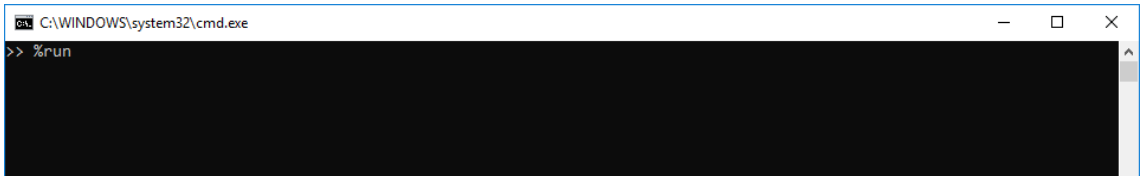
- Paso 2 [Cliente]: Tener el cliente listo para ejecutarse, es decir, tener listo el archivo ejecutable del cliente, y el script de Python que vamos a enviar al servidor para que lo ejecute. Como ya hemos mencionado anteriormente en cliente no necesita llevar a cabo ninguna de las acciones explicadas en el punto [4.1 Preparación del Entorno de Desarrollo]. Esta sería la estructura necesaria para ejecutarse posteriormente:



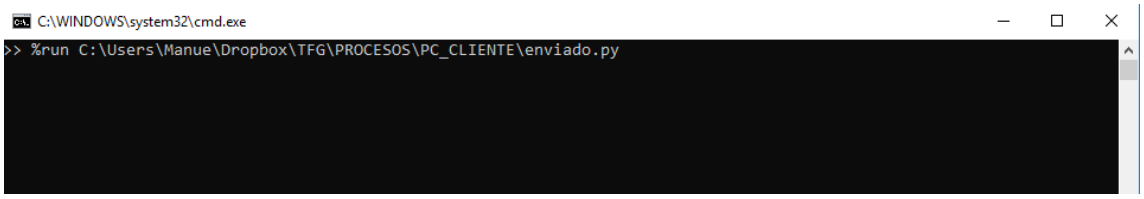
- Paso 3 [Servidor]: Arrancar el servidor.



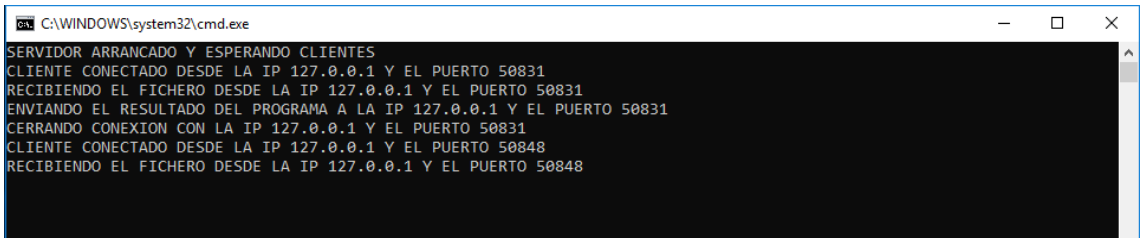
- Paso 4 [Cliente]: Ejecutar el cliente.



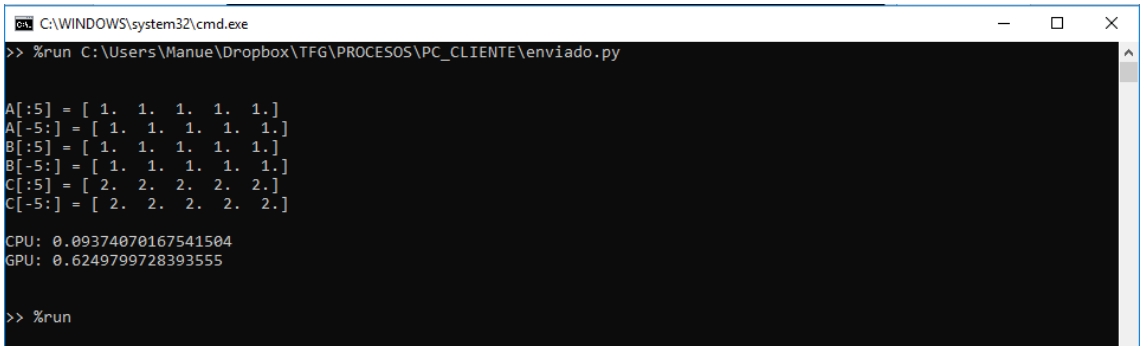
- Paso 5 [Cliente]: Enviar el programa al servidor y esperar a que este responda.



- Paso 6 [Servidor]: El servidor recibirá el archivo, lo ejecutará y devolverá el resultado al cliente. Dejando el siguiente registro tanto en la pantalla como en el log:



- Paso 7 [Cliente]: Finalmente el cliente recibirá el resultado enviado por el servidor y se le vuelve a dar la opción de enviar un nuevo script de Python.



4.10 Manuales de Usuario

Debido a que este proyecto está destinado a un uso concreto dentro del Departamento de Ciencias de la Computación de la Universidad Politécnica Superior de Alcalá de Henares. Vamos a finalizar incluyendo un manual de uso para cada uno de los roles que tiene la aplicación que en nuestro caso van a ser tres. Se van a diseñar por tanto tres manuales, uno para el instalador, otro para el administrador y otro para los clientes.

4.10.1 Manual de Instalación

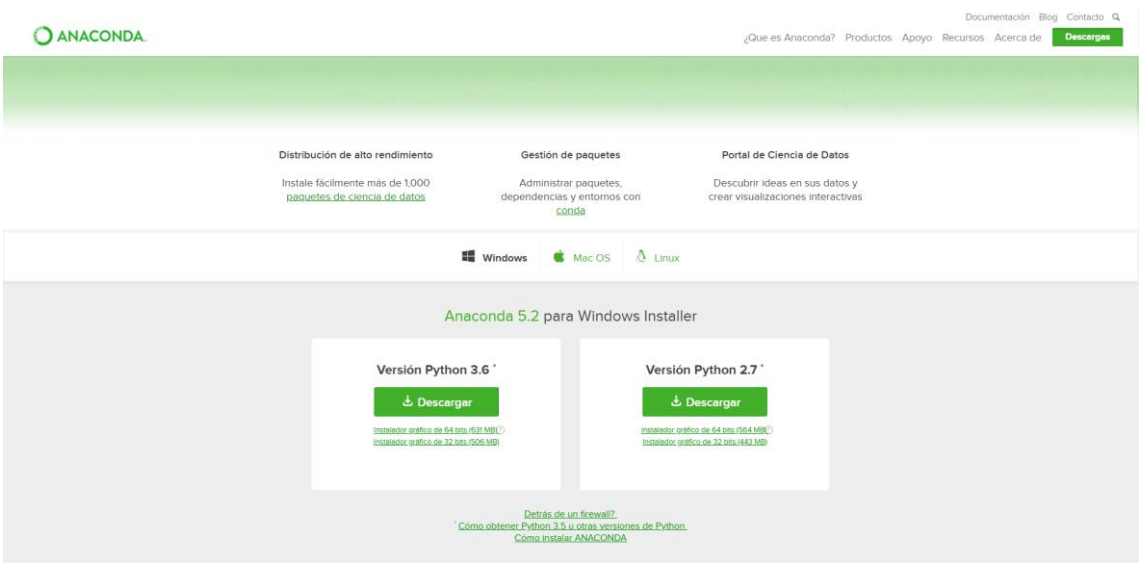
Partiendo de un entorno preinstalado con Windows 10 y una arquitectura hardware estándar, debemos contar con un hardware específico como debe ser una tarjeta gráfica que soporte tecnología CUDA, en nuestro caso hemos elegido la tarjeta Nvidia Geforce GTX 760 de 4GB (aunque el mismo manual puede servir para cualquier otra tarjeta que acepte tecnología CUDA), que es una buena opción en relación calidad-precio y tiene una potencia más que suficiente para realizar nuestro proyecto. Para apoyar esta afirmación, aquí podemos ver una tabla con las principales características de la tarjeta:

Especificaciones de la GPU	
Núcleos CUDA	1152
Frecuencia de reloj normal	980
Frecuencia acelerada	1033
Tasa de relleno de texturas	94.1
Especificaciones de la Memoria	
Frecuencia de la memoria (Gbps)	6.0
Cantidad de memoria	4096 MB
Interfaz de memoria	256-bit GDDR5
Ancho de banda máximo	192.2
Características de la Tarjeta	
Entorno de programación	CUDA
DirectX	11
OpenGL	4.3
Especificaciones Generales	
Máxima resolución digital	4096x2160
Máxima resolución VGA	2048x1536
Conectividad multimedia	DisplayPort, Dual Link y HDMI

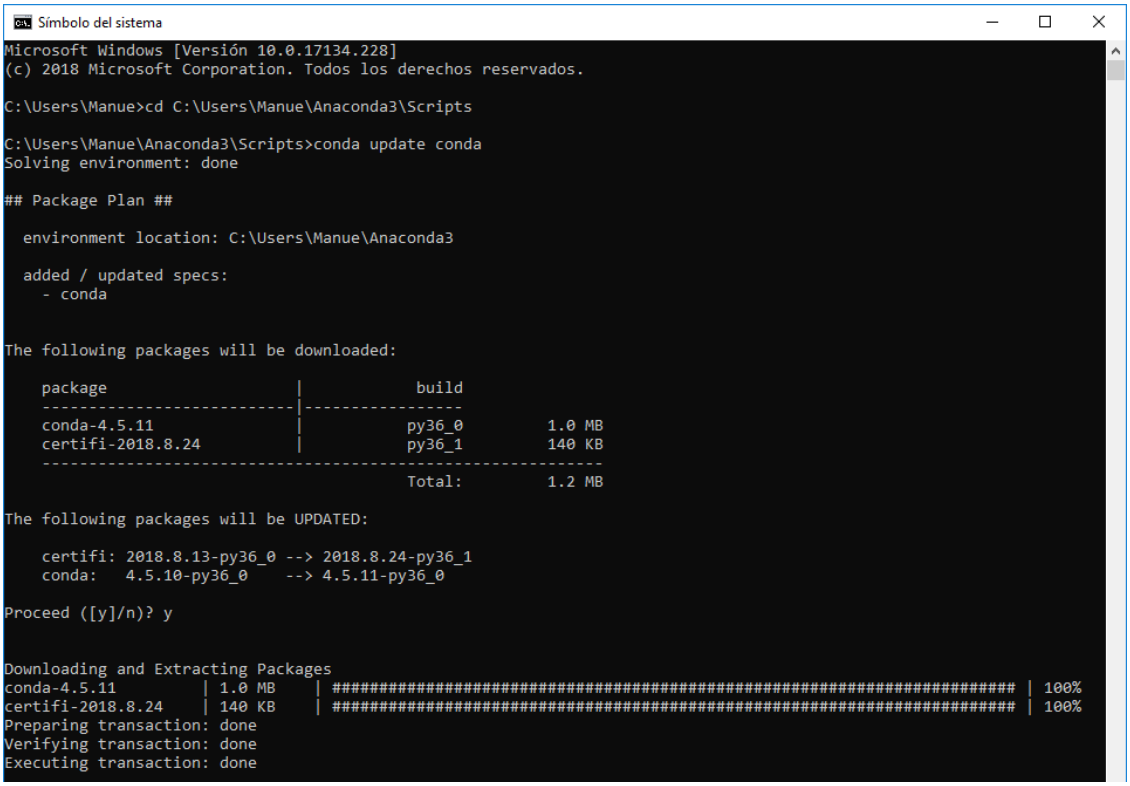
Con respecto al software, lo primero que tenemos que hacer es instaladr el CUDA Toolkit, el cual permite desarrollar, optimizar e implementar aplicaciones en sistemas embebidos acelerados por GPU, estaciones de trabajo de escritorio, centros de datos empresariales, plataformas basadas en la nube y supercomputadoras HPC. Actualmente la versión más reciente del CUDA Toolkit es la versión 9.2, sin embargo, si miramos la

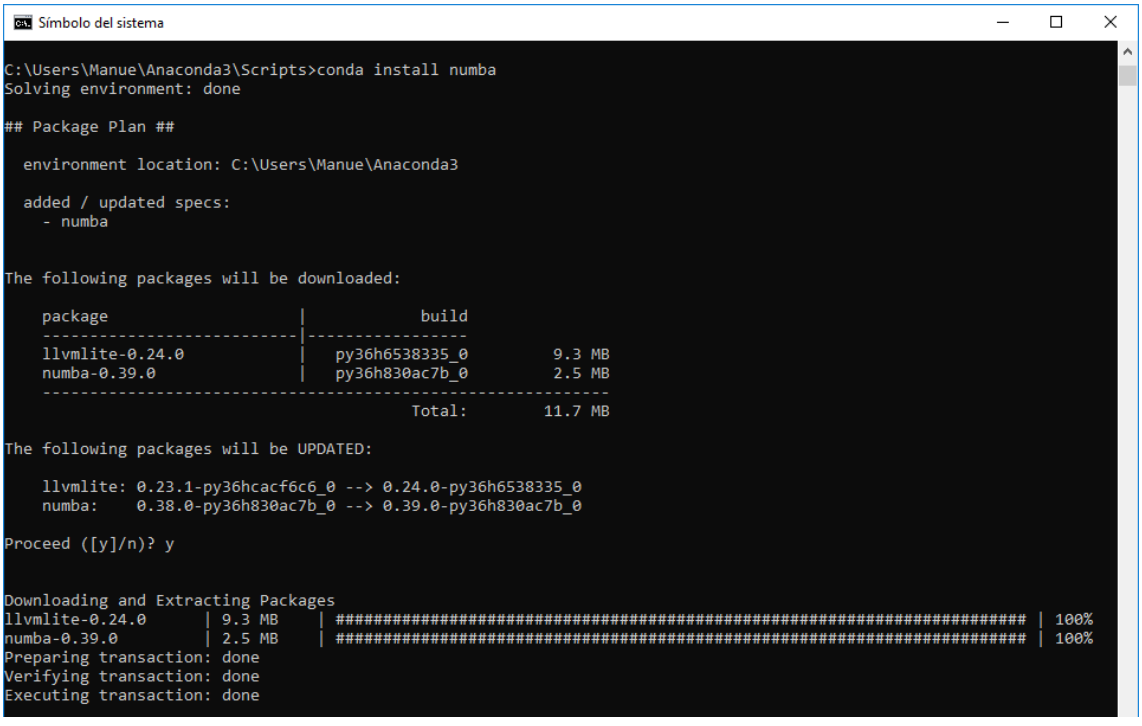
lista que relaciona el modelo de tarjeta gráfica y la versión de CUDA y que podemos encontrar en la página oficial de NVIDIA, para nuestra tarjeta debemos instalar la versión 3 del CUDA Toolkit.

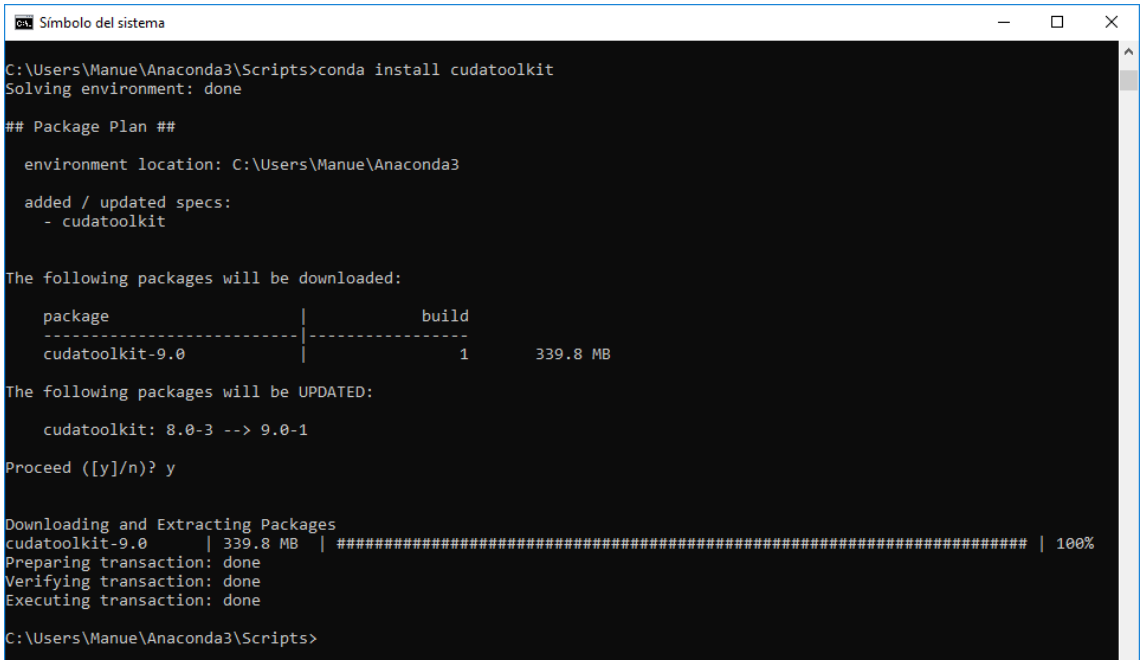
Una vez instalado CUDA, vamos a pasar ahora a instalar el entorno de programación de Python. La distribución que hemos elegido es Anaconda, concretamente la versión de Python 3.6. Como podemos ver en la página oficial de Anaconda, actualmente existe la posibilidad de descargar la versión con Python 2.7 y, aunque es cierto que existe más documentación de esta versión antigua y que todavía existe un gran porcentaje de uso de esta versión, queremos crear una aplicación duradera en el tiempo y que no se quede obsoleta con el paso de los años.



Otro de los motivos por lo que hemos elegido Anaconda como distribución, es porque tiene una integración relativamente sencilla con el paquete Numba, que es, como ya hemos comentado en puntos anteriores, el que nos permite combinar la potencia de CUDA y Python en un mismo entorno. Gracias a la familia de comandos “conda” que otorga Anaconda, la instalación de este paquete es muy cómoda. Como podemos ver aquí:


Símbolo del sistema
Microsoft Windows [Versión 10.0.17134.228]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.
C:\Users\Manue>cd C:\Users\Manue\Anaconda3\Scripts
C:\Users\Manue\Anaconda3\Scripts>conda update conda
Solving environment: done
Package Plan
environment location: C:\Users\Manue\Anaconda3
added / updated specs:
- conda
The following packages will be downloaded:
package | build
conda-4.5.11 | py36_0 1.0 MB
certifi-2018.8.24 | py36_1 140 KB
Total: 1.2 MB
The following packages will be UPDATED:
certifi: 2018.8.13-py36_0 --> 2018.8.24-py36_1
conda: 4.5.10-py36_0 --> 4.5.11-py36_0
Proceed ([y]/n)? y
Downloading and Extracting Packages
conda-4.5.11 | 1.0 MB | ## | 100%
certifi-2018.8.24 | 140 KB | ## | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

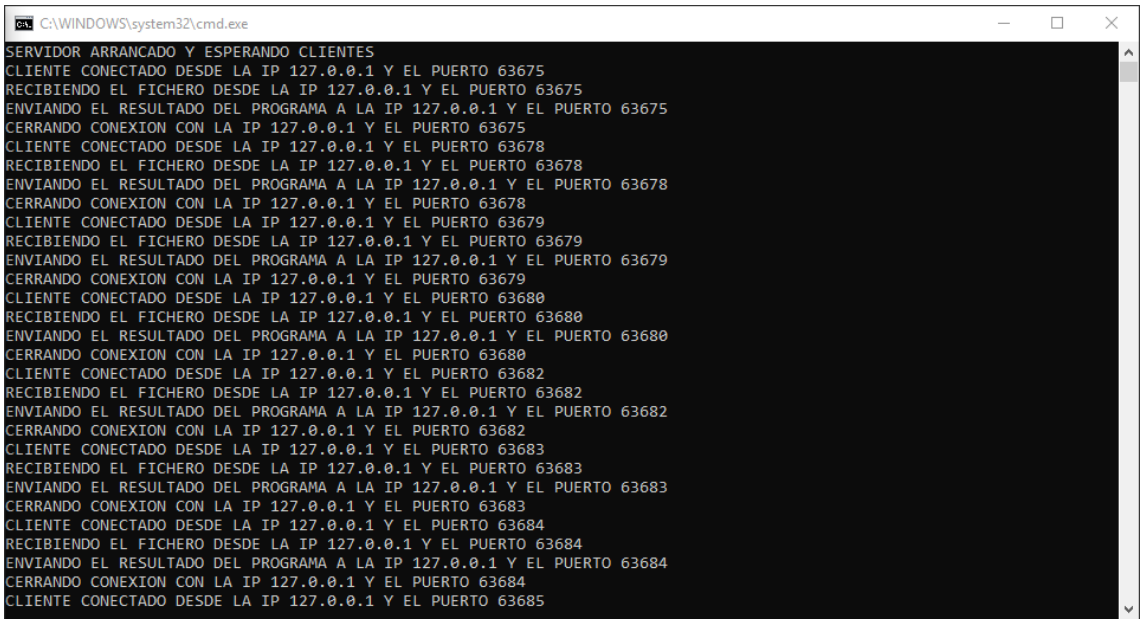

Símbolo del sistema
C:\Users\Manue\Anaconda3\Scripts>conda install numba
Solving environment: done
Package Plan
environment location: C:\Users\Manue\Anaconda3
added / updated specs:
- numba
The following packages will be downloaded:
package | build
llvmlite-0.24.0 | py36h6538335_0 9.3 MB
numba-0.39.0 | py36h830ac7b_0 2.5 MB
Total: 11.7 MB
The following packages will be UPDATED:
llvmlite: 0.23.1-py36hcacf6c6_0 --> 0.24.0-py36h6538335_0
numba: 0.38.0-py36h830ac7b_0 --> 0.39.0-py36h830ac7b_0
Proceed ([y]/n)? y
Downloading and Extracting Packages
llvmlite-0.24.0 | 9.3 MB | ## | 100%
numba-0.39.0 | 2.5 MB | ## | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done



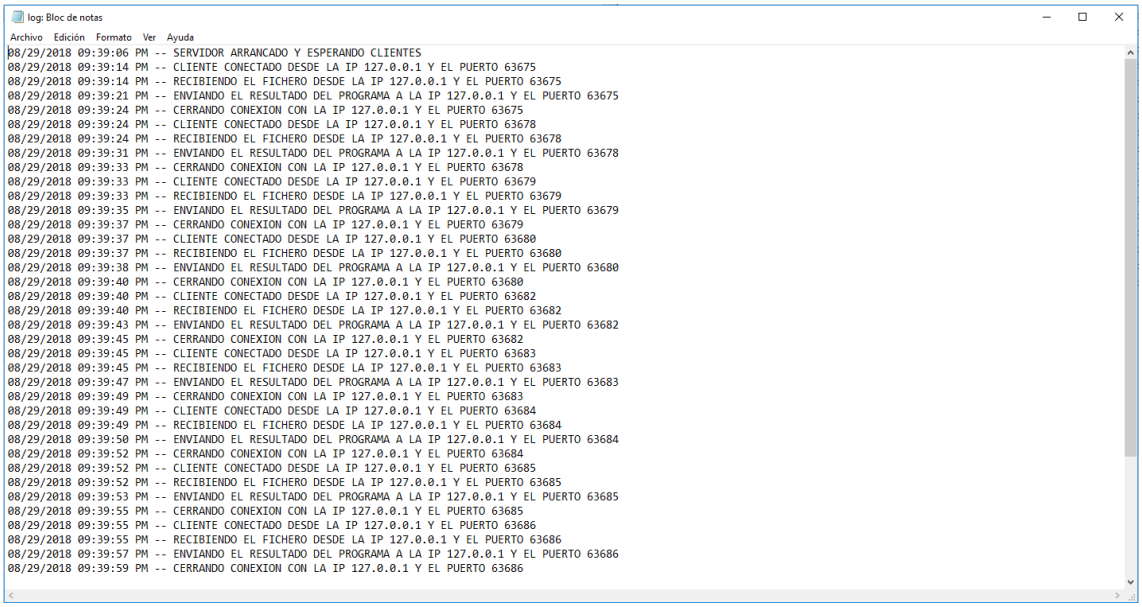
Una vez terminado esto, ya tendríamos todo el entorno funcionando y listo para comenzar a explotar nuestra aplicación, como veremos en los siguientes manuales.

4.10.2 Manual de Administrador

Una vez instalada el entorno en el servidor, simplemente queda arrancar el script "servidor.py" en una terminal de Anaconda Prompt y observar todo el flujo del servidor en la pantalla:



Como también hemos mencionado en el punto donde explicábamos la funcionalidad final del servidor, el log del servidor también tiene mucha utilidad cuando surge algún problema y que hay que depurar para ver que ha podido ocurrir, en este log aparecen los mismos mensajes que en el terminal donde hemos ejecutado el servidor añadiendo fecha y hora de cuando se produjeron:

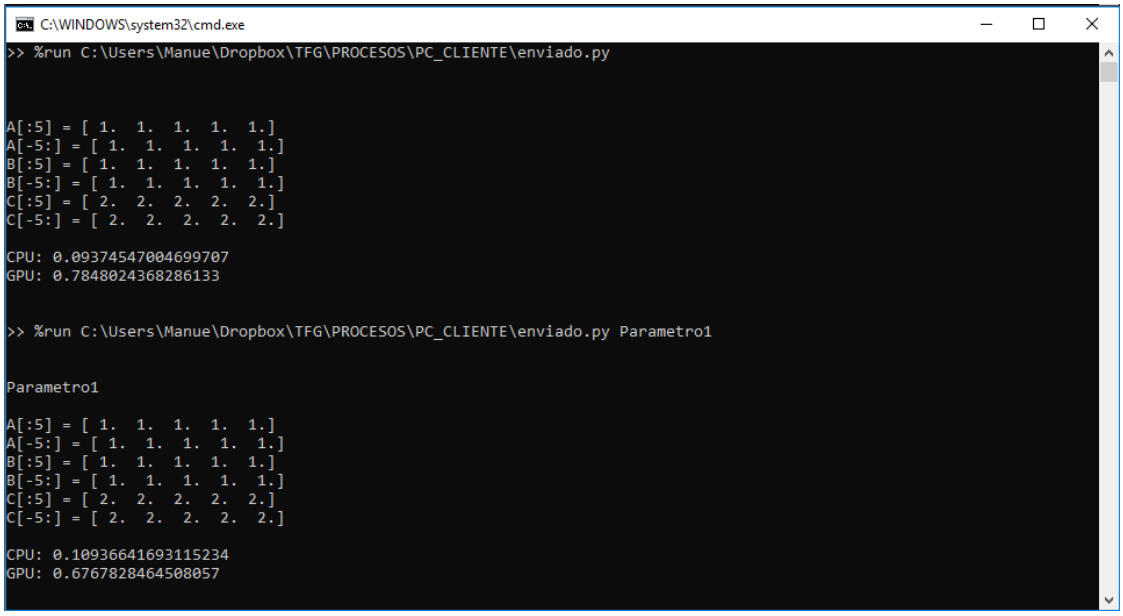


4.10.3 Manual de Cliente

Todo el desarrollo de la aplicación está enfocado en que este manual sea lo más sencillo posible y que el feedback que tenga el usuario sea lo más similar posible a que está ejecutando el script de Python con CUDA en su propia máquina.

Para que el usuario pueda conectarse al servidor y utilizar la aplicación no necesita tener instalado en su ordenador ningún entorno de desarrollo de Python ni de CUDA, simplemente debe ejecutar el archivo “cliente.exe” que hemos generado, y tener un script de Python (programado sobre algún editor avanzado como Sublime Text o simplemente sobre el mismo Bloc de Notas) en algún directorio accesible de su equipo.

En el apartado anterior donde se explica la funcionalidad final del programa del cliente, podemos ver un ejemplo de cómo sería una ejecución normal del cliente. Si el script de Python a ejecutar necesita de algún input para su correcto funcionamiento, deberemos añadirlo/s después del path al script en cuestión:



Como podemos observar también en la imagen de arriba, una vez enviamos un script para su ejecución y obtengamos el resultado, se nos brinda la posibilidad de volver a enviar otra vez otro archivo (puede ser el mismo, el mismo con modificaciones o uno diferente).

5 Coste del Proyecto

5.1 Presupuesto de Ejecución Material

Se denomina coste total de la ejecución material, a la suma del coste de los equipos y del coste por tiempo de trabajo.

5.1.1 Coste de Equipos

Supone el coste de comprar los equipos y servicios necesarios para llevar a cabo el proyecto. En nuestro caso hemos necesitado los servicios que se muestran en la siguiente tabla:

EQUIPO	PRECIO	DURACIÓN	USO	TOTAL
Ordenador	800 €	3 años	2.5 meses	56 €
Conexión a Internet	26 €	-	2.5 meses	52 €
Impresora	60 €	3 años	0.25 meses	0,42 €
Geforce GTX 760	265 €	3 años	2.5 meses	18,55 €

Coste total de los equipos	126,97 €

5.1.2 Coste del Tiempo de Trabajo

Suponiendo que el salario de un ingeniero en informática es de 12 euros la hora, el coste por tiempo de trabajo es el que se obtiene en esta tabla:

FUNCIÓN	NUMERO DE HORAS	EUROS/HORA	TOTAL
Ingeniero Informático	250	12	3.000 €

Coste total del tiempo de trabajo	3.000 €

5.1.3 Coste de Ejecución Material

Es la suma de los importes del coste de materiales y de la mano de obra que hemos calculado previamente.

CONCEPTO	COSTE
Coste de equipos	126,97 €
Coste por tiempo de trabajo	3.000 €
Coste de ejecución material	3.126,97 €

5.2 Gastos Generales y Beneficio Industrial

Normalmente se trata de los gastos necesarios para disponer de instalaciones en las que desempeñar el trabajo, además de otros gastos adicionales. Los gastos generales y el beneficio industrial son el resultado de aplicar un recargo del 20% sobre el Coste Total de Ejecución Material, resultando:

Gastos Generales y Beneficio Industrial..... 625,39 €

5.3 Presupuesto de Ejecución por Contrata

El presupuesto de ejecución por contrata es el resultado de sumar el coste total de ejecución y los gastos generales.

Presupuesto de Ejecución por Contrata..... 3.752,36 €

5.4 Presupuesto de Ejecución por Contrata

Los honorarios facultativos por la ejecución de este proyecto se determinan de acuerdo a las tarifas de los honorarios de los ingenieros en trabajos particulares vigentes a partir del 1 de Septiembre de 1997, dictadas por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación.

IMPORTE	COEFICIENTE REDUCTOR	PORCENTAJE
Hasta 5 millones	C = 1	7 %
Desde 5 millones	C = 0,9	7 %

Los derechos del visado se calcularán aplicando la siguiente fórmula: 0,07 × P × C, donde P es el presupuesto de ejecución material y C es el coeficiente reductor.

DERECHOS DE VISADO	COSTE
0,07 × 3.752,36 × 1	262,67 €
Importe total de los honorarios:	262,67 €

5.5 Importe Total del Presupuesto

El importe total del presupuesto de este proyecto se calcula sumando el presupuesto de ejecución por contrata y los honorarios. A dicho valor se le aplicará el 21% de I.V.A.

Presupuesto de Ejecución por Contrata	3.752,36 €
Honorarios	262,67 €
SUBTOTAL	4.015,03 €
21% de I.V.A.	843,16 €
IMPORTE TOTAL	4.858,19 €

Finalmente, el importe total del proyecto asciende a la cantidad de CUATRO MIL OCHOCIENTOS CINCUENTA Y OCHO EUROS CON DIECINUEVE CENTIMOS.

6 Resumen, Conclusiones y Trabajos Futuros

6.1 Resumen

El proyecto desde su inicio tiene como objetivo el permitir a los usuarios ejecutar código CUDA de manera remota en un servidor remoto con GPU y que posteriormente puedan visualizar el resultado de esta ejecución en su propia máquina.

Las tecnologías que se utilizan son Python y CUDA, las cuales unidas a librerías como NumbaPro otorgan una gran variedad de opciones y herramientas con las que conseguir el objetivo que se ha planteado.

Una de las primeras funcionalidades pensadas para el proyecto fue la de implementar un sistema cliente-servidor, que permitiese en un principio la comunicación entre un solo cliente y un solo servidor. Y que permitiese al cliente ejecutar la aplicación sin contar con tecnologías Python y CUDA en su máquina. Posteriormente y viendo que la aplicación va a ser utilizada por un gran número de usuarios y previsiblemente de manera simultánea, se adaptó la arquitectura ya creada para soportar la comunicación entre varios clientes y un solo servidor.

Una vez la comunicación entre clientes y servidores es paralela y estable, hubo que implementar un algoritmo que permitiera el envió de ficheros por parte del cliente y atendiendo a las limitaciones de memoria que tiene nuestro sistema, ya que en caso de sobrecargar la conexión esta se caerá y no permitirá que el resto de la aplicación funcione correctamente. Para este propósito se fragmenta el archivo en enviar en el equipo de cliente y se reconstruye el mismo una vez llega al servidor. Con esto conseguimos enviar archivo sea cual sea su tamaño.

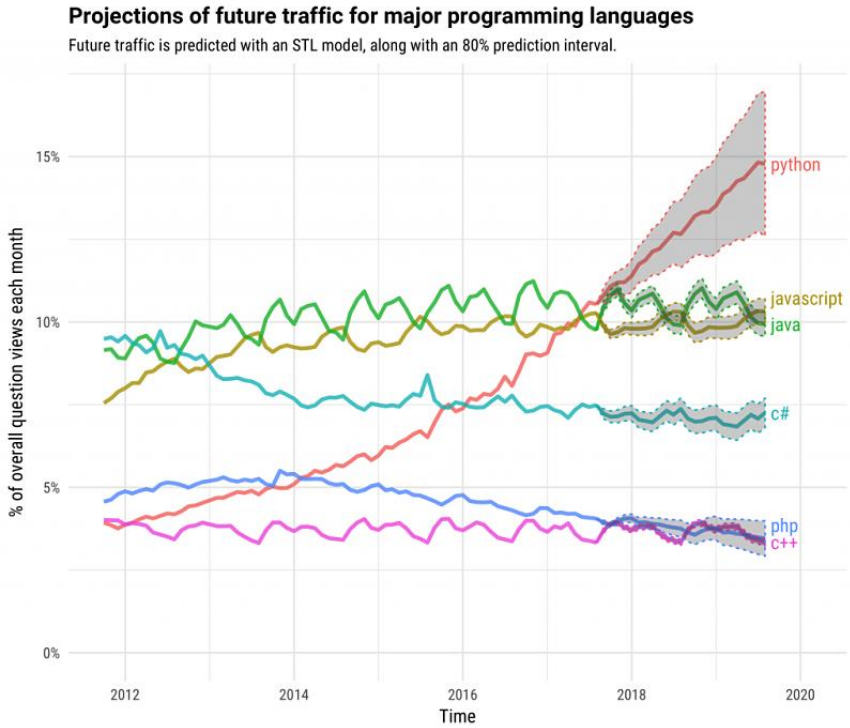
Finalmente hubo que tener en cuenta el cómo realizar la ejecución del archivo recibido en el servidor, atendiendo a los inputs y outputs que pueda tener. Por un lado los inputs son enviados por el cliente acompañando al archivo como un parámetro más, y el output final (tanto el resultante de una ejecución exitosa como errónea) es capturado y enviado por el servidor al cliente correspondiente. Todo esto se hace monitorizando el tráfico del servidor para poder tener información con la que actuar en caso de caída o error en alguna conexión.

6.2 Conclusiones

Una vez acabado el proyecto, hemos podido comprender mejor dos tecnologías que están muy de moda actualmente.

Por un lado la tecnología CUDA, la cual está permitiendo que la transición del tradicional computamiento en CPU a la nueva computación compartida entre la CPU y la GPU. Permite explotar las características de esta última, a la hora de ejecutar algoritmos que típicamente han sido ejecutados por la CPU, y que debido a su gran paralelismo son susceptibles de actualizarse para ejecutar en GPU.

Y por otro lado Python, sin duda el lenguaje de programación que más ha crecido en los últimos tiempos como podemos ver esta gráfica que incluso, se atreve a estimar su crecimiento hasta 2020, cuando superara a lenguajes como Java y C#:



Finalmente hemos comprendido la necesidad que está surgiendo en los desarrolladores de comenzar a diseñar e implementar aplicaciones usando cualquiera de estas dos tecnologías, juntas o separadas. En esa necesidad radica el porqué de este proyecto.

6.3 Trabajos Futuros

Este proyecto abre la posibilidad a crear, en base a él, mejores versiones del concepto.

La principal mejora que puede realizarse sería la posibilidad de interactuar con el servidor de una manera más dinámica, esto es, recibir los mensajes de que devuelva la aplicación que estamos ejecutando de manera inmediata. Actualmente se reciben todos los outputs resultado de la ejecución al final y, si bien esto no supone un gran inconveniente a la hora de desarrollar y probar nuevo software, sí que hace que la experiencia del usuario sea menos similar a la que tendría si ejecutase ese mismo programa en su propia máquina.

Otra de las acciones que se puede llevar a cabo, es la de mejorar en la forma que se monitorea el servidor. Ya que actualmente solo se está utilizando un archivo de texto en el que se graba, con fecha y hora, las principales interacciones que tiene cada cliente con el servidor.

Finalmente, y aunque no sea una mejora exactamente dentro del proyecto, se podría actualizar la GPU con la que cuenta el servidor por una mejor. Aunque actualmente contamos con un hardware bastante potente y que otorga un buen rendimiento. Pero Nvidia sigue innovando y, por ejemplo, acaba de lanzar su nueva serie GeForce GTX 2000 con tarjetas más potentes y actualizadas que por supuesto son compatibles con tecnología CUDA.

7 Bibliografía

Para la realización de este proyecto se ha utilizado la siguiente bibliografía:

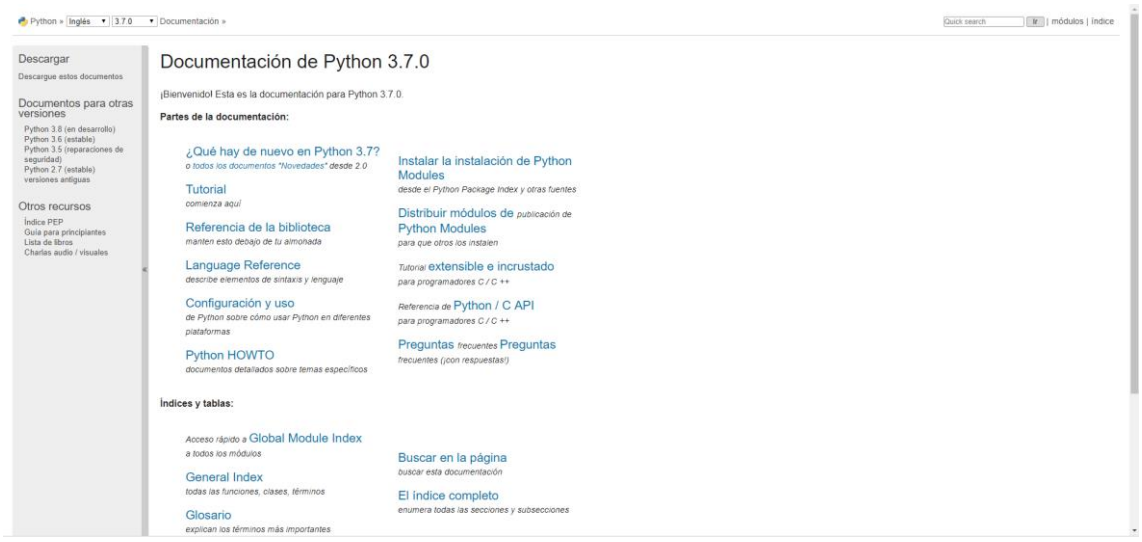
- https://devblogs.nvidia.com/numba-python-cuda-acceleration/

- https://es.wikipedia.org/wiki/Cliente-servidor

- http://www.nvidia.es/object/cuda-parallel-computing-es.html

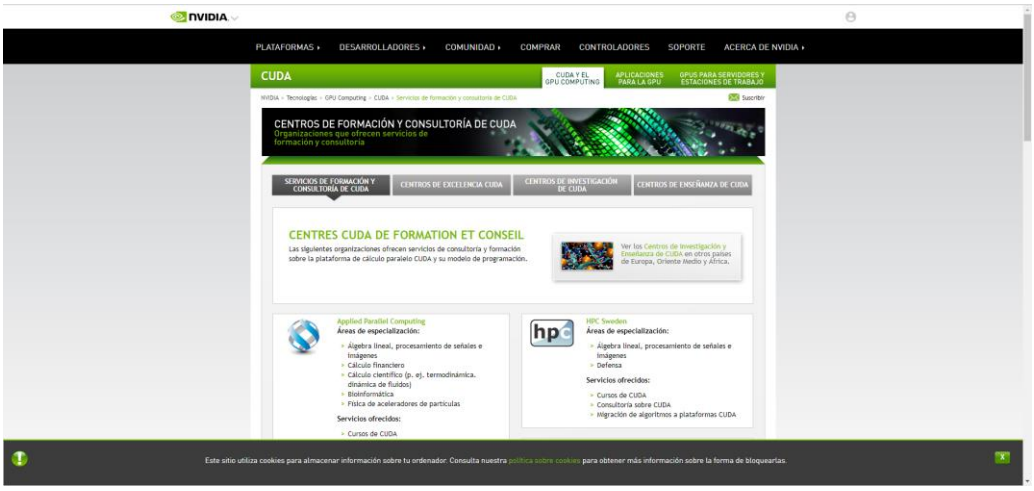
- http://www.nvidia.es/object/gpu-applications-es.html

- https://docs.python.org/3/

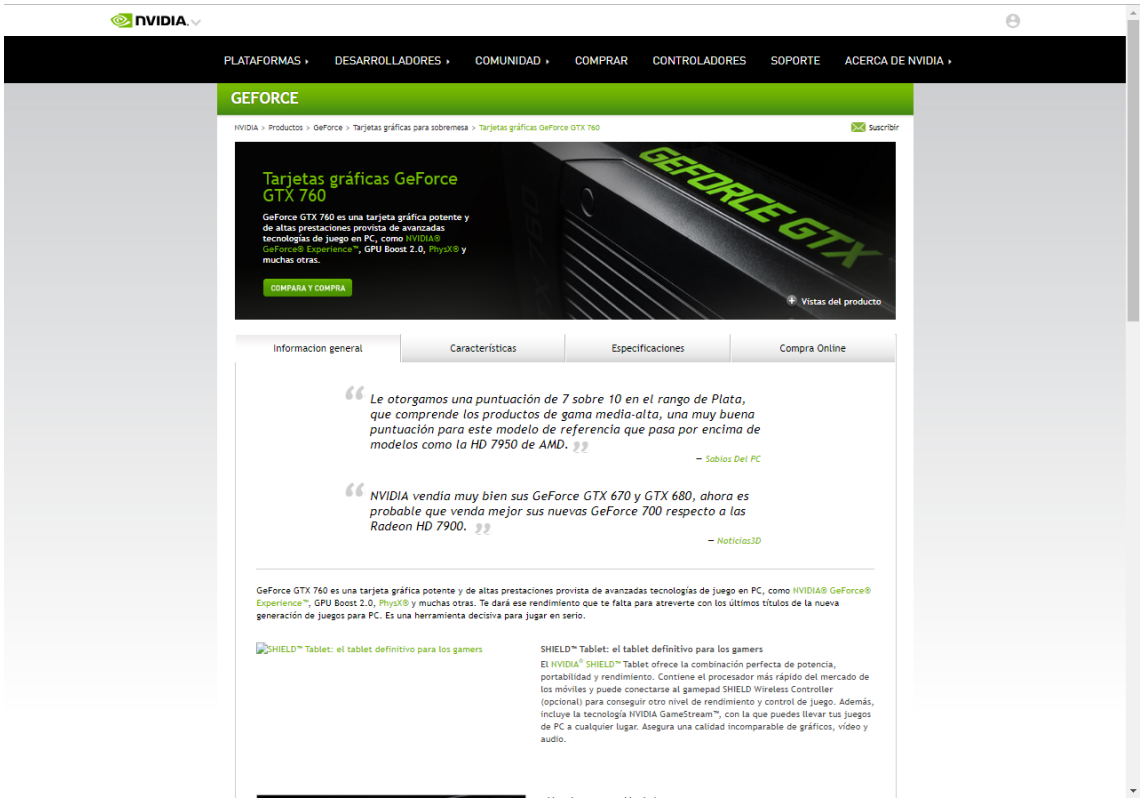


- https://developer.nvidia.com/how-to-cuda-python

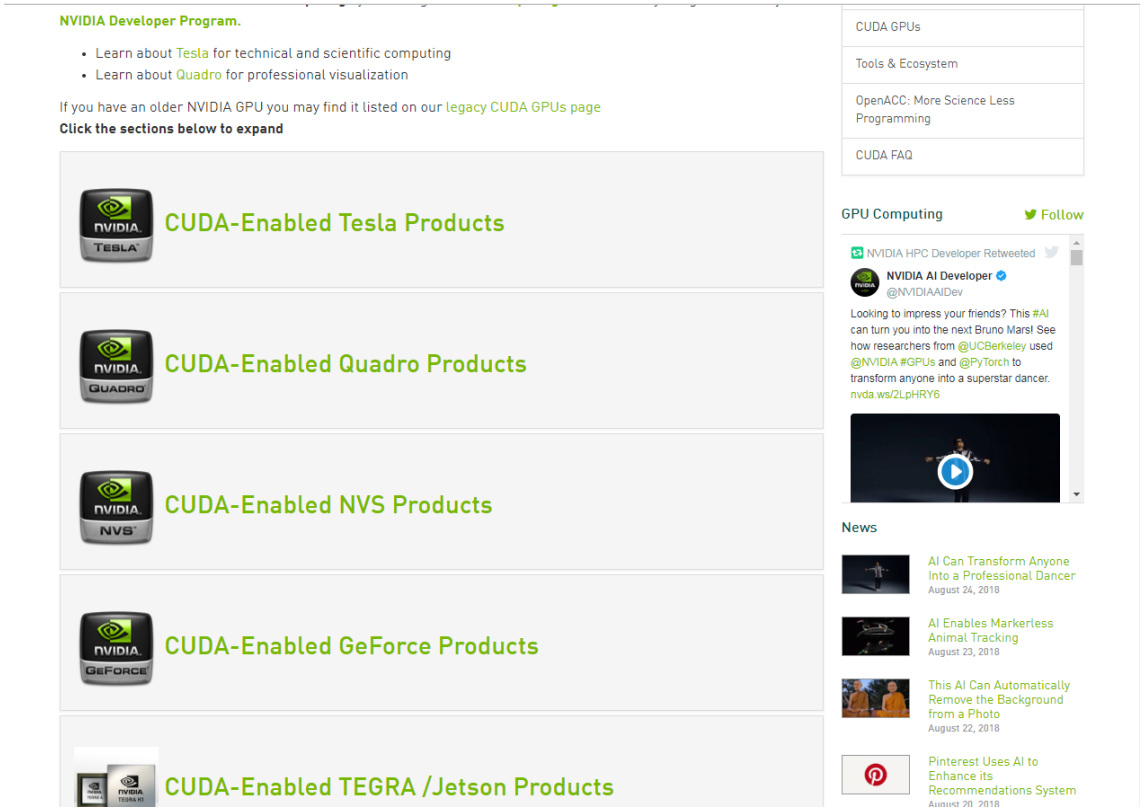
- http://www.nvidia.es/object/cuda-training-services-es.html



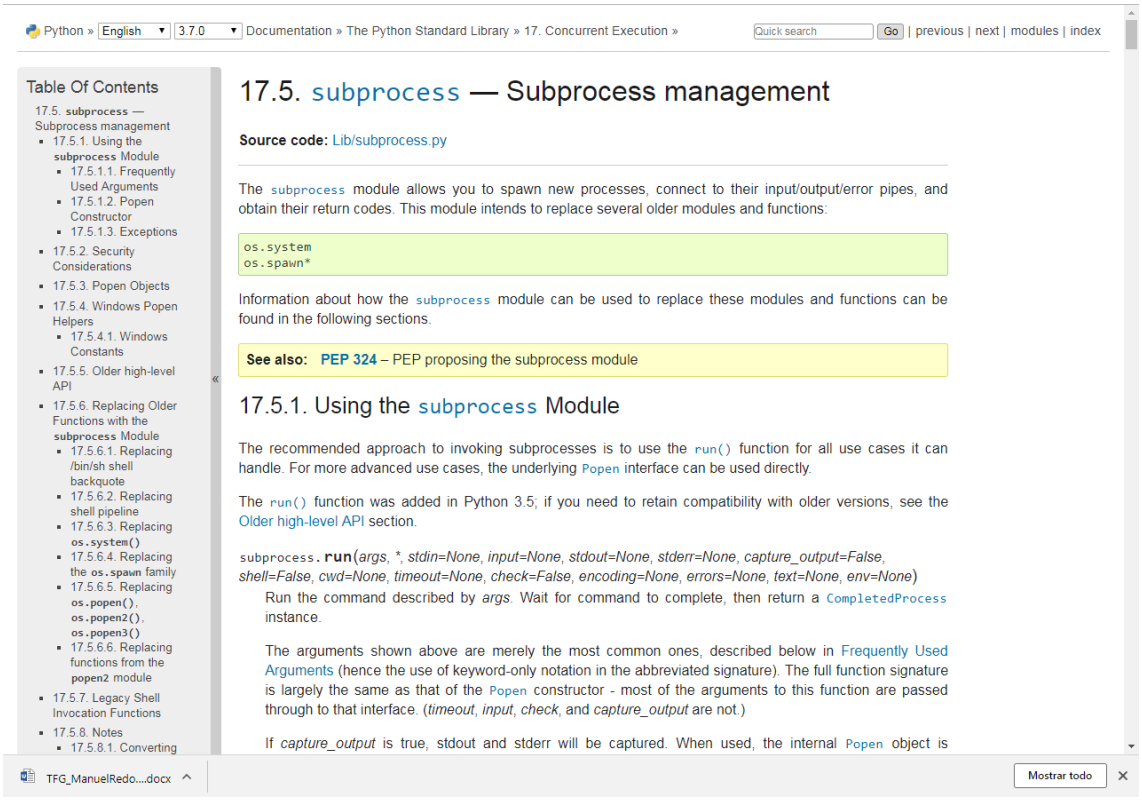
- https://www.nvidia.es/object/geforce-gtx-760-es.html#pdpContent=0



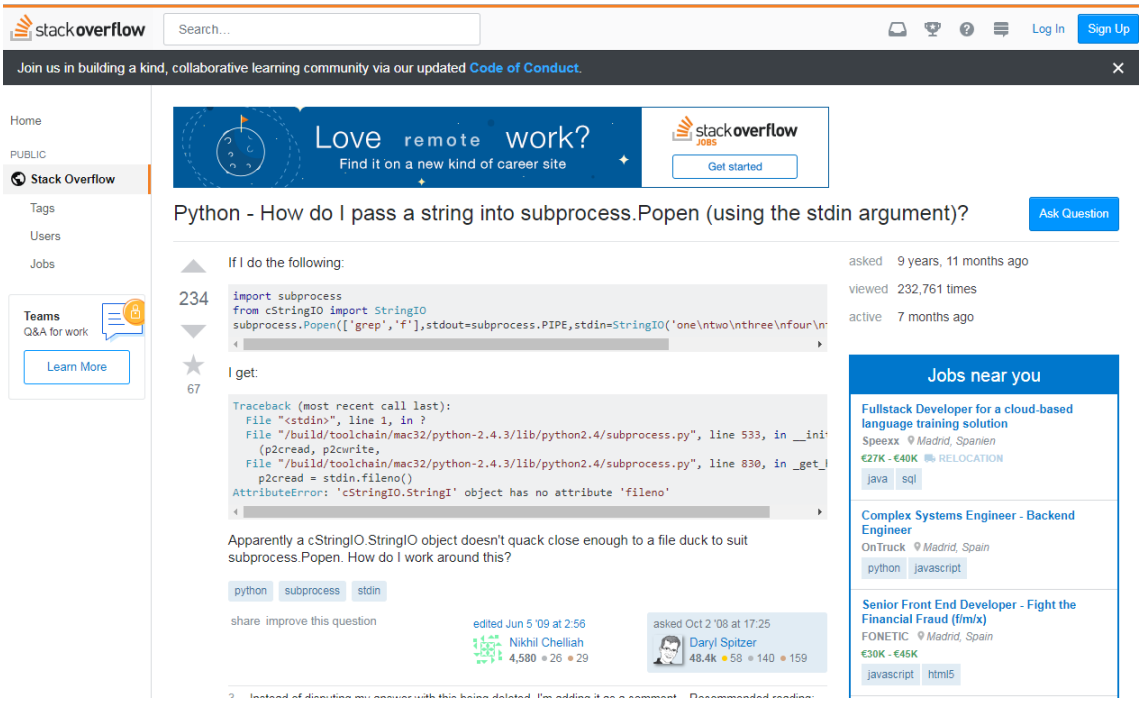
- https://developer.nvidia.com/cuda-gpus



- https://docs.python.org/3/library/subprocess.html

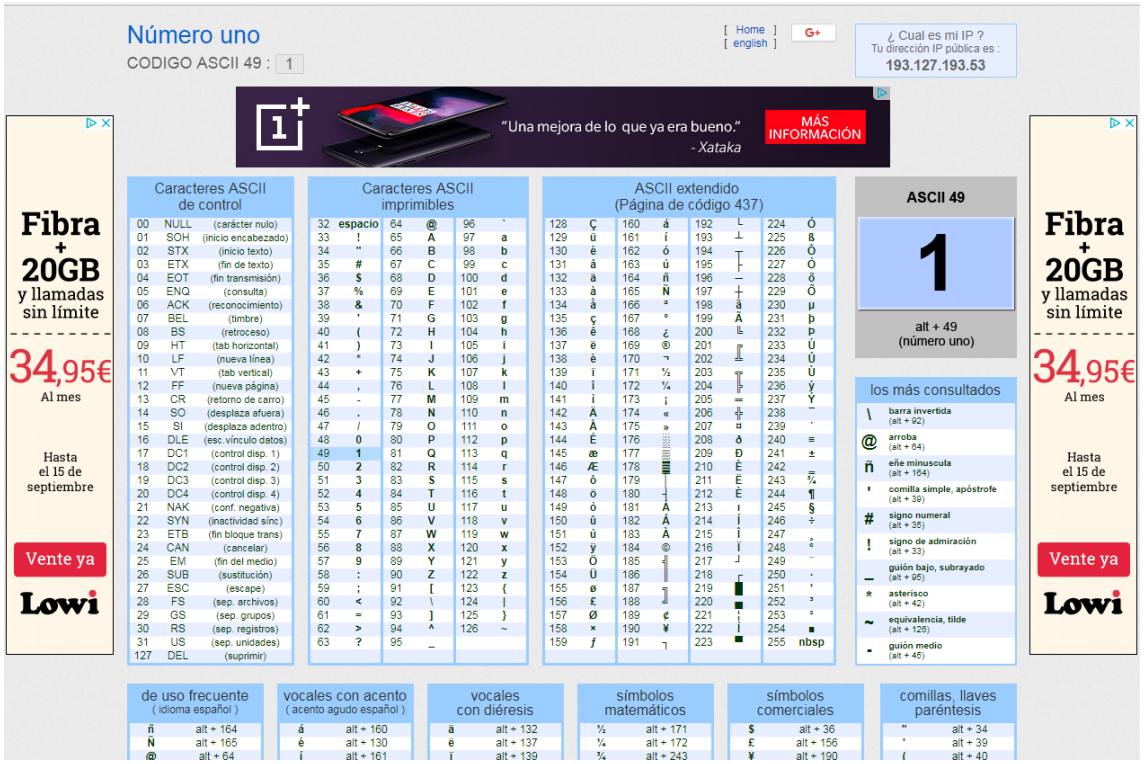


- https://stackoverflow.com/questions/163542/python-how-do-i-pass-a-string-into-subprocess-popen-using-the-stdin-argument

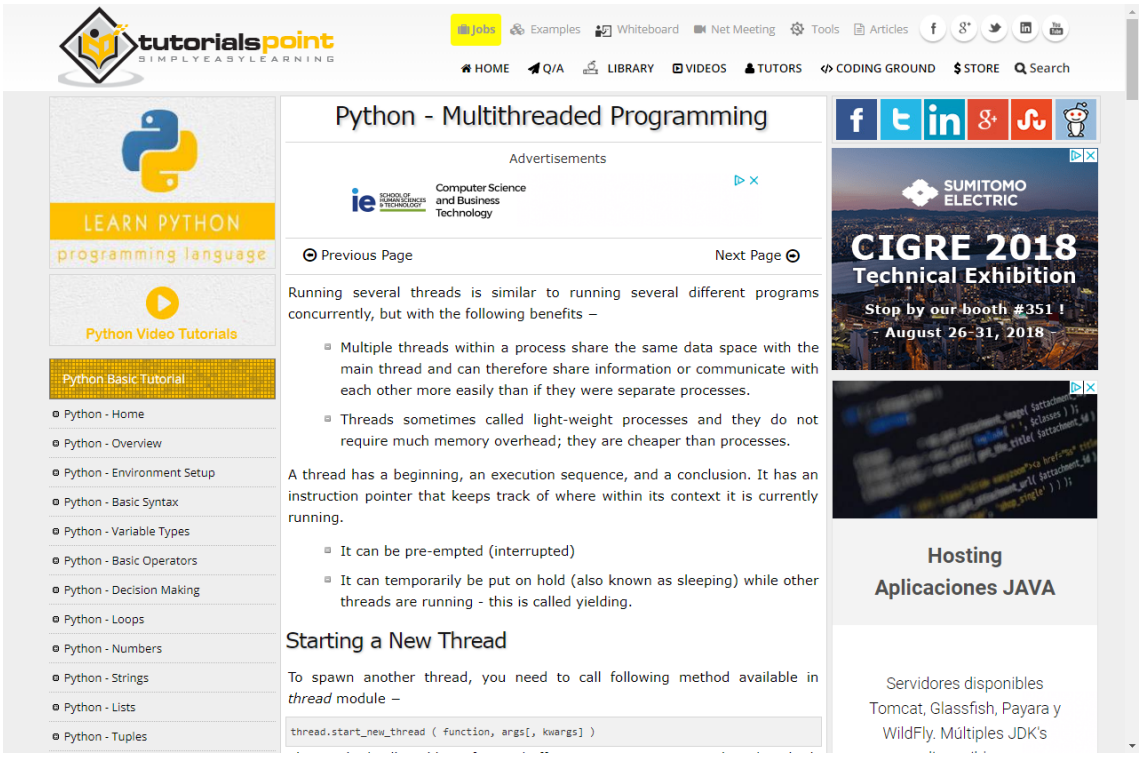


- https://recursospython.com/codigos-de-fuente/enviar-archivo-via-socket/

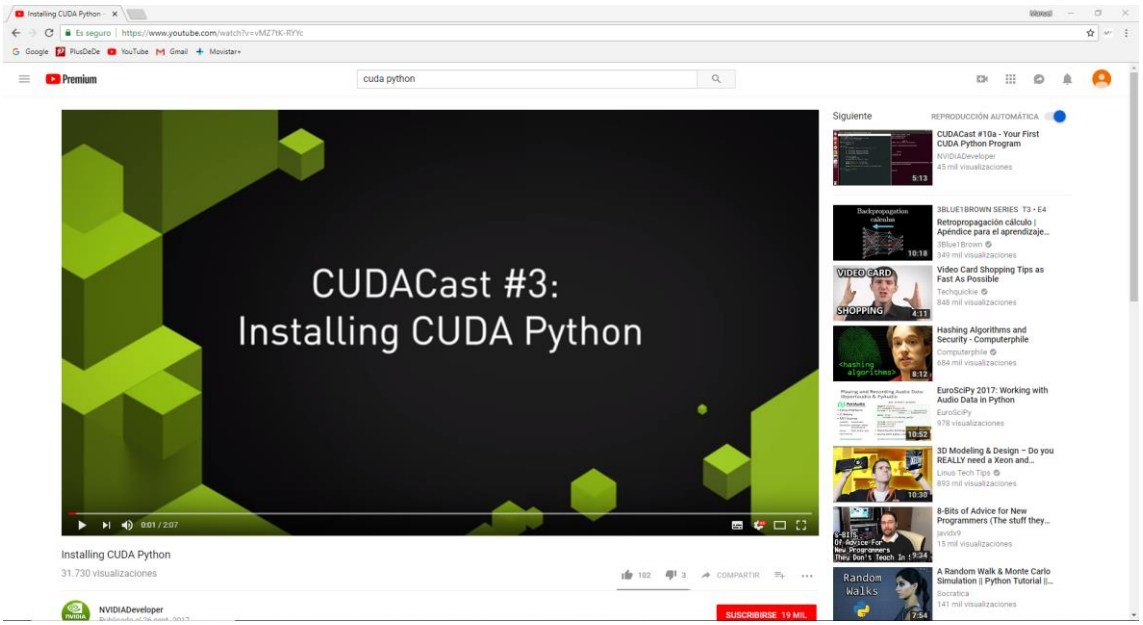
- https://elcodigoascii.com.ar/codigos-ascii/numero-uno-1-codigo-ascii-49.html



- https://www.tutorialspoint.com/python/python_multithreading.htm



- https://www.youtube.com/watch?v=vMZ7tK-RYYc



8 Anexos

8.1 Código del Script SERVER.py

```
import subprocess
import threading
import logging
import socket

logging.basicConfig(filename='log.log',level=logging.DEBUG, format='%(asctime)s --
%(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')

socket_servidor = socket.socket()
socket_servidor.bind(("localhost", 9999))
socket_servidor.listen(128)

logging.info("SERVIDOR ARRANCADO Y ESPERANDO CLIENTES")
print("SERVIDOR ARRANCADO Y ESPERANDO CLIENTES")

def ejecutar(socket_cliente):

    logging.info("RECIBIENDO EL FICHERO DESDE LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))
    print("RECIBIENDO EL FICHERO DESDE LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))

    python_script = open(str(address[0]) + "_" + str(address[1]) + ".py", "wb")

    while(1):

        try:

            input_data = socket_cliente.recv(1024)

        except socket.error:

            break

        else:

            if input_data[0] == 1:

                break

            else:

                python_script.write(input_data)


    python_script.close()

    input_data = socket_cliente.recv(1024)

    if input_data[0] == 1:

        input_data = ""

    logging.info("ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))
```

```
    print("ENVIANDO EL RESULTADO DEL PROGRAMA A LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))

    process = subprocess.Popen(["python", str(address[0]) + "_" + str(address[1]) +
".py"], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    out = process.communicate(input=input_data)[0]
    socket_cliente.send(out)
    socket_cliente.close()

    logging.info("CERRANDO CONEXION CON LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))
    print("CERRANDO CONEXION CON LA IP %s Y EL PUERTO %s" %(address[0],address[1]))

while(1):

    socket_cliente, address = socket_servidor.accept()

    logging.info("CLIENTE CONECTADO DESDE LA IP %s Y EL PUERTO %s"
%(address[0],address[1]))
    print("CLIENTE CONECTADO DESDE LA IP %s Y EL PUERTO %s" %(address[0],address[1]))

    hilo = threading.Thread(target=ejecutar, args=(socket_cliente,))
    hilo.start()

socket_servidor.close()
```

8.2 Código del Script CLIENTE.py

```
# IMPORTAMOS LA LIBRERIA "SOCKET" PARA TRABAJAR CON SOCKETS
import socket

while(1):

    # ABRIMOS EL SOCKET CLIENTE
    socket_cliente = socket.socket()

    # CONECTAMOS CON EL SOCKET SERVIDOR
    socket_cliente.connect(("localhost", 9999))

    # ESPERAMOS A QUE EL USUARIO INTRODUZCA LA RUTA DEL FICHERO A ENVIAR
    lista = input(">> %run ").split() #
C:\Users\Manue\Dropbox\TFG\PROCESOS\PC_CLIENTE\enviado.py

    # ABRIMOS, ENVIAMOS AL SERVIDOR Y CERRAMOS EL FICHERO

    while(1):

        python_script = open(lista[0], "rb")
        row = python_script.readline()

        while row:

            socket_cliente.send(row)
            row = python_script.readline()

        break

    python_script.close()
    socket_cliente.send(bytes(chr(1), "utf-8"))
```

```
    # ENVIAMOS LOS ARGUMENTOS AL SERVIDOR

    if(len(lista) > 1):

        socket_cliente.send(bytes("\n".join(lista[1:]), "utf-8"))

    else:

        socket_cliente.send(bytes(chr(1), "utf-8"))

    # RECIBIMOS DEL SERVIDOR EL OUTPUT DE EJECUTAR EL FICHERO ENVIADO
    input_data = socket_cliente.recv(1024)

    # IMPRIMIMOS EL OUTPUT EN LA PANTALLA DEL CLIENTE
    print("\n" + input_data.decode("utf-8"))

    # CERRAMOS EL SOCKET CLIENTE
    socket_cliente.close()
```

8.3 Código del Script ENVIADO.py

```
from numba import vectorize
import numpy as np
import time

def VectorAddCPU(a, b):

    return a + b

@vectorize(['float32(float32, float32)'], target='cuda')
def VectorAddGPU(a, b):

    return a + b

A = np.ones(32000000, dtype=np.float32)
B = np.ones(32000000, dtype=np.float32)

startGPU = time.time()
C = VectorAddGPU(A, B)
endGPU = time.time()

startCPU = time.time()
C = VectorAddCPU(A, B)
endCPU = time.time()

print("")
print(input())
print("")
print("A[:5] = " + str(A[:5]))
print("A[-5:] = " + str(A[-5:]))
print("B[:5] = " + str(B[:5]))
print("B[-5:] = " + str(B[-5:]))
print("C[:5] = " + str(C[:5]))
print("C[-5:] = " + str(C[-5:]))
print("")
print("CPU: " + str(endCPU - startCPU))
print("GPU: " + str(endGPU - startGPU))
print("")
```

8.4 Código del Script SETUP.py

```
from cx_Freeze import setup, Executable

base = None

executables = [Executable("cliente.py", base=base)]

packages = ["socket"]

options = {

    'build_exe': {

        'packages':packages,

    },

}

setup(

    name = "<CLIENTE>",
    options = options,
    version = "<1.0>",
    description = '<TFG>',
    executables = executables

)
```

Universidad de Alcalá
Escuela Politécnica Superiorl

ESCUELA POLITECNICA
SUPERIOR