Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Comparativa del rendimiento de una GPU usando OpenMP, OpenACC o CUDA mediante un ejemplo de dinámica molecular

Autor: Jorge Riaño de la Cruz

Tutor(a): Santiago Rodríguez de la Fuente

Madrid, junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: COMPARATIVA DEL RENDIMIENTO DE UNA GPU USANDO OPENMP, OPENACC O CUDA MEDIANTE UN EJEMPLO DE DINÁMICA MOLECULAR

Julio 2024

Autor: Jorge Riaño de la Cruz

Tutor: Santiago Rodríguez de la Fuente

Depto. de Arquitectura y Tecnología de Sistemas Informáticos ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

La idea central de este TFG es comparar el uso y rendimiento de una GPU y una CPU cuando se programa con diferentes técnicas (OpenACC, OpenMP o CUDA) para acelerar aplicaciones computacionalmente costosas. La programación de GPU y CPU se puede hacer con el uso de directivas tales como son OpenMP y OpenACC, que permiten mantener el código original hasta cierto punto, en cambio, el lenguaje CUDA, el cual requiere un mayor trabajo a la hora de modificar el código para adaptarlo a la forma en que este lenguaje funciona, solo permite la programación en GPU. En este TFG no solo se realizarán análisis en las diferencias de rendimiento de cada versión, sino que también se medirá el tiempo necesario para desarrollar y depurar cada una de ellas. Para ello, se usarán dos ejemplos sencillos, fáciles y conocidos, aunque computacionalmente costosos, los cuales serán el algoritmo de cálculo del número pi y, un simulador simple de dinámica molecular.

Abstract

The main idea of this TFG is to compare the use and performance of a GPU and a CPU when programming with different techniques (OpenACC, OpenMP or CUDA) to accelerate computationally expensive applications. GPU and CPU programming can be done both with the use of directives such as OpenMP and OpenACC, which allow to keep the original code to some extent, while CUDA language, which requires more work to modify the code to adapt it to the way this language works, only allows GPU programming. In this dissertation we will not only analyze the performance differences of each version, but we will also measure the time required to develop and debug each of them. For this, two simple, easy and known examples will be used, although computationally expensive, which will be the algorithm for calculating the number pi and a simple molecular dynamics simulator.

Índice

Introducción

Quisiera comenzar contextualizando nuestro trabajo de fin de grado, el cual hemos llevado a cabo de manera colaborativa junto a otros tres compañeros, cada uno con su respectivo tutor. Durante las primeras etapas, nos dedicamos en equipo a explorar y comprender las diferentes directivas mediante la ejecución de programas que calculaban el número pi, proporcionados por nuestros profesores. Tras lograr resultados consistentes, cada uno de nosotros se enfocó en su propia área de simulación, siendo la mía la dinámica molecular.

La dinámica molecular es conocida por ser una aplicación que demanda una ejecución prolongada debido a la intensa carga de trabajo que implica, especialmente en entornos distribuidos. El objetivo central de nuestro proyecto ha sido optimizar estos códigos, buscando incrementar su velocidad mediante técnicas de paralelización y vectorización. Además, exploraremos el uso de unidades de procesamiento gráfico (GPU) para potenciar aún más el rendimiento, en comparación con las tradicionales unidades de procesamiento central (CPU).

Para llevar a cabo esta tarea, nos enfocaremos en dos directivas clave de paralelización: OpenMP, que nos permite paralelizar bucles y funciones de manera eficiente, y OpenACC, ideal para la paralelización en entornos que incluyen tanto CPU como GPU. También aprovecharemos la plataforma CUDA, diseñada para aprovechar al máximo las capacidades de las GPU. Estas herramientas nos brindarán una amplia gama de opciones para optimizar y comparar nuestros resultados, buscando siempre mejorar la eficiencia y el rendimiento de nuestras simulaciones.

Estado del arte

A continuación, voy a hacer un breve resumen a modo de introducción de cada una de las directivas de las que voy a hacer uso durante este trabajo.

OpenMP

OpenMP (Open Multi-Processing) es una API (Interfaz de Programación de Aplicaciones) que facilita la programación de aplicaciones con memoria compartida para sistemas multiproceso. Su objetivo principal es permitir a los programadores agregar paralelismo de manera sencilla a programas escritos en lenguajes como C, C++ y Fortran, con el fin de mejorar el rendimiento en sistemas con múltiples núcleos de procesamiento.

Historia: Los inicios de OpenMP se remontan a 1997, cuando Intel formó la OpenMP Architecture Review Board (OpenMP ARB) para desarrollar estructuras de repetición en paralelo en el lenguaje Fortran. Desde entonces, ha evolucionado hasta convertirse en un estándar ampliamente adoptado en la comunidad de programadores.

Uso: OpenMP es sumamente útil para tareas que requieren procesar un gran volumen de información, ya que permite reducir los tiempos de carga mediante el uso de hilos. Algunos campos que se benefician de esta tecnología incluyen el supercómputo, el análisis de procesamiento de imágenes 3D, el procesamiento de datos y el cálculo matemático.

Soporte: La API de OpenMP incluye un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que permiten a los programadores especificar cómo paralelizar el código. Por ejemplo, se pueden agregar directivas en el código para indicar bucles que deben ejecutarse en paralelo, distribuir tareas entre múltiples hilos y gestionar la sincronización entre ellos.

OpenMP es portátil y escalable, lo que significa que las aplicaciones desarrolladas con esta API pueden ejecutarse en una amplia variedad de plataformas, desde computadoras de escritorio hasta supercomputadoras.

OpenACC

OpenACC (Open Accelerators) es un estándar de programación diseñado para facilitar la computación paralela en sistemas heterogéneos compuestos por CPU y GPU. Este estándar permite a los programadores de lenguajes como C, C++ o Fortran utilizar directivas del compilador para acelerar el rendimiento del código, de manera similar a OpenMP. El modelo de ejecución de OpenACC incluye cuatro niveles de paralelismo: thread, gang, worker y vector. Los hilos individuales se denominan threads, los grupos de hilos se conocen como gangs, los conjuntos de hilos que pueden trabajar conjuntamente en operaciones SIMD se llaman workers, y los hilos que operan en paralelo en instrucciones SIMD se denominan vectors.

Historia: Originalmente, OpenACC tenía la intención de ser compatible con todos los aceleradores, pero se centró en las GPUs de Nvidia. Su lanzamiento inicial se realizó en 2011 durante la conferencia de Supercomputación. Las versiones posteriores, como la 2.0 en 2013, mejoraron el soporte para estructuras de datos complejas. Las versiones 2.5 y 2.7 ampliaron la compatibilidad y corrigieron errores. Las versiones 3.0, 3.1 y 3.2, lanzadas entre 2019 y 2021, aumentaron la eficiencia y la flexibilidad de la plataforma.

Empresas relacionadas: NVIDIA habilita las directivas de OpenACC en sus compiladores para GPUs Nvidia. Cray Inc ofrece tecnología para acelerar el rendimiento en aplicaciones exigentes. The Portland Group (PGI) es un proveedor de compiladores paralelos de alto rendimiento. CAPS Enterprise es un proveedor de soluciones para la migración e implementación de aplicaciones en procesadores de muchos núcleos

Soporte: Compiladores comerciales como PGI, Cray, y CAPS ofrecen soporte para OpenACC. Está disponible en IDEs para C/C++ y Fortran.

Uso: OpenACC beneficia a campos como química, biología, física y análisis de datos, donde se procesan grandes cantidades de datos rápidamente. Por ejemplo, en la predicción del clima, donde se requiere un procesamiento rápido de grandes conjuntos de datos para obtener predicciones útiles sin colapsar el sistema de cómputo. Aunque nosotros lo utilizaremos en el tema de la dinámica molecular.

CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de computación en paralelo desarrollada por Nvidia, que permite a los programadores utilizar una variante del lenguaje C (CUDA C) para codificar algoritmos en las GPU de Nvidia. Se puede utilizar mediante wrappers para Python, Fortran, Julia y Java. Funciona en todas las GPU Nvidia de la serie G8X en adelante.

El objetivo de CUDA es aprovechar el paralelismo ofrecido por las GPU frente a las CPU de propósito general, gracias a sus múltiples núcleos que permiten el lanzamiento de un gran número de hilos simultáneos. El primer SDK se publicó en febrero de 2007 y actualmente es compatible con Windows, Linux y macOS.

Historia: CUDA fue lanzado en 2006 como la primera solución de computación general en GPU. Ian Buck, líder del equipo que presentó Brook en 2003, se unió a Nvidia y lideró el lanzamiento de CUDA. El SDK se hizo público en 2007, inicialmente para Windows y Linux, con soporte para la micro-arquitectura Tesla. La relación con Apple tuvo conflictos, y CUDA perdió soporte en macOS Mojave.

Empresas relacionadas: Nvidia, Cray Inc, The Portland Group (PGI), CAPS Enterprise.

Soporte de compilador: CUDA cuenta con soporte en compiladores comerciales como los de PGI, Cray, y CAPS, así como en compiladores de código abierto y académicos.

Uso: CUDA se utiliza en una amplia gama de campos, incluyendo animación, astrofísica, big data, bioinformática, modelado climático, síntesis de voz, entre otros. Las aplicaciones varían desde el renderizado de gráficos en motores como Unreal Engine hasta la simulación de dinámica molecular y el análisis de grandes conjuntos de datos.

Características del hardware

Para la realización de este trabajo, voy a hacer uso de tres servidores distintos (albaricoque, espino y limonero), para poder tener más tiempos entre los que comparar.

Albaricoque

```
jriano@albaricoque:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  24
  On-line CPU(s) list:   0-23
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
    CPU family:          6
    Model:               63
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           2
    Stepping:            2
    CPU max MHz:         3200.0000
    CPU min MHz:         1200.0000
    BogoMIPS:            4789.22
```

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.239.06   Driver Version: 470.239.06   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:03:00.0 Off |                  N/A |
| 26%   33C    P8    14W / 250W |     20MiB /  6082MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  Off  | 00000000:04:00.0 Off |                  N/A |
| 30%   34C    P8    13W / 250W |      9MiB /  6083MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA GeForce ...  Off  | 00000000:81:00.0 Off |                  N/A |
| 26%   36C    P8    14W / 250W |      9MiB /  6083MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2074      G   /usr/lib/xorg/Xorg                 10MiB |
|    0   N/A  N/A      3345      G   /usr/bin/gnome-shell                3MiB |
|    1   N/A  N/A      2074      G   /usr/lib/xorg/Xorg                  3MiB |
|    2   N/A  N/A      2074      G   /usr/lib/xorg/Xorg                  4MiB |
+-----------------------------------------------------------------------------+
```

En el caso de albaricoque, tenemos tres GPUs, si recorremos los valores de izquierda a derecha podemos observar primero, el uso del ventilador, el siguiente valor sería la temperatura de la GPU, después nos encontramos al estado de rendimiento de la GPU, el consumo de energía es el siguiente valor, y se indica tanto el que se está usando, como el valor máximo, si continuamos,

nos encontramos el uso de memoria gráfica, tanto la que está en uso, como la máxima posible y ya por último estaría el uso de la GPU, que en mi caso, he tomado la captura mientras no se estaba utilizando ninguna.

Limonero

```
jriano@limonero:~$ lscpu
Arquitectura:                         x86_64
  modo(s) de operación de las CPUs:   32-bit, 64-bit
  Address sizes:                      46 bits physical, 48 bits virtual
  Orden de los bytes:                 Little Endian
CPU(s):                               24
  Lista de la(s) CPU(s) en línea:     0-23
ID de fabricante:                     GenuineIntel
  Nombre del modelo:                  Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
    Familia de CPU:                   6
    Modelo:                           63
    Hilo(s) de procesamiento por núcleo: 2
    Núcleo(s) por «socket»:           6
    «Socket(s)»                       2
    Revisión:                         2
    CPU MHz máx.:                     3200,0000
    CPU MHz mín.:                     1200,0000
    BogoMIPS:                         4789.55
```

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2070        Off |   00000000:04:00.0 Off |                  N/A |
|  0%   44C    P8             26W /  175W |      14MiB /   8192MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:81:00.0 Off |                  N/A |
| 39%   48C    P8             22W /  260W |       5MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      1630      G   /usr/lib/xorg/Xorg                             9MiB |
|    0   N/A  N/A      2107      G   /usr/bin/gnome-shell                           3MiB |
|    1   N/A  N/A      1630      G   /usr/lib/xorg/Xorg                             4MiB |
+-----------------------------------------------------------------------------------------+
```

En el caso de albaricoque, tenemos solamente dos GPUs, si recorremos los valores de izquierda a derecha podemos observar primero, el uso del ventilador, el siguiente valor sería la temperatura de la GPU, después nos encontramos al estado de rendimiento de la GPU, el consumo de energía es el siguiente valor, y se indica tanto el que se está usando, como el valor máximo, si continuamos, nos encontramos el uso de memoria gráfica, tanto la que está en uso, como la máxima posible y ya por último estaría el uso de la GPU, que en mi caso, he tomado la captura mientras no se estaba utilizando ninguna.

Espino

```
jriano@espino:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         40 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  24
  On-line CPU(s) list:   0-23
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
    CPU family:          6
    Model:               44
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           2
    Stepping:            2
    Frequency boost:     enabled
    CPU max MHz:         2401.0000
    CPU min MHz:         1600.0000
    BogoMIPS:            4800.13
```

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce GTX TITAN X     Off | 00000000:08:00.0 Off |                  N/A |
| 24%   61C    P0              76W / 250W |      0MiB / 12288MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce GTX TITAN X     Off | 00000000:81:00.0 Off |                  N/A |
| 25%   61C    P0              74W / 250W |      0MiB / 12288MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
|   2  NVIDIA GeForce GTX TITAN X     Off | 00000000:82:00.0 Off |                  N/A |
| 28%   63C    P0              61W / 250W |      0MiB / 12288MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

En el caso de espino, tenemos tres GPUs también, y como en los dos servidores anteriores, si recorremos los valores de izquierda a derecha podemos observar primero, el uso del ventilador, el siguiente valor sería la temperatura de la GPU, después nos encontramos al estado de rendimiento de la GPU, el consumo de energía es el siguiente valor, y se indica tanto el que se está usando, como el valor máximo, si continuamos, nos encontramos el uso de memoria gráfica, tanto la que está en uso, como la máxima posible y ya por último estaría el uso de la GPU, que en mi caso, he tomado la captura mientras no se estaba utilizando ninguna.

Herramientas de análisis

Para realización de la optimización de los códigos he hecho uso de tres herramientas de análisis.

Nvprof:

Nvprof es una herramienta de perfilado de NVIDIA que permite a los desarrolladores analizar el rendimiento de aplicaciones CUDA. Su principal objetivo es identificar cuellos de botella y oportunidades de optimización en el código que se ejecuta en la GPU. A continuación, explicaré brevemente los beneficios de esta herramienta:

Optimización del rendimiento: Ayuda a identificar y corregir ineficiencias en el código CUDA.

Mejora del uso de recursos: Permite maximizar el uso de la GPU, mejorando el rendimiento general de la aplicación.

Análisis detallado: Proporciona información granular sobre el comportamiento del código en la GPU, lo cual es esencial para optimizaciones avanzadas.

nvidia-smi:

La siguiente herramienta seria nvidia-smi. nvidia-smi (NVIDIA System Management Interface) es una herramienta de línea de comandos que proporciona información sobre el estado y la configuración de las GPUs NVIDIA. Es utilizada para monitorear y gestionar el rendimiento y la utilización de las GPUs en tiempo real. A continuación, los beneficios de esta:

Monitoreo en tiempo real: Proporciona una vista en tiempo real del estado y el rendimiento de las GPUs.

Prevención de problemas: Ayuda a prevenir el sobrecalentamiento y otros problemas de hardware mediante la monitorización constante.

Gestión eficiente: Permite a los administradores de sistemas y desarrolladores gestionar de manera eficiente los recursos de la GPU.

VTune:

Por último, he utilizado VTune, Intel VTune Profiler es una herramienta de análisis de rendimiento que ayuda a los desarrolladores a optimizar el rendimiento de aplicaciones en sistemas basados en CPUs Intel. Permite el perfilado detallado de aplicaciones, identificando cuellos de botella y oportunidades de optimización. Y los beneficios de esta herramienta serían:

Optimización detallada: Proporciona un análisis profundo del rendimiento de la CPU, ayudando a identificar y solucionar cuellos de botella.

Soporte multiplataforma: Permite el análisis de aplicaciones en diferentes sistemas operativos y arquitecturas de hardware.

Mejora de la concurrencia: Ayuda a optimizar aplicaciones multihilo, mejorando la eficiencia y el rendimiento en sistemas multiprocesador.

Códigos

Debido a la complejidad del tema, primero se han realizado pruebas con un ejemplo de un código muy sencillo, el cual aparece en toda la bibliografía, que es el cálculo del número pi. Este ejemplo se realiza como una toma de contacto para ir adaptándonos tanto al uso de las directivas, como para perfeccionar el uso correcto de las flags de compilación de cada versión. Las primeras semanas nos llevo bastante tiempo elegir los flags correctos y actualizados, pero una vez elegidos y comprobado su correcto funcionamiento, se han usado esos mismos flags sin hacer más cambios significativos, que más adelante serán indicados en cada versión. La siguiente parte trata del código de dinámica molecular el cual he tenido que implementar sus diferentes versiones, ya con todas las flags definidas correctamente.

Aunque hayamos ejecutado el código en todos los servidores y comparado los resultados, los tiempos mostrados serán los obtenidos en limonero, ya que hemos decidido que es el que mejor funciona y menos problemas da, pero en el caso de obtener unos resultados muy distintos en alguno de los otros servidores, se indicará.

Códigos número pi

Para este programa, tenemos un código en C que calcula el valor aproximado de pi utilizando el método de integración numérica conocido como "método de rectángulos" o "método de Riemann". El cálculo se realiza dividiendo el intervalo [0, 1] en un gran número de subintervalos pequeños y aproximando el área bajo la curva de la función $f(x) = \frac{4}{1+x^2}$ mediante la suma de áreas de rectángulos.

Para la toma de tiempos y cálculos de speedUps vamos a tomar en cuenta 2 ejecuciones, una en la que tomaremos el número de pasos como 2^30 -1 que será una ejecución de prueba, y otra que será el número de pasos de 2^33-1, la cual, será una ejecución más grande, por lo tanto, más lenta y costosa.

Para ejecutar estos códigos habrá que utilizar estos comandos:

Ejecución prueba: ./ejecutable 1

Ejecución grande: ./ejecutable 4

SECUENCIAL

Para la compilación de este código hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando: gcc-11 -O3

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador: pgcc -O3 -fast -Minfo=all -mp

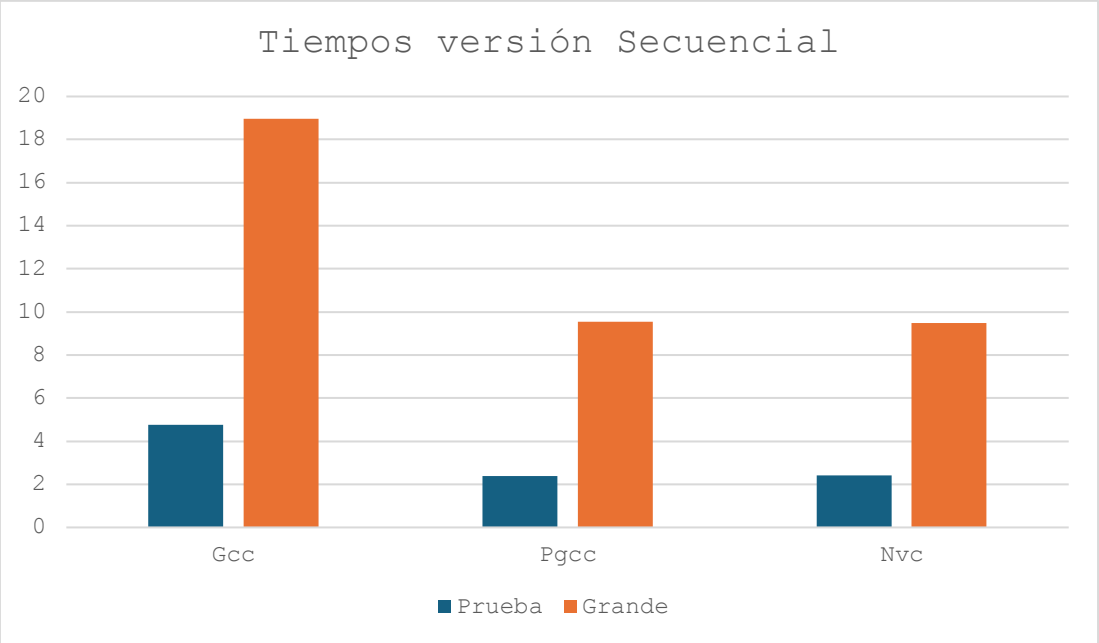
Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando: nvc -O3 -fast -Minfo=all -mp

Este código en C calcula una aproximación del valor de pi utilizando el método de integración numérica de rectángulos. Primero, se incluyen las bibliotecas necesarias y se definen algunas constantes y variables. Luego, el programa verifica si se proporciona un argumento en la línea de comandos para ajustar el número de pasos en el cálculo. Después de calcular el número total de pasos, se inicia un bucle para calcular la suma acumulativa de áreas bajo la curva de la función anteriormente explicada. Una vez completado el bucle, se calcula el valor aproximado de pi y se mide el tiempo de ejecución. Finalmente, se imprimen los resultados en pantalla, incluyendo el valor aproximado de pi, el número de pasos utilizado y el tiempo de ejecución, junto con el valor real de pi y el error relativo.

Los resultados de este código son:

Ejecución prueba	Gcc	Pgcc	Nvc
Secuencial	4,77	2,38	2,42

Ejecución grande	Gcc	Pgcc	Nvc
Secuencial	18,95	9,56	9,50



OPENMP

CPU:

Para la compilación de este código en la versión de CPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenmp

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fopenmp -fast -Minfo=all

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fopenmp -fast -Minfo=all

En comparación con la versión secuencial, esta versión utiliza OpenMP para paralelizar el bucle principal del cálculo. La directiva “`#pragma omp parallel for simd private(x) reduction(+: sum)`” indica al compilador que el bucle ‘for’ siguiente debe ser paralelizado, distribuyendo las iteraciones entre múltiples hilos de ejecución. La directiva “`private(x)`” en OpenMP indica que cada hilo en la región paralela debe tener su propia copia privada de la variable x. Esto significa que cada hilo tendrá su propia versión independiente de la variable x, que es inicializada en cada iteración del bucle paralelo y solo es accesible dentro del contexto de ese hilo específico.

```
41  //#pragma omp teams distribute parallel for simd private(x) reduction(+:sum)
42  #pragma omp parallel for simd private(x) reduction(+:sum)
43  // #pragma omp simd private(x) reduction(+:sum)
44  for (i=0;i< num_steps; i++) {
45      x = (i+0.5)*step;
46      sum += 4.0/(1.0+x*x);
47  }
48  pi = step * sum;
```

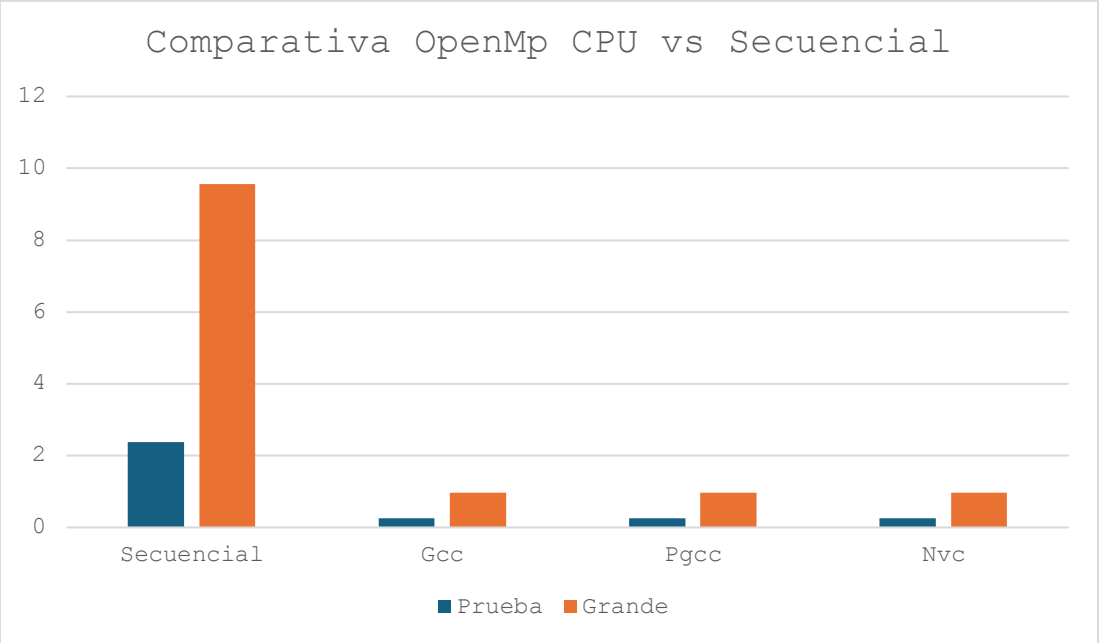
Esto evita problemas de concurrencia cuando varios hilos intentan acceder o modificar la misma variable x. Por otro lado, la directiva “`reduction(+: sum)`” especifica que se debe realizar una operación de reducción en la variable sum al final de la región paralela. La operación + indica que se debe realizar una suma de los valores individuales de sum en cada hilo paralelo. Al final de la región paralela, los valores parciales de sum se suman juntos para obtener un único resultado final. Esto asegura que la variable sum se agregue correctamente, combinando las contribuciones de todos los hilos paralelos. Cada hilo calcula una parte de las iteraciones del bucle, y luego se realiza una operación de reducción para sumar los resultados parciales y obtener el

resultado final. Esto permite que el cálculo se realice de manera más eficiente al utilizar múltiples núcleos de procesamiento simultáneamente.

Los tiempos de esta versión son considerablemente más rápidos que en la versión secuencial, obteniendo un speedUp de x9,52 en la ejecución de prueba y un speedUp de x9,89 en la ejecución grande.

Ejecución prueba	Gcc	Pgcc	Nvc
OpenMp CPU	0,26	0,25	0,26

Ejecución grande	Gcc	Pgcc	Nvc
OpenMp CPU	0,96	0,97	0,97



GPU:

Para la compilación de este código en la versión de GPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenmp -foffload=nvptx-none -foffload=-misa=sm_35 -fcf-protection=none -fno-stack-protector -no-pie

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fopenmp -fast -Minfo=all -target=gpu -gpu=cc70

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fopenmp -fast -Minfo=all -target=gpu -gpu=cc70

La principal diferencia entre esta versión y la versión secuencial radica en las directivas OpenMP utilizadas en los bucles de cálculo de pi. Mientras que la versión secuencial simplemente utiliza un bucle “for” estándar para calcular las sumas parciales, esta versión emplea directivas OpenMP específicas para offloading y paralelización en dispositivos acelerados. La directiva “`#pragma omp target map(from: sum)`” indica que el código dentro del bloque OpenMP debe ser ejecutado en un dispositivo de cómputo acelerado, como una GPU. La cláusula “`map(from: sum)`” especifica que la variable sum debe ser mapeada de vuelta al host después de que la región paralela haya finalizado su ejecución en el dispositivo.

```
22  #pragma omp target map(from:sum)
23  #pragma omp teams distribute parallel for simd private(x) reduction(+:sum)
24  // #pragma omp simd private(x) reduction(+:sum)
25  for (i=0;i< 100; i++) {
26      x = (i+0.5)*step;
27      sum += 4.0/(1.0+x*x);
28  }
29  pi = step * sum;
```

Esto significa que los cambios realizados en la variable sum durante la ejecución en el dispositivo serán reflejados en la versión de la variable en el host una vez que la ejecución haya terminado. La segunda directiva “`#pragma omp teams distribute parallel for simd private(x) reduction(+: sum)`” paraleliza un bucle for utilizando la extensión SIMD (Single Instruction, Multiple Data) para procesar varias iteraciones del bucle en paralelo. Cada hilo ejecuta su propia versión del bucle, lo que permite procesar múltiples iteraciones simultáneamente. Además, esta directiva especifica que la variable “x” debe ser privada para cada hilo, evitando problemas de concurrencia al tener cada hilo su propia copia de x. Por último, indica que se

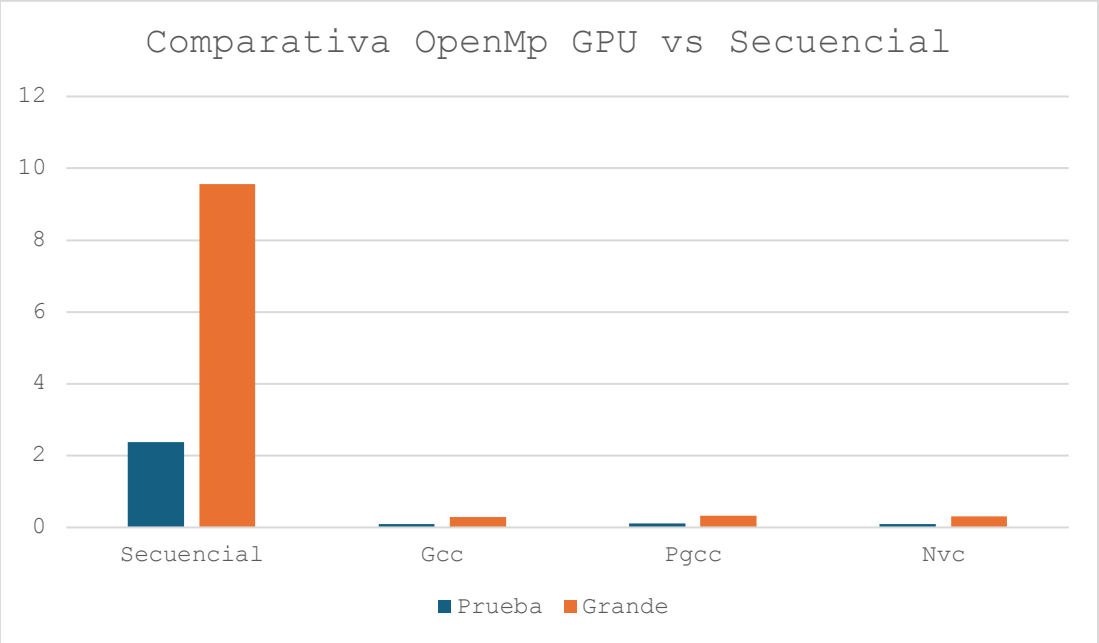
debe realizar una operación de reducción en la variable “sum” al final del bucle para obtener el resultado final de forma correcta.

```
48   #pragma omp target map(from:sum)
49   #pragma omp teams distribute parallel for simd private(x) reduction(+:sum)
50   // #pragma omp simd private(x) reduction(+:sum)
51   for (i=0;i< num_steps; i++) {
52       x = (i+0.5)*step;
53       sum += 4.0/(1.0+x*x);
54   }
55   pi = step * sum;
```

Los resultados de estas ejecuciones son incluso más rápidos que los de la versión de OpenMp en su versión en la CPU, obteniendo un speedUp de x26,44 en la ejecución de prueba y un speedUp de x32,75 en la ejecución grande.

Ejecución prueba	Gcc	Pgcc	Nvc
OpenMp GPU	0,09	0,11	0,1

Ejecución grande	Gcc	Pgcc	Nvc
OpenMp GPU	0,29	0,33	0,3



OPENACC

CPU:

Para la versión del código en OpenAcc, solo hay un código, este mismo se puede compilar tanto para la CPU como para la GPU, por lo tanto, dividiremos igualmente los tiempos como si fueran dos versiones:

Para la compilación de este código en la versión de CPU hemos utilizado estos comandos:

En openAcc no existe una compilación de gcc para la versión de CPU, solamente para la GPU, por lo tanto, en la versión de CPU solamente tendremos dos tiempos para comparar.

Para la primera utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -acc -ta=multicore

Para la segunda utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -acc -ta=multicore

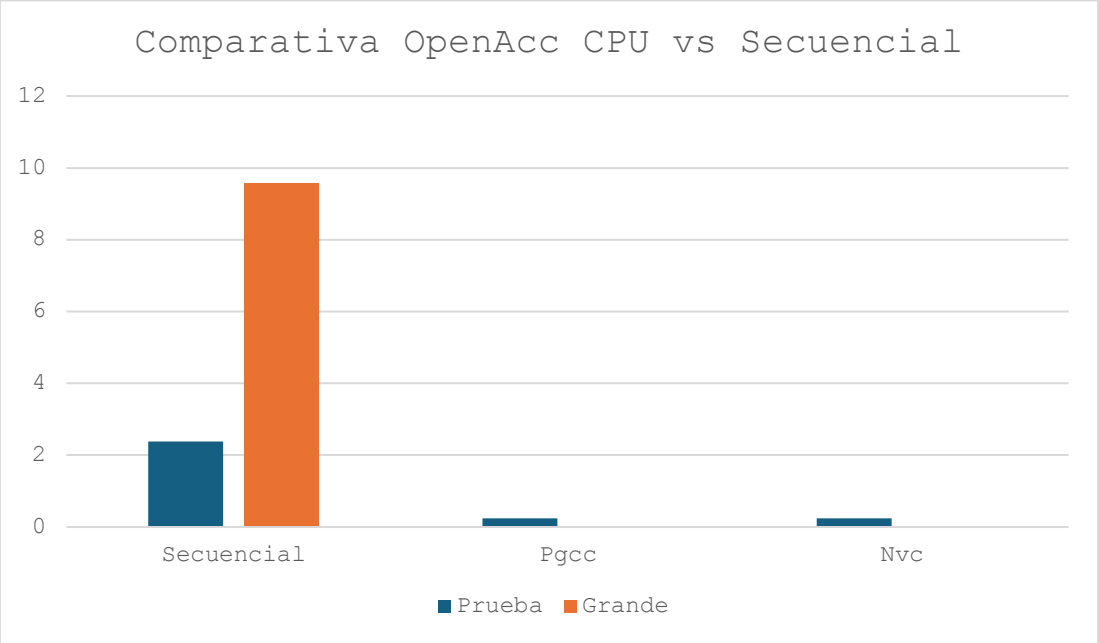
La principal diferencia entre este código y la versión secuencial radica en cómo se paraleliza el bucle de cálculo de pi. Mientras que la versión secuencial no utiliza ningún tipo de paralelización, esta versión utiliza OpenACC para la paralelización en un dispositivo acelerado, como una GPU. La directiva `“#pragma acc parallel loop private(x) reduction(+: sum)”` indica que el bucle siguiente debe ser paralelizado y ejecutado en un dispositivo de cómputo acelerado. La cláusula `“private(x)”` garantiza que cada hilo tenga su propia copia privada de la variable x, evitando problemas de concurrencia. La cláusula `“reduction(+: sum)”` especifica que se debe realizar una operación de reducción en la variable sum al final del bucle para obtener el resultado final de forma correcta, sumando los valores parciales de sum de cada hilo juntos. Dentro de este bucle paralelo, la variable “x” se declara como privada para cada hilo, lo que evita problemas de concurrencia al tener cada hilo su propia copia de “x”. Además, se realiza una operación de reducción en la variable “sum” para obtener el resultado final de forma correcta. Esto asegura que la variable sum se agregue correctamente, combinando las contribuciones de todos los hilos paralelos.

```
23    #pragma acc parallel loop private(x) reduction(+:sum)
24    for (i=0;i< 100; i++) {
25        x = (i+0.5)*step;
26        sum += 4.0/(1.0+x*x);
27    }
28    sum=0;
```

Ejecución prueba	Pgcc	Nvc
OpenAcc CPU	0,25	0,25

Ejecución grande	Pgcc	Nvc
OpenAcc CPU	No funciona	No funciona

Como podemos observar en las tablas de ejecución, este código no se ha podido realizar mediante las ejecuciones grandes, ya que estas requieren de un numero elevadísimo de cálculos, lo cual, es normal que produzca errores y no permita a la CPU realizar estas operaciones. Por lo tanto, solo podemos observar un speedUp en las ejecuciones de prueba, que es de x9,52



GPU:

Para la compilación de este código en la GPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenacc -foffload=nvptx-none -foffload=-misa=sm_35 -fcf-protection=none -fno-stack-protector -no-pie

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -acc -target=gpu -gpu=cc70

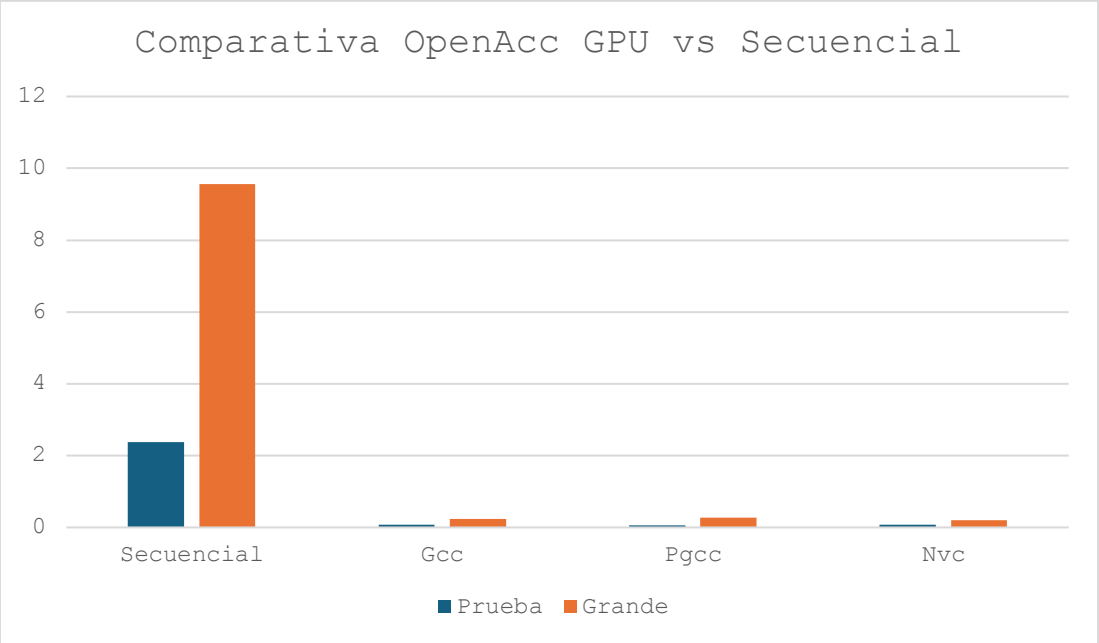
Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -acc -target=gpu -gpu=cc70

Para esta compilación, pese a que sea el mismo código, observamos un aumento considerable de velocidad al ejecutar el programa. Los speedUps obtenidos mediante esta compilación son de x39,67 para la ejecución de prueba, y de x47,5 para la ejecución grande lo cual son unos speedUps altísimos, esto se debe al uso de la GPU frente a la CPU.

Ejecución prueba	Gcc	Pgcc	Nvc
OpenAcc GPU	0,08	0,06	0,08

Ejecución grande	Gcc	Pgcc	Nvc
OpenAcc GPU	0,24	0,27	0,2



CUDA

Para la compilación de este código hemos utilizado estos comandos:

La primera compilación utilizando CUDA desde su carpeta, ejecutaremos el comando:

-arch=all

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -mp

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -mp

La diferencia principal entre este código y la versión secuencial radica en cómo se realiza el cálculo de pi y la gestión de la paralelización.

En la versión secuencial, el cálculo de pi se realiza en serie, utilizando un solo hilo de ejecución. Los pasos se recorren en un bucle for estándar y la suma de los resultados parciales se acumula en una variable.

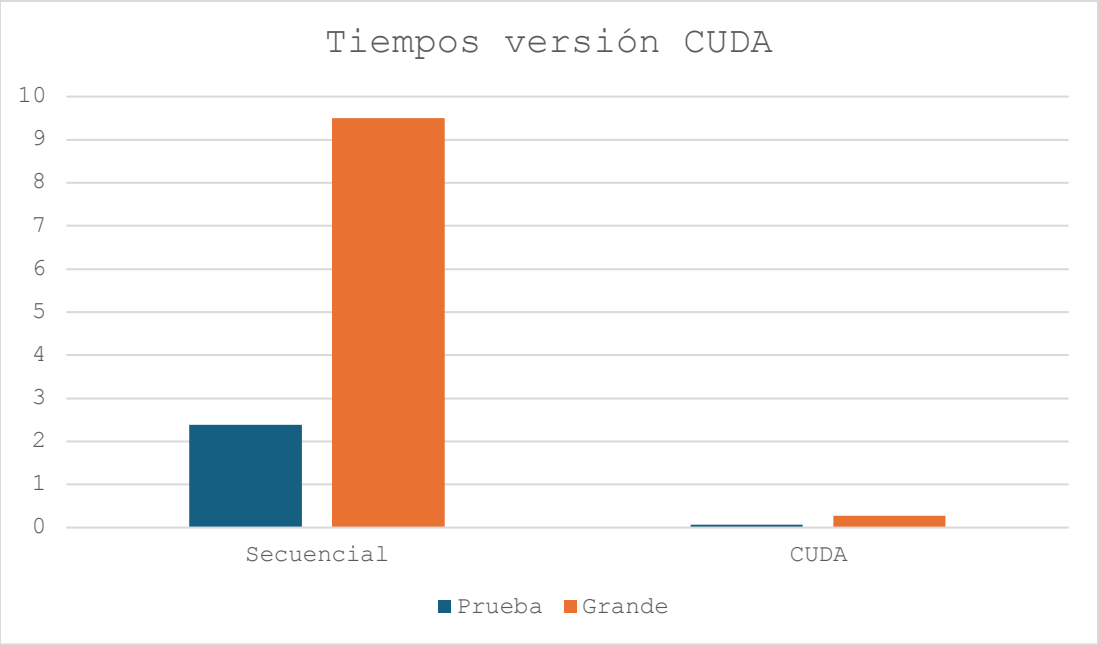
En cambio, en este código se utiliza programación en paralelo con CUDA. Se definen dos funciones kernel: “pi_gpu y reduction1”, que se ejecutarán en paralelo en la GPU. La función “pi_gpu” calcula la suma parcial de los términos de la serie de pi para cada hilo en paralelo, utilizando una estrategia de división de trabajos. Posteriormente, se realiza una reducción en paralelo utilizando la función “reduction1” para sumar todas las sumas parciales y obtener el resultado final.

Además, se utilizan directivas como “__global__” para definir funciones kernel, “blockIdx.x y threadIdx.x” para obtener el identificador del bloque y del hilo respectivamente, y “cudaMalloc, cudaMemcpy y cudaDeviceSynchronize” para gestionar la memoria y la sincronización entre el host y el dispositivo.

Para esta versión del código hemos obtenido unos resultados excelentes, obteniendo así un speedUp de x34 en la ejecución de prueba y un speedUp de x33,92 en la ejecución grande

Ejecución prueba	CUDA
CUDA	0,07

Ejecución grande	CUDA
CUDA	0,28



Conclusión

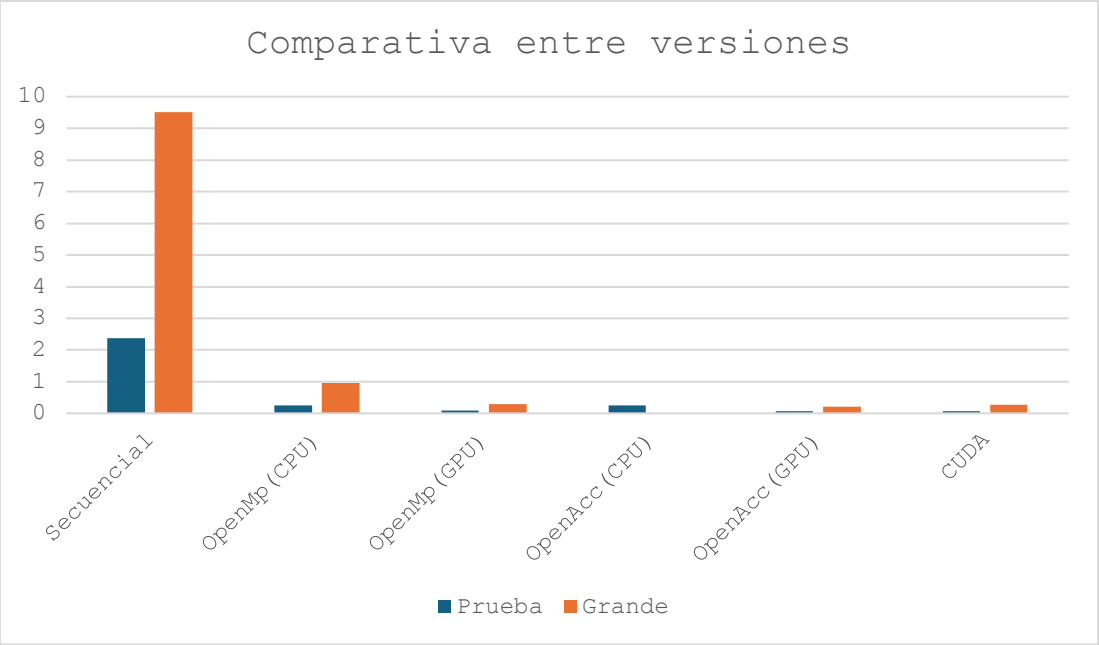
Para llegar a una conclusión, he decidido escoger los tiempos más bajos de cada versión (aunque sean iguales aproximadamente), y desde ellos realizar una conclusión lógica y con sentido. Para ello, primero utilizaré unas tablas de tiempos, como las utilizadas anteriormente y después expondré gráficamente la diferencia de los tiempos de todas, donde:

Seq = versión secuencial, Mp = versión OpenMp, Acc = versión OpenAcc,

(CPU) = versión en CPU y (GPU) = versión en GPU

Ejecución prueba	Seq	Mp(CPU)	Mp(GPU)	Acc(CPU)	Acc(GPU)	CUDA
Tiempo(segundos)	2,38	0,25	0,09	0,25	0,06	0,07

Ejecución grande	Seq	Mp(CPU)	Mp(GPU)	Acc(CPU)	Acc(GPU)	CUDA
Tiempo(segundos)	9,50	0,96	0,29	-	0,2	0,28



Como podemos observar claramente tanto en el gráfico como en las tablas, hay una grandísima diferencia entre las versiones paralelizadas y la versión secuencial, lo que refleja que la paralelización en este código es muy útil y por tanto facilita y reduce mucho el tiempo de ejecución respecto de la versión secuencial. En cuanto a las versiones paralelizadas podemos observar una

diferencia significativa entre las versiones que realizan sus cálculos en la CPU respecto a las que lo realizan en la GPU. Estas últimas podemos observar que realizan los cálculos relativamente más rápido, lo cual ayudaría a cálculos mucho más grandes, ya que, en estos tiempos tan bajos, un speedUp de x4,16 no es muy relevante, en cambio en operaciones exponencialmente más grandes puede suponer un ahorro de tiempo muy considerable. En cuanto a las versiones de cada código podemos deducir(excepto en el caso de la versión secuencial que sí que hay una clara diferencia entre la versión de gcc y las demás) que el compilador no influye en el resultado de los tiempos, porque prácticamente obtenemos el mismo tiempo en todos los casos, ya que nunca obtenemos el mismo tiempo dos veces en cada ejecución, es decir, si estuviésemos probando muchas más veces el Código, nos saldrían resultados distintos(tanto más altos como más bajos) por lo tanto deduzco que el resultado no depende tanto del compilador, sino, de la paralelización del código y de donde se realice esta si en la CPU o en la GPU.

Códigos dinámica molecular

Antes de empezar, hay que decir que el objetivo fundamental del trabajo es comparar el cálculo de un ejemplo de dinámica molecular con diferentes tecnologías. Este código implementa una simulación de dinámica molecular en C. Utiliza el esquema de integración de tiempo de Verlet para actualizar las posiciones, velocidades y aceleraciones de las partículas en el sistema. Los componentes principales incluyen la inicialización de partículas, el cálculo de fuerzas y energías, y la actualización de los estados de las partículas en cada paso de tiempo, permitiendo observar cómo evolucionan en el tiempo bajo la influencia de un potencial de interacción central.

En cuanto a los contenidos de este apartado, voy a explicar cómo van a estar estructurados los apartados que vienen a continuación.

Primero de todo, para cada versión del código, voy a indicar los comandos de compilación de cada versión con sus compiladores, en cuanto a sus carpetas, serán indicadas en este mismo apartado al final de este. Después indicaré los problemas encontrados en cada código, y las versiones realizadas hasta encontrar la versión óptima, que será la versión final del trabajo.

En cada versión comentaré las técnicas de optimización utilizadas, y por qué funcionan mejor o peor que otras. Además, en cada versión mostraré tanto en formato tabular como en formato grafico los tiempos de cada versión, con sus compiladores, comparados con el mejor tiempo de la versión secuencial indicando además su speedUp(siempre comparándolo con el menor tiempo obtenido entre los tres compiladores).

Por último, realizaré una conclusión en la cual compararé los mejores tiempos de cada versión entre sí, explicando porque se han obtenido mejores o peores tiempos en las distintas versiones.

Las carpetas en las que se encuentran los distintos compiladores son:

En Limonero

El compilador pgcc:

/opt/nvidia/hpc_sdk/Linux_x86_64/24.1/compilers/bin/pgcc

El compilador nvc:

/opt/nvidia/hpc_sdk/Linux_x86_64/24.1/compilers/bin/nvc

El compilador CUDA:

/usr/local/cuda/bin/nvcc

En albaricoque

El compilador pgcc:

/opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/pgcc

El compilador nvc:

/opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/nvc

El compilador CUDA:

/usr/local/cuda/bin/nvcc

En espino

El compilador pgcc:

/opt/nvidia/hpc_sdk/Linux_x86_64/21.3/compilers/bin/pgcc

El compilador nvc:

/opt/nvidia/hpc_sdk/Linux_x86_64/21.3/compilers/bin/nvc

El compilador CUDA:

/usr/local/cuda/bin/nvcc

Estos son los compiladores en cada servidor.

En cuanto a las ejecuciones, vamos a ejecutar el siguiente comando, que en todos los servidores es el mismo:

./ejecutable 3 4096 1000 0.1

El cual es un tamaño bastante grande y puede servirnos para observar unas mejoras en los tiempos considerablemente altas.

SECUENCIAL

Para la compilación de este código hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -mp

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -mp

El código implementa una simulación de dinámica molecular en C. Comienza incluyendo varias librerías estándar de C necesarias para realizar operaciones matemáticas, manejar entrada y salida de datos, y medir el tiempo de ejecución. Luego, declara varias funciones que serán utilizadas más adelante en el programa.

La función principal “main” recibe argumentos desde la línea de comandos o solicita al usuario que ingrese ciertos parámetros necesarios para la simulación, como la dimensión espacial (para la ejecución de pruebas 2 y para la ejecución grande 3), el número de partículas (para las pruebas 200 y para la ejecución grande 4096) y el número de pasos de tiempo (para la ejecución de pruebas 200 y para la ejecución grande 1000). Luego, asigna memoria para varias variables que almacenarán posiciones, velocidades, aceleraciones y fuerzas de las partículas.

Después de la inicialización, se ejecuta un bucle principal que avanza la simulación en el tiempo. En cada paso de tiempo, se actualizan las posiciones y velocidades de las partículas utilizando el método de “Verlet”. Luego se calculan las fuerzas y las energías potencial y cinética mediante la función compute

La función compute calcula las fuerzas entre todas las partículas y determina las energías potencial y cinética del sistema. Utiliza la función “dist” para calcular la distancia entre dos partículas en el espacio. Esta función es la que más tiempo consume, ya que es la que realiza la mayoría de los cálculos.

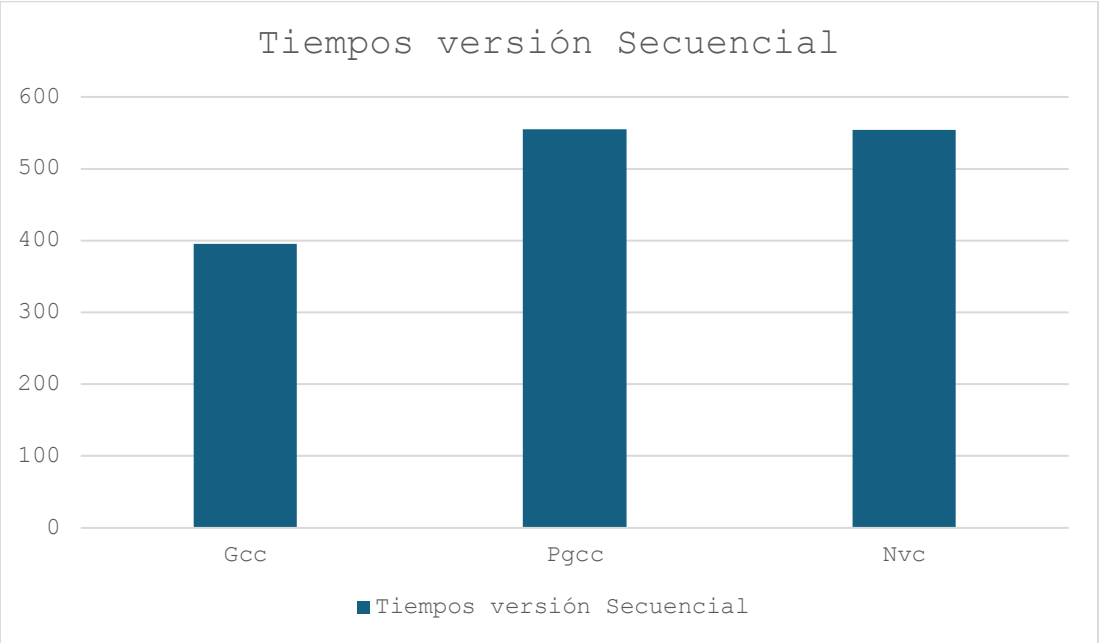
Una vez que se han calculado las energías, se imprimen los resultados y se actualiza el tiempo de impresión. Esto se repite hasta que se alcanza el número total de pasos de tiempo especificado

Finalmente, se imprime el tiempo total de ejecución de la simulación y se libera la memoria asignada dinámicamente antes de finalizar el programa. En

resumen, el código realiza una simulación secuencial de dinámica molecular, calculando las interacciones entre partículas y evolucionando el sistema en el tiempo.

Los tiempos obtenidos para este código son los que tomaremos como referencia para calcular el speedUp del resto de códigos optimizados.

Tiempo ejecución	Gcc	Pgcc	Nvc
Secuencial	395,54	555,267	553,954



He probado también a implementar una versión optimizada de este código, pero el código secuencial ya está muy bien hecho y no hay forma de optimizarlo sin utilizar paralelismo. La única parte del código que he pensado en que se puede optimizar es la parte de la función “initialize”, pero aun aprovechando la optimización de los bucles, no se consigue una mejoría en cuanto a los tiempos de ejecución. Por lo tanto, para comparar con el resto de las ejecuciones utilizaré el menor tiempo obtenido, es decir el tiempo de la compilación en gcc, la que se llevó a cabo en 395,54 segundos.

OPENMP

CPU

Para la compilación de este código en la versión de CPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenmp

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fopenmp -fast -Minfo=all

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fopenmp -fast -Minfo=all

En cuanto a la versión del código utilizando las directivas de OpenMp dirigidas hacia la CPU he de decir que ha sido la versión que he encontrado más sencilla de realizar y la que menor tiempo me costó implementar, ya que utiliza unas directivas bastante triviales y el uso de la CPU hace que la optimización de bucles del código sea rápida y sencilla. No obstante, tuve que informarme primero bastante sobre el uso de estas directivas.

Primero de todo, para poder utilizar estas directivas he tenido que incluir la librería de OpenMp al principio del código, con “#include <omp.h>”, que proporciona soporte para programación paralela en sistemas multiprocesador de memoria compartida.

Después he tenido que comprobar que funciones eran las que se tomaban más tiempo al ejecutar el programa, que obviamente, era la función compute, por lo tanto, tenía que buscar una manera de poder optimizar esta función sin cometer errores en el paso de las variables. Para ello tuve que utilizar la directiva “`#pragma omp parallel for private(i, j, d, d2, rij) reduction(+:pe,ke)`” la cual indica al compilador que el bucle “for” siguiente, debe ser ejecutado de forma paralela. Además, este indica que las variables “i, j, d, d2, rij” deben ser privadas para cada hilo de ejecución, por lo que cada hilo utilizará una copia independientemente de los otros hilos. También, la cláusula “`reduction(+:pe,ke)`” especifica que las variables “pe y ke” deben ser combinadas al final de la ejecución del bucle, sumando los resultados parciales de cada hilo.

```
163     #pragma omp parallel for private(i, j, d, d2, rij) reduction(+:pe,ke)
164     for (k = 0; k < np; k++) {
```

A continuación, se inicializan las fuerzas dentro del bucle paralelo para cada partícula y, mediante la función “dist” se calculará la distancia entre las

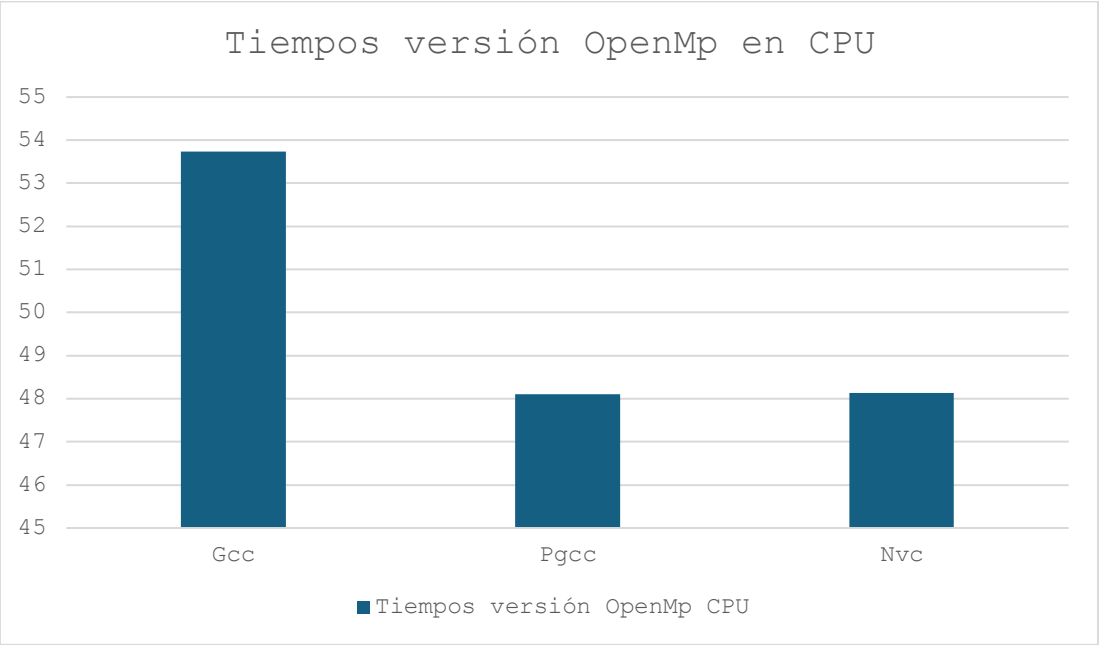
partículas “k y j” y, si estas están lo suficientemente cenca, se actualizan las fuerzas y la energía potencial “pe”.

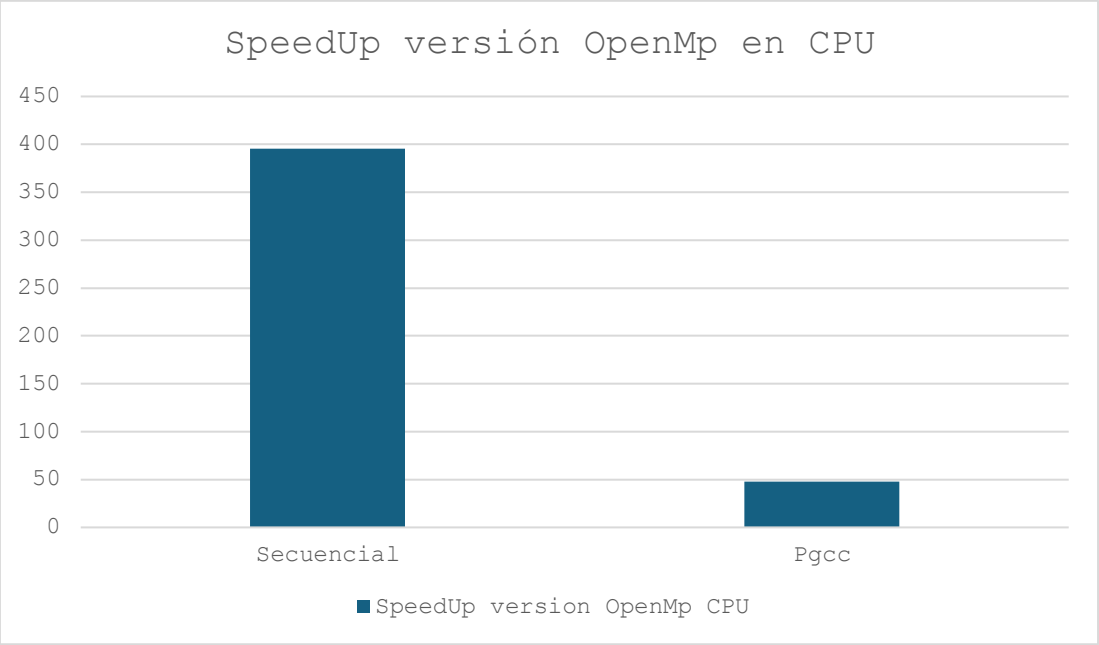
El resto del código se encarga de inicializar los parámetros y variables, leer la entrada del usuario, gestionar la memoria dinámica para las matrices de posiciones, velocidades, fuerzas y aceleraciones, y realizar la simulación iterativa mediante las funciones “initialize, update, y compute”.

Tras la explicación de esta versión, pasaremos a estudiar los resultados obtenidos en las pruebas realizadas:

Como se puede observar en la siguiente tabla y en la siguiente gráfica, esta versión ha mejorado mucho a la versión secuencial, obteniendo unos resultados de 48,098 segundos frente a los 395,54 de la versión secuencial, obteniendo un speedUp de x8,22, que, en estas ejecuciones tan costosas, es una mejora considerablemente alta.

Tiempo ejecución	Gcc	Pgcc	Nvc
OpenMp CPU	53,73	48,098	48,128





GPU

Para la compilación de este código en la versión de CPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenmp -foffload=nvptx-none -foffload=-misa=sm_35 -fcf-protection=none -fno-stack-protector -no-pie arch.c -o exe -lm -foffload=-lm -foffload=-latomic

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fopenmp -fast -Minfo=all -target=gpu -gpu=cc70

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fopenmp -fast -Minfo=all -target=gpu -gpu=cc70

Para intentar mejorar la versión CPU, he probado distintos métodos, los cuales no ha resultado efectivos. Primero he intentado paralelizar la función update para que ejecutara los cálculos en la GPU, utilizando estas directivas

```
“#pragma omp target map(to:pos[:np*nd], vel[:np*nd], f[:np*nd], acc[:np*nd]) map(from:pos[:np*nd], vel[:np*nd], acc[:np*nd])
```

y

```
#pragma omp teams distribute parallel for private(i)”,
```

pero al introducir estas directivas, los valores de los resultados daban erróneos, todos los valores de la energía cinética eran ceros, por lo tanto, probe otros métodos. El siguiente método fue tratar de optimizar el “main”, utilizando directivas como

```
“#pragma omp task, #pragma omp parallel, #pragma omp single y #pragma omp taskwait”,
```

pero el resultado fue el mismo, los valores de las energías eran erróneos.

Después probé a paralelizar la función “update” solamente para la CPU, para que asegure que las operaciones se realicen utilizando la CPU, reduciendo así el tiempo de ejecución y sin comprometer a la GPU para que no haya errores de precisión. Esta versión mejoraba un poco a la versión en CPU, pero no lo suficiente como para ser una GPU, por lo tanto, busqué otras alternativas.

Volví a intentar optimizar la función “update” para GPU utilizando un “collapse(2)”, el cual resulto bastante rápido, pero de nuevo los resultados eran erróneos, por lo que probé también a optimizar el “main” de nuevo con otra

directiva, esta vez un "`#pragma omp target data map(tofrom:pos[0:np*nd], vel[0:np*nd], force[0:np*nd], acc[0:np*nd])`",

pero nada, el resultado seguía siendo erróneo. Por lo que me decidí a cambiar de técnica y probar otra forma de mejorar esta versión.

Posteriormente probe a ejecutar desde el "main", la función "update" utilizando hilos, por lo que en cada hilo se ejecutara una iteración de este bucle externo, lo cual, debería mejorar la velocidad de ejecución, pero esta versión no mejoraba a la versión anterior, por lo tanto, decidí probar a paralelizar solamente la función "compute", que es la más costosa computacionalmente hablando.

Para esta versión, la versión final, la cual realiza las operaciones en la GPU, he tenido que utilizar otras directivas, ya que las utilizadas anteriormente, estaban enfocadas a la CPU, a continuación, explicare que directivas he utilizado y cuál es su función dentro del código.

La primera directiva que nos encontramos sería la directiva que paraleliza el bucle de la función compute, que es la directiva

"`#pragma omp target map(tofrom:pe,ke,f[:np*nd]) map(to:pos[:np*nd],vel[:np*nd])`",

la cual, podemos dividir en tres directivas individuales importantes. La primera sería "`#pragma omp target`", esta directiva indica que la región de código que sigue debe ejecutarse en un dispositivo, en este caso, una GPU. La siguiente sería "`map(tofrom:pe,ke,f[:np*nd])`", la cual, especifica que las variables "pe, ke y f" deben ser transferidos desde el host (CPU) al dispositivo (GPU) y viceversa, es decir se transferirán al inicio a la GPU y al final de la región de código serán devueltas al host, además también define una región de memoria para "f", que consta de "np*nd" elementos. Por último, encontramos "`map(to:pos[:np*nd],vel[:np*nd])`" que especifica que las variables "pos y vel" deben ser transferidos al dispositivo, pero no son necesarios en el host después de la ejecución, lo cual nos da un aumento de velocidad bastante alto.

La siguiente directiva que nos encontramos, es en la siguiente línea dentro de la función compute, que es la directiva "`#pragma omp teams distribute parallel for private(i, j, d, d2, rij) reduction(+:pe,ke)`". Esta consta de cinco directivas independientes, que son, la primera, "`#pragma omp teams`", que se encarga de crear un grupo de equipos, donde cada equipo puede tener múltiples hilos. La siguiente sería "`distribute`", que se encarga de distribuir las iteraciones del bucle for entre los distintos equipos creados anteriormente. La siguiente sería "`parallel for`", la cual, se

encarga de que dentro de cada equipo las iteraciones del bucle se paralelicen entre los distintos hilos disponibles. A continuación, encontramos la directiva `"private(i, j, d, d2, rij)"`, la cual, indica que las variables "i, j, d, d2, rij" deben ser privadas para cada hilo, por lo tanto, cada hilo tendrá su propia copia de estas variables. Por último, encontramos `"reduction(+:pe,ke)"`, esta, realiza una reducción en las variables "pe y ke" utilizando la operación de suma, por lo que cada hilo calculará su variable "pe y ke" y al final, estas se combinarán para obtener el resultado final.

```
163     #pragma omp target map(tofrom:pe,ke,f[:np*nd]) map(to:pos[:np*nd],vel[:np*nd])
164     #pragma omp teams distribute parallel for private(i, j, d, d2, rij) reduction(+:pe,ke)
165     for (k = 0; k < np; k++) {
```

Tras todas las implementaciones en esta versión, obtuve unos tiempos bastante rápidos, los cuales, se podían deducir que se habían obtenido gracias a la mejora de velocidad de la GPU.

Los tiempos obtenidos en esta versión, como podemos observar en las tablas y gráficos mostrados posteriormente, son, pese a la versión de "gcc" que es muy lenta, el resto son muy rápidos, obteniendo nuestro mejor tiempo en la versión de Nvc, con una ejecución de 16,1402s, logrando un speedUp respecto a la versión secuencial de x24,5, lo cual es muy buen tiempo para una versión en GPU.

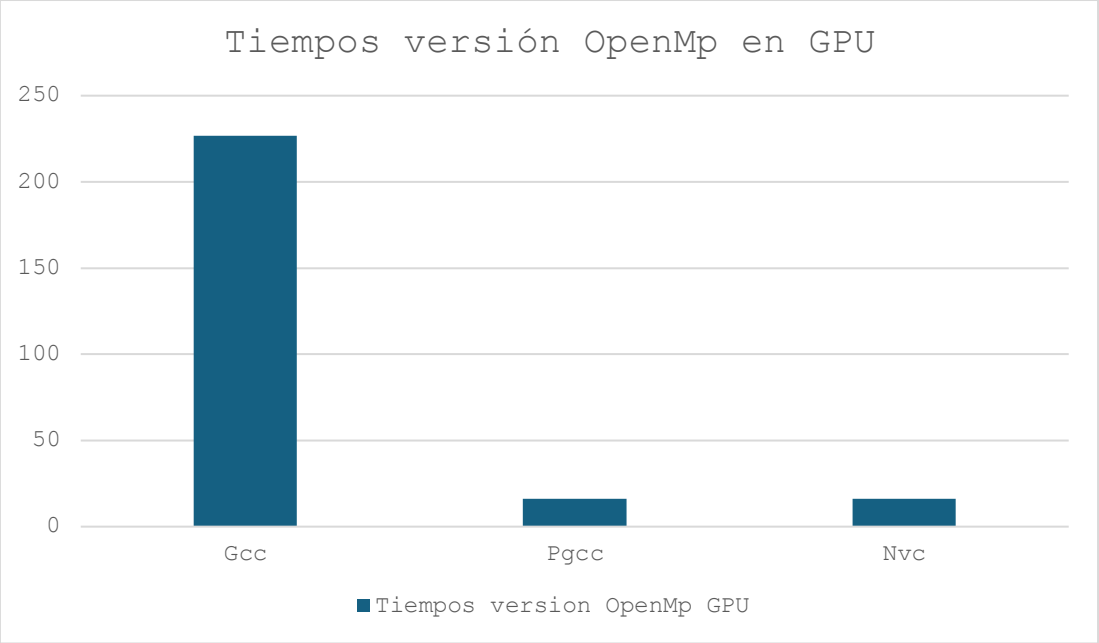
A continuación, voy a mostrar y explicar detalladamente el uso de la GPU para esta versión del código, ya que, al ser realizado en la GPU, podemos observar su uso fácilmente, con el comando: watch -n 1 nvidia-smi. El cual nos muestra varias características sobre el uso de las GPU de NVIDIA en el sistema.

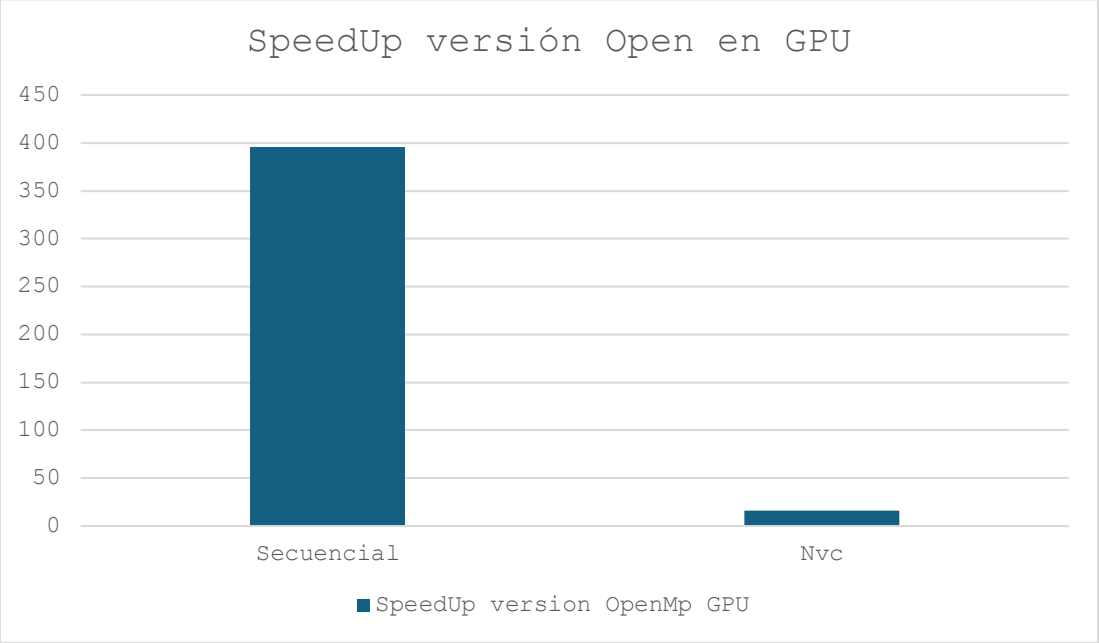
```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2070        Off |   00000000:04:00.0 Off |                  N/A |
|  0%   40C    P8             26W /  175W |      17MiB /   8192MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:81:00.0 Off |                  N/A |
| 43%   54C    P2            132W /  260W |     166MiB /  11264MiB |     99%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

En este caso, tenemos dos GPUs, la NVIDIA GeForce RTX 2070 y la NVIDIA GeForce RTX 2080. Se puede observar que la primera, no está siendo utilizada, pero la segunda, está siendo utilizada al 99%, lo cual nos lo indicaría el valor de la derecha, el valor indicado por GPU-Util, si recorremos ahora los valores de izquierda a derecha podemos observar que el ventilador está siendo utilizado al 43% de su velocidad, la temperatura de la GPU es de 54ºC, el estado de rendimiento de la GPU es p2, el consumo de energía es de 132W, y

el máximo es de 260W y por último, el uso de memoria gráfica es de 166Mib, de 11264Mib posibles.

Tiempo ejecución	Gcc	Pgcc	Nvc
OpenMp GPU	226,681	16,1694	16,1402





OPENACC

CPU

Para la compilación de este código en la versión de CPU hemos utilizado estos comandos:

En cuanto a la compilación de OpenAcc en su versión de CPU, no hemos podemos conseguir una compilación de gcc, por lo tanto, para la versión de CPU solo utilizaré las versiones de compilación de pgcc y de nvc.

Para la primera compilación utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -acc -ta=multicore

Para la segunda, utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -acc -ta=multicore

En cuanto a la complejidad de este código, he de decir que ha sido mucho mayor que para la versión de OpenMp, y ahora explicaré las versiones anteriores obtenidas y los problemas que he encontrado durante la paralelización de la versión secuencial con esta directiva.

Para mi primera versión de OpenAcc, lo que hice fue, para empezar, incluir la librería de “openacc.h” para poder utilizar las directivas necesarias de OpenAcc, que a continuación explicaré e indicare los problemas que he obtenido.

Primero, nos encontramos con la primera directiva, que fue “#pragma acc routine” lo cual indica que la función “dist” puede ser llamada en una región paralela y, además debe ser compilada para ejecutarse en el dispositivo paralelo.

```
11    #pragma acc routine
12    double dist ( int nd, double r1[], double r2[], double dr[] );
```

La segunda directiva que nos encontramos en mi código es la de “`#pragma acc data copyin(pos[0:nd*np],vel[0:nd*np], force[0:(nd*np)], acc[0:(nd*np)], potential, kinetic, nd, np)`”, que indica que las variables “pos, vel, force, acc, potential, kinetic, nd y np” deben ser copiadas desde la CPU al dispositivo antes de ejecutar la región paralela y volver a ser copiadas después. La siguiente directiva que nos encontraríamos seria “`#pragma acc parallel loop private(i)`” la cual indica que el bucle anidado debe actualizar las posiciones, velocidades y

posiciones de las partículas. La cláusula "`private(i)`" indica que la variable "i" debe ser privada para evitar condiciones de carrera.

Mas adelante en el código, nos encontraríamos la directiva "`#pragma acc parallel loop private(d, d2, i, j, rij) reduction(+:pe, ke)`" dentro de la función compute. Esta cláusula indicaría que las variables "d, d2, i, j, rij" deberían ser privadas para cada iteración y que las variables "pe y ke" deben ser reducidas al final de la región paralela. Por último, tendríamos la directiva "`#pragma acc parallel loop private(i)`" dentro de la función "initialize" que haría lo mismo que la anteriormente nombrada.

A priori esta versión del código debería funcionar correctamente y, en realidad daba unos tiempos bastante mejorados respecto a la versión secuencial, pero al realizar varias pruebas, me di cuenta de que los resultados obtenidos no eran exactamente correctos, por lo tanto, tuve que cambiar el código y probar otras técnicas o formas de implementar las directivas y probé esta nueva versión, la cual es la mi versión final.

En cuanto a esta versión, no cambia mucho respecto de mi versión anterior, y los cambios que he realizado han sido poco significativos en cuanto al código, pero los necesarios para que el código ejecute correctamente, devuelva los resultados esperados y los tiempos sean mejores que los de la versión secuencial.

Estos cambios han sido, para empezar, eliminar las directivas del bucle del "main", las cuales no aportaban una gran mejoría, pero sí que suponían una gran dificultad al tener que estar copiando las variables a la CPU y traerlas de vuelta, lo cual, podía suponer un error en los cálculos, que a posteriori, hemos comprobado que así era.

En cuanto a la directiva de la función "compute", no ha cambiado nada, ya que es la mejor manera de optimizar esta versión del código. Además, como ya he dicho en la versión del código en OpenMp, la función "compute" es la que realiza la mayoría de los cálculos y, por lo tanto, la que hay que buscar paralelizar de la forma óptima para que el código tarde lo menos posible. En cuanto a la directiva de la función "initialize" he decidido eliminarla también, por el mismo motivo que las anteriores, no optimiza nada el código.

```
222   #pragma acc parallel loop private(d, d2, rij, i, j) reduction(+:pe, ke)
223   for ( k = 0; k < np; k++ )
224   {
```

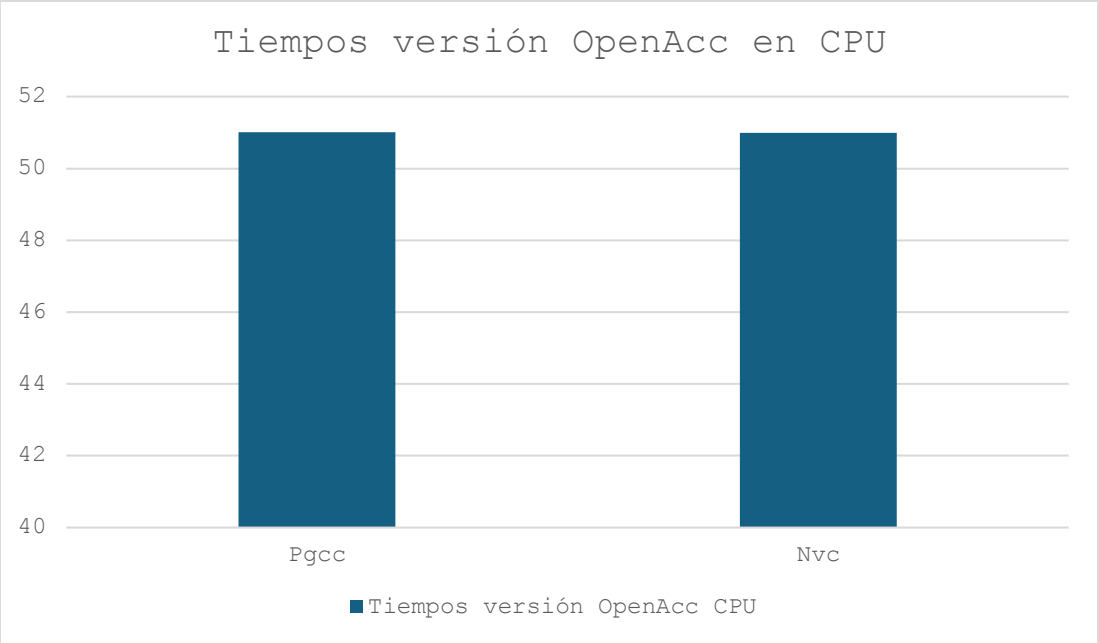
Por último, he decidido añadir la directiva "#pragma acc parallel loop private(i)" dentro de la función compute, para paralelizar el bucle externo y permitir que las actualizaciones de las posiciones, velocidades y aceleraciones de las partículas se realicen de forma paralela aumentando así, la eficiencia en los

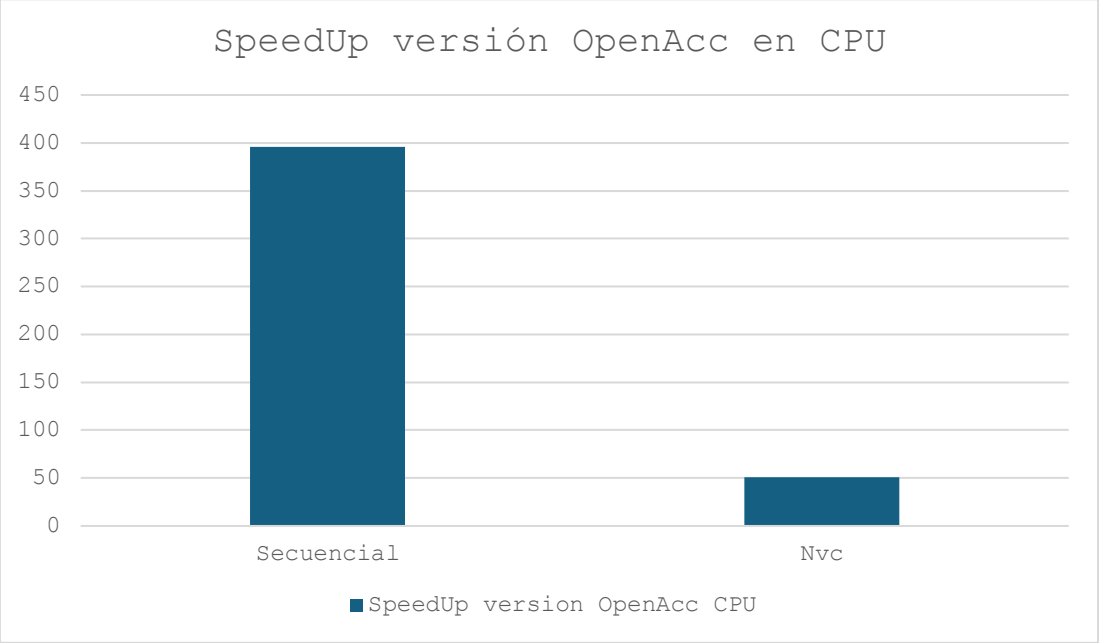
sistemas con varios núcleos de procesamiento (cuantos más núcleos, más eficiencia).

```
345   #pragma acc parallel loop private(i)
346   for ( j = 0; j < np; j++ )
347   {
348     for ( i = 0; i < nd; i++ )
349     {
```

Tras esta versión, he estado probando e intentando utilizar otras técnicas de paralelización, pero todas han resultado sin éxito, debido a que la mayoría de las técnicas que he intentado implementar, realizaban mal los cálculos y devolvían resultados erróneos debido a que algunas variables no se actualizaban correctamente. Otros errores que he experimentado han sido enviar las variables a la CPU, pero luego no devolverlas y por lo tanto obtener unos resultados de ceros en todas las energías y, por último, intentar enviar y devolver las variables a la CPU de una forma muy poco óptima y por ello obtener unos tiempos altísimos, los cuales, no me servían.

Tiempo ejecución	Pgcc	Nvc
OpenAcc CPU	51	50,9845





GPU

Para la compilación de este código en la versión de GPU hemos utilizado estos comandos:

La primera compilación utilizando gcc ejecutaremos el comando:

gcc-11 -O3 -fopenacc -foffload=nvptx-none -foffload=-misa=sm_35 -fcf-protection=none -fno-stack-protector -no-pie arch -o exe -lm -foffload=-latomic -foffload=-lm

Para la segunda utilizaremos “Portland Group Compiler Collection” desde la carpeta en la que se encuentra este compilador:

pgcc -O3 -fast -Minfo=all -acc -target=gpu -gpu=cc70

Para la última utilizaremos "NVIDIA C Compiler", desde su respectiva carpeta, ejecutaremos el comando:

nvc -O3 -fast -Minfo=all -acc -target=gpu -gpu=cc70

En cuanto a la implementación de este código, he de decir que ha sido muy costosa, ya que he estado probando múltiples códigos, utilizando múltiples y diferentes técnicas de paralelización para llegar a la implementación final, la cual he obtenido un resultado y unos tiempos óptimos.

Para las primeras versiones realizadas utilizando la OpenAcc dirigido a GPU, utilizando las herramientas de profiling, he observado que la función “compute”, es la que mas tiempo se toma, pero en este caso, la función “main” también puede optimizar mucho el código, por lo tanto, he intentado mejorar esto, intentando paralelizar el bucle principal del “main”. Para ello he utilizado la directiva “`#pragma acc data copyin(nd, np, pos[0 : nd * np], vel[0 : nd * np]) create(force[0 : nd * np])`”. Esta directiva debería permitir administrar la transferencia de datos entre la CPU y la GPU, especificando que datos se copiarán a la GPU. Además, esta debería asignar memoria en la GPU antes de la ejecución paralela. Esta directiva me devolvió un error indicando que los datos no se podían encontrar en la GPU, debido bien a que no se habían copiado correctamente desde la memoria la host antes de comenzar la ejecución paralela o que no se habían creado correctamente. Por lo tanto, debía revisar las cláusulas “copyin y create”.

Refiriéndome al código final, el inicio es igual que el de la versión en CPU, incluyendo la librería de “openacc.h” para la paralelización, esta vez en la GPU. También en la función “dist” indico que es una rutina de openAcc.

En cuanto a la función main, hay que fijarse que incluyo la directiva “`#pragma acc data copyin(nd, np)`” en el inicio del main, para asegurarme de que las variables “nd y np” se copian correctamente en la GPU desde la CPU, lo

cual hace que estas variables sean accesibles en el dispositivo para las operaciones de paralelización.

```
76    #pragma acc data copyin(nd,np)
```

Después, en el bucle principal del "main" podemos encontrar la directiva "`#pragma acc enter data copyin(pos[0:nd*np], vel[0:nd*np], force[0:(nd*np)], acc[0:(nd*np)])`" pero no en el bucle exterior, sino que la introduzco una vez inicializado el código, lo que hace que se copien explícitamente las variables "pos, vel, force y acc" desde la memoria del host a la memoria del dispositivo. Esto viene muy bien para las regiones paralelas. Para finalizar el "main", utilizo la directiva "`#pragma acc exit data delete(pos[0:nd*np], vel[0:nd*np], force[0:nd*np], acc[0:nd*np])`" la cual elimina todas las copias de las variables "pos, vel, force y acc" de la memoria del dispositivo, liberando así memoria en la GPU.

```
161     for ( step = 0; step <= step_num; step++ )
162     {
163       if ( step == 0 ) {
164         initialize ( np, nd, pos, vel, acc );
165   #pragma acc enter data copyin(pos[0:nd*np], vel[0:nd*np], force[0:(nd*np)], acc[0:(nd*np)])
166       }

183   #pragma acc exit data delete(pos[0:nd*np], vel[0:nd*np], force[0:nd*np], acc[0:nd*np])
```

En cuanto nos fijamos en la función "compute", nos encontramos con la siguiente directiva, "`#pragma acc data present(pos[0:nd*np], vel[0:nd*np], f[0:(nd*np)])`", la cual, se asegura de que las variables "pos, vel y f" están presentes en la memoria del dispositivo, para evitar copias adicionales y permitir que las regiones paralelas accedan a estos datos directamente. Seguidamente, está la directiva "`#pragma acc parallel loop gang vector_length(128) private(d, d2, rij, i, j, k) firstprivate(np, nd, PI2) reduction(+:pe, ke)`", la cual tiene varias partes para analizar. La primera seria "`gang`", la cual define un paralelismo donde cada "gang" es un numero de hilos que trabajan a la vez. Después tendríamos la "`vector_length(128)`" la cual define la longitud del vector, es decir, el número de hilos por "gang" que se ejecutarán en paralelo. A continuación, nos encontramos el "`private(d, d2, rij, i, j, k)`" que declara que estas variables son privadas para cada iteración del bucle, evitando así condiciones de carrera. Seguido a esto, está el "`firstprivate(np, nd, PI2)`" el cual se encarga de Inicializar estas variables con sus valores del host al comienzo de cada "gang". Ya por último estaría el "`reduction(+:pe, ke)`" que define una operación de reducción para sumar los valores de "pe y ke" a través de todas las iteraciones del bucle en paralelo.

```
265   #pragma acc data present(pos[0:nd*np],vel[0:nd*np],f[0:(nd*np)])
266   #pragma acc parallel loop gang vector_length(128) private(d, d2, rij, i, j, k) firstprivate(np, nd, PI2) reduction(+:pe, ke)
267     for ( k = 0; k < np; k++ )
268     {
```

Luego encontraríamos de nuevo la directiva "`#pragma acc loop vector private(d, d2, rij, i, j, k)`" la cual indicaría que las iteraciones del bucle deben ejecutarse en paralelo en la GPU y que las variables indicadas son privadas en cada iteración. El siguiente sería el "`#pragma acc atomic`" que su función será asegurar que la operación siguiente se ejecuta de manera atómica y sin interrupciones ni condiciones de carrera.

```
270         for ( i = 0; i < nd; i++ )
271           f[i+k*nd] = 0.0;
272       #pragma acc loop vector private(d, d2, rij, i, j, k)
```

```
287           for ( i = 0; i < nd; i++ )
288       #pragma acc atomic
289             f[i+k*nd] = f[i+k*nd] - rij[i] * sin ( 2.0 * d2 ) / d;
290         }
```

Por último, en la función "update" encontramos las mismas directivas que al inicio de la función compute, es decir,

"`#pragma acc data present(pos[0:nd*np], vel[0:nd*np], f[0:nd*np], acc[0:nd*np])`"

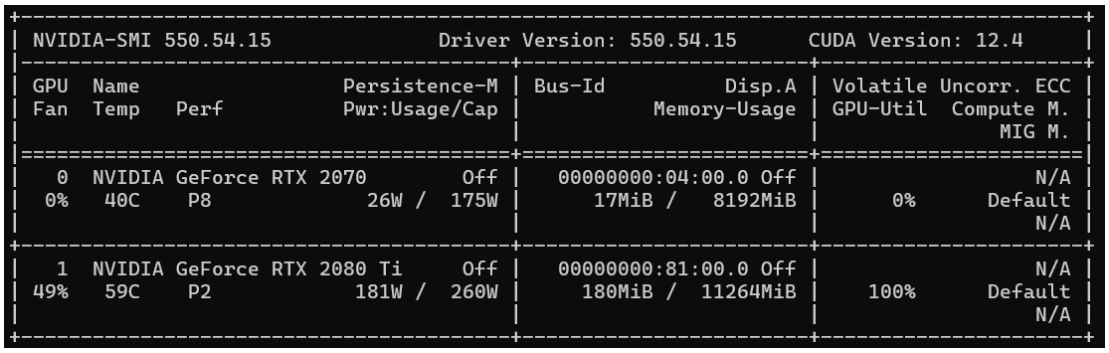
y

"`#pragma acc parallel loop gang vector_length(128) firstprivate(np, nd, dt, rmass)`". Que obviamente tienen la misma función que las directivas explicadas anteriormente en la función compute.

```
539   #pragma acc data present(pos[0:nd*np],vel[0:nd*np],f[0:(nd*np)], acc[0:nd*np])
540   #pragma acc parallel loop private(i, j) firstprivate(np, nd, rmass, dt)
541     for ( j = 0; j < np; j++ )
542     {
```

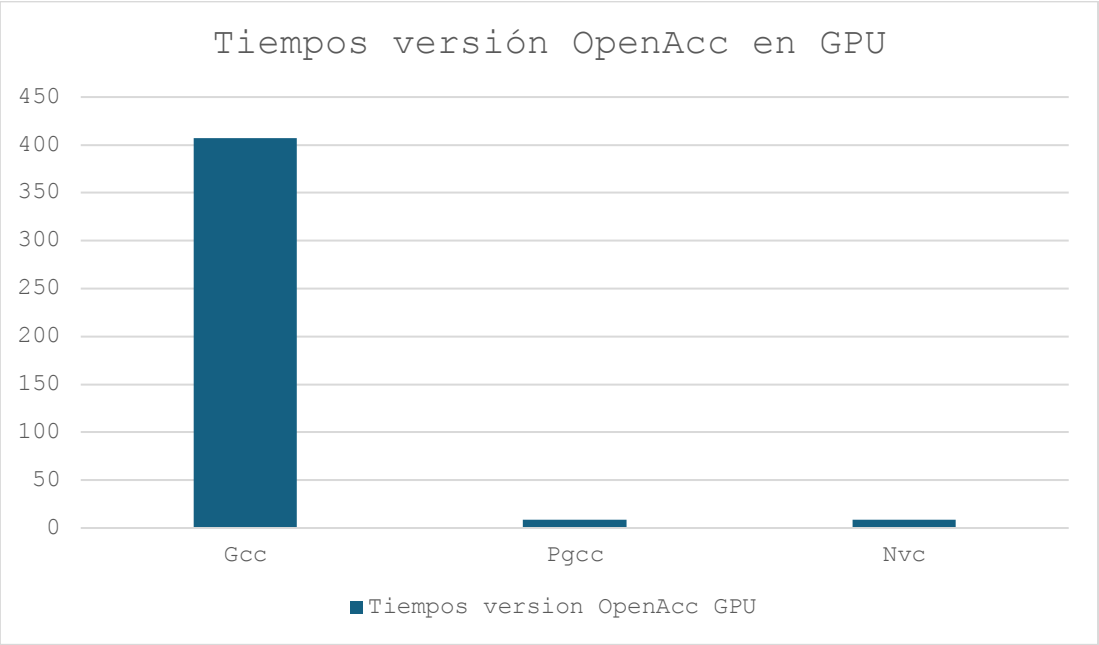
Tras todas estas implementaciones obtuve unos resultados bastante rápidos, pese a que la versión de gcc da unos resultados muy altos, las otras versiones han resultado ser mucho más rápidas. Obteniendo así el mejor tiempo en el compilador nvc, resultando un tiempo de 8,59811 segundos, logrando así un speedUp de x46 lo cual mejora muchísimo al tiempo obtenido en la versión de CPU

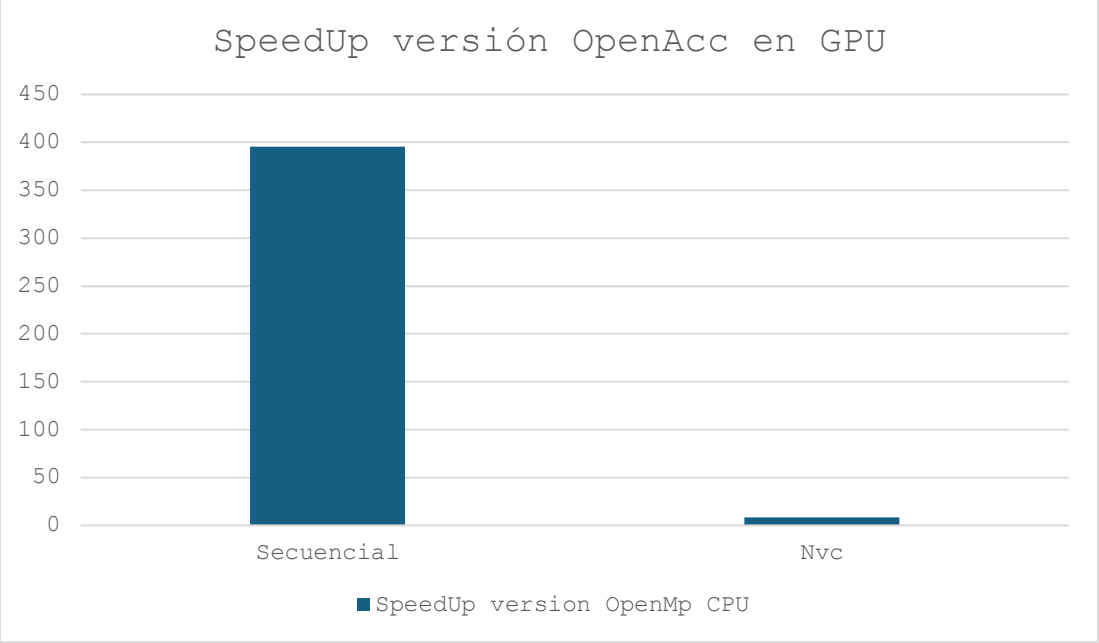
A continuación, voy a mostrar y explicar detalladamente el uso de la GPU para esta versión del código, ya que, al ser realizado en la GPU, podemos observar su uso fácilmente, con el comando: watch -n 1 nvidia-smi. El cual nos muestra varias características sobre el uso de las GPU de NVIDIA en el sistema.



En este caso, tenemos las mismas dos GPUs indicadas en la parte de OpenMp, la NVIDIA GeForce RTX 2070 y la NVIDIA GeForce RTX 2080. Se puede observar que la primera, no está siendo utilizada, pero la segunda, está siendo utilizada al 100%, lo cual nos lo indicaría el valor de la derecha, el valor indicado por GPU-Util, si recorremos ahora los valores de izquierda a derecha podemos observar que el ventilador está siendo utilizado al 49% de su velocidad, la temperatura de la GPU es de 59ºC, el estado de rendimiento de la GPU es p2, el consumo de energía es de 181W, y el máximo es de 260W y por último, el uso de memoria gráfica es de 180Mib, de 11264Mib posibles.

Tiempo ejecución	Gcc	Pgcc	Nvc
OpenAcc GPU	407,387	8,6342	8,59811





Para el código en esta versión, podría implementarse solamente un código, y simplemente compilarlo con la CPU y la GPU para comprobar los distintos resultados, pero en el código que tiene la versión de GPU, al compilarlo con la CPU da unos tiempos más altos, por lo tanto, he decidido hacerlo en dos

códigos separados de modo que puedo obtener los mejores tiempos en cada una de las versiones en vez de hacer solo un código y que no sea óptimo en las dos versiones.

CUDA

Para la compilación de este código hemos utilizado este comando:

/usr/local/cuda/bin/nvcc -Wno-deprecated-gpu-targets -arch=sm_75

Para conseguir la última versión del código, la que mayor optimización tiene, he tenido muchos problemas, ya que he estado probando una gran cantidad de optimizaciones y paralelismo posibles en este lenguaje, hasta que por fin he conseguido llegar a una optimización bastante alta.

Mi primera versión de este código, en la ejecución de prueba me daba unos resultados coherentes, aunque en cuanto a los tiempos de ejecución eran más altos que en la versión secuencial, por lo tanto, podría parecer que este código no era optimo, aunque en las ejecuciones más grandes los resultados eran más rápidos que en la versión secuencial, aunque no mucho tampoco, por lo tanto, me dispuse a intentar optimizar más este código.

En mi segunda versión intenté aprovechar la paralelización de bucles y la optimización de memoria en la función “compute” que es la que ocupa el mayor tiempo en el código. Esta versión sí que era más rápida que la anterior, y en la ejecución de prueba los resultados eran muy similares a los de la versión secuencial, aunque levemente mayores, pero en este caso en las ejecuciones grandes sí que podíamos observar una gran mejora de tiempos, ya que estos se reducían a un tercio respecto de la versión secuencial, pero aún me daban unos tiempos mayores que las versiones de OpenMp y openAcc, así que decidí que seguramente podría mejorar bastante estos tiempos, por lo tanto decidí utilizar otras técnicas de paralelización y de optimización.

En esta tercera versión del código decidí probar la técnica de paralelización mediante división del trabajo en bloques y la reducción de datos. Sorprendentemente en esta versión obtuve unos resultados increíblemente rápidos comparados con los anteriores y con las otras versiones, tanto en las ejecuciones de prueba como en las ejecuciones grandes. El problema de estos resultados es que la acumulación de energías tanto potencial como cinética me daban unos números incoherentes debido a un incorrecto almacenamiento de las variables.

Debido al anterior error, implementé una nueva versión, esta vez arreglando el problema del almacenamiento de las variables. En esta versión, muy similar a la anterior, desgraciadamente obtuve de nuevo unos resultados muy similares a los de la segunda versión, por lo tanto, no estaba contento del todo y decidí seguir probando, aunque utilizando otros métodos, ya que me di cuenta de que en la forma en la que estaba intentando optimizar el código no iba a resultar efectiva porque después de tanto tiempo habiendo estado intentando optimizarlo mediante ese método, no había servido para optimizar mucho el código.

En esta quinta versión, decidí darle un cambio al código e implementarlo mediante otra técnica: la división del espacio en celdas y calculando las interacciones solo entre las partículas de la misma celda o de las adyacentes, lo cual reduciría significativamente el número de interacciones a calcular en cada paso del tiempo, pero, de nuevo el tiempo se elevó muchísimo, llegando a ser más alto que en la versión secuencial tanto en las ejecuciones de prueba como en las ejecuciones más altas, por lo que esta versión tampoco me servía.

Para la siguiente versión, decidí cambiar de estrategia de nuevo, y esta vez utilicé la optimización de fuerzas dentro de cada bloque para reducir la cantidad de cálculos redundantes y mejorar la eficiencia, aprovechando la memoria compartida, lo cual debería reducir la cantidad de accesos a memoria global y, por tanto, mejorar la eficiencia del código, y así fue, los tiempos mejoraron bastante respecto de la mejor versión, que hasta el momento era la segunda (que también eran tiempos parecidos a la cuarta versión), tanto en las ejecuciones de prueba, como en las ejecuciones más grandes, pero desafortunadamente, los tiempos seguían siendo más altos que los esperados, por lo que de nuevo decidí cambiar de estrategia.

Para esta versión he decidido reducir la cantidad de datos que se transfieren entre la memoria global y la memoria compartida en cada bloque, cargando un subconjunto de las posiciones de partículas en la memoria compartida en cada bloque y luego calcular las fuerzas solo entre este subconjunto y todas las partículas en la memoria global. En resumen, cada bloque de hilos carga un subconjunto de posiciones de partículas en la memoria compartida. Luego, cada hilo en el bloque calcula las fuerzas entre este subconjunto y todas las partículas en la memoria global. Sorprendentemente esta versión sí que ha conseguido unos tiempos muy rápidos, mejorando mucho a la versión secuencial. El fallo de esta versión era que no estaba consiguiendo implementar bien los hilos para luego realizar las llamadas correspondientes a las funciones, por lo tanto, los resultados eran incorrectos.

Por lo tanto, decidí cambiar de estrategia de nuevo e implementé una versión que en vez de reducir los tiempos utilizando el `"atomicAdd"` en el compute y que este se encargase de hacer todas las reducciones en él mismo, decidí implementar una función “reduction”, la cual se encarga de reducir un arreglo de datos sumando sus elementos, se utiliza para calcular la energía total.

```
304  __global__ void reduction(double *g_data, int n, double* out) {
305      int tile = TILE_SIZE * blockIdx.x;
306      __shared__ double data[BLOCK_SIZE];
307
308      int my_idx = tile + threadIdx.x;
309      data[threadIdx.x] = (my_idx < n) ? g_data[my_idx] : 0.0;
310
311      int next_idx = my_idx + blockDim.x;
312      if (next_idx < n) {
313          data[threadIdx.x] += g_data[next_idx];
314      }
315
316      __syncthreads();
317
318      for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
319          if (threadIdx.x < stride) {
320              data[threadIdx.x] += data[threadIdx.x + stride];
321          }
322          __syncthreads();
323      }
324
325      if (threadIdx.x == 0) {
326          atomicAdd(out, data[0]);
327      }
328  }
```

Pero el cambio mas significativo en el código fue, en la función main, para la cual, eliminé la función initialize, y la implementé en el main, lo cual hacía más óptimo el código.

```
117      int j;
118      int seed = 123456789;
119      r8mat_uniform_ab(nd, np, 0.0, 10.0, &seed, h_pos);
120
121      for (int j = 0; j < np; j++) {
122          for (int i = 0; i < nd; i++) {
123              h_vel[i + j * nd] = 0.0;
124              h_acc[i + j * nd] = 0.0;
125          }
126      }
127
128      for(j = 0 ; j < np ; j++){
129          h_ken[j] = 0;
130          h_pen[j] = 0;
131      }
```

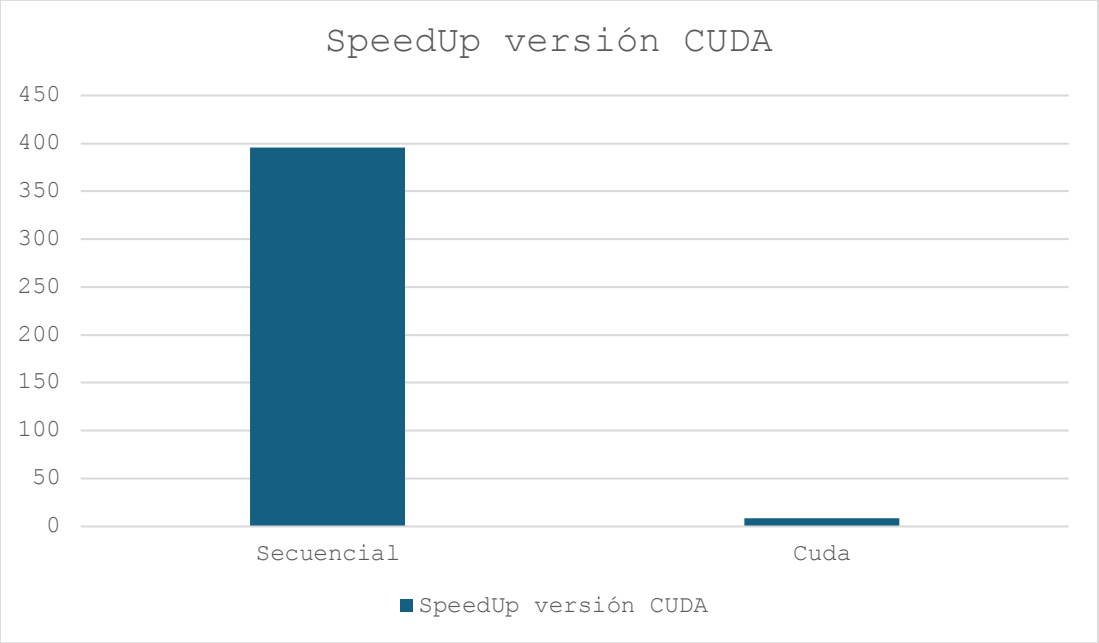
También tuve que modificar el interior de "`if(step== step_num)`", ya que encontré una manera de hacer que el código se ejecutase más rápido,

incluyendo las llamadas a la función reduce en él y, copiando y liberando la memoria necesaria.

El uso de la GPU para la optimización de este código ha sido basado en cuatro técnicas, la primera seria el paralelismo masivo, permitiendo a la GPU ejecutar miles de hilos en paralelo y que cada hilo realice unos cálculos específicos, esta técnica ha sido usada tanto en el "kernel compute", como en el "kernel update" calculando las fuerzas y energías de cada partícula como actualizando las posiciones y velocidades de la partícula respectivamente, la siguiente técnica que he utilizado es la reducción paralela, que es la comentada anteriormente para el "kernel reduction", la cual, es una técnica que se usa para sumar elementos de manera eficiente en paralelo, esta utiliza memoria compartida y sincronización para sumar las energías potencial y cinética de las partículas en paralelo, lo cual acelera el tiempo en comparación con una versión secuencial, la tercera, ha sido la transferencia de datos, que es la más importante, aunque la más sencilla, ya que sin ella, no podría utilizar la GPU, ya que si no se pasan los datos correctamente y se liberan correctamente, el programa dará errores, como me ha pasado a mi en las versiones anteriores y, por último, he decidido utilizar la memoria compartida externa en la función compute "compute", cargando, cada hilo, las posiciones de su partícula en la memoria compartida.

El resultado en esta versión es de 9,82323 segundos, lo cual acelera muchísimo respecto de la versión secuencial, logrando un speedUp de x40,29. Si que es cierto, que tratándose de una GPU y de un lenguaje como es CUDA, podríamos esperar un tiempo un poco más bajo, pero aun así, estoy bastante contento con el resultado obtenido, aunque estoy seguro de que existen mejores formas y diferentes técnicas para lograr un speedUp mayor para este lenguaje. Pese a esto, es un tiempo bastante bueno y por ello he decidido tomarlo como mi versión final.

Tiempo ejecución	CUDA
CUDA	9,82323




```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2070        Off |   00000000:04:00.0 Off |                  N/A |
|  0%   44C    P8             26W /  175W |      17MiB /   8192MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:81:00.0 Off |                  N/A |
| 55%   62C    P2            158W /  260W |     166MiB /  11264MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

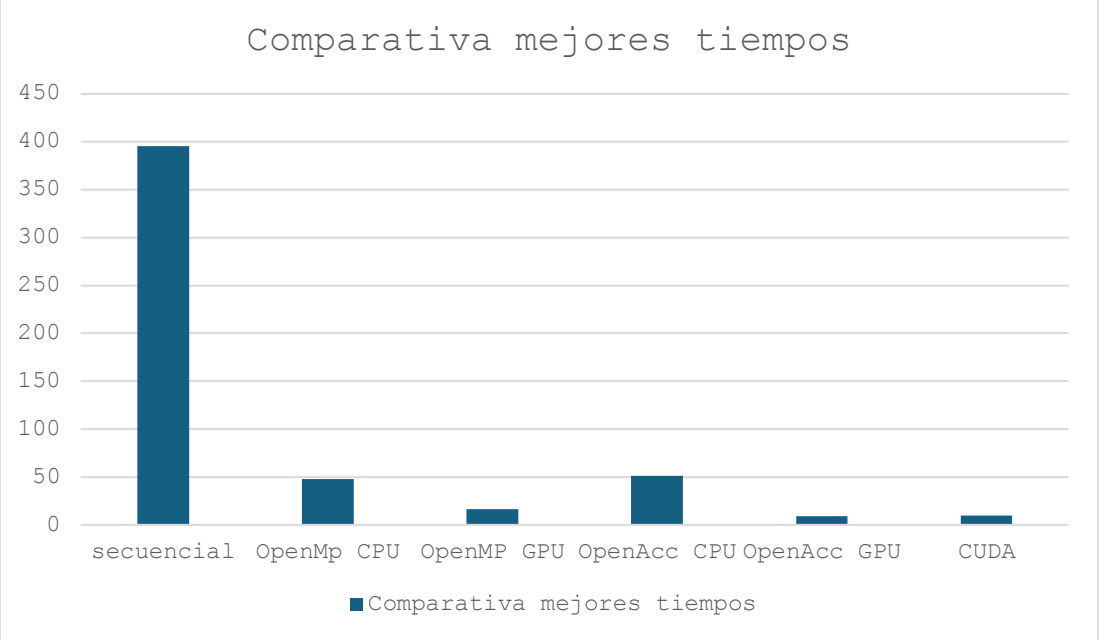
En este caso, tenemos las mismas dos GPUs indicadas anteriormente. Se puede observar que la primera, no está siendo utilizada, pero la segunda, está siendo utilizada al 100%, lo cual nos lo indicaría el valor de la derecha, el valor indicado por GPU-Util, si recorremos ahora los valores de izquierda a derecha podemos observar que el ventilador está siendo utilizado al 55% de su velocidad, la temperatura de la GPU es de 62ºC, el estado de rendimiento de la GPU es p2, el consumo de energía es de 158W, y el máximo es de 260W y por último, el uso de memoria gráfica es de 166Mib, de 11264Mib posibles.

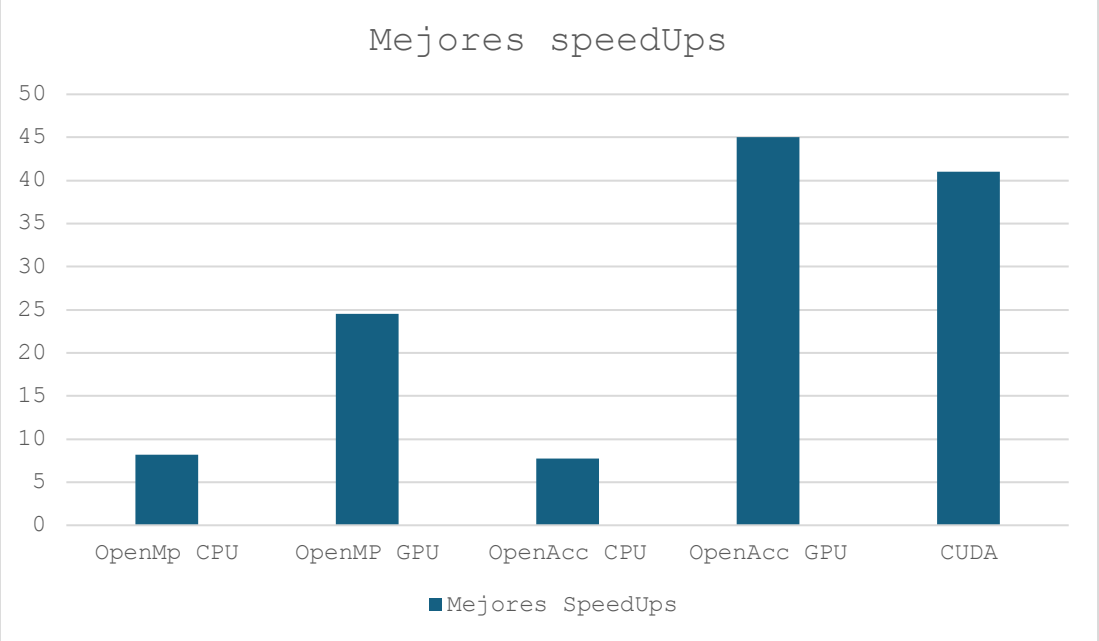
Conclusión

Para poder sacar una conclusión clara, primero debemos analizar cada código, su complejidad, el tiempo que me ha llevado implementarlo y la dificultad de aprendizaje y dominio de este, para luego poder comparar la mejoría de tiempos, y si mereciera la pena utilizar unas directivas u otras o, utilizar CUDA.

Por lo tanto, para ello empezaré mostrando una gráfica con todos los mejores tiempos de cada versión, mostrando sus speedUps, y con lo comentado anteriormente sobre la dificultad y tiempo de implementación de cada código haré una valoración personal sobre si merece la pena o no invertir tanto tiempo para obtener unos resultados mucho mejores o no, todo ello bajo mi punto de vista, como estudiante que parte prácticamente de cero en el uso de las directivas y del lenguaje CUDA.

Tiempo ejecución	Secuencial	OpenMp CPU	OpenMp GPU	OpenAcc CPU	OpenAcc GPU	CUDA
Segundos	395,54	48,098	16,1402	50,9845	8,59811	9,82323





Como se puede observar a simple vista, podemos diferenciar dos grupos en cuanto a sus tiempos: los que realizan las operaciones en la CPU y los que las realizan en la GPU. Siendo mucho más rápidos los tiempos obtenidos en las versiones de GPU.

En cuanto a las versiones de CPU, tanto de OpenMp como de OpenAcc, he de decir que no me ha costado tanto aprenderlas e implementarlas, por lo tanto, para mí, son unas directivas las cuales merecería la pena aprender desde cero, ya que no llevan demasiado tiempo para estudiarlas y, en cuanto a su implementación no tiene una complejidad muy grande, por lo tanto cualquier persona que tenga un mínimo de experiencia programando o un conocimiento básico de c, debe ser capaz de implementar unas mejoras en los tiempos de ejecución de prácticamente cualquier programa que requiera de alto coste computacional. Para decantarme sobre una de las dos, he de decir que OpenMp es mucho más sencillo que OpenAcc y, si quieres realizar mejoras en cuanto a la CPU, recomendaría 100% usar OpenMp, que además esta directiva está enfocada más a la paralelización en CPU.

Para las versiones de GPU, voy a hacer de nuevo dos grupos, uno en el que entrarán tanto OpenMp como OpenAcc y dejaré aparte CUDA, ya que merece un apartado aislado.

En cuanto a las versiones de OpenAcc y OpenMp para la GPU, he de decir que han sido bastante más costosas, tanto su estudio, como su implementación, ya que he encontrado múltiples problemas (como ya he explicado en cada apartado) en cuanto al tráfico de variables entre la CPU y GPU. Por lo tanto, he tenido que estar probando diversas técnicas, utilizando diferentes pragmas, hasta encontrar el mejor en cada situación. Para la versión de OpenMp final, a simple vista parece bastante sencilla, ya que simplemente paralelizo la función “compute”, pero para llegar a ella primero he tenido que probar bastantes versiones anteriores. En cuanto a la versión de OpenAcc sí que me ha costado bastante más encontrar la versión final, ya que, desde las primeras versiones, los resultados me han dado erróneos, y aunque poco a poco he ido mejorando los tiempos en cada versión, encontrar una versión la cual hiciera todas las operaciones bien y se ejecutase en tiempos bajos, me ha costado muchísimo.

Bien es cierto que la directiva OpenAcc está enfocada a la paralelización en la GPU, por ello es más complicada de dominar, ya que la dificultad reside en realizar los pasos de variables de CPU a GPU correctamente. Por ello en los resultados se puede observar una mejoría considerable respecto de la versión de OpenMp. Desde mi experiencia en este trabajo diría que no merece tampoco la pena utilizar y aprender las directivas de OpenAcc en comparación con las de OpenMp, ya que la dificultad es mucho más baja y la mejora de tiempos no es demasiado grande. Si que es verdad que en ejecuciones las cuales tomen unos tiempos elevadísimos, si puede llegar a merecer la pena aprender al completo esta directiva y utilizarla para optimizar códigos mucho más costosos, pero para este tipo de códigos, yo me quedaría con OpenMp aún.

Por último, quedaría comparar ambas versiones con la versión de CUDA, la cual, debido a su implementación y complejidad, debería dar los resultados más bajos entre todas las GPUs, aunque este no es mi caso, aunque estoy muy cerca de ello. En el momento en el que escogí este trabajo, partía de cero tanto en ambas directivas como en el lenguaje CUDA, por ello el estudio e implementación de los códigos partía desde la misma base, es decir, ninguna. Para ello las primeras semanas de trabajo, como se indicó en el plan de trabajo las dediqué al estudio de las directivas y de CUDA, hasta llegar ya a empezar a implementar las versiones de los códigos finales. Por ello, en cuanto a tiempo dedicado, sin duda, CUDA ha sido al que mas tiempo le he dedicado, y al final no he obtenido los resultados esperados, ya que podría haberse optimizado más y obtener unos mayores speedUps. Por lo tanto, en mi opinión, he de decir que aunque CUDA sea la versión más rápida de todas, en cuanto a tiempo de estudio y de implementación, bajo mi punto de vista, para una persona principiante, no merece la pena enseñarle este lenguaje, ya que sería mucho mas eficaz enseñarle a utilizar las directivas de OpenMp y OpenAcc debido a su rapidez y a su eficiencia, que en el caso de las GPUs debería ser similar a la velocidad de CUDA, pero para mí, en el caso de una empresa que se dedique a la optimización con el uso de GPUs, sería mucho mas eficaz tener a una persona especializada en CUDA, ya que debería obtener unos resultados mejores, que además, no le debería tomar mucho tiempo para llegar a ellos si conoce bien este lenguaje.

Impacto

En cuanto al impacto de este trabajo, creo que con el único objetivo con el que se podría relacionar seria con el de “Trabajo decente y crecimiento económico”, debido a que como he comentado en la conclusión, para una empresa, tener una persona especializada en la optimización utilizando la GPU, ya sea tanto en las directivas de OpenMp y OpenAcc, como en el lenguaje CUDA, puede ahorrar muchísimo dinero y tiempo, ya que una persona que parta desde cero en estos temas, por lo general, tarda bastante en comprender los conceptos y empezar a utilizar las directivas correctamente, además de que seguramente no encuentre la forma mas óptima de utilizar las directivas. Por lo tanto, tener a una persona, ayudaría tanto al trabajo decente, como al crecimiento económico de dicha empresa.

Anexos

Repositorio con los códigos del cálculo del número pi:

https://github.com/rinaldinho99/optimized-pi-calculation

Repositorio con los códigos de Dinámica molecular:

https://github.com/rinaldinho99/optimized-molecular-dynamics-

Informe de originalidad generado por la herramienta Turnitin:

https://drive.google.com/file/d/1Q-aKVVz6UXE9x2BvQzIp-wfMaCsXidYC/view?usp=drive_link

Bibliografía

Códigos cálculo de pi:

Entregados por los profesores

Código dinámica molecular secuencial:

Entregado por los profesores

OpenMP:

Documentación entregada por los profesores (como transparencias impartidas en asignaturas como CAR)

https://es.wikipedia.org/wiki/OpenMP

https://www.openmp.org/

The OpenMP Common Core: Making OpenMP Simple Again. De Timothy G. Mattson (Autor), Yun (Helen) He (Autor), Alice E. Koniges (Autor)

OpenACC:

Documentación entregada por los profesores (como transparencias impartidas en asignaturas como CAR)

https://es.wikipedia.org/wiki/OpenACC

https://www.openacc.org/

Parallel Programming with OpenACC. de Rob Farber (Redactor)

CUDA:

Documentación entregada por los profesores (como transparencias impartidas en asignaturas como CAR)

https://es.wikipedia.org/wiki/CUDA

https://biblus.us.es/bibing/proyectos/abreproy/11926/fichero/Segunda+parte+TECNOLOGIA+CUDA%252Fi_cuda.pdf

https://riubu.ubu.es/bitstream/handle/10259/3933/Programacion_en_CUDA.pdf?sequence=1

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 03 20:36:04 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)