



ARTÍCULO DE INVESTIGACIÓN

Tecnologías de programación paralela y concurrente



Gerardo Daniel Naranjo Gallegos, A01209499.

A01209499@itesm.mx

Tecnológico de Monterrey, Campus Querétaro.

Materia: Multiprocesadores.

Profesor: Pedro Oscar Pérez Murueta.

pperezm@tec.mx

1 DE DICIEMBRE DE 2019

Contenido

Resumen	2
Introducción	3
Desarrollo	4
Librería cheader.h y cppheader.h.....	4
Secuencial en C.....	4
Secuencial en C++	5
Secuencial en Java	6
Paralelo en OpenMP	6
Paralelo TBB.....	7
Paralelo en CUDA.....	8
Paralelo con <i>Threads</i> en Java	8
Paralelo en Java con Fork/Join	9
Análisis de resultados	10
Comparativa entre tecnologías secuenciales.....	10
Comparativa entre tecnologías paralelas.....	11
Comparativa entre secuenciales y paralelas de Java	12
Comparativa entre secuencial y paralela de C y C++	13
Comparativa de todas las tecnologías implementadas:.....	14
Conclusión	15
Referencias	16
Anexos	18
Librería cheader.h.....	18
Librería cppheader.h	20

Implementación en C.....	22
Implementación en C++.....	25
Implementación en Java.....	28
Implementación en OpenMP	31
Implementación en TBB	34
Implementación en CUDA	37
Implementación Threads.....	41
Implementación Fork/Join.....	44

Resumen

En este artículo de investigación se desarrollarán algoritmos que resuelvan un problema de programación, en distintas herramientas que los puedan hacer paralelizados y de manera secuencial, con el objetivo de comparar las técnicas de programación paralela y concurrente.

El problema por resolver será operaciones con vectores. Se utilizará el servidor otorgado por el profesor, con el objetivo de realizar una comparación equitativa.

Introducción

Este es un artículo de investigación, llevado a cabo para la materia de Multiprocesadores con el profesor Pedro Oscar Pérez Murueta durante el semestre agosto-diciembre del 2019, que presenta un problema de programación resuelto con las técnicas vistas en clase: *threads* en Java, *Fork/Join* en Java, OpenMP en C, Intel *Threading Building Blocks* (TBB) en C++ y CUDA.

Este artículo pretende desarrollar las habilidades de: dominio del idioma inglés, pensamiento crítico, visión del entorno internacional, comunicación escrita, y uso eficiente de la informática y las telecomunicaciones.

La importancia de una buena programación, de la paralelización de los códigos y de la optimización de los recursos de la computadora ha sido tratado a lo largo del curso y se puede obtener información interesante de artículos como *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software* y *Welcome to the Jungle: Or, A Heterogeneous Supercomputer in Every Pocket*, ambos escritos por Sutter's Mill. (Herb Sutter, 2011) (Herb Sutter, 2005)

El problema seleccionado para este artículo de investigación es la suma y resta de dos vectores, utilizando todas las técnicas de paralelización vistas, debido a que el objetivo de este artículo es la comparación de las distintas herramientas vistas en clase y no el desarrollo complejo de un algoritmo de programación.

La suma y resta de dos vectores consiste en la suma o resta de cada uno de sus componentes, dando como resultado un tercer vector. Por ejemplo, si tenemos un arreglo $A = (3, 5, 7)$ y un arreglo $B = (2, 4, 6)$ el resultado de la sumatoria sería un tercer vector: $C = ((3 + 2), (5 + 4), (7 + 6)) = (5, 9, 13)$ y el resultado de la resta para los mismos vectores sería el vector: $D = ((3 - 2), (5 - 4), (7 - 6)) = (1, 1, 1)$.

Desarrollo

Para desarrollar las distintas técnicas utilizadas es recomendable iniciar resolviendo el problema de forma secuencial, es decir, en C, C++ y Java. Posteriormente, desarrollar la parte paralela (*threads*, *fork/join*, OpenMP, TBB y CUDA).

Para el desarrollo, de manera opcional, es posible utilizar software de entorno de desarrollo integrado (IDE) o algún editor de código fuente, con el objetivo de que resulte más fácil. En este caso, se optó por utilizar Atom. (Atom, 2019)

Todos los códigos elaborados se encontrarán en la sección de [Anexos](#), al final del documento y anexos a la carpeta del mismo. En todas las tecnologías utilizadas, se implementaron vectores de 100 millones de datos cada uno; siendo el primero llenado de manera aleatoria y el segundo con números naturales ascendentes (1, 2, 3, 4...).

Finalmente, cabe mencionar que todas las pruebas fueron realizadas en el servidor del profesor, con el objetivo de obtener datos más certeros y tener constancia del poder de cómputo.

Librería `cheader.h` y `cppheader.h`

Todos los códigos desarrollados utilizan la librería “`cheader.h`” o “`cppheader.h`”, la cual ha sido otorgada por el profesor y sirve para obtener el tiempo de ejecución de un programa, rellenar arreglos con números consecutivos, rellenar arreglos con números aleatorios y mostrar arreglos en la terminal.

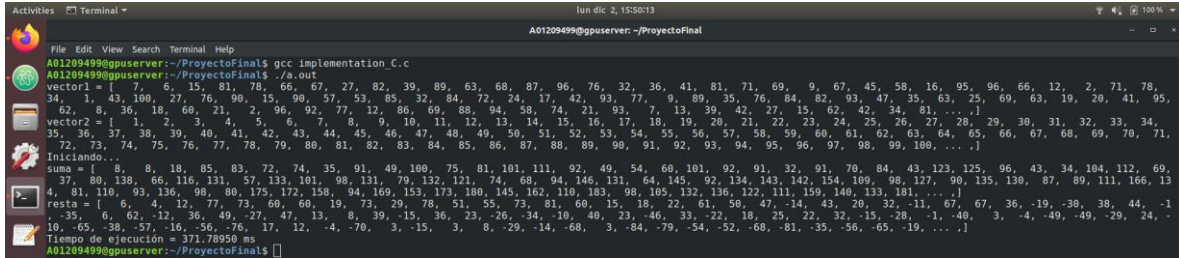
Ambas librerías pueden ser encontradas en la sección de Anexos, al final del documento, junto con los códigos desarrollados.

Secuencial en C

El desarrollo de forma secuencial en C representa la base para su posterior implementación paralelizada en tecnologías como OpenMP o CUDA. Presenta la ventaja de ser un lenguaje orientado a Unix en sus inicios y de ser de fácil implementación. (Brian Kernighan, 1991)

A continuación, se adjunta la Imagen 1, en donde se puede observar una terminal en el servidor compilando el código y ejecutándolo, comprobando que se crean ambos vectores correctamente y que se efectúan la suma y resta de manera correcta.

Finalmente, se puede observar que el tiempo de ejecución del algoritmo es de 371.78950 milisegundos.



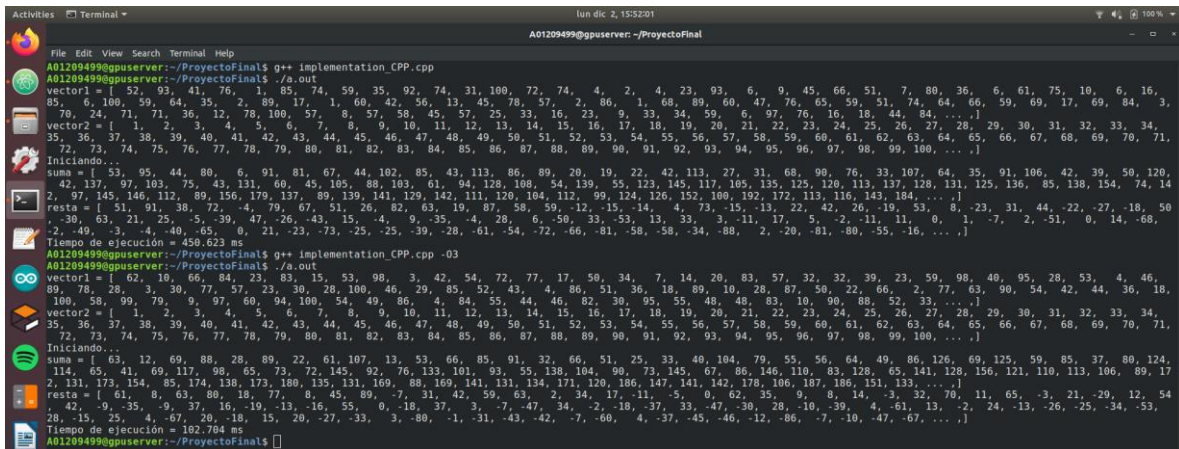
```
Activities Terminal
lun dic 2, 15:50:13
A01209499@ppuserver: ~/ProyectoFinal
A01209499@ppuserver:~/ProyectoFinal$ gcc implementation C.c
A01209499@ppuserver:~/ProyectoFinal$ ./a.out
vector1 = [ 7, 6, 15, 81, 78, 66, 67, 27, 82, 39, 89, 63, 68, 87, 96, 76, 32, 36, 41, 81, 71, 69, 9, 67, 45, 58, 16, 95, 96, 66, 12, 2, 71, 78, 34, 1, 43, 100, 27, 76, 90, 15, 90, 57, 53, 85, 32, 84, 72, 24, 17, 42, 93, 77, 9, 89, 35, 76, 84, 82, 93, 47, 35, 63, 25, 69, 63, 19, 20, 41, 95, 62, 5, 36, 18, 60, 21, 2, 86, 82, 77, 12, 86, 69, 88, 94, 58, 74, 21, 93, 7, 13, 39, 42, 27, 15, 62, 42, 34, 81, ... ]
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ... ]
Iniciando...
suma = [ 8, 8, 18, 85, 83, 72, 74, 35, 91, 49, 100, 75, 81, 101, 111, 92, 49, 54, 60, 101, 92, 91, 32, 91, 70, 84, 43, 123, 125, 96, 43, 34, 104, 112, 69, 37, 80, 138, 66, 116, 131, 57, 133, 101, 98, 131, 79, 132, 121, 74, 68, 94, 146, 131, 64, 145, 92, 134, 143, 142, 154, 109, 98, 127, 90, 135, 130, 87, 89, 111, 166, 13, 4, 81, 110, 93, 136, 98, 80, 175, 172, 158, 94, 169, 153, 173, 180, 145, 162, 110, 183, 98, 105, 132, 136, 122, 111, 159, 148, 133, 181, ... ]
resta = [ 6, 4, 12, 77, 73, 60, 60, 19, 73, 29, 78, 51, 55, 73, 61, 60, 15, 16, 22, 61, 50, 47, 14, 43, 20, 32, 11, 67, 67, 36, 19, 30, 38, 44, 1, 10, 65, 38, 57, 10, 56, 76, 17, 12, 4, 70, 3, 15, 3, 8, 29, 14, 68, 3, 84, 79, 54, 52, 68, 81, 35, 56, 65, 19, ... ]
Tiempo de ejecución = 371.78950 ms
A01209499@ppuserver:~/ProyectoFinal$
```

Imagen 1. Algoritmo en C.

Secuencial en C++

Este lenguaje está basado en C, optimizado para la orientación a objetos. Servirá de base para la implementación de las tecnologías en paralelo. (Balagurusamy, E., 2007)

A continuación, en la Imagen 2, se puede observar la compilación del código y su ejecución. Su tiempo de procesamiento es de 450.623 milisegundos, que es bastante lento por comparación con C, pero este algoritmo presenta una característica de compilación, una optimización de nivel 3, que reduce el tiempo de ejecución a 102.704 milisegundos, haciéndolo más de 3 veces más rápido que el algoritmo en C, que no presenta optimización.



```
Activities Terminal
lun dic 2, 15:52:01
A01209499@ppuserver: ~/ProyectoFinal
A01209499@ppuserver:~/ProyectoFinal$ g++ implementation_CPP.cpp
A01209499@ppuserver:~/ProyectoFinal$ ./a.out
vector1 = [ 52, 93, 41, 76, 1, 85, 74, 59, 35, 92, 74, 31, 100, 72, 74, 4, 2, 4, 23, 93, 6, 9, 45, 66, 51, 7, 80, 36, 6, 61, 75, 10, 6, 16, 85, 6, 100, 59, 64, 35, 2, 89, 17, 1, 60, 42, 56, 13, 45, 78, 57, 2, 86, 1, 68, 89, 60, 47, 76, 65, 59, 51, 74, 64, 66, 59, 69, 17, 69, 84, 3, 70, 24, 71, 71, 36, 12, 78, 100, 57, 8, 57, 58, 45, 57, 25, 33, 16, 23, 9, 33, 34, 59, 6, 97, 76, 16, 18, 44, 84, ... ]
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ... ]
Iniciando...
suma = [ 53, 95, 44, 80, 6, 91, 81, 67, 44, 102, 85, 43, 113, 86, 89, 20, 19, 22, 42, 113, 27, 31, 68, 90, 76, 33, 107, 64, 35, 91, 106, 42, 39, 50, 120, 42, 137, 97, 103, 75, 43, 131, 60, 45, 105, 88, 103, 61, 94, 128, 108, 54, 139, 55, 123, 145, 117, 105, 135, 125, 120, 113, 137, 128, 131, 125, 136, 85, 138, 154, 74, 14, 2, 97, 145, 146, 112, 89, 156, 179, 137, 89, 139, 141, 129, 142, 111, 120, 104, 112, 99, 124, 126, 152, 100, 192, 172, 113, 116, 143, 184, ... ]
resta = [ 51, 91, 30, 72, 4, 79, 67, 51, 26, 62, 63, 19, 67, 58, 59, 12, 15, 14, 4, 73, 15, 13, 22, 42, 26, 19, 53, 8, 23, 31, 44, 22, 27, 18, 50, 38, 63, 21, 25, 5, 39, 47, 26, 43, 15, 4, 9, 35, 4, 28, 6, 50, 33, 53, 13, 33, 3, 11, 17, 5, 2, 11, 11, 8, 1, 7, 2, 51, 0, 14, 68, 2, 49, 3, 4, 40, 65, 0, 21, 23, 73, 25, 25, 39, 28, 61, 54, 72, 66, 81, 58, 58, 34, 88, 2, 20, 81, 80, 55, 16, ... ]
Tiempo de ejecución = 450.623 ms
A01209499@ppuserver:~/ProyectoFinal$ g++ -O3 implementation_CPP.cpp
A01209499@ppuserver:~/ProyectoFinal$ ./a.out
vector1 = [ 62, 10, 66, 84, 23, 83, 15, 53, 98, 3, 42, 54, 72, 77, 17, 50, 34, 7, 14, 20, 83, 57, 32, 32, 39, 23, 59, 98, 40, 95, 28, 53, 4, 46, 89, 78, 28, 3, 30, 77, 57, 23, 30, 28, 100, 46, 29, 85, 52, 43, 4, 86, 51, 36, 18, 89, 10, 26, 87, 50, 22, 66, 2, 77, 63, 90, 54, 42, 44, 36, 18, 100, 50, 90, 79, 9, 97, 60, 84, 100, 54, 49, 86, 4, 84, 55, 44, 46, 82, 30, 95, 55, 48, 48, 83, 10, 90, 80, 52, 33, ... ]
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ... ]
Iniciando...
suma = [ 63, 12, 69, 88, 28, 89, 22, 61, 107, 13, 53, 66, 85, 91, 32, 66, 51, 25, 33, 40, 104, 79, 55, 56, 64, 49, 86, 126, 69, 125, 59, 85, 37, 80, 124, 114, 65, 41, 69, 117, 90, 65, 73, 72, 145, 92, 76, 133, 101, 93, 55, 138, 104, 90, 73, 145, 67, 86, 146, 110, 83, 128, 65, 141, 128, 156, 121, 110, 113, 106, 89, 17, 2, 131, 173, 154, 85, 174, 138, 173, 180, 135, 131, 169, 88, 169, 141, 131, 134, 171, 120, 186, 147, 141, 142, 178, 106, 187, 186, 151, 133, ... ]
resta = [ 61, 8, 63, 80, 18, 77, 8, 45, 89, 7, 31, 42, 59, 63, 2, 34, 17, 11, 5, 0, 62, 35, 9, 8, 14, 3, 32, 70, 11, 65, 3, 21, 29, 12, 54, 42, 9, 35, 9, 37, 16, 19, 13, 16, 55, 0, 18, 37, 3, 7, 47, 34, 2, 18, 37, 33, 47, 30, 28, 10, 39, 4, 61, 13, 2, 24, 13, 26, 25, 34, 53, 28, 15, 25, 4, 67, 20, 18, 15, 20, 27, 33, 3, 80, 1, 31, 43, 42, 7, 60, 4, 37, 45, 46, 12, 86, 7, 10, 47, 67, ... ]
Tiempo de ejecución = 102.704 ms
A01209499@ppuserver:~/ProyectoFinal$
```

Imagen 2. Implementación en C++.

Secuencial en Java

Java se programa de forma diferente a C o C++; es un lenguaje orientado a objetos. Servirá de base para la implementación paralela mediante *Fork/Join* y *Threads*. (J. Steven Perry, 2012)

En la Imagen 3, se puede observar cómo se compila el código y cómo se ejecuta. El tiempo de procesamiento es de 100.29 milisegundos. Los algoritmos en Java no cuentan con optimización (como C++), no obstante, resulta ser el más rápido de ejecución en forma secuencial.

```

Activities Terminal
lun dic 2, 15:55:52
A01209499@ppuserver: ~/ProyectoFinal

File Edit View Search Terminal Help
A01209499@ppuserver:~/ProyectoFinal$ javac calculate_vectors.java
A01209499@ppuserver:~/ProyectoFinal$ java calculate_vectors
vector1 = [ 33, 95, 72, 25, 53, 36, 44, 15, 35, 99, 52, 72, 60, 34, 25, 1, 61, 16, 27, 15, 59, 1, 86, 97, 44, 98, 18, 24, 8, 27, 32, 75, 94, 98,
32, 18, 66, 6, 27, 75, 77, 76, 62, 46, 10, 73, 18, 66, 28, 24, 20, 79, 74, 67, 29, 78, 63, 50, 46, 91, 45, 28, 77, 68, 47, 100, 47, 76, 60, 36, 19,
34, 92, 56, 53, 84, 31, 73, 21, 39, 17, 96, 35, 74, 44, 62, 70, 11, 17, 23, 45, 2, 98, 37, 97, 46, 77, 75, 11, 13, ..., ]
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ..., ]
suma = [ 34, 97, 75, 29, 58, 42, 51, 23, 42, 109, 63, 84, 73, 40, 49, 17, 78, 34, 46, 35, 80, 23, 109, 121, 69, 124, 45, 52, 37, 57, 63, 107, 127, 132, 67,
54, 182, 44, 66, 115, 118, 118, 105, 90, 55, 119, 65, 114, 77, 74, 71, 131, 127, 121, 84, 134, 120, 188, 105, 151, 106, 90, 140, 132, 112, 166, 114, 144, 129, 106, 90, 10
6, 165, 130, 128, 160, 108, 151, 100, 119, 98, 178, 118, 158, 129, 148, 157, 99, 106, 113, 136, 94, 191, 131, 192, 142, 174, 173, 110, 113, ..., ]
resta = [ 32, 93, 69, 21, 48, 30, 37, 7, 24, 89, 41, 60, 47, 20, 10, -15, 44, -2, 8, -5, 38, -21, 63, 73, 19, 72, -9, -4, -21, -3, 1, 43, 61, 64, -3
, -18, 29, -32, -12, 35, 36, 34, 19, 2, -35, 27, -29, 18, -21, -26, -31, 27, 21, 13, -26, 22, 6, -8, -13, 31, -16, -34, 14, 4, -18, 34, -20, 8, -9, -34, -52, -
38, 19, -18, -22, 0, -46, -5, -58, -41, -64, 14, -48, -10, -41, -24, -17, -77, -72, -67, -46, -90, 3, -57, 2, -50, -20, -23, -88, -87, ..., ]
Tiempo de ejecución = 100.29080 ms
A01209499@ppuserver:~/ProyectoFinal$
  
```

Imagen 3. Implementación en Java.

Paralelo en OpenMP

El primer algoritmo de ejecución en paralelo es el favorito de muchos, OpenMP. Este algoritmo resulta ser muy fácil de paralelizar, una vez elaborado el código secuencial. El lenguaje OpenMP resulta ser una API enfocada a la programación multiproceso, con memoria compartida. (Leonardo Dagum, 1998) (OpenMP, 2019)

El algoritmo desarrollado es en gran parte el código secuencial. En la Imagen 4 se puede observar cómo se compila y ejecuta, obteniendo un tiempo de 98.2668 milisegundos, representando un avance respecto a los secuenciales; pero la verdadera mejora es al realizarle una optimización de nivel 3, obteniendo 74.8306 milisegundos.

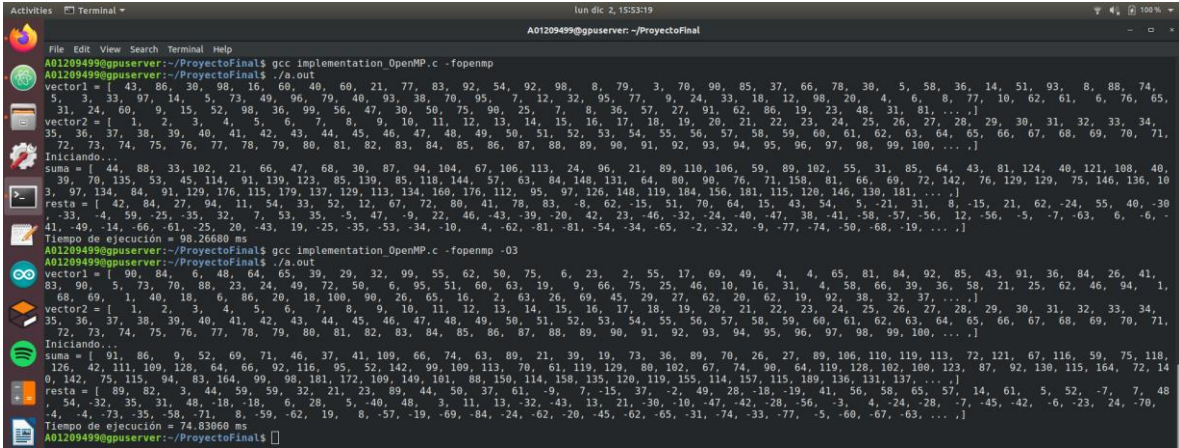


Imagen 4. Implementación en OpenMP.

Paralelo TBB

Intel *Threading Building Blocks* (TBB) es una biblioteca actualizada constantemente que se puede encontrar en GitHub, desarrollada por Intel específicamente para potenciar el paralelismo de códigos escritos en C++; una alternativa al uso de *Threads*. (Intel Corporation, 2019)

La Imagen 5 contiene su compilación y ejecución, dando un tiempo de ejecución de 119.506 milisegundos, comparablemente mejor que el tiempo en C++. Además, como C++, también cuenta con una optimización de hasta nivel 3, reduciendo su tiempo hasta 73.1463 milisegundos.

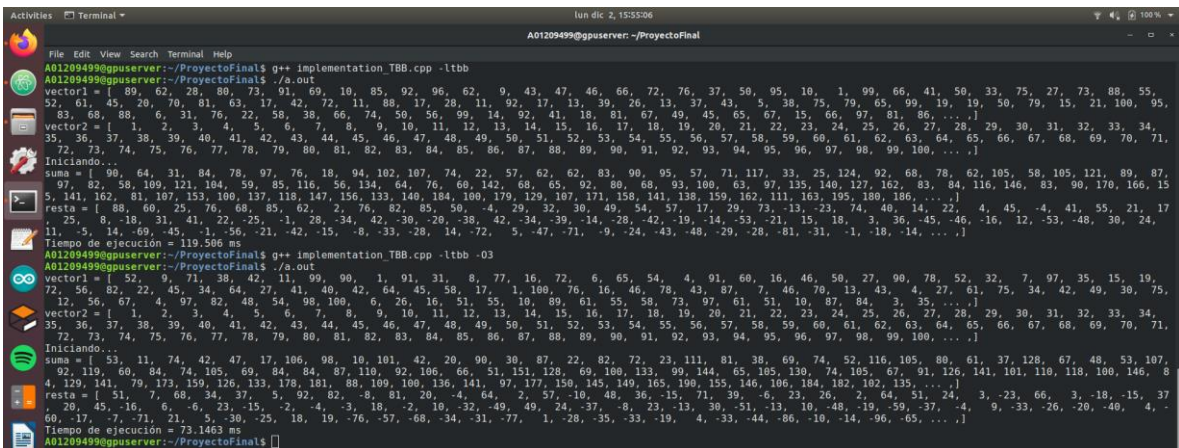


Imagen 5. Implementación en TBB.

Paralelo en CUDA

CUDA es una tecnología muy particular. Es una plataforma desarrollada por NVIDIA para otorgar herramientas de desarrollo, para ser usada en su software, ya que es enfocado al procesamiento de algoritmos en paralelo en una GPU y no en CPU, como el resto de las tecnologías, lo que resulta en un procesamiento increíblemente rápido, sobre todo en aplicaciones de gráficos (como procesamiento de imágenes). (NVIDIA Corporation, 2019)

En la Imagen 6, se puede observar su compilación y ejecución, dando un increíble tiempo de ejecución de 0.00410 milisegundos, es decir, 4.1 microsegundos; siendo, por mucho, la tecnología más rápida.

```

Activities Terminal
lun dic 2, 16:21:17
A01209499@ppuserver: ~/ProyectoFinals
A01209499@ppuserver:~/ProyectoFinals$ nvcc implementation_CUDA.cu -o implementation_CUDA
nvcc warning: The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
implementation_CUDA.cu(70): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(71): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(98): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(99): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(70): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(71): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(98): warning: conversion from a string literal to 'char **' is deprecated
implementation_CUDA.cu(99): warning: conversion from a string literal to 'char **' is deprecated
A01209499@ppuserver:~/ProyectoFinals$ ./implementation_CUDA
a = [ 10, 46, 68, 25, 61, 42, 57, 34, 33, 6, 88, 100, 72, 10, 58, 42, 26, 97, 60, 20, 31, 77, 57, 48, 14, 82, 35, 75, 97, 60, 10, 58, 58, 79, 83, 1,
8, 70, 39, 3, 55, 96, 91, 54, 67, 100, 63, 61, 77, 59, 72, 48, 42, 48, 57, 89, 61, 90, 75, 87, 86, 35, 96, 44, 44, 24, 26, 61, 45, 64, 16, 99, 12,
58, 4, 30, 9, 67, 90, 37, 77, 13, 36, 18, 12, 92, 59, 72, 81, 33, 10, 19, 19, 5, 62, 62, 80, 87, 75, 25, 2, ..., l
b = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 3
6, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ..., l
Iniciando...
c = [ 11, 48, 71, 29, 66, 48, 64, 42, 42, 16, 99, 112, 85, 24, 73, 58, 43, 115, 79, 40, 52, 99, 80, 72, 39, 108, 62, 103, 126, 90, 41, 90, 91, 63, 118, 5
4, 107, 77, 42, 95, 137, 133, 97, 111, 145, 109, 108, 125, 108, 122, 99, 94, 101, 111, 144, 117, 147, 133, 146, 146, 96, 158, 107, 108, 89, 92, 128, 113, 133, 86, 170, 84,
131, 78, 105, 85, 144, 168, 116, 157, 94, 118, 101, 96, 177, 145, 159, 169, 122, 100, 110, 111, 98, 156, 157, 176, 184, 173, 124, 102, ..., l
d = [ 9, 44, 65, 21, 56, 36, 50, 26, 24, -4, 77, 80, 59, -4, 43, 26, 9, 79, 41, 0, 10, 55, 34, 24, -11, 56, 8, 47, 68, 30, -21, 26, 25, -5, 48, -1
8, 33, 1, -36, 15, 55, 49, 11, 23, 55, 17, 14, 29, 10, 22, -3, -10, -5, 3, 34, 5, 33, 17, 20, 20, -20, 34, -19, -20, -41, -40, -6, -23, -5, -54, 20, -60,
-15, -70, -45, -67, -10, 12, -42, -3, -68, -46, -65, -72, 7, -27, -15, -7, -56, -80, -72, -73, -88, -32, -33, -16, -10, -23, -74, -98, ..., l
Tiempo de ejecución = 0.00410 ms
A01209499@ppuserver:~/ProyectoFinals$

```

Imagen 6. Implementación en CUDA.

Paralelo con *Threads* en Java

Java presenta dos formas de paralelización, mediante el uso de *Threads* y mediante el uso de *Fork/Join*. La primera es una tecnología que permite a Java crear objetos (*Threads*) con el aforo de correr simultáneamente al método principal. Una forma de verlo es como contar con dos variables *program counter* en un mismo algoritmo; es decir, el mismo algoritmo corre en diferentes núcleos del procesador. (González, 2010)

Dentro de la Imagen 7 se puede observar cómo se compila el algoritmo y cómo se ejecuta, obteniendo un tiempo de ejecución de 65.3 milisegundos. Resultando ser más rápido que Java. Ya que es basado en Java, no existe una optimización como en C++.

```

A01209499@ppuserver:~/ProyectoFinal$ javac calculate_vectors JavaThreads.java
A01209499@ppuserver:~/ProyectoFinal$ java calculate_vectors JavaThreads
vector1 = [ 62, 100, 5, 66, 78, 77, 18, 50, 18, 60, 39, 87, 23, 27, 23, 89, 68, 60, 62, 22, 61, 63, 9, 98, 75, 85, 95, 42, 31, 65, 68, 43, 89, 5, 3, 9, 21, 50, 81, 67, 33, 45, 41, 93, 48, 58, 7, 20, 70, 40, 46, 56, 72, 67, 45, 44, 100, 74, 73, 43, 44, 75, 30, 31, 64, 10, 49, 58, 25, 46, 43, 38, 89, 2, 88, 78, 57, 13, 89, 71, 50, 92, 51, 13, 23, 87, 34, 34, 44, 3, 42, 13, 69, 56, 90, 73, 43, 86, 99, 25, ..., 1
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ..., 1
suma = [ 63, 102, 6, 78, 83, 83, 25, 58, 27, 78, 39, 99, 36, 41, 38, 105, 85, 78, 81, 42, 82, 85, 32, 122, 100, 111, 122, 70, 68, 95, 99, 75, 122, 39, 38, 45, 58, 88, 120, 107, 74, 87, 84, 137, 93, 104, 54, 68, 127, 99, 97, 108, 125, 121, 100, 100, 157, 132, 132, 103, 105, 137, 93, 95, 129, 82, 116, 126, 94, 116, 114, 10, 2, 162, 76, 163, 154, 134, 91, 168, 151, 131, 174, 134, 97, 108, 173, 121, 122, 133, 93, 133, 105, 162, 150, 185, 169, 140, 184, 198, 125, ..., 1
resta = [ 61, 98, 2, 62, 73, 71, 11, 42, 9, 58, 28, 75, 10, 13, 8, 73, 51, 42, 43, 2, 48, 41, -14, 74, 50, 59, 68, 14, 2, 35, 37, 11, 56, -29, -32, -27, -16, 12, 42, 27, -8, 3, -2, 49, 3, 12, -40, -28, 29, -10, -5, 4, 19, 13, -10, -12, 43, 16, 14, -17, -17, 13, -33, -33, -1, -50, -18, -10, -44, -24, -28, -42, 16, -72, 13, 2, -20, -65, 10, -9, -31, 10, -32, -71, -62, 1, -53, -54, -45, -87, -49, -79, -24, -38, -5, -23, -54, -12, 0, -75, ..., 1
Tiempo de ejecución = 65.30800 ms
A01209499@ppuserver:~/ProyectoFinal$
  
```

Imagen 7. Implementación con *Threads* en Java.

Paralelo en Java con Fork/Join

Fork/Join es un *framework* de Java que permite realizar algoritmos a ejecutar en múltiples núcleos del procesador al máximo, en donde los subprocesos a paralelizar sean independientes entre sí; su manera de funcionar es distribuyendo el procesamiento en un *pool* de *Threads*. (Oracle, 2019)

En la Imagen 8, se observa el tiempo de ejecución en 0.21 milisegundos, siendo muchísimo mejor que Java y que Java con *Threads*. Tampoco cuenta con optimización, al ser basado en Java y no en C++; no obstante, la mejora es realmente significativa.

```

A01209499@ppuserver:~/ProyectoFinal$ javac calculate_vectors FORKJOIN.java
A01209499@ppuserver:~/ProyectoFinal$ java calculate_vectors FORKJOIN
vector1 = [ 65, 100, 86, 82, 62, 73, 57, 68, 11, 10, 21, 79, 64, 60, 17, 87, 16, 73, 88, 68, 50, 48, 56, 39, 54, 49, 47, 5, 88, 100, 88, 15, 65, 22, 49, 87, 12, 43, 50, 97, 49, 29, 87, 86, 40, 53, 77, 29, 75, 60, 49, 67, 1, 40, 91, 85, 46, 53, 41, 62, 10, 93, 45, 12, 55, 59, 83, 33, 94, 50, 96, 45, 62, 23, 85, 17, 73, 39, 76, 8, 60, 88, 18, 84, 33, 70, 69, 74, 97, 66, 33, 41, 86, 76, 62, 50, 21, 31, 8, 54, ..., 1
vector2 = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, ..., 1
suma = [ 66, 102, 89, 86, 67, 79, 64, 76, 20, 20, 32, 91, 77, 74, 32, 103, 33, 91, 107, 88, 71, 70, 79, 63, 79, 75, 74, 33, 117, 130, 111, 47, 98, 56, 84, 123, 49, 81, 97, 137, 90, 71, 110, 130, 85, 99, 124, 77, 124, 110, 100, 119, 54, 94, 146, 141, 103, 111, 100, 122, 77, 155, 168, 76, 120, 125, 150, 101, 163, 126, 167, 11, 7, 135, 97, 160, 93, 150, 117, 155, 88, 141, 170, 101, 160, 118, 156, 156, 162, 166, 156, 124, 133, 179, 170, 157, 155, 118, 129, 107, 154, ..., 1
resta = [ 64, 98, 83, 78, 57, 67, 50, 60, 2, 0, 10, 67, 51, 46, 2, 71, -1, 55, 69, 48, 29, 26, 33, 15, 29, 23, 20, 23, 59, 70, 49, -17, 32, -12, 14, 51, -25, 5, 19, 57, 8, -13, 24, 42, -5, 7, 30, -19, 26, 10, -2, 15, -52, -14, 36, 29, -11, -5, -18, 2, -45, 31, -18, -52, -10, -7, 16, -35, 25, -14, 25, -27, -11, -51, 10, -59, -2, 39, -3, -72, -21, 6, -65, 0, -52, -16, -18, -14, 8, -24, -58, -51, -7, -18, -33, -37, -76, -67, -91, -46, ..., 1
Tiempo de ejecución = 0.21000 ms
A01209499@ppuserver:~/ProyectoFinal$
  
```

Imagen 8. Implementación en Java con Fork/Join

Análisis de resultados

Comparativa entre tecnologías secuenciales

El principal objetivo de este artículo de investigación es el análisis de los resultados obtenidos, mediante una comparativa de los mismos. Se obtendrá el *SpeedUp*, que es la división del tiempo de ejecución en secuencial sobre el tiempo de ejecución en paralelo:

$$SpeedUp = \frac{Tiempo\ Secuencial}{Tiempo\ Paralelo}$$

A continuación, se presenta una tabla comparativa de las tecnologías secuenciales y sus tiempos de ejecución:

Comparativa de Tecnologías Secuenciales		
Tecnología	Tiempo de ejecución	Tiempo con optimización
C	371.78950	-
C++	450.62300	102.70400
Java	100.29000	-

Tabla 1: Comparativa de tecnologías secuenciales.

Por lo tanto, podemos determinar que Java tiene un procesamiento de hasta 4 veces más rápido que C y C++, en promedio. Sin embargo, con la optimización de C++ se logra una gran mejora, quedando casi a la par que Java.

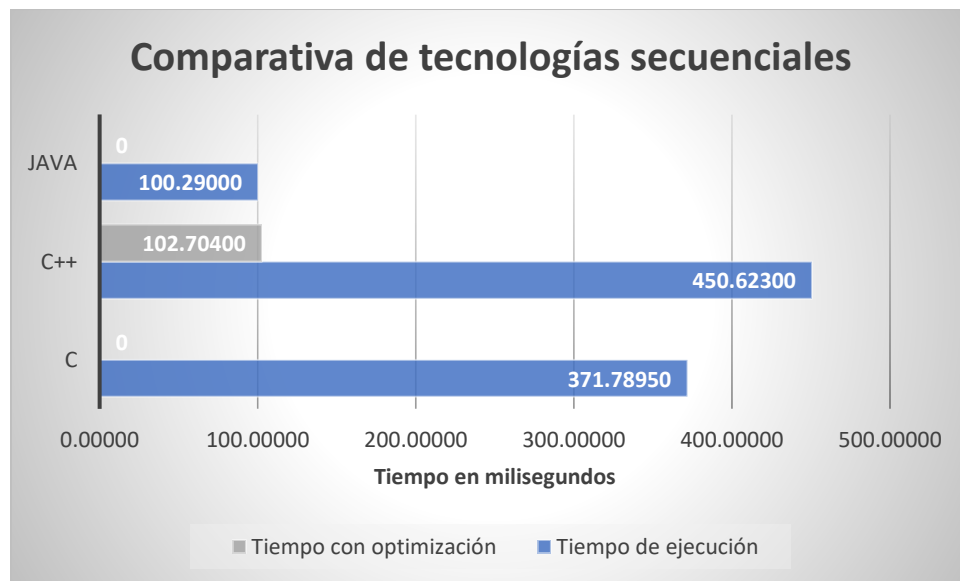


Gráfico 1: Comparativa entre tecnologías secuenciales.

Comparativa entre tecnologías paralelas

Continuando, se realizará la misma comparativa para las tecnologías paralelas:

Comparativa de Tecnologías Paralelas		
Tecnología	Tiempo de ejecución	Tiempo con optimización
OpenMP	98.26680	74.83060
TBB	119.50600	73.14630
CUDA	0.00390	-
Threads	65.30000	-
Fork/Join	0.21000	-

Tabla 2: Comparativa de tecnologías Paralelas.

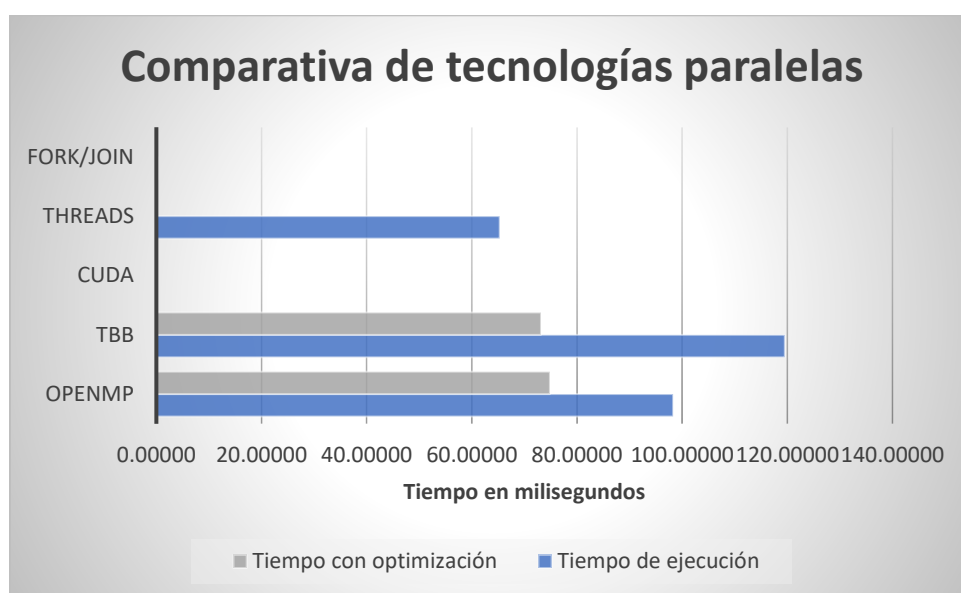


Gráfico 2: Comparativa entre tecnologías paralelas.

Se puede apreciar que la tecnología que menor tiempo de procesamiento tiene es CUDA y la más próxima a ella es Java con Fork/Join, que sigue estando muy alejada.

CUDA sobre Fork/Join representa una diferencia de velocidad de $\frac{0.21}{0.0039} = 53.846$.

La razón de la gran rapidez de CUDA es que utiliza el procesamiento en GPU y no en CPU. Para fines de la comparativa, podemos verlo como que CUDA tiene muchísimos núcleos de procesamiento (tantos como tenga la tarjeta de NVIDIA), mientras que el CPU se limita a 8 o 16, generalmente, de Intel.

Comparativa entre secuenciales y paralelas de Java

Comenzando con la tecnología de Java, se realizará la comparativa entre los tiempos de ejecución de forma secuencial y paralela:

Comparativa entre Secuencial y Paralelo de Java		
Tecnología	Tiempo de ejecución	SpeedUp
Java	100.29000	-
Threads	65.30000	1.53583
Fork/Join	0.21000	477.57143

Tabla 3: Comparativa entre tiempos de ejecución en secuencial y paralelo de Java.



Gráfico 3: Comparativa entre tiempos de ejecución en secuencial y paralelo de Java.

En esta comparativa resulta evidente la ventaja en el uso de la tecnología en paralelo de *Fork/Join* para la ejecución del algoritmo, obteniendo un SpeedUp increíble de 477.57.

La razón de esto es que la tecnología *Fork/Join* utiliza un *pool* de *Threads* para resolver el algoritmo paralelizable, creando una aceleración impresionante respecto a Java.

Comparativa entre secuencial y paralela de C y C++

Las tecnologías C y C++ dan paso a las tecnologías de OpenMP, TBB y CUDA, por esta razón se eligió realizar una comparativa entre todas las anteriores y no solo algunas de ellas. Resultarán interesantes los resultados presentados a continuación:

Comparativa tecnologías secuenciales y paralelas de C y C++			
Tecnología	Tiempo de ejecución	SpeedUp respecto a C	SpeedUp respecto a C++
C	371.78950	-	-
C++	450.62300	-	-
OpenMP	98.26680	3.783470104	4.585709517
TBB	119.50600	3.111053002	3.770714441
CUDA	0.00390	95330.64103	115544.359

Tabla 4.1: Comparativa tecnologías secuenciales y paralelas de C y C++

Comparativa tecnologías secuenciales y paralelas de C++ optimizadas		
Tecnología	Tiempo con optimización	SpeedUp respecto a C++
C++	102.70400	-
OpenMP	74.83060	1.372486657
TBB	73.14630	1.404090159
CUDA	0.00390	26334.35897

Tabla 4.2: Comparativa tecnologías secuenciales y paralelas de C++ optimizadas

Sin la necesidad de presentar un gráfico, es evidente que CUDA resulta ser, por mucho, la tecnología con menor tiempo de procesamiento, sin importar que se le compare con o sin optimización a las demás tecnologías. El motivo de esto es explicado en la sección [Comparativa entre tecnologías paralelas](#), de este documento.

Dejando de lado a CUDA, se puede concluir que los mejores *SpeedUp* son aquellos respecto a C++ sin optimización, pero una vez que se optimizan, el *SpeedUp* disminuye. No obstante, se puede concluir, al igual que en la comparativa de Java, que las tecnologías paralelas ayudan enormemente a aprovechar los recursos de un multiprocesador.

Comparativa de todas las tecnologías implementadas:

Finalmente, a manera de resumen, una comparativa de todas las tecnologías implementadas en este artículo de investigación:

Comparativa de Tecnologías		
Tecnología	Tiempo de ejecución	Tiempo con optimización
C	371.78950	-
C++	450.62300	102.70400
Java	100.29000	-
OpenMP	98.26680	74.83060
TBB	119.50600	73.14630
CUDA	0.00390	-
Threads	65.30000	-
Fork/Join	0.21000	-

Tabla 5: Comparativa de todas las tecnologías implementadas.

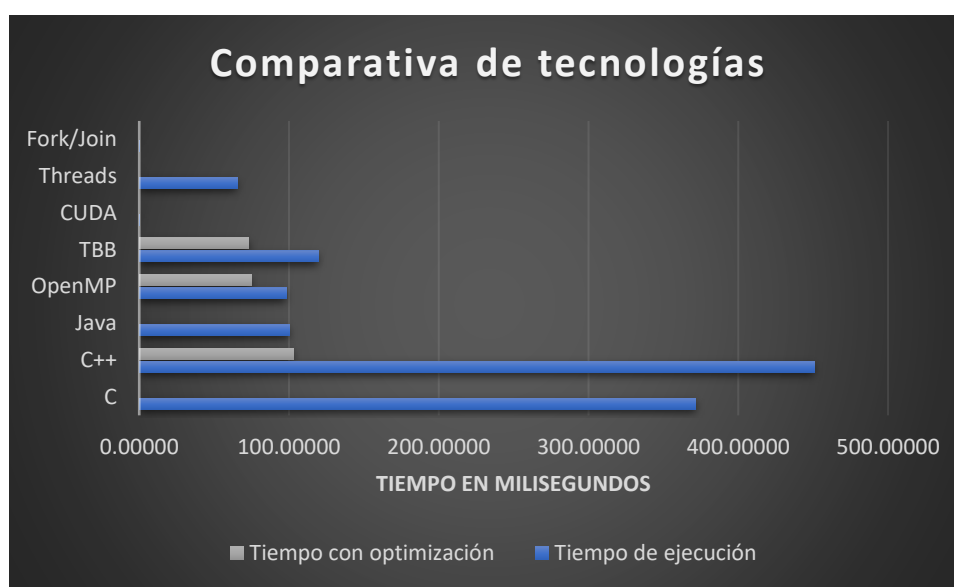


Gráfico 5: Comparativa de todas las tecnologías implementadas.

Conclusión

Con base en los resultados obtenidos en este artículo de investigación, podemos concluir que la solución de algoritmos en paralelo representa un aprovechamiento de los recursos disponibles del dispositivo, específicamente del procesador o GPU, dando como resultado la resolución del problema en un tiempo extremadamente inferior a la forma secuencial.

En este artículo se realizó la prueba con un algoritmo “sencillo”, de operaciones con vectores, pero cuando hablamos de otro algoritmo más complejo, el *SpeedUp* obtenido será mucho más apreciable.

Me gustaría recalcar que la tecnología de CUDA fue superior la resto, logrando un excepcional tiempo de ejecución. Sin olvidar que el procesamiento es realizado en el GPU, por lo que el primer paso es contar con uno... y no es el caso de todas las personas. Además, he de señalar que CUDA, al ser resuelto en GPU, trabajará mejor para problemas asociados a vectores, matrices y todo lo relacionado al procesamiento de imágenes, entre otros. He de recordar también que CUDA entiende el cómputo heterogéneo, es decir, CPU y GPU combinados para obtener lo mejor de cada uno.

Consecutivamente, he de marcar que la solución del algoritmo mediante la técnica de *Fork/Join* en Java resultó la mejor opción (sin tomar en cuenta CUDA ni GPU). Logrando un increíble *SpeedUp* entre el mismo Java secuencial y Java con *Threads*.

Finalmente, en cuanto a las soluciones en forma secuencial, Java también resultó ganador.

Para concluir, si he de optar por tomar un lenguaje en particular, me quedo con Java, porque tanto en la parte secuencial como paralela resultó ser el mejor; con excepción de CUDA, pero CUDA requiere de un GPU.

Referencias

- Atom. (2019). *Atom*. Obtenido de Atom: <https://atom.io/>
- Balagurusamy, E. (2007). *Programación orientada a objetos con C++*. Madrid: McGraw-Hill.
- Brian Kernighan, D. R. (1991). *El lenguaje de programación C*. Pearson Educación.
- González, A. (Agosto de 2010). *Hilos (Threads) en Java*. Obtenido de Departamento de Electrónica | Universidad Técnica Federico Santa María: <http://profesores.elo.utfsm.cl/~agv/elo330/2s10/lectures/Java/threads/JavaThreads.html>
- Herb Sutter. (30 de marzo de 2005). *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Obtenido de <http://www.gotw.ca/publications/concurrency-ddj.htm>
- Herb Sutter. (2011). *Welcome to the Jungle*. Obtenido de Herb Sutter: <https://herbsutter.com/welcome-to-the-jungle/>
- Intel Corporation. (2019). *Intel®Threading Building Blocks*. Obtenido de Intel: <https://software.intel.com/en-us/tbb>
- J. Steven Perry. (03 de diciembre de 2012). *Conceptos básicos del lenguaje Java*. Obtenido de IBM.com: <https://www.ibm.com/developerworks/ssa/java/tutorials/j-introtojava1/index.html>
- Leonardo Dagum, R. M. (1998). *OpenMP: An IndustryStandard API for SharedMemory Programming*. IEEE COMPUTATIONAL SCIENCE & ENGINEERING.
- NVIDIA Corporation. (2019). *CUDA Zone*. Obtenido de NVIDIA: <https://developer.nvidia.com/cuda-zone>
- OpenMP. (2019). *OpenMP Books*. Obtenido de OpenMP: <https://www.openmp.org/resources/openmp-books/>

Oracle. (2019). *Fork/Join*. Obtenido de The Java Tutorials:
<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

Anexos

Librería cheader.h

```

#ifndef TIMER_H_
#define TIMER_H_

#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>

#define N 10
#define DISPLAY 100
#define MAX_VALUE 10000

struct timeval startTime, stopTime;
int started = 0;

void start_timer() {
    started = 1;
    gettimeofday(&startTime, NULL);
}

double stop_timer() {
    long seconds, useconds;
    double duration = -1;

    if (started) {
        gettimeofday(&stopTime, NULL);
        seconds = stopTime.tv_sec - startTime.tv_sec;
        useconds = stopTime.tv_usec - startTime.tv_usec;
        duration = (seconds * 1000.0) + (useconds /
1000.0);
        started = 0;
    }
    return duration;
}

void random_array(int *array, int size) {
    int i;
    srand(time(0));
    for (i = 0; i < size; i++) {
        array[i] = (rand() % 100) + 1;
    }
}

```

```
void fill_array(int *array, int size) {
    int i;
    for (i = 0; i < size; i++) {
        array[i] = (i % MAX_VALUE) + 1;
    }
}

void display_array(char *text, int *array) {
    int i;
    printf("%s = [%4i", text, array[0]);
    for (i = 1; i < DISPLAY; i++) {
        printf(",%4i", array[i]);
    }
    printf(", ... ,]\n");
}

#endif /* TIMER_H_ */
```

Librería cppheader.h

```
#ifndef TIMER_H
#define TIMER_H

#include <ctime>
#include <cstdio>
#include <cstdlib>
#include <sys/time.h>
#include <sys/types.h>

const int N = 10;
const int DISPLAY = 100;
const int MAX_VALUE = 10000;

class Timer {
private:
    timeval startTime;
    bool started;

public:
    Timer() :started(false) {}
    void start(){
        started = true;
        gettimeofday(&startTime, NULL);
    }

    double stop(){
        timeval endTime;
        long seconds, useconds;
        double duration = -1;
        if (started) {
            gettimeofday(&endTime, NULL);
            seconds = endTime.tv_sec - startTime.tv_sec;
            useconds = endTime.tv_usec -
startTime.tv_usec;
            duration = (seconds * 1000.0) + (useconds /
1000.0);
            started = false;
        }
        return duration;
    }
};

void random_array(int *array, int size) {
    int i;
```

```
        srand(time(0));
        for (i = 0; i < size; i++) {
            array[i] = (rand() % 100) + 1;
        }
    }

void fill_array(int *array, int size) {
    int i;
    for (i = 0; i < size; i++) {
        array[i] = (i % MAX_VALUE) + 1;
    }
}

void display_array(const char *text, int *array) {
    int i;
    printf("%s = [%4i", text, array[0]);
    for (i = 1; i < DISPLAY; i++) {
        printf(",%4i", array[i]);
    }
    printf(", ... ,]\n");
}

#endif
```


Implementación en C

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: C.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
//  
// LIBRERÍAS  
//  
// -----  
-----//  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include "utils/cheader.h"  
  
// -----  
-----//  
// DEFINIR  
VARIABLES QUE NO CAMBIAN  
//  
// -----  
-----//  
  
#define SIZE 100000000  
  
// -----  
-----//
```

```
//
                                FUNCIÓN
                                //
// -----
-----//

void calculate_vectors (int* suma, int* resta, int* vector1,
int* vector2) {
    for (int i = 0; i < SIZE; i++) {
        suma[i] = vector1[i] + vector2[i];
        resta[i] = vector1[i] - vector2[i];
    }
}

// -----
-----//
//
                                MAIN
                                //
// -----
-----//

int main(int argc, char const *argv[]) {
    // Definir variables a utilizar:
    int *vector1, *vector2, *suma, *resta;
    double ms = 0.0;

    // Inicializar variables:
    vector1 = (int *) malloc(sizeof(int) * SIZE);
    vector2 = (int *) malloc(sizeof(int) * SIZE);
    suma = (int *) malloc(sizeof(int) * SIZE);
    resta = (int *) malloc(sizeof(int) * SIZE);

    // Rellenar vectores utilizados:
    random_array(vector1, SIZE);
    fill_array(vector2, SIZE);

    // Mostrar ambos vectores:
    display_array("vector1", vector1);
    display_array("vector2", vector2);

    // Imprimir mensaje de inicio:
    printf("Iniciando... \n");

    // Método FOR para ejecutar la función:
    for (int i = 0; i < N; i++) {
        start_timer();
    }
}
```

```
        calculate_vectors(suma, resta, vector1, vector2);
        ms += stop_timer();
    }

    // Imprimir/Mostrar en la terminal los vectores
    resultantes:
    display_array("suma", suma);
    display_array("resta", resta);

    // Imprimir/Mostrar en terminal el tiempo de ejecución del
    programa:
    printf("Tiempo de ejecución = %.5lf ms \n", (ms / N));

    // Liberar la memoria utilizada por las variables:
    free(vector1);
    free(vector2);
    free(suma);
    free(resta);

    // Fin del programa:
    return 0;
}
```

Implementación en C++

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: C++.  
  
*  
  
*-----  
-----*/  
  
  
// -----  
-----//  
//  
  
LIBRERÍAS  
  
// -----  
-----//  
  
#include <iostream>  
#include "utils/cppheader.h"  
  
// -----  
-----//  
//  
DEFINIR NAMESPACE  
//  
// -----  
-----//  
  
using namespace std;  
  
// -----  
-----//
```

```
//
// -----
// -----//
const int SIZE = 100000000;

// -----
// -----//
//
// FUNCIÓN
//
// -----
// -----//

class calculate_vectors {
private:
    int *vector1, *vector2, *suma, *resta, size;

public:
    calculate_vectors(int *arrayD, int *arrayC, int *arrayA,
int *arrayB, int s)
        : resta(arrayD), suma(arrayC), vector1(arrayA),
vector2(arrayB), size(s) {}

    void calculate() {
        for (int i = 0; i < size; i++) {
            suma[i] = vector1[i] + vector2[i];
            resta[i] = vector1[i] - vector2[i];
        }
    }
};

// -----
// -----//
//
// MAIN
//
// -----
// -----//

int main(int argc, char const *argv[]) {
    // Definir variables a utilizar:
    Timer t;
    double ms = 0.0;
```

```

// Inicializar variables:
int *vector1 = new int[SIZE];
int *vector2 = new int[SIZE];
int *suma = new int[SIZE];
int *resta = new int[SIZE];

// Rellenar vectores utilizados:
random_array(vector1, SIZE);
fill_array(vector2, SIZE);

// Mostrar ambos vectores:
display_array("vector1", vector1);
display_array("vector2", vector2);

// Inicializar a la función:
calculate_vectors cv(resta, suma, vector1, vector2,
SIZE);

// Imprimir mensaje de inicio:
cout << "Iniciando..." << endl;

// Método FOR para ejecutar la función:
for (int i = 0; i < N; i++) {
    t.start();
    cv.calculate();
    ms += t.stop();
}

// Imprimir/Mostrar en la terminal los vectores
resultantes:
display_array("suma", suma);
display_array("resta", resta);

// Imprimir/Mostrar en terminal el tiempo de ejecución
del programa:
cout << "Tiempo de ejecución = " << (ms/N) << " ms" <<
endl;

// Liberar la memoria utilizada por las variables:
delete [] vector1;
delete [] vector2;
delete [] suma;
delete [] resta;

// Fin del programa:
return 0;
}

```

Implementación en Java

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: JAVA.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
  
                                FUNCIÓN  
  
// -----//  
-----//  
  
public class calculate_vectors {  
    private int vector1[], vector2[], suma[], resta[];  
  
    public calculate_vectors(int resta[], int suma[], int  
vector1[], int vector2[]) {  
        this.resta = resta;  
        this.suma = suma;  
        this.vector1 = vector1;  
        this.vector2 = vector2;  
    }  
  
    public void calculate() {  
        for (int i = 0; i < suma.length; i++) {  
            suma[i] = vector1[i] + vector2[i];  
            resta[i] = vector1[i] - vector2[i];  
        }  
    }  
}
```



```
// -----
-----//
//
//                                MAIN
//
// -----
-----//

public static void main(String args[]) {
    // Definir variables a utilizar:
    final int SIZE = 100_000_000;
    long startTime = 0;
    long stopTime = 0;
    double acum = 0.0;

    // Inicializar variables:
    int vector1[] = new int[SIZE];
    int vector2[] = new int[SIZE];
    int suma[] = new int[SIZE];
    int resta[] = new int[SIZE];

    // Rellenar vectores utilizados:
    Utils.randomArray(vector1);
    Utils.fillArray(vector2);

    // Mostrar ambos vectores:
    Utils.displayArray("vector1", vector1);
    Utils.displayArray("vector2", vector2);

    // Llamada a la función:
    calculate_vectors e = new calculate_vectors(resta,
suma, vector1, vector2);

    // Método FOR para ejecutar la función:
    for (int i = 0; i < Utils.N; i++) {
        startTime = System.currentTimeMillis();
        e.calculate();
        stopTime = System.currentTimeMillis();
        acum += (stopTime - startTime);
    }

    // Imprimir/Mostrar en la terminal los vectores
resultantes:
    Utils.displayArray("suma", suma);
    Utils.displayArray("resta", resta);
}
```

```
        // Imprimir/Mostrar en terminal el tiempo de
ejecución del programa:
        System.out.printf("Tiempo de ejecución = %.5f",
(acum / Utils.N));
        System.out.printf(" ms \n");
    }
}
```

Implementación en OpenMP

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: OpenMP.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
  
LIBRERÍAS  
  
// -----  
-----//  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include "utils/cheader.h"  
  
// -----  
-----//  
// DEFINIR  
VARIABLES QUE NO CAMBIAN  
//  
// -----  
-----//  
  
#define SIZE 100000000  
  
// -----  
-----//
```

```
//
                                FUNCIÓN
                                //
// -----
-----//

void calculate_vectors(int *resta, int *suma, int *vector1,
int *vector2, int size) {
    #pragma omp parallel for shared(resta, suma, vector1,
vector2, size)
    for (int i = 0; i < size; i++) {
        suma[i] = vector1[i] + vector2[i];
        resta[i] = vector1[i] - vector2[i];
    }
}

// -----
-----//
//
                                MAIN
                                //
// -----
-----//

int main(int argc, char const *argv[]) {
    // Definir variables a utilizar:
    int *vector1, *vector2, *suma, *resta;
    double ms = 0.0, result = 0.0;

    // Inicializar variables:
    vector1 = (int *) malloc(sizeof(int) * SIZE);
    vector2 = (int *) malloc(sizeof(int) * SIZE);
    suma = (int *) malloc(sizeof(int) * SIZE);
    resta = (int *) malloc(sizeof(int) * SIZE);

    // Rellenar vectores utilizados:
    random_array(vector1, SIZE);
    fill_array(vector2, SIZE);

    // Mostrar ambos vectores:
    display_array("vector1", vector1);
    display_array("vector2", vector2);

    // Imprimir mensaje de inicio:
    printf("Iniciando...\n");

    // Método FOR para ejecutar la parte en paralelo:
```

```
        for (int j = 0; j < N; j++) {
            start_timer();
            calculate_vectors(resta, suma, vector1, vector2,
SIZE);
            ms += stop_timer();
        }

        // Imprimir/Mostrar en la terminal los vectores
resultantes:
        display_array("suma", suma);
        display_array("resta", resta);

        // Imprimir/Mostrar en terminal el tiempo de ejecución
del programa:
        printf("Tiempo de ejecución = %.5lf ms\n", (ms / N));

        // Liberar la memoria utilizada por las variables:
        free(vector1);
        free(vector2);
        free(suma);
        free(resta);

        // Fin del programa:
        return 0;
    }
```

Implementación en TBB

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: TBB.  
  
*  
  
*-----  
-----*/  
// -----  
-----//  
//  
  
LIBRERÍAS  
  
// -----  
-----//  
  
#include <iostream>  
#include <tbb/task_scheduler_init.h>  
#include <tbb/parallel_for.h>  
#include <tbb/blocked_range.h>  
#include "utils/cppheader.h"  
  
// -----  
-----//  
//  
  
DEFINIR NAMESPACE  
  
// -----  
-----//  
  
using namespace std;  
using namespace tbb;  
  
// -----  
-----//
```



```

// Inicializar variables:
int *vector1 = new int[SIZE];
int *vector2 = new int[SIZE];
int *suma = new int[SIZE];
int *resta = new int[SIZE];

// Rellenar vectores utilizados:
random_array(vector1, SIZE);
fill_array(vector2, SIZE);

// Mostrar ambos vectores:
display_array("vector1", vector1);
display_array("vector2", vector2);

// Imprimir mensaje de inicio:
cout << "Iniciando..." << endl;

// Método FOR para ejecutar la parte en paralelo:
for (int i = 0; i < N; i++) {
    t.start();
    parallel_for(blocked_range<int>(0, SIZE, GRAIN),
        calculate_vectors(resta, suma, vector1,
vector2));
    ms += t.stop();
}

// Imprimir/Mostrar en la terminal los vectores
resultantes:
display_array("suma", suma);
display_array("resta", resta);

// Imprimir/Mostrar en terminal el tiempo de ejecución
del programa:
cout << "Tiempo de ejecución = " << (ms/N) << " ms" <<
endl;

// Liberar la memoria utilizada por las variables:
delete [] vector1;
delete [] vector2;
delete [] suma;
delete [] resta;

// Fin del programa:
return 0;
}

```

Implementación en CUDA

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: CUDA.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
  
LIBRERÍAS  
  
// -----  
-----//  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include "utils/cheader.h"  
  
// -----  
-----//  
// DEFINIR  
VARIABLES QUE NO CAMBIAN  
//  
// -----  
-----//  
  
#define MIN(vector1,vector2)  
(vector1<vector2?vector1:vector2)  
#define SIZE 100000000  
#define THREADS 256  
#define BLOCKS MIN(32, (SIZE + THREADS - 1)/ THREADS)
```

```

// -----
-----//
//
//                                FUNCIÓN
//
// -----
-----//

__global__ void calculate(int *resta, int *suma, int
*vector1, int *vector2) {
    int i = threadIdx.x + (blockIdx.x * blockDim.x);
    if (i < SIZE) {
        suma[i] = vector1[i] + vector2[i];
        resta[i] = vector1[i] - vector2[i];
    }
}

// -----
-----//
//
//                                MAIN
//
// -----
-----//

int main(int argc, char const *argv[]) {
    // Definir variables a utilizar:
    int *vector1, *vector2, *suma, *resta;
    int *device_vector1, *device_vector2, *device_suma,
*device_resta;
    double ms = 0.0;

    // Inicializar variables:
    vector1 = (int *) malloc(sizeof(int) * SIZE);
    vector2 = (int *) malloc(sizeof(int) * SIZE);
    suma = (int *) malloc(sizeof(int) * SIZE);
    resta = (int *) malloc(sizeof(int) * SIZE);

    // Rellenar vectores utilizados:
    random_array(vector1, SIZE);
    fill_array(vector2, SIZE);

    // Mostrar ambos vectores:
    display_array("a", vector1);
    display_array("b", vector2);

```

```

        // Asignar memoria a las variables en el dispositivo
        (GPU):
        cudaMalloc((void**) &device_vector1, SIZE *
sizeof(int));
        cudaMalloc((void**) &device_vector2, SIZE *
sizeof(int));
        cudaMalloc((void**) &device_suma, SIZE * sizeof(int));
        cudaMalloc((void**) &device_resta, SIZE * sizeof(int));

        // Copiar vectores a vectores en el dispositivo (GPU):
        cudaMemcpy(device_vector1, vector1, SIZE * sizeof(int),
cudaMemcpyHostToDevice);
        cudaMemcpy(device_vector2, vector2, SIZE * sizeof(int),
cudaMemcpyHostToDevice);

        // Imprimir mensaje de inicio:
        printf("Iniciando...\n");

        // Método FOR para ejecutar la parte en paralelo en la
GPU:
        for (int j = 0; j < N; j++) {
            start_timer();
            calculate<<<BLOCKS, THREADS>>>(device_resta,
device_suma, device_vector1, device_vector2);
            ms += stop_timer();
        }

        // Copiar vectores resultantes del GPU al CPU:
        cudaMemcpy(suma, device_suma, SIZE * sizeof(int),
cudaMemcpyDeviceToHost);
        cudaMemcpy(resta, device_resta, SIZE * sizeof(int),
cudaMemcpyDeviceToHost);

        // Imprimir/Mostrar en la terminal los vectores
resultantes:
        display_array("c", suma);
        display_array("d", resta);

        // Imprimir/Mostrar en terminal el tiempo de ejecución
del programa:
        printf("Tiempo de ejecución = %.5lf ms\n", (ms / N));

        // Liberar la memoria utilizada en el GPU por las
variables:
        cudaFree(device_vector1);
        cudaFree(device_vector2);

```

```
        cudaFree(device_suma);  
        cudaFree(device_resta);  
  
        // Liberar la memoria del CPU utilizada por las  
variables:  
        free(vector1);  
        free(vector2);  
        free(suma);  
        free(resta);  
  
        // Fin del programa:  
        return 0;  
}
```

Implementación Threads

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: THREADS.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
  
// FUNCIÓN  
  
// -----  
-----//  
  
public class calculate_vectors_JavaThreads extends Thread {  
    private int vector1[], vector2[], suma[], resta[],  
    inicio, fin;  
  
    public calculate_vectors_JavaThreads(int resta[], int  
    suma[], int vector1[], int vector2[], int inicio, int fin) {  
        this.vector1 = vector1;  
        this.vector2 = vector2;  
        this.suma = suma;  
        this.resta = resta;  
        this.inicio = inicio;  
        this.fin = fin;  
    }  
  
    public void run() {  
        for (int i = inicio; i < fin; i++) {  
            suma[i] = vector1[i] + vector2[i];  
            resta[i] = vector1[i] - vector2[i];  
        }  
    }  
}
```

```

    }
}

// -----
-----//
//
//                                MAIN
//
// -----
-----//

public static void main(String args[]) {
    // Definir variables a utilizar:
    final int SIZE = 100_000_000;
    calculate_vectors_JavaThreads threads[];
    int block;
    long inicioTiempo, finTiempo;
    double acum = 0.0;

    // Inicializar variables:
    int vector1[] = new int[SIZE];
    int vector2[] = new int[SIZE];
    int suma[] = new int[SIZE];
    int resta[] = new int[SIZE];

    // Rellenar vectores utilizados:
    Utils.randomArray(vector1);
    Utils.fillArray(vector2);

    // Mostrar ambos vectores:
    Utils.displayArray("vector1", vector1);
    Utils.displayArray("vector2", vector2);

    // Asignación del tamaño del bloque, según el
    número de threads disponibles en el dispositivo:
    block = SIZE / Utils.MAXTHREADS;
    threads = new
calculate_vectors_JavaThreads[Utils.MAXTHREADS];

    // Método FOR para ejecutar la parte en paralelo:
    for (int j = 1; j <= Utils.N; j++) {
        for (int m = 0; m < threads.length; m++) {
            if (m != threads.length - 1) {
                threads[m] = new
calculate_vectors_JavaThreads(resta, suma, vector1, vector2,
(m * block), ((m + 1) * block));
            } else {

```

```

        threads[m] = new
calculate_vectors_JavaThreads(resta, suma, vector1, vector2,
(m * block), Utils.N);
    }
}

    inicioTiempo = System.currentTimeMillis();
    for (int k = 0; k < threads.length; k++) {
        threads[k].start();
    }
    for (int l = 0; l < threads.length; l++) {
        try {
            threads[l].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    finTiempo = System.currentTimeMillis();
    acum += (finTiempo - inicioTiempo);
}

    // Imprimir/Mostrar en la terminal los vectores
resultantes:
    Utils.displayArray("suma", suma);
    Utils.displayArray("resta", resta);

    // Imprimir/Mostrar en terminal el tiempo de
ejecución del programa :
    System.out.printf("Tiempo de ejecución = %.5f ms
\n", (acum / Utils.N));
}
}

```


Implementación Fork/Join

```
/*-----  
-----  
  
*  
  
* Multiprocesadores: Proyecto final.  
  
* Fecha: 01 de diciembre del 2019.  
  
* Autor: Gerardo Daniel Naranjo Gallegos, A01209499.  
  
* Implementación: FORK/JOIN.  
  
*  
  
*-----  
-----*/  
  
// -----  
-----//  
//  
  
LIBRERÍAS  
  
// -----  
-----//  
  
import java.util.concurrent.RecursiveAction;  
import java.util.concurrent.ForkJoinPool;  
  
// -----  
-----//  
//  
  
FUNCIÓN  
  
// -----  
-----//  
  
public class calculate_vectors_FORKJOIN extends  
RecursiveAction {  
    // Variables que no cambian:  
    private static final int SIZE = 100_000_000;  
    private static final int MIN = 100_000;
```

```

        private int vector1[], vector2[], suma[], resta[],
inicio, fin;

        public calculate_vectors_FORKJOIN(int inicio, int
fin,int resta[], int suma[], int vector1[], int vector2[]) {
            this.inicio = inicio;
            this.fin = fin;
            this.resta = resta;
            this.suma = suma;
            this.vector1 = vector1;
            this.vector2 = vector2;
        }

        protected void computeDirectly() {
            for (int i = inicio; i < fin; i++) {
                suma[i] = vector1[i] + vector2[i];
                resta[i] = vector1[i] - vector2[i];
            }
        }

        @Override
        protected void compute() {
            if ( (fin - inicio) <= MIN ) {
                computeDirectly();
            } else {
                int mitad = inicio + ( (fin - inicio) / 2 );
                invokeAll(
                    new calculate_vectors_FORKJOIN(inicio,
mitad, resta, suma, vector1, vector2),
                    new calculate_vectors_FORKJOIN(mitad,
fin, resta, suma, vector1, vector2)
                );
            }
        }

// -----
-----//
//
//
//
//
// -----
-----//

        public static void main(String args[]) {
            // Definir variables a utilizar:
            ForkJoinPool pool;
            long inicioTiempo, finTiempo;

```

```

double acum = 0.0;

// Inicializar variables:
int vector1[] = new int[SIZE];
int vector2[] = new int[SIZE];
int suma[] = new int[SIZE];
int resta[] = new int[SIZE];

// Rellenar vectores utilizados:
Utils.randomArray(vector1);
Utils.fillArray(vector2);

// Mostrar ambos vectores:
Utils.displayArray("vector1", vector1);
Utils.displayArray("vector2", vector2);

// Método FOR para ejecutar la parte en paralelo:
for (int j = 0; j < Utils.N; j++) {
    inicioTiempo = System.currentTimeMillis();
    pool = new ForkJoinPool(Utils.MAXTHREADS);
    pool.invoke(new calculate_vectors_FORKJOIN(0,
Utils.N, resta, suma, vector1, vector2));
    finTiempo = System.currentTimeMillis();
    acum += (finTiempo - inicioTiempo);
}

// Imprimir/Mostrar en la terminal los vectores
resultantes:
Utils.displayArray("suma", suma);
Utils.displayArray("resta", resta);

// Imprimir/Mostrar en terminal el tiempo de
ejecución del programa :
System.out.printf("Tiempo de ejecución = %.5f ms
\n", (acum / Utils.N));
}
}

```