**INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY**

**CAMPUS QUERETARO**

# Paralelismo Serie de Nilakantha

## Artículo de Investigación

Franco Garcia Pedregal- A01273527

**TE-3061.1 MULTIPROCESADORES**

**PROFESOR: PEDRO OSCAR PÉREZ M.**

**Agosto - Diciembre 2020**

# Índice

# Capítulo 1 introducción

## 1.1 Resumen

Este documento tiene como problema el aproximado de PI mediante la Serie de Nilakantha el cual se puede paralelizar utilizando al menos cuatro de las cinco herramientas que fueron vistas durante el semestre; las cinco herramientas que fueron cubiertas fueron las siguientes:

• Threads en Java.

• Fork/Join framework en Java.

• OpenMP.

• Intel Threading Building Blocks.

• CUDA.

De estas cinco herramientas las que se decidieron implementar para este trabajo fueron: *Threads en Java, OpenMP,Intel Threading Building Blocks y CUDA.* El problema también será resuelto de manera secuencial para los *lenguajes* utilizados, se medirá el *Speed-Up* alcanzado con cada una de las tecnologías y posteriormente se realizará un análisis comparativo sobre todas estas.

El algoritmo analizado fue de una serie de Aproximación a PI, en específico la serie creada por Keḷallur Nilakantha Somayaji las aproximaciones a PI son sumamente importante ya que este número siempre está presente en nuestra vida diario a donde sea que vayamos:

- Números complejos,
- Secuencias recursivas,
- Secuencias logísticas,
- Series,
- Integrales...

## 1.2 Introduccion

El Ambiente con el cual se realizó este proyecto fue el siguiente:

| | |
|---|---|
| **OS:** | Windows 10 Pro 20H2 64 bits, procesador x64 |
| **CPU:** | Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz |
| **RAM:** | 16.0 GB |
| **GPU** | NVIDIA GeForce RTX 2060; 6.0GB |

Durante este articulo el problema que será resuelto de manera secuencial y en paralelo será el de la aproximación a PI con un numero de 100,000,000 iteraciones. Cuando hablamos de aproximación al número buscamos una convergencia después n términos *Figura 0.1*

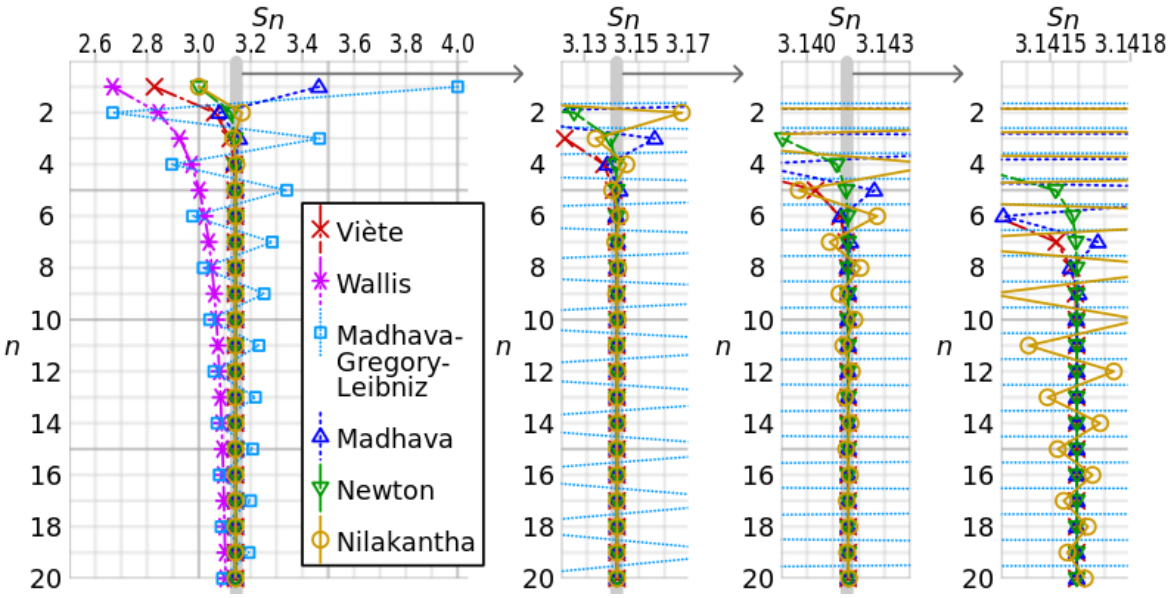


*Figura 0.1 Comparación de la convergencia de varias series infinitas históricas*

Buscamos una convergencia con el menor número de intentos esta serie es considerablemente mas complicada que la de Leibniz sin embargo encuentra una convergencia mucho más rápida que la ya antes mencionado.

Gran parte de la investigación y programación de esta serie es basada en la serie de Leibniz debido a las similitudes en cuanto a tipo de problema a resolver.

$$\pi = 3 + 4\sum_{k=0}^{\infty} \frac{-1^k}{(2k+2)(2k+3)(2k+4)}$$

$$= 3 + 4\left(\frac{1}{2\cdot 3\cdot 4} - \frac{1}{4\cdot 5\cdot 6} + \frac{1}{6\cdot 7\cdot 8} - \frac{1}{8\cdot 9\cdot 10}\ldots\right)$$

*Ecuación 1 Serie de Nilakantha*

$$\pi = 4\left(\sum_{k=1}^{\infty} \frac{(-1)^{(k+1)}}{(2k-1))}\right)$$

$$= 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$$

*Ecuación 2 Serie de Leibniz*

Como podemos observar hay una amplia similitud entre ambas ecuaciones

# Capítulo 2 Desarrollo

## 2.1 Sequencial

Tanto la implementación secuencial en C como la implementación secuencial en Java no presentaron mayor problema; a continuación, se muestra una comparación los parámetros que se obtuvieron en las dos tecnologías; se tomó un N de 10 el cual nos permite ejecutar N veces nuestra solución para así obtener un cálculo más exacto de la ejecución del mismo esta constante proviene de la Librería "Utils" provista por el instructor.

| Java | C |
|---|---|
| 3412.3 ms | 2124.21 ms |

Cabe destacar que el lenguaje Java al ser un lenguaje orientado a objetos tiene un menor desempeño con el uso de cálculos precisos.

## 2.2 Paralelo

Los resultados obtenidos al ocupar las técnicas de paralelización vistas en el curso fueron sumamente satisfactorios

| Threads | OpenMp | Intel TBB | Cuda |
|---|---|---|---|
| 509.5 ms | 379.19 ms | 97.8864 ms | 0.0051 ms |

### 2.2.1 Threads

Trabajar con java, un lenguaje de programación orientada a objetos, el cual además se ejecuta sobre una máquina virtual permite tener un código portable que es posible ser ejecutado de manera muy similar en diferentes plataformas sin la necesidad de portar el código con ajustes diferentes a arquitecturas y SO

Para escalar la versión básica del algoritmo fue considerablemente complicado más la asistencia en internet y el acceso a las implementaciones de Leibniz logro facilitar el trabajo. El uso de threads se logra tener una gran mejora de velocidad de casi 7 veces más rápida contra la implementación secuencial.

### 2.2.2 OpenMP

OpenMP se basa en el modelo fork-join, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en K hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join).

Cuando se incluye una directiva de compilador OpenMP esto implica que se incluye una sincronización obligatoria en todo el bloque. Es decir, el bloque de código se marcará como paralelo y se lanzarán hilos según las características que nos dé la directiva, y al final de ella habrá una barrera para la sincronización de los diferentes hilos (salvo que implícitamente se indique lo contrario con la directiva nowait). Este tipo de ejecución se denomina fork-join.

Para el algoritmo el pragma de paralelismo se ejecuto dentro de nuestra función para calcular PI con una variable N compartida entre cada uno de los hilos y al final cada uno de los hilos fue unido en la variable resultado.

### 2.2.3 Intel TBB

Intel TBB implenta *task stealing* (robo de tareas) para balancear la carga de trabajo sobre los núcleos de procesamiento disponibles con el fin de incrementar el aprovechamiento de los núcleos y la escalabilidad de los programas. Inicialmente la carga de trabajo se divide uniformemente entre los núcleos de procesamiento disponibles. Si alguno de ellos termina su trabajo mientras otro todavía tiene una carga significativa en su cola de tareas, el gestor de tareas reasigna parte de este

trabajo al núcleo inactivo. Esta capacidad de reasignación dinámica desacopla la programación de la máquina, permitiendo que las aplicaciones escritas usando esta biblioteca se escalen para usar todos los núcleos de procesamiento disponibles si ningún cambio en el código fuente o los ejecutables.

Para el algoritmo se fueron creando pequeñas tareas del mismo tipo para que fueran agregándose a la cola de tareas y cada una fuera “robada” para cargarse en los núcleos de este. Y al finalizar ejecución fueran acumulándose en la variable Pi.

### 2.2.4 CUDA

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPU en aplicaciones con un gran coste aritmético frente a realizar numerosos accesos a memoria principal, lo que podría actuar de cuello de botella.

El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo para poder incrementar el número de núcleos computacionales. Este diseño contiene tres puntos claves, que son la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización.

La estructura que se utiliza en este modelo está definida por un grid, dentro del cual hay bloques de hilos que están formados por como máximo 512 hilos distintos.

Cada hilo está identificado con un identificador único, que se accede con la variable threadIdx. Esta variable es muy útil para repartir el trabajo entre distintos hilos. threadIdx tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos. Así, cada elemento de una matriz, por ejemplo, lo podría tratar su homólogo en un bloque de hilos de dos dimensiones.

Al igual que los hilos, los bloques se identifican mediante blockIdx (en este caso con dos componentes x e y). Otro parámetro útil es blockDim, para acceder al tamaño de bloque.

El algoritmo es un tanto complicado de explicar pero imaginemos como una cascada donde en la parte mas alta pones la tarea que va a realizar cada core y cuando caen son unidos cada una de las tareas en este caso lo que tenemos es un tremendo poder de procesamiento cuando en otras tecnologías gracias a las especificaciones de la computadora teníamos 12 cores aquí tenemos cerca 256 entonces esto reduce mucho el tiempo de ejecución la sintaxis es un tanto compleja pero cuando comienzas a investigar un poco mas todo toma forma.

# Capítulo 3 Conclusiones

## 3.1 Speed-up

Realizando un cálculo de las mejoras obtenidas por tecnología utilizando la fórmula de Speed Up para el cálculo de rendimientos

$$\text{Speedup}(n_t) = \frac{Time_{best\_sequential\_algorithm}}{Time_{parallel\_implementation}(n_t)}$$

Se obtuvieron los siguientes resultados:

| Threads | OpenMp | Intel TBB | Cuda |
|---|---|---|---|
| 6.6973503 | 5.6019787 | 21.700811 | 416512.61 |

Como podemos observar el gran poder computacional de Cuda es muchísimo mas grande que cualquiera de las otras tecnologías sin embargo al no ser de fácil acceso para muchos factores en la industria se ha buscado el uso de otras tecnologías para este caso en particular fue cerca de medio millón de veces más rápido que la implementación de manera secuencial sin embargo una tecnología como Intel TBB que aun hace uso completo de nuestro CPU obtuvimos un rendimiento 21 veces más rápido depende del propósito y prepuesto con el que se cuente en el momento.

# Capítulo 4 Apéndice

## 4.1 Librería Utils

### 4.1.1 Utils (Java)

```
// ================================================================
//
// File  : Utils.java
// Author: Pedro Perez
// Description: This file contains the implementation of the functions
//              for initializing integer arrays.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

import java.util.Random;
```

```java
public class Utils {
    private static final int DISPLAY = 100;
    private static final int TOP_VALUE = 10_000;
    public static final Random r = new Random();

    public static final int MAXTHREADS =
Runtime.getRuntime().availableProcessors();
    public static final int N = 10;

    public static void randomArray(int array[]) {
        for (int i = 0; i < array.length; i++) {
            array[i] = r.nextInt(TOP_VALUE) + 1;
        }
    }

    public static void fillArray(int array[]) {
        for (int i = 0; i < array.length; i++) {
            array[i] = (i % TOP_VALUE) + 1;
        }
    }

    public static void displayArray(String text, int array[]) {
        int limit = (int) Math.min(DISPLAY, array.length);

        System.out.printf("%s = [%4d", text, array[0]);
        for (int i = 1; i < limit; i++) {
            System.out.printf(",%4d", array[i]);
        }
        System.out.printf(", ..., ]\n");
    }
}
```

## 4.1.2 Utils (C)

```c
// ================================================================
//
// File: utils.h
// Author: Pedro Perez
// Description: This file contains the implementation of the
//              functions used to take the time and perform the
//              speed up calculation; as well as functions for
//              initializing integer arrays.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================

#ifndef UTILS_H
#define UTILS_H

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

```c
#include <sys/types.h>

#define MIN_VAL(a,b) (((a)<(b))?(a):(b))
#define MAX_VAL(a,b) (((a)>(b))?(a):(b))

#define N           10
#define DISPLAY     16
#define TOP_VALUE   10000

typedef enum color {BLUE, GREEN, RED} Color;

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;

    srand(time(0));
    for (i = 0; i < size; i++) {
        array[i] = (i % TOP_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
```

### 4.1.3 Utils (Cuda)

```
%%cuda --name header.h
#ifndef HEADER_H
#define HEADER_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(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 /* HEADER_H */
```

## 4.2 Implementaciones Secuenciales

### 4.2.1 C Secuencial

```
/*------------------------------------------------------------------
*
* Multiprocesadores: C
* Fecha: 27-Nov-2020
* Autor: A01273527 Franco Garcia Pedregal
*
*
*-----------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include <math.h>
//#include <omp.h>
#define size 100000000 //1e8
float Pi(){
    float resultado, n;
    resultado;

    //#pragma omp parallel for shared(n) reduction(+:resultado)
    for(int j=0; j<size ; j++){
        float aux;
        aux=pow(-1.0,j)/((2.0*j+2.0)*(2.0*j+3.0)*(2.0*j+4.0));
        resultado+=aux;

    }
    resultado=(resultado*4.0)+3.0;

    return resultado;
}

int main(int argc, char* argv[]) {
    int i;
    double ms, result;

    printf("Starting...\n");
```

```
    ms = 0;
    for (i = 0; i < N; i++) {
        start_timer();

        result = Pi();

        ms += stop_timer();
    }
    printf("sum = %lf\n", result);
    printf("avg time = %.5lf ms\n", (ms / N));

    return 0;
}
```

## 4.2.2 Java Secuencial

```
// ================================================================
//
// File: Lab4_S.java
// Author: Pedro Perez
// Description: This file contains the code to perform the numerical
//              integration of a function within a defined interval.
//              The time this implementation takes will be used as
//              the basis to calculate the improvement obtained with
//              parallel technologies.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ================================================================
//*
//* Multiprocesadores: C
//* Fecha: 27-Nov-2020
//* Autor: A01273527 Franco García Pedregal
//
import java.lang.*;
public class JavaSeq{

    private static final int RECTS = 100000000; //1e8
    private double resultado;

    public JavaSeq(){

    }

    public double getResult() {
        return resultado;
    }

    public void calculate() {
        resultado = 0;
        for (int i = 0; i < RECTS; i++) {
            double aux;
            aux=(double)Math.pow(-
1,i)/((double)(2*i+2)*(double)(2*i+3)*(double)(2*i+4));
```

```
            resultado+=aux;
        }
        resultado=(4*resultado)+3;
    }

    public static void main(String args[]) {
        long startTime, stopTime;
        double ms;

        System.out.printf("Starting...\n");
        ms = 0;
        JavaSeq e = new JavaSeq();
        for (int i = 0; i < Utils.N; i++) {
            startTime = System.currentTimeMillis();

            e.calculate();

            stopTime = System.currentTimeMillis();

            ms += (stopTime - startTime);
        }
        System.out.printf("result = %.5f\n", e.getResult());
        System.out.printf("avg time = %.5f\n", (ms / Utils.N));
    }
}
```

## 4.3 IMPLEMENTACIONES Paralelas

### 4.3.1 Java Threads

```
// ============================================================================
//
// File: Example4.java
// Author: Pedro Perez
// Description: This file implements the multiplication of a matrix
//              by a vector. The time this implementation takes will
//              be used as the basis to calculate the improvement
//              obtained with parallel technologies.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ============================================================================
//*
//* Multiprocesadores: C
//* Fecha: 27-Nov-2020
//* Autor: A01273527 Franco García Pedregal
//
import java.lang.*;

public class JTh extends Thread{
    private static final int RECTS = 1_000_000_00;
    private double resultado;
    private int start, end;
```

```
    public JTh(int start, int end) {
        this.resultado = 0;
        this.start = start;
        this.end = end;
    }

    public double getResult() {
        return resultado;
    }

    public void run() {
        resultado = 0;
        for (int i = start; i < end; i++) {
            double aux;
            aux=(double)Math.pow(-
1,i)/((double)(2*i+2)*(double)(2*i+3)*(double)(2*i+4));
            resultado+=aux;
        }
    }

    public static void main(String args[]) {
        long startTime, stopTime;
        int block;
        JTh threads[];
        double ms, result = 0;


        block = RECTS / Utils.MAXTHREADS;
        threads = new JTh[Utils.MAXTHREADS];

        System.out.printf("Starting with %d threads...\n",
Utils.MAXTHREADS);
        ms = 0;
        for (int j = 1; j <= Utils.N; j++) {
            for (int i = 0; i < threads.length; i++) {
                if (i != threads.length - 1) {
                    threads[i] = new JTh( (i * block), ((i + 1) *
block));
                } else {
                    threads[i] = new JTh((i * block), RECTS);
                }
            }

            startTime = System.currentTimeMillis();
            for (int i = 0; i < threads.length; i++) {
                threads[i].start();
            }
            for (int i = 0; i < threads.length; i++) {
                try {
                    threads[i].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            stopTime = System.currentTimeMillis();
            ms +=  (stopTime - startTime);
```

```
            if (j == Utils.N) {
                result = 0;
                for (int i = 0; i < threads.length; i++) {
                    result += threads[i].getResult();
                }
            }
        }
        result=(result*4)+3;
        System.out.printf("result = %.5f\n", result);
        System.out.printf("avg time = %.5f\n", (ms / Utils.N));
    }
}
```

### 4.3.2 OpenMP

```
/*------------------------------------------------------------------
*
* Multiprocesadores: C
* Fecha: 27-Nov-2020
* Autor: A01273527 Franco Garcia Pedregal
*
*
*-----------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include <math.h>
#include <omp.h>
#define size 100000000 //1e8
float Pi(){
    float resultado, n;
    resultado;

    #pragma omp parallel for shared(n) reduction(+:resultado)
    for(int j=0; j<size ; j++){
        float aux;
        aux=pow(-1.0,j)/((2.0*j+2.0)*(2.0*j+3.0)*(2.0*j+4.0));
        resultado+=aux;

    }
    resultado=(resultado*4.0)+3.0;

    return resultado;
}

int main(int argc, char* argv[]) {
    int i;
    double ms, result;

    printf("Starting...\n");
    ms = 0;
    for (i = 0; i < N; i++) {
        start_timer();
```

```
        result = Pi();

        ms += stop_timer();
    }
    printf("sum = %lf\n", result);
    printf("avg time = %.5lf ms\n", (ms / N));

    return 0;
}
```

### 4.3.3 Intel TBB

```
// ========================================================================
//
// File: example2.cpp
// Author: Pedro Perez
// Description: This file contains the code to perform the numerical
//              integration of a function within a defined interval
//              using Intel's TBB.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ========================================================================
/*-----------------------------------------------------------------

* Multiprocesadores: TBB

* Fecha: 27-Nov-2020

* Autor: A01273527 Franco Garcia Pedregal
*-----------------------------------------------------------------*/
#include <iostream>
#include <iomanip>
#include <cmath>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>

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

using namespace std;
using namespace tbb;

const int RECTS = 100000000; //1e8

class Integration {
private:
    float pi;

public:
    Integration() : pi(0){}
```

```
    Integration(Integration &obj, split) : pi(0) {}
    float getResult() const {
        return pi;
    }

    //  void operator() (const blocked_range<int> &r) const {
    void operator() (const blocked_range<int> &r) {
        for (int i = r.begin(); i != r.end(); i++) {
            float aux = (i % 2 == 0)? 1 : -1;
            pi = pi + (aux*4.0) /
(((2.0*i)+2.0)*((2.0*i)+3.0)*((2.0*i)+4.0));
        }
    }

    void join(const Integration &x) {
        pi += x.pi;
    }
};

int main(int argc, char* argv[]) {
    double ms;
    float pi = 0;


    cout << "Starting..." << endl;
    ms = 0;
    for (int i = 0; i < N; i++) {
        start_timer();

        Integration obj;
        parallel_reduce(blocked_range<int>(0, RECTS), obj);
        pi = obj.getResult();

        ms += stop_timer();
    }
    cout << "result = " << setprecision(15) << (pi+3.0) << endl;
    cout << "avg time = " << setprecision(15) << (ms / N) << " ms" <<
endl;

    return 0;
}
```

## 4.3.4 CUDA

```
%%cu
 #include <stdio.h>
 #include <stdlib.h>
 #include "/content/src/header.h"
 #define MIN(a,b) (a<b?a:b)
 #define SIZE   1e8
 #define THREADS    256
 #define BLOCKS MIN(32, (SIZE + THREADS - 1)/ THREADS)

 __global__ void sum(double *result) {
    __shared__ double cache[THREADS];
```

```
    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int cacheIndex = threadIdx.x;

    double acum = 0;
    while (tid < SIZE) {
        acum += pow(-1,tid)/((2.0*tid+2.0)*(2.0*tid+3.0)*(2.0*tid+4.0));
        tid += blockDim.x * gridDim.x;
    }

    cache[cacheIndex] = acum;

    __syncthreads();

    int i = blockDim.x / 2;
    while (i > 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0) {
        result[blockIdx.x] = cache[cacheIndex];
    }
}

int main(int argc, char* argv[]) {
    int i, *array, *d_a;
    double *results, *d_r;
    double ms;

    results = (double*) malloc( BLOCKS * sizeof(double) );
    cudaMalloc( (void**) &d_r, BLOCKS * sizeof(double) );

    printf("Starting...\n");
    ms = 0;
    for (i = 1; i <= N; i++) {
        start_timer();
        sum<<<BLOCKS, THREADS>>> (d_r);
        ms += stop_timer();
    }

    cudaMemcpy(results, d_r, BLOCKS * sizeof(double),
cudaMemcpyDeviceToHost);

    double acum = 0;
    for (i = 0; i < BLOCKS; i++) {
        acum += results[i];
    }

    printf("sum = %lf\n", ((acum*4.0)+3.0));
    printf("avg time = %.5lf\n", (ms / N));

    cudaFree(d_r);
    free(results);
    return 0;
```

# Capítulo 5 Agradecimiento

## 5.1 Agradecimiento

Me gustaría agradecer a nuestro profesor Pedro Oscar Pérez Murueta una excelente persona fue un placer poder asistir a este curso cambia mucho los paradigmas que uno tiene como programador y más durante todas las tecnologías que están saliendo al mercado actualmente me parece una clase perfecta y actual de algo que nos enfrentaremos como ingenieros recién graduados me voy con el gusto de querer explorar mas acerca de estos temas para hacer de su uso conjunto con tecnologías como la inteligencia artificial y Machine Learning. Igual a mis compañeros que son fundamentales para el aprendizaje de esta materia el apoyarnos mutuamente en dudas el trabajar en equipo con mi Colaborador Dagoberto Prado Ayala y ver como logramos resolver cada uno de los objetivos y trabajos de esta materia me llena de satisfacción.

# Capítulo 6 Referencias

## 6.1 Referencias


→, V. (2020). Approximate pi using several different methods in C#. Retrieved 27 November 2020, from http://csharphelper.com/blog/2015/03/approximate-pi-using-several-different-methods-in-c/

CUDA. (2020). Retrieved 27 November 2020, from https://es.wikipedia.org/wiki/CUDA

Intel Threading Building Blocks. (2020). Retrieved 27 November 2020, from https://es.wikipedia.org/wiki/Intel_Threading_Building_Blocks

Nilakantha Somayaji. (2020). Retrieved 27 November 2020, from https://en.wikipedia.org/wiki/Nilakantha_Somayaji

OpenMP. (2020). Retrieved 27 November 2020, from https://es.wikipedia.org/wiki/OpenMP#:~:text=OpenMP%20es%20una%20interfaz%20de,modelo%20de%20ejecuci%C3%B3n%20fork%2Djoin.

Repunta precio del piloncillo – Zafranet. (2020). Retrieved 27 November 2020, from https://www.zafranet.com/2016/12/repunta-precio-del-piloncillo/

Sanders, J., & Kandrot, E. (2010). *CUDA C by example*. Boston, Mass.: Addison-Wesley.

Superprof, V. (2020). Definición y Usos del Número Pi | Superprof. Retrieved 27 November 2020, from https://www.superprof.mx/blog/usos-constante-arquimedes/