



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Tarjetas Gráficas y Aceleradores

## OpenACC

### Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



# Introducción

- OpenACC (for Open Accelerators) es un estándar de programación paralela.
- Desarrollado por Cray, CAPS, Nvidia y PGI, está diseñado para simplificar la programación paralela de sistemas heterogéneos.
  
- Última versión: 2.7 (Nov/2018)
- Versión gratuita: PGI Community Edition ([disponible en www.pgroup.com](http://www.pgroup.com))
  
- Simplificando “OpenMP para GPUs”



# Introducción

## 3 Posibles enfoques para la Programación Heterogénea

Aplicación

Librerías

Directivas de  
Compilación

Lenguajes de  
Programación

Fácil de usar  
Más Rendimiento

Fácil de usar  
Código portable

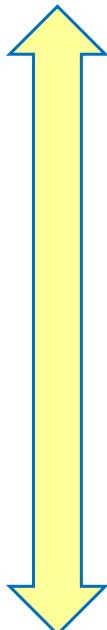
Más Rendimiento  
Más Flexible



# Introducción: Simplicidad & Rendimiento

## Simplicidad

### Librerías (optimizadas)



- Cambios mínimos (o ninguno) para librerías estándar. Alto Rendimiento
- Limitado a las librerías disponibles

### Directivas de Compilación

- Basado en los Lenguajes de programación existentes, es simple y familiar.
- El rendimiento no será óptimo porque las directivas no siempre pueden aprovechar los detalles de la arquitectura.

### Lenguajes de Programación Paralelos

- Exponen los detalles de bajo nivel para obtener el máximo rendimiento
- En general, más difícil de aprender y más costoso de desarrollar.

## Rendimiento



# Introducción

## 3 Posibles enfoques para la Programación Heterogénea

Aplicación

Librerías

OpenACC

CUDA  
OpenCL

Fácil de usar  
Más Rendimiento

Fácil de usar  
Código portable

Más Rendimiento  
Más Flexible



# OpenACC, estándar en directivas para GPU

## SIMPLE

- Las directivas son un camino sencillo para acelerar aplicaciones de cálculo intensivo

## ABIERTO

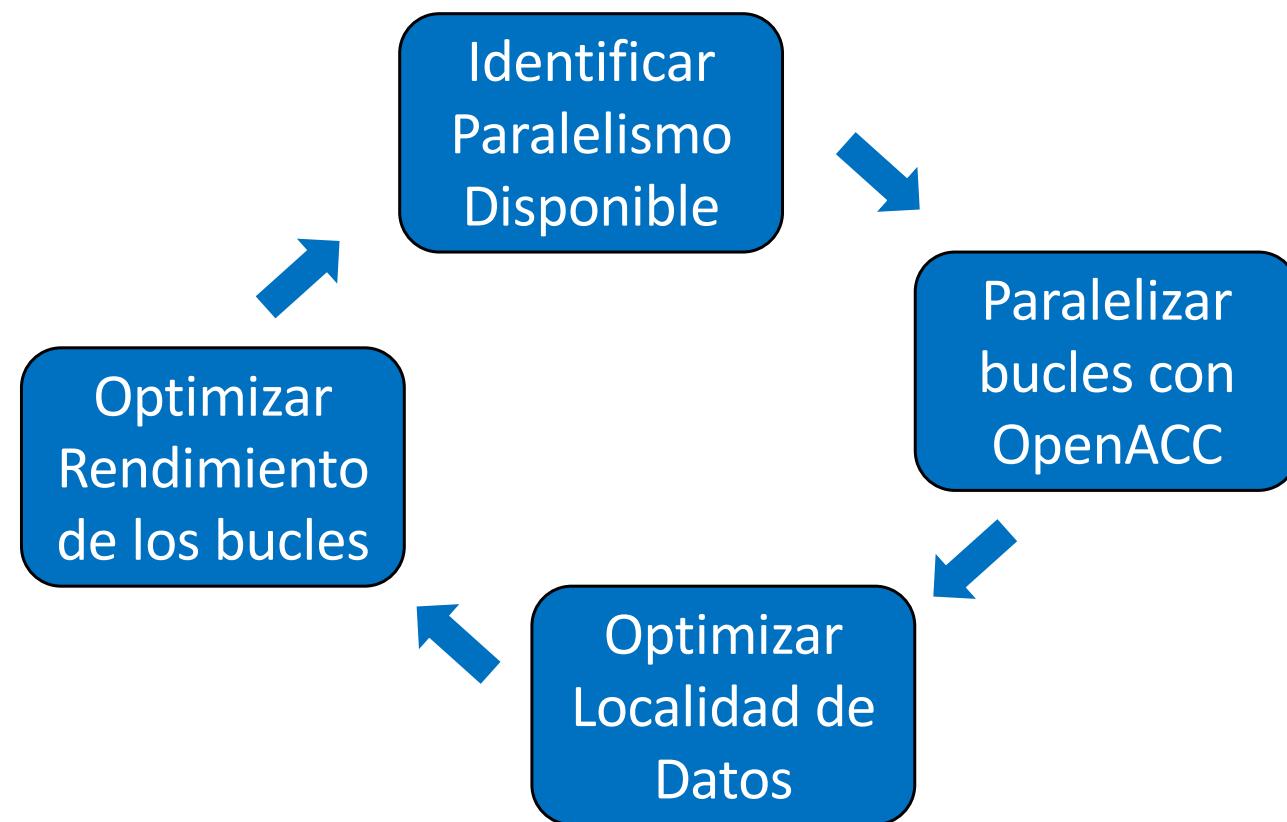
- OpenACC es un conjunto estándar de directivas para GPU, que hacen que la programación de GPUs sea sencilla y portable a un sistema paralelo o a un multi-core

## PORTABLE

- Las directivas para la GPU expresan paralelismo a alto nivel, permitiendo la portabilidad a un extenso rango de arquitecturas con el mismo código

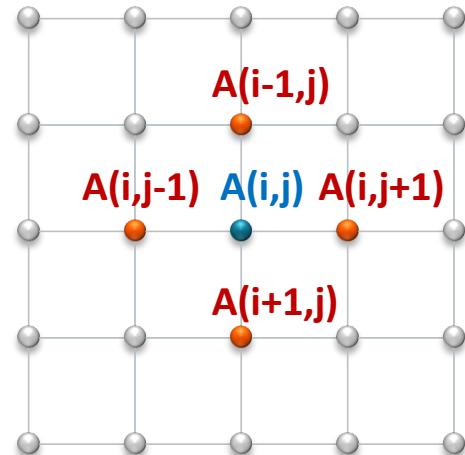


# Ciclo de diseño de una aplicación paralela con OpenACC



# Ejemplo: Iteración de Jacobi

- Algoritmo iterativo que converge al valor correcto, calculando el nuevo valor de un punto, a partir del valor de los puntos vecinos
- Resolución de la ecuación de Laplace en 2D:  $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$

□ Algoritmo iterativo que converge al valor correcto, calculando el nuevo valor de un punto a partir del valor de los puntos vecinos  
○ Resolución de la ecuación de Laplace en 2D:  $\nabla^2 f(x, y) = 0$

# Iteración de Jacobi en C

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++) {  
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                               A[i][j-1]+ A[i][j+1]);  
  
            err = max(err, abs(AA[i][j] - A[i][j]));  
        }  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++)  
            A[i][j] = AA[i][j];  
  
    iter++;  
}
```

Iteración general

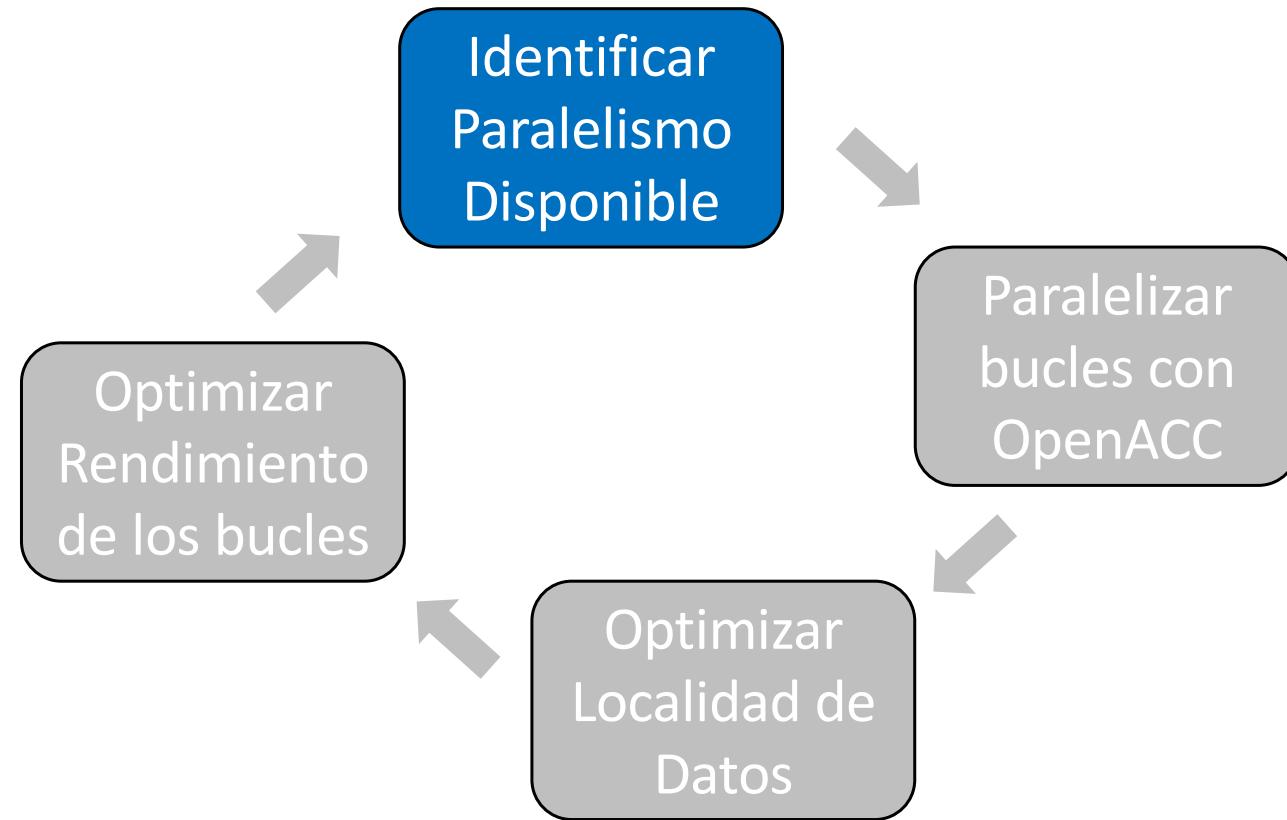
Cálculo nuevo valor de A

Cálculo del error máximo

Actualizar el array



# Ciclo de diseño de una aplicación paralela con OpenACC



# Iteración de Jacobi en C

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++) {  
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                               A[i][j-1]+ A[i][j+1]);  
  
            err = max(err, abs(AA[i][j] - A[i][j]));  
        }  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++)  
            A[i][j] = AA[i][j];  
  
    iter++;  
}
```

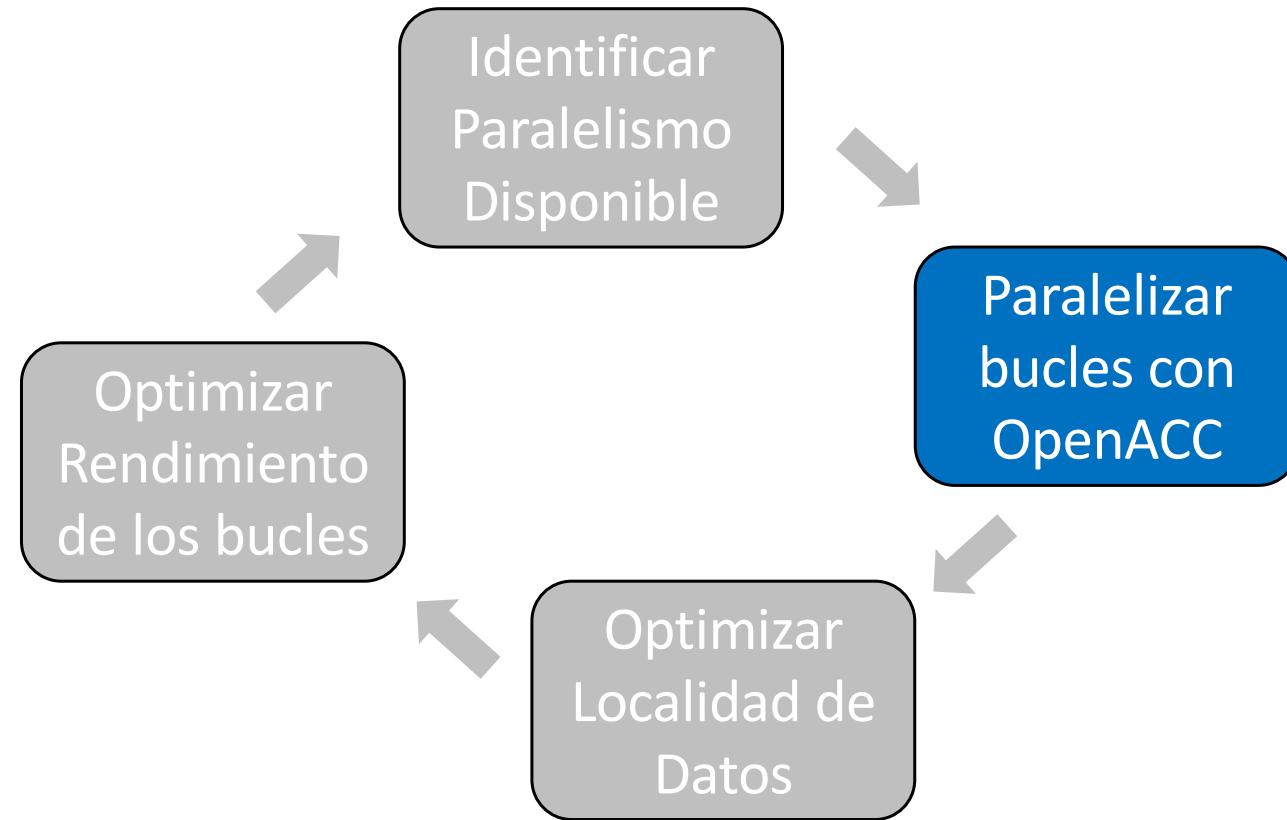
Bucle Secuencial

Bucle paralelo

Bucle Paralelo



# Ciclo de diseño de una aplicación paralela con OpenACC



# OpenACC parallel loop directive

```
for (int i=0; i<N; i++) {  
    y[i] = a*x[i] + y[i];  
}
```

Bucle potencialmente paralelo

```
#pragma acc parallel loop  
for (int i=0; i<N; i++) {  
    y[i] = a*x[i] + y[i];  
}
```

OpenACC generará un kernel para la GPU

**parallel**, el programador identifica un bloque de código con paralelismo. El compilador generará un kernel.

**loop**, el programador identifica un bucle que será paralelizado dentro de un kernel.

En general, **parallel** y **loop** suelen aparecer juntos.



# OpenACC parallel loop reduction directive

```
for (int i=0; i<N; i++) {  
    sum = sum + v[i];  
}
```

Reducción  
clásica

```
#pragma acc parallel loop reduction(sum:+)  
for (int i=0; i<N; i++) {  
    sum = sum + v[i];  
}
```

OpenACC generará un  
kernel para la GPU

**reduction**, se aplica una reducción sobre la variable especificada, con el operador indicado. Posibles operadores: +, \*, max, min, y operadores lógicos.



# OpenACC kernels directive

```
#pragma acc kernels
{
    for (int i=0; i<N; i++) {
        x[i] = 3.4;
        y[i] = 1.0;
    }
    for (int i=0; i<N; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

El compilador identificara 2 bucles paralelos y generará 2 kernels

**kernels**, indica que esa región puede contener paralelismo y el compilador es el encargado de decidir qué se puede parallelizar.



# Iteración de Jacobi con OpenACC

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
    #pragma acc kernels  
    {  
        for (int i=1; i<Nfil-1; i++)  
            for (int j=1; j<Ncol-1; j++) {  
                AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                                  A[i][j-1]+ A[i][j+1]);  
                err = max(err, abs(AA[i][j] - A[i][j]));  
            }  
        for (int i=1; i<Nfil-1; i++)  
            for (int j=1; j<Ncol-1; j++)  
                A[i][j] = AA[i][j];  
    }  
    iter++;  
}
```

El compilador buscará paralelismo en esta región.



# Iteración de Jacobi con OpenACC

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
    #pragma acc parallel loop reduction(err:max)  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++) {  
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                            A[i][j-1]+ A[i][j+1]);  
            err = max(err, abs(AA[i][j] - A[i][j]));  
        }  
    #pragma acc parallel loop  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++)  
            A[i][j] = AA[i][j];  
    iter++;  
}
```

BUCLE PARALELO

BUCLE PARALELO



# OpenACC kernels vs parallel loop

## parallel loop

- Requiere que el programador analice el código para asegurar que hay paralelismo.
- Puede paralelizar cosas que el compilador no encontraría
- Es el camino más simple para converger con OpenMP

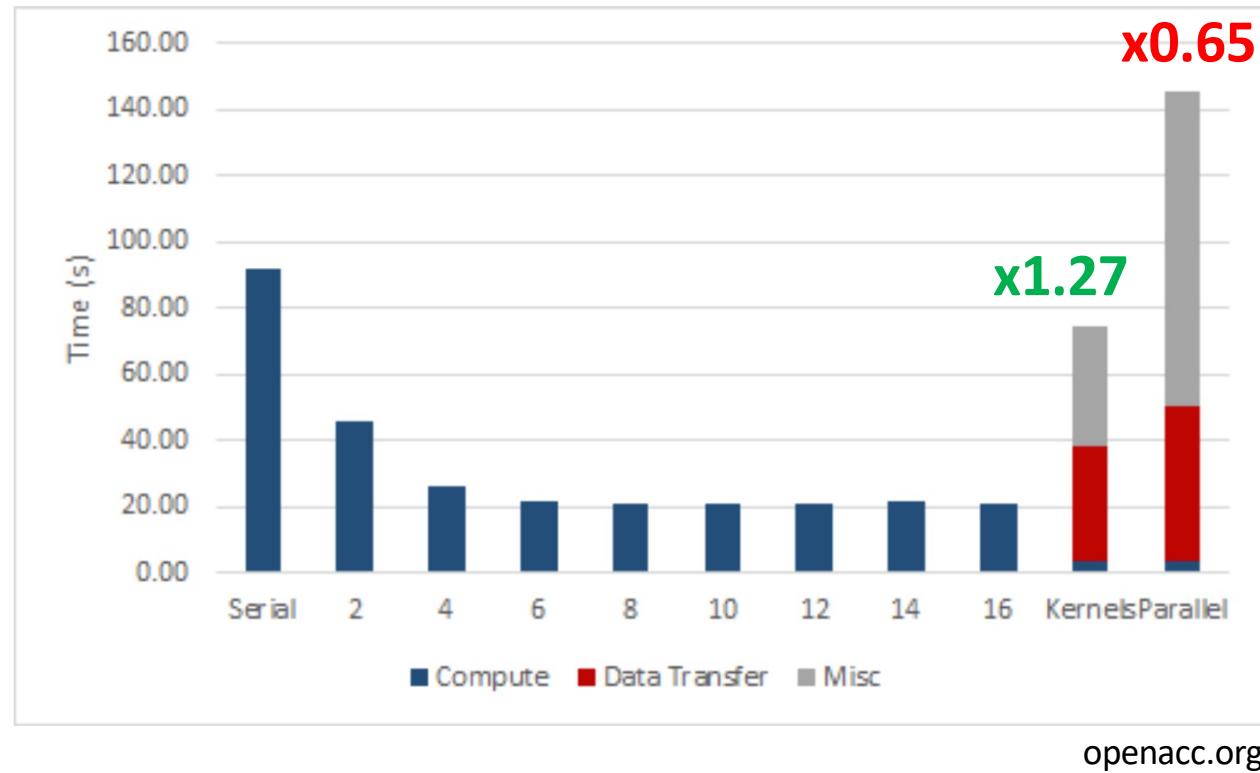
## kernels

- El compilador realiza el análisis y paralleliza lo que crea que es seguro. Si hay dudas no paralleliza.
- Se cubre más área de código con una simple directiva.
- Deja las manos libres al compilador para optimizar lo que crea oportuno.

Ambas aproximaciones son igual de válidas y el rendimiento puede ser equivalente



# ¿Y el Rendimiento?



# ¿Y el Rendimiento?

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
    #pragma acc parallel loop reduction(err:max)  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++) {  
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                               A[i][j-1]+ A[i][j+1]);  
            err = max(err, abs(AA[i][j] - A[i][j]));  
        }  
    #pragma acc parallel loop  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++)  
            A[i][j] = AA[i][j];  
    iter++;  
}
```

NO HA MEJORADO, VAMOS INCLUSO MÁS LENTOS QUE EL CÓDIGO SECUENCIAL EN LA CPU.

Las herramientas visuales son muy útiles para averiguar las causas de la pérdida de rendimiento



# Transferencias de datos

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
#pragma acc parallel loop reduction(err:max)  
for (int i=1; i<Nfil-1; i++)  
    for (int j=1; j<Ncol-1; j++) {  
        AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                           A[i][j-1]+ A[i][j+1]);  
        err = max(err, abs(AA[i][j] - A[i][j]));  
    }  
#pragma acc parallel loop  
for (int i=1; i<Nfil-1; i++)  
    for (int j=1; j<Ncol-1;  
         A[i][j] = AA[i][j];  
    iter++;  
}
```

Se hacen 4 copias  
de A y AA por  
iteración del  
bucle while.

Copia A, AA [HtoD]

Copia A, AA [DtoH]

Copia A, AA [HtoD]

Copia A, AA [DtoH]

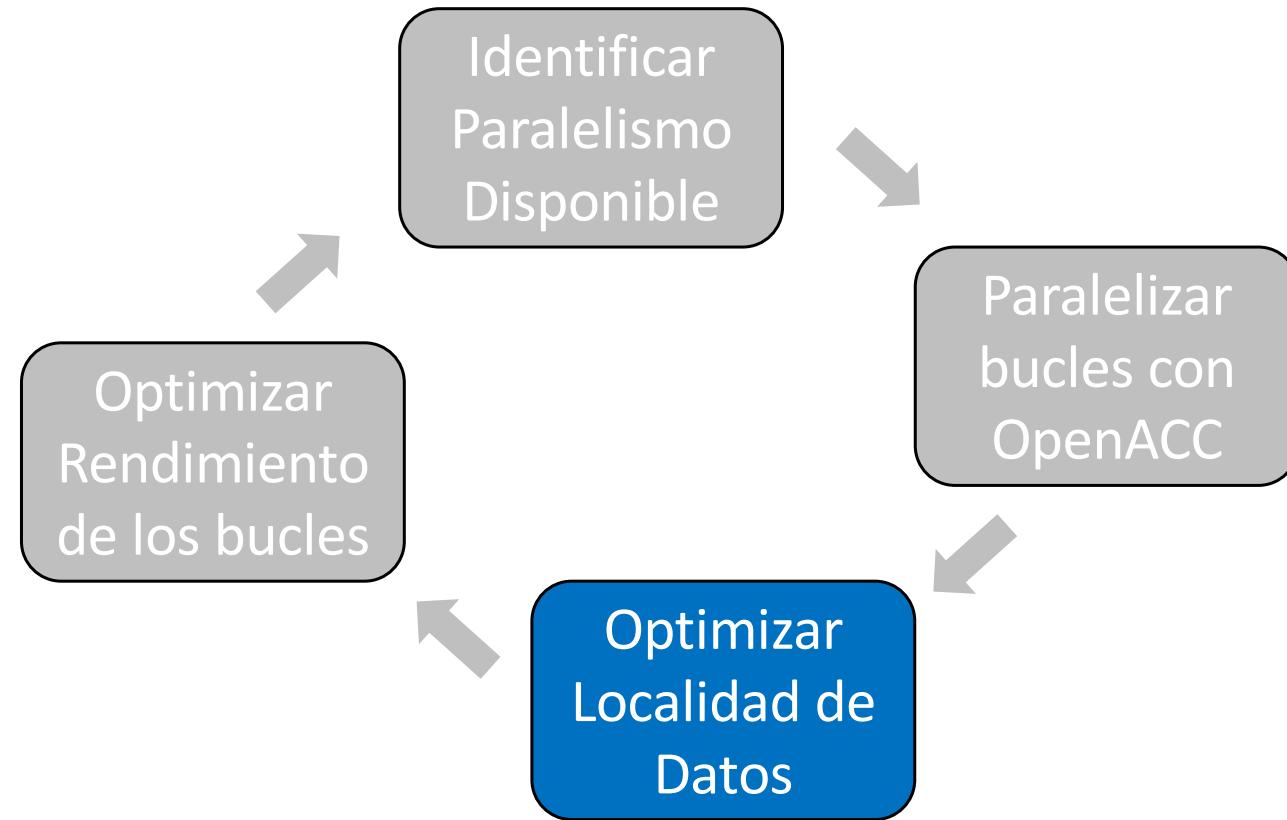
# Identificar Localidad de datos

```
while (err > xxx && iter < MaxIter) {  
    err = 0.0;  
  
    #pragma acc parallel loop reduction(err:max)  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++) {  
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+  
                               A[i][j-1]+ A[i][j+1]);  
            err = max(err, abs(AA[i][j] - A[i][j]));  
        }  
    #pragma acc parallel loop  
    for (int i=1; i<Nfil-1; i++)  
        for (int j=1; j<Ncol-1; j++)  
            A[i][j] = AA[i][j];  
    iter++;  
}
```

¿Necesita la CPU A y AA entre estos 2 bucles?

¿Necesita la CPU A y AA entre iteraciones del bucle while?

# Ciclo de diseño de una aplicación paralela con OpenACC



# Definición de regiones de datos

```
#pragma acc data [clausulas]
{
    #pragma acc parallel loop
    . . .
    #pragma acc parallel loop
    . . .
}
```

Los arrays definidos dentro de esta región se quedarán en la GPU hasta que se acabe su ejecución.

El constructor **data**, define una región de código en la que los arrays de la GPU permanecen en la GPU y son compartidos entre todos los kernels de esa región.



# Definición de regiones de datos: cláusulas

- **copy (list)**, asigna memoria de la GPU, copia datos HtoD cuando entra en la región, y copia datos DtoH cuando sale de la región.
- **copyin (list)**, asigna memoria de la GPU y copia datos HtoD cuando entra en la región.
- **copyout (list)**, asigna memoria de la GPU y copia datos DtoH cuando sale de la región.
- **create (list)**, asigna memoria de la GPU sin copia.
- **present (list)**, los datos ya están en la GPU, pertenecen a otra región de datos.
  
- Combinaciones: **present\_or\_copy**, **present\_or\_copyin**,  
**present\_or\_copyout**, **present\_or\_create**



# Definición de regiones de datos: descripción arrays

- A veces el compilador no puede determinar el tamaño de los arrays y hay que especificarlo de forma explícita:

```
#pragma acc data copyin(x[0:N]) copyout(y[0:N])
```

La sintaxis es **x[start:count]**.

- Importante: estas cláusulas pueden usarse en **data**, **parallel** o **kernels**.



# Identificar Localidad de datos

```
#pragma acc data copy(A) create(AA)
while (err > xxx && iter < MaxIter) {
    err = 0.0;
    #pragma acc parallel loop reduction(err:max)
    for (int i=1; i<Nfil; i++)
        for (int j=1; j<Ncol; j++) {
            AA[i][j] = 0.25*(A[i-1][j]+ A[i+1][j]+
                               A[i][j-1]+ A[i][j+1]);
            err = max(err, abs(AA[i][j] - A[i][j]));
        }
    #pragma acc parallel loop
    for (int i=1; i<Nfil; i++)
        for (int j=1; j<Ncol; j++)
            A[i][j] = AA[i][j];
    iter++
}
```

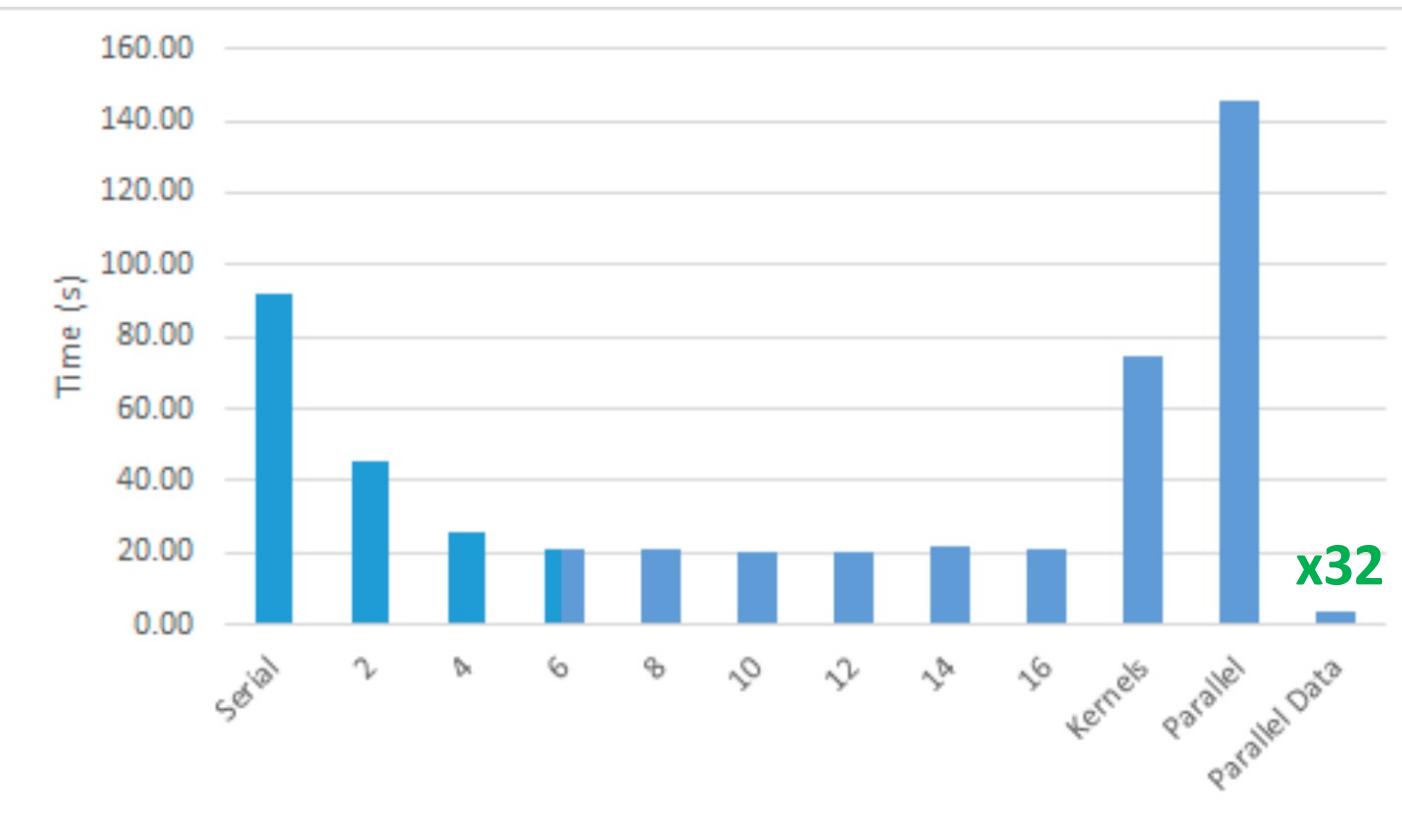
Al entrar en la región se asigna espacio en el device para A y se copia HtoD.

Al entrar en la región se asigna espacio en el device para AA. Es un array temporal que se destruirá al salir.

Al salir de la región se copiará A DtoH.



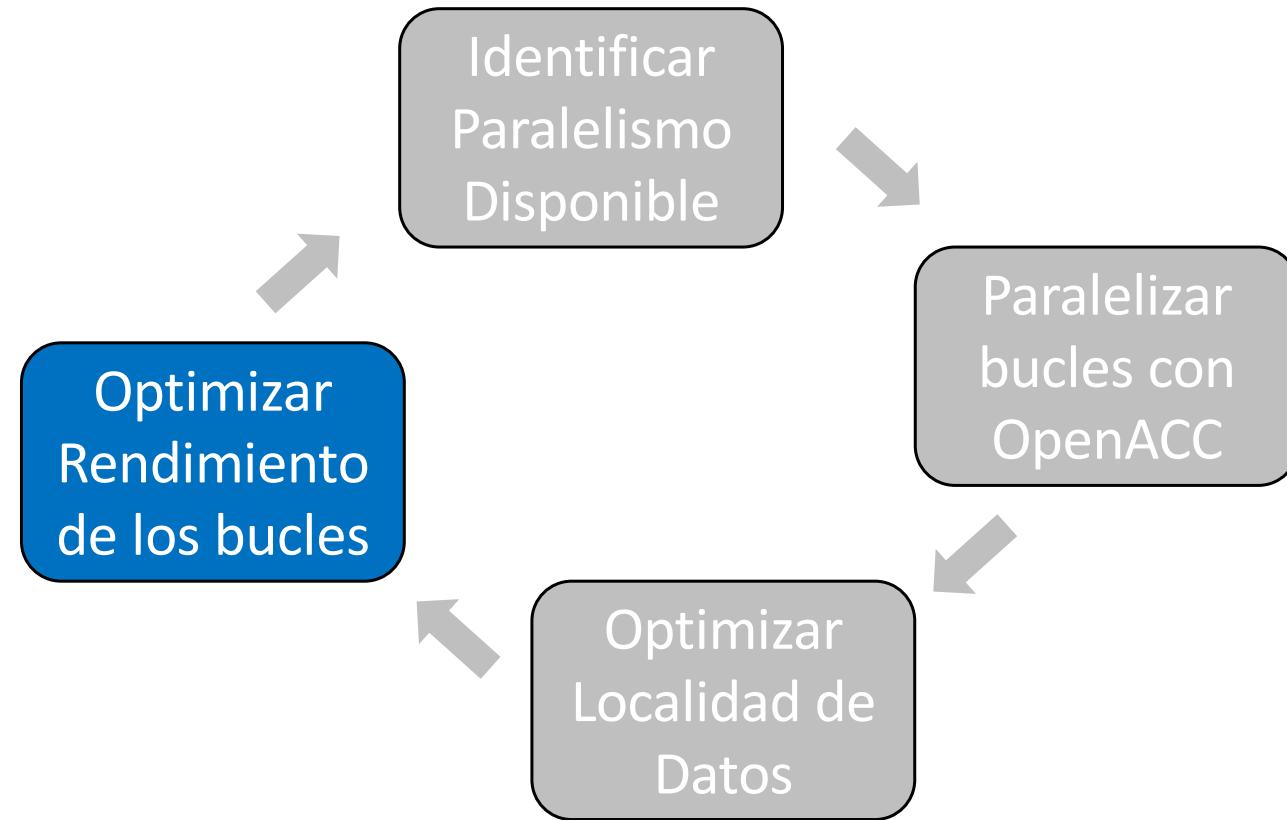
# Rendimiento



openacc.org



# Ciclo de diseño de una aplicación paralela con OpenACC

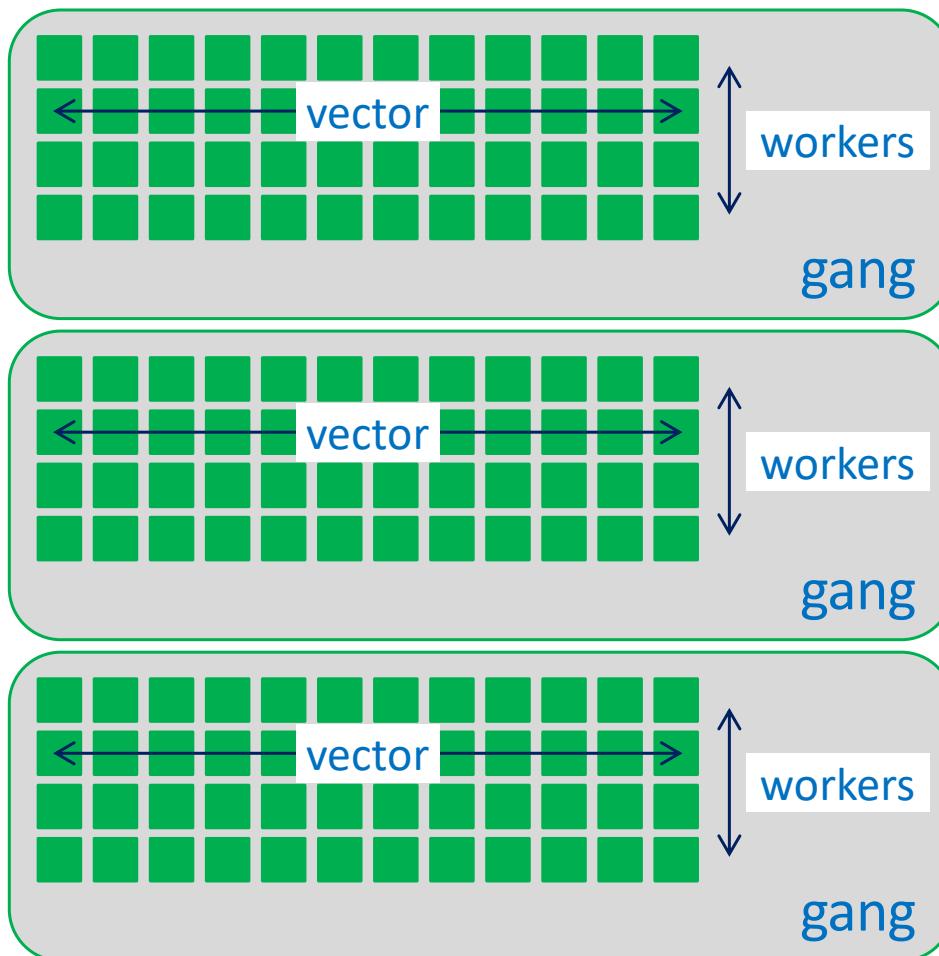


# Optimizar Rendimiento de los Bucles

- Optimizar a este nivel puede provocar problemas de portabilidad.
- El compilador puede realizar esta tarea. Pero, en general, si queremos obtener rendimientos elevados hay que optimizar a este nivel.
- La razón fundamental es que el compilador desconoce, en general, cuantas iteraciones se realizan en un bucle.
- Tenemos 3 niveles de paralelismo:
  - **Vector**, los threads trabajan juntos (paralelismo SIMD/SIMT)
  - **Workers**, calculan 1 vector
  - **Gangs**, tienen 1 o más workers que comparten recursos (cache, SM, ...).  
Los gangs trabajan independientemente unos de otros.



# Open ACC, 3 niveles de paralelismo



- Clausula **gang**, partitiona el bucle en gangs
  - Clausula **worker**, partitiona el bucle en workers
  - Clausula **vector**, vectoriza el bucle
- 
- Normalmente se combinan
  - Nota: Las GPUs de NVIDIA necesitan tamaño de vector múltiplo de 32 (*warp size*)



# 3 niveles de paralelismo: gang, worker, vector

```
#pragma acc parallel loop gang
for (i=0; i<N; i++)
    #pragma acc parallel loop vector
    for (j=0; j<M; j++)
        . . .
```

Fuerza al compilador a vectorizar el bucle más interno.  
Por defecto, el compilador intenta vectorizar con tamaño de vector 128.

```
#pragma acc parallel vector_length(32)
. . .
#pragma acc parallel loop vector(64)
for (j=0; j<M; j++)
. . .
```

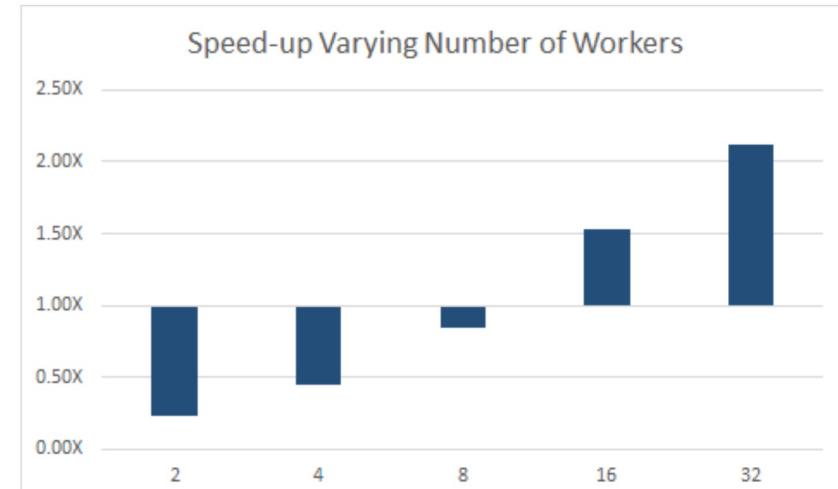
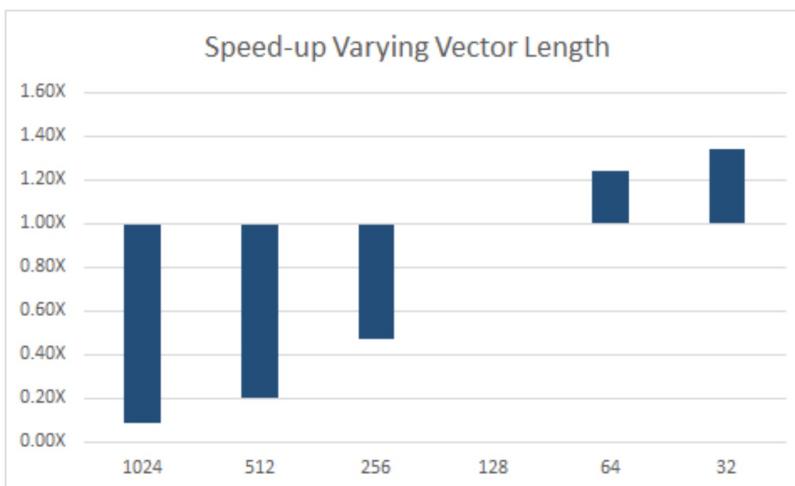
Se puede redefinir el tamaño de vector de formas diversas.

Se puede controlar el tamaño de estos elementos con las cláusulas: `num_gangs(n)`, `num_workers(n)`, `vector_length(n)`.



# 3 niveles de paralelismo: gang, worker, vector

- Variar el número de gangs, workers o tamaño de vector puede impactar en el rendimiento final. Equivale a jugar con el número de Bloques y Threads por Bloque en CUDA.



openacc.org

- Encontrar los valores óptimos de `num_gangs(n)`, `num_workers(n)` y `vector_length(n)`, puede ser costoso. Aunque las mejoras de rendimiento pueden ser sustanciales.



# Collapse Clause

- Se aplica a un grupo de bucles anidados.

```
#pragma acc parallel loop collapse(2)
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        . . .
```



```
#pragma acc parallel loop
for (ij=0; i<N*M; ij++)
    . . .
```

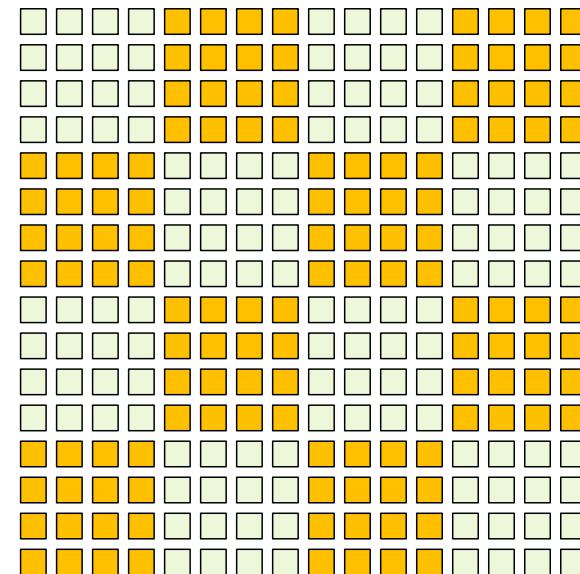
- Colapsar los bucles externos permite crear más gangs.
- Colapsar los bucles internos permite aumentar el tamaño de los vectores
- Si se puede, colapsarlo todo es buena idea.



# Tile Clause

- Partir los bucles en pequeños bloques para aprovechar la localidad.

```
#pragma acc loop tile(4,4)
for (i=1; i<N-1; i++)
    for (j=1; j<M-1; j++)
        AA[i][j] = 0.25* (A[i-1][j] +
                            A[i+1][j] +
                            A[i][j-1] +
                            A[i][j+1])
```



# Productividad vs Rendimiento

Un mismo problema se puede atacar de 3 formas diferentes:

## OpenACC

- Productividad Alta
- Rendimiento limitado, dependiente del problema

## OpenACC + CUDA

- Productividad y Rendimiento razonable

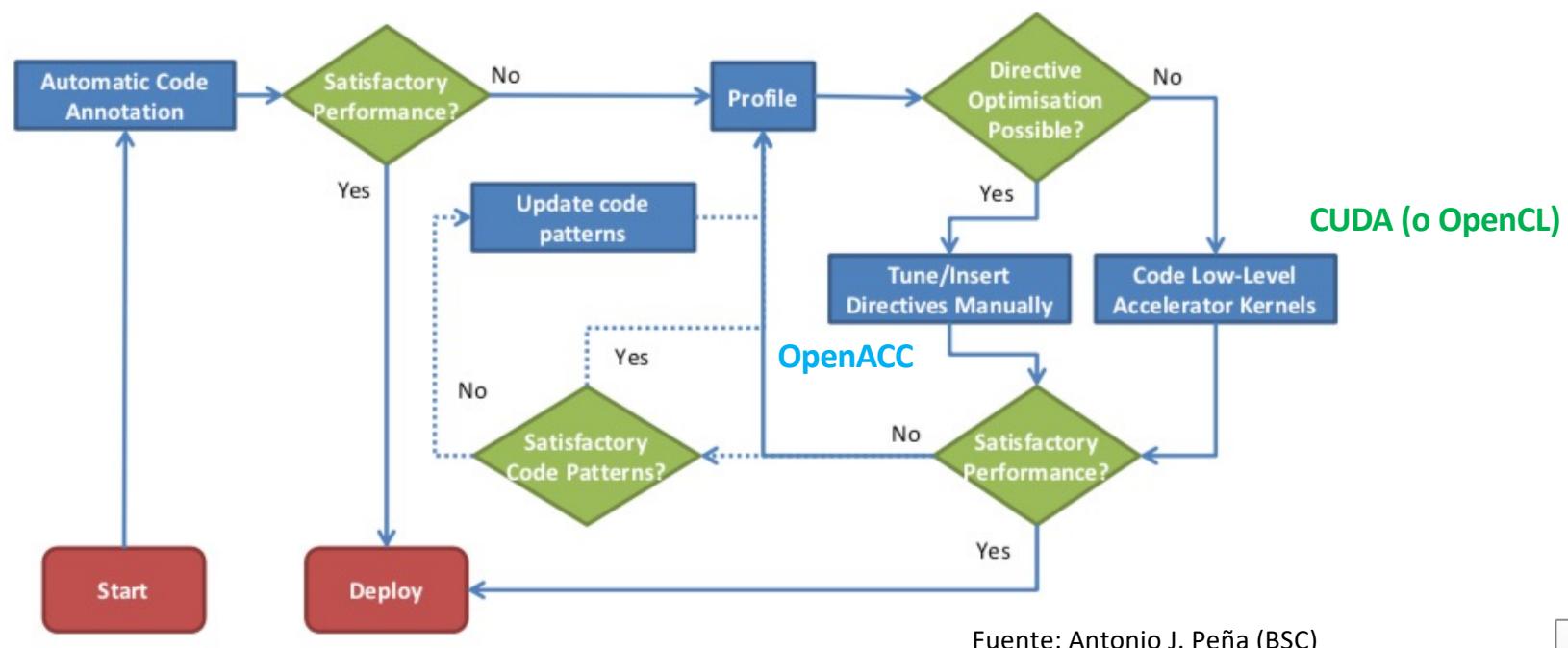
## Cuda

- Productividad Baja
- Rendimiento Elevado, dependiente del problema



# Productividad vs Rendimiento

## Proposed Methodology for End Users



Fuente: Antonio J. Peña (BSC)



# Conclusiones

## OpenACC

- SIMPLE
- ROBUSTO
- PORTABLE

## Múltiples casos de éxito

### Riken Japón, NICAM Climate Modeling

- 7.8x speed up
- 5% código modificado

### Unin. Illinois, PowerGrid – MRI Reconstruction

- 70x speed up
- 2 días de trabajo

### LS-DALTON

- <100 líneas de código modificadas
- 1 semana de trabajo
- Gran rendimiento, 8-12x speedup



# Referencias

- Página de OpenACC: [www.openacc.org](http://www.openacc.org)
- OpenACC Programming and Best Practices Guide.  
June 2015
- OpenACC, Parallel Programming with OpenACC  
Edited by Rob Farber  
Morgan Kaufmann Publishers, 2017





UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Departament d'Arquitectura de Computadors

# Tarjetas Gráficas y Aceleradores

## OpenACC

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

