

CUDA

Operaciones Atómicas



`X++;`

read-modify-write operation

1. Read the value in `x`.
2. Add 1 to the value read in step 1.
3. Write the result back to `x`.

Condición de carrera

Uno puede tener como resultado 9 u 8

Estaría bueno que exista algo que impida este comportamiento. Estas justamente son las operaciones atómicas. Si un hilo está empleando un valor, traba esa variable para los demás hilos hasta que se deje de usar. Esto se debe hacer con cuidado porque puede romper el paralelismo

Table 9.2 Two threads incrementing the value in `x`

| STEP | EXAMPLE |
|--|---------------------------------|
| 1. Thread A reads the value in <code>x</code> . | A reads 7 from <code>x</code> . |
| 2. Thread A adds 1 to the value it read. | A computes 8. |
| 3. Thread A writes the result back to <code>x</code> . | <code>x <- 8.</code> |
| 4. Thread B reads the value in <code>x</code> . | B reads 8 from <code>x</code> . |
| 5. Thread B adds 1 to the value it read. | B computes 9. |
| 6. Thread B writes the result back to <code>x</code> . | <code>x <- 9.</code> |

Table 9.3 Two threads incrementing the value in `x` with interleaved operations

| STEP | EXAMPLE |
|---|---------------------------------|
| Thread A reads the value in <code>x</code> . | A reads 7 from <code>x</code> . |
| Thread B reads the value in <code>x</code> . | B reads 7 from <code>x</code> . |
| Thread A adds 1 to the value it read. | A computes 8. |
| Thread B adds 1 to the value it read. | B computes 8. |
| Thread A writes the result back to <code>x</code> . | <code>x <- 8.</code> |
| Thread B writes the result back to <code>x</code> . | <code>x <- 8.</code> |

Read-Modify-Write aritmética o lógica en memoria global o compartida

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. In the case of `float2` or `float4`, the read-modify-write operation is performed on each element of the vector residing in global memory. For example, `atomicAdd()` reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. Atomic functions can only be used in device functions.

The atomic functions described in this section have ordering `cuda::memory_order_relaxed` and are only atomic at a particular `scope`:

- Atomic APIs with `_system` suffix (example: `__atomicAdd_system`) are atomic at scope `cuda::thread_scope_system`.
- Atomic APIs without a suffix (example: `__atomicAdd`) are atomic at scope `cuda::thread_scope_device`.
- Atomic APIs with `_block` suffix (example: `__atomicAdd_block`) are atomic at scope `cuda::thread_scope_block`.

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                      unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 *address, __half2 val);
__half atomicAdd(__half *address, __half val);
__nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);
__nv_bfloat16 atomicAdd(__nv_bfloat16 *address, __nv_bfloat16 val);
float2 atomicAdd(float2* address, float2 val);
float4 atomicAdd(float4* address, float4 val);
```

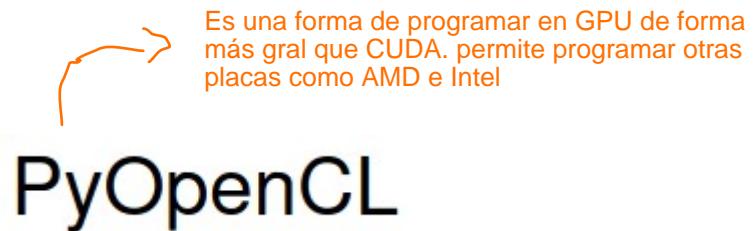
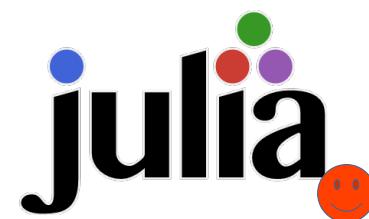
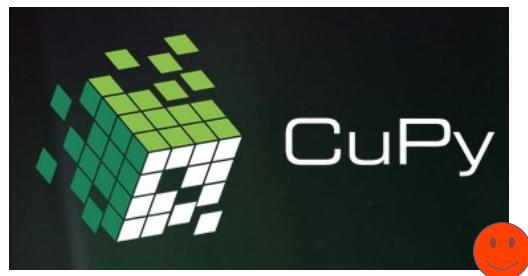
reads the 16-bit, 32-bit or 64-bit `old` located at the address `address` in global or shared memory, computes `(old + val)`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

Manos a la obra

/share/apps/icnpg/clases/Cuda_Basico/atomics/

ICNPG_atomicas.ipynb

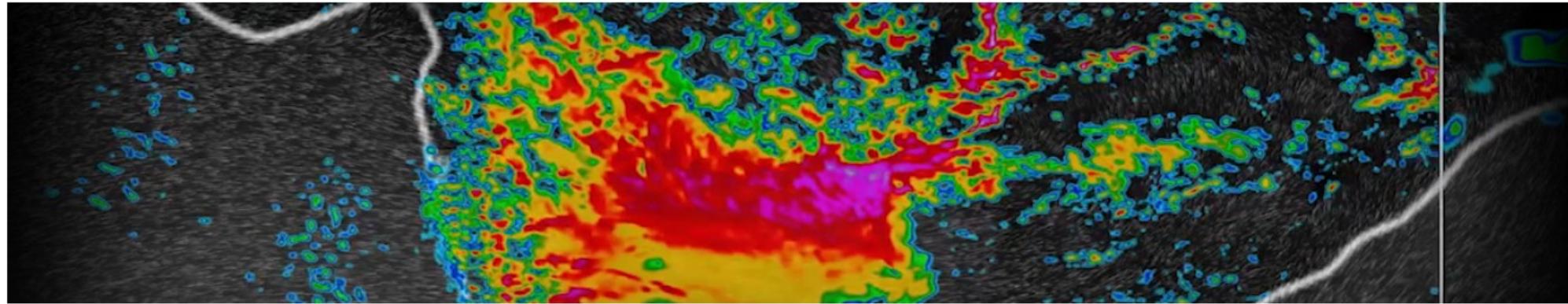
- histograma.cu
- histograma.py
- histograma_thrust_sin_atomics.cu
- reduccion.cu



La magia de los compiladores

OPENACC





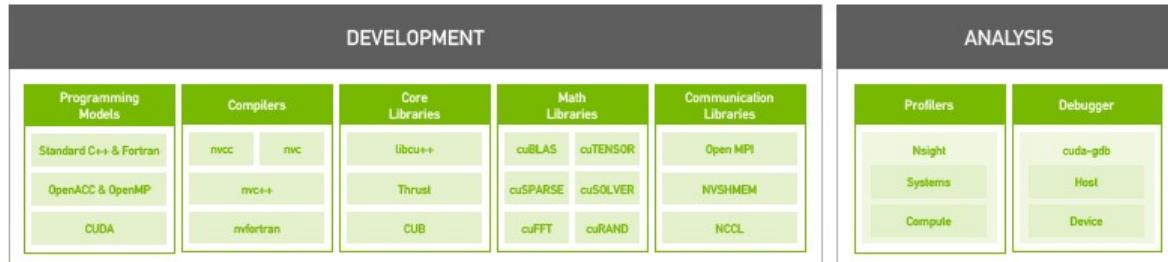
OpenACC no está en el Toolkit común, sino en el toolkit

NVIDIA HPC SDK

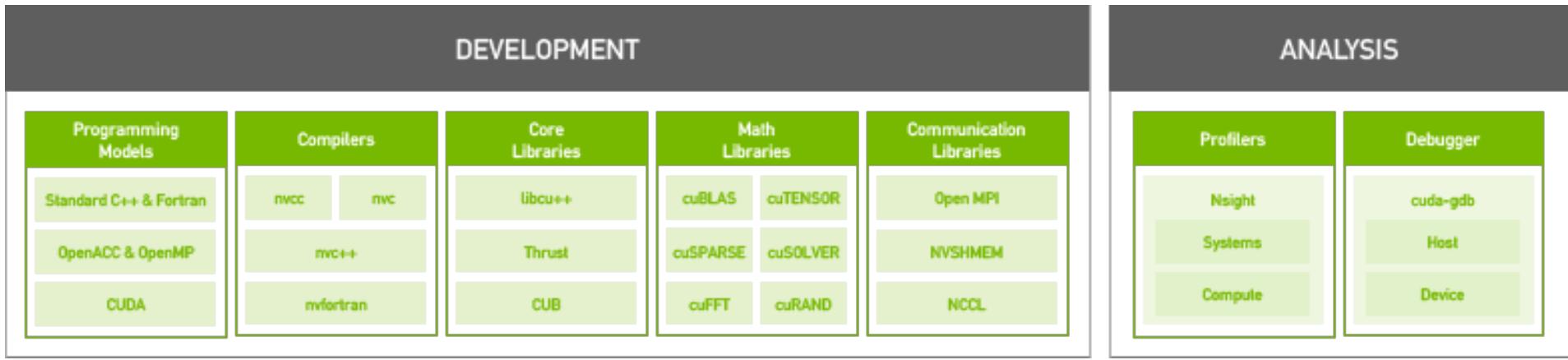
Está instalado en el cluster de sólidos.
El módulo se llama nvhpc-21.9

A Comprehensive Suite of Compilers, Libraries and Tools for HPC

The NVIDIA HPC Software Development Kit (SDK) includes the proven compilers, libraries and software tools essential to maximizing developer productivity and the performance and portability of HPC applications.



<https://developer.nvidia.com/hpc-sdk>



Compilers

HPC Compilers Documentation

HPC Compiler Documentation Library

nv

nv is a C11 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs. It invokes the C compiler, assembler, and linker for the target processors with options derived from its command line arguments. nv supports ISO C11, supports GPU programming with OpenACC, and supports multicore CPU programming with OpenACC and OpenMP.

nvc++

nvc++ is a C++17 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs. It invokes the C++ compiler, assembler, and linker for the target processors with options derived from its command line arguments. nvc++ supports ISO C++17, supports GPU programming with C++17 parallel algorithms (pSTL) and OpenACC, and supports multicore CPU programming with OpenACC and OpenMP.

nvfortran

nvfortran is a Fortran compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs. It invokes the Fortran compiler, assembler, and linker for the target processors with options derived from its command line arguments. nvfortran supports ISO Fortran 2003 and many features of ISO Fortran 2008, supports GPU programming with CUDA Fortran and OpenACC, and supports multicore CPU programming with OpenACC and OpenMP.

nvcc

nvcc is the CUDA C and CUDA C++ compiler driver for NVIDIA GPUs. nvcc accepts a range of conventional compiler options, such as for defining macros and include/library paths, and for steering the compilation process. nvcc produces optimized code for NVIDIA GPUs and drives a supported host compiler for AMD, Intel, OpenPOWER, and Arm CPUs.

Programming Models

C++ Parallel Algorithms

C++ 17 Parallel Algorithms introduce parallel and vector concurrency through execution policies and are supported in the NVC++ compiler.

OpenACC Getting Started Guide

This guide introduces the NVIDIA OpenACC implementation, including examples of how to write, build and run programs using the OpenACC directives.

OpenMP

This section describes using OpenMP, a set of compiler directives, an applications programming interface (API), and a set of environment variables for specifying parallel execution in Fortran, C++ and C programs.

CUDA C++ Programming Guide

A comprehensive guide to understanding and developing and optimizing code in the CUDA C++ programming environment.

CUDA Fortran Programming Guide

This guide describes how to program with CUDA Fortran, a small set of extensions to Fortran that supports and is built upon the NVIDIA CUDA programming model. CUDA Fortran is available on a variety of 64-bit operating systems for both x86 and OpenPOWER hardware platforms. CUDA Fortran includes runtime APIs and programming examples.

¡El compilador se encarga!

DEVELOPER BLOG

Uno puede usar la librerías estándar de C++ y si lo compila con nvc++, directamente paralleliza algunas operaciones. Lo malo es que no permite optimizar demasiado y hace copias de CPU a GPU (y visceversa)



Standard Parallelism

All data movement between host memory and GPU device memory is performed implicitly and automatically under the control of CUDA Unified Memory.

```
std::sort(employees.begin(), employees.end(),
          CompareByLastName());
```

HPC

Aug 04, 2020

Accelerating Standard C++ with GPUs Using stdpar

By David Olsen, Graham Lopez and Bryce Adelstein Lelbach

Tags: accelerated computing, and compilers, C++, HPC SDK, Languages, programming languages, standard parallelism

Discuss (2)

```
nvc++ -stdpar program.cpp -o program
```

```
std::sort(std::execution::par,
          employees.begin(), employees.end(),
          CompareByLastName());
```

<https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/>

¡El compilador se encarga!

DEVELOPER BLOG

Fortran

Standard Parallelism

HPC

Nov 16, 2020

Accelerating Fortran DO CONCURRENT with GPUs and the NVIDIA HPC SDK

By Guray Ozen and Graham Lopez

Tags: compilers, CUDA, Fortran, HPC SDK, Languages, programming languages, standard parallelism

 Discuss (17)

<https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/>

```
1 subroutine saxpy(x,y,n,a)
2   real :: a, x(:), y(:)
3   integer :: n, i
4   do i = 1, n
5     y(i) = a*x(i)+y(i)
6   enddo
7 end subroutine saxpy
```



```
1 subroutine saxpy(x,y,n,a)
2   real :: a, x(:), y(:)
3   integer :: n, i
4   do concurrent (i = 1: n)
5     y(i) = a*x(i)+y(i)
6   enddo
7 end subroutine saxpy
```

¡El compilador se encarga!

DEVELOPER BLOG



DATA SCIENCE | HPC

Nov 10, 2020

Accelerating Python on GPUs with nvc++ and Cython

By [Ashwin Srinath](#)

Tags: [accelerated computing](#), [Cython](#), [HPC SDK](#), [Python](#)

Discuss (6)

```
from libcpp.algorithm cimport sort, copy_n
from libcpp.vector cimport vector
from libcpp.execution cimport par

def cppsort(int[:] x):
    cdef vector[int] temp
    temp.resize(len(x))
    copy_n(&x[0], len(x), temp.begin())
    sort(par, temp.begin(), temp.end())
    copy_n(temp.begin(), len(x), &x[0])
```

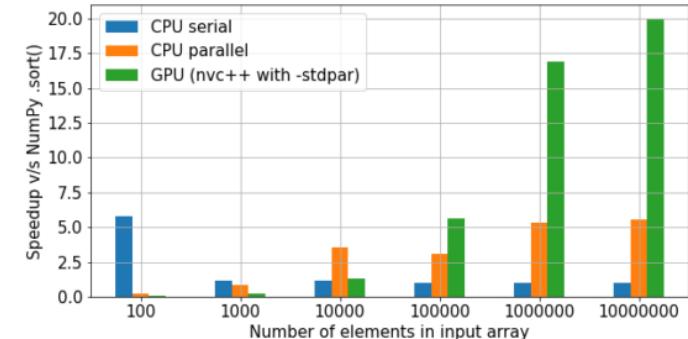


Figure 3. Speedup obtained vs. NumPy for sorting a sequence of integers (larger is better). CPU benchmarks were run on a system with an Intel Xeon Gold 6128 CPU. GPU benchmarks were run on an NVIDIA A100 GPU.

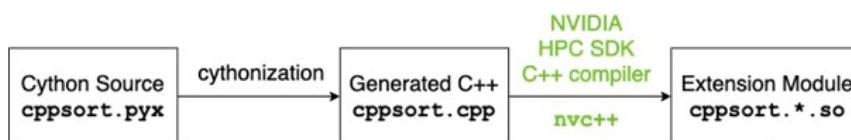


Figure 2. Building an extension module using the nvc++ compiler.

```
In [3]: x = np.array([4, 3, 2, 1], dtype="int32")
In [4]: cppsort(x) # uses the GPU!
```

Key Features

Direct GPU Programming

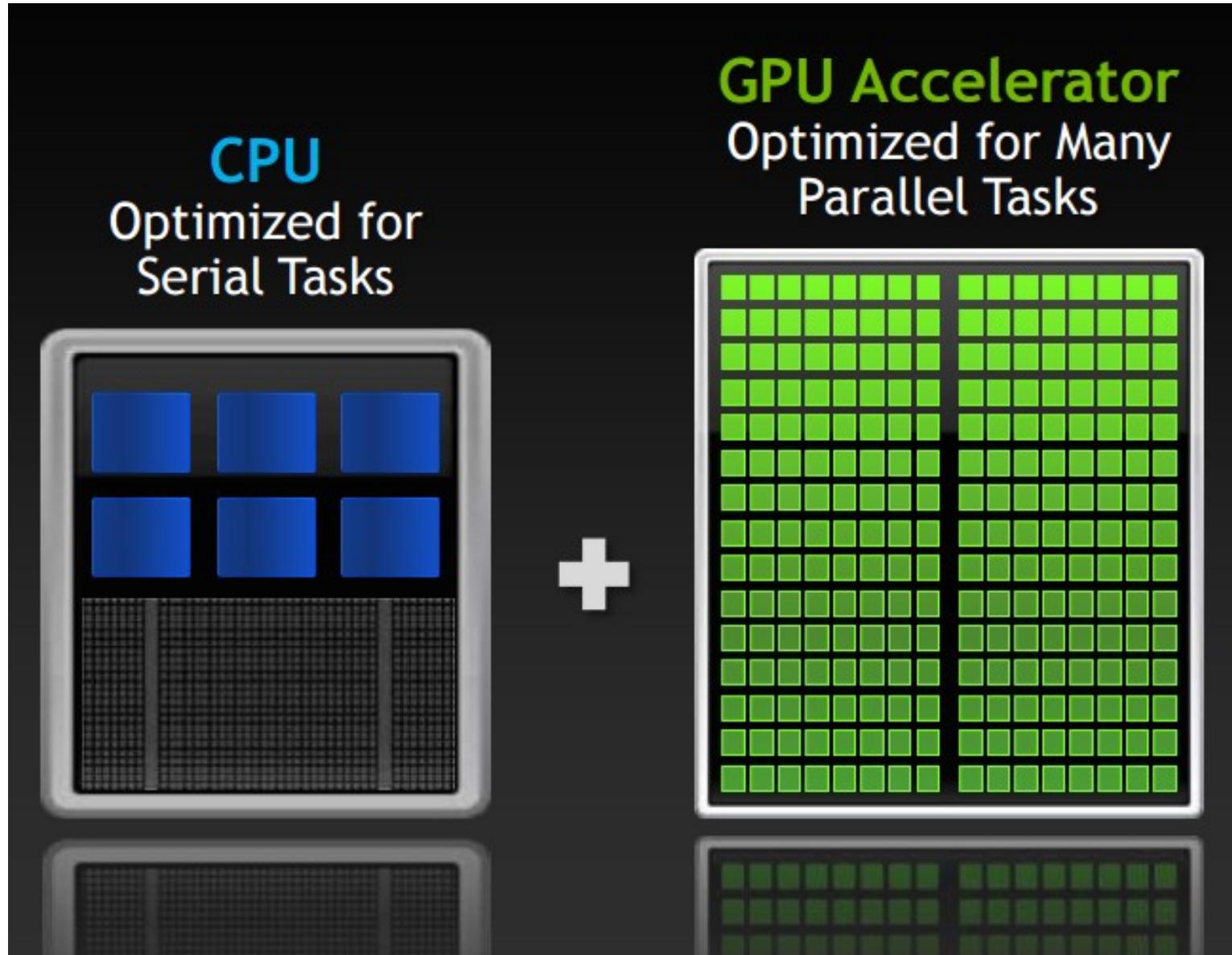
CUDA Fortran is a low-level explicit programming model with substantial runtime library components that gives expert Fortran programmers direct control over all aspects of GPU programming. CUDA Fortran enables programmers to access and control all the newest GPU features including CUDA Managed Data, Cooperative Groups and Tensor Cores. CUDA Fortran includes several productivity enhancements such as Loop Kernel Directives, module interfaces to the NVIDIA GPU math libraries and OpenACC interoperability features.



| Host code | Device Code |
|---|--|
| <pre>program t1 use cudafor use mytests integer, parameter :: n = 100 integer, allocatable, device :: iarr(:) integer h(n) istat = cudaSetDevice(0) allocate(iarr(n)) h = 0; iarr = h call test1<<<1,n>>> (iarr) h = iarr print *,& "Errors: ", count(h.ne.(/ (i,i=1,n) /)) deallocate(iarr) end program t1</pre> | <pre>module mytests contains attributes(global) & subroutine test1(a) integer, device :: a(*) i = threadIdx%x a(i) = i return end subroutine test1 end module mytests</pre> |

nvfortran -cuda

Computación heterogénea



Otra forma de acelerar aplicaciones ...



Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages
(CUDA, ..)

High Level
Languages
(Matlab, ..)

CUDA Libraries are
interoperable with OpenACC

CUDA Language is
interoperable with OpenACC

Easiest Approach

Maximum
Performance

No Need for
Programming Expertise

¿ Que es OPENACC ?



<http://www.openacc.org/>

¿Que es OPENACC?

- Las *directivas de OpenACC* son “hints al compilador” que permiten, sin requerir un nuevo lenguaje como CUDA o OPENCL, acelerar códigos **seriales** en **C/C++ y Fortran** rápidamente en
 - I. NVIDIA GPUs
 - II. x86 o ARM CPUs
 - III. Intel Xeon Phi
 - IV. AMD Radeon,
- *El compilador se encarga de paralelizar el código serial, con ayuda de la información que le damos. Mas allá de eso, delegamos completamente la implementación (muy similar a OPENMP).*

<http://www.openacc.org/>

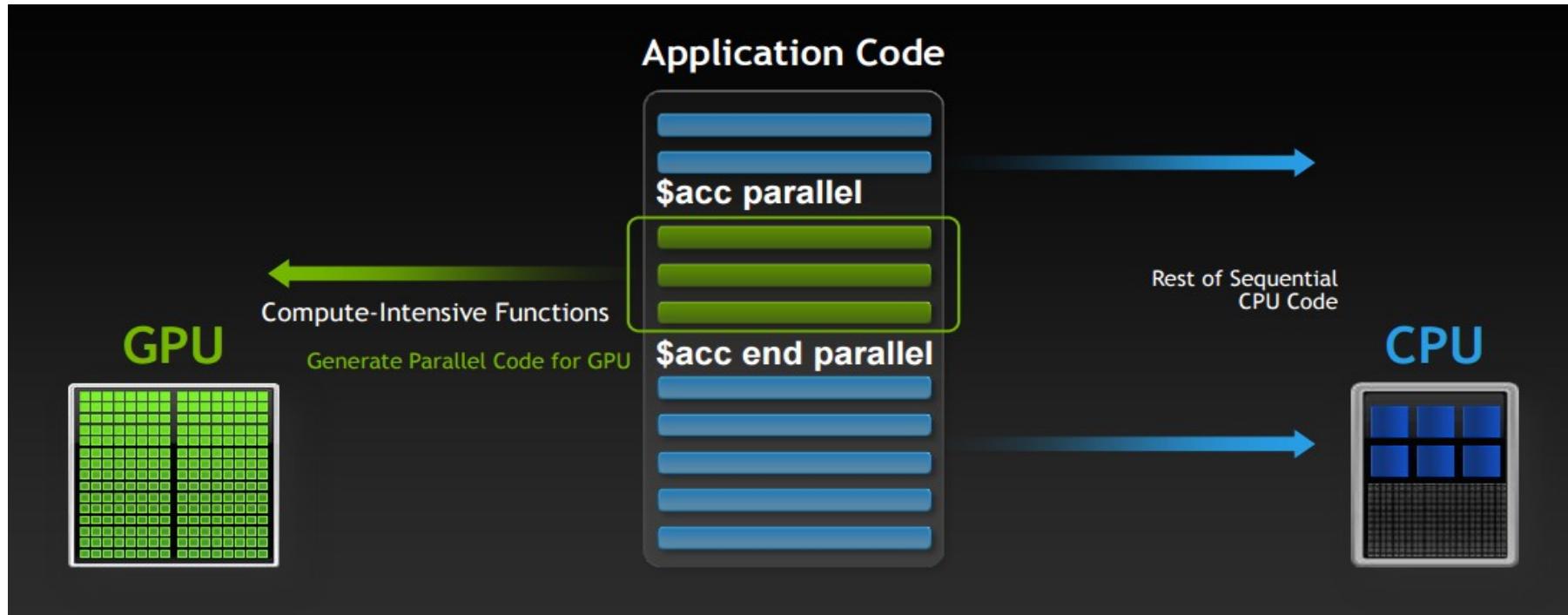
¿Para que sirve OPENACC?

- **Tienen un código serial C/C++/fortran y quieren intentar acelerarlo “rápidamente”:**
 - Modificando mínimamente el código serial original ...
 - Que aún con estas mínimas “adiciones” sea portable y pueda correr normalmente en CPU, como siempre ...
 - Sin necesidad de saber CUDA, OPENCL, OPENMP, etc
 - Sin pretender máxima-performance pero si un mínimo esfuerzo redituable

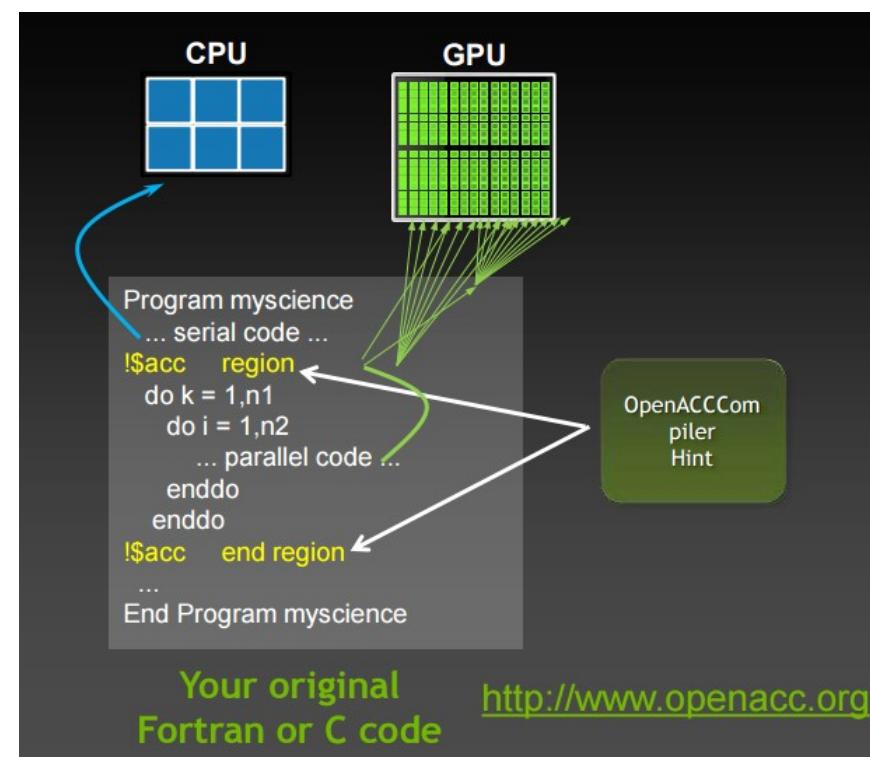
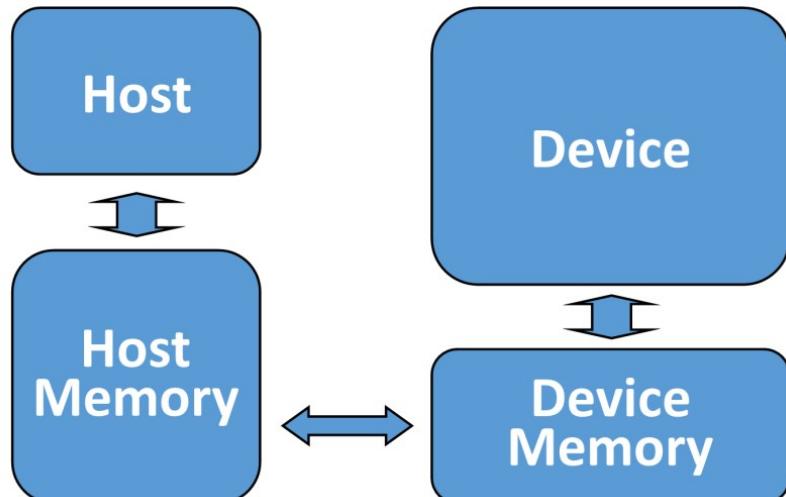
Por debajo se generan kernels de CUDA pero lo hace el compilador!

¿Para quien es OPENACC?

- Heredaste códigos seriales y querés acelerarlos...
- Querés parallelizar pero tenés poco tiempo para programar y/o aprender otro lenguaje
- Querés brindar un “servicio de aceleración” a alguien, sin destruirle el código
- Querés portar a GPU o multicore un software ya hecho (aplicación, librería, etc) con unos mínimos toques...



A través de hints se le dice al compilador qué sección de código se podría paralelizar y luego él se encarga de paralelizarlo intentando con equivocarse. Le podemos dar mayores o menores restricciones



Serial Code

Single CPU Core Performance: 1x

```
.  
. .  
. .  
for(int j=1;j<ny-1;j++) {  
    for(int k=i1;k<nz-1;k++) {  
        for(int i=1;i<nx-1;i++) {  
            Anext[Index3D (nx,ny,i,j,k)] =  
                (A0[Index3D (nx,ny,i,j,k+1)] +  
                 A0[Index3D (nx,ny,i,j,k-1)] +  
                 A0[Index3D (nx,ny,i,j+1,k)] +  
                 A0[Index3D (nx,ny,i,j-1,k)] +  
                 A0[Index3D (nx,ny,i+1,j,k)] +  
                 A0[Index3D (nx,ny,i-1,j,k)])*c1  
                -A0[Index3D (nx,ny,i,j,k)]*c0;  
        }  
    }  
}
```

Parallel Code for GPU

Add One OpenACC Directive
Tesla K40 Perf: 13.6x

- Podemos trabajar cada i,j,k por separado
- ((Cualquier otro compilador no le presta atención a esto))

```
#pragma acc parallel loop collapse(3)  
for(int j=1;j<ny-1;j++) {  
    for(int k=i1;k<nz-1;k++) {  
        for(int i=1;i<nx-1;i++) {  
            Anext[Index3D (nx,ny,i,j,k)] =  
                (A0[Index3D (nx,ny,i,j,k+1)] +  
                 A0[Index3D (nx,ny,i,j,k-1)] +  
                 A0[Index3D (nx,ny,i,j+1,k)] +  
                 A0[Index3D (nx,ny,i,j-1,k)] +  
                 A0[Index3D (nx,ny,i+1,j,k)] +  
                 A0[Index3D (nx,ny,i-1,j,k)])*c1  
                -A0[Index3D (nx,ny,i,j,k)]*c0;  
        }  
    }  
}
```

Cuantos más hints se le da, mejor el kernel que construirá el compilador

Dual socket E5-2698 v3 @2.3GHz (Haswell), 16 cores per socket, 256 GB memory, 1x Tesla K40

Benchmark: Parboil Stencil from University of Illinois with 1000 iterations

Source code for Parboil: <http://impact.crhc.illinois.edu/Parboil/parboil.aspx>

Los códigos para hacer benchmark son códigos para testear performance

<https://developer.nvidia.com/openacc-toolkit>

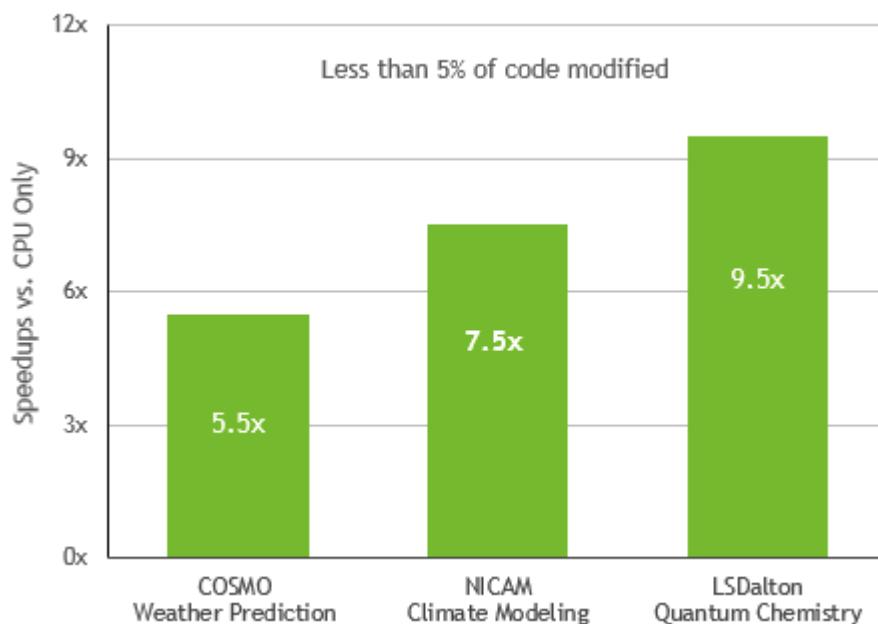
The PGI Community Edition with OpenACC offers scientists and researchers a quick path to accelerated computing with less programming effort. By inserting compiler "hints" or directives into your C, C++ or Fortran code, the PGI OpenACC compiler will offload and run your code on the GPU and CPU.

In addition to the PGI OpenACC compilers, the PGI Community Edition includes GPU-enabled libraries and developer tools to help you with your GPU acceleration effort.

Get Your Free PGI Community Edition Today

[DOWNLOAD](#)

- **Simple:** Minimum code modifications to start on GPUs
- **Powerful:** Up to 10x faster application performance
- **Portable:** Same source code runs on CPUs and GPUs



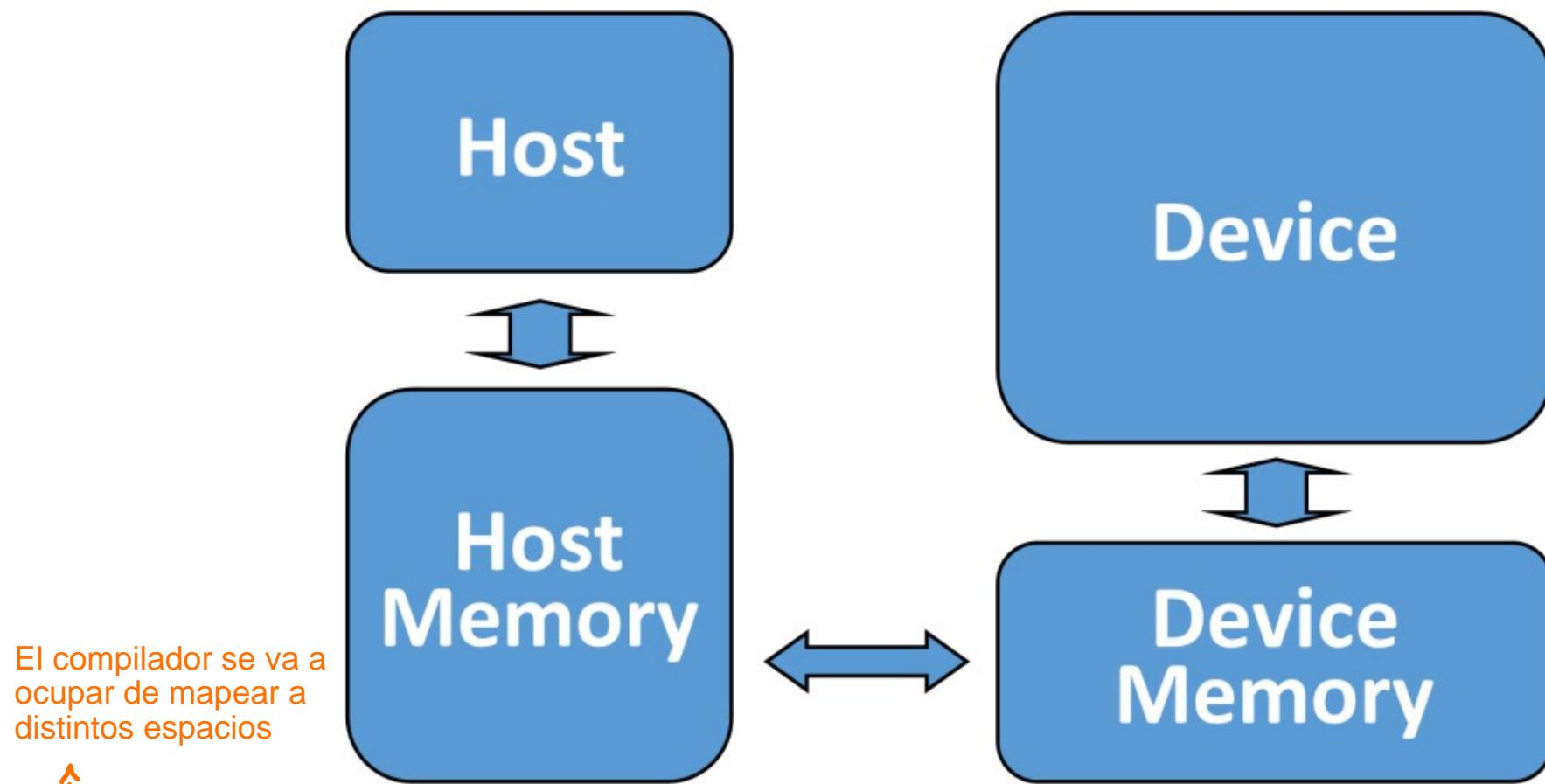
LS-DALTON: Benchmark on Oak Ridge Titan Supercomputer, AMD CPU vs Tesla K20X GPU. Test input: Alanine-3 on CCSD(T) module..

COSMO: additional information [here](#).

NICAM: Benchmark on TiTech TSUBAME 2.5, Westmere CPU vs. K20X, additional information [here](#).

El acelerador no tiene que ser necesariamente GPU, sino que tamb se puede usar CPU Multicore

Modelo OPENACC

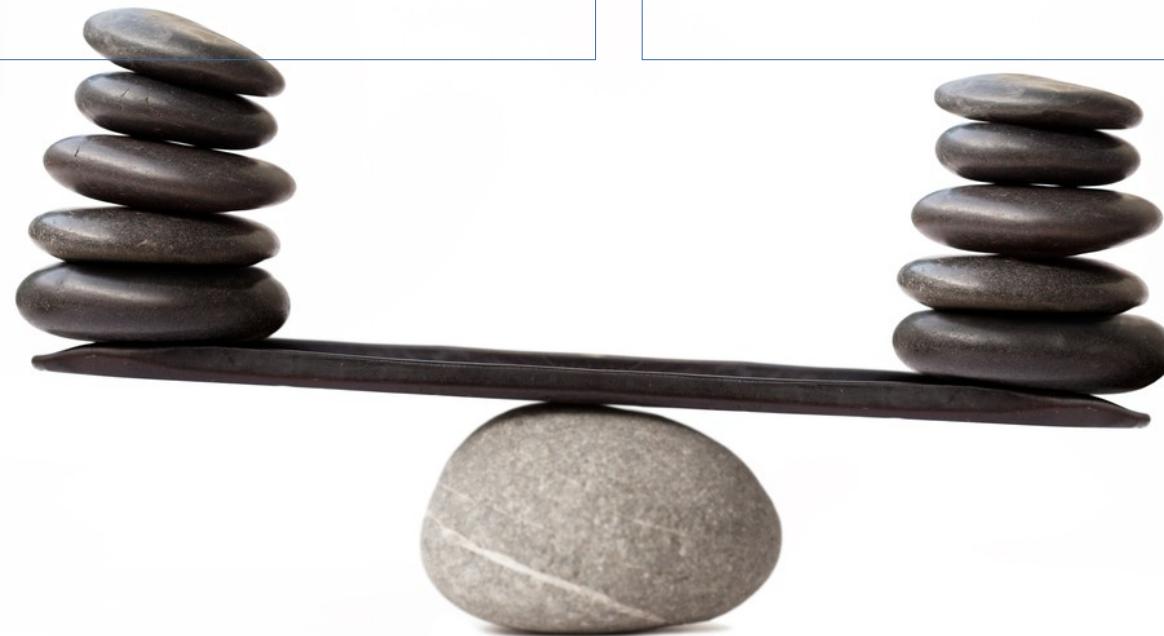


For developers coming to OpenACC from other accelerator programming models, such as CUDA or OpenCL, where host and accelerator memory is frequently represented by two distinct variables (`host_A[]` and `device_A[]`, for instance), it's important to remember that **when using OpenACC a variable should be thought of as a single object**, regardless of whether it's backed by memory in one or more memory spaces.

Ventajas y desventajas

- Muy simple
- Muy Portable
- Fácil de acelerar código serial

- Performance menor a la posible con optimizaciones en CUDA y alguna de sus libs.



La performance puede ser mejorada gracias a su interoperabilidad con CUDA y sus libs, sacrificando portabilidad...

Compiladores

Commercial Compilers



Contact Cray Inc for more information.



NVIDIA HPC SDK. Free download.



Contact National Supercomputing
Center in Wuxi for more information.

Open Source Compilers



GCC 10

Includes support for OpenACC 2.6



Sourcery CodeBench (AMD GCN) Lite for X86_64 GNU/Linux

OPENACC y OPENMP

Familiar to OpenMP Programmers



OpenMP

CPU

```
main() {  
    double pi = 0.0; long i;  
  
    #pragma omp parallel for reduction(+:pi)  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

OpenACC

CPU

GPU

```
main() {  
    double pi = 0.0; long i;  
  
    #pragma acc kernels  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

$$\int_0^1 dx \frac{4}{1+x^2} = \pi$$

Esto se puede hacer en thrust con un transform para construir el integrando y luego un reduce

Ecuación de Laplace

- Ecuación de difusión
- Ecuación del calor
- Membrana elástica
- Electrostática...
- Etc.

$$V = f_N(x)$$

$$(\partial_x^2 + \partial_y^2)V = 0$$

$$V = f_O(y)$$

$$V(x, y) = ?$$

$$V = f_E(y)$$

$$V = f_S(x)$$

Método de Jacobi

$$(\partial_x^2 + \partial_y^2)V = 0$$

$$V(x = ih, y = jh) = V_{i,j},$$

$$V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{i,j} \approx 0$$

$$V_{i,j} \approx \frac{1}{4}(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1})$$

| | | |
|-------------|-------------|-------------|
| | $V_{i,j+1}$ | |
| $V_{i-1,j}$ | $V_{i,j}$ | $V_{i+1,j}$ |
| | $V_{i,j-1}$ | |

Método de Jacobi

for($n=0....)$ $V_{i,j}^{n+1} = \frac{1}{4}(V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n)$

| | | |
|---------------|-----------------|---------------|
| | $V_{i,j+1}^n$ | |
| $V_{i-1,j}^n$ | $V_{i,j}^{n+1}$ | $V_{i+1,j}^n$ |
| | $V_{i,j-1}^n$ | |

Si uno lo itera llega a la solución

Se puede demostrar que converge a una única solución cuando $n \rightarrow \infty$

$$V_{i,j}^* = \frac{1}{4}(V_{i+1,j}^* + V_{i-1,j}^* + V_{i,j+1}^* + V_{i,j-1}^*) \Rightarrow [\partial_x^2 + \partial_y^2]V^*(x, y) = 0$$

Método de Jacobi

$$V_{i,j}^{n+1} = \frac{1}{4}(V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n)$$

| | | |
|---------------|-----------------|---------------|
| | $V_{i,j+1}^n$ | |
| $V_{i-1,j}^n$ | $V_{i,j}^{n+1}$ | $V_{i+1,j}^n$ |
| | $V_{i,j-1}^n$ | |

$$\partial_t V = [\partial_x^2 + \partial_y^2] V \Rightarrow V_{i,j}^{n+1} = V_{i,j}^n + \frac{\Delta t}{h^2} [V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n - 4V_{i,j}^n]$$

Si elijo $\Delta t/h^2 = 1/4 \rightarrow V_{i,j}^{n+1} = V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n$ me queda el método de Jacobi

Algoritmo

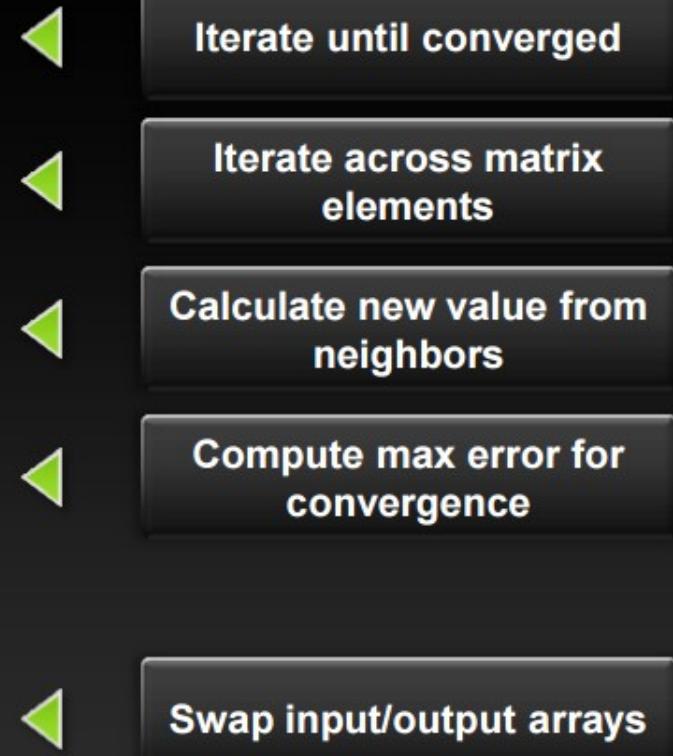
$$V_{i,j}^{n+1} = \frac{1}{4}(V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n)$$

| Current Array | | | | | | | | Next Array | | | | | | | |
|---------------|-----|-----|-----|-----|------|------|-----|------------|-----|-----|-----|-----|-----|-----|-----|
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 2.0 | 4.0 | 6.0 | 8.0 | 10.0 | 12.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 3.0 | 5.0 | 7.0 | 9.0 | 11.0 | 13.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 2.0 | 6.0 | 1.0 | 3.0 | 7.0 | 5.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

Programa serial

Jacobi Iteration C Code

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
  
    for( int j = 1; j < n-1; j++ ) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            error = max(error, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
    for( int j = 1; j < n-1; j++ ) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```





Programa serial

Jacobi Iteration Fortran Code

```
do while ( err > tol .and. iter < iter_max )  
  err=0._fp_kind
```

Iterate until converged

```
  do j=1,m  
    do i=1,n
```

Iterate across matrix elements

```
    Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &  
                                A(i , j-1) + A(i , j+1))
```

Calculate new value from neighbors

```
    err = max(err, Anew(i,j) - A(i,j))  
  end do  
end do
```

Compute max error for convergence

```
do j=1,m-2  
  do i=1,n-2  
    A(i,j) = Anew(i,j)  
  end do  
end do
```

Swap input/output arrays

```
  iter = iter +1  
end do
```

Manos a la obra...



Manos a la obra...

- `module load nvhpc-21.9`
- `cp -a /share/apps/icnpg/clases/clases_openacc .` La ejecución en el cluster no se hace con gpushort porque las placas asignadas son viejas y necesitamos las más nuevas donde esté instalado OpenAcc
- `cd clases_openacc/jacobi/cudacast`
- Version serial: `laplace2d.c`
 - **CPU-seq:** `g++ laplace2d_serial.c -o serial.out`
- Versiones OPENACC (todas con `make`):
 - **GPU:** `pgcc -I. -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c`
 - **CPU-seq:** `pgcc laplace2d.c -o laplace2d -I.`
 - **CPU-paralela:** `pgcc -I. -acc -ta=multicore -Minfo=accel -o laplace2d_acc_multicore laplace2d.c`
- Version OPENMP
 - **OMP:** `pgcc -I. -fast -mp -Minfo -o laplace2d_omp laplace2d.c`
- Correr:
 - `qsub JobGPU ./ejecutable`
- Analizar output: *¿Donde se consume casi todo el tiempo de la corrida?*

OPENMP → paralelización CPU multicore

OpenMP C Code

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
  
#pragma omp parallel for shared(m, n, Anew, A)  
    for( int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            error = max(error, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
#pragma omp parallel for shared(m, n, Anew, A)  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

OpenMP Fortran Code

```
do while ( err > tol .and. iter < iter_max )  
    err=0._fp_kind  
  
!$omp parallel do shared(m,n,Anew,A) reduction(max:err)  
    do j=1,m  
        do i=1,n  
  
            Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &  
                                         A(i , j-1) + A(i , j+1))  
  
            err = max(err, Anew(i,j) - A(i,j))  
        end do  
    end do  
  
!$omp parallel do shared(m,n,Anew,A)  
    do j=1,m-2  
        do i=1,n-2  
            A(i,j) = Anew(i,j)  
        end do  
    end do  
  
    iter = iter +1  
end do
```

- Directivas para indicar paralelización de loops en hilos de la CPU

OPENMP vs OPENACC

mismo código, distinta compilación

C

```
...
#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] +
                               A[j][i-1] + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error,
                       fabsf(Anew[j][i]-A[j][i])
                     );
    }
}
...
```

Fortran

```
...
 !$omp do reduction( max:error )
 !$acc kernels
 do j=1,m-2
     do i=1,n-2
         Anew(i,j) = 0.25_fp_kind * ( A(i+1,j) + A(i-1,j) + &
                                         A(i,j-1) + A(i,j+1) )
         error = max( error, abs(Anew(i,j)-A(i,j)) )
     end do
 end do
 !$acc end kernels
 !$omp end do
 ...

```



- **Serial en f90:** pgcc -I. -fast -o laplace2d_omp laplace2d.c
- **Openacc en f90:** pgf90 -acc -ta=nvidia -Minfo=accel -o laplace2d_f90_acc laplace2d.f90
- **Openmp en f90:** pgf90 -fast -mp -Minfo -o laplace2d_f90_omp laplace2d.f90
- **Multicore f90:** pgcc -I. -acc -ta=multicore -Minfo=accel -o laplace2d_acc_multicore laplace2d.c

Paso 1

- make laplace2d_omp

```
pgcc -I../common -fast -mp -Minfo -o laplace2d_omp laplace2d.c  
main:
```

53, Loop not fused: dependence chain to sibling loop
Generated an alternate version of the loop
Generated vector sse code for the loop
59, Loop not fused: function call before adjacent loop
Generated vector sse code for the loop

72, Parallel region activated
80, Parallel loop activated with static block schedule
Generated an alternate version of the loop
Generated vector sse code for the loop

85, Barrier
87, Parallel region terminated

88, Parallel region activated
Parallel loop activated with static block schedule
Generated vector sse code for the loop

94, Barrier
Parallel region terminated

100, Parallel region activated
Parallel loop activated with static block schedule
102, Generated an alternate version of the loop
Generated vector sse code for the loop

110, Begin critical section
End critical section
Barrier
Parallel region terminated

112, Parallel region activated
Parallel loop activated with static block schedule
114, Memory copy idiom, loop replaced by call to __c_mcopy4
120, Barrier
Parallel region terminated

- make laplace2d_acc

```
pgcc -I../common -acc -ta=nvidia,time -Minfo=accel -o laplace2d_acc  
laplace2d.c  
main:
```

99, Generating copyout(Anew[1:4094][1:4094])
Generating copyin(A[:4096][:4096])
100, Loop is parallelizable
102, Loop is parallelizable

```
#pragma omp parallel shared(Anew)  
{  
    int tid = omp_get_thread_num();  
    if (tid == 0)  
    {  
        int nthreads = omp_get_num_threads();  
        printf("Number of threads = %d\n", nthreads); /* */  
    }  
}  
#pragma omp parallel for shared(Anew)  
{  
    for (int j = 1; j < n; j++)  
    {  
        Anew[j] = #pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)  
        Anew[j] = #pragma acc kernels  
        {  
            for( int j = 1; j < n-1; j++)  
            {  
                for( int i = 1; i < m-1; i++ )  
                {  
                    Anew[j][i] = Anew[j][i] * ( 1.0f + A[j][i-1]  
                        + A[j+1][i]);  
                    [j][i]-A[j][i]));  
                }  
            }  
        }  
    }  
}  
#pragma omp parallel for shared(m, n, Anew, A)  
#pragma acc kernels  
{  
    for( int j = 1; j < n-1; j++)  
    {  
        for( int i = 1; i < m-1; i++ )  
        {  
            A[j][i] = Anew[j][i];  
        }  
    }  
}  
if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);  
iter++;
```

Paso 1

- make laplace2d_omp

```
pgcc -I.../common -fast -mp -Minfo -o laplace2d_omp laplace2d.c
main:
53,     #pragma acc kernels
54,         for( int j = 1; j < n-1; j++)
55,             {
56,                 for( int i = 1; i < m-1; i++ )
57,                     {
58,                         Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
59,                                                 + A[j-1][i] + A[j+1][i]);
60,                         error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
61,                     }
62,             }
63,             Barrier
64,             Parallel region terminated
65,             #pragma acc kernels
66,                 for( int j = 1; j < n-1; j++)
67,                     {
68,                         for( int i = 1; i < m-1; i++ )
69,                             {
70,                                 A[j][i] = Anew[j][i];
71,                             }
72,                     }
73,                     Barrier
74,                     Parallel region terminated
75,                     #pragma acc kernels
76,                         for( int j = 1; j < n-1; j++)
77,                             {
78,                                 for( int i = 1; i < m-1; i++ )
79,                                     {
80,                                         error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
81,                                     }
82,                             }
83,                             Barrier
84,                             Parallel region terminated
85,                             Barrier
86,                             Parallel region terminated
87,                             Parallel region terminated
88,                             Parallel region activated
89,                             Parallel loop activated
90,                             Generated vector sse code for the loop
91,                             Generated vector sse code for the loop
92,                             Generated vector sse code for the loop
93,                             Generated vector sse code for the loop
94,                             Barrier
95,                             Parallel region terminated
96,                             Parallel region terminated
97,                             Parallel region terminated
98,                             Parallel region terminated
99,                             Parallel region terminated
100,                            Parallel region activated
101,                            Parallel loop activated with static block schedule
102,                            Generated an alternate version of the loop
103,                            Generated vector sse code for the loop
104,                            Generated vector sse code for the loop
105,                            Generated vector sse code for the loop
106,                            Generated vector sse code for the loop
107,                            Generated vector sse code for the loop
108,                            Generated vector sse code for the loop
109,                            Generated vector sse code for the loop
110,                            Begin critical section
111,                            End critical section
112,                            Barrier
113,                            Parallel region terminated
114,                            Parallel region activated
115,                            Parallel loop activated with static block schedule
116,                            Memory copy idiom, loop replaced by call to __c_mcopy4
117,                            Barrier
118,                            Parallel region terminated
```

- make laplace2d_acc

```
pgcc -I.../common -acc -ta=nvidia,time -Minfo=accel -o laplace2d_acc
laplace2d.c
main:
99, Generating copyout(Anew[1:4094][1:4094])
100, Generating copyin(A[1:4096][1:4096])
101, Loop is parallelizable
102, Loop is parallelizable
103, Accelerator kernel generated
104, Generating Tesla code
105, 100, #pragma acc loop gang /* blockIdx.y */
106, 102, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
107, 106, Max reduction generated for error
108, 111, Generating copyin(Anew[1:4094][1:4094])
109, 112, Generating copyout(A[1:4094][1:4094])
110, 112, Loop is parallelizable
111, 114, Loop is parallelizable
112, 114, Accelerator kernel generated
113, 115, Generating Tesla code
114, 112, 112, #pragma acc loop gang /* blockIdx.y */
115, 114, 114, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Paso 1



Ayúdenme a
completar
ésta tabla

| Ejecución | Tiempo | Aceleración respecto al serial |
|------------------|--------|--------------------------------|
| Serial | | |
| Openmp 2 threads | | |
| Openmp 3 threads | | |
| Openmp 4 threads | | |
| OpenACC | | |

¿Qué paso?

nvprof ./laplace2d_acc

```
==7093== Profiling application: ./laplace2d_acc
```

```
==7093== Profiling result:
```

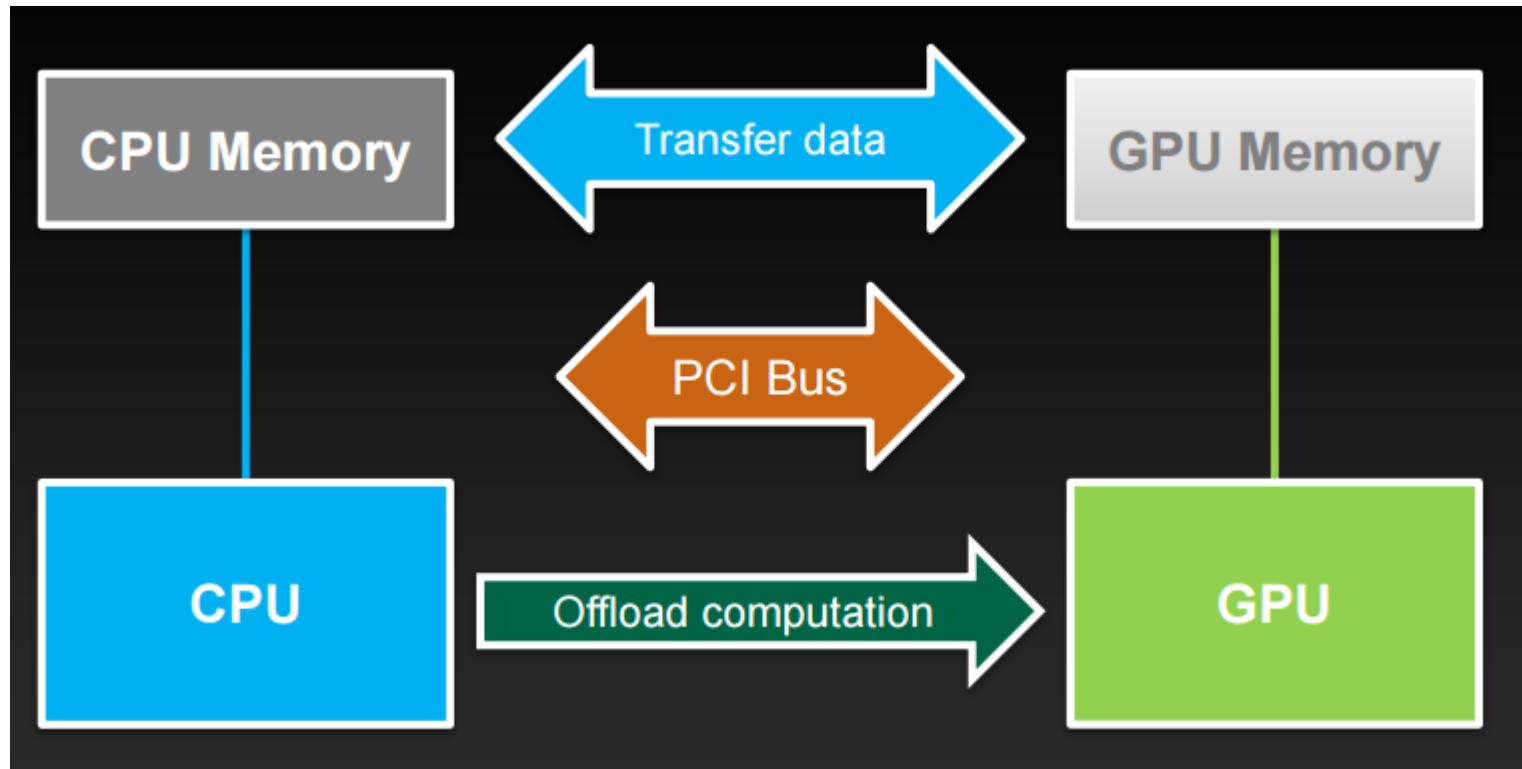
| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|----------|----------|--------------------|
| 42.54% | 24.2059s | 9000 | 2.6895ms | 3.4870us | 3.8239ms | [CUDA memcpy DtoH] |
| 41.52% | 23.6242s | 9000 | 2.6249ms | 959ns | 3.5642ms | [CUDA memcpy HtoD] |
| 10.27% | 5.84493s | 1000 | 5.8449ms | 5.8436ms | 5.8544ms | main_102_gpu |
| 5.20% | 2.95895s | 1000 | 2.9590ms | 2.9559ms | 2.9621ms | main_114_gpu |
| 0.46% | 263.71ms | 1000 | 263.71us | 262.65us | 265.52us | main_106_gpu_red |

```
==7093== API calls:
```

| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|-------|----------|--------------------|
| 98.05% | 57.1905s | 44996 | 1.2710ms | 404ns | 5.8903ms | cuEventSynchronize |

aaaa Excesivo movimiento de datos !!!!!

El precio de transferir...



CUDA Application Design and Development, R. Farber

THREE RULES OF GPGPU PROGRAMMING

Observation has shown that there are three general rules to creating high-performance GPGPU programs:

1. Get the data on the GPGPU and keep it there.
2. Give the GPGPU enough work to do.
3. Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

Básicamente es entender las latencias...

¿Compilador precavido o tonto?

¿Necesitamos **A** y **Anew** en el host cada iteración de Jacobi?

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;
```

A, Anew resident on host

Copy

#pragma acc kernels

A, Anew resident on accelerator

These copies happen
every iteration of the
outer while loop!*

```
for( int j = 1; j < n-1; j++ ) {  
    for(int i = 1; i < m-1; i++) {  
        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                             A[j-1][i] + A[j+1][i]);  
        error = max(error, abs(Anew[j][i] - A[j][i]));  
    }  
}
```

A, Anew resident on host

Copy

A, Anew resident on accelerator

...

¿Compilador precavido o tonto?

¿Necesitamos **A** y **Anew** en el host cada iteración de Jacobi?

```
while ( error > tol && iter < iter_max )
{
    error = 0.f;
    #pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
    #pragma acc kernels
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
            error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
        }
    }
    #pragma omp parallel for shared(m, n, Anew, A)
    #pragma acc kernels
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }
    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
}
```

¿Que haría Ud aquí si fuera el Compilador?

¿Compilador precavido o tonto?

¿Necesitamos **A** y **Anew** en el host cada iteración de Jacobi?

```
do while ( error .gt. tol .and. iter .lt. iter_max )
    error=0.0_fp_kind

 !$omp parallel do shared(m, n, Anew, A) reduction( max:error )
!$acc kernels
    do j=1,m-2
        do i=1,n-2
            Anew(i,j) = 0.25_fp_kind * ( A(i+1,j     ) + A(i-1,j     ) + &
                                         A(i     ,j-1) + A(i     ,j+1) )
            error = max( error, abs(Anew(i,j)-A(i,j)) )
        end do
    end do
!$acc end kernels
 !$omp end parallel do

    if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
    iter = iter +1

 !$omp parallel do shared(m, n, Anew, A)
!$acc kernels
    do j=1,m-2
        do i=1,n-2
            A(i,j) = Anew(i,j)
        end do
    end do
!$acc end kernels
 !$omp end parallel do

end do
```

¿Compilador precavido o tonto?

¿Necesitamos **A** y **Anew** en el host cada iteración de Jacobi?

```
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
#pragma acc kernels
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
            error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
        }
    }

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

    iter++;
}
```

Che, compilador!,
NO me traigas
los datos al host
aquí porque no
los voy a necesitar
aún!

¿Que haría Ud aquí si
fuera el Compilador?

Explotar localidad de Datos

Defining data regions



- The **data** construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
!$acc data
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do

  do i=1,n
    a(i) = b(i) + c(i)
  end do
!$acc end data
```



Data Region

Arrays a, b, and c will remain on the GPU until the end of the data region.

Localidad de los datos

ANTES COPIABAMOS 4 VECES CADA ITERACION

```
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma omp parallel for shared(m, n, Anew, A)
reduction(max:error)
```

Aquí el compilador copia A, Anew de host a device

```
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}
```

Aquí el compilador, por las dudas, actualiza A, Anew en el host

```
#pragma omp parallel for shared(m, n, Anew, A)
```

Aquí el compilador copia A, Anew de host a device

```
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}
```

Aquí el compilador, por las dudas, actualiza A, Anew en el host

```
if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

iter++;
```

AHORA NOS AHORRAMOS UN MONTON DE COPIAS

Aquí el compilador copia A, Anew al device

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}
#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

iter++;
```

Recién aquí el compilador actualiza A, Anew en el host como implica el “copy”

Data Regions

Data Construct



Fortran

```
!$acc data [clause ...]  
    structured block  
!$acc end data
```

C

```
#pragma acc data [clause ...]  
    { structured block }
```

General Clauses

```
if( condition )  
async( expression )
```

Manage data movement. Data regions may be nested.

Data Regions

Data Clauses



- `copy (list)` Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
 - `copyin (list)` Allocates memory on GPU and copies data from host to GPU when entering region.
 - `copyout (list)` Allocates memory on GPU and copies data to the host when exiting region.
 - `create (list)` Allocates memory on GPU but does not copy.
 - `present (list)` Data is already present on GPU from another containing data region.
- and `present_or_copy[in|out], present_or_create, deviceptr.`

Data Regions

Array Shaping



- Compiler sometimes cannot determine size of arrays
 - Must specify explicitly using data clauses and array “shape”
- C

```
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```
- Fortran

```
!$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```
- Note: data clauses can be used on data, kernels or parallel

Data Regions

Update Construct



Fortran

```
!$acc update [clause ...]
```

Clauses

host(list)

device(list)

C

```
#pragma acc update [clause ...]
```

if(expression)
async(expression)

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.
Data movement can be conditional, and asynchronous.

¿Compilador precavido y/o tonto ?

ANTES

```
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma omp parallel for shared(m, n, Anew, A)
reduction(max:error)
```

Aquí el compilador copia A, Anew de host a device

```
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                               + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}
```

Aquí el compilador, por las dudas, actualiza A, Anew en el host

```
#pragma omp parallel for shared(m, n, Anew, A)
```

Aquí el compilador copia A, Anew de host a device

```
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}
```

Aquí el compilador, por las dudas, actualiza A, Anew en el host

```
if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
iter++;
```

Aquí el compilador copia A, Anew al device

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                               + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}
#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
iter++;
```

Recién aquí el compilador actualiza A, Anew en el host como implica el “copy”

Paso 2

- Agregar donde corresponda:
 - **#pragma acc data copy(A), create(Anew)**

Paso 2



Ayudenme de
nuevo a
completar
esta tabla

| Ejecución | Tiempo | Aceleración respecto al serial |
|------------------|--------|--------------------------------|
| Serial | | |
| Openmp 2 threads | | |
| Openmp 3 threads | | |
| Openmp 4 threads | | |
| OpenACC | | |



~15 X

¿Qué paso?

Antes [~150 s, 0.19 x]

```
==7093== Profiling application: ./laplace2d_acc
```

```
==7093== Profiling result:
```

| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|----------|----------|--------------------|
| 42.54% | 24.2059s | 9000 | 2.6895ms | 3.4870us | 3.8239ms | [CUDA memcpy DtoH] |
| 41.52% | 23.6242s | 9000 | 2.6249ms | 959ns | 3.5642ms | [CUDA memcpy HtoD] |
| 10.27% | 5.84493s | 1000 | 5.8449ms | 5.8436ms | 5.8544ms | main_102_gpu |
| 5.20% | 2.95895s | 1000 | 2.9590ms | 2.9559ms | 2.9621ms | main_114_gpu |
| 0.46% | 263.71ms | 1000 | 263.71us | 262.65us | 265.52us | main_106_gpu_red |

```
==7093== API calls:
```

| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|-------|----------|--------------------|
| 98.05% | 57.1905s | 44996 | 1.2710ms | 404ns | 5.8903ms | cuEventSynchronize |

Ahora [~4 s, 15 x]

```
==6200== Profiling application: ./laplace2d_acc
```

```
==6200== Profiling result:
```

| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|----------|----------|--------------------|
| 61.57% | 2.36598s | 1000 | 2.3660ms | 2.3647ms | 2.3704ms | main_94_gpu |
| 29.62% | 1.13818s | 1000 | 1.1382ms | 1.1365ms | 1.1402ms | main_106_gpu |
| 7.54% | 289.84ms | 1000 | 289.84us | 289.09us | 290.98us | main_98_gpu_red |
| 0.70% | 26.767ms | 1008 | 26.554us | 896ns | 3.9257ms | [CUDA memcpy HtoD] |
| 0.58% | 22.279ms | 1010 | 22.058us | 2.0800us | 2.5660ms | [CUDA memcpy DtoH] |

```
==6200== API calls:
```

| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---------|----------|-------|----------|-------|----------|--------------------|
| 90.78% | 3.86609s | 5042 | 766.78us | 950ns | 3.9326ms | cuEventSynchronize |

Expresar paralelismo +
Explotar localidad de datos =



~15 X

Consejo: 4 pasos

- **Identificar el Paralelismo:**

Mirar, donde el programa pierde mas tiempo, si hay posibilidad de paralelización (loops, loops anidados, etc)

- **Expresar el Paralelismo:**

Poner directivas en los loops para avisarle al compilador donde puede parallelizar. Generalmente da lugar a un código más lento (paso 2)... pero ¡a no desalentarse!.

- **Expresar localidad de datos:**

El compilador no es tan astuto pero es precavido, no puede tomar ciertas decisiones por uno. Decirle entonces como y cuando los datos son necesarios en la GPU o CPU (paso 2).

- **Optimizar:**

Una ayudita extra para mejorar la performance (paso 3), reestructurar código o mejorar los patrones de acceso o usar info del hardware... Si no esta satisfecho con las posibilidades que brinda openacc programe la parte crítica en CUDA!, y sacrifique un poco la portabilidad...

Openacc API

Directive Syntax

- Fortran

`!$acc directive [clause [,] clause] ...]`

Often paired with a matching end directive surrounding a structured code block

`!$acc end directive`

- C

`#pragma acc directive [clause [,] clause] ...]`

Often followed by a structured code block

Openacc API

Kernels Construct

Fortran

```
!$acc kernels [clause ...]
  structured block
 !$acc end kernels
```

Clauses

```
if( condition )
async( expression )
```

Also any data clause

C

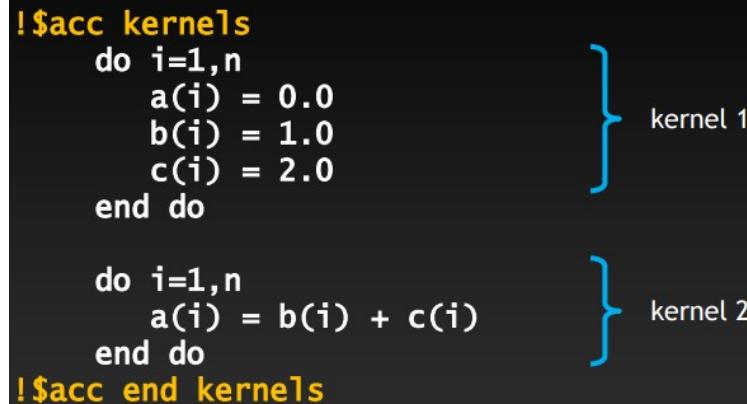
```
#pragma acc kernels [clause ...]
{ structured block }
```

Kernels Construct

Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do

  do i=1,n
    a(i) = b(i) + c(i)
  end do
 !$acc end kernels
```



```
#pragma acc data copy(A, Anew)
  while ( error > tol && iter < iter_max )
  {
    error = 0.f;

# pragma acc kernels
  for( int j = 1; j < n-1; j++)
  {
    for( int i = 1; i < m-1; i++ )
    {
      Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                               + A[j-1][i] + A[j+1][i]);
      error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
  }

# pragma acc kernels
  for( int j = 1; j < n-1; j++)
  {
    for( int i = 1; i < m-1; i++ )
    {
      A[j][i] = Anew[j][i];
    }
    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter,
error);

    iter++;
  }
```

Openacc API

Loop Construct

Fortran

```
!$acc loop [clause ...]
    loop
!$acc end loop
```

C

```
#pragma acc loop [clause ...]
{ loop }
```

Combined directives

```
!$acc parallel loop [clause ...]      !$acc parallel loop [clause ...]
!$acc kernels loop [clause ...]       !$acc kernels loop [clause ...]
```

Detailed control of the parallel execution of the following loop.

Loop Clauses Inside parallel Region



gang

Shares iterations across the gangs of the parallel region.

worker

Shares iterations across the workers of the gang.

vector

Execute the iterations in SIMD mode.

Loop Clauses



collapse(n)

Applies directive to the following *n* nested loops.

seq

Executes the loop sequentially on the GPU.

private(list)

A copy of each variable in list is created for each iteration of the loop.

reduction(operator:list)

private variables combined across iterations.

```
#pragma acc data copy(A), create(Anew)
    while ( error > tol && iter < iter_max )
    {
        error = 0.f;

# pragma acc parallel loop collapse(2) reduction(max:error)
        for( int j = 1; j < n-1; j++ )
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                         + A[j-1][i] + A[j+1][i] );
                error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
            }
        }

# pragma acc parallel loop collapse(2)
        for( int j = 1; j < n-1; j++ )
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
```

¿Parallel o Kernel?

- Parallel

- Prescriptivo (como openmp)
- Usa un kernel en la región
- El compilador va a acelerar la región *aun si da resultados incorrectos*
- *Mayor control*

- Kernels

- Descriptivo
- Usa uno o mas kernels en la región
- *Más seguro:* podría no acelerar nada si hay ambiguedad
- *Mejor para empezar a explorar paralelismo*

Ambos definen regiones a ser aceleradas, pero la obligación al compilador es diferente

```
!$acc kernels
    do i=1,n
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
    end do
}
do i=1,n
    a(i) = b(i) + c(i)
end do
 !$acc end kernels
```

The compiler
identifies 2 parallel
loops and generates
2 kernels.

Se trata de agregar #pragma ...

| I\$acc kernels | I\$acc parallel | I\$acc data | I\$acc loop | I\$acc wait |
|----------------------|----------------------|----------------------|-----------------------|------------------|
| #pragma acc kernels | #pragma acc parallel | #pragma acc data | #pragma acc loop | #pragma acc wait |
| Clauses | Clauses | Clauses | Clauses | |
| if() | if() | if() | collapse() | |
| async() | async() | async() | within kernels region | |
| copy() | num_gangs() | | gang() | |
| copyin() | num_workers() | | worker() | |
| copyout() | vector_length() | | vector() | |
| create() | reduction() | | seq() | |
| present() | copyin() | copyin() | private() | |
| present_or_copy() | copyout() | copyout() | reduction() | |
| present_or_copyin() | create() | create() | | |
| present_or_copyout() | present() | present() | | |
| present_or_create() | present_or_copy() | deviceptr() in .c | | |
| deviceptr() | present_or_copyin() | deviceptr() in .f | | |
| | present_or_copyout() | | | |
| | present_or_create() | | | |
| | deviceptr() | | | |
| | private() | | | |
| | firstprivate() | | | |

```

#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma acc parallel loop collapse(2) reduction(max:error)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}

#pragma acc parallel loop collapse(2)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

iter++;
}

```

Referencias

<http://www.openacc.org/content/education>

OpenACC Directives for Accelerators

- [OpenACC: Directives for GPUs](#)
- [FREE OpenACC Online Course](#)
- [Six Ways to SAXPY](#)
- [An OpenACC Example \(Part I, Part II\)](#)

OpenACC using the PGI Compilers

- [PGI Accelerator Compilers with OpenACC Getting Started Guide \(PDF\)](#)
- [OpenACC Kernels and Parallel Constructs](#)
- [The PGI Accelerator Compilers with OpenACC](#)
- [5X in 5 Hours: Porting a 3D Elastic Wave Simulator using OpenACC](#)
- [Understanding the CUDA Data Parallel Threading Model](#)

OpenACC using the Cray Compilers

- [A Quick Tour of OpenACC](#)
- [OpenACC and the Cray Compilation Environment](#)

Developer Support Forums

- [Stackoverflow](#)
- [PGI](#)

Seminars & Classes

Europe

- [CINECA SCAI](#)
- [HLRS](#)
- [NVIDIA Europe](#)
- [PRACE](#)

North America

- [Univ. of Houston](#)
- [XSEDE](#)

To add a seminar, email training@openacc.org

Videos

- [Introduction to OpenACC](#)
- [Unstructured Data Lifetimes in OpenACC 2.0 \(CUDAcast\)](#)
- [C++ Class Management with OpenACC 2.0](#)
- [Enabling OpenACC Performance Analysis](#)

Learn More

- [Debuggers and Other Tools](#)
- [Interoperability with MPI](#)

Consulting & Training Services

- [Acceleware](#)
- [Applied Parallel Computing](#)
- [SagivTech](#)

OpenACC Programming and Best Practices Guide

¿Qué pasa si dentro del loop llamo a una función que está definida fuera del main?

June 2015

3 Parallelize Loops

| |
|--|
| The Kernels Construct |
| The Parallel Construct |
| Differences Between Parallel and Kernels |
| The Loop Construct |
| Routine Directive |
| Case Study - Parallelize |
| Atomic Operations |

4 Optimize Data Locality

| |
|---|
| Data Regions |
| Data Clauses |
| Unstructured Data Lifetimes |
| Update Directive |
| Best Practice: Offload Inefficient Operations to Maintain Data Locality |
| Case Study - Optimize Data Locality |

Ejemplo de datos con tiempo de vida estructurado.

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max )
{
    error = 0.f;

#pragma acc parallel loop collapse(2) reduction(max:error)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
        error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
    }
}

#pragma acc parallel loop collapse(2)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

iter++;
}
```

A, Anew viven aqui arriba

Datos con “tiempo de vida desestructurado”

```
void jacobi(float A[][4096],float Anew[][4096])
{
    int n = 4096;
    int m = 4096;

    int iter = 0;
    const float tol = 1.0e-5f;
    float error = 1.0f;
    int iter_max = 1000;

    while ( error > tol && iter < iter_max )
    {
        error = 0.f;
#pragma acc parallel loop collapse(2) reduction(max:error)

```

- enter data
- exit data

```
int main(int argc, char** argv)
{
    // declaracion, inicializacion de A, Anew en el host
    // etc
#pragma acc enter data copyin(A[0:n][0:m],Anew[0:n][0:m])
    jacobi(A,Anew);
    otra_funcion_acelerada(A,Anew);
#pragma acc exit data delete(A[0:n][0:m],Anew[0:n][0:m])
    // declaracion, inicializacion de A, Anew en el
host
    // etc
}
```

OpenACC Programming and Best Practices Guide

June 2015

3 Parallelize Loops

| |
|--|
| The Kernels Construct |
| The Parallel Construct |
| Differences Between Parallel and Kernels |
| The Loop Construct |
| Routine Directive |
| Case Study - Parallelize |
| Atomic Operations |

4 Optimize Data Locality

| |
|---|
| Data Regions |
| Data Clauses |
| Unstructured Data Lifetimes |
| Update Directive |
| Best Practice: Offload Inefficient Operations to Maintain Data Locality |
| Case Study - Optimize Data Locality |

UPDATE DIRECTIVES:

Sincronizan los contenidos de las memorias que uno quiera, cuando uno quiera

```
int main(int argc, char** argv)
{
    // declaracion, inicializacion de A, Anew en el host
    // etc
    #pragma acc enter data copyin(A,Anew)

    jacobi(A,Anew);

    funcion_acelerada(A,Anew);

    #pragma acc update self(A)
    funcion_no_acelerada(A);
    #pragma acc update device(A)

    otra_funcion_acelerada(A,Anew);

    #pragma acc exit data delete(A[0:n]
    [0:m],Anew[0:n][0:m])
    // declaracion, inicializacion de A, Anew en el host
    // etc
}
```

Best Practice: variables in an OpenACC code should always be thought of as a singular object, rather than a host copy and a device copy

OPENACC y Template Classes

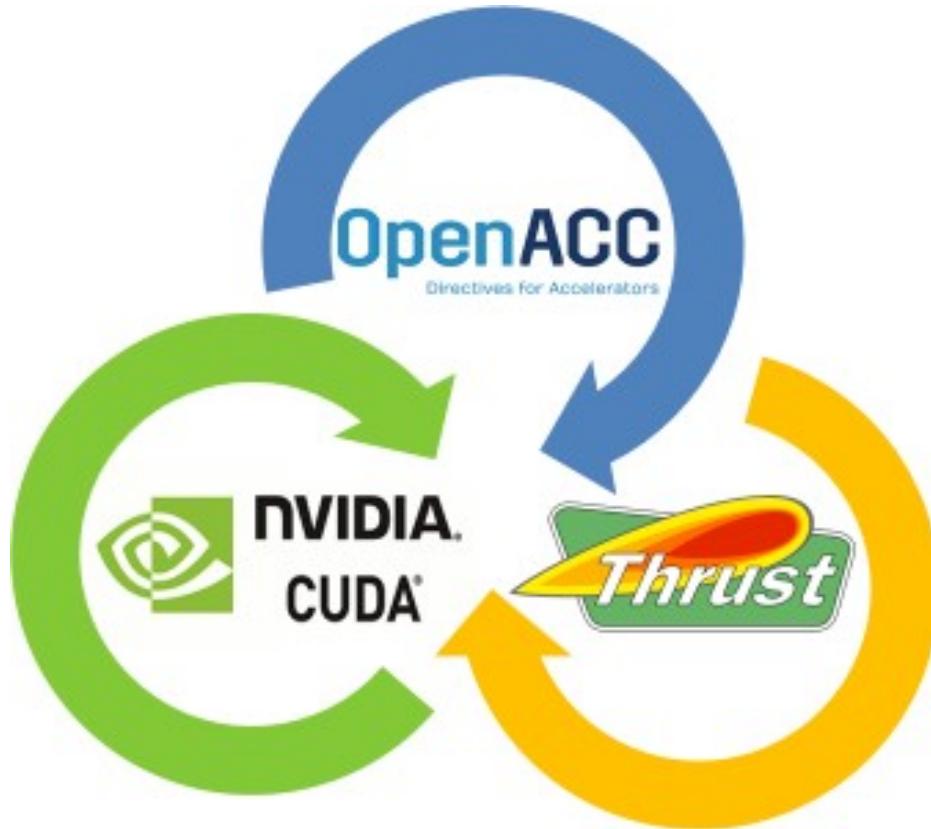
C++

```
template<typename vtype> class myvector
{
    vtype* mydata;
    size_t mysize;
public:
    inline vtype & operator[]( int i ) const {return mydata[i];}
    inline size_t size(){ return mysize; }
    myvector( int size_ ){
        mysize = size_;
        mydata = new vtype[mysize];
        #pragma acc enter data copyin(this)
        #pragma acc enter data create(mydata[0:mysize])
    }
    ~myvector(){
        #pragma acc exit data delete(mydata[0:mysize],this)
        delete [] mydata;
    }
    void updatedev(){
        #pragma acc update device(mydata[0:mysize])
    }
    void updatehost(){
        #pragma acc update host(mydata[0:mysize])
    }
};
```

```
template<typename vtype> void
    axpy( myvector<vtype>& y, myvector<vtype>& x,
vtype a ){
    #pragma acc parallel loop present(x,y)
    for( int i = 0; i < y.size; ++i )
        y[i] += a*x[i];
}

template<typename vtype> void
    test( myvector<vtype>& a, myvector<vtype>& b ){
    a.updatedev();
    b.updatedev();
    axpy<vtype>( a, b, 2.0 );
    modify( b );
    b.updatedev();
    axpy<vtype>( a, b, 1.0 );
    a.updatehost();
}
```

Interoperabilidad



- `host_data` construct
- `deviceptr` clause
- `acc_map_data()` API function.

Usar CUDA (+libs) en un programa OpenACC con *host_data region*

```
...
#pragma acc data create(x[0:n]) copyout(y[0:n])
{
    #pragma acc kernels
    {
        for( i = 0; i < n; i++){
            x[i] = 1.0f;
            y[i] = 0.0f;
        }
    }
    #pragma acc host_data use_device(x,y)
    {
        cublasSaxpy(n, 2.0, x, 1, y, 1);
        // kernel<<<...>>>(x,y,n);
        //
        thrust::transform(thrust::device,x,x+n,y,y);
        // etc...
    }
}
...
```

```
program main
    use cublas
    integer, parameter :: N = 2**20
    real, dimension(N) :: X, Y

    !$acc data create(x) copyout(y)

    !$acc kernels
    X(:) = 1.0
    Y(:) = 0.0
    !$acc end kernels

    !$acc host_data use_device(x,y)
    call cublassaxpy(N, 2.0, x, 1, y, 1)
    !$acc end host_data

    !$acc end data

    print *, y(1)
end program
```

The host_data region gives the programmer a way to expose the device address of a given array to the host for passing into a function. This data **must have already been moved** to the device previously. The host_data region accepts only the use_device clause, which specifies which device variables should be exposed to the host.

Usar OPENACC en un programa de CUDA con *deviceptr*

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
    #pragma acc kernels deviceptr(x,y)
    {
        for(int i=0; i<n; i++)
        {
            y[i] += a*x[i];
        }
    }
}

void set(int n, float val, float * restrict arr)
{
    #pragma acc kernels deviceptr(arr)
    {
        for(int i=0; i<n; i++)
        {
            arr[i] = val;
        }
    }
}

int main(int argc, char **argv)
{
    float *x, *y, tmp;
    int n = 1<<20;

    cudaMalloc((void**)&x,(size_t)n*sizeof(float));
    cudaMalloc((void**)&y,(size_t)n*sizeof(float));

    set(n,1.0f,x);
    set(n,0.0f,y);

    saxpy(n, 2.0, x, y);
    cudaMemcpy(&tmp,y,
    (size_t)sizeof(float),cudaMemcpyDeviceToHost);
    printf("%f\n",tmp);
    return 0;
}
```

El array ya esta en device, tomo su puntero...

Usa CUDA API
(podria usar openACC API o thrust)

In this case OpenACC provides the `deviceptr` data clause, which may be used where any data clause may appear. This clause informs the compiler that the variables specified are already device on the device and no other action needs to be taken on them.

[http://www.openacc.org/sites/default/files/
OpenACC_Programming_Guide_0.pdf](http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf)

Usar OPENACC en un programa de CUDA con *deviceptr*

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
    #pragma acc kernels deviceptr(x,y)
    {
        for(int i=0; i<n; i++)
        {
            y[i] += a*x[i];
        }
    }
}

void set(int n, float val, float * restrict arr)
{
    #pragma acc kernels deviceptr(arr)
    {
        for(int i=0; i<n; i++)
        {
            arr[i] = val;
        }
    }
}
```

El array ya esta en device, tomo su puntero...

```
int main(int argc, char **argv)
{
    float *x, *y, tmp;
    int n = 1<<20;

    x = acc_malloc((size_t)n*sizeof(float));
    y = acc_malloc((size_t)n*sizeof(float));

    set(n,1.0f,x);
    set(n,0.0f,y);

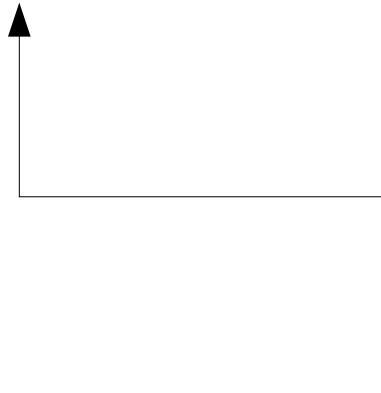
    saxpy(n, 2.0, x, y);
    acc_memcpy_from_device(&tmp,y,(size_t)sizeof(float));
    printf("%f\n",tmp);
    acc_free(x);
    acc_free(y);
    return 0;
}
```

Usa Openacc API
(podria usar CUDA API
o thrust ...)

In this case OpenACC provides the deviceptr data clause, which may be used where any data clause may appear. This clause informs the compiler that the variables specified are already device on the device and no other action needs to be taken on them.

Usar OPENACC en un programa de Thrust con *deviceptr*

```
void saxpy(int n, float a, float *  
restrict x, float * restrict y)  
{  
    #pragma acc kernels deviceptr(x,y)  
    {  
        for(int i=0; i<n; i++)  
        {  
            y[i] += a*x[i];  
        }  
    }  
}
```



```
int main(int argc, char **argv)  
{  
    int N = 1<<20;  
    thrust::host_vector y(N);  
  
    thrust::device_vector d_x(N);  
    thrust::device_vector d_y(N);  
  
    thrust::fill(d_x.begin(),d_x.end(), 1.0f);  
    thrust::fill(d_y.begin(),d_y.end(), 0.0f);  
  
    saxpy(N, 2.0,  
          thrust::raw_pointer_cast(d_x.data()),  
          thrust::raw_pointer_cast(d_y.data()));  
  
    y = d_y;  
    printf("%f\n",y[0]);  
    return 0;  
}
```

Usar OPENACC en un programa de CUDA con *acc_map_data()*

```
void map(float * restrict harr, float * restrict darr, int size)
{
    acc_map_data(harr, darr, size);
}

void saxpy(int n, float a, float * restrict x, float * restrict y)
{
    // x , y estan presentes en device
    // porque llamamos a acc_map_data()
    #pragma acc kernels present(x,y)
    {
        for(int i=0; i<n; i++)
        {
            y[i] += a*x[i];
        }
    }
}

void set(int n, float val, float * restrict arr)
{
    #pragma acc kernels present(x,y)
    {
        for(int i=0; i<n; i++)
        {
            arr[i] = val;
        }
    }
}
```

```
int main(int argc, char **argv)
{
    float *x, *y, *dx, *dy, tmp;
    int n = 1<<20;

    x = (float*) malloc(n*sizeof(float));
    y = (float*) malloc(n*sizeof(float));
    cudaMalloc((void**)&dx,(size_t)n*sizeof(float));
    cudaMalloc((void**)&dy,(size_t)n*sizeof(float));

    map(x, dx, n*sizeof(float));
    map(y, dy, n*sizeof(float));

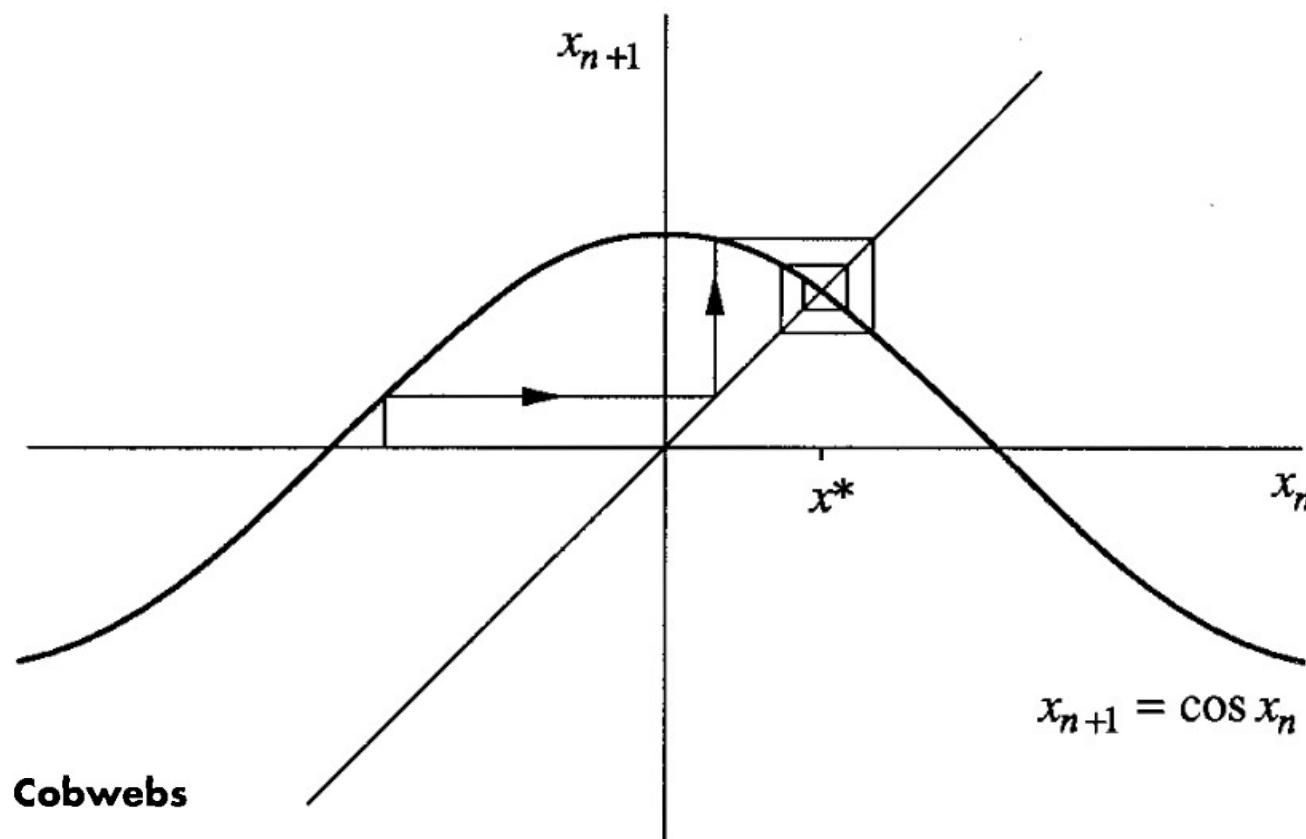
    set(n,1.0f,x);
    set(n,0.0f,y);

    saxpy(n, 2.0, x, y);
    cudaMemcpy(&tmp,dy,
    (size_t)sizeof(float),cudaMemcpyDeviceToHost);
    printf("%f\n",tmp);
    return 0;
}
```

The *acc_map_data* routine is a great way to “set it and forget it” when it comes to sharing memory between CUDA and OpenACC ...

Mapeos unidimensionales

Given $x_{n+1} = \cos x_n$, how does x_n behave as $n \rightarrow \infty$?



Converge a un punto fijo... ¿qué otra cosa puede pasar en general?

Manos a la obra...

- `cd openacc-interoperability`
- `make nombre_ejemplo`
- `qsub submit_gpu.sh nombre_ejemplo`
 - *nombre_ejemplo* es alguno de estos:
 - * **cuda_main** - calling OpenACC from CUDA C
 - * **openacc_c_main** - Calling CUDA from OpenACC in C
 - * **openacc_c_cublas** - Calling CUBLAS from OpenACC in C
 - * **thrust** - Mixing OpenACC and Thrust in C++
 - * **cuda_map** - Using OpenACC acc__map__data with CUDA in C
 - * **cuf_main** - Calling OpenACC from CUDA Fortran
 - * **cuf_openacc_main** - Calling CUDA Fortran from OpenACC
 - * **openacc_cublas** - Calling CUBLAS from OpenACC in CUDA Fortran
 - * **acc_malloc** - Same as cuda_main, but using the OpenACC API
 - * **openacc_streams** - Mixes OpenACC async queues and CUDA streams
 - * **openacc_cuda_device** - Calls a CUDA __device__ kernel within an OpenACC
 - Author: Jeff Larkin <jlarkin@nvidia.com>

Referencias

- OpenACC Course Recordings - <https://developer.nvidia.com/openacc-courses>
- PGI Website - <http://www.pgroup.com/resources>
- OpenACC on StackOverflow - <http://stackoverflow.com/questions/tagged/openacc>
- OpenACC Toolkit - <http://developer.nvidia.com/openacc-toolkit>
- Parallel Forall Blog - <http://devblogs.nvidia.com/parallelforall/>
- GPU Technology Conference - <http://www.gputechconf.com/>
- OpenACC Website - <http://openacc.org/>

Openacc + avanzado

- Optimización de loops
- Asincronismo
- Multi GPU

Quizás más adelante....