Departamento de Computación

OpenACC – Primeros pasos

Elaboró: José Antonio Ayala Barbosa

7 agosto 2021

Facultad de Ingeniería UNAM
División de Ingeniería Eléctrica (DIE)

Tabla de contenido

OpenAcc

Modelo de programación basado en directivas a alto nivel para C/C++ y Fortran. Está diseñado para requerir significantemente menor esfuerzo en la programación heterogénea del cómputo de alto rendimiento que alternativas como CUDA.

Su modelo se basa en insertar "consejos/pistas" (hints) con directivas en el código para ayudar a paralelizar los programas. Cada directiva da una pista al compilador sobre lo que podría ayudar a paralelizar una sección de código, pero el compilador es el que realiza la traducción, por lo que permite al programador enfocarse en resolver el problema en vez de preocuparse de la arquitectura.

OpenAcc puede utilizar concurrentemente distintas plataformas de hardware como x86, ARM, unidades GPU, FPGA, OpenPower, etc.

Sintaxis OpenAcc

OpenAcc no es un lenguaje como tal, es una colección de directivas de compilación, bibliotecas de rutinas, y variables de entorno combinadas en un API para ser soportadas por C/C++ o Fortran.

El compilador interpreta el código y genera código ejecutable en paralelo. Si no se cuenta con un compilador que lo soporte, simplemente se ignorarán las directivas lo que permite mayor flexibilidad de utilización del código generado.

Las directivas siempre siguen la sintaxis siguiente en C/C++

#pragma acc <directiva> [cláusula [[,] cláusula] . . .] new-line

Si se requiere utilizar más de una línea se pude continuar con una diagonal invertida (\ ) al inicio de la nueva línea.

Directivas

Dentro de la nomenclatura propia de OpenAcc, la siguiente palabra después de la etiqueta *acc* es una directiva que le indica al compilador analizar "algo" acerca del bloque de código que sigue. OpenAcc detecta tres tipos de directivas:

1. Directivas de cómputo: Marcan el bloque de código que se puede acelerar explotando la capacidad inherente paralela o distributiva de los datos en distintos threads.
    a. parallel
    b. kernels

   c. routine
   d. loop
2. Directivas de administración de datos: OpenAcc busca eliminar movimiento y el intercambio de datos entre unidades de procesamiento, por lo que se especifica el tratamiento del acceso a los datos.
   a. data
   b. update
   c. cache
   d. atomic
   e. declare
   f. enter data (directiva con dos palabras)
   g. exit data (directiva con dos palabras)
3. Directivas de sincronización: OpenAcc permite ejecutar múltiples estructuras concurrentemente, por lo que es necesario sincronizar su salida.
   a. wait

Cláusulas

Cada directiva puede recibir como argumento una o más cláusulas, cada cláusulas agrega más información sobre lo que el compilador necesita realizar con una estructura. Las cláusulas caen en 3 categorías:

1. Manejo de datos. Asignan un comportamiento al manejo de la información dentro de una estructura:
   a. default
   b. private
   c. firstprivate
   d. copy
   e. copyin
   f. copyout
   g. create
   h. delete
   i. deviceptr
2. Distribución del trabajo. Asignan comportamiento a los threads generados.
   a. seq
   b. auto
   c. gang
   d. worker
   e. vector
   f. tile
   g. num_gangs
   h. num_workers

   i. vector_length
3. Control de flujo. Controlan la ejecución en paralelo.
   a. Ejecución condicional en paralelo -> *if* o *if_present*
   b. Sobrecarga de dependencias -> *independent* o *reduction*
   c. Especificación explicita de paralelismo -> *async* y *wait*

Variables de entorno

Para incluir el API de OpenAcc en un programa es necesario incluir la cabecera:

#include "openacc.h"

Para indicar que utilizaremos diversos dispositivos que se distribuirán el procesamiento, lo realizamos con las siguientes variables de entorno.

ACC_DEVICE_TYPE
ACC_DEVICE_NUM

Construcciones de cómputo (Compute constructs)

OpenAcc distribuye el trabajo en threads paralelos y así buscar el mejor performance reduciendo el tiempo de ejecución global del programa, utiliza cuatro construcciones de estructuras diferentes.

1. Kernels.

Actúa como un centinela para decirle al compilador que el bloque de código que rodea podría acelerarse.

```
#pragma acc kernels
    for (j = 0 ; j < n ; j++ ){
       for (k = 0 ; k < m ; k++ ){
          c[j][k] = a[j][k];
          a[j][k] = c[j][k] + b[j][k];
       }
    }
```

2. Parallel

Está diseñada para que el programador especifique explícitamente que el bloque de código se realice en paralelo, y el programador debe ser quien cuide las redundancias del código, ya que puede haber condiciones de carrera y eventual pérdida de información.

```
#pragma acc parallel
    for (j = 0 ; j < n ; j++ ){
        for (k = 0 ; k < m ; k++ ){
            c[j][k] = a[j][k];
            a[j][k] = c[j][k] + b[j][k];
        }
    }
```

Al utilizar kernels, el compilador es quien decide cuántos threads se crearán para resolver en paralelo.

Supongamos que tenemos el siguiente código:

```
#pragma acc kernels
    for (j = 0 ; j < n ; j++ ){
        for (k = 0 ; k < m ; k++ ){
            c[j][k] = a[j][k];
            a[j][k] = c[j][k] + b[j][k];
            d[j][k] = a[j][k] - 5;
        }
    }
```

El compilador intuiría que se puede separar en dos estructuras para hacer más eficiente la obtención del resultado ya que no existe dependencia de datos entre las instrucciones, dándonos los siguientes códigos:

```
#pragma acc parallel loop
    for (j = 0 ; j < n ; j++ ){
        for (k = 0 ; k < m ; k++ ){
            c[j][k] = a[j][k];
            a[j][k] = c[j][k] + b[j][k];
        }
    }
```

```
#pragma acc parallel loop
    for (j = 0 ; j < n ; j++ ){
        for (k = 0 ; k < m ; k++ ){
            d[j][k] = a[j][k] - 5;
        }
    }
```

3. Loop

Puede ser usado dentro de la construcción kernels o dentro de parallels e indica al compilador que cada iteración del bucle es independiente entre sí, y ayuda cuando el compilador no puede determinar por él mismo si hay o no dependencias. Se puede escribir en jerarquía de *#pragmas* o pueden colapsarse en uno solo, dependiendo del caso.

#pragma acc parallel *#pragma acc loop*
#pragma acc parallel loop

4. Routine

Marca que una función es potencialmente paralelizable. Puede utilizarse con diferentes sintaxis.

#pragma acc routine *extern void foo(int *a, int *b, int n);*
#pragma acc routine (foo)
#pragma acc routine *void foo(int *a, int *b, int n){* *…* *}*

Ambiente de datos

OpenAcc está diseñado para manejar ambientes donde los construcutores están ejecutandose en memoria separada del programa principal. Cada copia generada estará predefinida como *private*, lo que significa que habrá cuantas copias determine el compilador para realizar el procesamiento.

Las variables escalares estarán definidas como *firstprivate*.

Existen dos tipos de conceptos relacionados a los objetos de dato:

1. Data region: Scope dinámico de un bloque estructurado asociado a una construcción de datos implícito o explícito.
2. Data timelife: Empieza cuando un objeto se crea por primera vez dentro de un dispositivo y termia cuando el objeto no se encuentra disponible.

Directivas de datos

Existen dos tipos:

1. Estructuradas. Región que se define en un simple scope léxico cuando el lifetime de los datos inicia o termina.

```
#pragma acc data copy(a)
{
        <use a>
}
```

   La vida de a inicia cuando empieza la llave y termina cuando se cierra.

2. No estructuradas. Su delimitación no está necesariamente en el mismo scope

```
void foo(int *array, int n){
#pragma acc enter data copyin(array[o:n])
}
```

```
void bar(int *array, int n){
#pragma acc exit data copyiout(array[o:n])
}
```

   Deben ponerse las directivas en el lugar correcto para aumentar el rendimiento del programa, de lo contrario se pueden generar copias innecesarias de datos.

Directivas de cláusula

Especifican el manejo de ciertas variables o arreglos. Existen siete:

1. *create*. Asigna datos e inicia el lifetime de un objeto en el device.
2. *present*. Se asegura que el objeto esté disponible en el device.
3. *copy*. Copia los datos del host al device en el inicio de la región, y posteriormente cuando termina la ejecución de l construcción regresa la información al host.
4. *copyin*. Solamente copia los datos del host al device.
5. *copyout*. Solamente copia los datos del device al host.
6. *delete*. Determina si el objeto está presente, si no, no hace nada, pero si está lo borra del ambiente del device forzando que las referencias se vuelvan cero y limpiando la memoria.
7. *deviceptr*. Le dice al compilador que el apuntador en la cláusula contiene una dirección que se encuentra en el device, lo que permite tomar control de la colocación de los datos fuera de OpenAcc pero aun usarlos dentro del sistema vía API.

Directivas de caché

Provee un mecanismo para informar que los datos deben moverse a una memoria más rápida, si es posible.

```
#pragma acc loop
    for (j = 0 ; j < n ; j++ ){
    #pragma acc cache(b[j])
        b[j] = b[j]*c;
```

```
}
```

En este ejemplo caché le dice al compilador que cada iteración del bucle for usa un elemento del arreglo b. Así el compilador puede asignar tantos threads como necesite para que cada iteración la realice en paralelo.

```
#pragma acc data copy(a[0:n])
```

Transferencias parciales de datos

Cuando sólo se utilizará una fracción de un arreglo, podemos indicárselo al compilador con la información necesaria de los límites para que no pierda tiempo en copiar toda la estructura de datos.

Niveles de paralelismo

Podemos definir niveles donde ocurrirá el paralelismo.

vectors	Es el elemento de granularidad más fina, su símil en SIMD es el thread.
workers	Son el intermediario entre gangs y vectors, permiten el mapeo de los vectors en el hardware.
gangs	Es un grupo o bloque de workers, en diferentes gangs, los workers pueden tener trabajos independientes.

#pragma acc parallel num_gangs(#), vector_length(#)

Compilación

Compilador	**Banderas**	**Banderas adicionales**
PGI	-acc	-ta=target architecture -Minfo=accel
GCC	-fopenacc	-foffload=offload target
OpenUH	Compilar: -fopenacc Link: -lopenacc	-Wb -accarch:target architecture
Cray	-h pragma=acc	- msgs

OpenAcc viene preinstalado en los compiladores antes listados. El compilador que tiene la documentación más extensa es PGI donde se puede utilizar sin requerir de activar ninguna variable.

Para los demás es requerido instalar CUDA toolkit, por lo que es necesario tener hardware de NVIDIA, mientras que en PGI funcionará como OpenMP utilizando únicamente los cores del procesador.

pgcc -acc -ta=multicore –o jacobiACC driverA.c
pgcc -acc -ta=nvidia –o jacobiACC driverA.c

Se adjunta la presentación [9] con un ejemplo sobre cómo transformar código secuencial C en paralelo con OpenACC. También se adjuntan los archivos de las actividades ahí marcadas. Material obtenido en el "Congreso de Supercómputo ISUM 2018".

En [8] se puede encontrar material de cursos impartidos por NVIDIA.

Referencias

Teoría

[1] https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf
[2] https://www.pgroup.com/resources/docs/20.4/pdf/pgirn204-x86.pdf
[3] https://fisica.cab.cnea.gov.ar/gpgpu/images/2017/intro_openacc_last.pdf
[4] https://www.hpcwire.com/2015/10/29/pgi-accelerator-compilers-add-openacc-support-for-x86-multicore-cpus-2/
[5] https://exascaleproject.org/wp-content/uploads/2017/04/Messina-ECP-Presentation-HPC-User-Forum-2017-04-18.pdf

Código

[6] http://www.openacc.org/content/tools
[7] https://gcc.gnu.org/wiki/OpenAcc
[8] https://developer.nvidia.com/openacc-courses
[9] http://www.inf.ufrgs.br/erad2018/downloads/minicursos/eradrs2018-openacc.pdf