EBook Gratis

APRENDIZAJE opencl

Free unaffiliated eBook created from
Stack Overflow contributors.

#opencl

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: opencl

It is an unofficial and free opencl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official opencl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con opencl

Observaciones

Esta sección proporciona una descripción general de qué es opencl y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema importante dentro del tema, y vincularlo con los temas relacionados. Dado que la Documentación para opencl es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Opencl es una api que pone gpus, cpus y algunos otros aceleradores (como un pcie-fpga) en un buen uso de los cálculos tipo C99 pero con una ventaja de concurrencia muy amplia. Una vez que se realiza la instalación y la implementación básica, solo los cambios simples en una cadena del kernel (o su archivo) aplican un algoritmo a N hilos de hardware de forma automática.

Un desarrollador podría querer usarlo porque será mucho más fácil optimizar el espacio o la velocidad de la memoria que hacer lo mismo en opengl o direct-x. También es libre de derechos. La concurrencia dentro de un dispositivo es implícita, por lo que no hay necesidad de múltiples subprocesos explícitos para cada dispositivo. Pero para las configuraciones de múltiples dispositivos, aún se necesita un cpu-multi-threading. Por ejemplo, cuando se envía un trabajo de 1000 hilos a una CPU, el controlador maneja la sincronización de hilos. Simplemente diga qué tan grande debe ser un grupo de trabajo (como 256 cada uno conectado a la memoria local virtual) y dónde están los puntos de sincronización (solo cuando sea necesario).

Usar gpu para operaciones de propósito general es casi siempre más rápido que la CPU. Puede ordenar las cosas más rápido, multiplicar las matrices 10 veces más rápido y unirlas a la izquierda en tablas "sql" en memoria "no". Cualquier gpu de 200 $ para computadoras de escritorio terminará más rápido en una carga de trabajo de física (método de elementos finitos) que cualquier 200 $ cpu. Opencl lo hace más fácil y portátil. Cuando haya terminado de trabajar en C #, puede pasar fácilmente a la implementación de java-opencl utilizando los mismos núcleos y el proyecto de C ++ (por supuesto, utilizando JNI con compilación de C ++ adicional).

Para la parte de gráficos, no siempre tienes que enviar buffers entre cpu y gpu. Puede trabajar puramente en gpu usando la opción "interoperabilidad" en la parte de creación de contexto. Con interoperabilidad, puede preparar geometrías en el rendimiento límite de gpu. No se requiere pci-e para ningún dato de vértice. Solo se envía un "comando" y el trabajo se realiza solo dentro de la tarjeta gráfica. Esto significa que no hay CPU-sobrecarga para los datos. Opencl prepara datos de geometría, opengl los procesa. La CPU se libera. Por ejemplo, si un solo hilo de cpu puede construir una esfera vertical de 32x32 en 10000 ciclos, entonces un gpu con opencl puede construir 20 esferas en 1000 ciclos.

Examples

¿Qué es OpenCL?

OpenCL es la abreviatura de **Open C** omputing **L** angguage. OpenCL es un marco para la programación paralela en plataformas heterogéneas, llamadas *dispositivos informáticos* , que van desde CPUs a través de GPUs a plataformas más especiales como FPGAs. OpenCL proporciona una interfaz estándar para la computación paralela en estos dispositivos informáticos, pero también un paralelismo entre dispositivos. Especifica un lenguaje de programación, basado en C99, y requisitos mínimos de funciones básicas implementadas en dispositivos con capacidad OpenCL. Además, OpenCL describe un modelo abstracto de computación y memoria, siendo lo más general posible para hacer la reutilización de código entre diferentes plataformas de manera directa.

Prerrequisitos

Si tiene una CPU o tarjeta gráfica (GPU) moderna dentro de su máquina, es probable que tenga todo listo para los primeros pasos en OpenCL. Averiguar si su procesador es compatible con OpenCL se puede hacer generalmente a través de la página de inicio del fabricante, un buen comienzo es la documentación oficial en

https://www.khronos.org/conformance/adopters/conformant-products#opencl

¿Qué es OpenCL?

Open Computing Language (OpenCL) es un marco para escribir programas que se ejecutan en CPU, GPU y otros procesadores y aceleradores paralelos.

OpenCL especifica un lenguaje de programación (basado en C) que proporciona acceso a la memoria en chip nombrada, un modelo para ejecutar tareas en paralelo y la capacidad de sincronizar esas tareas.

Implementación C # de OpenCL 1.2: número de plataformas para un sistema AMD en ventanas de 64 bits

OpenCL es un api de bajo nivel, por lo que primero debe implementarse en "espacio C". Para eso, uno necesita descargar archivos de cabecera desde el sitio de Khronos. Mi hardware es AMD y capaz de la versión 1.2, descargando

```
opencl.h
cl_platform.h
cl.h
cl_ext.h
cl_egl.h
cl_dx9_media_sharing.h
cl_d3d10.h
cl_d3d11.h
cl_gl.h
cl_gl_ext.h
cl.hpp
```

de esta pagina

debería ser suficiente para los enlaces de C ++, así que después de agregar estos archivos a su proyecto y de establecer las ubicaciones de archivos binarios (y de biblioteca) adecuadas (

> $ (AMDAPPSDKROOT) \ lib \ x86_64 para la biblioteca amd de 64 bits (se prefieren las bibliotecas amd de la aplicación sdk)

,

> C: \ Windows \ SysWOW64 para opencl.dll de 64 bits (archivo .so si ICD es de un sistema Linux)

por ejemplo, pero diferente para Intel-Nvidia), puede comenzar a consultar una lista de plataformas (amd, intel, xilinx, nvidia) después de instalar los controladores adecuados (como carmesí para amd). Los controladores son para ejecutar la aplicación opencl (usando ICD), las bibliotecas y los archivos de encabezado son para que el desarrollo sea breve.

Para consultar plataformas:

```
#define __CL_ENABLE_EXCEPTIONS
#include "stdafx.h"
#include <vector>
#include <CL/cl.hpp>

extern "C"
    {
       // when this class is created, it contains a list of platforms in "platforms" field.
       class OpenClPlatformList
       {
           public:
               std::vector<cl::Platform> platforms;
               int platformNum;
               OpenClPlatformList()
               {
                   platforms= std::vector< cl::Platform>();
                   cl::Platform::get(&platforms);
                   platformNum= platforms.size();
               }
        };


        // this is seen from C# when imported. Creates an object in memory.
        __declspec(dllexport)
            OpenClPlatformList * createPlatformList()
        {
            return new OpenClPlatformList();
        }

        __declspec(dllexport)
            int platformNumber(OpenClPlatformList * hList)
        {
            return hList->platformNum;
        }


        __declspec(dllexport)
```

```
            void deletePlatformList(OpenClPlatformList * p)
        {
            if (p != NULL)
                delete p;
            p = NULL;
        }


    }
```

podría estar integrado en un archivo DLL (como OCLImplementation.dll)

y usarlo desde el lado C #,

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;


namespace WrapperCSharp
{
    public class WrapperCSharp
    {
        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern IntPtr createPlatformList();

        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern int platformNumber(IntPtr hList);

        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern void deletePlatformList(IntPtr hList);
    }
}
```

Por supuesto, el archivo C # debe ver el archivo dll, simplemente colocándolo cerca del ejecutable del proyecto lo resuelve.

Ahora, si la computadora de muestra tiene al menos una plataforma compatible con Opencl,

```
IntPtr platformList = createPlatformList(); // just an address in C-space
int totalPlatforms = platformNumber(platformList); // AMD+NVIDIA systems should have "2"
deletePlatformList(platformList); //
```

La variable totalPlatforms debe tener al menos el valor "1". Luego, puede usar las plataformas variables en C-space usando funciones adicionales para recorrer en iteración todas las plataformas para consultar todos los dispositivos, como CPU, GPU y aceleradores de propósitos especiales como phi o algunos fpga.

Uno no se limita a escribir todos estos envoltorios de C ++ en C # para proyectos de tiempo crítico. Hay muchos envoltorios escritos para C #, Java y otros lenguajes. Para java, hay "Aparapi" que es el api convertidor de "bycecode java a opencl-c" que lleva lo que se escribe en java a una versión gpu-paralela sobre la marcha, por lo que es algo portátil.

OpenCL y C

Para C # existen muchos envoltorios que ofrecen una interfaz para comunicarse con OpenCL.

- OpenCL.NET: este es uno de los envoltorios de nivel más bajo que hay. Ofrece una implementación completa de la API de OpenCL para C # sin agregar ningún tipo de abstracción. Por lo tanto, los ejemplos de C \ C ++ son fácilmente portados para esta biblioteca. La única página de proyecto está actualmente en codeplex, que se cierra el 15.12.2017 pero el paquete está disponible en NuGet

https://openclnet.codeplex.com/

- NOpenCL: esta biblioteca ofrece una interfaz abstracta entre C # y OpenCL.

  El objetivo a corto plazo es proporcionar una capa abstracta fácil de usar que proporcione acceso a toda la capacidad de OpenCL sin sacrificar el rendimiento.

https://github.com/tunnelvisionlabs/NOpenCL

- Cloo:

  Cloo es una biblioteca de código abierto, fácil de usar y administrada que permite que las aplicaciones .NET / Mono aprovechen al máximo el marco OpenCL.

https://sourceforge.net/projects/cloo/

Lea Empezando con opencl en línea: https://riptutorial.com/es/opencl/topic/3694/empezando-con-opencl

Capítulo 2: Configuración básica de OpenCL

Introducción

Antes de utilizar OpenCL, uno tiene que configurar su código para usarlo. Este tema se enfoca en cómo hacer que Opencl se ejecute y ejecute en su proyecto y ejecute un núcleo básico. Los ejemplos se basan en el contenedor OpenCL.NET de C #, pero como el contenedor no agrega abstracción a OpenCL, el código probablemente se ejecutará con muy pocos cambios en C / C ++ también.

Las llamadas en C # pueden tener el siguiente aspecto: 'Cl.GetPlatformIDs'. Para el Api OpenCL de estilo C, se llamaría 'clGetPlatformIDs' y para el estilo C ++ one 'cl :: GetPlatformIDs'

Observaciones

- NVidia, AMD e Intel tienen implementaciones ligeramente diferentes de OpenCL, pero las diferencias conocidas están (según mi experiencia) limitadas a los requisitos de paréntesis y las conversiones implícitas. A veces NVidia colapsará tu kernel mientras trata de averiguar la sobrecarga correcta para un método. En este caso, es útil ofrecer un reparto explícito para ayudar a la GPU. El problema fue observado para los núcleos compilados en tiempo de ejecución.
- Para obtener más información sobre las llamadas utilizadas en este tema, basta con google 'OpenCL' seguido del nombre de la función. El grupo Khronos tiene una documentación completa sobre todos los parámetros y tipos de datos disponibles en su sitio web.

Examples

Inicializando el dispositivo de destino

Los núcleos OpenCL pueden ejecutarse en la GPU o en la CPU. Esto permite soluciones alternativas, donde el cliente puede tener un sistema muy obsoleto. El programador también puede elegir limitar su funcionalidad a la CPU o GPU.

Para comenzar a utilizar OpenCL, necesitará un 'Contexto' y un 'Dispositivo'. Ambas son estructuras definidas por la API de OpenCL (también conocida como cl :: Context o clContext & ~ Device) y definen el procesador de destino utilizado.

Para obtener su dispositivo y contexto, necesita consultar una lista de plataformas disponibles, cada una de las cuales puede alojar múltiples dispositivos. Una plataforma representa su GPU física y CPU, mientras que un dispositivo puede distinguir aún más las unidades informáticas contenidas. Para las GPU, la mayoría de las plataformas solo tendrán un dispositivo. Pero una CPU puede ofrecer una GPU integrada adicional junto a sus capacidades de CPU.

El contexto gestiona la memoria, las colas de comandos, los diferentes kernels y programas. Un

contexto puede limitarse a un solo dispositivo, pero también hacer referencia a múltiples dispositivos.

Una nota rápida de la API antes de comenzar a codificar: Casi todas las llamadas a OpenCL le dan un valor de error, ya sea como valor de retorno o mediante un valor de referencia (puntero en C). Ahora vamos a empezar.

```
ErrorCode err;
var platforms = Cl.GetPlatformIDs(out err);
if(!CheckError(err, "Cl.GetPlatformIDs")) return;
foreach (var platform in platforms) {
    foreach (var device in Cl.GetDeviceIDs(platform, DeviceType.Gpu, out err)) {
        if(!CheckError(err, "Cl.GetDeviceIDs")) continue;
        [...]
    }
}
```

Este fragmento de código consulta todos los dispositivos GPU disponibles en el sistema. Ahora puede agregarlos a una lista o comenzar su contexto directamente con la primera coincidencia. La función 'CheckError (...)' es una utilidad simple, que verifica si el código de error tiene el valor de éxito o uno diferente y puede ofrecerle algún registro. Se recomienda utilizar una función o macro por separado, porque llamará mucho a eso.

ErrorCode es solo una enumeración en el tipo de datos cl_int para C #, C / C ++ puede comparar el valor int con constantes de error predefinidas como se muestra aquí: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/ xhtml / errors.html

También es posible que desee comprobar si el dispositivo admite todas las funciones necesarias, de lo contrario, sus núcleos podrían bloquearse en el tiempo de ejecución. Puede consultar una capacidad del dispositivo con

```
Cl.GetDeviceInfo(_device, DeviceInfo.ImageSupport, out err)
```

Este ejemplo le pregunta al dispositivo si puede ejecutar funciones de imagen. Para el siguiente y último paso, debemos construir nuestro contexto a partir de los dispositivos recopilados.

```
_context = Cl.CreateContext(null, 1, new[] { _device }, ContextNotify, IntPtr.Zero, out err);
```

Algunas cosas están pasando aquí. Para la gente de C / C ++, IntPtr es una dirección de puntero en C #. Me concentraré en las partes importantes aquí.

- El segundo parámetro define la cantidad de dispositivos que desea usar
- El tercer parámetro es una matriz de esos dispositivos (o un puntero en C / C ++)
- Y el tercer parámetro es un puntero de función para una función de devolución de llamada. Esta función se utilizará siempre que ocurran errores dentro del contexto.

Para su uso futuro, deberá conservar los dispositivos usados y el contexto en algún lugar.

Cuando hayas terminado toda tu interacción con OpenCL, necesitarás liberar el contexto nuevamente con

```
Cl.ReleaseContext(_context);
```

Compilando tu Kernel

Los kernels se pueden compilar en tiempo de ejecución en el dispositivo de destino. Para ello, necesitas

- el código fuente del kernel
- El dispositivo de destino en el que compilar
- un contexto construido con el dispositivo de destino

Una actualización rápida de terminología: un programa contiene una colección de núcleos. Puede pensar en un programa como un archivo fuente completo de C / C ++ / C #, mientras que los núcleos son los diferentes miembros de la función de ese archivo.

Primero necesitarás crear un programa a partir de tu código fuente.

```
var program = Cl.CreateProgramWithSource(_context, 1, new[] { source }, null, out err);
```

Puede combinar varios archivos de origen en un solo programa y compilarlos juntos, lo que le permite tener núcleos en diferentes archivos y compilarlos juntos de una sola vez.

En el siguiente paso, deberá compilar el programa en su dispositivo de destino.

```
err = Cl.BuildProgram(program, 1, new[] { _device }, string.Empty, null, IntPtr.Zero);
```

Ahora viene una pequeña advertencia: el código de error solo le dice si la llamada a la función fue exitosa, pero no si su código realmente se compiló. Para verificar eso, tenemos que consultar alguna información adicional.

```
BuildStatus status;
status = Cl.GetProgramBuildInfo(program, _device, ProgramBuildInfo.Status, out
err).CastTo<BuildStatus>();
if (status != BuildStatus.Success) {
    var log = Cl.GetProgramBuildInfo(program, _device, ProgramBuildInfo.Log, out err);
}
```

La gente de C / C ++ puede ignorar la conversión al final y simplemente comparar el entero devuelto con la constante correspondiente.

La primera llamada verifica si nuestra compilación fue realmente exitosa. Si no, podemos recuperar un registro y ver exactamente dónde salieron las cosas mal. Vea las observaciones de algunos pitfals comunes con respecto a diferentes plataformas.

Una vez que se construye el programa, necesitas extraer tus diferentes núcleos del programa compilado. Para ello creas tus kernels con

```
_kernel = Cl.CreateKernel(_program, kernel, out err);
```

donde 'kernel' es una cadena del nombre del kernel. Cuando haya terminado con su kernel, necesita liberarlo con

```
Cl.ReleaseKernel(_kernel);
```

Creando una cola de comandos

Para iniciar cualquier operación en sus dispositivos, necesitará una cola de comandos para cada dispositivo. La cola hace un seguimiento de las diferentes llamadas que hizo al dispositivo de destino y las mantiene en orden. La mayoría de los comandos también se pueden ejecutar en modo de bloqueo o no bloqueo.

Crear una cola es bastante sencillo:

```
_queue = Cl.CreateCommandQueue(_context, _device, CommandQueueProperties.None, out err);
```

La interacción básica con su cola de comandos es poner en cola las diferentes operaciones que desea realizar, por ejemplo, copiar datos desde y hacia su dispositivo e iniciar un kernel.

Cuando haya terminado de usar la cola de comandos, debe liberar la cola con una llamada a

```
Cl.ReleaseCommandQueue(_queue);
```

Ejecutando el Kernel

Así que ahora vamos a lo real, ejecutando sus núcleos en el dispositivo paralelo. Lea acerca de los conceptos básicos de hardware para comprender completamente el envío del kernel.

Primero deberá establecer los argumentos del kernel antes de llamar al kernel. Esto se hace a través de

```
err = Cl.SetKernelArg(_kernel, $argumentIndex, $argument);
```

Si no establece todos los argumentos antes de lanzar el kernel, el kernel fallará.

Antes de lanzar nuestro kernel, debemos calcular el 'tamaño de trabajo global' y el 'tamaño de trabajo local'.

el tamaño de trabajo global es el número total de subprocesos que se lanzarán en su GPU. El tamaño de trabajo local es el número de subprocesos dentro de cada bloque de subprocesos. El tamaño de la obra local se puede omitir si el núcleo no necesita ningún requisito especial. Pero si se especifica el tamaño del trabajo local, el tamaño del trabajo global debe ser un múltiplo del tamaño del trabajo local.

Los tamaños de trabajo pueden ser unidimensionales, bidimensionales o tridimensionales. La elección de cuántas dimensiones desea depende totalmente de usted y puede elegir la que mejor se adapte a su algoritmo.

Ahora que decidimos nuestros tamaños de trabajo, podemos llamar el núcleo.

```
Event clevent;
err = Cl.EnqueueNDRangeKernel(_queue, _kernel, $dimensions, null, $globalWorkSize,
$localWorkSize, 0, null, out clevent);
```

Las $ dimensiones definen nuestro número deseado de dimensiones, $ globalWorkSize es un conjunto de dimensiones $ dimensiones con el tamaño de Trabajo global y lo mismo para $ localWorkSize. El último argumento le da un objeto que representa su kernel actualmente ejecutado.

Lea Configuración básica de OpenCL en línea:
https://riptutorial.com/es/opencl/topic/10169/configuracion-basica-de-opencl

Capítulo 3: Ejemplo de kernel generador de números pseudoaleatorios

Parámetros

Parámetro	Detalles
__global unsigned int * rnd_buffer	unsigned int está estandarizado por el estándar OpenCL como de 32 bits
*	__global significa la memoria principal del dispositivo para el acceso de lectura / escritura
*	rnd_buffer es solo un nombre en el alcance de "opencl program" (no host sino dispositivo)

Examples

Usando la función de hash entero de Thomas Wang

Función auxiliar que toma una semilla y evalúa:

```
uint wang_hash(uint seed)
{
        seed = (seed ^ 61) ^ (seed >> 16);
        seed *= 9;
        seed = seed ^ (seed >> 4);
        seed *= 0x27d4eb2d;
        seed = seed ^ (seed >> 15);
        return seed;
 }
```

otra función auxiliar que la usa para inicializar una ubicación de búfer mostrada por "id":

```
 void wang_rnd_0(__global unsigned int * rnd_buffer,int id)
 {
     uint maxint=0;
     maxint--;
     uint rndint=wang_hash(id);
     rnd_buffer[id]=rndint;
 }
```

y otro haciendo salida flotante extra entre 0 y 1.

```
 float wang_rnd(__global unsigned int * rnd_buffer,int id)
 {
     uint maxint=0;
```

```
    maxint--; // not ok but works
    uint rndint=wang_hash(rnd_buffer[id]);
    rnd_buffer[id]=rndint;
    return ((float)rndint)/(float)maxint;
}
```

Inicializador kernel:

```
__kernel void rnd_init(__global unsigned int * rnd_buffer)
{
      int id=get_global_id(0);
      wang_rnd_0(rnd_buffer,id);  // each (id) thread has its own random seed now
}
```

Kernel de iteración única:

```
__kernel void rnd_1(__global unsigned int * rnd_buffer)
{
     int id=get_global_id(0);

     // can use this to populate a buffer with random numbers
     // concurrently on all cores of a gpu
     float thread_private_random_number=wang_rnd(rnd_buffer,id);
}
```

Lea Ejemplo de kernel generador de números pseudoaleatorios en línea: https://riptutorial.com/es/opencl/topic/5893/ejemplo-de-kernel-generador-de-numeros-pseudoaleatorios

Capítulo 4: Fundamentos de hardware de OpenCL

Introducción

Este tema presenta algunos de los mecanismos básicos subyacentes de la computación paralela que son necesarios para comprender completamente y utilizar OpenCL.

Examples

Hilos y Ejecución

La clave del paralelismo es usar varios subprocesos para resolver un problema (duh.) Pero hay algunas diferencias con la programación clásica de subprocesos múltiples en cómo se organizan los subprocesos.

Primero hablemos de su GPU típica, por simplicidad, me centraré en

Una GPU tiene muchos núcleos de procesamiento, lo que la hace ideal para ejecutar muchos subprocesos en paralelo. Esos núcleos están organizados en Streaming Processors (SM, término de NVidia), de los cuales una GPU tiene un número dado.

Todos los subprocesos que se ejecutan dentro de un SM se denominan "bloque de subprocesos". Puede haber más hilos en un SM que tiene núcleos. El número de núcleos define el llamado 'Tamaño de deformación' (término NVidia). Los hilos que se encuentran dentro de un bloque de hilos están programados en las denominadas "deformaciones".

Un rápido ejemplo de seguimiento: un NVidia SM típico tiene 32 núcleos de procesamiento, por lo tanto, su tamaño de deformación es 32. Si mi bloque de hilos ahora tiene 128 hilos para ejecutar, se procesarán en 4 deformaciones (4 deformaciones * 32 tamaño de deformación = 128 trapos).

El tamaño de la urdimbre es bastante importante al elegir el número de subprocesos más adelante.

Todos los hilos dentro de una sola urdimbre comparten un solo contador de instrucciones. Eso significa que esos 32 subprocesos están verdaderamente sincronizados, ya que cada subproceso ejecuta todos los comandos al mismo tiempo. Aquí hay un error de rendimiento: ¡Esto también se aplica a las declaraciones de ramificación en su núcleo!

Ejemplo: tengo un kernel que tiene una sentencia if y dos ramas. 16 de mis hilos dentro de una urdimbre ejecutarán la rama uno, los otros 16 la rama dos. Hasta la instrucción if, todos los hilos dentro de la deformación están sincronizados. Ahora la mitad de ellos elige una rama diferente. Lo que sucede es que la otra mitad permanecerá inactiva hasta que la declaración incorrecta haya terminado de ejecutarse en los primeros 16 subprocesos. Entonces esos hilos estarán inactivos hasta que los otros 16 hilos terminen su rama.

Como puede ver, los malos hábitos de bifurcación pueden ralentizar considerablemente su código paralelo, porque ambas declaraciones se ejecutan en el peor de los casos. Si todos los hilos dentro de una deformación deciden que solo necesitan una de las declaraciones, la otra se omite por completo y no se produce ningún retraso.

Sincronizar hilos tampoco es un asunto simple. **Sólo** puede sincronizar los hilos withing un único SM. Todo lo que se encuentra fuera del SM es inescrutable desde el interior del núcleo. Tendrás que escribir núcleos separados y lanzarlos uno tras otro.

Memoria de la GPU

La GPU ofrece seis regiones de memoria diferentes. Se diferencian en su latencia, tamaño y accesibilidad de diferentes hilos.

- Memoria global: la memoria más grande disponible y una de las pocas para intercambiar datos con el host. Esta memoria tiene la mayor latencia y está disponible para todos los subprocesos.
- Memoria constante: una parte de solo lectura de la memoria global, que solo puede ser leída por otros hilos. Su ventaja es la menor latencia en comparación con la memoria global.
- Memoria de textura: también una parte de la memoria constante, específicamente diseñada para texturas
- Memoria compartida: esta región de memoria se coloca cerca del SM y solo se puede acceder a ella mediante un único bloque de subproceso. Ofrece una latencia mucho menor que la memoria global y un poco menos de latencia que la memoria constante.
- Registros: Solo accesible por un solo hilo y la memoria más rápida de todos. Pero si el compilador detecta que no hay suficientes registros para las necesidades del núcleo, subcontratará las variables a la memoria local.
- Memoria local: una parte de la memoria accesible solo mediante subprocesos en la región de memoria global. Se usa como respaldo para los registros, para evitarlos si es posible.

Acceso a la memoria

El escenario típico para el uso de la memoria es almacenar los datos de origen y los datos procesados en la memoria global. Cuando se inicia un bloqueo de subprocesos, primero copia todas las partes relevantes en la memoria compartida antes de obtener sus partes en los registros.

La latencia de acceso a la memoria también depende de su estrategia de memoria. Si accede ciegamente a los datos, obtendrá el peor rendimiento posible.

Los diferentes recuerdos están organizados en los llamados 'bancos'. Cada solicitud de memoria para un banco se puede manejar en un solo ciclo de reloj. El número de bancos en la memoria compartida es igual al tamaño de la deformación. La velocidad de la memoria se puede aumentar evitando el acceso al banco en conflicto dentro de una sola deformación.

Para copiar la memoria compartida desde o hacia la memoria global, la forma más rápida es "alinear" sus llamadas de memoria. Esto significa que el primer hilo en una deformación debe acceder al primer elemento en el banco de la memoria compartida y global. El segundo hilo del

segundo elemento y así sucesivamente. Esta llamada se optimizará en una sola instrucción de transferencia de memoria que copia todo el banco a la memoria de destino de una sola vez.

Lea Fundamentos de hardware de OpenCL en línea:
https://riptutorial.com/es/opencl/topic/10168/fundamentos-de-hardware-de-opencl

Capítulo 5: Fundamentos del Kernel

Introducción

Este tema tiene como objetivo explicar los fundamentos de la escritura de los núcleos para el funcionamiento.

Examples

Kernel en escala de grises

Permite construir un kernel para generar una imagen en escala de grises. Utilizaremos los datos de imagen que se definen mediante uints para cada componente y con el pedido RGBA.

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                               CLK_ADDRESS_CLAMP_TO_EDGE |
                               CLK_FILTER_LINEAR;

__kernel void Grayscale(__read_only image2d_t input, __write_only image2d_t output) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int2 size = get_image_dim(input);

    if(all(gid < size)){
        uint4 pixel = read_imageui(input, sampler, gid);
        float4 color = convert_float4(pixel) / 255;
        color.xyz = 0.2126*color.x + 0.7152*color.y + 0.0722*color.z;
        pixel = convert_uint4_rte(color * 255);
        write_imageui(output, gid, pixel);
    }
}
```

Ahora veamos ese código paso a paso. La primera línea crea una variable en la __ región de memoria constante de tipo sampler_t. Esta muestra se utiliza para especificar aún más el acceso a nuestros datos de imagen. Consulte la documentación de Khronos para obtener una documentación completa.

Asignamos la entrada como read_only y la salida como write_only antes de llamar a nuestro kernel, así que agregamos esos modificadores aquí.

image2d y image3d siempre se asignan en la memoria global, por lo tanto, podemos omitir el modificador __global aquí.

Luego obtenemos nuestro ID de hilo que define el píxel que vamos a convertir a escala de grises. También consultamos el tamaño para asegurarnos de que nuestro hilo no está accediendo a la memoria no asignada. Esto definitivamente bloqueará tu kernel si olvidas eso.

Después de asegurarnos de que somos un hilo legítimo, leemos nuestro píxel fuera de nuestra imagen de entrada. Luego lo convertimos en flotante para evitar la pérdida de lugares decimales, hacemos algunos cálculos, lo convertimos y lo escribimos en la salida.

Kernel Skelleton

Caminemos por el núcleo más simple que hay y algunas variaciones de él.

```
__kernel void myKernel() {
}
```

Un kernel que puede iniciarse desde su código principal se identifica con la palabra clave __kernel. Una función de Kernel solo puede tener el tipo de retorno void.

```
__kernel void myKernel(float a, uint b, byte c) {

}
```

Por supuesto, puedes crear más funciones que no estén expuestas como núcleos. En este caso, simplemente puede omitir el modificador __kernel.

Una función puede exponer variables como cualquier otra función C / C ++. La única diferencia es cuando quieres hacer referencia a la memoria. Esto se aplica a todos los punteros, ya sean argumentos o se utilicen en el código.

```
float*  ptr;
```

es un puntero a una región de memoria a la que solo tiene acceso el subproceso en ejecución. De hecho es lo mismo que

```
__private float* ptr;
```

Hay cuatro modificadores de región de memoria diferentes disponibles. Dentro del núcleo, normalmente no tiene que preocuparse por eso, pero para los argumentos es esencial.

- __global: este modificador se refiere a un puntero que se coloca en la memoria global
- __constante: se refiere a un puntero de memoria constante
- __local: se refiere a un puntero de memoria compartida
- __privado: se refiere a un puntero de memoria local

Además podemos definir cómo queremos acceder a la memoria.

- sin modificador: leer y escribir
- __solo lectura
- __escribir solamente

Esos indicadores deben coincidir con la forma en que asignamos el búfer de memoria a nuestro host.

ID de Kernel

Para trabajar correctamente con los datos, cada subproceso necesita conocer su posición en el

conjunto de subprocesos / bloque de subprocesos. Esto puede ser archivado con

```
get_local_id($dim);
get_global_id($dim);
```

Esas dos funciones devuelven la posición del hilo en relación con el bloque de subprocesos o todos los subprocesos.

```
get_working_dim();
```

Obtiene el número total de dimensiones con las que se lanzó el kernel.

```
get_local_size($dim);
get_global_size($dim);
```

Obtiene el número total de subprocesos en el bloque de subprocesos o en total para una dimensión determinada.

Advertencia: asegúrese siempre de que su hilo no exceda el tamaño de sus datos. Es muy probable que esto ocurra y siempre se debe revisar.

Vectores en OpenCL

Cada tipo fundamental de opencl tiene una versión vectorial. Puede usar el tipo de vector agregando el número de componentes deseados después del tipo. El número de componentes admitidos es 2,3,4,8 y 16. OpenCL 1.0 no ofrece tres componentes.

Puedes inicializar cualquier vector de dos maneras:

- Proporcionar un solo escalar
- Satisfacer todos los componentes

```
float4 a = (float4)(1); //a = (1, 1, 1, 1)
```

o

```
float4 b = (float4)(1, 2, 3, 4);
float4 c = (float4)(1, (float3)(2));
```

o cualquier otra combinación de vectores que satisfagan el número de componentes. Para acceder a los elementos de un vector puedes utilizar diferentes métodos. Puedes usar indexación:

```
a[0] = 2;
```

o utilizar literales. La ventaja de los literales es que puedes combinarlos de la forma que quieras, así que hazlo en un momento. Puedes acceder a todos los componentes vectoriales con

```
a.s0 = 2; // same as a[0] = 2
```

También puedes combinar múltiples componentes en un nuevo vector.

```
a.s02 = (float2)(0, 0); // same as  a[0] = 0; a[2] = 0; or even a.s20 = (float2)(0, 0)
```

Puede cambiar el orden de los componentes de la forma que desee.

```
a.s1423 = a.s4132; // flip the vector
```

pero no puedes hacer algo como

```
a.s11 = ... // twice the same component is not possible
```

Hay algunos métodos abreviados convenientes para acceder a componentes vectoriales. Las siguientes abreviaturas solo se aplican a los tamaños 2, 4, 8 y 16

```
a.hi //=a.s23 for vectors of size 4, a.4567 for size 8 and so on.
a.lo //=a.s01
a.even //=a.s02
a.odd //=a.13
```

Para tamaños de vectores 2,3 y 4 hay algunas abreviaturas adicionales.

```
a.x //=a.s0
a.y //=a.s1
a.z //=a.s2
a.w //=a.s3
```

Kernel de corrección gamma

Veamos un núcleo de corrección gamma

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                               CLK_ADDRESS_CLAMP_TO_EDGE |
                               CLK_FILTER_LINEAR;

__kernel void Gamma(__read_only image2d_t input, __write_only image2d_t output, __constant
float gamma) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int2 size = get_image_dim(input);

    if(all(gid < size)){
        uint4 pixel = read_imageui(input, sampler, gid);
        float4 color = convert_float4(pixel) / 255;
        color = pow(color, (float4)(gamma));
        pixel = convert_uint4_rte(color * 255);
        write_imageui(output, gid, pixel);
    }
}
```

Ahora veamos ese código paso a paso. La primera línea crea una variable en la __ región de memoria constante de tipo sampler_t. Esta muestra se utiliza para especificar aún más el acceso a nuestros datos de imagen. Consulte la documentación de Khronos para obtener una documentación completa.

Asignamos la entrada como read_only y la salida como write_only antes de llamar a nuestro kernel, así que agregamos esos modificadores aquí.

image2d y image3d siempre se asignan en la memoria global, por lo tanto, podemos omitir el modificador __global aquí. Nuestro valor gamma se encuentra en la memoria __ constante, por lo que también lo especificamos.

Luego obtenemos nuestro ID de hilo que define el píxel que vamos a gamma correcto. También consultamos el tamaño para asegurarnos de que nuestro hilo no está accediendo a la memoria no asignada. Esto definitivamente bloqueará tu kernel si olvidas eso.

Después de asegurarnos de que somos un hilo legítimo, leemos nuestro píxel fuera de nuestra imagen de entrada. Luego lo convertimos en flotante para evitar la pérdida de lugares decimales, hacemos algunos cálculos, lo convertimos y lo escribimos en la salida.

Lea Fundamentos del Kernel en línea: https://riptutorial.com/es/opencl/topic/10174/fundamentos-del-kernel

Capítulo 6: Interacción de la memoria del host

Introducción

Este tema destaca diferentes formas de colocar datos en algún lugar donde su dispositivo pueda acceder a ellos.

Examples

Leyendo una matriz

Para leer una matriz del dispositivo al host, se llama

```
clEnqueueReadBuffer($queue, $memobj, $blocking, $offset, $size, $target, 0, null, null);
```

La cola $ es el CommandQueue que se usó para asignar la memoria en el dispositivo. El $ memobj contiene la dirección a la memoria del dispositivo, $ desplazamiento y $ tamaño definen con más precisión de dónde y cuántos datos se copian. El $ target es un puntero a la memoria del host donde se almacenarán los datos. El $ target debe asignarse y tener un tamaño adecuado.

Leyendo una textura

Leer una imagen es casi como leer una matriz. La única diferencia es que el tamaño y la compensación deben ser tridimensionales.

```
clEnqueueReadImage($queue, $memobj, $blocking, $offset, $size, $stride, $slice_pitch, $target,
0, null, null);
```

El $ stride define cuántos bytes tiene una fila. Normalmente esto es solo ancho * (bytes por píxel), pero alguien podría querer cambiarlo para alinear los datos con los bancos de memoria. Lo mismo ocurre con $ slice_pitch, solo que este valor es para la tercera dimensión.

Escribiendo una textura 2D

Para copiar una textura al dispositivo hay dos pasos necesarios.

1. Asignar la memoria en el dispositivo.
2. Copia la imagen al dispositivo.

```
  _mem = clCreateImage2D($context, $mem_flags, $image_format, $width, $height, $stride,
$source, &err);
```

Las $ mem_flags definen cómo se asigna la memoria. Puede ser de solo lectura, solo escritura o

ambos. Además, puede definir dónde y cómo se asigna la memoria. $ width, $ height y $ stride son bastante explicativos.

Si su mem_flags copia los datos, ya está. Si desea hacerlo manualmente en un momento posterior, deberá llamar a otra función cuando esté listo.

```
err = clEnqueueWriteImage($queue, _mem, $blocking, $offset, $size, $stride, $slice_pitch,
$source, 0, null, null);
```

El valor $ offset y $ size definen la región de la imagen que desea copiar en la memoria de destino. El $ stride define cuántos bytes tiene una fila. Normalmente esto es solo ancho * (bytes por píxel), pero alguien podría querer cambiarlo para alinear los datos con los bancos de memoria. Lo mismo ocurre con $ slice_pitch, solo que este valor es para la tercera dimensión. Tanto $ stride como $ slice_pitch tienen que coincidir con sus datos de entrada.

Banderas de memoria

Al asignar Memoria, tiene la opción de elegir entre diferentes modos:

- Memoria de sólo lectura
- Escribe solo memoria
- Memoria de lectura / escritura

La memoria de solo lectura se asigna en la región de memoria constante, mientras que las otras dos se asignan en la región global global.

Además de la accesibilidad, puede definir dónde se asigna su memoria.

- No especificado: su memoria está asignada en la memoria del dispositivo como cabría esperar. El puntero $ source se puede establecer en nulo.
- CL_MEM_USE_HOST_PTR: Esto le dice al dispositivo que los datos están en la RAM del sistema y no deben ser movidos. En su lugar, los datos se manipulan directamente en el ram.
- CL_MEM_COPY_HOST_PTR: Le dice al dispositivo que copie todos los valores de la dirección dada a la memoria del dispositivo o, usando CL_MEM_ALLOC_HOST_PTR, a una región de memoria separada en la memoria ram del sistema.
- CL_MEM_ALLOC_HOST_PTR: Le dice al dispositivo que asigne espacio en el ram del sistema. Si se utiliza como único parámetro, el puntero $ source se puede establecer en nulo.

En cuanto a la velocidad, el acceso a la memoria global del dispositivo es el más rápido. Pero también necesitas llamarlo dos veces para copiar datos. Usar el puntero del Host es el más lento, mientras que Alloc_host_ptr ofrece una velocidad más alta.

Cuando se usa use_host_ptr, el dispositivo hace exactamente eso: usa sus datos en el ram del sistema, que por supuesto es paginado por el sistema operativo. Por lo tanto, cada llamada de memoria tiene que pasar por la CPU para manejar posibles resultados de página. Cuando los datos están disponibles, la CPU los copia en la memoria anclada y los pasa al controlador DMA

utilizando preciados ciclos de reloj de CPU. Por el contrario, alloc_host_ptr asigna memoria anclada en el ram del sistema. Esta memoria se coloca fuera del mecanismo de intercambio de páginas y, por lo tanto, tiene una disponibilidad garantizada. Por lo tanto, el dispositivo puede omitir la CPU por completo al acceder al ram del sistema y utilizar DMA para copiar rápidamente los datos al dispositivo.

Escribiendo una matriz

Escribir una matriz consiste en dos pasos:

1. Asignación de la memoria.
2. Copiando los datos

Para asignar la memoria, una simple llamada a

```
_mem = clCreateBuffer($queue, $mem_flags, $size, $host_ptr, &err);
```

es suficiente. Si decidió copiar el puntero del host a través de mem_flags, habrá terminado. De lo contrario, puede copiar los datos cuando lo desee con

```
err = clEnqueueWriteBuffer($queue, _mem, $blocking, $offset, $size, $source, 0, null, null);
```

Lea Interacción de la memoria del host en línea:
https://riptutorial.com/es/opencl/topic/10173/interaccion-de-la-memoria-del-host

Capítulo 7: Operaciones atómicas

Sintaxis

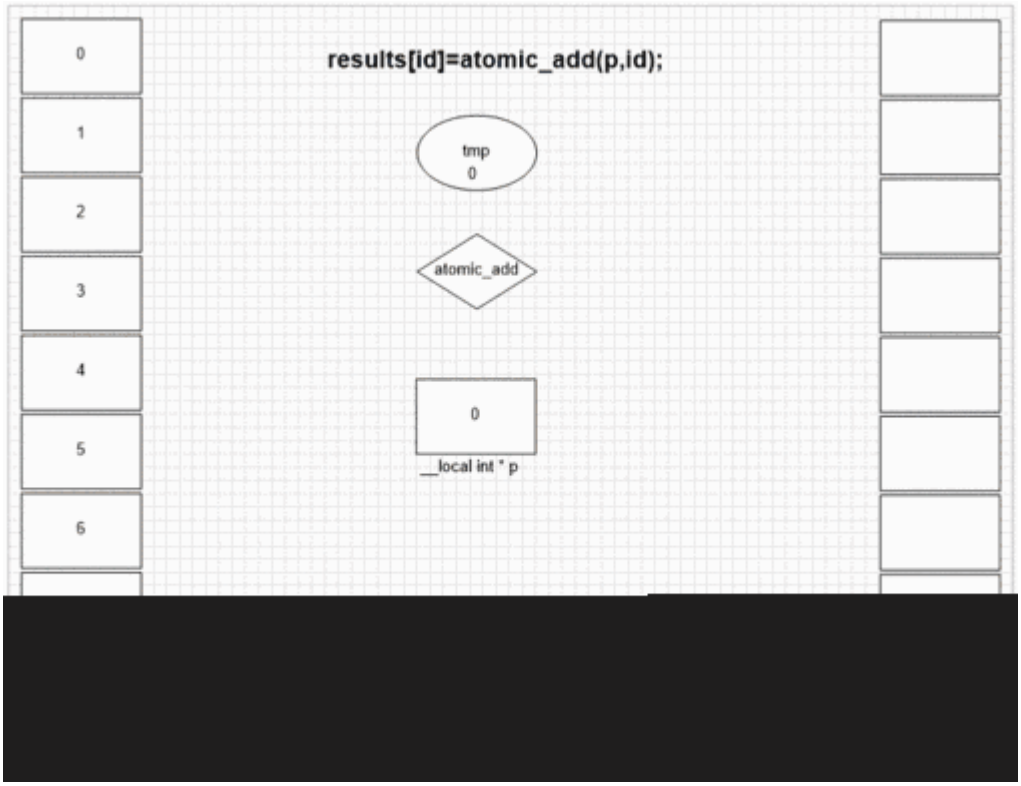
- int atomic_add (volatile __global int * p, int val)
- unsigned int atomic_add (__global volatile unsigned int * p, int int sin signo)
- int atomic_add (volatile __local int * p, int val)
- unsigned int atomic_add (volatile __local unsigned int * p, unsigned int val)

Parámetros

pag	val
puntero a celda	añadido a la celda

Observaciones

El rendimiento depende del número de operaciones atómicas y del espacio de memoria. El trabajo en serie casi siempre ralentiza la ejecución del kernel debido a que gpu es una matriz SIMD y cada unidad en una matriz espera otras unidades si no realizan el mismo tipo de trabajo.



Examples

Función de adición atómica

```
int fakeMalloc(__local int * addrCounter,int size)
{
    // lock addrCounter
    // adds size to addrCounter's pointed cell value
    // unlock
    // return old value of addrCounter's pointed cell

    // serial between all threads visiting -> slow
    return atomic_add(addrCounter,size);
}

__kernel void vecAdd(__global float* results )
{
   int id = get_global_id(0);
   int lid=get_local_id(0);
   __local float stack[1024];
   __local int ctr;
   if(lid==0)
      ctr=0;
   barrier(CLK_LOCAL_MEM_FENCE);
   stack[lid]=0.0f;                    // parallel operation
   barrier(CLK_LOCAL_MEM_FENCE);
   int fakePointer=fakeMalloc(&ctr,1); // serial operation
   barrier(CLK_LOCAL_MEM_FENCE);
   stack[fakePointer]=lid;             // parallel operation
   barrier(CLK_GLOBAL_MEM_FENCE);
   results[id]=stack[lid];
}
```

Salida de primeros elementos:

algunas veces

192 193 194 195 196 197 198

algunas veces

0 1 2 3 4 5 6

algunas veces

128 129 130 131 132 133 134

para una configuración con rango local = 256.

Cualquiera que sea el hilo que visite fakeMalloc primero, coloca su propia ID de hilo local en la primera celda de resultados. El SIMD interno y la estructura de frente de onda del ejemplo gpu permiten que los 64 hilos vecinos coloquen sus valores en las celdas de resultados vecinas. Otros dispositivos pueden poner los valores de forma más aleatoria o totalmente en orden, dependiendo de la implementación abierta de dichos dispositivos.

Lea Operaciones atómicas en línea: https://riptutorial.com/es/opencl/topic/6758/operaciones-atomicas

Creditos

S. No	Capítulos	Contributors
1	Empezando con opencl	benshope, Community, CShark, Dithermaster, Dschoni, huseyin tugrul buyukisik, Umar Arshad
2	Configuración básica de OpenCL	CShark
3	Ejemplo de kernel generador de números pseudoaleatorios	huseyin tugrul buyukisik, silverclaw
4	Fundamentos de hardware de OpenCL	CShark
5	Fundamentos del Kernel	CShark
6	Interacción de la memoria del host	CShark
7	Operaciones atómicas	huseyin tugrul buyukisik